Efficient Algorithms

Winter Semester 1999/2000

Prof. Dr. Sami Khuri[1]

Draft, Version of May 3, 2000

[1]These are the lecture notes of a course taught at Technische Universität München. The notes were typed by Christian Osendorfer.

Contents

Chapter 1

Introduction

Algorithm:

- The origin of the word **algorithm** is "al-kawarizmi", a ninth-century Persian mathematician:

  أبو جعفر محمد إبن موسى الخوارزمي

- A set of rules for solving a problem in a finite number of steps. (*Machine independent and language independent*).

Computer program:

- Implementation of an algorithm
- Series of well stated instructions to be performed by the computer
- These instructions are stated in a programming language. (*Machine independent*)

1.1 Designing an algorithm

Only a handful of techniques are used in the thousands of algorithms that are written to solve various problems. They describe the general manner in which the algorithm operates.

Technique: General method in which an algorithm operates.

Most common techniques:

- Divide-and-conquer
- Greedy technique
- Dynamic programming
- Backtracking (branch and bound)

1.2 Analyzing an algorithm

Analysis: Make use of mathematics to be able to predict the **time** and **space** the execution of its implemented program would require.

1.2.1 Measuring criteria

Consider the following measures to compare the work done by different algorithms:

- **Execution time (CPU time)**
  Not a good measure since machine dependent (example: different for different architecture and technologies).

- **Number of (executable) statements**
  It depends on the programming language being used and on the programmer's style and is therefore not a good criterion.

- **Number of repetitions in loop**
  It is not a good measure since the amount of work can vary from one pass to another. It is also language dependent.

- **Count basic operations**
  Choose a basic operation and count the number of times it is performed. This is a good measuring criterion.

  EXAMPLES:

Problem	**Basic operation**
Find key in list	Comparison (of key with entry in list)
Multiply two matrices	Multiplication (of two numbers)
Sorting	Comparison (of two entries in list)
Traverse (binary tree)	Traversing one link

1.2.2 Why do we need to analyze an algorithm ?

- Analysis is generally more reliable than experimentation. An experiment is input-data dependent, analysis can give guarantees about all inputs.

- Analysis helps in choosing among several algorithms that solve the given problem. [We don't need to implement and test all the different algorithms].

- Analysis allows us to predict the performance of a program before writing the code.

- Analysis allows to get a better understanding of the solution; to anticipate the "slow" and "fast" parts, and to try and improve the "slow" ones.

1.2.3 Θ, O and Ω Notations

Given a particular input (of size n, for example), the **running time** of an algorithm $T(n)$ is the number of basic operations (or steps) it performs. $T(n)$ is a function of the input size n. Its leading term is the **growth rate** of $T(n)$. We are interested in the **order of growth** (rate of growth) of $T(n)$.

Assume that $f(n)$ and $g(n)$ are running time functions.

1. **Θ-Notation**
   $f(n)$ is $\Theta(g(n))$

$$\text{iff } \exists\, c_1 \in \mathbf{R}^+, c_2 \in \mathbf{R}^+ \text{and } \exists\, n_0 \in \mathbf{Z}^+$$

$$\text{such that: } c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\, n > n_0.$$

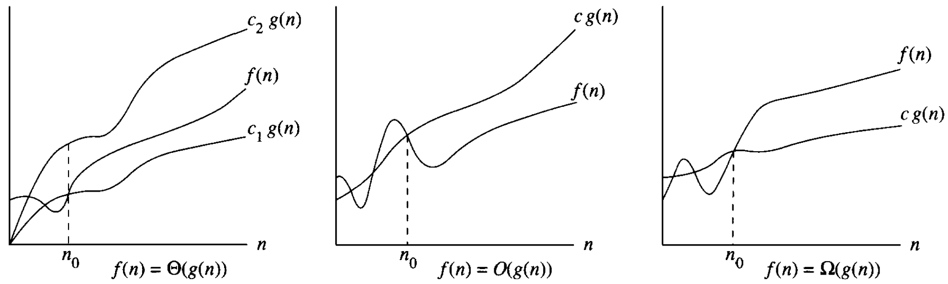


Figure 1.1: The Θ, O and Ω Notations [CLR90, p. 25]

Note:

(a) $f(n)$ is sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for $n > n_0$.

(b) $f(n)$ is asymptotically bounded from above and below by $g(n)$.

2. **O-Notation**
$f(n)$ is $O(g(n))$

$$\text{iff } \exists\, c \in \mathbf{R}^+ \text{and } \exists\, n_0 \in \mathbf{Z}^+$$
$$\text{such that: } f(n) \leq cg(n)\ \forall\, n > n_0.$$

Note:

(a) $f(n)$ is of order $g(n)$.

(b) $f(n)$ is asymptotically upper bounded by $g(n)$.

3. **Ω-Notation**
$f(n)$ is $\Omega(g(n))$

$$\text{iff } \exists\, c \in \mathbf{R}^+ \text{and } \exists\, n_0 \in \mathbf{Z}^+$$
$$\text{such that: } cg(n) \leq f(n)\ \forall\, n > n_0.$$

Note:

(a) $f(n)$ is asymptotically lower bounded by $g(n)$.

Results:

- $f(n)$ is $O(g(n)) \Leftrightarrow g(n)$ is $\Omega(f(n))$
- $f(n)$ is $\Theta(g(n)) \Leftrightarrow \begin{cases} f(n) \text{ is } O(g(n)) \\ \quad\text{and} \\ f(n) \text{ is } \Omega(g(n)) \end{cases}$

1.3 Examples

A. Given a polynomial $p(n) = a_0 + a_1 n + \cdots + a_k n^k$, where $a_k > 0$.
Show that $p(n)$ is $\Theta(n^k)$.
Proof: We want to show $\exists\, c_1, c_2 \in \mathbf{R}^+$ and $n_0 \in \mathbf{Z}^+$ such that:

$$c_1 n^k \leq a_0 + a_1 n + \cdots + a_k n^k \leq c_2 n^k.$$

1. Show $a_0 + a_1 n + \cdots + a_k n^k \le c_2 n^k$.
   Let $c_2 = |a_0| + |a_1| + \cdots + |a_k|$ and $n_0 = 1$.
   For all $n > n_0$ we have:

$$
\begin{aligned}
p(n) &= a_0 + a_1 n + \cdots + a_k n^k \\
&\le |a_0| + |a_1| n + \cdots + |a_k| n^k \\
&\le |a_0| n^k + |a_1| n^k + \cdots + |a_k| n^k \\
&= n^k (|a_0| + |a_1| + \cdots + |a_k|) = c_2 n^k
\end{aligned}
$$

   which is what we wanted to prove.

2. Show $c_1 n^k \le a_0 + a_1 n + \cdots + a_k n^k$.
   Let $c_1 = \frac{a_k}{2}$.
   We want to show that:

$$\frac{a_k}{2} n^k \le a_0 + a_1 n + \cdots + a_k n^k$$

$$\text{i.e. } a_0 + a_1 n + \cdots + \frac{a_k}{2} n^k \ge 0$$

$$\text{or } \underbrace{n^k}_{>0} \left(\frac{a_k}{2} + \frac{a_{k-1}}{n} + \cdots + \frac{a_0}{n^k} \right) > 0$$

$$\text{If } \frac{a_k}{2} \ge \frac{|a_{k-1}|}{n} + \cdots + \frac{|a_0|}{n^k} \ge 0$$

$$\text{then } \left(\frac{a_k}{2} + \frac{a_{k-1}}{k} + \cdots + \frac{a_0}{n^k} \right) > 0 \text{ always}$$

$$\text{even if } a_0 < 0, a_1 < 0 \ldots a_{k-1} < 0.$$

   Let $M = \max\{|a_0|, |a_1|, \ldots, |a_{k-1}|\}$.
   Let $n_0 = \max\left\{1, \frac{2Mk}{a_k}\right\}$.
   For any $n > n_0$ and all $i, 0 \le i \le k-1$ and $a_i \ne 0$ we have:

$$\frac{|a_i|}{n^{k-i}} \le \frac{|a_i|}{n} \text{ since } 0 \le i \le k-1 \text{ (i.e. } 1 \le k-i)$$

$$\text{and } \frac{|a_i|}{n} \le \frac{M}{n} \text{ since } M = \max\{|a_0|, |a_1|, \ldots, |a_{k-1}|\}.$$

$$\text{And } \frac{M}{n} \le \frac{a_k}{2k} \text{ since } \frac{2Mk}{a_k} \le n_0 < n.$$

$$\text{So } \frac{|a_i|}{n^{k-i}} \le \frac{a_k}{2k} \text{ for } n > n_0.$$

$$\text{And } \left(\frac{a_k}{2k} \right) k \ge \frac{|a_{k-1}|}{n} + \frac{|a_{k-2}|}{n^2} + \cdots + \frac{|a_0|}{n^k}.$$

$$\Rightarrow \frac{a_k}{2} \ge \frac{|a_{k-1}|}{n} + \frac{|a_{k-2}|}{n^2} + \cdots + \frac{|a_0|}{n^k}$$

   which is what we wanted to prove. $\square$

B.i) Show that $\log_2 n$ *is* $O(n)$.

Proof: We want to show that $\exists\, c, \exists\, n_0$ so that $\log_2 n \leq cn\ \forall\, n > n_0$.

$2^n = (1+1)^n = \sum_{i=0}^{n} \binom{n}{i} = 1 + n + \frac{n(n-1)}{2} + \ldots + \frac{n(n-1)}{2} + n + 1$

$\Rightarrow 2^n \geq 1 + n + n^2 - n + n + 1\ (\forall\, n > 5)\ = n^2 + n + 2 > n$

So we have $2^n \geq n\ (\forall\, n > 5)$

$\Rightarrow \log_2 2^n \geq \log_2 n\ \Rightarrow n \log_2 2 \geq \log_2 n\ \Rightarrow \log_2 n \leq n$.

So for $c = 1$ and $n_0 = 4$ we finally have $\log_2 n \leq n\ \forall\, n > n_0$. $\square$

B.ii) Show that $\log_2 n$ *is NOT* $\Omega(n)$.

Proof: Show that $\forall\, c \in \mathbf{R}^+$ and $\forall\, n_0 \in \mathbf{Z}^+$, $\exists\, n \in \mathbf{Z}^+, n > n_0$ such that:

$$\log_2 n \leq cn \text{ or } n \leq 2^{cn}$$

We have: $2^{cn} = (1 + (2^c - 1))^n = \sum_{i=0}^{n} \binom{n}{i} (2^c - 1)^i > \binom{n}{2}(2c-1)^2\ \forall\, n \geq 2$.

We want to show $2^{cn} > n$, which is achieved by showing that $\binom{n}{2}(2^c - 1)^2 > n$

$\Leftrightarrow \frac{n(n-1)}{2}(2^c-1)^2 > n\ \Leftrightarrow \frac{(n-1)(2^c-1)^2}{2} > 1$

$\Leftrightarrow n - 1 > \frac{2}{(2^c-1)^2} \Leftrightarrow n > 1 + \frac{2}{(2^c-1)^2}$

So for any c and n_0 we define $n_1 = \max\left\{n_0, 1 + \frac{2}{(2c-1)^2}\right\}$

If $n > n_1$ then $2^{cn} > n \Rightarrow \log_2 n < cn$. So $\log_2 n$ is not $\Omega(n)$. $\square$

C. Show that the growth rate of $\sum_{i=1}^{n} \frac{1}{i}$ *is* $\Theta(\log_2 n)$.

Proof: Bounds for $\sum_{i=1}^{2^k} \frac{1}{i}$:

$$1 + \frac{1}{2} + \overbrace{\frac{1}{4} + \frac{1}{4}} + \overbrace{\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}} + \ldots + \frac{1}{2^k}$$

$$< 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \ldots + \frac{1}{2^k}$$

$$< 1 + \underbrace{\frac{1}{2} + \frac{1}{2}} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}} + \frac{1}{8} + \ldots + \frac{1}{2^k}$$

$$\Rightarrow 1 + \sum_{i=1}^{k} (2^{i-1} \frac{1}{2^i} < \sum_{i=1}^{n} \frac{1}{i} < \sum_{i=1}^{k-1} 2^i \frac{1}{2^i} + \frac{1}{2^k}$$

$$\Rightarrow 1 + \frac{1}{2} \sum_{i=1}^{k} 1 < \sum_{i=1}^{n} \frac{1}{i} < \sum_{i=1}^{k} 1 + \frac{1}{2^k}$$

$$\Rightarrow 1 + \frac{k}{2} < \sum_{i=1}^{2^k} \frac{1}{i} < k + \frac{1}{2^k}$$

For any n: $2^k \leq n \leq 2^{k+1}$ i.e. $k = \lfloor \log_2 n \rfloor$ where $\lfloor x \rfloor$ denotes the greatest integer not exceeding x.

$$\Rightarrow 1 + \frac{k}{2} < \sum_{i=1}^{2^k} \frac{1}{i} < \sum_{i=1}^{n} \frac{1}{i} < \sum_{i=1}^{2^{k+1}} \frac{1}{i} < k + 1 + \frac{1}{2^{k+1}}$$

$$k = \lfloor \log_2 n \rfloor \Rightarrow \log_2 n - 1 < k \leq \log_2 n.$$

$$\text{So we have: } 1 + \frac{k}{2} < \sum_{i=1}^{n} \frac{1}{i} < \sum_{i=1}^{2^{k+1}} \frac{1}{i} < k + 1 + \frac{1}{2^{k+1}}$$

$$1 + \frac{\log_2 n - 1}{2} < 1 + \frac{k}{2} < \sum_{i=1}^{n} \frac{1}{i} < \log_2 n + 1 + \frac{1}{2^{\log_2 n}}$$

$$\text{Since } k + 1 + \frac{1}{2^{k+1}} < \log_2 n + 1 + \frac{1}{2^{\log_2 n}}$$

$$\Rightarrow \frac{1}{2} + \frac{\log_2 n}{2} < \sum_{i=1}^{n} \frac{1}{i} < \log_2 n + 1 + \frac{1}{n} < \log_2 n + 2$$

$$\frac{\log_2 n}{2} < \sum_{i=1}^{n} \frac{1}{i} < 2\log_2 n \text{ since } \log_2 n > 2 \text{ for } n \geq 5.$$

For $c_1 = \frac{1}{2}, c_2 = 2$ and $n_0 = 4$ we have $\frac{\log_2 n}{2} < \sum_{i=1}^{n} \frac{1}{i} < 2\log_2 n \; \forall n > 4$.
So we have proved that $\sum_{i=1}^{n} \frac{1}{i}$ is $\Theta(\log_2 n)$. □

1.4 Some common running times

1	$O(1)$ means that the number of executions of the basic operation is fixed.
$\log n$	logarithmic running time
n	linear running time
$n \log n$	arises when the problem is broken into independent subproblems. They are then solved and combined
n^2	quadratic [e.g. some sorting algorithms]
n^3	cubic running time
2^n	exponential; not practical

As n gets large, the difference between exponential and polynomial growth rates becomes **tremendous**.

In the next four chapters we study the most common algorithmic techniques mentioned in Section 1.1, namely, divide and conquer, greedy, dynamic programming and backtracking algorithms.

	Size n					
Growth Rates	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 second	24.3 second	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
5^n	.059 second	58 minutes	6.5 years	3855 centuries	2 x 10^8 centuries	1.3 x 10^{13} centuries

Table 1.1: Comparison of several polynomial and exponential time complexity functions [GJ79, p. 7]

Chapter 2

Divide and Conquer

The divide-and-conquer paradigm is based on the following idea: Divide problem into smaller parts of the same type. Find solutions for parts. Combine solutions to give solution to original problem.
Examples: Binary search and Quicksort
Implementations of divide and conquer algorithms are often recursive. The running time functions of divide and conquer algorithms can be written in the following way:

$$T(n) = aT(\frac{n}{b}) + d(n)$$

$$T(1) = 1 \text{ , where}$$

- $T(n)$: amount of work to solve original problem
- a: the number of subproblems we have to solve
- $\frac{n}{b}$: number of parts the problem is divided into
- $d(n)$ is the amount of work required to combine a subproblems of size $\frac{n}{b}$

2.1 Quicksort (partition exchange sort)

Quicksort was invented by *C.A.R. Hoare*. Previous fast sorts had used extra storage.

- **Problem:**
  Given an unsorted array L of size n, we want to sort L in ascending order.
- **Idea:**
  It would be nice, if we could find the median of an array and split the array in two equal parts.
- **Technique:**
  At each step an element of L is placed in its final position in the array. This means that all entries before are smaller, and all entries after it larger. The partition element can be chosen:

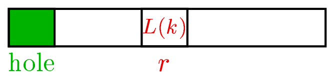
  i) first element in L
  ii) any random element in L
  iii) median of any three elements in L
  iv) middle element of L

In what follows, we choose the partition element according to *iv*). We make use of Split, an algorithm that chooses an element X from the subarray that is currently being sorted and rearranges the entries, finding an integer i such that $L(r) \leq X$ for $k \leq r < i$, $L(i) = X$, and $L(r) \geq X$ for $i < r \leq m$. Thus, X (which is in the i^{th} position) is now in its final correct position and can be ignored.

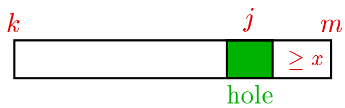
Remark: Holes are really copying operations. Just assume that when we copy we leave a hole. Quicksort is a recursive procedure.

```
QUICKSORT(L, k, m)                          Sort L(k)...L(m)
  begin
    if k < m then do
    begin
      SPLIT(L, k, m, i);                    Divide
      QUICKSORT(L, k, i-1);
                                            Conquer
      QUICKSORT(L, i+1, m)
    end
  end
```

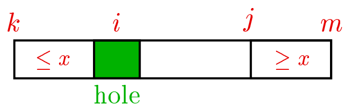
```
SPLIT(L, k, m, i)
  begin
    r:=⌊(k+m)/2⌋;
    X =L(r);
    L(r)=L(k);                              {start with hole at k}
    i:=k;
    j:=m;
```




```
    repeat
    begin
      while (i<j) and (L(j)≥ X) do
        j:=j-1
      if i=j then exit                      {from repeat}
      else
        begin
        L(i):=L(j);                         {hole at j}
        i:=i+1
        end
```

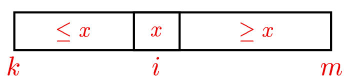



```
      while (i<j) and (L(i)≤ X) do
        i:=i+1
      if i=j then exit                      {from repeat}
      else
        begin
        L(j):=L(i);                         {hole at i}
        j:=j-1
        end
```




```
    end                                     {of repeat}
    L(i):=X
  end
```

Array after the execution of Split:



Remark: SPLIT can be written without the 2 exits. But SPLIT as written here is clearer for explaining the steps and analysis.

2.1.1 Worst-case analysis

Occurs when the splitting element X is the largest (or smallest) element in the current subfile $L(k)$, $L(k+1)$, ..., $L(m-1)$, $L(m)$. Thus, SPLIT produces an empty right (or left) subfile. This means that only one **while** loop will be executed each time through the repeat loop.
Passes in repeat loop:

1^{st} pass	n elements	$n-1$ comparisons
2^{st} pass	$n-1$ elements	$n-2$ comparisons
$\vdots$	$\vdots$	$\vdots$
$n-1$ pass	2 elements	1 comparison

$\Rightarrow\ W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

$\Rightarrow$ Quicksort is $O(n^2)$ in the worst case.

2.1.2 Average case analysis

It can be shown that $AC(n)$ is $O(n \log_2 n)$, where $AC(n)$ denotes the average number of comparisons required for a file of size n. We assume that the element chosen to split is equally likely to end up in each of $L(k)$, $L(k+1)$, ..., $L(m)$.

$$AC(n) = n + 1 + \frac{1}{n} \sum_{i=1}^{n} [AC(i-1) + AC(n-i)]$$

where

$n+1$	1^{st} pass requires at most $n+1$ comparisons (even if implemented in a different way) regardless of where the partition element is
$\frac{1}{n}$	i has equal probability of being $1, 2, \ldots, n$.
$AC(i-1)$	left subarray
$AC(n-i)$	right subarray
$AC(1) = 0$	
$AC(0) = 0$	

$$\begin{aligned} \Rightarrow nAC(n) &= (n+1)n + \tfrac{n}{n}\left[\sum_{i=1}^{n} AC(i-1) + \sum_{i=1}^{n} AC(n-i)\right] \\ &= (n+1)n + \left[\sum_{i=1}^{n} AC(i-1) + \sum_{i=1}^{n} AC(i-1)\right] \end{aligned}$$

$$\Rightarrow [AC(n)]n = (n+1)n + 2\sum_{i=1}^{n} AC(i-1)$$

$$\textbf{Note:}\ \sum_{i=1}^{n} AC(n-i) = AC(n-1) + AC(n-2) + \ldots + AC(0)$$

$$nAC(n) = n(n+1) + 2[AC(0) + AC(1) + \ldots + AC(n-1)] \qquad (1)$$

and

$$(n-1)AC(n-1) = (n-1)n + 2[AC(0) + AC(1) + \ldots + AC(n-2)] \qquad (2)$$

Subtract to obtain (perform (1) − (2)):

$$nAC(n) - (n-1)AC(n-1) = n(n+1) - n(n-1) + 2[AC(n-1)]$$

$$\Rightarrow nAC(n) - (n-1)AC(n-1) = n[n+1-n+1] + 2AC(n-1)$$

$$\Rightarrow nAC(n) = (n-1+2)AC(n-1) + 2n$$

$$nAC(n) = (n+1)AC(n-1) + 2n$$

Multiply both sides by $\frac{1}{n(n+1)}$.

$$\Rightarrow \frac{AC(n)}{n+1} = \frac{AC(n-1)}{n} + \frac{2}{n+1}$$

Thus

$$\begin{array}{ccccc} \frac{AC(n-1)}{n} & = & \frac{AC(n-2)}{n-1} & + & \frac{2}{n} \\ \frac{AC(n-2)}{n-1} & = & \frac{AC(n-3)}{n-2} & + & \frac{2}{n-1} \\ \vdots & \vdots & \vdots & & \\ \frac{AC(2)}{3} & = & \frac{AC(1)}{2} & + & \frac{2}{3} \end{array}$$

Add to get

$$\begin{aligned} \frac{AC(n)}{n+1} &= \frac{AC(1)}{2} + 2\left[\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right] \\ &= 2\sum_{i=3}^{n+1}\frac{1}{i} \\ &= 2\left[\sum_{i=1}^{n+1}\frac{1}{i} - \left(1 + \frac{1}{2}\right)\right] \end{aligned}$$

$$\Rightarrow AC(n) = 2(n+1)\left(\sum_{i=1}^{n+1}\frac{1}{i} - \frac{3}{2}\right)$$

$$\Rightarrow AC(n) \text{ is } \Theta(n\log_2 n), \text{ since}$$

$$\sum_{i=1}^{n}\frac{1}{i} \text{ is } \Theta(\log_2 n) \text{ and } \Theta(n) < \Theta(n\log_2 n)$$

□

2.2 General divide&conquer

The general Divide&Conquer procedure, DIV&CONQ makes use of OPERATE, which is a procedure that runs efficiently on small problem instances.

```
DIV&CONQ(X)
    if X is sufficiently small or simple then return OPERATE(X)
    else
        decompose X in smaller instances X_1, X_2, ..., X_2
        for i ← 1 to n
            Y_i ← DIV&CONQ(X_i)
        recombine the Y_i's to obtain a solution Y for X
        return Y
```

2.2.1 Other examples

- **Binary Search**:
  Search a sorted array for a particular "key".
  Idea: Check the middle element of the array; if it equals "key" then we are done. If it is less than "key" then perform a binary search of upper half, otherwise perform a binary search of lower half. **Note**: Only one of the two subproblems is further considered.

  $$\text{In general: } T(n) = T\left(\frac{n}{2}\right) + c \quad \text{(n is a power of 2)}$$

  or more precisely:

  $$T(n) = \underbrace{T(\left\lfloor \frac{n}{2} \right\rfloor)}_{\substack{\text{Time for}\\ \text{solving}\\ \text{subproblem}\\ \text{of size } \frac{n}{2}}} + \underbrace{1}_{\substack{\text{Amount of}\\ \text{work needed}\\ \text{to split problem}}}$$

- Max-Min **Problem**
  Find the maximum and the minimum values in an unsorted array.
  Idea: Break the array in half. Find the maximum and minimum of each half separately (**Note:** both subproblems are solved here). Then, the minimum is the smaller of the two minima and the maximum is the larger of the two maxima.

  $$\text{Here: } T(n) = 2T(\frac{n}{2}) \ + \ c$$

  when n is a power of 2.

- Mergesort **Problem**
  Sort an array in ascending/descending order.
  Idea: Split the array in half and recursively sort each half (using Mergesort). Then merge the two halves with one pass through each half.

  $$\text{Here : } T(n) = 2T(\frac{n}{2}) \ + \ cn \text{ when } n = 2^j$$

In the next section, we study a general recurrence relation encountered in the analysis of many algorithms.

2.2.2 The general recurrence relation

Consider:

$$\begin{array}{ll} T(n) = aT(\frac{n}{b}) \ + \ d(n) & n > 1 \\ T(n) = 1 & n = 1 \end{array}$$

We hope that $d(n)$ is a nice function, otherwise we generally approximate it.

Particular Case:
Suppose that $n = b^j$ and $d(n) = cn^i = \Theta(n^i)$. We show that the closed form of

$$\begin{array}{ll} T(n) = aT(\frac{n}{b}) \ + \ d(n) & n > 1 \\ T(n) = 1 & n = 1 \end{array}$$

$$\text{is : } T(b^j) = a^j \ + \ cb^{ji} \sum_{k=0}^{j-1} \left(\frac{a}{b^i}\right)^k \qquad (*)$$

Proof:
Use induction on j
Basis: $j = 0$

$T(b^0) = a^0 + cb^0(0) = a^0 = 1$
which is correct, since we
are given T(1)=1

Induction hypothesis:
Suppose that $(*)$ is true for $j = m - 1$

$$\text{i.e. } T(b^{m-1}) = a^{m-1} + cb^{(m-1)i} \sum_{k=0}^{m-2} \left(\frac{a}{b^i}\right)^k$$

Show that $(*)$ is true for $j = m$.
We know that:

$$T(b^m) = aT(\frac{b^m}{m}) + c(b^m)^i \text{ from the recurrence relation}$$

$$\begin{aligned}
\Rightarrow T(b^m) &= aT(b^{m-1}) \ + \ cb^{mi} \\
&= a\left[a^{m-1} + cb^{(m-1)i} \sum_{k=0}^{m-2} \left(\tfrac{a}{b^i}\right)^k\right] \ + \ cb^{mi} \quad \text{(by induction hypothesis)} \\
&= a^m + acb^{(m-1)i} \sum_{k=0}^{m-2} \left(\tfrac{a}{b^i}\right)^k + cb^{mi} \\
&= a^m + \tfrac{acb^{mi}}{b^i} \sum_{k=0}^{m-2} \left(\tfrac{a}{b^i}\right)^k + cb^{mi} \\
&= a^m + cb^{mi} \sum_{k=0}^{m-2} \left(\tfrac{a}{b^i}\right)\left(\tfrac{a}{b^i}\right)^k + cb^{mi} \\
&= a^m + cb^{mi} \left[\sum_{k=0}^{m-2} \left(\tfrac{a}{b^i}\right)^{k+1} + 1\right]
\end{aligned}$$

$$\text{But } \sum_{k=0}^{m-2} \left(\frac{a}{b^i}\right)^{k+1} + 1 = \sum_{k=0}^{m-1} \left(\frac{a}{b^i}\right)^k$$

$$\text{Therefore } T(b^m) = a^m \ + \ cb^{mi} \sum_{k=0}^{m-1} \left(\frac{a}{b^i}\right)^k$$

which is what we wanted to show. And therefore, the closed form is true for **any** value of $j > 0$, i.e.

$$T(b^j) = a^j + cb^{ji} \sum_{k=0}^{j-1} \left(\frac{a}{b^i}\right)^k$$

We can still write the above in a more precise form by calculating the above summation

$$\sum_{k=0}^{j-1} \left(\frac{a}{b^i}\right)^k$$

which depends on the relationship between a and b^i.

- **Case 1:** $a = b^i$**:**

$$a = b^i \Rightarrow \sum_{k=0}^{j-1} \left(\frac{a}{b^i}\right)^k = \sum_{k=0}^{j-1} 1 = j$$

$$\Rightarrow T(b^j) = a^j + cjb^{ji} \Rightarrow T(b^j) = (b^i)^j + cjb^{ji}$$

$$\Rightarrow T(b^j) = b^{ji}(1 + cj)$$

- **Case 2:** $a > b^i$**:**

$$\textit{Recall:} \sum_{k=0}^{j-1} n^k = \frac{n^j - 1}{n - 1} \text{ if } n \neq 1$$

$$\Rightarrow T(b^j) = a^j + cb^{ji} \left[\frac{\left(\frac{a}{b^i}\right)^j - 1}{\left(\frac{a}{b^i}\right) - 1}\right]$$

- **Case 3:** $a < b^i$

$$T(b^j) = a^j + cb^{ji} \left[\frac{\left(\frac{a}{b^i}\right)^j - 1}{\left(\frac{a}{b^i}\right) - 1}\right]$$

$$\text{but since } \left(\frac{a}{b^i}\right)^j - 1 < 0 \text{ and } \frac{a}{b^i} - 1 < 0 \text{ rewrite it:}$$

$$T(b^j) = a^j + cb^{ji} \left[\frac{1 - \left(\frac{a}{b^i}\right)^j}{1 - \left(\frac{a}{b^i}\right)}\right]$$

Since $n = b^j$, let us calculate the growth rates with respect to n.
¿From Mathematics we know:

- $n = b^j \Rightarrow \log_b n = \log_b b^j \Rightarrow \log_b n = j$
- $a^{\log_b n} = \left(\underbrace{b^{\log_b a}}_{\text{a}}\right)^{\log_b n} = b^{(\log_b a)(\log_b n)} = \left(\underbrace{b^{\log_b n}}_{\text{n}}\right)^{\log_b a} = n^{\log_b a}$
- $\left(\frac{a}{b^i}\right)^{\log_b n} = \frac{a^{\log_b n}}{b^{i \log_b n}} = \frac{n^{\log_b a}}{b^{\log_b n^i}} = \frac{n^{\log_b a}}{n^i} = n^{\log_b a - i}$
- **Case 1:** $a = b^i$**:**
  - $a = b^i \Rightarrow \log_b a = \log_b b^i \Rightarrow \log_b a = i$
  - $T(b^j) = (b^j)^i(1 + cj)$

$$\Rightarrow T(n) = n^i(1 + c\log_b n)$$

$$\text{Therefore } T(n) \text{ is } \Theta(n^i \log_b n)$$

- **Case 2:** $a > b^i$**:**
  - $a > b^i \Rightarrow i < \log_b a$
  - $b^j = n \Rightarrow j = \log_b n$
  - $T(b^j) = a^j + cb^{ji} \left[\frac{\left(\frac{a}{b^i}\right)^j - 1}{\frac{a}{b^i} - 1}\right]$

$$\Rightarrow T(n) = n^{\log_b a} + cn^i \left[\frac{n^{\log_b a - i} - 1}{\frac{a}{b^i} - 1}\right] = n^{\log_b a} + \frac{cn^{\log_b a} - cn^i}{\frac{a}{b^i} - 1}$$

$$T(n) = n^{\log_b a}\left[1 + \frac{c}{\frac{a}{b^i} - 1}\right] - \frac{cn^i}{\frac{a}{b^i} - 1}$$

Thus the leading function of $T(n)$ is $c_1 n^{\log_b a}$ or $c_2 n^i$.
But $a > b^i \Rightarrow \log_b a > i$ and therefore

$$T(n) \text{ is } \Theta(n^{\log_b a})$$

- **Case 3:** $a < b^i$**:**
  - $a < b^i \Rightarrow i > \log_b a$
  - $T(b^j) = a^j + cb^{ji}\left[\frac{1-\left(\frac{a}{b^i}\right)^j}{1-\frac{a}{b^i}}\right]$

$$\begin{aligned}
\Rightarrow T(n) &= n^{\log_b a} + cn^i\left[\frac{1-n^{\log_b a - i}}{1-\frac{a}{b^i}}\right] \quad \text{(see Case 2)} \\
&= n^{\log_b a} + c\left[\frac{n^i - n^{\log_b a}}{1-\frac{a}{b^i}}\right] \\
&= n^{\log_b a}\left[1 - \frac{c}{1-\frac{a}{b^i}}\right] + n^i\left[\frac{c}{1-\frac{a}{b^i}}\right]
\end{aligned}$$

And with $i > \log_b a$ we have

$$T(n) \text{ is } \Theta(n^i)$$

We summarize the results in the next section.

2.2.3 Summary

The solution of the following recurrence relation

$$\begin{aligned}
T(1) &= 1 \\
T(n) &= aT(\tfrac{n}{b}) + cn^i \qquad n > 1
\end{aligned}$$

depends on the relationship between a and b^i:

Case	Growth Rate
$a < b^i$	$\Theta(n^i)$
$a = b^i$	$\Theta(n^i \log_b n)$
$a > b^i$	$\Theta(n^{\log_b a})$

And the closed form is:

$$T(b^j) = a^j + cb^{ji}\sum_{k=0}^{j-1}\left(\frac{a}{b^i}\right)^k$$

2.2.4 Rates of growth of some divide&conquer algorithms

- **Binary Search:**
Worst case analysis: $W(n) = 1 + W(\lfloor \frac{n}{2} \rfloor)$
With $n = 2^j \Rightarrow W(n) = 1 + W(\frac{n}{2})$

$$\begin{array}{lrcl} \text{i.e.} & W(2^j) & = & 1 + W(\frac{2^j}{2}) \\ \text{and} & W(1) & = & 1 \end{array}$$

So $a = 1$, $b = 2$ $i = 0$ and $c = 1$.
$a = 1 = 2^0 = b^i$, therefore the answer is given by Case 1:

$$T(b^j) = b^{ji}(1 + cj)$$

$$\text{So } W(n) = n^0(1 + 1\log_2 n) = 1 + \log_2 n$$

$$\Rightarrow \Theta(\log_2 n)$$

- Max-Min **Problem:**

$C(n)$	Total number of comparisons required to find the maximum and minimum values for an array A with n elements

$$C(1) = 0,\ C(2) = 1$$

$$C(n) = 2C\left(\frac{n}{2}\right) + 2 \quad \text{if } n > 2 \text{ is even.}$$

$$C(n) = C\left(\frac{n+1}{2}\right) + C\left(\frac{n-1}{2}\right) + 2 \quad \text{if } n > 2 \text{ is odd.}$$

If $n = 2^j$ i.e.:

$$\begin{array}{rcl} C(2^j) & = & 2C\left(\frac{2^j}{2}\right) + 2 \\ C(1) & = & 0 \end{array}$$

So $a = 2$, $b = 2$, $i = 0$ and $c = 2 \Rightarrow a > b^i$.
Therefore the answer is given by Case 2:

$$C(n) \text{ is } \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

- Mergesort **Problem:**
The number of comparisons used by Mergesort can be described by the following recurrence relation:

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

If $n = 2^j$

$$\Rightarrow C(n) = 2C\left(\frac{n}{2}\right) + cn$$

i.e. $a = 2$, $b = 2$ and $i = 1 \Rightarrow a = b^i$ so the answer is given by Case 1:

$$C(n) \text{ is } \Theta(n^i \log_b n) = \Theta(n^1 \log_2 n)$$

$$\Rightarrow \Theta(n \log_2 n)$$

Note: Exact solution:

$$C(n) = n\lfloor \log_2 n \rfloor + 2n - 2^{\lfloor \log_2 n \rfloor + 1}$$

In the next section we study Strassen's matrix multiplication algorithm.

2.3 Strassen's matrix multiplication [CLR90, Section 31.2]

Given two $n \times n$ matrices A and B, the straightforward algorithm to compute the product matrix C will have a complexity of $\Theta(n^3)$.

Suppose n is even. Partition A and B into $\frac{n}{2} \times \frac{n}{2}$ submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The product matrix, $C = AB$, can be written as:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Consequently:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Each of the 4 equations comprises of 2 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and the addition of their $\frac{n}{2} \times \frac{n}{2}$ products.

So a divide and conquer algorithm based on the above strategy will have as running time function (we assume here that n is a power of 2):

$$\begin{aligned} T(n) &= 8T\left(\tfrac{n}{2}\right) + cn^2 \quad n > 1 \\ T(n) &= 1 \end{aligned}$$

Here $a = 8$, $b = 2$ and $i = 2$, so $a > b^i$ and $T(n) = \Theta(n^{\log_2 8})$ (see Case 2). So $T(n)$ is $\Theta(n^3)$ which is not an improvement over the straightforward algorithm.

Strassen's method:
Strassen reduced the number of subproblems from 8 to 7 by using the following idea:
Define the following:

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

We can easily check that:

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

So a divide and conquer algorithm based on Strassen's strategy will have a running time function (assuming again that n is a power of 2):

$$\begin{aligned} T(n) &= 7T(\tfrac{n}{2}) + dn^2 \quad n > 1 \\ T(1) &= 1 \end{aligned}$$

Here $a = 7$, $b = 2$ and $i = 2$, so $T(n)$ is $\Theta(n^{\log_2 7}) = \Theta(n^{2.81})$.
But is Strassen's algorithm really better than the straightforward method ?

It depends on the value of n and whether the matrix is dense (few zero entries) or not. For large values of n (greater than 50), Strassen's algorithm is preferable. Why ?

We need more work to break down the problem into 7 subproblems: Multiplications and sums (or differences) of $\frac{n}{2} \times \frac{n}{2}$ matrices, instead of the 4 we have with the first divide and conquer algorithm.

By using more advanced techniques, we can multiply two $n \times n$ matrices in $\Theta(n^{2.376})$.

In the next section, we study another algorithm that is based on the divide and conquer technique.

2.4 Fast Fourier Transform [CLR90, Chapter 32]

1. **Multiplication of polynomials**
   If we have two polynomials, given in coefficient form:

$$A(x) = \sum_{j=0}^{n-1} a_j xj \qquad\qquad B(x) = \sum_{j=0}^{n-1} b_j x^j$$

   then we can write the product

$$C(x) = A(x)B(x) = \sum_{j=0}^{2n-2} c_j x^j \text{ where } c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

   Example:

$$A(x) = x^2 + 3x - 1\,; \qquad B(x) = x^2 - 2$$

$$\text{i.e.} \quad \begin{array}{lcccccccccc} A(x) & = & -1 & + & 3x & + & x^2 & + & 0x^3 & + & 0x^4 \\ B(x) & = & -2 & + & 0x & + & x^2 & + & 0x^3 & + & 0x^4 \end{array}$$

   and we get the following (*Note*: $n = 3$)

$$\begin{array}{lllllllllllll} c_0 & = & a_0b_0 & = & 2 \\ c_1 & = & a_0b_1 & + & a_1b_0 & = & -6 \\ c_2 & = & a_0b_2 & + & a_1b_1 & + & a_2b_0 & = & -3 \\ c_3 & = & a_0b_3 & + & a_1b_2 & + & a_2b_1 & + & a_3b_0 & = & 3 \\ c_4 & = & a_0b_4 & + & a_1b_3 & + & a_2b_2 & + & a_3b_1 & + & a_4b_0 & = & 1 \end{array}$$

   So $C(x) = 2 - 6x - 3x^2 + 3x^3 + x^4$.
   Clearly, computing $C(x) = A(x)B(x)$ has a growth rate of $\Theta(n^2)$.

2. **Point-value representation**
   A polynomial $A(x)$ of degree $n - 1$ can be represented by n point-value pairs:

$$\{(x_0, y_0),\ (x_1, y_1), \ldots , (x_{n-1}, y_{n-1})\}$$

   where:

   a) $y_k = A(x_k)$ b) all x_k are distinct.

   A polynomial has many different point-value representations, since any set of n distinct points x_0, x_1 … x_{n-1} can be used as a basis for the representation.

   Example:

$$A(x) = -1 + 3x + x^2 \qquad\qquad B(x) = -2 + x^2$$

One possible point-value representation of $A(x)$ and $B(x)$:

$$\begin{aligned} A(x) &= \{(-1,-3),(1,3),(2,9)\} \\ B(x) &= \{(-1,-1),(1,-1),(2,2)\} \end{aligned}$$

(a) Coefficient form $\longrightarrow$ Point-value representation
To compute a point-value representation for a given polynomial in coefficient form, choose n distinct points $x_0, \ldots, x_{n-1}$ and evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$ by using Horner's method:

$$A(x_k) = a_0 + x_k(a_1 + x_k(a_2 + \cdots + x_k(a_{n-2} + x_k a_{n-1}))\cdots)$$

Since it takes $\Theta(n)$ operations to evaluate $A(x_k)$ for each x_k, it takes $\Theta(n^2)$ to evaluate all n points. This operation is known as **evaluation**.

(b) Point-value representation $\longrightarrow$ Coefficient form (a.k.a. **interpolation**).
Interpolation is the process of determining the coefficient form of a polynomial from a point-value representation. Lagrange's formula is often used for interpolation:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\overset{j \neq k}{\Pi} (x - x_j)}{\overset{j \neq k}{\Pi} (x_k - x_j)}$$

EXAMPLE:

$$A(x) = \{(-1,-3),(1,3),(2,9)\}$$

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\overset{j \neq k}{\Pi} (x - x_j)}{\overset{j \neq k}{\Pi} (x_k - x_j)}$$

$$\Rightarrow A(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} +$$

$$+ y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

$$= -\frac{1}{2}(x-1)(x-2) - \frac{3}{2}(x+1)(x-2) + 3(x+1)(x-1)$$

$$\Rightarrow A(x) = x^2 + 3x - 1$$

Result: Interpolation is a well-defined operation.
Given any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs, there is a unique polynomial $A(x)$ of degree n such that:

$$y_k = A(x_k) \text{ for } k = 0, 1, \ldots\ n-1$$

The proof uses the Vandermonde matrix.

3. **Operations in point-value representation**
$A(x)$ and $B(x)$ are two polynomials in point-value representation where the first component of the pairs of the two polynomials are the same:

$$A(x) = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

$$B(x) = \{(x_0, y_0'), (x_1, y_1'), \ldots, (x_{n-1}, y_{n-1}')\}$$

$$\text{Then: } C(x) = A(x) + B(x)$$

$$\Rightarrow C(x) = \{(x_0, y_0 + y_0'), (x_1, y_1 + y_1'), \ldots, (x_{n-1}, y_{n-1} + y_{n-1}')\}$$

For multiplication, extend the point-value representations:

$$A(x) = \{(x_0, y_0) \cdots (x_{n-1}, y_{n-1}), (x_n, y_n), \dots (x_{2n-1}, y_{2n-1})\}$$
$$B(x) = \{(x_0, y'_0) \dots (x_{n-1}, y'_{n-1}), (x_n, y'_n), \dots (x_{2n-1}, y'_{2n-1})\}$$

Then a point-value representation for

$$C(x) = A(x)B(x) \text{ is:}$$

$$C(x) = \{(x_0, y_0 y'_0), (x_1, y_1 y'_1) \dots (x_{2n-1}, y_{2n-1} y'_{2n-1})\}.$$

So now we have a $\Theta(n)$ procedure to multiply two polynomials in point-value representation.

Example:

Extend

$$\begin{aligned} & A(x) = \{(-1,-3),(1,3),(2,9)\} \\ \text{and} \quad & B(x) = \{(-1,-1),(1,-1),(2,2)\} \end{aligned}$$

to obtain:

$$A(x) = \{(-1,-3),(1,3),(2,9),(-2,-3),(0,-1)\}$$

and

$$B(x) = \{(-1,-1),(1,-1),(2,2),(-2,2),(0,-2)\}$$

Recall: $A(x) = x^2 + 3x - 1$ and $B(x) = x^2 - 2$.

Compute $C(x) = A(x)B(x)$ from the point-value representations of $A(x)$ and $B(x)$:

$$C(x) = \{(-1,3),(1,-3),(2,18),(-2,-6),(0,2)\}$$

We can write $C(x)$ in coefficient form by using Lagrange's formula; but it is easier to check and see that $C(x)$ here is indeed $C(x) = x^4 + 3x^3 - 3x^2 - 6x + 2$, since $C(-1) = 3$, $C(1) = -3$, $C(2) = 18$, $C(-2) = -6$ and $C(0) = 2$. □

So we have:

Conversion between coefficient representation and point-value representation requires:

(a) generating $2n$ points

(b) evaluating $y_k = A(x_k)$ is an $\Theta(n)$ operation using Horner's rule.

So the conversion requires $\Theta(n^2)$ operations.

We can do better by choosing the evaluation points more carefully.

4. **Complex roots of unity**

A complex n^{th} root of unity is a complex number ω such that

$$\omega^n = 1\,.$$

There are exactly n such roots. These are

$$e^{\frac{2\pi i k}{n}} \text{ for } k = 0, 1, 2, \dots, n-1\,.$$

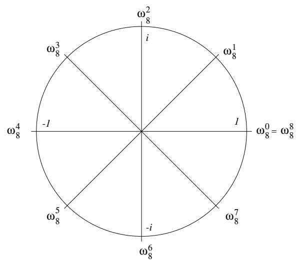
We use the definition of the exponential of a complex number

$$e^{iu} = \cos(u) + i\sin(u)$$

to interpret $e^{\frac{2\pi i k}{n}}$.

$$\text{So } e^{\frac{2\pi i k}{n}} = \cos\left(\frac{2\pi k}{n}\right) + i\sin\left(\frac{2\pi k}{n}\right) \text{ for } k = 0, 1, 2, \dots, n-1\,.$$

Example: $n = 8$



The 8 complex roots of unity $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ are equally spaced around the circle of unit radius at the origin of the complex plane. $\square$

The principle n^{th} root of unity is

$$\omega_n = e^{\frac{2\pi i}{n}}.$$

The other roots: $\omega_n^2, \omega_n^3, \omega_n^4, \ldots, \omega_n^{n-1}$ are powers of the principle n^{th} root of unity.

Remark: The n complex n^{th} roots of unity, $G_n = \{w_0^0, w_n^1, w_n^2, \ldots, w_n^{n-1}\}$, form a group under multiplication $\circ$.

- Closure

$$\forall\ w_n^j \in G_n \wedge w_n^k \in G_n :\ w_n^j \circ w_n^k \in G_n$$

  Note that

$$w_n^j \circ w_n^k = w_n^{j+k} = w_n^{(j+k) \bmod n}$$

$\circ$	w_4^0	w_4^1	w_4^2	w_4^3
w_4^0	1	w_4^1	w_4^2	w_4^3
w_4^1	w_4^1	w_4^2	w_4^3	1
w_4^2	w_4^2	w_4^3	1	w_4^1
w_4^3	w_4^3	1	w_4^1	w_4^2

- Identity element: $w_n^0 = 1$

$$\forall\ w_n^j \in G_n : w_n^j \circ w_n^0 = w_n^0 \circ w_n^j = w_n^j$$

- Associativity:

$$\forall\ w_n^j, w_n^k, w_n^l \in G_n : \left(w_n^j \circ w_n^k\right) \circ w_n^l = w_n^j \circ \left(w_n^k \circ w_n^l\right)$$

- Inverses:

$$\forall\ w_n^j \in G_n, \exists! w_n^k = w_n^{n-j} : w_n^j \circ w_n^{n-j} = w_n^{n-j} \circ w_n^j = w_n^n = w_n^0 = 1$$

5. **Properties of ω**

   (a) Cancellation Lemma:
   For any $n \geq 0$, $k \geq 0$, $d \geq 0$ we have $\omega_{dn}^{dk} = \omega_n^k$.
   Proof:

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^{k} \\ &= \omega_n^k \end{aligned}$$

   (b) *For any even integer $n > 0$, $\omega_n^{n/2} = \omega_2 = -1$.* In the example $\omega_8^4 = -1$ and $\omega_2 = e^{\pi i} = \cos\pi + i\sin\pi = -1$.

(c) Halving Lemma:

If $n > 2$ and even, then the squares of the n complex n^{th} roots of unity are the $\frac{n}{2}$ complex $\left(\frac{n}{2}\right)^{th}$ roots of unity (i.e. they evaluate to 1 when raised to $\left(\frac{n}{2}\right)^{th}$ power).

Proof:

By the cancellation lemma, we know that

$$\left(w_n^k\right) = w_{2\cdot\left(\frac{n}{2}\right)}^{2k} = w_{\frac{n}{2}}^{k} \quad \forall\ k > 0$$

And for $\omega_n^{k+n/2}$ we get:

$$\begin{aligned}
\left(\omega_n^{k+n/2}\right)^2 &= \omega_n^{2k+n} \\
&= \omega_n^{2k}\omega_n^n \\
&= \omega_n^{2k} \\
&= \left(\omega_n^k\right)^2 \\
&= \omega_{n/2}^k
\end{aligned}$$

□

EXAMPLE: w_8^3 is an 8^{th} root of unity; $\left(w_8^3\right)^2 = w_{2\cdot 4}^{2\cdot 3} = w_4^3$ is a 4^{th} root of unity.

Note that in the above proof, we have also shown that by squaring the complex n^{th} roots of unity, we obtain each $\left(\frac{n}{2}\right)^{th}$ root of unity exactly twice.

For EXAMPLE: The 8 complex 8^{th} roots of unity are:

$$w_8^0,\ w_8^1,\ w_8^2,\ w_8^3,\ w_8^4,\ w_8^5,\ w_8^6,\ w_8^7$$

By squaring each one, we have:

$$w_8^0,\ w_8^2,\ w_8^4,\ w_8^6,\ w_8^8 = w_8^0,\ w_8^{10} = w_8^2,\ w_8^{12} = w_8^4,\ w_8^{14} = w_8^6$$

which is equivalent (by the cancellation lemma) to:

$$w_4^0,\ w_4^1,\ w_4^2,\ w_4^3,\ w_4^0,\ w_4^1,\ w_4^2,\ w_4^3$$

Thus we see that by squaring all complex 8^{th} roots of unity, each $\left(\frac{8}{2}\right)^{th}$ root of unity is obtained exactly twice.

In other words, w_n^k and $w_n^{k+n/2}$ have the same square (and will therefore give two copies of an $\frac{n}{2}^{th}$ root of unity).

Note that the proof of this theorem is just part of the above proof. In other words:

Proof:

$$\begin{aligned}
\left(\omega_n^{k+n/2}\right)^2 &= \omega_n^{2k+n} \\
&= \omega_n^{2k}\omega_n^n \\
&= \omega_n^{2k}(1) \\
&= \left(\omega_n^k\right)^2
\end{aligned}$$

An alternative proof:

$$w_n^{n/2} = -1 \ \Rightarrow\ w_n^{k+n/2} = w_n^{n/2}w_n^k = -w_n^k$$

and so $\left(w_n^{k+n/2}\right)^2 = \left(w_n^k\right)^2$ □

(d) Summation Lemma:
For any integer $n \geq 1$ and nonnegative integer k not divisible by n, we have:

$$\sum_{j=0}^{n-1} \left(w_n^k\right)^j = 0$$

Proof: Recall

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \quad \text{for } x \neq 1 \text{ geometric series}$$

$$\begin{aligned}
\sum_{j=0}^{n-1} \left(w_n^k\right)^j &= \frac{\left(w_n^k\right)^n - 1}{w_n^k - 1} = \\
&= \frac{\left(w_n^n\right)^k - 1}{w_n^k - 1} = \\
&= \frac{1^k - 1}{w_n^k - 1} = \\
&= 0
\end{aligned}$$

Note that $w_n^k \neq 1$ since k is not divisible by n. □

6. **The Fast Fourier Transform**
Rather than dividing the polynomial in the middle (as is the case with other divide and conquer algorithms) we'll divide it into two parts by putting alternate terms in each part. To evaluate

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \,, \; n \text{ is even}$$

we divide it into two parts:

$$A^{even}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{\frac{n}{2}-1}$$

and

$$A^{odd}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{\frac{n}{2}-1} \,.$$

$A^{even}(x)$ contains all even-index coefficients of $A(x)$. $A^{odd}(x)$ contains the odd-index coefficients. We have

$$A(x) = A^{even}(x^2) + x A^{odd}(x^2) \,.$$

The n^{th} roots of unity are convenient for this decomposition because as we saw above, squaring a root of unity gives us another root of unity. So evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to:

(a) evaluating $A^{even}(x)$ and $A^{odd}(x)$ at $\left(\omega_n^0\right)^2, \left(\omega_n^1\right)^2, \left(\omega_n^2\right)^2, \cdots, \left(\omega_n^{n-1}\right)^2$. But these are the complex $\left(\frac{n}{2}\right)^{th}$ roots of unity. By the halving Lemma we know that there are only $\frac{n}{2}$ roots.
(b) Combine the results according to: $A(x) = A^{even}(x^2) + x A^{odd}(x^2)$.

7. **Recursive Fast Fourier Transform**
Assume that n is a power of 2.

```
RecurFFT(a, n)                              // a is the list of n coefficients
    begin
        if n = 1 then return a
        ω_n = e^(2πi/n)
        ω = 1
        A^even = (a_0, a_2, a_4 ··· , a_{n-2})
        A^odd = (a_1, a_3, a_5 ··· , a_{n-1})
        y^even = RecurFFT(A^even, n/2)
        y^odd = RecurFFT(A^odd, n/2)
        for k = 0 to n/2 - 1
            y_k = y_k^even + ω y_k^odd
            y_{k+n/2} = y_k^even - ω y_k^odd
            ω = ω ω_n
        return y
    end
```

8. Example:
Evaluate
$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_7x^7$$
at the 8 complex 8^{th} roots of unity
$$w_8^0,\ w_8^1,\ w_8^2,\ \ldots, w_8^7 \quad (*)$$
Recall: Since $w_8^4 = -1$ (property (b)), evaluating $A(x)$ at points $(*)$ is the same as evaluating $A(x)$ at:
$$w_8^0,\ w_8^1,\ w_8^2,\ w_8^3,\ -w_8^0,\ -w_8^1,\ -w_8^2,\ -w_8^3$$
Squaring each term gives two copies of the 4 complex roots of the 4^{th} roots of unity:
$$w_4^0,\ w_4^1,\ w_4^2,\ w_4^3,\ w_4^0,\ w_4^1,\ w_4^2,\ w_4^3$$
So by using:
$$A(x) = A^{even}(x^2) + xA^{odd}(x^2)$$
we can recombine the results of computing A^{even} and A^{odd} at the 4^{th} roots of unity. More precisely:

(a) Evaluate $A^{even}(x)$ and $A^{odd}(x)$ at the 4^{th} roots of unity.
(b) Substitute each of the 8^{th} roots of unity for x in
$$A(x) = A^{even}(x^2) + xA^{odd}(x^2)$$

$$\begin{array}{rcccc}
A(w_8^0) & = & A^{even}(\overbrace{w_4^0}^{=(w_8^0)^2}) & + & w_8^0A^{odd}(\overbrace{w_4^0}^{=(w_8^0)^2}) \\
A(w_8^1) & = & A^{even}(w_4^1) & + & w_8^1A^{odd}(w_4^1) \\
A(w_8^2) & = & A^{even}(w_4^2) & + & w_8^2A^{odd}(w_4^2) \\
A(w_8^3) & = & A^{even}(w_4^3) & + & w_8^3A^{odd}(w_4^3) \\
A(w_8^4) & = & A^{even}(w_4^0) & - & w_8^0A^{odd}(w_4^0) \\
A(w_8^5) & = & A^{even}(w_4^1) & - & w_8^1A^{odd}(w_4^1) \\
A(w_8^6) & = & A^{even}(w_4^2) & - & w_8^2A^{odd}(w_4^2) \\
A(w_8^7) & = & A^{even}(w_4^3) & - & w_8^3A^{odd}(w_4^3)
\end{array}$$

For instance:

$$\begin{aligned}
A(w_8^0) &= a_0 + a_1 w_8^0 + a_2(w_8^0)^2 + a_3(w_8^0)^3 + \\
&\quad a_4(w_8^0)^4 + a_5(w_8^0)^5 + a_6(w_8^0)^6 + a_7(w_8^0)^7 \\
A(w_8^0) &= a_0 + a_2(w_8^0)^2 + a_4(w_8^0)^4 + a_6(w_8^0)^6 + \\
&\quad w_8^0[a_1 + a_3(w_8^0)^2 + a_5(w_8^0)^4 + a_7(w_8^0)^6] \\
A(w_8^0) &= a_0 + a_2 w_4^0 + a_4(w_4^0)^2 + a_6(w_4^0)^3 + \\
&\quad w_8^0[a_1 + a_3 w_4^0 + a_5(w_4^0)^2 + a_7(w_4^0)^3] \\
\Rightarrow\ A(w_8^0) &= A^{even}(w_4^0) + w_8^0 A^{odd}(w_4^0)
\end{aligned}$$

and also,

$$\begin{aligned}
A(w_8^4) &= a_0 + a_1 w_8^4 + a_2(w_8^4)^2 + a_3(w_8^4)^3 + \\
&\quad a_4(w_8^4)^4 + a_5(w_8^4)^5 + a_6(w_8^4)^6 + a_7(w_8^4)^7 \\
A(w_8^4) &= a_0 + a_1(-w_8^0) + a_2(-w_8^0)^2 + a_3(-w_8^0)^3 + \\
&\quad a_4(-w_8^0)^4 + a_5(-w_8^0)^5 + a_6(-w_8^0)^6 + a_7(-w_8^0)^7 \\
&= a_0 + a_2(w_8^0)^2 + a_4(w_8^0)^4 + a_6(w_8^0)^6 - w_8^0[a_1 + \\
&\quad a_3(w_8^0)^2 + a_5(w_8^0)^4 + a_7(w_8^0)^6] \\
A(w_8^4) &= a_0 + a_2 w_4^0 + a_4(w_4^0)^2 + a_6(w_4^0)^3 - \\
&\quad w_8^0[a_1 + a_3 w_4^0 + a_5(w_4^0)^2 + a_7(w_4^0)^3] \\
\Rightarrow\ A(w_8^0 &= A^{even}(w_4^0) - w_8^0 A^{odd}(w_4^0)
\end{aligned}$$

In general, to evaluate $A(x)$ at the n^{th} roots of unity, recursively evaluate $A^{even}(x)$ and $A^{odd}(x)$ at the $\frac{n}{2}^{th}$ roots of unity and perform the n substitutions as was done with $n = 8$.

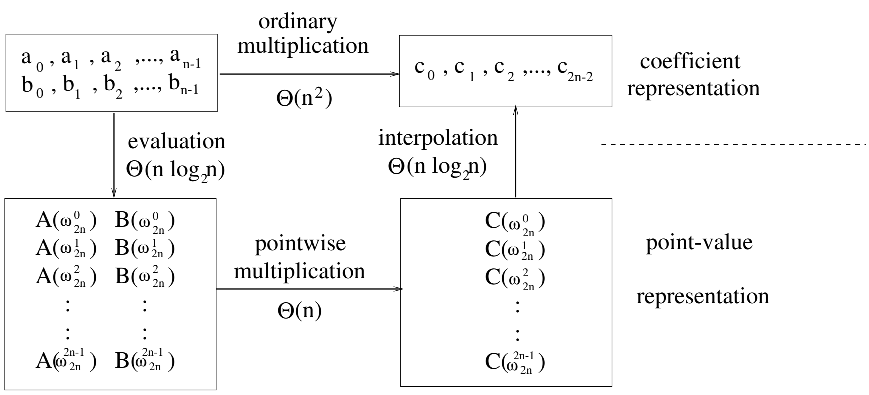
9. **Analysis of the algorithm**

The running time function of RecurFFT is given by

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)\,.$$

Here $a = 2$, $b = 2$ and $i = 1$, so $a = b^i$. And therefore T(n) is $\Theta(n \log_2 n)$.

The whole process of efficiently multiplying two polynomials is summarized in the following graph:



In the next section we study another family of recurrence relations, that are not generally encountered when studying algorithms based on the divide and conquer approach.

2.5 Other recurrence relations

Recurrence relations that do not necessarily come from divide and conquer algorithms.
Bubble Sort:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(n-1) + n - 1 \end{aligned}$$

The recurrence relation can be solved by:

(a) **Substitution**: Then use induction to prove that the closed form is correct

(b) using **homogeneous linear equation technique**

(c) using **generating functions**.

2.5.1 Linear recurrence relations with constant coefficients [Liu68, Chapter 7],[Liu85, Chapter 10]

$$c_0T(n) + c_1T(n-1) + c_2T(n) + \ldots + c_kT(n-k) = f(n)$$

is a linear recurrence relation with constant coefficients of k^{th} order. Here c_i's are constants, $c_0 \neq 0\,,\ c_k \neq 0$.
Remark: If k consecutive values are know, the other values can be determined (i.e. $T(n)$ will be uniquely defined).
Example:

$$3T(n) - 5T(n-1) + 2T(n-2) = n^2 + 5$$

is a $2nd$ order linear recurrence relation with constant coefficients. If $T(3)$=3 and $T(4) = 6$, then:

$$3T(4) - 5T(3) + 2T(2) = 4^2 + 5$$

$$\Rightarrow T(2) = 9$$

$$3T(5) - 5T(4) + 2T(3) = 5^2 + 5 \Rightarrow T(5) = 18 \text{ and so on...}$$

k non-consecutive values might not constitute an appropriate set of boundary conditions (problem dependent).
The total solution of a linear recurrence relation with constant coefficients is given by:

$$T(n) = T_h(n) + T_p(n)\,,$$

where

a.) $T_h(n)$ is the homogeneous solution and satisfies the linear recurrence relation when the right-hand side of the equation is set to 0 (i.e. $f(n) = 0$).

b.) $T_p(n)$ is the particular solution and satisfies the linear recurrence relation with $f(n)$ on the right-hand side.

A homogeneous solution of a linear recurrence relation is of the form $A\alpha^n$, where α is a characteristic root and A is a constant determined by the boundary conditions.
Note that by substituting $A\alpha^n$ for $T(n)$, we get:

$$c_0A\alpha^n + c_1A\alpha^{n-1} + c_2A\alpha^{n-2} + \cdots + c_kA\alpha^{n-k} = 0$$

$$\Rightarrow\ A\alpha^{n-k}(c_0\alpha^k + c_1\alpha^{k-1} + \cdots + c_k) = 0$$

$$\Leftrightarrow\ c_0\alpha^k + c_1\alpha^{k-1} + \cdots + c_k = 0$$

which is called the characteristic equation of the recurrence relation. The characteristic equation of the k^{th} degree has k characteristic roots.

1. If the k characteristic roots are distinct then:

$$T_h(n) = A_1\alpha_1^n + A_2\alpha 2^n + \cdots + A_k\alpha_k^n$$

where $\alpha_1, \alpha_2, \ldots, \alpha_k$ are the roots of the characteristic equation and $A_1, A_2, \ldots, A_k$ are constants determined by the boundary conditions.

2. If the k characteristic roots are not distinct, then some of the roots are multiple roots. If α_1 is a root of multiplicity m, then:

$$(A_1 n^{m-1} + A_2 n^{m-2} + \cdots + A_{m-1}n + A_m)\alpha^n$$

is a homogeneous solution.

EXAMPLES:

1. Solve the following recurrence relation:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ \left.\begin{aligned} T(0) &= 0 \\ T(1) &= 1 \end{aligned}\right\} &\text{boundary conditions} \end{aligned}$$

(i) Find $T_h(n)$, so solve $T(n) - T(n-1) - T(n-2) = 0$. Note that $f(n) = 0$ and so there is no particular solution here i.e. $T_p(n) = 0$.

$$T(n) = A\alpha^n \Rightarrow A\alpha^n - A\alpha^{n-1} - A\alpha^{n-2} = 0$$

$$\Rightarrow A\alpha^{n-2}(\alpha^2 - \alpha - 1) = 0$$

$\alpha^2 - \alpha - 1 = 0$ is the characteristic equation.

$$\Rightarrow \alpha_1 = \frac{1+\sqrt{5}}{2} \text{ and } \alpha_2 = \frac{1-\sqrt{5}}{2} \text{ are the two distinct roots.}$$

Therefore

$$T_h(n) = A_1\left(\frac{1-\sqrt{5}}{2}\right)^n + A_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

is the homogeneous solution.

(ii) to calculate A_1 and A_2 use the boundary conditions:

$$T_h(0) = A_1\left(\frac{1+\sqrt{5}}{2}\right)^0 + A_2\left(\frac{1-\sqrt{5}}{2}\right)^0$$

$$\Rightarrow A_1 + A_2 = 1 \quad (\text{since } T(0) = 1 \text{ given})$$

$$T_h(1) = A_1\left(\frac{1+\sqrt{5}}{2}\right) + A_2\left(\frac{1-\sqrt{5}}{2}\right)$$

$$\Rightarrow \frac{(A_1 + A_2) + \sqrt{5}(A_1 - A_2)}{2} = 1 \quad (\text{since } T(1) = 1 \text{ given})$$

$$\Rightarrow A_1 - A_2 = \frac{1}{\sqrt{5}} \quad (\text{since } A_1 + A_2 = 1\,)$$

Solve

$$\begin{aligned} A_1 + A_2 &= 1 \\ A_1 - A_2 &= \tfrac{\sqrt{5}}{5} \end{aligned} \quad \Rightarrow \quad \left\{ \begin{aligned} A_1 &= \tfrac{5+\sqrt{5}}{10} \\ A_2 &= \tfrac{5-\sqrt{5}}{10} \end{aligned} \right.$$

So

$$T(n) = T_h(n) + T_p(n)$$

$$= \frac{5+\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{10}\right)^n + \frac{5-\sqrt{5}}{10}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

2. Solve
$$T(n) + 6T(n-1) + 12T(n-2) + 8T(n-3) = 0$$
$$T(n) = A\alpha^n$$
$$\Rightarrow \ \alpha^3 + 6\alpha^2 + 12\alpha + 8 = 0$$
is the characteristic equation.
$$\alpha^3 + 6\alpha^2 + 12\alpha + 8 = (\alpha + 2)(\alpha^2 + 4\alpha + 4) = (\alpha + 2)^3$$
So $\alpha = -2$ is a characteristic root of multiplicity 3.
And therefore
$$T_h(n) = (A_1 n^2 + A_2 n + A_3)(-2)^n$$
is a homogeneous solution. To determine A_1, A_2 and A_3 use the boundary conditions.
Remark: Recall $T(n) = T_h(n) + T_p(n)$. There is no general method for finding the particular solution. Inspection is a good technique that sometimes works.

In general, when the right-hand side of the recurrence relation is of the form n^t, the corresponding particular solution will be of the form:
$$P_1 n^t + P_2 n^{t-1} + \cdots + P_t n + P_{t+1}$$
where P_i's are constants.

3. Find the particular solution of:
$$T(n) + 5T(n-1) + 6T(n-2) = 3n^2$$
Solution:
$$\text{try } T_p(n) = P_1 n^2 + P_2 n + P_3$$
$$\Rightarrow \ P_1 n^2 + P_2 n + P_3 + 5P_1(n-1)^2 +$$
$$5P_2(n-1) + 5P_3 + 6P_1(n-2)^2 + 6P_2(n-2) + 6P_3 = 3n^2$$
$$\Rightarrow \ 12P_1 n^2 + (34P_1 - 12P_2)n + (29P_1 - 17P_2 + 12P_3) = 3n^2$$
$$\Rightarrow \ \begin{cases} 12P_1 & = 3 \\ 34P_1 - 12P_2 & = 0 \\ 29P_1 - 17P_2 + 12P_3 & = 0 \end{cases} \ \Rightarrow \ \begin{cases} P_1 & = \frac{1}{4} \\ P_2 & = \frac{17}{24} \\ P_3 & = \frac{115}{288} \end{cases}$$
Therefore
$$T_p(n) = \frac{1}{4}n^2 + \frac{17}{24}n + \frac{115}{288}\,.$$

When $f(n)$ is of the form of a polynomial of degree t:
$$F_1 n^t + F_2 n^{t-1} + \cdots + F_t n + F_{t+1}$$
then the corresponding solution will be of the form:
$$P_1 n^t + P_2 n^{t-1} + P_3 n^{t-2} + \cdots + P_t n + P_{t+1}\,.$$

In general:

Form of $f(n)$	Form of $T_p(n)$
β^n	$P\beta^n$, if β is **not** a characteristic root of the recurrence relation
$(F_1 n^t + F_2 n^{t-1} + \cdots + F_t n + F_{t+1})\beta^n$	$(P_1 n^t + P_2 n^{t-1} + \cdots + P_t n + P_{t+1})\beta^n$, if β is **not** a characteristic root of the recurrence relation AND $n^{m-1}(P_1 n^t + P_2 n^{t-1} + \cdots + P_t n + P_{t+1})\beta^n$ if β is a characteristic root of multiplicity $m-1$ of the recurrence relation.

EXAMPLES

1. Solve

$$\begin{aligned} &T(n) + 5T(n-1) + 6T(n-2) = 42 \cdot 4^n \\ &T(2) = 278 \\ &T(3) = 962 \end{aligned}$$

(a) **homogeneous solution**

$$T(n) = A\alpha^n \Rightarrow \ \alpha^3 + 5\alpha + 6 = 0$$

$$\Rightarrow \begin{cases} \alpha_1 = -2 \\ \alpha_2 = -3 \end{cases}$$

Therefore $T_h(n) = A_1(-2)^n + A_2(-3)^n$.

(b) **particular solution**

$$f(n) = 42 \cdot 4^n \text{ and 4 is not a chracteristic root.}$$

So try $T_p(n) = P4^n$.

$$P4^n + 5P4^{n-1} + 6P4^{n-2} = 42 \cdot 4^n$$

$$P4^{n-2}(4^2 + 5(4) + 6) = 42 \cdot 4^n$$

$$\frac{P4^n \cdot 42}{16} = 42 \cdot 4^n \ \Rightarrow \ P = 16$$

Therefore the particular solution is

$$T_p(n) = 16 \cdot 4^n$$

So the total solution is:

$$T(n) = T_h(n) + T_p(n) = A_1(-2)^n + A_2(-3)^n + 16 \cdot 4^n$$

(c) **use boundary conditions**

$$T(2) = 278\,, \ T(3) = 962$$

For $n = 2$:

$$T(2) + A_1 \cdot 4 + A_2 \cdot 9 + 16 \cdot 16$$

$$\text{So } \ 278 = 4A_1 + 9A_2 + 256 \qquad (1)$$

For $n = 3$:

$$T(3) = A_1(-2)^3 + A_2(-3)^3 + 16 \cdot 4^3$$
$$\Rightarrow 962 = -8A_1 - 27A_2 + 1024 \qquad (2)$$

Solving (1) and (2) we get:

$$A_1 = 1 \text{ and } A_2 = 2$$

So the solution is:

$$T(n) = (-2)^n + 2(-3)^n + 16 \cdot 4^n$$

2. Solve
$$\begin{array}{l} T(n) + T(n-1) = 3n2^n \\ T(0) = \frac{5}{3} \end{array}$$

   (a) **homogeneous solution**

$$T(n) = A\alpha^n \Rightarrow \begin{array}{l} A\alpha^n + A\alpha^{n-1} = 0 \\ A\alpha^{n-1}(\alpha+1) \\ \alpha = -1 \end{array}$$

   (b) **particular solution**

$$f(n) = 3n2^n \textbf{ is of the form } (F_1 n + F_2)\beta^n$$

where β is not a characteristic root. So try

$$T_p(n) = (P_1 n + P_2)2^n$$
$$\Rightarrow \ (P_1 n + P_2)2^n + [P_1(n-1) + P_2]2^{n-1} = 3n2^n$$
$$\Rightarrow \ 3P_1 n - P_1 + 3P_2 = 6n \ \Rightarrow \ 3P_1 = 6 \text{ and } 3P_2 - P_1 = 0$$
$$\Rightarrow P_1 = 2 \text{ and } P_2 = \frac{2}{3}$$
$$\Rightarrow \ T_p(n) = (2n + \frac{2}{3})2^n$$

   (c) **use boundary conditions**

$$\begin{aligned} T(n) &= T_h(n) + T_p(n) \\ &= A(-1)^n + (2n + \tfrac{2}{3})2^n \end{aligned}$$

$$T(0) = \frac{5}{3} \ \Rightarrow \ \frac{5}{3} = A + \frac{2}{3} \ \Rightarrow \ A = 1$$

Therefore

$$T(n) = (-1)^n + (2n + \frac{2}{3})2^n$$

3. Solve

$$\begin{array}{rcl} T(n) - 2T(n-1) &=& 3 \cdot 2^2 \\ T(0) &=& 1 \end{array}$$

   (a) **homogeneous solution:**

$$T(n) = A\alpha^n \Rightarrow \ A\alpha^n - 2A\alpha^{n-1} = 0$$
$$\Rightarrow \ A\alpha^{n-1}(\alpha - 2) = 0$$
$$\Rightarrow \ T_h(n) = A \cdot 2^n$$

(b) **particular solution**

$$f(n) = 3 \cdot 2^n \text{ and 2 is a characteristic root of multiplicity 1}$$

So try

$$T_p(n) = nP2^n$$
$$\Rightarrow \; Pn2^n - 2P(n-1)2^{n-1} = 3 \cdot 2^n \; \Rightarrow \; P = 3$$

So $T_p(n) = 3n(2)^n$.

(c) **use boundary value**

$$T(n) = T_h(n) + T_p(n) = A \cdot 2^n + 3n2^n$$

$$T(0) = 1 \; \Rightarrow \; 1 = A + 0 \; \Rightarrow \; A = 1$$

So $T(n) = 3n(2)^n + (2)^n = 2^n(3n+1)$.

4. Solve

$$\begin{aligned} T(n+2) &= 4T(n+1) - 4T(n) + 2^n \\ T(0) &= 2 \\ T(1) &= 4 \end{aligned}$$

(a) **homogeneous solution:**

$$T(n) = A\alpha^n \Rightarrow \; A\alpha^n(\alpha^2 - 4\alpha + 4) = 0 \Rightarrow (\alpha - 2)^2 = 0$$

$\alpha = 2$ is a root of multiplicity 2. So

$$T_h(n) = (A_1 n + A_2)2^n$$

(b) **particular solution:**

$$T(n+2) - 4T(n+1) + 4T(n) = 2^n$$

$f(n) = 2^n$ and 2 is a root of multiplicity 2. So try

$$T_p(n) = Pn^2 2^n$$

$$P(n+2)^2 2^{n+2} - 4P(n+1)^2 2^{n+1} + 4Pn^2 2^n = 2^n$$
$$\Rightarrow \; P = \frac{1}{8} \; \Rightarrow \; T_p(n) = n^2 2^{n-3}$$

(c) **use boundary conditions:**

$$T(n) = T_h(n) + T_p(n) = A_1 n 2^n + A_2 2^n + n^2 2^{n-3}$$

$$\begin{aligned} T(0) &= 2 \\ T(1) &= 4 \end{aligned} \; \Rightarrow \; \begin{aligned} A_1 &= 2 \\ A_2 &= -\tfrac{1}{8} \end{aligned}$$

Therefore $T(n) = 2^{n+1} - n2^{n-3} + n^2 2^{n-3}$.

5. Solve

$$\begin{aligned} T(n) &= T(n-1) - T(n-2) \\ T(1) &= 1 \\ T(2) &= 0 \end{aligned}$$

$\alpha^2 - \alpha + 1 = 0$ is the characteristic equation.

$$\Rightarrow \ \alpha_1 = \frac{1-\sqrt{3}i}{2} \ \alpha_2 = \frac{1+\sqrt{3}i}{2}$$

$$T_h(n) = A_1 \left(\frac{1-\sqrt{3}i}{2}\right)^n + A_2 \left(\frac{1+\sqrt{3}i}{2}\right)^n$$

Note: $T(n) = T(n-1) - T(n-2) \Rightarrow T(2) = T(1) - T(0)$

$$\Rightarrow T(0) = 1$$

$$T_h(n) = A_1 \left(\frac{1-\sqrt{3}i}{2}\right)^n + A_2 \left(\frac{1+\sqrt{3}i}{2}\right)^n$$

$$T(1) = 1 \ \Rightarrow \ 1 = A_1 \left(\frac{1-\sqrt{3}i}{2}\right) + A_2 \left(\frac{1+\sqrt{3}i}{2}\right) \qquad (1)$$

$$T(0) = 1 \ \Rightarrow \ 1 = A_1 + A_2$$

$$\Rightarrow \ \frac{1+\sqrt{3}i}{2} = A_1 \left(\frac{1+\sqrt{3}i}{2}\right) + A_2 \left(\frac{1+\sqrt{3}i}{2}\right) \qquad (2)$$

$$(2) - (1) \ \Rightarrow \ \frac{1+\sqrt{3}i}{2} - 1 = A_1 \left(\frac{1+\sqrt{3}i}{2} - \frac{1-\sqrt{3}i}{2}\right) + A_2 \cdot 0$$

$$\Rightarrow \ \frac{1+\sqrt{3}i-2}{2} = A_1 \left(\frac{2\sqrt{3}i}{2}\right) \Rightarrow \ A_1 = \frac{(\sqrt{3}i-1)(\sqrt{3}i)}{(2\sqrt{3}i)(\sqrt{3}i)}$$

$$A_1 = \frac{3+\sqrt{3}i}{6} \ \Rightarrow \ A_2 = 1 - A_1 = \frac{3-\sqrt{3}i}{6}$$

So

$$T(n) = \left(\frac{3+\sqrt{3}i}{6}\right)\left(\frac{1-\sqrt{3}i}{2}\right)^n + \left(\frac{3-\sqrt{3}i}{6}\right)\left(\frac{1+\sqrt{3}i}{2}\right)^n$$

Note: We don't have a particular solution here.

6. Solve

$$R^2(n) - 2R^2(n-1) = 1$$

$$R(0) = 2$$

Note: $R^2(n)$ means $[R(n)]^2$.
Let $T(n) = R^2(n)$.

$$\begin{aligned} R^2(n) - 2R^2(n-1) &= 1 \\ R(0) &= 2 \end{aligned} \quad \Rightarrow \quad \begin{aligned} T(n) - 2T(n-1) &= 1 \\ T(0) = R^2(0) &= 4 \end{aligned}$$

$$T(n) = A\alpha^n \ \Rightarrow \ A\alpha^n - 2A\alpha^{n-1} = 0$$

$$A\alpha^{n-1}(\alpha - 2) = 0 \ \Rightarrow \ \alpha = 2$$

So $T_h(n) = A2^n$.

$$T_p(n) = B \qquad B - 2B = 1 \Rightarrow B = -1$$

So $T(n) = A2^n - 1$.

$$T(0) = 4 \Rightarrow 4 = A - 1 \Rightarrow A = 5$$

Therefore $T(n) = 5 \cdot 2^n - 1$. And $R(n) = \sqrt{5 \cdot 2^n - 1}$.

In the next section we show how generating functions can be used to solve recurrence relations.

2.5.2 Generating functions

To better understand generating functions, we start by defining **power series**.

$$\sum_{k=0}^{\infty} a_k x^k = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k + \cdots$$

is called a power series

- expansion about the origin
- think of a power series as a polynomial of infinite degree

If the expansion is about the point $x = a$, then the power series is a Taylor series and has the form

$$\sum_{k=0}^{\infty} a_k (x-a)^k = a_0 + a_1(x-a) + a_2(x-a)^2 + \cdots + a_k(x-a)^k + \cdots$$

Fact: The convergence set of a power series is always an interval with a as the midpoint of the interval of convergence. The interval of convergence might be reduced to a point, or may consist of all real numbers.

Fact: If the Taylor series $f(x) = \sum_{k=0}^{\infty} a_k(x-a)^k$ and all of the series obtained by differentiating it [i.e. $f'(x) = \sum_{k=1}^{\infty} k a_k x^{k-1}$, $f''(x) = \ldots$ etc.] converge for $a - \tau < x < a + \tau$ where $\tau > 0$, then:

$$a_k = \frac{f^{(k)}(a)}{k!} \text{ for } k = 0, 1, 2, \ldots$$

Examples:

① *Show that the geometric series* $\sum_{k=0}^{\infty} x^k$ *converges for* $-1 < x < 1$ *i.e. interval of convergence is the open interval:* $(-1; 1)$.

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad -1 < x < 1$$

$$\text{Let } f(x) = \frac{1}{1-x}\,,\ f'(x) = \frac{1}{(1-x)^2}$$

$$f''(x) = \frac{2}{(1-x)^3}\,,\ f^{(k)}(x) = \frac{k!}{(1-x)^{k+1}}$$

$$\text{So}\quad \frac{f^{(k)}(0)}{k!} = \frac{k!}{k!(1-0)^{k+1}} = 1$$

$$\text{i.e.}\quad a_k = 1 \text{ for } k = 0, 1, 2, \ldots$$

So $f(x) = f(0) + \frac{f'(0)}{1!}(x-0) + \frac{f''(0)}{2!}(x-0)^2 + \cdots + \frac{f^{(k)}(0)}{k!}(x-0)^k + \cdots$.

$$f(x) = 1 + x + x^2 + \cdots + x^k + \cdots$$

$$\text{i.e.}\ \frac{1}{1-x} = 1 + x + x^2 + \cdots + x^k + \cdots$$

□

② *Show that if* $f(x) = \frac{1}{(1-x)^2}$ *then* $a_k = k+1$. *In other words, show that*

$$\sum_{k=0}^{\infty}(k+1)x^k = \frac{1}{(1-x)^2}$$

$$f(x) = \frac{1}{(1-x)^2}\ ,\ \ f'(x) = \frac{2}{(1-x)^3}\ ,\ \ f''(x) = \frac{2\cdot 3}{(1-x)^4}$$

$$\text{and}\quad f^{(k)}(x) = \frac{(k+1)!}{(1-x)^{k+2}}$$

So

$$a_k = \frac{f^{(k)}(0)}{k!} = \frac{(k+1)!}{k!(1-0)^{k+2}} = k+1$$

We showed that $a_k = k+1$, i.e.

$$\frac{1}{(1-x)^2} = \sum_{k=0}^{\infty}(k+1)x^k$$

Notation: $G(x) = \frac{1}{(1-x)^2}$.
$G(x)$ is called the **generating function** of $a_0, a_1, a_2, \ldots$.
Here $G(x) = \frac{1}{(1-x)^2}$ is the generating function of $1, 2, 3, \ldots, k+1, \ldots$.

□

Other method for tackling $G(x) = \frac{1}{(1-x)^2}$: We know

$$\frac{1}{1-x} = 1 + x + x^2 + \cdots + x^k + \cdots$$

By differentiating both sides k times (and dividing by $k!$), we obtain:

$$\frac{1}{(1-x)^{k+1}} = 1 + (k+1)x + \frac{(k+1)(k+2)}{2!}x^2 + \cdots + \frac{(n+k)!}{k!n!}x^n + \cdots$$

Now let $k = 1$. □

③ *Show that if* $G(x) = (1+x)^r$ *then* $a_k = \binom{r}{k}$, *for any* $r \in \mathbf{R}$.

$$\begin{aligned}
G(x) &= (1+x)^r \\
G'(x) &= r(1+x)^{r-1} \\
G''(x) &= r(r-1)(1+x)^{r-2} \\
&\ \vdots \qquad \vdots \\
G^{(k)}(x) &= r(r-1)(r-2)\cdots(r-(k-1))(1+x)^{r-k} \\
&= \frac{[1\cdot 2\cdots(r-k-1)(r-k)](r-k+1)\cdots r}{1\cdot 2\cdot 3\cdots(r-k)}(1+x)^{r-k} \\
&= \frac{r!}{(r-k)!}(1+x)^{r-k}
\end{aligned}$$

$$\Rightarrow G^{(k)}(0) = \frac{r!}{(r-k)!}(1+0)^{r-k}$$

$$\text{And } \frac{G^{(k)}(0)}{k!} = \frac{r!}{k!(r-k)!}(1+0)^{r-k} \Rightarrow a_k = \binom{r}{k}$$

So $(1+x)^r = 1 + \binom{r}{1}x + \binom{r}{2}x^2 + \cdots + \binom{r}{k}x^k + \cdots$.
In other words: $G(x) = (1+x)^r$ is the generating function of:

$$\binom{r}{0},\ \binom{r}{1},\ \binom{r}{2},\ \ldots,\ \binom{r}{k},\ \ldots$$

$\square$

④ *Show that* $G(x) = \ln\left(\frac{1}{1-x}\right)$ *is the generating function of :*

$$0,\ \frac{1}{2},\ \frac{1}{3},\ \ldots \frac{1}{k},\ \ldots$$

$$\begin{aligned} G'(x) &= \frac{1}{1-x} \\ G''(x) &= \frac{1}{(1-x)^2} \\ G'''(x) &= \frac{2}{(1-x)^3} \\ &\vdots \\ G^{(k)}(x) &= \frac{(k-1)!}{(1-x)^k} \end{aligned}$$

$$\text{So } \frac{G^{(k)}(0)}{k!} = \frac{(k-1)!}{k!(1-0)^k}$$

i.e. $a_k = \frac{1}{k}$. And

$$\ln\left(\frac{1}{1-x}\right) = 0 + x + \frac{1}{2}x^2 + \cdots + \frac{1}{k}x^k + \cdots$$

$\square$

Other Method
We know:

$$1 + x + x^2 + x^3 + \cdots + x^n + \cdots = \frac{1}{1-x}$$

We integrate both sides to obtain the following:

$$x + \frac{1}{2}x^2 + \frac{1}{3}x^3 + \cdots + \frac{1}{n+1}x^{n+1} + \cdots = \int \frac{1}{1-x}dx$$

But $\int \frac{1}{1-x}dx = \ln \frac{1}{1-x}$.

$$\text{So } \begin{cases} a_0 &= 0 \\ a_k &= \frac{1}{k} \text{ for } k \geq 1 \end{cases}$$

$\square$

⑤ *Show that the generating function of*

$$\left\{\begin{array}{ll} a_0 & = 0 \\ a_k & = \frac{1}{k} \text{ if } k \text{ is odd} \\ a_k & = -\frac{1}{k} \text{ if } k \text{ is even} \end{array}\right\} \quad \text{is } \ln(1+x)$$

We know

$$\begin{aligned} y + \frac{y^2}{2} + \frac{y^3}{3} + \cdots + \frac{y^n}{n} + \cdots &= \ln\left(\frac{1}{1-y}\right) \\ y + \frac{y^2}{2} + \frac{y^3}{3} + \cdots + \frac{y^n}{n} + \cdots &= -\ln(1-y) \\ -y - \frac{y^2}{2} - \frac{y^3}{3} - \cdots - \frac{y^n}{n} - \cdots &= \ln(1-y) \end{aligned}$$

Let $y = -x$.

$$\Rightarrow \; x - \frac{x^2}{2} \pm \cdots = \ln(1+x)$$

So

$$\begin{gathered} a_0 = 0 \\ a_k = \tfrac{1}{k} \text{ if } k \text{ is odd} \\ a_k = -\tfrac{1}{k} \text{ if } k \text{ is even} \end{gathered}$$

has $G(x) = \ln(1+x)$ as generating function.

□

Other method:

$$\frac{1}{1-x} = 1 + x + x^2 + \cdots + x^k + \cdots$$

$$\Rightarrow \; \frac{1}{1-(-x)} = 1 + (-x) + (-x)^2 \mp \cdots + (-x)^k + \cdots$$

$$\Rightarrow \; \frac{1}{1-(-x)} = \sum_{k=0}^{\infty} (-x)^k$$

Integrating both sides, we get:

$$\begin{aligned} \ln(1+x) &= \int_0^x \frac{dx}{1+x} \\ &= \int_0^x \sum_{k=0}^{\infty} (-x)^k \, dx \\ = \left(\begin{array}{c}\text{since convergence is uniform} \\ \text{inside closed intervals}\end{array}\right) &= \sum_{k=0}^{\infty} \int_0^x (-x)^k \, dx \\ &= \sum_{k=0}^{\infty} \frac{(-1)^k x^{k+1}}{k+1} \\ &= \sum_{k=1}^{\infty} \frac{(-1)^{k+1} x^k}{k} \\ &= -\sum_{k=1}^{\infty} \frac{(-x)^k}{k} \end{aligned}$$

i.e.

$$\ln(1+x) = 0 + x - \frac{x^2}{2} \pm \cdots + \frac{(-x)^k}{k} \cdots$$

which is what we wanted to show. □

⑥ $G(x) = e^x$, $G^k(x) = e^x$ *for all* k.

$$\Rightarrow \; G^k(0) = e^0 = 1$$

Therefore $a_k = \frac{G^{(k)}(0)}{k!} = \frac{1}{k!}$.
And e^x is the generating function of 1, 1, $\frac{1}{2!}$, $\frac{1}{3!}$, ... In other words:

$$e^x = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \cdots + \frac{1}{k!}x^k + \cdots$$

□

⑦ *Show that* e^{rx} *is the generating function of*

$$1\,,\;\; r\,,\;\; \frac{r^2}{2!}\,,\;\; \frac{r^3}{3!}\,,\;\; \cdots\,,\;\; \frac{r^k}{k!}\,,\;\; \cdots$$

We have:

$$1 + y + \frac{y^2}{2!} + \frac{y^3}{3!} + \cdots + \frac{y^n}{n!} + \cdots = e^y$$

Let $y = rx$.

$$\Rightarrow \; 1 + rx + \frac{(rx)^2}{2!} + \frac{(rx)^3}{3!} + \cdots + \frac{(rx)^n}{n!} + \cdots = e^{rx}$$

$$\Rightarrow \; 1 + rx + \frac{r^2}{2!}x^2 + \frac{r^3}{3!}x^3 + \cdots + \frac{r^n}{n!}x^n + \cdots = e^{rx}$$

So the generating function of

$$1\,,\;\; r\,,\;\; \frac{r^2}{2!}\,,\;\; \frac{r^3}{3!}\,,\;\; \ldots\,,\;\; \frac{r^k}{k!}\,,\;\; \ldots$$

is e^{rx}. □

The seven examples are summarized in the following table:

Some common generating functions

	Sequence a_k, $k = 0, 1, 2, \ldots$	Generating function $G(x)$
①	$a_k = 1$	$\frac{1}{1-x}$
②	$a_k = k + 1$	$\frac{1}{(1-x)^2}$
③	$a_k = \binom{r}{k}$, $r \in \mathbf{R}$	$(1+x)^r$
④	$a_0 = 0$, $a_k = \frac{1}{k}$ if $k \geq 1$	$\ln \frac{1}{1-x}$
⑤	$a_0 = 0$ $a_k = \frac{1}{k}$ if k is odd $a_k = -\frac{1}{k}$ if k is even	$\ln(1+x)$
⑥	$a_k = \frac{1}{k!}$	e^x
⑦	$a_k = \frac{r^k}{k!}$, $r \in \mathbf{R}$	e^{rx}

Operations on Generating Functions

$F(x)$ is the generating function of $a_0, a_1, a_2, \ldots, a_k, \ldots$
and $G(x)$ is the generating function of $b_0, b_1, \ldots, b_k, \ldots$.
In other words:

$$F(x) = \sum a_k x^k \text{ and } G(x) = \sum b_k x^k$$

operation on generating function	k-th element of the corresponding sequence
Multiplication by constant: cF	ca_k
Addition: $F+G$	$a_k + b_k$
Convolution: $F \cdot G$	$\sum_{i=0}^{k} a_i b_{k-i}$
Right shift: xF	a_{k-1}
Partial Sum: $F(x)/(1-x)$	$\sum_{i=0}^{k} a_i$
Differentiation (Index multiply): $x\frac{dF}{dx}(x)$	ka_k
Integration (Index divide): $\int_0^x F(t)dt$	a_{k-1}/k
Scaling: $F(cx)$	$c^k a_k$

Show that the sequence

$$t_k = \sum_{i=0}^{k} a_i b_{k-i} \ , \quad k = 0, 1, 2, \ldots$$

has generating function $F(x)G(x)$. This operation is called convolution (see table).
Solution:

$$t_0 + t_1 x + t_2 x^2 + \cdots + t_k x^k + \cdots$$

$$\begin{aligned}
&= a_0 b_0 + (a_0 b_1 + b_0 a_1)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 \\
&\quad + \cdots + (a_0 b_k + a_1 b_{k-1} + \cdots + a_i b_{k-i} + \cdots + a_k b_0)x^k + \cdots \\
&= a_0(b_0 + b_1 x + b_2 x^2 + \cdots + b_k x^k + \cdots) \\
&\quad + a_1 x(b_0 + b_1 x + b_2 x^2 + \cdots + b_k x^k + \cdots) \\
&\quad \vdots \\
&\quad + a_k x^k (b_0 + b_1 x + b_2 x^2 + \cdots + b_k x^k + \cdots) \\
&\quad \vdots \\
&= (a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k + \cdots)(b_0 + b_1 x + b_2 x^2 + \cdots + b_k x^k \cdots) \\
&= F(x)G(x)
\end{aligned}$$

□

2.5.3 Using generating functions to solve recurrence relations

① Solve

$$a_n = 4a_{n-1}$$
$$a_0 = 1$$

$$\begin{aligned} G(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_nx^n + \cdots \\ 4G(x) &= 4a_0 + 4a_1x + 4a_2x^2 + \cdots + 4a_nx^n + \cdots \end{aligned}$$

$$\begin{aligned} \Rightarrow\ G(x) - 4xG(x) &= a_0 + (a_1 - 4a_0)x + (a_2 - 4a_1)x^2 + \\ &\quad + \cdots + (a_n - 4a_{n-1})x^n + \cdots \end{aligned}$$

But $a_0 = 1$ and $a_n - 4a_{n-1} = 0 \quad \forall n.$

$$\Rightarrow\ G(x)[1 - 4x] = 1$$

$$\Rightarrow\ G(x) = \frac{1}{1 - 4x}$$

We know that

$$1 + x + x^2 + x^3 + \cdots + x^n + \cdots = \frac{1}{1 - x}$$

$$\Rightarrow\ \frac{1}{1 - 4x} = \sum_{n=0}^{\infty} (4x)^n = \sum_{n=0}^{\infty} 4^n x^n$$

$$\Rightarrow\ a_n = 4^n \text{ which is the coefficient of the } n^{th} \text{ term.}$$

Solve

$$a_n - 3a_{n-1} + 2a_{n-2} = 2^n$$
$$a_0 = 0\ ,\ \ a_1 = 1$$

$$\begin{aligned} G(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_nx^n + \cdots \\ 3xG(x) &= 3a_0x + 3a_1x^2 + 3a_2x^3 + \cdots + 3a_{n-1}x^n + \cdots \\ 2x^2G(x) &= 2a_0x^2 + 2a_1x^3 + 2a_2x^4 + \cdots + 2a_{n-2}x^n + \cdots \end{aligned}$$

$$\Rightarrow\ G(x) - 3xG(x) + 2x^2G(x)$$

$$= a_0 + (a_1 - 3a_0)x + (a_2 - 3a_1 + 2a_0)x^2 + \ \cdots + (a_n - 3a_{n-1} + 2a_{n-2})x^n + \cdots$$

But $a_0 = 0$, $a_1 - 3a_0 = 1$, $a_2 - 3a_1 + 2a_0 = 2^2$ and $a_n - 3a_{n-1} + 2a_{n-2} = 2^n$.

$$\Rightarrow\ (1 - 3x + 2x^2)G(x) = x + 2^2x^2 + 2^3x^3 + \cdots + 2^nx^n + \cdots$$

$$= x + \sum_{n=2}^{\infty} 2^n x^n$$

$$= x + \left[\sum_{n=0}^{\infty} (2x)^n - (2x)^0 - (2x)^1\right]$$

$$= x + \left[\frac{1}{1-2x} - 1 - 2x\right]$$

$$\begin{aligned}
(1-3x+2x^2)G(x) &= x + \left(\frac{1-1+2x-2x+4x^2}{1-2x}\right) \\
&= x + \frac{4x^2}{1-2x} \\
(1-x)(1-2x)G(x) &= x + \frac{4x^2}{1-2x}
\end{aligned}$$

$$\Rightarrow\ G(x) = \frac{x}{(1-x)(1-2x)} + \frac{4x^2}{(1-x)(1-2x)^2}$$

Solve

$$\frac{x}{(1-x)(1-2x)} = \frac{A}{1-x} + \frac{B}{1-2x} \text{ to get } A = -1\,,\ \ B = 1$$

and solve

$$\frac{x^2}{(1-x)(1-2x)^2} = \frac{A}{1-x} + \frac{B}{1-2x} + \frac{C}{(1-2x)^2}$$

$$\Rightarrow\ A = 1\,,\ \ B = -\frac{3}{2}\,,\ \ C = \frac{1}{2}$$

Therefore

$$G(x) = \left(\frac{1}{1-2x} - \frac{1}{1-x}\right) + 4\left[\frac{1}{1-x} - \frac{3}{2(1-2x)} + \frac{\frac{1}{2}}{(1-2x)^2}\right]$$

$$\begin{aligned}
\frac{1}{1-x} &= \sum_{i=0}^{\infty} x^i \ \Rightarrow\ a_n = 1 \\
\frac{1}{1-2x} &= \sum_{i=0}^{\infty} (2x)^i \ \Rightarrow\ a_n = 2^n \\
& \quad \text{differentiating both sides} \\
\frac{2}{(1-2x)^2} &= \sum_{i=0}^{\infty} (i+1)2^{i+1}x^i \\
\Rightarrow\ \frac{1}{(1-2x)^2} &= \sum_{i=0}^{\infty} (i+1)2^i x^i
\end{aligned}$$

So

$$G(x) = \frac{1}{1-2x} - \frac{1}{1-x} + 4\left(\frac{1}{1-x} - \frac{\frac{3}{2}}{1-2x} + \frac{\frac{1}{2}}{(1-2x)^2}\right)$$

is the generating function of a sequence whose n^{th} term is:

$$\begin{aligned}
a_n &= 2^n - 1 + 4\left(1 - \frac{3}{2}(2^n) + \frac{1}{2}(n+1)2^n\right) \\
&= (1 - 6 + 2n + 2)2^n - 1 + 4
\end{aligned}$$

So

$$a_n = (2n-3)2^n + 3$$

is the solution of

$$a_n - 3a_{n-1} + 2a_{n-2} = 2^n \text{ with } a_0 = 0\,,\ \ a_1 = 1$$

□

② Use generating functions to solve

$$\begin{aligned} a_n &= 4a_{n-2} \\ a_0 &= 0 \\ a_1 &= 1 \end{aligned}$$

$$\begin{aligned} G(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \cdots \\ 4x^2 G(x) &= 4a_0 x^2 + 4a_1 x^3 + 4a_2 x^4 + \\ & \quad \cdots + 4a_n x^{n+2} + \cdots \\ (1-4x^2)G(x) &= a_0 + a_1 x + (a_2 - 4a_0)x^2 + \\ & \quad \cdots + (a_n - 4a_{n-2})x^n + \cdots \\ (1-4x^2)G(x) &= a_0 + a_1 x + \sum_{i=2}^{\infty}(a_i - 4a_{i-2})x^i \end{aligned}$$

But $a_0 = 0$, $a_1 = 1$ and $a_i - 4a_{i-2} = 0$.

$$G(x) = \frac{x}{1-4x^2} = \frac{x}{(1-2x)(1+2x)}$$

$$\frac{x}{(1-2x)(1+2x)} = \frac{A}{1-2x} + \frac{B}{1+2x}$$

$$\Rightarrow\ A = \frac{1}{4}\,,\ \ B = -\frac{1}{4}$$

$$G(x) = \frac{\frac{1}{4}}{1-2x} + \frac{-\frac{1}{4}}{1+2x}$$

We know

$$\frac{1}{1-y} = 1 + y + y^2 + y^3 + \cdots + y^n + \cdots$$

Hence with $y = 2x$ we get:

$$\frac{1}{1-2x} = 1 + 2x + 2^2x^2 + 2^3x^3 + \cdots + 2^n x^n + \cdots$$

and with $y = -2x$ we obtain:

$$\frac{1}{1-(-2x)} = 1 + (-2)x + (-2)^2x^2 + (-2)^3x^3 + \cdots + (-2)^n x^n + \cdots$$

$$\Rightarrow\ a_n = \frac{1}{4}2^n - \frac{1}{4}(-1)^n 2^n$$

Therefore

$$\text{for } n \text{ even} \ \ \Rightarrow\ a_n = 0$$

$$\text{for } n \text{ odd} \ \ \Rightarrow\ a_n = 2^{n-1}$$

□

③ Find the generating function of the sequence of cubes:

$$0, 1, 8, 27, 64, \ldots, n^3, \ldots$$

Solution:

$$1 + x + x^2 + x^3 + \ldots + x^n + \ldots = \frac{1}{1-x}$$

differentiate both sides:

$$1 + 2x + 3x^2 + 4x^3 + \ldots + nx^{n-1} + \ldots = \frac{1}{(1-x)^2}$$

multiply by x:

$$x + 2x^2 + 3x^3 + 4x^4 + \ldots nx^n + \ldots = \frac{x}{(1-x)^2}$$

differentiate both sides:

$$1 + 2^2x + 3^2x^2 + 4^2x^3 + \ldots n^2x^{n-1} + \ldots = \frac{1+x}{(1-x)^3}$$

multiply by x:

$$x + 2^2x^2 + 3^2x^3 + 4^2x^4 + \ldots n^2x^n + \ldots = \frac{x+x^2}{(1-x)^3}$$

differentiate both sides:

$$1 + 2^3x + 3^3x^2 + 4^3x^3 + \ldots n^3x^{n-1} + \ldots = \frac{x^2+4x+1}{(1-x)^4}$$

multiply by x:

$$0 + x + 2^3x^2 + 3^3x^3 + 4^3x^4 + \ldots n^3x^n + \ldots = \frac{x(x^2+4x+1)}{(1-x)^4}$$

So

$$G(x) = \frac{x(x^2+4x+1)}{(1-x)^4}$$

is the generating function of $0, 1, 8, 27, 64, \ldots, n^3, \ldots$.

□

Chapter 3

Greedy Algorithms

It is the most straightforward technique. It is easy to design and easy to implement. While solving combinatorial optimization problems, we build feasible solutions. In general, a sequence of decisions has to be made.

Greedy algorithms are shortsighted. Every time we make one of these decisions, we look at all the current information and make the least costly choice possible.

This choice is made without any regards to the effect that it might have on later choices. The final feasible solution may or may not be the Optimal. It depends on the problem under investigation.

3.1 Combinatorial Optimization Problems

To introduce (and denote) combinatorial optimization problems, we make use of the following model by Stinson.

Problem instance Some given combinatorial optimization problem

Constraints Some constraints that must be met

Feasible solution A solution of the problem for which the constraints are met

Objective function A function that assigns a non-negative value to each feasible solution

Optimal solution Optimum feasible solution. That is, the feasible solution which maximizes (if profit) or minimizes (if cost) the objective function.

In the next sections, we study problems that can be solved by designing greedy algorithms.

3.2 The 0–1 knapsack problem

We make use of Stinson's framework, discussed in the previous section, to introduce the $0-1$ knapsack problem.

Problem instance

objects	1	2	...	i	...	n
weights	w_1	w_2	...	w_i	...	w_n
profits	p_1	p_2	...	p_i	...	p_n

KNAPSACK with capacity M

Feasible solution

a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$ with associated weight $W(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i \leq M$.

Objective function

Profit $P(\mathbf{x})$ for $\mathbf{x}$ feasible such that $P(\mathbf{x}) = \sum_{i=1}^{n} x_i p_i$.

Optimal solution

Maximum-profit feasible solution □

The popularity of the 0-1 KNAPSACK is due to:

- it is the simplest integer linear programming problem
- it is often considered as a subproblem to complex ones
- it is a very practical problem
- it is easy to comprehend but difficult to solve

A closely related, but easier to solve problem, is the fractional knapsack problem.

3.2.1 The fractional knapsack problem

In the fractional KNAPSACK problem (or rational KNAPSACK problem), the setup is the same, except that we can take fractions of an object. In other words, a feasible solution is given by a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in [0, 1]$ (rather than $x_i \in \{0, 1\}$).

3.2.2 The greedy algorithm for the knapsack problem

i.) Rename the objects such that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_i}{w_i} \geq \cdots \geq \frac{p_n}{w_n}$$

ii.) Place objects $1, 2, \ldots$ updating the cumulative weight at each stage.

iii.) Stop at the j^{th} object if either

a) there is no room for object j:

i) for 0-1 KNAPSACK: $\sum_{i=1}^{j} w_i > M$ and so reject object j

ii) for fractional KNAPSACK: allow fraction of object j to fill up knapsack

or

b) all objects are chosen (i.e. $j = n$, and $w_1 + w_2 + \cdots + w_n \leq M$).

EXAMPLE:

objects	1	2	3	4	5
weight	11	12	8	7	9
profit	23	24	15	13	16

$M = 26$

Note: The objects are in decreasing order (with respect to $\frac{p_i}{w_i}$)
Solving the rational knapsack problem, we get the solution vector

$$\mathbf{x} = (1, 1, \frac{3}{8}, 0, 0)$$

with a total profit of $23 + 24 + \frac{3}{8}(15) = \frac{421}{8}$.

If we use the greedy technique for the 0-1 KNAPSACK problem, we obtain a solution vector

$$\mathbf{x} = (1, 1, 0, 0, 0)$$

with profit $P(\mathbf{x}) = 47$. But $\mathbf{x}' = (1, 0, 1, 1, 0)$ gives a profit of $23 + 15 + 13 = 51$, so the greedy algorithm doesn't give the optimal solution. □

3.2.3 Optimality of the greedy algorithm

Theorem 3.2.1 *The greedy algorithm for the* KNAPSACK *problem always yields the optimal solution when applied to the fractional* KNAPSACK *problem.*

Proof: Suppose that the greedy algorithm generates the solution $\mathbf{x} = (x_1, \ldots, x_n)$, with profit $P = \sum p_i x_i$. Let $\mathbf{y} = (y_1, \ldots, y_n)$ be an optimal solution, with profit $Q = \sum p_i y_i$. Since $\mathbf{y}$ is optimal, we must have $P \leq Q$; we want to prove that $P = Q$.

First observe that if $\sum w_i \leq M$, then the optimal solution is $(1, 1, \ldots, 1)$, and the greedy algorithm generates this solution (no matter what order the objects are considered in). So, we can assume that $\sum w_i > M$. Also, we must have $\sum w_i y_i = M$, or $\mathbf{y}$ cannot be optimal. What we do is show how to transform Q into P, maintaining optimality.

By the way in which the greedy algorithm works, there must be an index k , $1 \leq k \leq n$, such that $x_i = 1$ if $i < k$, $0 \leq x_k < 1$, and $x_i = 0$ if $i > k$. So $\mathbf{x}$ has the form $(1, \ldots, 1, x_k, 0, \ldots, 0)$. Also, observe that we must have $\sum w_i x_i = M$. (This is also a consequence of the way the greedy algorithm works).

Suppose j is the first position in which $x_j \neq y_j$. We distinguish 3 cases:

- Case 1: $j > k$
- Case 2: $j = k$
- Case 3: $j < k$

Case 1: Here $\mathbf{x} = (1, \ldots, 1, x_k, 0, \ldots, 0)$ and $\mathbf{y} = (1, \ldots, x_k, \ldots, y_j, \ldots)$, where $x_k \geq 0$ and $y_j > 0$. This case is in fact impossible, as $\mathbf{y}$ exceeds the knapsack capacity M. We see this as follows:

$$\begin{aligned} \sum_{i=1}^{n} w_i y_i &= \sum_{i=1}^{k} w_i y_i + \sum_{i=k+1}^{n} w_i y_i \\ &= \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i y_i \\ &= M + \sum_{i=k+1}^{n} w_i y_i \\ &> M \end{aligned}$$

since $y_j > 0$.

Case 2: Here $\mathbf{x} = (1, \ldots, 1, x_k, 0, \ldots, 0)$, and $\mathbf{y} = (1, \ldots, 1, y_k, \ldots)$, where $x_k, y_k \geq 0$. First, we must have $y_k < x_k$; for if $y_k > x_k$, then $\mathbf{y}$ exceeds the capacity M by an argument similar to that in Case 1. Hence, $x_k > y_k \geq 0$. Now, since

$$\sum_{i=1}^{n} w_i y_i = \sum_{i=1}^{n} w_i x_i = M$$

and

$$\sum_{i=1}^{k-1} w_i y_i = \sum_{i=1}^{k-1} w_i x_i$$

we have

$$w_k x_k = \sum_{i=k}^{n} w_i y_i$$

or

$$w_k(x_k - y_k) = \sum_{i=k+1}^{n} w_i y_i$$

We also have:

$$\begin{aligned}
Q - P &= \sum_{i=k+1}^{n} p_i y_i - p_k(x_k - y_k) \\
&= \sum_{i=k+1}^{n} \frac{p_i}{w_i} w_i y_i - \frac{p_k}{w_k} w_k(x_k - y_k) \\
&\leq \sum_{i=k+1}^{n} \frac{p_k}{w_k} w_i y_i - \frac{p_k}{w_k} w_k(x_k - y_k) \\
&= \frac{p_k}{w_k} \left(\sum_{i=k+1}^{n} w_i y_i - w_k(x_k - y_k) \right) \\
&= 0
\end{aligned}$$

Hence, $Q \leq P$. Since we also have $Q \geq P$, we must have $Q = P$. Therefore, $\mathbf{x}$ is also an optimal solution.

Case 3: This is the general case; we have $\mathbf{x} = (1, \ldots, 1, x_k, 0, \ldots, 0)$ and $\mathbf{y} = (1, \ldots, 1, y_j, \ldots)$, where $0 \leq y_j < x_j = 1$ (since $j < k$). First, we have

$$\sum_{i=1}^{n} w_i y_i = \sum_{i=1}^{n} w_i x_i$$

or

$$w_j(1 - y_j) + \sum_{i=j+1}^{n} w_i x_i = \sum_{i=j+1}^{n} w_i y_i$$

since $x_j = 1$. Hence,

$$w_j(1 - y_j) \leq \sum_{i=j+1}^{n} w_i y_i$$

This is excess weight available for y_j. We construct a new feasible solution. Choose u_i , $j+1 \leq i \leq n$ such that $0 \leq u_i \leq y_i$ for $j+1 \leq i \leq n$; and so that

$$w_j(1 - y_j) = \sum_{i=j+1}^{n} w_i u_i$$

Now, define $\mathbf{z} = (z_1, \ldots, z_n)$, by

$$
\begin{aligned}
z_i &= y_i \text{ if } 1 \leq i \leq j-1 \\
z_i &= 1 \text{ , if } i = j \\
z_i &= y_i - u_i \text{ , if } j+1 \leq i \leq n
\end{aligned}
$$

First, we observe that $0 \leq z_i \leq 1$ for $1 \leq i \leq n$. Also, we calculate

$$
\sum_{i=1}^{n} w_i z_i = \sum_{i=1}^{j-1} w_i y_i + w_j + \sum_{i=j+1}^{n} w_i(y_i - u_i)
$$

$$
= \sum_{i=1}^{j-1} w_i y_i + w_j - w_j(1 - y_j) + \sum_{i=j+1}^{n} w_i y_i
$$

$$
= \sum_{i=1}^{n} w_i y_i
$$

$$
= M
$$

so $\mathbf{z}$ is a feasible solution.

Now the profit for $\mathbf{z}$ is

$$
R = \sum_{i=1}^{n} p_i z_i
$$

We calculate

$$
\begin{aligned}
R - Q &= \sum_{i=1}^{n} p_i(z_i - y_i) \\
&= p_j(1 - y_j) - \sum_{i=j+1}^{n} p_i u_i \\
&= \frac{p_j}{w_j} w_j(1 - y_j) - \sum_{i=j+1}^{n} \frac{p_i}{w_i} w_i u_i \\
&\geq \frac{p_j}{w_j} w_j(1 - y_j) - \sum_{i=j+1}^{n} \frac{p_j}{w_j} w_i u_i \\
&= \frac{p_j}{w_j} \left(w_j(1 - y_j) - \sum_{i=j+1}^{n} w_i u_i \right) \\
&= 0
\end{aligned}
$$

Hence, $R \geq Q$. But $R \leq Q$, since $\mathbf{y}$ is optimal. It follows that $R = Q$ and $\mathbf{z}$ is also optimal. Now $z_i = x_i$ for $1 \leq i \leq j$, so $\mathbf{z}$ agrees with $\mathbf{x}$ in one more coordinate (from the beginning) than $\mathbf{y}$ does. So, we repeat this process as often as necessary, until $\mathbf{y}$ is transformed into $\mathbf{x}$. Since all solutions produced are optimal, $\mathbf{x}$ must also be an optimal solution. This completes the proof.

□

In the next section we study the well-known problem of Huffman codes.

3.3 Huffman codes

Huffman codes are a widely used and very efficient technique for compressing data. They were designed by David Huffman in the early 1950's. Huffman codes are very popular because they

- construct optimal **prefix** codes (a.k.a **prefix–free** or **instantaneous** codes). They are codes in which no codeword is a prefix to a longer codeword
- are very simple
- are often used as part of more elaborate data compression techniques. For example, they are used in data compression standards for fax, speech, image (e.g. JPEG) and video.

Huffman's greedy algorithm uses a table of the frequencies of occurrence of each symbol to build up an optimal way of representing each symbol as a binary string. An optimal code is always represented by a full binary tree (a binary tree, in which every nonleaf node has two children). Hence, given a tree T corresponding to a prefix code, it is always a simple matter to compute the number of bits required to encode a file:
For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree (that is, $d_T(c)$ is the length of the codeword for character c). Thus the number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the **cost** of tree T.

EXAMPLE:
Suppose we have a 100,000 character file with the following frequencies (the file has only 6 different symbols: $\{a, b, c, d, e, f\}$; see the table).

	a	b	c	d	e	f
frequencies in thousands	45	13	12	16	9	5
fixed-length codeword	000	001	010	011	100	101
variable length codeword	0	100	101	111	1101	1100

Figure 3.1 shows the two trees corresponding to the coding schemes in the above table.

The cost of tree T_1 is 300000 bits ($3 \cdot 100000$), but for tree T_2 only

$$[45 \cdot (1) + 13 \cdot (3) + 12 \cdot (3) + 16 \cdot (3) + 9 \cdot (4) + 5 \cdot (4)]1000 = 224,000$$

bits are required.

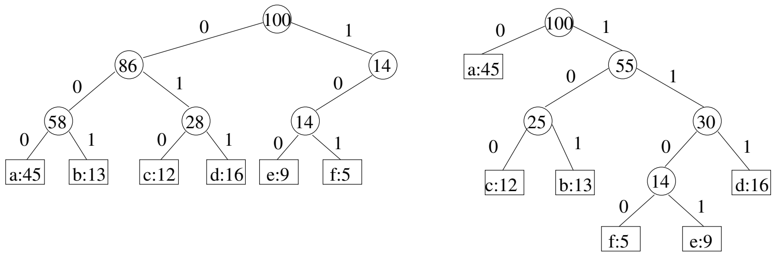


Figure 3.1: The left tree representeds the fixed-length code, the right tree corresponds to the optimal prefix code. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree.

3.3.1 Constructing Huffman Codes

Huffman's greedy algorithm HUFFMAN(C) constructs an optimal prefix code called **Huffman Code** by building the corresponding binary tree T in a bottom-up manner. In the following algorithm, we assume that C is an alphabet of n characters and that for each $c \in C$ the frequency $f[c]$ is given.

HUFFMAN*(C)*

---

begin
  $n \leftarrow |C|$
  $Q \leftarrow C$ *initialize priority queue*
  for $i \leftarrow 1$ **to** $n - 1$
    do $z \leftarrow$ ALLOCATE-NODE()
      $x \leftarrow left[z] \leftarrow$ EXTRACT-MIN*(Q)*
      $y \leftarrow right[z] \leftarrow$ EXTRACT-MIN*(Q)*
      $f[z] \leftarrow f[x] + f[y]$
      INSERT*(Q, z)*
  return EXTRACT-MIN*(Q)*
end

In procedure HUFFMAN*(C)*, the priority queue Q is implemented as a binary heap (see below). Therefore, the initialization of Q can be done in $O(n)$ and each (heap-)operation requires $O(\log_2 n)$. Since the loop has $n - 1$ iterations, it is performed in $O(n \log_2 n)$, which is the running time of HUFFMAN*(C)*.

Remark:

cost of a single merger	=	sum of frequencies of the two items being merged

So merging the two items with lowest frequencies is a **greedy** step.

We take a short break from the study of greedy algorithms to introduce priority queues that can be used as data structures for the implementation of many algorithms.

3.3.2 Priority queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. The operations of a priority queue are:

- INSERT*(S, x)* put element x in S
- MAXIMUM*(S)* returns the element of S with largest key

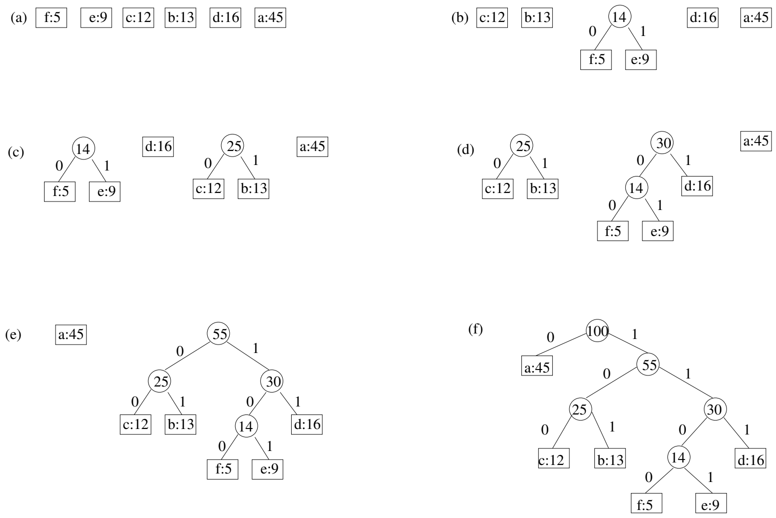


Figure 3.2: The steps of Huffman's algorithm for the frequencies given in the above table. Each step shows the contents of the priority queue sorted into increasing order by frequency. At each step the two trees with lowest frequency are merged.

- EXTRACT-MAX*(S)* removes and returns the element of S with largest key

Note that MAXIMUM and EXTRACT-MAX are often replaced by MINIMUM and EXTRACT-MIN, as in the previous section.

We can implement a priority queue by using **heaps**. The operations on heaps are:

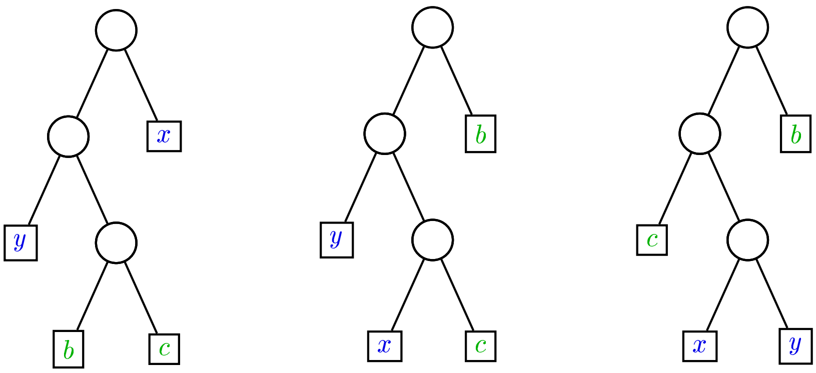
1. HEAP-MAXIMUM, which returns the value $A[1]$ in the heap. Recall that the first entry of the heap A contains the largest element. **Running time:** $\Theta(1)$

2. HEAP-EXTRACT-MAX, which removes and returns the content of $A[1]$. So it makes use of HEAPIFY.

```
HEAP-EXTRACT-MAX(A)
  begin
    if heap-size (A) < 1
      then "heap underflow"
    max ← A[1]
    A[1] ← A[heap-size (A)]
    heap-size (A) ← heap-size (A) − 1
    HEAPIFY(A,1)
    return max
  end
```

Running time: $O(\log_2 n)$ which, recall, is the amount of work needed to heapify.

3. HEAP-INSERT, which inserts a node into heap A by first adding a new leaf then traversing the tree (only one path), bottom up, to find the proper place for the new element.



T
Optimal Tree

T'
Interchange x and b

T''
Interchange y and c

Figure 3.3: The trees T, T' and T''

```
HEAP-INSERT(A, key)
  begin
    heap-size(A) ← heap-size(A)+1
    i ← heap-size (A)
    while i > 1 and A[parent(i)] < key
      do A[i] ← A[parent(i)]
      i ← parent(i)
    A[i] ← key
  end
```

Running time: $O(\log_2 n)$ which is the cost of going up the tree.

We present two ingredients that are generally present in problems that can be generally solved by greedy algorithms.

3.3.3 Elements of the greedy strategy

Two ingredients that are present in most problems that lend themselves to a greedy strategy:

Greedy choice property
: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Optimal substructure
: an optimal solution to the problem contains within it optimal solutions to subproblems.

We now use this principle to prove the correctness of Huffman's algorithm.

3.3.4 Correctness of Huffman's algorithm

To show that Huffman's procedure yields a prefix code with minimum cost, we have to show:

① **Greedy-choice property**
Show that merging the lowest two frequency characters does not cause the tree to become non-optimal. We show that any optimal tree can be reorganized such that the two characters (with lowest frequency) have the same parent.

② **Optimal substructure**
After making an optimal first choice, the problem is reduced to finding a Huffman code for a smaller alphabet.

Proof:

① **Greedy-choice property**
Let x and y be two characters in C having lowest frequencies. We can always construct an optimal prefix code for C in which the encodings for x and y have the same length and differ only in the last bit. We assume that $f(x) \leq f(y)$. Suppose that T is an optimal tree with $f(b) \leq f(c)$ where b and c are siblings of maximum depth. We thus have: $f(x) \leq f(b)$ and $f(y) \leq f(c)$. We construct a tree T' by interchanging x and b and then we exchange the positions in T' of y and c to produce a tree T'' (see Figure 3.3). We'll show that T and T'' have equal costs:

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\
&= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x) \\
&= \underbrace{(f(b) - f(x))}_{\substack{\geq 0 \\ x \text{ has lowest} \\ \text{frequency} \\ (\text{as } x \text{ and } y \\ \text{are of lowest} \\ \text{frequency})}} \cdot \underbrace{(d_T(b) - d_T(x))}_{\substack{\geq 0 \\ b \text{ is a leaf} \\ \text{of maximum} \\ \text{depth in T}}} \geq 0
\end{aligned}$$

Thus $B(T) - B(T') \geq 0$. But $B(T) \leq B(T')$ since $B(T)$ is the cost of the optimal tree T.
Therefore $B(T) = B(T')$.

Similarly, we can show that

$$B(T') = B(T'')$$

and thus

$$B(T) = B(T'')$$

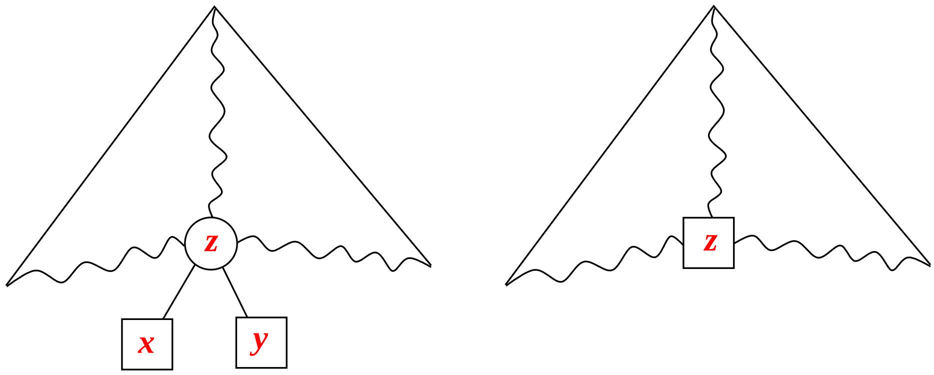
□

② **Optimal substructure property**
Show that T', the tree obtained from T by removing the two leaves x and y (see Figure 3.4), represents an optimal prefix code for alphabet $C' = C - \{x, y\} \cup \{z\}$. For each $c \in C - \{x, y\} :\ d_T(c) = d_{T'}(c)$. Consequently

$$f(c)d_T(c) = f(c)d_T(c) \qquad \forall\ c \in C - \{x, y\}$$

Note: $d_T(x) = d_T(y) = d_{T'}(z) + 1$.
So

$$\begin{aligned}
f(x)d_T(x) + f(y)d_T(x) &= (f(x) + f(y))(d_{T'}(z) + 1) \\
&= f(z)d_{T'}(z) + (f(x) + f(y)) \\
\Rightarrow B(T) &= B(T') + f(x) + f(y)
\end{aligned}$$



T: full binary tree representing optimal prefix code over C

$T' = T \setminus \{x, y\}$ where z has frequency $f(z) = f(x) + f(y)$

Figure 3.4: Optimal substructure property

Suppose T' is not optimal for C'. Then there exists T'' whose leaves are characters in C' such that:

$$B(T'') < B(T')$$

Since z is treated as a character in C', it appears as a leaf in T'. Now consider the tree T''' that is obtained by attaching x and y to T'' by making them children of z. Note that T''' represents a prefix code for C.

$$\begin{aligned} B(T''') &= B(T'') + f(x) + f(y) \\ &< B(T') + F(x) + f(y) \text{ since } B(T'') < B(T') \\ &= B(T) \end{aligned}$$

So we obtained $B(T''') < B(T)$, contradicting the optimality of T.
Therefore T' is optimal for C'.

3.4 The general greedy procedure

Most problems amenable to greedy techniques have algorithms that are very similar to GREEDY*(C)*.

GREEDY*(C)*	*C is a set of candidates*
begin	
$S \leftarrow \emptyset$	*we construct the solution in the set S*
while $(C \neq \emptyset)$ **and** $\neg$SOLUTION(S)	
$x \leftarrow$ SELECT(C)	
$C \leftarrow C - \{x\}$	
if FEASIBLE$((S) \cup \{x\})$	
then $S \leftarrow S \cup \{x\}$	
if SOLUTION(S)	
then return S	
else return "no solution"	
end	

Remark: The SELECTION() procedure is related to the objective function.

Next we consider scheduling problems.

3.5 Minimum tardy task problem

We start by giving an EXAMPLE:

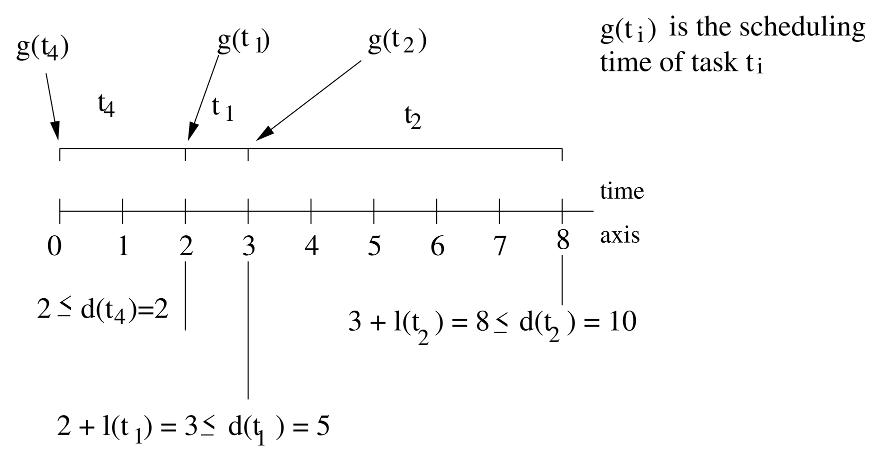
$$T = \{t_1, t_2, t_3, t_4\}$$

is a set of tasks. We denote by:

- $l(t)$: length it takes to accomplish task t
- $w(t)$: weight of task t (penalty)
- $d(t)$: deadline by which task t should be done

	t_1	t_2	t_3	t_4
$l(t_i)$	1	5	3	2
$d(t_i)$	5	10	5	2
$w(t_i)$	3	2	5	4

Consider $T_1 = \{t_1, t_2, t_3\}$. We want to schedule the tasks of T_1 on a "uniprocessor". Since $d(t_4) < d(t_1) < d(t_2)$ it makes sense to schedule t_4 first, followed by t_1 and then t_2.



So $T_1 = \{t_1, t_2, t_4\}$ is a feasible solution. Let W denote the sum of the weights of the unscheduled tasks (i.e. the tardy task weight).

$$W = \sum_{t \in (T - T_1)} w(t)$$

In our example $T - T_1 = \{t_3\}$ and therefore $W = w(t_3) = 5$. □

Note that at most one task is executed at a time, and each scheduled task finishes before its deadline.

The minimum tardy task problem is a minimization problem that has a maximization counterpart. Just replace the weights by profits and now the objective function is the sum of the profits of the scheduled tasks.

Remark:

$S \subseteq T$ is feasible $\Leftrightarrow$ the tasks in S can be scheduled in increasing order by deadline without violating any deadline

In other words, if $|T| = n$ and $S \subseteq T$ with $|S| = k$, we only have to check **one** of the $k!$ possible schedules for the feasibility of S.

3.6 Scheduling with Deadlines

Once again we use Stinson's framework to introduce the scheduling with deadlines problem.

Problem instance

tasks	1	2	...	i	...	n in T	
lengths	l_1	l_2	...	l_i	...	l_n	$l_i > 0$
deadlines	d_1	d_2	...	d_i	...	d_n	$d_i > 0$
profits	p_1	p_2	...	p_i	...	p_n	$p_i > 0$

Feasible solution

any schedule of $S \subseteq T$ such that:

$$g : S \to Z^+ \cup \{0\}$$

where

i) $g(i) < g(j) \Rightarrow g(i) + l_i \leq g(j)$

ii) $g(i) + l_i \leq d_i$

Objective function

a profit $P = \sum\limits_{i \in S} p_i$

Optimal solution

maximum-profit schedule S

EXAMPLE:

Problem instance

tasks	1	2	3	4	5	6	7	8
length	2	4	1	7	4	3	5	2
deadlines	3	5	6	8	10	15	16	20
profits	15	20	16	19	10	25	17	18

- $S = \{1, 3, 5, 6\}$
  S is feasible with $P = 66$
- $S' = \{2, 3, 4, 6, 8\} \Rightarrow g(4) + l_4 = 5 + 7$ and $d_4 = 8$ i.e. $g(4) + l_4 > d_4$
  Therefore S' is infeasible. □

Remark: If $l_1 = l_2 = l_3 = \ldots = l_n = 1$ then the problem can be solved in polynomial time. An obvious greedy algorithm for constructing S is to add at each step the job with the highest value of p_i among those not yet considered, provided that the chosen set of jobs remains feasible.

```
MIN-TARDY-TASK(T)
  begin
    rename tasks such that p_1 ≥ p_2 ≥ ... ≥ p_n
    S ← ∅
    for i = 1 to n
      if S ∪ {t_i} is feasible
      then S ← S ∪ {t_i}
    return P = ∑_{t∈S} p(t)
  end
```

Note that the tasks in $S \cup \{t_i\}$ are first sorted by increasing deadline and then scheduled . Obviously this can be a very inefficient algorithm unless we use appropriate data structures in the implementation.

Remark: MIN-TARDY-TASK*(T)* gives the optimal solution when $l_1 = l_2 = \ldots = l_n = 1$.
The next topic we tackle is the problem of finding minimum spanning trees (MST).

3.7 Minimum Spanning Trees (MST)

An acyclic graph is a graph that has no cycles. If *any* 2 distinct nodes have an edge between them then the graph is complete. A tree is an acyclic, connected graph. A spanning tree of a graph G is a tree that is a subgraph of G and contains every vertex of G. A spanning tree of minimal total weight, i.e. the sum of the weights of all edges is minimum, is called a minimum spanning tree (MST). We study two algorithms for finding minimum spanning trees: Kruskal's and Prim's.

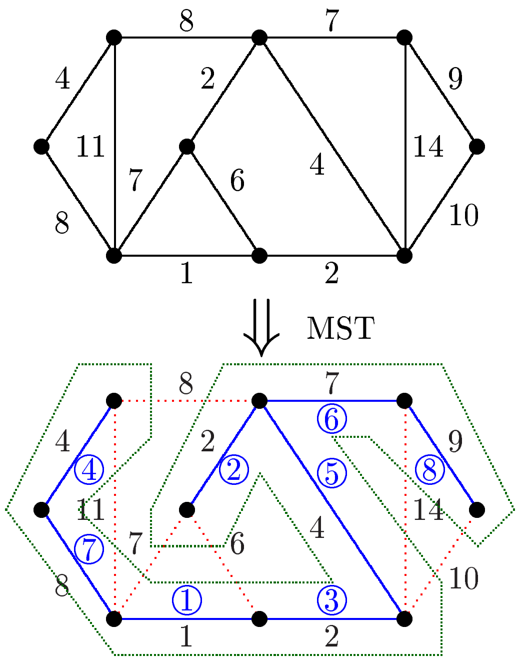
3.7.1 Kruskal's MST Algorithm

Given a weighted graph, at each step Kruskal's algorithm chooses the lowest-weighted remaining edge form anywhere in the graph, discarding any edge that would form a cycle with those already chosen. At any time the edges chosen so far will form a forest but not necessarily one tree.

Kruskal's procedure uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices of a tree. The operations needed are:

- FIND-SET*(u)*: returns a representative element from the set that contains u.
- UNION*(u, v)*: combines the two trees that contain u and v.

EXAMPLE: Given the following graph, Kruskal's method will construct the MST depicted in the second graph. The tree is in bold and the order in which the edges are chosen is indicated by circled numbers.



```
Kruskal
begin
  A ← ∅
  for each vertex v ∈ V                    creates |V| trees each one
    do Make-Set(v)                         containing one vertex
  sort edges of E by
  nondecreasing weight w
  for each edge (u,v) ∈ E, in order
    by nondecreasing weight
    do if Find-Set(u)≠ Find-Set(v)         u and v are from different trees
         then A ← A ∪ {(u,v)}
              Union(u, v)
  return A
end
```

The running time of Kruskal's algorithm depends on the implementation of the disjoint-set data structure:

- initialization takes $O(|V|)$
- sorting the edges takes $O(|E| \log_2 |E|)$
- there are $|E|$ iterations of the loop (i.e. on disjoint-set data structure). If the disjoint-set forest data structure (see next subsection) with the union by rank and path compression procedures is used, then the work performed for the loop is $O(|E|\alpha(|E|, |V|))$ where α is the functional inverse of ackermann's function (see [CLR90, Section 22.4]).

  Since $\alpha(|E|, |V|) = O(\log_2 |E|)$, the total running time of Kruskal's algorithm is $O(|E| \log_2 |E|)$.

In the next section we give a brief overview of the disjoint-set data structure that can be used to implement Kruskal's algorithm.

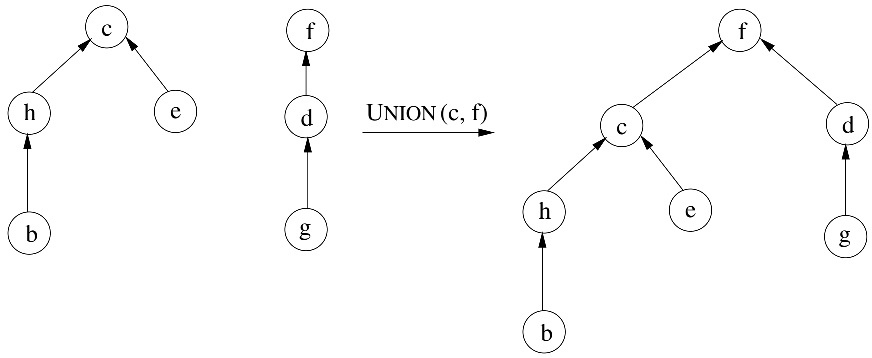
3.7.2 Disjoint-set data structure

A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a representative, which is just some member of the set. We maintain the sets as rooted trees. The operations required are:

- Make-Set*(x)*: Creates a new set, whose only member (and thus representative) is pointed to by x. Note that x is not in any other set since they are disjoint sets.
- Union*(x, y)*: Unites the sets that contain x and y to form a new set. Note that the 2 sets were disjoint prior to the call to Union*(x, y)*.
- Find-Set*(x)*: Returns a pointer to the representative of the (unique) set containing x. If the sets are represented by trees, then Find-Set*(x)* consists in going up the tree to the root node. Visited nodes are called "find path".

Disjoint-set forests

Each member points to its parent. The root contains the representative and is its own parent.



To efficiently implement the operations of disjoint-set forests, we make use of two heuristics:

1. **Union by rank**: Make the tree with fewer nodes point to the root of the tree with more nodes. With each node, we associate a rank. $Rank[x]$ is an upper bound on the number of edges in the longest path between x and a descendant leaf (i.e., on the depth of the subtree rooted at x).

2. **Path compression**: Make each node visited during FIND-SET (i.e. nodes that belong to the "find path") point directly to the root. This is done to speed up future FIND-SET operations. Note that path compression does not alter the ranks of the nodes.

We now introduce the procedures associated with the disjoint-set data structure.

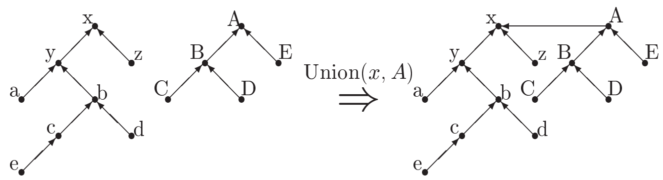
```
MAKE-SET(x)
  begin
    parent[x] ← x
    rank[x] ← 0
  end
```

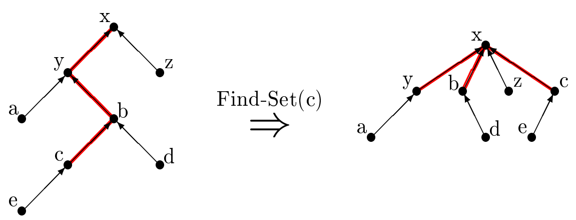
```
UNION(x, y)
  begin
    LINK(FIND-SET(x),FIND-SET(y))
  end
```

```
LINK(x, y)
  begin
    if rank[x] > rank[y]
      then parent[y] ← x
      else parent[x] ← y
           if rank[x] = rank[y]
             then rank[y] ← rank[y] + 1
  end
```

```
FIND-SET(x)                                   makes 2 passes
                                              for path compression
  begin
    if x ≠ parent[x]
      then parent[x] ← FIND-SET(parent[x])
    return parent[x]
  end
```

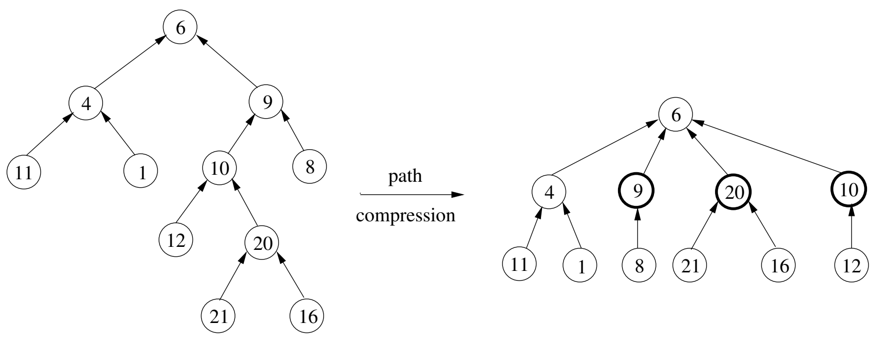
Example for Union and Find-Set





□

Example for path-compression



□

Figure 3.5: Find-Set(20): nodes in the find-path (i.e. $9, 10, 20$) now point directly to the root. The remaining nodes are not touched.

We now study the second MST algorithm mentioned in the introduction of this subsection.

3.7.3 Prim's minimum spanning tree algorithm

Prim's algorithm selects an arbitrary starting vertex, then branches out from the tree constructed so far by choosing a new vertex and edge at each iteration. The choice is done **greedily**. During the course of the algorithm, the vertices are divided into three disjoint sets:

- red vertices: nodes in the tree constructed so far
- blue vertices: nodes not in the tree but adjacent to some red node
- black vertices: nodes that are neither red nor blue

Convention: An edge between a red node and a blue node is a violet edge. An edge in the tree is colored red.

PrimColor MST algorithm

1. color an arbitrary node red, the rest blue and black
2. **while not** (all nodes red)
   do
   select the shortest violet edge;
   color it and its blue endpoint red;
   update colors of nodes and edges
   end

Remark: The current red nodes and red edges form a subtree of a minimum spanning tree.

With efficient implementation that involves "good" data structures the above algorithm can have as running time

$$O(|E| + |V| \log_2 |V|)$$

(which is a big improvement over the $O(|V|^3)$ behavior exhibited by the above algorithm). We now give a more formal study of Prim's algorithm.

Prim's MST algorithm

Given a weighted graph, at each step Prim's algorithm chooses the lowest-weighted edge among the ones that have an end-point in the tree constructed so far.

Recall: The algorithm selects an arbitrary starting vertex, then branches out from the tree constructed so far by choosing a new vertex and edge at each iteration.

Procedure PRIM takes a connected graph G and the root r (starting vertex) as input. All vertices not in the tree constructed so far are in a priority queue Q based on a key field. $\mathsf{key}[v]$ is the minimum weight of any edge connecting v to a vertex in the tree. $\mathsf{key}[v] = \infty$ if there is no such edge. $\pi[v]$ names the "parent" of v in the tree. The MST will be:

$$\{(v, \pi[v]) | v \in V - \{r\}\} \text{ after executing } \textsc{Prim}(G, r)$$

```
PRIM(G, r)
  begin
    Q ← V
    for each u ∈ Q
      do key[v] ← ∞
    key[r] ← 0
    π(r) ← NIL                          since root r has no parent
    while Q ≠ ∅ do
        u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u] do
          if (v ∈ Q ∧ w(u, v) < key[v])
            then π[v] ← u
                 key[v] ← w(u, v)
  end
```

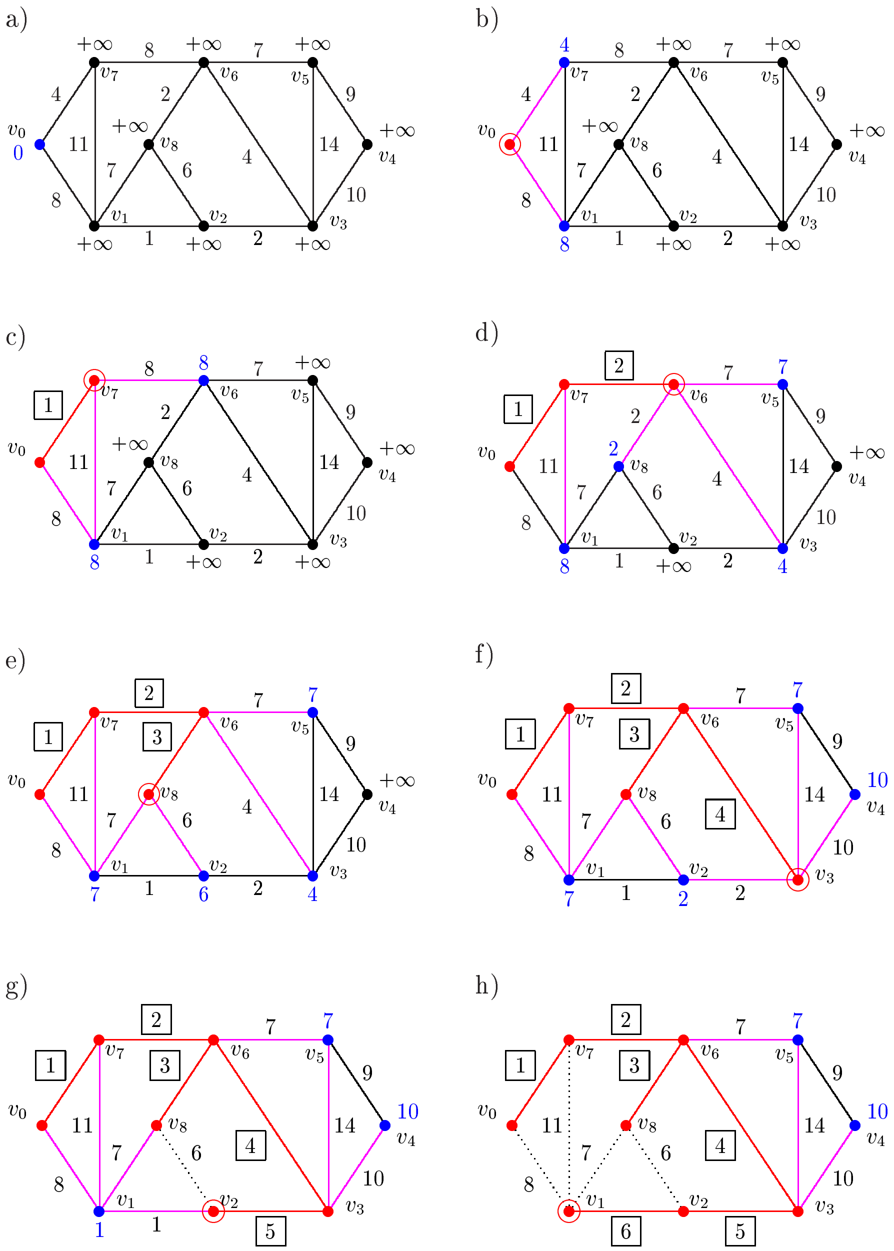
Vertices are removed one by one from Q to form the tree. If Q is implemented as a binary heap, BUILD-HEAP can be used for the initialization in $O(|V|)$ time. The total work for the loop is $|V|$ and since each EXTRACT-MIN operation takes $O(\log_2 |V|)$, $O(|V| \log_2 |V|)$ calls for EXTRACT-MIN are required. The inner **for** loop is executed $O(|E|)$ times, and $\mathsf{key}[v] \leftarrow w(u,v)$ is an $O(\log_2 |V|)$ operation on the heap (the test for membership in Q can be implemented in constant time by keeping a bit in every vertex that tells whether it is in Q or not, and updating the bit when the vertex is removed from Q). Therefore we obtain a total running time of

$$O(|V| \log_2 |V| + |E| \log_2 |V|) = O(|E| \log_2 |V|)$$

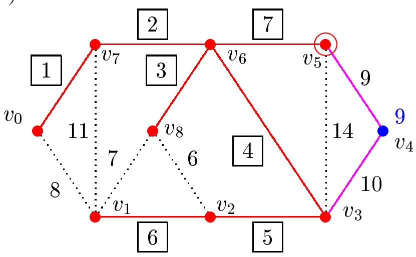
for Prim's algorithm. By using Fibonacci Heaps as priority queues, the running time can be improved to

$$O(|E| + |V| \log_2 |V|)$$

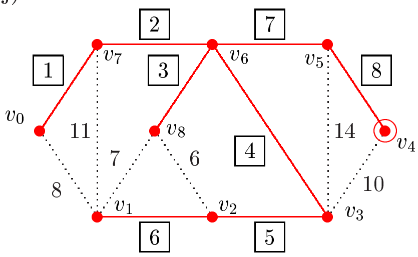
EXAMPLE: The following figures (a)–(j) show the execution of Prim's algorithm.



i)

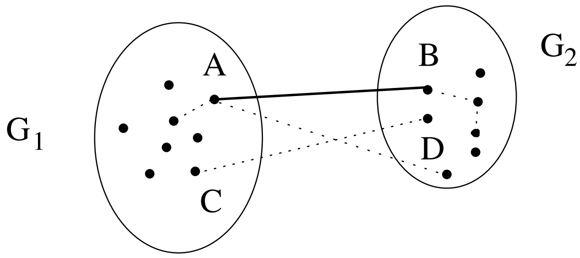


j)



The greedy algorithms by Kruskal and Prim work due to following.

Theorem 3.7.1 *Given any division of the vertices of a graph into two set G_1 and G_2, the MST of the graph contains the shortest of the edges with one endpoint in G_1 and the other in G_2.*



Proof (*by contradiction*): Suppose that $< A, B >$ is the shortest edge with one endpoint A in G_1 and the other endpoint B in G_2.
Suppose that T is the MST of the graph and $< A, B > \notin T$. Consider $T \cup \{< A, B >\}$.

$T \cup \{< A, B >\}$ contains a cycle since it now has n edges. Note that T contains an edge $< C, D >$ with $C \in G_1$ and $D \in G_2$. Consider $T' = T \cup \{< A, B >\} - \{< C, D >\}$. T' is a spanning tree of the graph and $w(T') < w(T)$ since $w(< A, B >) < w(< C, D >)$. But this contradicts the minimality of T. Therefore $< A, B >$ must be in T. $\square$

Once again, we pause from greedy algorithms for a while to consider more sophisticated priority queues.

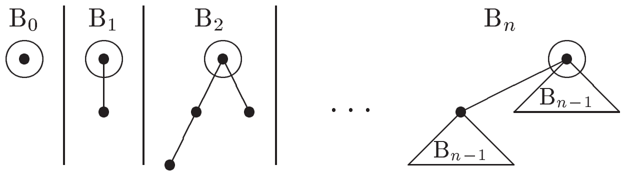
3.7.4 Binomial Heaps and Fibonacci Heaps

Ordinary binary heaps (as used in heapsort) are adequate to be used as priority queues as long as we don't need to merge trees. Merging (via UNION operation) uses heapify and therefore takes $\Theta(n)$ time in the worst case.

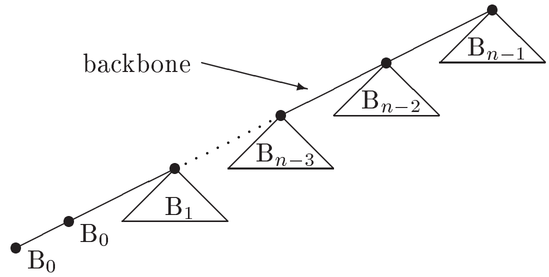
We now explore 2 priority queues that have better running times when performing the operation of merging trees. These two data structures – Binomial Heaps and Fibonacci Heaps (see Section 3.7.6) – are also known as mergeable heaps.

3.7.4.1 Binomial Heaps

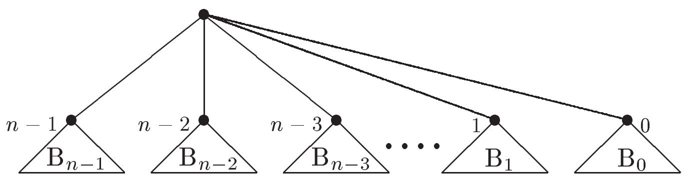
A Binomial Heap is a collection of Binomial trees. Binomial trees are defined recursively by:



Another way of looking on binomial trees may be:



or



Properties of B_n

1. *B_n has 2^n nodes.*
   Proof (*by induction*): B_0 has $2^0 = 1$ node. Assume that B_{k-1} has 2^{k-1} nodes. B_k consists of 2 copies of B_{k-1}, each having 2^{k-1} nodes (our induction hypothesis). Therefore B_k has $2^{k-1} + 2^{k-1} = 2^k$ nodes.

2. *The height of B_n is n.* Use induction on n.

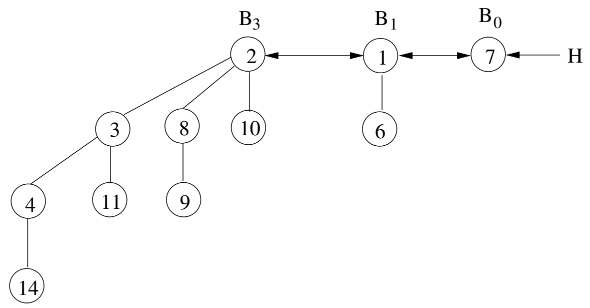
3. *The number of nodes of B_n at level k is the binomial coefficient $\binom{n}{k}$ (hence the name binomial tree).*
   Observation: Let $D(n, k)$ be the number of nodes of B_n at level k. A node at level k in B_{n-1} appears in B_n once at level $k + 1$ (left subtree) and once at level k (right subtree).
   Proof (*by induction on n*):
   B_0 has 1 node at level 0 ($\binom{0}{0} = 1$). Suppose the statement is true for B_{t-1}. For B_t, we have $D(t, i) = D(t - 1, i) + D(t - 1, i - 1)$ (by above observation). But $D(t - 1, i) = \binom{t-1}{i}$ and $D(t - 1, i - 1) = \binom{t-1}{i-1}$ by induction hypothesis. So $D(t, i) = \binom{t-1}{i} + \binom{t-1}{i-1} = \binom{t}{i}$. Therefore, there are $\binom{n}{k}$ nodes at level k of B_n.

4. *The root node of B_n has degree n which is greater than that of any other node. If the children of the root node are numbered from left to right by $n - 1, n - 2, \ldots, 0$, then child i ($0 \le i \le n - 1$) is the root node of subtree B_i.* It can be proven by using induction on n.

Remark: The maximum degree of any node in an n-node binomial tree is $\log_2 n$.

A **binomial heap** H is a set of binomial trees that satisfies the following binomial-heap properties:

EXAMPLE:



□

Figure 3.6: The binomial heap H has 11 nodes. $11 = (1011)_2$. So B_3, B_1 and B_0 are present with $8 + 2 + 1 = 11$ nodes.

1. Each binomial tree in H is heap-ordered: the key of a node is greater than or equal to the key of its parent. So the root node contains the smallest key in the tree.

2. There is at most one binomial tree in H whose root has a given degree. A binomial heap containing n nodes has at most $\lfloor \log_2 n \rfloor + 1$ binomial trees. The binary representation of n has $\lfloor \log_2 n \rfloor + 1$ bits b_i where

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i$$

Binomial tree B_i appears in the binomial heap H	$\Leftrightarrow$	Bit $b_i = 1$

Recall: B_i has 2^i nodes.
So the binomial heap H contains at most $\lfloor \log_2 n \rfloor + 1$ binomial trees.

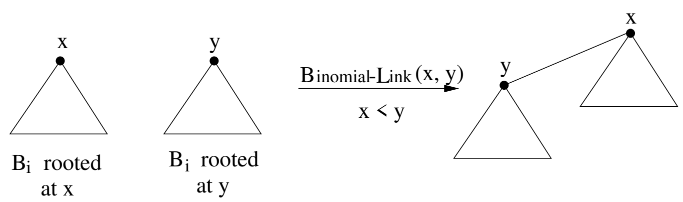
An example of a binomial heap is shown in Figure 3.6.

Operations on binomial heaps
Recall that a binomial heap is a collection of binomial trees of different sizes, each having the heap property (heap-ordered). So the minimum key is stored in one of the root nodes. The operations are:

① Creating a new binomial heap is an $\Theta(1)$ operation. The procedure returns an empty heap H.

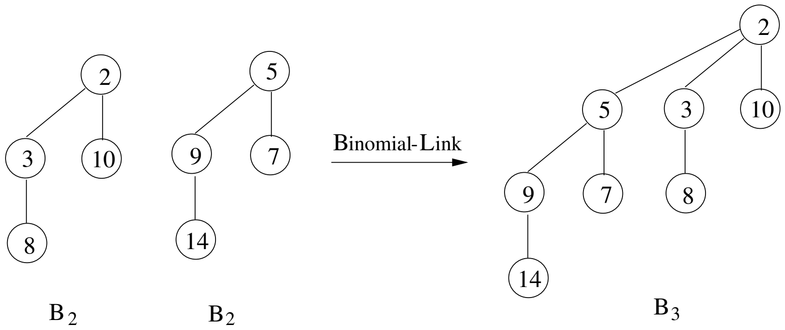
② Finding the minimum key is an $O(\log_2 n)$ operation, since we have to check the $\lfloor \log_2 n \rfloor + 1$ roots in the worst case.

③ Uniting two binomial heaps: two binomial heaps of different sizes can be united by using the procedure BINOMIAL-LINK that combines two binomial trees of equal sizes.



Make the root with the larger value (here y) into the i^{th} child of the root of the smaller value to form B_{i+1}. Growth rate $O(1)$.

EXAMPLE:



□

BINOMIAL-HEAP-UNION*(H_1, H_2)* merges two heaps H_1 and H_2, each consisting of a collection of binomial trees arranged in increasing order of size. Merging is done very much like binary addition. At each stage we have up to three ones, one carried from the previous position, and one from each operand:

- if there are none $\Rightarrow$ result: 0
- if there is just one $\Rightarrow$ result: 1
- if there are two ones $\Rightarrow$ result: 0+ carry 1 to next position
- if there are three ones $\Rightarrow$ result: 1+ carry 1 to next position

BINOMIAL-HEAP-UNION*(H_1, H_2)* first looks for B_0 in H_1 and H_2.

- if there are no B_0's $\Rightarrow$ do nothing at this stage
- if there is one B_0 $\Rightarrow$ keep it to form part of the result
- if there are two B_0's $\Rightarrow$ use BINOMIAL-LINK to merge them into B_1.

Now look for B_1's. There can be a maximum of three B_1's.

- if there is none $\Rightarrow$ do nothing
- if there is one B_1 $\Rightarrow$ keep it to form part of the result
- if there are two B_1's $\Rightarrow$ merge them into B_2
- if there are there B_1' s $\Rightarrow$ merge two of them into B_2; keep the third B_1 to form part of the result

and so on with B_2, B_3, ... As we go on, we join the roots of the binomial trees that are kept to form part of the result. We also keep track of the root with the smallest value (i.e. keep updating MIN). The merging operation has at most $\lfloor \log_2 n \rfloor$ stages each requiring at most one BINOMIAL-LINK which can be done in $O(1)$. So the complete merge takes $O(\log_2 n)$.

④ Inserting a node is done by BINOMIAL-HEAP-INSERT(H, x). Item x will be inserted in the binomial heap H by first creating a one element heap H' containing only x, and then using BINOMIAL-HEAP-UNION(H_1, H_2) to merge it with H. Creating H' takes $\Theta(1)$ as seen previously and merging H and H' takes $O(\log_2 n)$ again as already shown. So inserting a node takes $O(\log_2 n)$.

⑤ BINOMIAL-HEAP-EXTRACT-MIN*(H)* extracts the node with the minimum key. BINOMIAL-HEAP-EXTRACT-MIN performs the following steps:

EXAMPLE: Consider the following two binomial heaps H_1 and H_2.

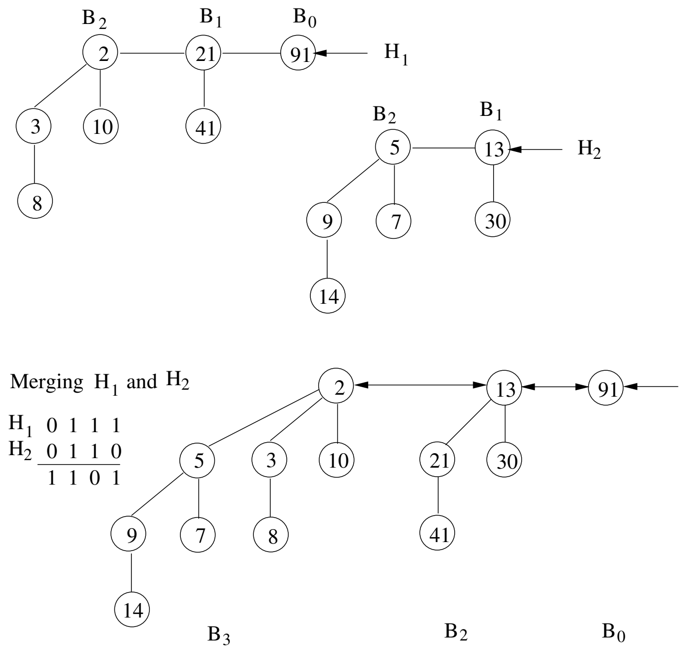


Figure 3.7: EXAMPLE for BINOMIAL-HEAP-UNION

□

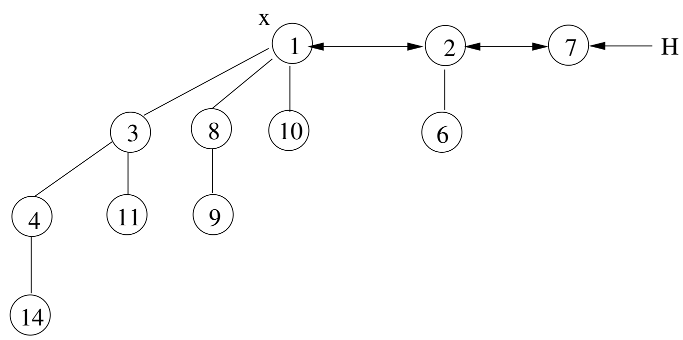
a) find the root x with minimum value and remove its binomial tree B from H, joining any parts of H into a new binomial heap H

b) remove x form B. Join the binomial trees that used to be the subtrees of x into a new binomial heap H_2.

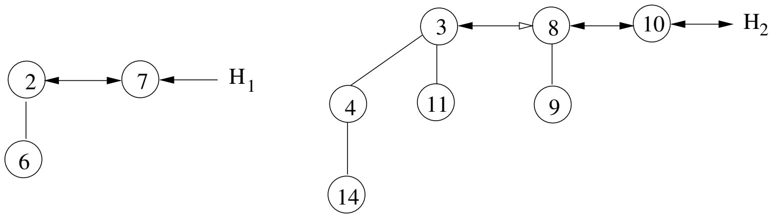
c) merge H_1 and H_2

Once again, extracting the node with minimum key is an $O(\log_2 n)$ operation.

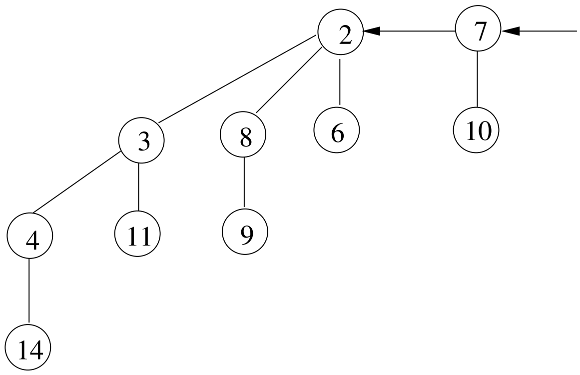
EXAMPLE for BINOMIAL-HEAP-EXTRACT-MIN*(H)*



after removing x from B_3 we get



after merging H_1 and H_2



□

⑥ Decreasing a key: BINOMIAL-HEAP-DECREASE-KEY*(H, x, k)* decreases the value of the node x in binomial heap H to a new value k. The procedure finds x, gives it its new value k and now checks to see if the heap-order is preserved. To do that, it just has to go up the tree as is the case with binary heaps. Going up the tree is an $O(\log_2 n)$ operation, since the maximum depth of x is $\lfloor \log_2 n \rfloor$

⑦ Deleting a key is performed by BINOMIAL-HEAP-DELETE*(H, x)*. BINOMIAL-HEAP-DELETE calls BINOMIAL-HEAP-DECREASE-KEY*(H, x, $-\infty$)* forcing x to "bubble up" to the root of its tree.
Then a call to BINOMIAL-HEAP-EXTRACT-MIN*(H)* will now remove x. Again the growth rate here is $O(\log_2 n)$.

We now study a new analyzing technique that is needed for a better understanding of algorithms that have parts that perform repetitive steps.

3.7.5 Amortized Analysis

In order to get a better understanding of Fibonacci Heaps, we need a new analyzing technique: **amortized analysis**. The amortized cost is the theoretical speed of a given set of operations.

It is an elegant method for analyzing the running times of certain algorithms in a more precise way. If a certain part of the algorithm is performed several times, each time with a different running time, then, instead of taking the worst case every time, we amortize the different costs; we get a tighter bound than the one produced by the worst case analysis. The bound produced by the worst case analysis is sometimes too pessimistic.

In an amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed. It is used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive.

The amortized cost is $O(f(n))$ when the worst case execution time of an entire sequence of n operations never exceeds $O(nf(n))$.

Amortized analysis is different from average-case analysis. There is no probability involved. We are averaging over a sequence of operations, not the possible running times of a single operation. The averaging process is called *amortization*. It is a clever way used by accountants to average large one-time costs over long periods of time. There are several different methods for deriving amortized bounds for a sequence of operations.

A. The aggregate method

We show that, for all n, a sequence of n operations takes worst-case time $T(n)$ in total. So in the worst case, the average cost, or the amortized cost, per operation is $\frac{T(n)}{n}$.
EXAMPLE: Incrementing a binary counter
$A[0 \ldots k]$ is an array of k bits representing a number X where

$$X = \sum_{j=0}^{k-1} A[j]2^j$$

So $A[0]$ holds the low-order bit, $A[k-1]$ the high-order bit. Assume $length(A) = k$. Consider procedure INCREMENT(A) that adds **one** to the binary representation of X.

```
INCREMENT(A)
begin
        j := 0
        while ((j < length(A)) ∧ (A[j] = 1))
                A[j] := 0
                j := j + 1
        if j < length(A)
                then A[j] := 1
end
```

Note that $A[j] = 1 \Leftrightarrow 2^j$ appears in the sum $X = \sum_{j=0}^{k-1} A[j]2^j$. The running time of procedure INCREMENT depends on the value of X, i.e. the contents of array A. If the first t bits of A are all ones, then INCREMENT takes $\Theta(t)$ time to flip the $t+1$ bits. So in general, if X is a number between 0 and n, INCREMENT takes $\Theta(\log_2 n)$ time in the worst-case, since the binary representation of n is $\lfloor \log_2 n \rfloor$ bits long.

Suppose we want to use INCREMENT to count from 0 to n. Start with $X = 0$, in other words

$$A[0] = A[1] = \ldots A[n-1] = 0$$

The worst-case running-time analysis yields $O(n \log_2 n)$ since we assume $\Theta(\log_2 n)$ per iteration. But do we flip $\Theta(\log_2 n)$ bits every time we call INCREMENT ?

j	bits that are flipped	total number of flips
1	$A[0]$	1
2	$A[0], A[1]$	2
3	$A[0]$	1
4	$A[0], A[1], A[2]$	3
5	$A[0]$	1
6	$A[0], A[1]$	2
7	$A[0]$	1
8	$A[0], A[1], A[2], A[3]$	4
9	$A[0]$	1
10	$A[0], A[1]$	2
11	$A[0]$	1
12	$A[0], A[1], A[2]$	3
13	$A[0]$	1
14	$A[0], A[1]$	2
15	$A[0]$	1
16	$A[0], A[1], A[2], A[3], A[4]$	5
total number of flips for $X = 0$ to $X = 16$ is 31		

We easily see that:

- $A[0]$ flips every time (or $\lfloor \frac{16}{1} \rfloor$ times)
- $A[1]$ flips every other time (or $\lfloor \frac{16}{2} \rfloor$ times)
- $A[2]$ flips every 4^{th} time (or $\lfloor \frac{16}{4} \rfloor$ times)
- $A[3]$ flips every 8^{th} time (or $\lfloor \frac{16}{8} \rfloor$ times)
- $A[4]$ flips every 16^{th} time; the next time it will flip will be for $X = 32$

In general: $A[j]$ flips every $(2^j)^{th}$ time (or $\lfloor \frac{16}{2^j} \rfloor$ times). So if we start with an array of zeros ($X = 0$), a sequence of n increments flips each bit $A[j]$ exactly $\lfloor \frac{n}{2^j} \rfloor$ times. Hence the total number of bits flipped for the entire sequence (counting from 0 to n) is :

$$\sum_{j=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^j} \right\rfloor < \sum_{j=0}^{\infty} \frac{n}{2^j} = 2n$$

Thus, on average, each call to INCREMENT flips only $\frac{2n}{n} = 2$ bits, and so runs in constant time: $O(1)$. So the worst-case analysis has an amortized cost of $O(1)$.

The standard analysis is **not** wrong. It delivers an upper bound on the total time complexity which is much larger than it needs to be. This happens because the analysis fails to notice that not all bits are flipped at every single call of INCREMENT.

The second method for deriving amortized bounds is called the accounting method.

B. Accounting method

Assign an amortized cost to each operation. Amortized cost may be more or less than the actual cost for some operations. If the amortized cost is more than the actual cost of the operation, we assign the difference to specific objects of the data structure as credit. The credit can be used later to help pay for operations whose amortized cost is less than their actual cost. No negative credit is allowed. The total amortized cost should be an upper bound to the total actual cost. The basic concept with amortized analysis is that when we perform certain operations (usually $O(1)$ operations), we "put money in the bank" (credit) which is saved and used on "rainy days" (when needed).

EXAMPLE: Incrementing a binary counter

Recall: The running time is proportional to the number of bits flipped, which we use as amortized cost.

Amortized cost of flipping a bit from 0 to 1 is \$2. One of these dollars is spent flipping the bit from 0 to 1. The other dollar is stored away as credit until we need to reset the same bit back to zero. Note that we always have enough credit saved up to pay for the next increment

```
INCREMENT(A)
begin
j := 0
while ((j < length(A)) ∧ (A[j] = 1))
    A[j] := 0                          cost: 0 !!
    j := j + 1
if j < length(A)
    then A[j] := 1                     cost: 2
end
```

Remark: The total amortized cost is always $\geq$ total worst-case cost.

X	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	amortized cost
1	0	0	0	0	1(1)	2
2	0	0	0	1(1)	0	2
3	0	0	0	1(1)	1(1)	2
4	0	0	1(1)	0	0	2
5	0	0	1(1)	0	1(1)	2
6	0	0	1(1)	1(1)	0	2
7	0	0	1(1)	1(1)	1(1)	2
8	0	1(1)	0	0	0	2
9	0	1(1)	0	0	1(1)	2
10	0	1(1)	0	1(1)	0	2
11	0	1(1)	0	1(1)	1(1)	2
12	0	1(1)	1(1)	0	0	2
13	0	1(1)	1(1)	0	1(1)	2
14	0	1(1)	1(1)	1(1)	0	2
15	0	1(1)	1(1)	1(1)	1(1)	2
16	1(1)	0	0	0	0	2
total amortized cost = 32						

Remark: (1) is the \$1 credit which is used to flip back the one to zero.

The amortized cost of an increment operation is \$2 (as is the case with the aggregate method). Also note that we always have enough credit to reset the flipped bits; credit is never negative.

The potential method is the third and last method for deriving amortized bounds.

C. The potential method

Rather than storing credit to represent prepaid work that can be spent later, we represent prepaid work as "potential energy" or just "potential". The term was borrowed from physics. The potential is associated with the entire data structure and not with specific parts of the data structure. Suppose that D_i represents the state or configuration of the data structure at some moment in time i. The state of D_i can be its size, its shape, the values it contains or other characteristics. The potential function Φ assigns a real number $\Phi(D_i)$ to D_i. The potential can be thought of as a measure of susceptibility of D_i to expensive operations. The more susceptible to expensive operations D_i is, the higher its potential $\Phi(D_i)$. Let c_i denote the actual cost of the i^{th} operation which transforms the data structure from state D_{i-1} to state D_i. The amortized cost $\overline{c_i}$ of the i^{th} operation is defined to be the sum of the actual cost and the change in potential:

$$\overline{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

The total amortized cost of the n operations is given by:

$$\begin{aligned} \sum_{i=1}^{n} \overline{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \end{aligned}$$

So the total amortized cost is the total actual cost plus the total change in potential. Note that if we can define a potential function Φ such that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \overline{c_i}$ will be an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

Since we don't know a priori the number of operations performed, it is safer to require that $\Phi(D_i) \geq \Phi(D_0)$ for all i. So the goal is to define Φ such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$. Choosing a potential function that works well is a matter of trial-and-error and of course experience. Different potential functions will lead to different time bounds. One should try different possibilities with the data structure at hand.

Example: Incrementing a binary counter A

Define the potential of the counter after the i^{th} increment operation to be the number of ones in the counter. Denote the number of ones by b_i. Initially, all bits are equal to zero, $b_0 = 0$ and so $\Phi(D_0) = 0$. Moreover, $\Phi(D_i) > 0$ for all $i > 0$. Assume that the i^{th} call to increment resets to 0 t_i bits. The actual cost, c_i, is given by:

$$\begin{aligned} c_i &= \text{number of bits set to 1} + \text{number of bits reset to 0} \\ &\leq 1 + t_i \text{ since at most one bit is set to one} \end{aligned}$$

The number of ones in A after the i^{th} operation is performed is given by

$$\begin{aligned} b_i &= b_{i-1} - \begin{pmatrix} \text{number of ones} \\ \text{that are reset} \\ \text{to zero} \end{pmatrix} + \begin{pmatrix} \text{at most one} \\ \text{set to one} \end{pmatrix} \\ \Rightarrow b_i &\leq b_{i-1} - t_i + 1 \end{aligned}$$

So

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= b_i - b_{i-1} \\ &\leq b_{i-1} - t_i + 1 - b_{i-1} \\ &= 1 - t_i \end{aligned}$$

Therefore the amortized cost of the i^{th} operation is:

$$\begin{aligned}\overline{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

So here again the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case of n INCREMENT operations is $O(n)$.

Note that b_i is a measure of susceptibility of the state of the array A to the expensive operation of flipping bits. A state with a large b_i is more apt to undergo flips than a state in which we have a smaller value for b_i. □

In summary, define a potential function for the data structure such that $\Phi(D_0) = 0$ and $\Phi(D_i) > 0$ for all $i > 0$. The amortized cost of an operation is its actual cost plus the change in potential. Amortized analysis is a technique for bounding the running time of an algorithm. It is effective when analyzing an algorithm that performs a lot of cheap operations and occasionally performs expensive operations. The analysis allows us to balance the expensive-but-rare operations against their cheap-and-frequent counterparts.
We now resume our study of priority queues.

3.7.6 Fibonacci Heaps

Fibonacci Heaps were invented by Fredman and Tarjan in 1984. Priority queues can be very efficiently implemented by Fibonacci heaps. We shall use the method of amortized analysis with potential function to analyze Fibonacci heaps.

Recall:

- we quantify some aspect of the data structure by using a potential function. This measure is usually proportional to the amount by which the data structure is inefficiently arranged
- if operations are only expensive when the data structure is complicated and expensive operations can clean up the data structure, and it takes many cheap operations to increase the complexity of the data structure, then we can amortize the cost of the expensive operations over the cost of the many cheap operations to obtain a low average cost

Fibonacci Heaps are mergeable heaps. They maintain the minimum key in the heap at all times. Unlike Binomial Heaps, with Fibonacci heaps we can have one more copy of B_i for any i and trees are not necessarily binomial trees. A Fibonacci heap is a circular doubly linked list, with a pointer to the minimum key. The various trees, comprising the Fibonacci heap, are heap-ordered. Every node in a Fibonacci heap has a pointer to four other nodes:

- its parent
- its "next" sibling
- its "previous" sibling
- one of its children

The sibling pointers are used to join the roots together into a circular doubly linked list.

The advantages of a circular, doubly linked list are:

EXAMPLE:

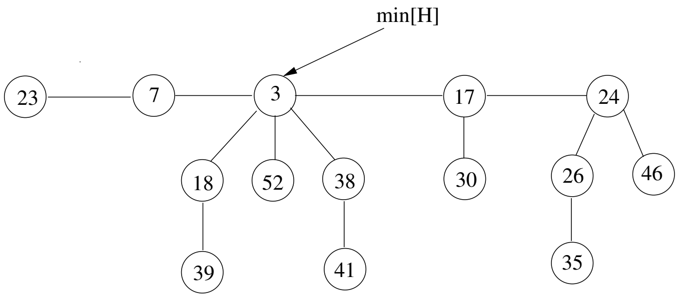


Figure 3.8: A high-level view of a Fibonacci Heap

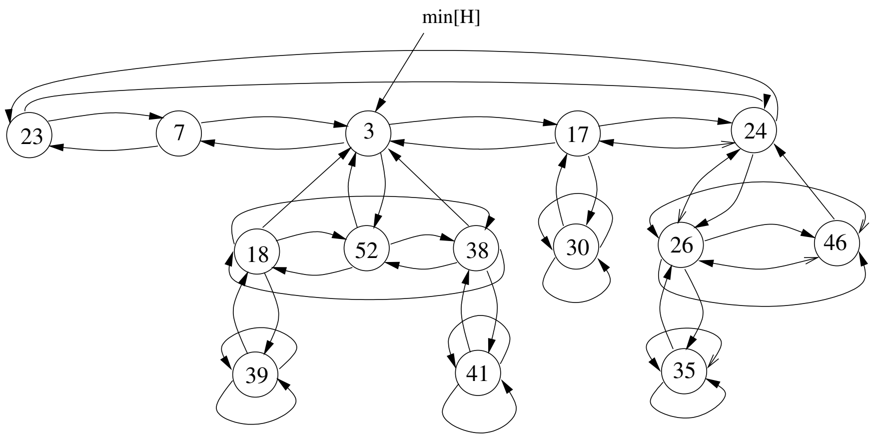


Figure 3.9: A detailed view of a Fibonacci Heap

□

1. removal of a node takes $O(1)$ time
2. concatenation of two circular, doubly linked lists into one circular, doubly linked list can be done in $O(1)$ time

The following table is a summary of the operations' running times for Binomial Heaps and Fibonacci Heaps.

procedure	Binomial Heap [worst-case]	Fibonacci Heap [amortized]
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$O(\log_2 n)$	$\Theta(1)$
MINIMUM	$O(\log_2 n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log_2 n)$	$O(\log_2 n)$
UNION	$O(\log_2 n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log_2 n)$	$\Theta(1)$
DELETE	$\Theta(\log_2 n)$	$O(\log_2 n)$

The number of items in the heap H at the time of the call to the procedure is denoted by n. As can be seen in the table, we use amortized analysis with Fibonacci heaps. In some operations, we shall mark nodes. If $t(H)$ denotes the number of trees in Fibonacci heap H and $m(H)$ denotes the number of marked nodes then the potential function of the Fibonacci heap is given by:

$$\Phi(H) = t(H) + 2m(H)$$

In the example, if nodes 18, 39 and 26 are marked then $\Phi(H) = 5 + 2 \cdot 3 = 11$. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. The most important idea with Fibonacci heap operations is to delay work as long as possible. When it is finally done, use the operation to simplify the data structure so that future work is easy.

Operations with fibonacci heaps

① Creating a new Fibonacci Heap
MAKE-FIB-HEAP creates an empty Fibonacci Heap H. $\Phi(H) = 0$ since $t(H) = m(H) = 0$. The amortized cost of MAKE-FIB-HEAP is $\Theta(1)$ (which is the actual cost)

② Inserting a node
FIB-HEAP-INSERT*(H, x)* inserts a node x in the Fibonacci Heap H by performing the following operations:

a) create a tree of type B_0 containing x
b) add this new tree to the list of roots of H
c) update *min[H]*, the minimum root pointer
d) increment $n[H]$ where $n[H]$ keeps track of the number of nodes of the Fibonacci Heap H

Let H' be the Fibonacci Heap obtained after performing FIB-HEAP-INSERT*(H, 21)* (Figure 3.10).

$$\begin{aligned}
t(H') &= t(H) + 1 \\
m(H') &= m(H) \\
\Phi(H') - \Phi(H) &= [t(H') + 2m(H')] - [t(H) + 2m(H)] \\
&= [t(H) + 1 + 2m(H)] - [t(H) + 2m(H)] \\
&= 1
\end{aligned}$$

So the amortized cost = actual cost $+1 = O(1) + 1 = O(1)$.

③ Finding the minimum node
FIB-HEAP-MIN*(H)* returns the minimum key by returning the value of the node pointed to by *min[H]*. Actual time: $O(1)$, which is also equal to the amortized cost since the potential of H does not change.

④ Uniting two Fibonacci Heaps
FIB-HEAP-UNION*(H_1, H_2)* merges H_1 and H_2, destroying them in the process, and returns a new heap H, by performing the following operations:

a) combine the root lists of H_1 and H_2 to form a single list of heap H
b) update the minimum root pointer *min[H]*
c) set $n[H] = n[H_1] + n[H_2]$

Example:

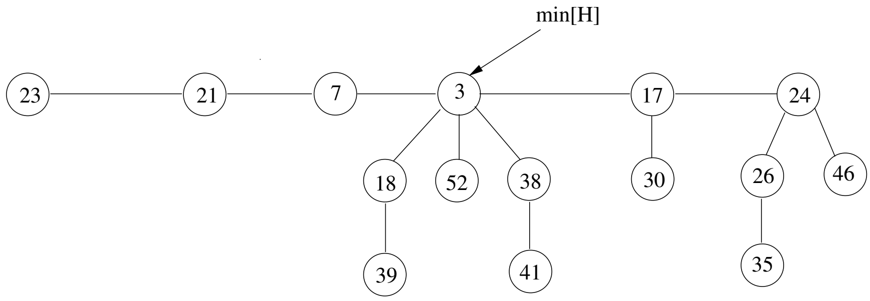


Figure 3.10: Fibonacci Heap H (Figure 3.8) after performing Fib-Heap-Insert*(H, 21)*

□

The change in potential is given by:

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= [t(H) + 2m(H)] - [(t(H_1) + 2m(H_1)) \\ &\quad + (t(H_2) + 2m(H_2))]\end{aligned}$$

But $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. So

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$$

The amortized cost and the actual cost of Fib-Heap-Union are therefore equal. The cost of both is $O(1)$.

⑤ Extracting the minimum node
Fib-Heap-Extract-Min*(H)* removes the minimum node and rearranges the heap. It is a complicated operation because this is where the delayed work of consolidating trees in the root list finally takes place. The main steps of Fib-Heap-Extract-Min*(H)* are:

a) $z := min[H]$
b) delete z from the root list of H and add the children of z to the root list of H
c) call procedure Consolidate*(H)* which scans the root list and repeatedly links trees of equal degree, and finds the new minimum
d) decrement $n[H]$ by 1

Most of the work done in Fib-Heap-Extract-Min*(H)* is performed by Consolidate*(H)*.

We want to reduce the number of trees in the Fibonacci Heap, i.e. we want to consolidate the root list of H. We link trees of equal degrees using procedure Consolidate*(H)*. But how do we know when we have trees of equal degree ?

The situation with binomial heaps is simple since the binomial trees are ordered in increasing order of degree. With Fibonacci Heaps, trees may occur in any order. A solution is to have the procedure Consolidate*(H)* use an array $A[1 \ldots D(n(H))]$ of the trees already scanned, indexed by degree.

Consolidate*(H)* initializes A to be empty. The procedure then tries to add a tree T of degree k in location $A[k]$. If that location is empty, the tree is

EXAMPLE: Extract the key with minimum value form the binomial heap H of Figure 3.10

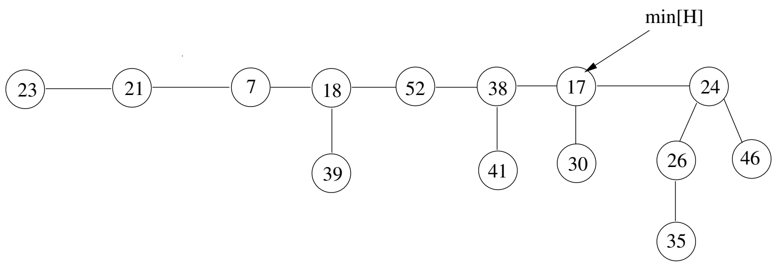


Figure 3.11: Heap H' (Figure 3.10) after performing steps a) and b) of FIB-HEAP-EXTRACT-MIN*(H)*. Note: $min[H]$ points to the right node of node 3

□

pointed to by $A[k]$ and CONSOLIDATE moves to another tree from the root list. But if location $A[k]$ is not empty, then we have another tree T' of degree k. CONSOLIDATE will then link T and T' by making the tree with the larger root a child of the tree with the smaller root and incrementing the smaller node's degree. Suppose that T'' is the resulting tree. CONSOLIDATE*(H)* will then recursively try to add T'' at location $A[k'']$, where k'' is the degree of T''. Note that at any given instant, if $A[k] = y$, then y is currently a root with degree k. At the end of the scan, when there are no more nodes in the root list of H, the trees are read out of A into a new root list with the new updated *min[H]*.

$D(n)$ is the maximum degree of any node in an n-node Fibonacci Heap. It can be shown that if only the mergeable heap operations (MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, UNION) are supported, then $D(n) = \lfloor \log_2 n \rfloor$.

EXAMPLE:

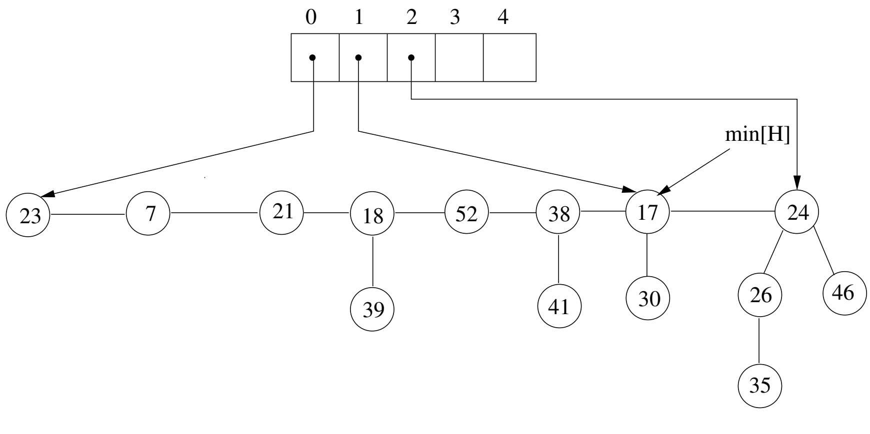
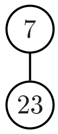


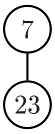
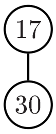
Figure 3.12:

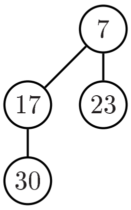
CONSOLIDATE*(H)* scans the roots 17, 24, and 23 (in that order) filling the array A of scanned trees as shown in Figure 3.12

Now the following steps take place:

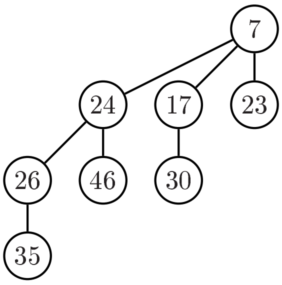
node 7

$A[0]$ is not empyt; so trees 7 and 23 get linked ⇒ 

$A[1]$ is not empty; trees  and  get linked

⇒ 

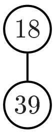
trees with roots at 7 and at 24 are linked

⇒ $A[3]$ points to 

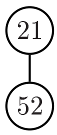
node 21

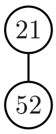
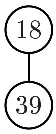
$A[0]$ points to 21

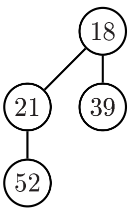
node 18

$A[1]$ points to 

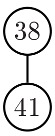
node 52

$A[0]$ is not empyt; link 52 and 21 ⇒ 

$A[1]$ is not empty, link  and 

⇒ $A[2]$ points to 

node 38

$A[1]$ points to 

The trees are then read out of A with updated min. □

Example:

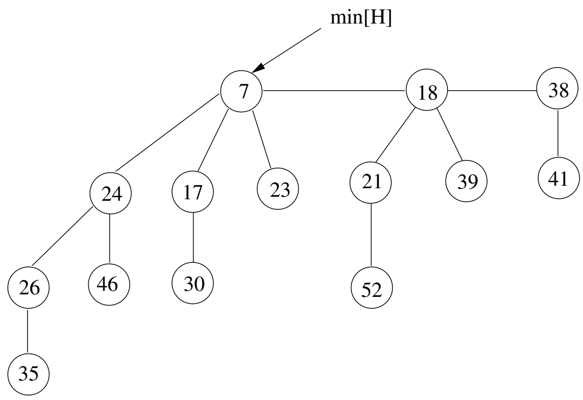


Figure 3.13: Result of FIB-HEAP-EXTRACT-MIN*(H)*

Let us analyze the work done to extract the minimum node. We shall show that the amortized cost of extracting the minimum node of an $n-$node Fibonacci Heap is $O(D(n))$. Let us compute the actual cost first. If the original heap had $t(H)$ roots then right prior to FIB-HEAP-EXTRACT-MIN*(H)*'s call to CONSOLIDATE*(H)*, the size of the root list is at most $D(n) + t(H) - 1$.
$D(n)$ is because of the minimum root's children, and
$t(H) - 1$ since we have already deleted the minimum root.

Since any root has at most $D(n)$ children, the operations of initializing the array A to null pointers and reading out the array at the end of CONSOLIDATE*(H)* take $O(D(n))$ time. Scanning the array checking for scanned trees in A by CONSOLIDATE*(H)* for every node in the root list is at most proportional to $D(n) + t(H)$. So the total actual work is

$$O(D(n) + t(H))$$

The potential before extracting the minimum is $t(H) + 2m(H)$, and the potential at the end of all operations is at most $(D(n) + 1) + 2m(H)$. Note that at most $D(n) + 1$ roots will remain after the execution of CONSOLIDATE*(H)*. Also neither procedure changes marked nodes. We know that

$$\text{amortized cost} \quad = \quad \text{actual cost } + \Phi(H') - \Phi(H)$$

So the amortized cost is at most:

$$\begin{aligned} & O(D(n) + t(H)) + [(D(n) + 1) + 2m(H)] - (t(H) + 2m(H)) = \\ = \quad & O(D(n)) + O(t(H)) - t(H) \\ = \quad & O(D(n)) \end{aligned}$$

Note that $O(t(H))$ and $t(H)$ cancel out because we can scale up the units of potential so as to balance the constant in $O(t(H))$.

6 Decreasing a key
the five operations seen so far are mergeable heap operations for which $m(H)$ plays no role in the potential function. Marking nodes is very important for

FIB-HEAP-DECREASE-KEY*(H, x, k)*. We say that x is marked if its mark is set to "true", and is unmarked if its mark is set to "false". A root is always unmarked. We mark x if it is not a root and it loses a child (i.e. one of its children is cut and put into the root list). We unmark x whenever it becomes a root. We shall make sure (later) that no marked node loses another child before it itself is cut (and so changes status to unmarked). Here we cut the node, change its key and insert it into the root list. If the parent of the node is unmarked, mark it. If the parent of the node is marked, cut it off also. Recursively perform the above until we get an unmarked node that we finally mark.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
  begin
    assume k ≤ old key, otherwise send error message
    key[x] := k
    y := parent[x]
    if (y ≠ NIL) ∧ (key[x] ≤ key[y])
        then CUT(H, x, y)                          since heap-order
                                                   violated
             CASCADINGCUT(H, y)
    if key[x] < key[min[H]]
        then min[H] := x
  end
```

```
CUT(H, x, y)
  begin
    remove x from the child list of y
    decrement degree(y)
    add x to the root list of H
    parent[x] := NIL
    mark[x] := false
  end
```

When the heap-order is violated, many changes in the heap may happen (recall HEAPSORT). Marking nodes ensures that the running time functions of the operations are within the bounds we want. Consider following scenario for node x:

(a) x was a root

(b) then x was linked to another node

(c) then 2 children of x were removed by cuts

When x loses the second child, x is cut from its parent and joins the root list.

```
CASCADINGCUT(H, y)
  begin
    z := parent[y]
    if z ≠ NIL
      then if mark[y]=false
             then mark[y] :=true
             else CUT(H, y, z)
                  CASCADINGCUT(H, z)
  end
```

We then set to *true* x's parent's mark, unless x's parent is a root. If the parent of x was already marked, then recursively cut the link between the parent and grandparent of x. These cuts may cascade upwards until reaching either a root or an unmarked node.

Note that marking prevents us from removing more than one child from a node that is not a root node. Whenever we try to remove a second child from a non-root node, we first convert the node into a root by cutting it away form its parent.

EXAMPLE:

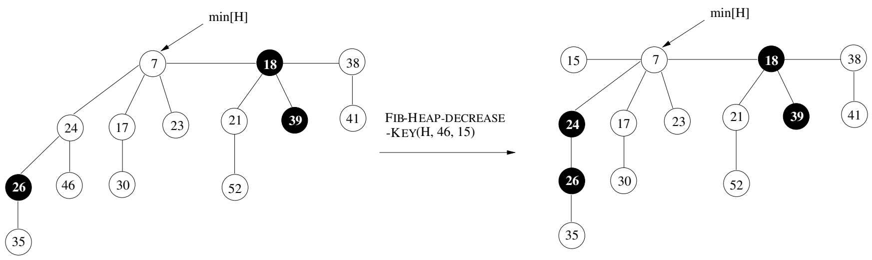


Figure 3.14: FIB-HEAP-DECREASE-KEY*(H, 46, 15)*

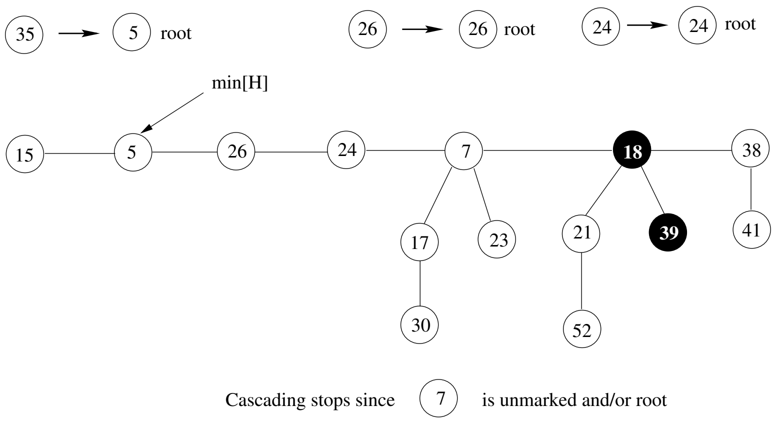


Figure 3.15: Based on the result of 3.14, we perform FIB-HEAP-DECREASE-KEY*(H, 35, 5)*

□

The amortized cost of FIB-HEAP-DECREASE-KEY is $O(1)$, as shown in the following:

(a) actual cost:
if we assume that FIB-HEAP-DECREASE-KEY calls CASCADINGCUT, which recursively calls itself c times, then the total actual time is $O(c)$. Note that c is bounded by the height of the tree

(b) change in potential
suppose that H is the original Fibonacci Heap (prior to the call to FIB-HEAP-DECREASE-KEY). Each recursive call of CASCADINGCUT, except for the last one (i.e. $c - 1$ calls), cuts a marked node and clears the

mark bit. If H' denotes the Fibonacci Heap after executing FIB-HEAP-DECREASE-KEY then:

$$t(H') = t(H) + \underbrace{(c-1)}_{\substack{\text{from}\\ \text{CASCADINGCUT}}} + \underbrace{1}_{\substack{\text{tree rooted}\\ \text{at } x}}$$

and

$$m(H') \leq m(H) - \underbrace{(c-1)}_{\substack{\text{unmarked by}\\ \text{CASCADINGCUT}}} + \underbrace{1}_{\substack{\text{last call to}\\ \text{CASCADINGCUT}\\ \text{may have marked}\\ \text{a node}}}$$

So $t(H') = t(H) + c$ and $m(H') \leq m(H) - c + 2$. Therefore

$$\begin{aligned}
\Phi(H') - \Phi(H) &= (t(H') + 2m(H')) - (t(H) + 2m(H)) \\
&= t(H') - t(H) + 2(m(H') - m(H)) \\
&\leq c + 2(-c+2) \\
&= 4 - c
\end{aligned}$$

Remember: amortized cost = actual cost + change in potential
so we get:
amortized cost $\leq O(c) + 4 - c = O(1)$ by scaling up the units of potential to offset the constant hidden in $O(c)$.

In FIB-HEAP-DECREASE-KEY, a CASCADINGCUT which causes the node above it to cut because it was already marked, which causes the node above it to be cut since it too was already previously marked, etc ... can increase the actual cost (c in our analysis). We can pay for this cost by our potential function. Every cost we incur from having to update pointers due to a marked node that was cut is offset by the decrease in the potential function when that previously marked node is now left unmarked in the root list. Note that the potential function has $\underline{2}m(H)$ as one of its terms. When a marked node y is cut by CASCADINGCUT, its mark becomes *false*, so the potential is reduced by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y joining the root list.

7 Deleting a node
a node can be deleted from an $n-$node Fibonacci Heap in $O(D(n))$ amortized time.

```
FIB-HEAP-DELETE(H, x)
  begin
  FIB-HEAP-DECREASE-KEY(H, x, −∞)
  FIB-HEAP-EXTRACT-MIN(H)
  end
```

So x joins the root list and becomes the node with minimum value, and is then removed. Since the amortized time of FIB-HEAP-DELETE is the sum of the amortized time of FIB-HEAP-DECREASE-KEY (which is $O(1)$) and the amortized time of FIB-HEAP-EXTRACT-MIN (which is $O(D(n))$), we have $O(D(n))$ as amortized time for FIB-HEAP-DELETE. □

Bounding the maximum degree

We claimed (in the table) that extracting the minimum and deleting a node from Fibonacci Heaps each requires $O(\log_2 n)$ work. But the analysis we did was in terms of $D(n)$, the upper bound on the degree of any node of an $n-$node Fibonacci Heap. To achieve $O(\log_2 n)$ we have to show that $D(n)$ is $O(\log_2 n)$.

Note that when the Fibonacci Heap undergoes only mergeable-heap operations MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, UNION then $D(n) = \lfloor \log_2 n \rfloor$. Let $size(x)$ denote the number of nodes in the subtree rooted at x ($size(x)$ includes x also). We will show that $size(x)$ is exponential in $degree[x]$.

Result 1: *If $degree[x] = k$, and $y_1, y_2, \ldots, y_k$ is the order in which each y_i was linked to x (from the earliest to latest), then $degree[y_i] \geq i-2$ for $i = 2, 3, \ldots, k$.*
Proof: Since y_i is linked to x after the linking of $y_1, y_2, \ldots, y_{i-1}$, we have $degree[x] \geq i-1$ since $y_1, y_2, \ldots, y_{i-1}$ are linked to x before y_i. So by the time y_i is linked to x, we have $degree[x] \geq i-1$. But y_i is linked to x only if $degree[x] = degree[y_i]$. So $degree[y_i] \geq i-1$ at the time y_i is linked to x. Since the time it has been linked, y_i has lost at most one child (recall that if it loses two children, it is then cut from x and moved to the root list). So $degree[y_i] \geq i-2$. □

Note that if we had **only** binomial trees in the heap, then **result 1** would have given $degree[y_i] = i-1$ (instead of $degree[y_i] \geq i-2$).

Result 2: *If F_k denotes the k^{th} Fibonacci Number then $F_{k+2} = 1 + \sum\limits_{i=0}^{k} F_i$.*
Proof: Use induction on k:

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 = 1 = F_2 \end{aligned}$$

So the property holds for $k = 0$. Assume that $F_{k+1} = 1 + \sum\limits_{i=0}^{k-1} F_i$.

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + (1 + \sum_{i=0}^{k-1} F_i) \text{ by induction hypothesis} \\ &= 1 + \sum_{i=0}^{k-1} F_i + F_k \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

So $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ for all $k \geq 0$. □

It can be shown by induction that $F_{k+2} \geq \Phi^k$ where $\Phi = \frac{1+\sqrt{5}}{2} = 1.618$.

Result 3: *If x is a node in a Fibonacci Heap and $degree[x] = k$, then $size(x) \geq F_{k+2} \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$.*
Proof: Let s_k denote the minimum possible value of $size(z)$ over all nodes z such that $degree[z] = k$. In other words:

$$s_k = \min\{size(z) | degree(z) = k\}$$

Note that $s_0 = 1$, $s_1 = 2$, $s_2 = 3$, in general $s_k \leq size(x)$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x.

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

But $s_k \geq F_{k+2}$. We show this by using induction on k. Recall that $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = \ldots$. Therefore

$$\begin{aligned} s_0 &= 1 \geq F_2 \\ s_1 &= 2 \geq F_3 \end{aligned}$$

Assume $s_i \geq F_{i+2}$ for $i = 0, 1, \ldots, k-1$.

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \text{ by induction hypothesis} \\ &= 1 + 1 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} \text{ by result 2} \\ &\Rightarrow size(x) \geq s_k \geq F_{k+2} \geq \Phi^k \end{aligned}$$

□

Result 4: The maximum degree $D(n) = O(\log_2 n)$
Proof: Let x be any node in an $n-$node Fibonacci Heap. Suppose $degree[x] = k$.

$$\Rightarrow n \geq size(x) \geq \Phi^k$$

$$\Rightarrow k \leq \log_\Phi n$$

So $D(n)$ is $O(\log_2 n)$. □

Chapter 4

Dynamic Programming

Dynamic Programming is a generalization of Divide-and-Conquer. It is based on the *principle of optimality*:

Optimal solutions can be built from optimal subsolutions.

Dynamic Programming is a **bottom-up** method, that checks all possible subproblems *once* and finds the *optimal solution*. Richard Bellman has written extensively about dynamic programming. The word "programming" in this context refers to the use of a tabular solution method.

> "Life can only be understood going backwards,
> but it must be lived going forwards."
> by **Soeren Kierkegaard**

(**Soeren Kierkegaard**, Danish philosopher, born 1813; founder of the ideas of individualism/existentialism; his philosophy was one of the main counterparts to Hegel's ideas)

4.1 General Remarks

Dynamic Programming can be used when the solution to a problem may be viewed as the result of a sequence of decisions. Sometimes an optimal sequence of decisions may be found by making the decisions **one at a time** without ever making an erroneous decision. This is true for all problems for which the greedy algorithm gives the optimum solution (EXAMPLE: fractional knapsack). For many other problems, stepwise decisions (based on local information) will not necessarily yield the optimum solution (EXAMPLE: $0-1$ integer knapsack). So enumerate **all** decision sequences and pick out the best. Dynamic Programming reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. There are two different approaches for writing the relation(s) formulating a dynamic programming:

forward: x_i is written in terms of optimal decision sequences for $x_{i+1}, \ldots, x_n$

backward: x_i is written in terms of optimal decision sequences for $x_1, \ldots, x_{i-1}$

Using dynamic programming usually entails breaking down the problem into stages at which the decisions take place, finding a relation that takes us backward from one stage to the previous one. Dynamic programming is usually used for formulating and solving a class of multistage decision problems.
We revisit the 0-1 knapsack problem and show how dynamic programming can be used to solve the problem.

4.2 The 0-1 Knapsack Problem

Problem instance

objects	1	2	...	i	...	n
weights	w_1	w_2	...	w_i	...	w_n
profits	p_1	p_2	...	p_i	...	p_n

KNAPSACK with capacity M

Feasible solution

A vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ where $x_i \in \{0, 1\}$ with associated weight: $W(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i \leq M$.

Objective function

Profit $P(\mathbf{x})$ for $\mathbf{x}$ feasible such that $P(\mathbf{x}) = \sum_{i=1}^{n} x_i p_i$.

Optimal solution

Maximum profit feasible solution

Let $P(j, m)$ be the profit of the **optimal solution** using $1, 2, \ldots, j$ objects and a portion of the knapsack of capacity m.
EXAMPLE:

	1	2	3	4	5	6
w_i	2	3	5	8	13	16
p_i	1	2	3	5	7	10

Then $P(3, 5) = 3$ since we either put objects 1 and 2 or object 3. $P(4, 11) = 7$ since we choose objects 2 and 4.
Note: $P(n, M) = P(n-1, M)$ if $w_n > M$, i.e. definitely do not choose object n.

$$P(n, M) = \max\{P(n-1, M), p_n + P(n-1, M - w_n)\} \text{ if } w_n \leq M$$

Note: $P(n-1, M)$ represents the case where object n is not chosen and $p_n + P(n-1, M - w_n)$ represents the case where object n is chosen.

In general

$$\begin{aligned} P(j, m) &= P(j-1, m) \text{ if } w_j > m \\ P(j, m) &= \max\{P(j-1, m), p_j + P(j-1, m - w_j)\} \text{ if } w_j \leq m \end{aligned}$$

with

$$\begin{aligned} P(1, m) &= p_1 \text{ if } w_1 \leq m \\ P(1, m) &= 0 \text{ if } w_1 > m \end{aligned}$$

The optimal solution is given by $P(n, M)$.

EXAMPLE:

objects	1	2	3	4	5	6
weights	1	2	5	8	12	14
profits	1	6	10	18	15	20

KNAPSACK with capacity 20

So the solution to the EXAMPLE is the vector $(1, 1, 1, 0, 1, 0)$ with a total profit of 32. (see Table 4.1).

6	0	6	7	7	10	11	16	17	17	18	19	19	22	23	28	29	29	29	31	32
5	0	6	7	7	10	11	16	17	17	18	19	19	22	23	28	29	29	29	31	32
4	0	6	7	7	10	11	16	17	17	18	19	19	22	23	28	29	29	29	29	29
3	0	6	7	7	10	11	16	17	17	17	17	17	17	17	17	17	17	17	17	17
2	0	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

capacity

Table 4.1: Solution for the EXAMPLE above. The leftmost column shows the objects.

4.2.1 Complexity of the dynamic programming algorithm for the 0-1 Knapsack problem

Each table entry can be calculated in constant time. So its time complexity is $\Theta(nM)$. The space requirement is also $\Theta(nM)$ to store the table. Note that M could be exponentially large compared to n, M could be $\Omega(2^n)$ for example. Then the algorithm would run in $\Omega(n2^n)$. If M is $O(p(n))$ for some polynomial $p(n)$, then the algorithm runs in $O(np(n))$. Algorithms that exhibit such behaviours are called pseudo-polynomial: an algorithm whose running time is a polynomial function of the magnitudes of the numbers in the problem instances. Recall that, on the other hand, a polynomial algorithm is polynomial in terms of the size of the input. Note that a traceback algorithm could be written to find the optimal knapsack from the various values of $P(j,m)$. The algorithm's running time is $\Theta(n)$. Since only two rows are needed to compute $P(j,m)$, we could discard the remaining previous rows. But the problem of tracing back to find the optimal filling (and not just the optimal profit) will not be solved. We need the whole table to determine the content of the optimal solution. Note that a top-down algorithm to calculate $P(n,M)$ would have a running time function $T(n) = 2T(n-1) + c$ since solving a problem of size n requires solving 2 problems of size $n-1$. So $T(n)$ is $\Omega(2^n)$ (as expected). □

Next we study the traveling salesperson problem, and show how one can use dynamic programming to solve it.

4.3 The Traveling Salesperson Problem

With the traveling saleman problem, we have n cities $1, 2, \ldots, n$ and $w(i,j)$ is the cost of going from city i to city j. A salesperson has to visit all the cities, starting with city 1 and ending at city 1 but without visiting a city more than once (except for city 1). The problem we would like to solve is: how can we find the tour (or circuit) with minimum cost?

The following is a formal definition of the traveling salesperson problem.

Problem instance

A weighted graph G in which each edge e has a nonnegative weight (or cost), $c(e)$.

Feasible solution

A *hamiltonian* circuit (one cycle which passes through *every* vertex *exactly once*).

Objective function

The cost of a hamiltonian circuit is defined to be:

$$C(HC) = \sum_{e \in HC} c(e) \ ,$$

where HC is a hamiltonian circuit

Optimal solution

A minimum-cost hamiltonian circuit.

$\square$

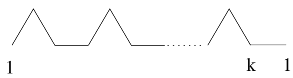
Let us use dynamic programming to solve this problem:

$G =< V; E > |V| = n$

$w(i,j)$	weight of edge ij (cost of ij, also denoted by $c(e)$)
HC	hamiltonian circuit (cycle starting and finishing at 1)

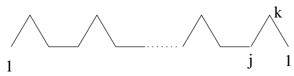
If k is the last vertex in HC before returning to vertex 1, then:

$$opt\ HC = \text{ optimum path from 1 to } k \text{ passing through all vertices } + w(k,1) :$$



Similarly:

$$opt\ HC = \text{ optimum path from 1 to } j + w(j,k) + w(k,1),\ j \in \{2,\ldots,n\} \backslash \{k\}$$



The process can be repeated.

For $S \subseteq \{2,\ldots,n\}$, $\quad k \in S$:

let $P(S,k) =$ cost of optimum path from 1 to k, such that vertices of path are in $S \backslash \{k\}$.

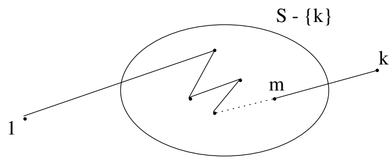
The cost of optimum HC is:

$$\min\{P(\{2,\ldots,n\},k) + w(k,1) \text{ for } 2 \le k \le n\}$$

This is a bottom - up computation:

$$P(S,k) = w(1,k),\ S = \{k\},\ \text{i.e. } |S| = 1$$

$$P(S,k) = \min\{P(S \backslash \{k\}, m) + w(m,k) \mid m \in S \backslash \{k\}\} \ , \text{if } |S| > 1$$



EXAMPLE (see Figure 4.1)
We are given the following graph in which we assume that all circuits start at 1.

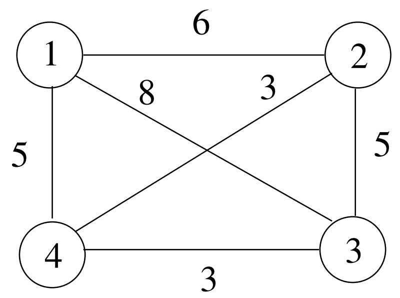


Figure 4.1:

The computation of the first values of S, k and P(S,k) are summarized in the following table:

S	**k**	**P(S, k)**
$\{2\}$	2	$6 = w(1,2)$, $\lvert S \rvert = 1$
$\{3\}$	3	$8 = w(1,3)$, $\lvert S \rvert = 1$
$\{4\}$	4	$5 = w(1,4)$, $\lvert S \rvert = 1$
$\{2,3\}$	2	$13 = \min\{P(\{3\},3) + w(2,3)\}$
		$= \min\{w(1,3) + w(2,3)\}$
		(see Figure 4.2)
$\{2,3\}$	3	$11 = \min\{P(\{2\},2) + w(2,3)\}$
		$= \min\{w(1,2) + w(2,3)\}$
		(see Figure 4.3)
$\{2,3,4\}$	4	$\min\{P(\{2,3\},2) + w(2,4)\},$
		$\{P(\{2,3\},3) + w(3,4)\}$
		$= \min\{13+3, 11+3\}$
		$= \min\{16, 14\} = 14$
Similarly for S={2,3,4}, with $k=3$ and with $k=2$		

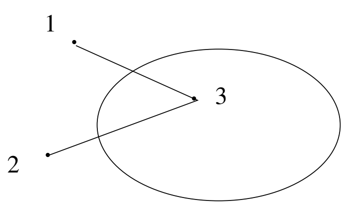


Figure 4.2:

The whole bottom-up process is shown in the diagram given by Figure 4.4.

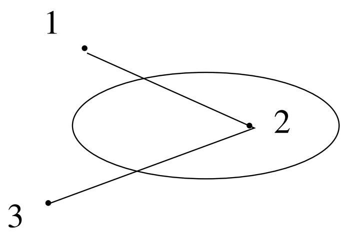


Figure 4.3:

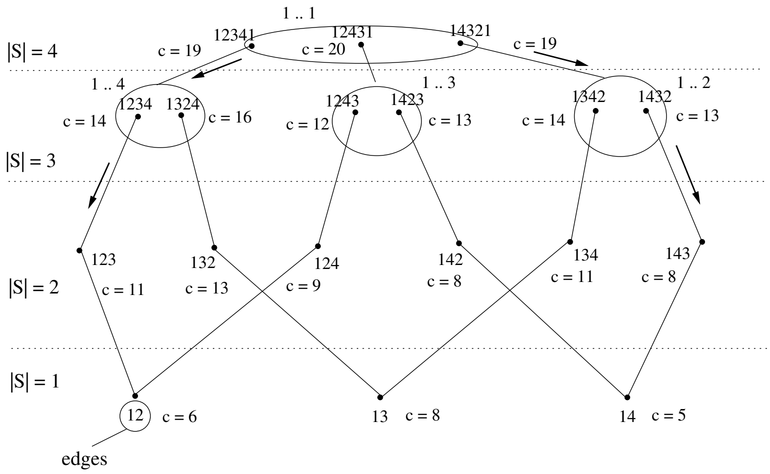


Figure 4.4:

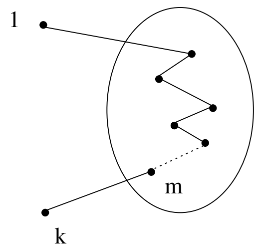
Remarks on the diagram

See Figure 4.4.

1 :

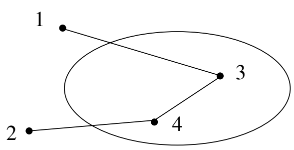
At the bottom level: $S = \{2\}, S = \{3\}, S = \{4\}$
At the first level: $S = \{2, 3\}, S = \{2, 4\}, S = \{3, 4\}$
At the last level: $S = \{2, 3, 4\}$

2:



is equivalent to path $1 \ldots\ m\ k$

EXAMPLE:



is 1 3 4 2, where $m = 4$ and $k = 2$
3:
c stands for cost, i.e. $c(p, k)$ is the cost of path p that ends in k

4.3.1 Complexity of the dynamic programming algorithm for the Traveling Salesperson Problem

We count the number of times $P(S, k)$ is computed. If $|S| = j$ then there are j possible k's. But there are $\binom{n-1}{j} = \frac{(n-1)!}{j!(n-1-j)!}$ possible subsets of $\{2, \ldots, n\}$ for each value of j.
EXAMPLE: In our example $n = 4$.

$$|S| = 2 \qquad \binom{3}{2} = \frac{3!}{2!1!} = 3 \text{ subsets of 2 elements.}$$

They are $\{2, 3\}$, $\{3, 4\}$ and $\{2, 4\}$.

$$|S| = 3 \qquad \binom{3}{3} = \frac{3!}{3!0!} = 1 \text{ subset of 3 elements.}$$

It is $\{2, 3, 4\}$. Therefore the number of $P(S, k) = \sum_{j=1}^{n-1} j\binom{n-1}{j}$
Remark:

$$(1 + x)^{n-1} = \sum_{j=0}^{n-1} \binom{n-1}{j} x^j$$

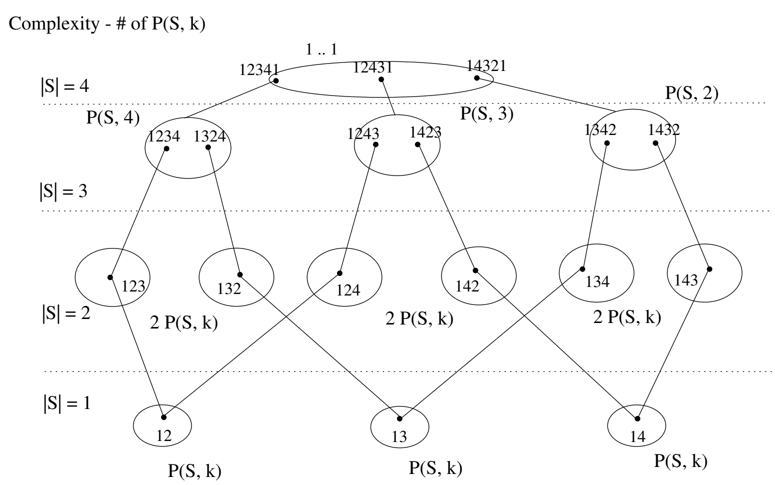
By taking the derivative on both sides we obtain:

$$(n-1)(1+x)^{n-2} = \sum_{j=0}^{n-1} j\binom{n-1}{j} x^{j-1} = \sum_{j=1}^{n-1} j\binom{n-1}{j} x^{j-1}$$

Let $x = 1$:

$$(n-1)2^{n-2} = \sum_{j=1}^{n-1} j\binom{n-1}{j}$$

So the total number of $P(S, k) = (n-1)2^{n-2}$, therefore the space complexity of the dynamic programming algorithm for the TRAVELING SALESPERSON PROBLEM is $\Theta(n2^n)$.



Time Complexity

For the space complexity, we computed the number of $P(S,k)$ to be calculated. For computing the time complexity of the dynamic programming algorithm we need to compute the time required to calculate all $P(S,k)$. The time required to calculate one $P(S,k)$ is $c \cdot (j-1)$ where $|S| = j$. But there are $\sum_{j=1}^{n-1} j\binom{n-1}{j}$ $P(S,k)$'s. So the total time taken is

$$\sum_{j=1}^{n-1} \left\{ j\binom{n-1}{j} [c \cdot (j-1)] \right\} = c\sum_{j=1}^{n-1} j(j-1)\binom{n-1}{j}$$

As was the case with the space complexity computation, use

$$(1+x)^{n-1} = \sum_{j=0}^{n-1} \binom{n-1}{j} x^j$$

Differentiate both sides with respect to x:

$$(n-1)(1+x)^{n-2} = \sum_{j=0}^{n-1} j\binom{n-1}{j} x^{j-1}$$

one more differentiation gives

$$(n-1)(n-2)(1+x)^{n-3} = \sum_{j=1}^{n-1} j(j-1)\binom{n-1}{j} x^{j-2}$$

And again let $x = 1$ to get

$$(n-1)(n-2)2^{n-3} = \sum_{j=1}^{n-1} j(j-1)\binom{n-1}{j}$$

So the total time is $c(n-1)(n-2)2^{n-3}$ for some constant c. Therefore the time complexity for the dynamic programming algorithm for the TRAVELING SALESPERSON PROBLEM is $\Theta(n^2 2^n)$.

Obviously $n^2 2^n$ grows very rapidly. Nevertheless, it can be shown that $n!$ is $\Theta(n^{n+\frac{1}{2}} e^{-n})$ [Stirling's approximation of $n!$] which is a much higher growth rate than $n^2 2^n$.
So the dynamic programming, even though exponential, is a significant improvement over an exhaustive enumeration.

n	exhaustive enumeration time: $\frac{n!}{2}$	dynamic programming space: $n2^n$	dynamic programming time: $n^2 2^n$
5	60	160	800
10	1,814,400	20,240	102,400
15	$6.55 \cdot 10^{11}$	491,520	7,372,800
20	$1.17 \cdot 10^{8}$	20,971,520	419,430,400

The number of hamiltonian circuits is $\frac{n!}{2}$ since we can proceed around each circuit in either of two directions. Note that for instance

$$20^2(2^{20}) \text{ microseconds} \quad < \quad 7 \text{ minutes}$$

but

$$\frac{20!}{2} \text{ microseconds} \quad > \quad 38{,}000 \textbf{ years!}$$

Next we consider the string alignment problem, that has many applications, especially in molecular biology, and show how dynamic programming can be used to solve the problem [CLR90].

4.4 The String Alignment Problem

A string is a sequence of characters from some alphabet.
Given two Strings S and T, how similar are they?
To answer this question we need to define a good "alignment" function between S and T.

EXAMPLE: $S = acdbcdbc$ and $T = bcdbcbb$ A possible alignment may be

a	c	d	b	c	d	b	c
b	c	d	b	c	-	b	b

where the special character "-" represents an insertion of a space. As for the alignment function, each column receives a certain value and the total score for the alignment is the sum of the values assigned to its columns. A column $\binom{x}{y}$ receives the value

- $+1$ if $x = y$, i.e. we have a match
- -1 if $x \neq y$, i.e. we have a mismatch
- -2 if $x = -$ or $y = -$, i.e. we inserted a space

So for the above example we obtain:

	a	c	d	b	c	d	b	c	
	b	c	d	b	c	-	b	b	
Score	-1	$+1$	$+1$	$+1$	$+1$	-2	$+1$	-1	$= +1$

Note that the alignment function rewards matches and penalizes mismatches and spaces. For any pairs of strings S and T and alignment function, there are many possible alignments.

Since the most popular applications of the string alignment problem are in the field of computational molecular biology, we start this section by giving a very brief introduction to molecular biology. The interested reader can find more information in the "Primer on Molecular Genetics" [CCS92] and in the introductory textbook on computational molecular biology [SM97].

4.4.1 Biology Background

All living organisms have a similar molecular chemistry (biochemistry). The main actors in the chemistry of life are molecules called:

proteins: which are responsible for what a living being is and does in a physical sense, and

nucleic acids: which encode the information necessary to produce proteins and are responding for passing along this recipe to subsequent generations

Understanding the structure and functions of the proteins and nucleic acids is the main focus of molecular biology.

Living organisms contain two kinds of nucleic acids:

a) ribonucleic acid (RNA)

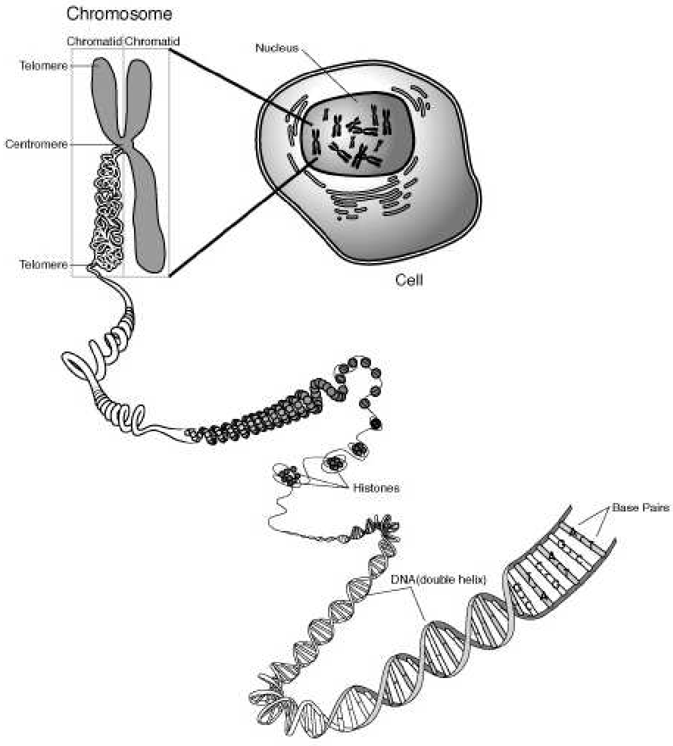


Figure 4.5: The human genome

b) deoxyribonucleic acid (DNA)

Besides the reproductive cell and red blood cell, every single cell in the human body contains the human genome: a complete set of instructions for making an organism. It contains the master blueprint for all cellular structures and activities for the lifetime of the cell or organism.

As can be seen in Figure 4.5, the human genome consists of tightly coiled threads of DNA and associated protein molecules, organized into structures called chromosomes. In other words, the complete set of chromosomes inside a cell is called a genome.

The number of chromosomes in a genome is characteristic of a species. Every cell in a human being has 23 pairs of chromosomes, while a mouse cell has 20 pairs. The human DNA contains about 3×10^9 base pairs (A-T or C-G). Each DNA strand consists of repeating nucleotide units composed of a phosphate group, a sugar (deoxyribose) and a base (guanine, cytosine, thiamine or adenine denoted by G, C, T and A). The double helix is formed by connecting complementary nucleotides A-T and C-G on two strands with hydrogen bonds. See Figure 4.6.

The four nitrogenous bases of DNA are arranged along the sugar phosphate backbone in a particular order, known as the DNA sequence, encoding all genetic instructions for an organism.

Each time a cell divides into two daughter cells, its full genome is duplicated. For humans, the duplication occurs in the nucleus.

During the cell division, the DNA molecule unwinds and the bonds between the base pairs break, allowing the strands to separate. Each strand directs the synthesis of a complementary new strand, with free nucleotides matching up with

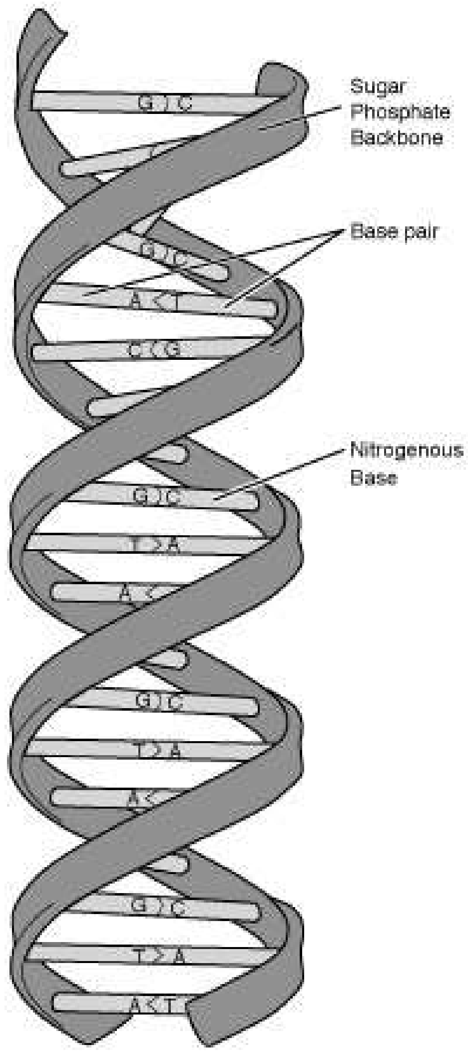


Figure 4.6: The DNA structure

their complementary bases on each of the separated strand.

A gene is a specific sequence of nucleotide bases along a chromosome carrying information for constructing a protein. Genes are part of the chromosomes.

In humans, genes consist of exons (protein-coding regions) and introns (noncoding: "junk data" regions). Approximately $5-10\%$ of the DNA is made up of exons while the rest are introns. See Figure 4.7.

The central dogma of molecular biology

See Figure 4.8.

DNA replication: In addition to being translated into proteins, the genetic information embodied in DNA can also be replicated into more DNA (as previously seen).

Transcription: The synthesis of an RNA copy from a sequence of DNA. The resulting RNA is the mRNA (messenger RNA).

Translation: The process in which the genetic code carried by the mRNA directs the synthesis of proteins.

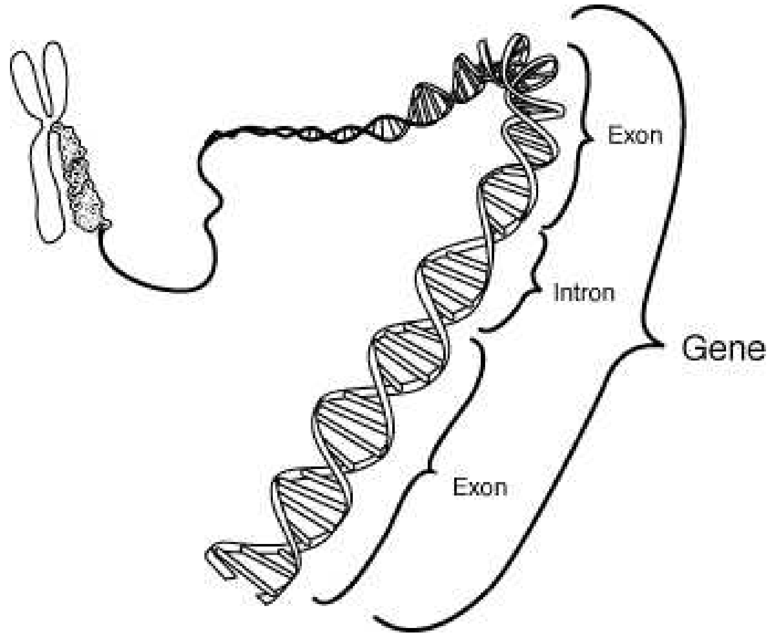


Figure 4.7: The gene of a DNA molecule

The human genome project

The human genome project is a multinational effort, begun by the USA in 1988, whose aim is to produce a complete physical map of all human chromosomes, as well as the entire human DNA sequence. As part of the project, genomes of other organisms such as bacteria, yeast, flies and mice are also being studied.

More specifically, the primary goal of the project is to make a series of descriptive diagrams (called maps) of each human chromosome at increasingly finer resolutions. Mapping involves

1. dividing the chromosomes into smaller fragments that can be propagated and characterized

2. ordering (mapping) them to correspond to their respective locations on the chromosomes.

The ultimate goal of genome research is to find all the genes in the DNA sequence and to develop tools for using this information in the study of human biology and medicine.

So far many virus genomes have been entirely sequenced, but their sizes are generally in the 1000 to $10,000$ base pair range. In 1966 the whole sequence of the yeast genome, which has $10,000,000$ base pairs, was mapped.

Mapping the human genome is still remote, as it is 100 times bigger than the largest genome sequenced so far. One of the reasons for which other organisms are part of the project is to perfect sequencing methods so that they can be applied to the human genome. A human chromosome has around 10^8 base pairs. But the largest pieces of DNA that can be sequenced in the laboratory are 700 base pairs long. The gap between what we can actually sequence and a chromosome size is at the heart of many problems in computational biology.

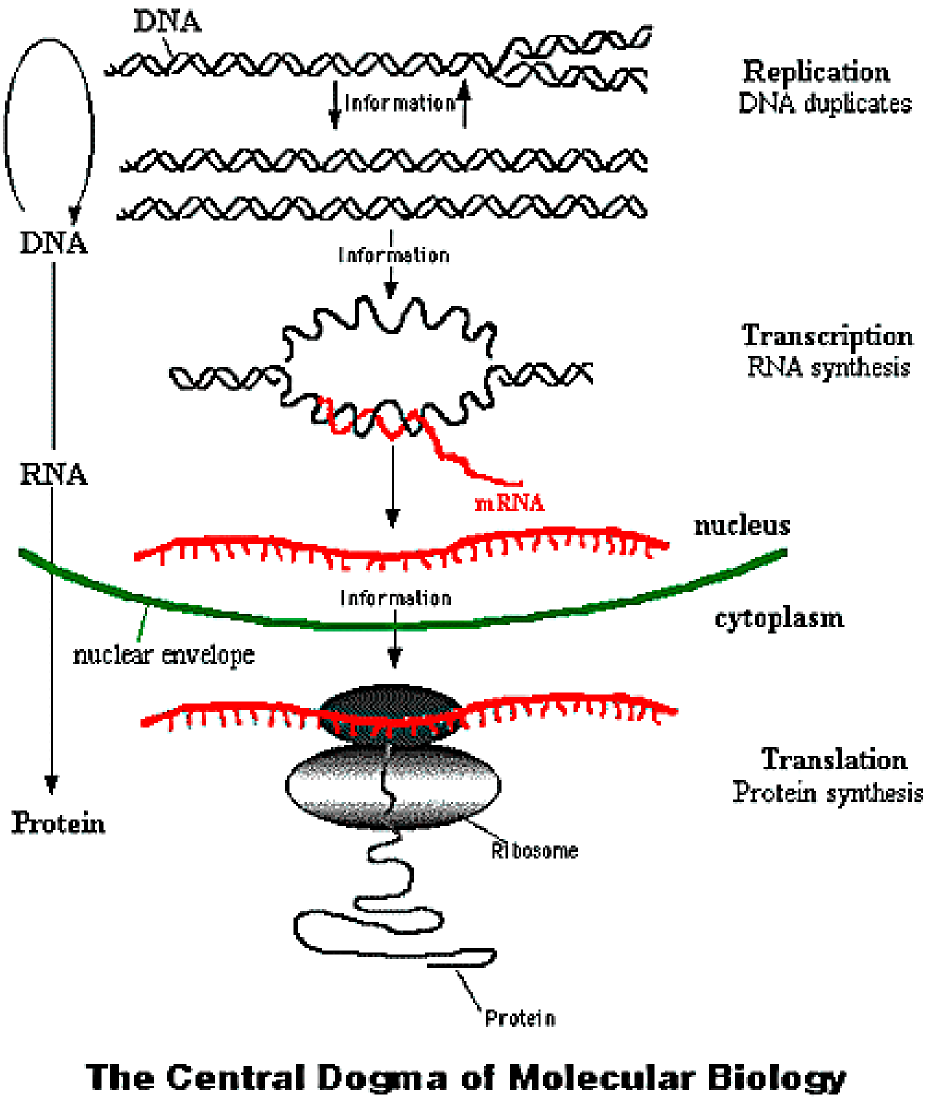


Figure 4.8:

Biologically motivated problems

The following are examples of biologically motivated problems that involve strings:

a) Reconstructing long DNA sequences from overlapping string fragments.

b) Determining physical and genetic maps from probe data under various experiments' protocols.

c) Storing, retrieving and comparing DNA strings.

d) Comparing two or more strings for similarities

e) Searching databases for related strings and substrings.

f) Exploring frequently occurring patterns of nucleotides.

g) Finding informative elements in protein and DNA sequences.

The aim of these problems is to learn about the functionality and/or the structure of protein without actually having to physically construct the protein itself. The research is based on the correct assumption that similar sequences produce similar proteins.

More specifically, the string alignment problem is found in the following problems:

a) Given two sequences with small differences (ex. insertions, deletions, substitutions or transpositions), find where the differences are.

   On average, the sequences are $10,000$ characters with differences every 100 characters.

   For example, we may want to compare the same gene sequenced by two different laboratories or check for errors in data that has been typed in multiple times.

b) Given two sequences, determine if there is a prefix of one which is a suffix of the other, and if so, find it.

   Typically, we want at least 15 characters of overleap and at least 80% of equal characters. We may want to answer the same question for multiple sequences where we know that many are unrelated.

c) Given two sequences, are there two substrings, one from each sequence, that are similar?

   We may want to answer the same question, comparing one sequence to thousands of others.

   This is useful for searching large biosequence databases for local similarities.

d) Given two sequences from the same family with evolutionary differences, find where the similarities occur.

The string alignment problem which consists in finding the best alignment between two strings while allowing certain mismatches between them, can be solved by using dynamic programming. Thus all the above problems can consequently be solved by dynamic programming.

4.4.2 The dynamic programming solution to the String Alignment Problem

The dynamic programming solves an instance of the String Alignment Problem by taking advantage of already computed solutions for smaller instances of the same problem.

Given two sequences, S and T, instead of determining the similarity between S and T as whole sequences only, the dynamic programming builds up the final solution by determining all similarities between arbitrary prefixes of S and T. The dynamic programming starts with shorter prefixes and uses previously computed results to solve the problem for large prefixes until it finally finds the solution for S and T.

We use the alignment function previously seen ($+1$ for a match, -1 for each mismatch and -2 for each space).

Given two strings S and T over some alphabet, with $|S| = n$ and $|T| = m$. Define $A(i,j)$ to be the value of an optimal alignment of strings:

$S[1], S[2], \ldots, S[i]$ and
$T[1], T[2], \ldots\ldots, T[j]$

So $A(n,m)$ is the value of an optimal alignment of S and T. The dynamic programming algorithm will compute each $A(i,j)$, $0 \le i \le n$ and $1 \le j \le m$ only **once** in a relatively simple way, by considering the values already computed for smaller indexes i and j. Define

$$A(i,0) = \sum_{k=1}^{i} v(S[k], -)$$

and

$$A(0,j) = \sum_{k=1}^{j} v(-, T[k])$$

where v is the alignment function. $A(i,0)$ means that the first i characters of S are aligned with no characters of T. In other words, the i characters of S are matched with i spaces (i.e. "-").Similarly for $A(0,j)$. In general:

$$A(i,j) = \max \begin{cases} A(i-1,j-1) + v(S[i],T[j]) \\ A(i-1,j) + v(S[i],-) \\ A(i,j-1) + v(-,T[j]) \end{cases}$$

Recall:

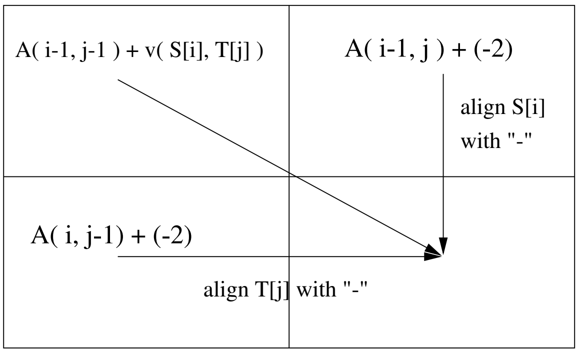
$$v(S[i],T[j]) = \begin{cases} +1 \text{ if } & S[i] = T[j] \\ -1 \text{ if } & S[i] \neq T[j] \end{cases}$$

and

$$v(S[i],-) = v(-,T[j]) = -2.$$

The dynamic programming algorithm uses a "matrix representation" of size $(n+1) \times (m+1)$. Note that $A(i,j)$ corresponds to the optimal alignment of the i^{th} prefix of S with the j^{th} prefix of T. The dynamic programming algorithm fills in the entries of the matrix by computing the values of $A(i,j)$ from top to bottom, left to right. The value of the optimal alignment is $A(n,m)$.

The various lines crossing the entries in the matrix show from which entry in the matrix we derived the maximum score for a given entry $A(i,j)$.



To determine the solution of the optimal alignment, simply retrace the steps from entry (n,m) to entry $(0,0)$ (see Figures 4.9 to 4.12).

The dynamic programming algorithm computes an optimal alignment in time $O(nm)$ since it needs constant time for each of the $(n+1)\times(m+1)$ entries (same for space complexity). Reconstructing a single alignment can be done in time $O(n+m)$.

The major drawback of dynamic programming is the fact that the matrix (or table), of size $(n+1)\times(m+1)$, uses $O(nm)$ space. It is easy to compute $A(n,m)$ in linear space since all we have to do at any given time during the computation is save two rows of the matrix, not more. The only values needed when computing $A(i,j)$ are found in rows i and $i-1$. But it is not easy to find the optimal alignment

		0	1	2	3	4	5	6	7
	i \ j	-	b	c	d	b	c	b	b
0	-	0	-2	-4	-6	-8	-10	-12	-14
1	a	-2	-1	-3	-5	-7	-9	-11	-13
2	c	-4	-3	0	-2	-4	-6	-8	-10
3	d	-6	-5	-2	1	-1	-3	-5	-7
4	b	-8	-5	-4	-1	2	0	-2	-4
5	c	-10	-7	-4	-3	0	3	1	-1
6	d	-12	-9	-6	-3	-2	1	2	0
7	b	-14	-11	-8	-5	-2	-1	2	3
8	c	-16	-13	-10	-7	-4	-1	0	1

Figure 4.9: The table for $S = acdbcdbc$ and $T = bcdbcbb$

in linear space. Ideally we would like to find methods that reduce the use of space without substantially increasing time requirements.

Hirschberg developed an elegant and practical space-reduction method that can be used with many dynamic programming problems. His method reduces the required space from $O(nm)$ to $O(m)$ while only doubling the worst-case time bound. What follows can be found in Chapter 12 of [Gus97].

Let $A^R(i,j)$ denote an optimal alignment value of the suffix of size i of S with the suffix of size j of T.

Theorem 4.4.1 $A(n,m) = \max_{0\leq k\leq m}[A(\frac{n}{2},k) + A^R(\frac{n}{2},m-k)]$.

Proof:

a) *Show that* $\max_{0\leq k\leq m}[A(\frac{n}{2},k) + A^R(\frac{n}{2},m-k)] \leq A(n,m)$.
For any fixed position k' in T, we can find an alignment of S and T consisting of

$$S_1S_2\ldots S_{\frac{n}{2}} \text{ and } T_1T_2\ldots T_{k'}$$

followed by another (disjoint) alignment of

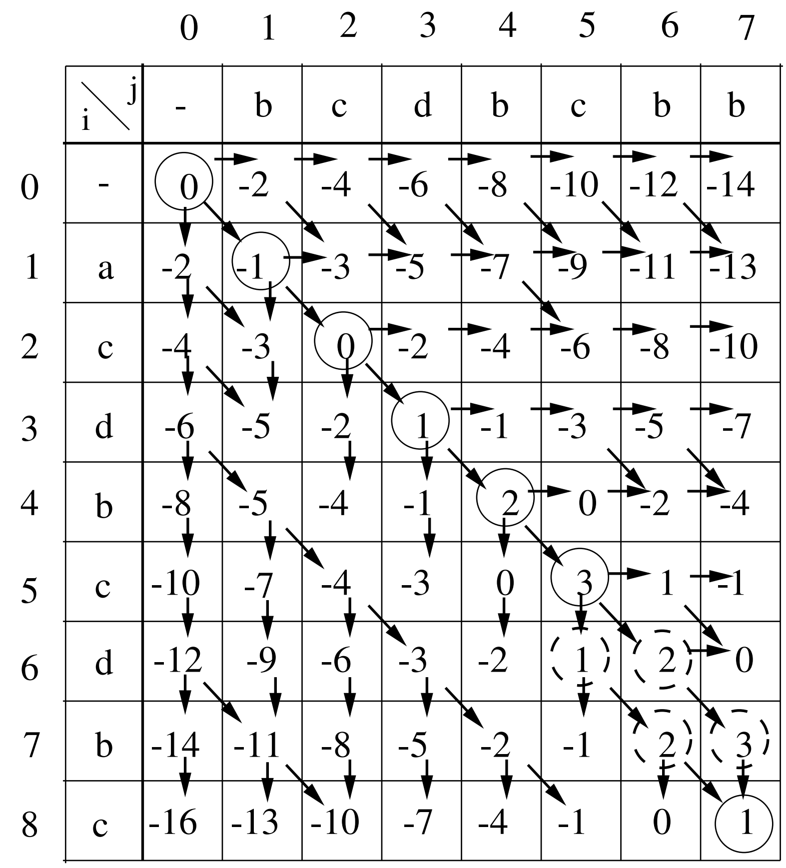
$$S_{\frac{n}{2}+1}S_{\frac{n}{2}+2}\ldots S_n \text{ and } T_{k'+1}T_{k'+2}\ldots T_m.$$

By definition of $A(i,j)$ and $A^R(i,j)$, the best alignment of

$$S_1\ldots S_{\frac{n}{2}} \text{ and } T_1\ldots T_{k'}$$

has value $A(\frac{n}{2},k')$. $A^R(\frac{n}{2},m-k')$ is the value of the best alignment of

$$S_{\frac{n}{2}+1}\ldots S_n \text{ and } T_{k'+1}\ldots T_m.$$



a c d b c d b c
b c d b c b b -

a c d b c d b c
b c d b c - b b

Figure 4.10: There are two optimal solutions

The combined alignment has value

$$A(\frac{n}{2}, k') + A^R(\frac{n}{2}, m - k')$$

which is less than or equal to $A(n,m)$. So for **any** k' we have

$$A(\frac{n}{2}, k') + A^R(\frac{n}{2}, m - k') \leq A(n,m)$$

Therefore

$$\max_{0 \leq k \leq m} (A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m - k)) \leq A(n,m)$$

b) *Show that* $A(n,m) \leq \max_{0 \leq k \leq m}(A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m-k))$.

Consider an optimal alignment of S and T. Let k' be the rightmost position in T that is aligned with a character at or before position $\frac{n}{2}$ in S. Then the optimal alignment of S and T consists of an alignment of

$$S_1 S_2 \ldots S_{\frac{n}{2}} \text{ and } T_1 T_2 \ldots T_{k'}$$

followed by an alignment of

$$S_{\frac{n}{2}} S_{\frac{n}{2}+1} \ldots S_n \text{ and } T_{k'+1} T_{k'+2} \ldots T_m.$$

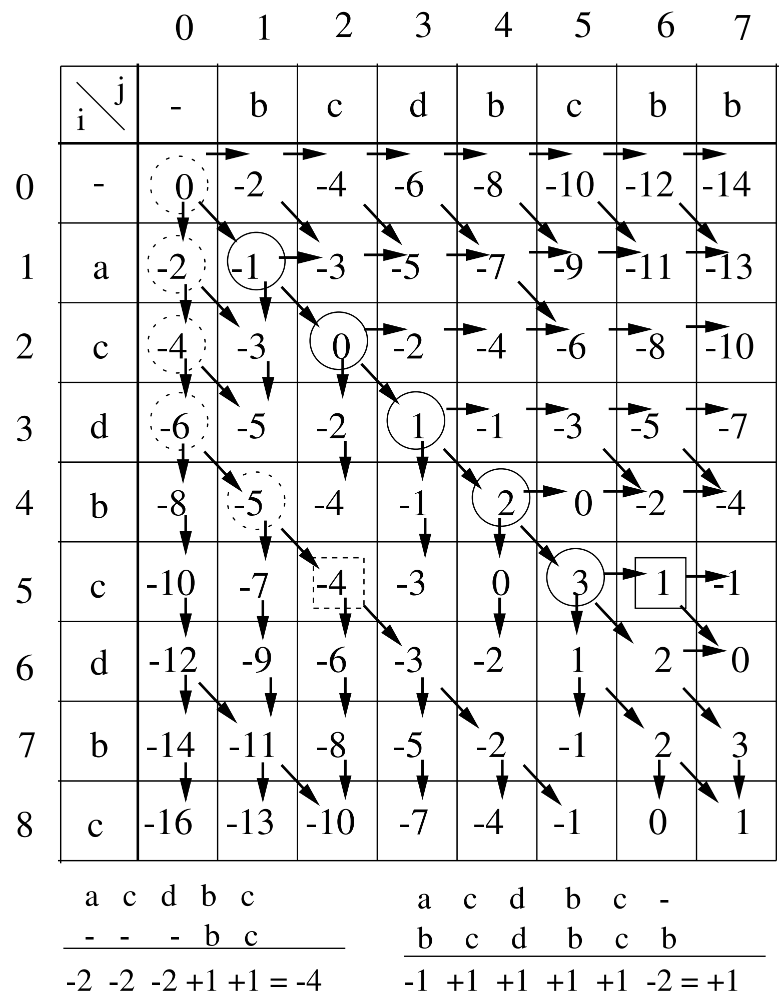


Figure 4.11: Solutions for $S = acdbc$, $T = bc$ and $S = acdbc$, $T = bcdbcb$

Let p denote the value of the alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$, and q denote the value of the alignment of $S_{\frac{n}{2}+1} \dots S_n$ and $T_{k'+1} \dots T_m$. Then p must be equal to $A(\frac{n}{2}, k')$, for if $p < A(\frac{n}{2}, k')$ then we can replace the alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$ with the alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$ that has value of $A(\frac{n}{2}, k')$. But that is impossible since it would create an alignment of S and T whose value is larger than the claimed optimal. So

$$p = A(\frac{n}{2}, k')$$

and by a similar reasoning we have:

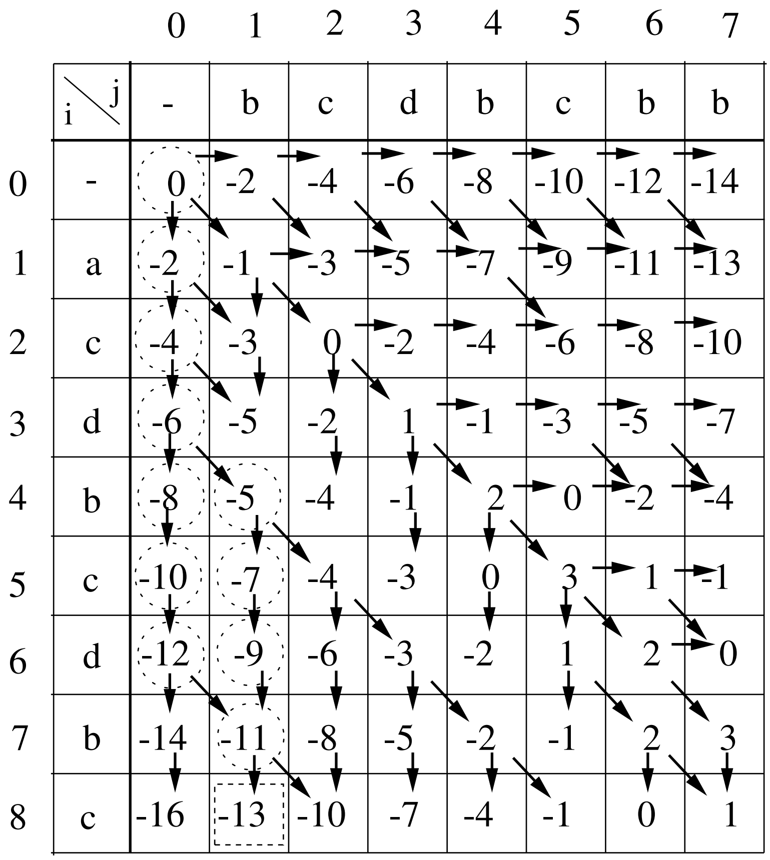
$$q = A^R(\frac{n}{2}, m - k').$$

So, $A(n,m) = A(\frac{n}{2}, k') + A^R(\frac{n}{2}, m - k')$ and

$$A(\frac{n}{2}, k') + A^R(\frac{n}{2}, m - k') \leq \max_{0 \leq k \leq m} (A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m - k))$$

Therefore

$$A(n,m) \leq \max_{0 \leq k \leq m} (A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m - k))$$



a c d b c d b c
\- - - - - - b -
7(-2) + 1 = -13

a c d b c d b c
\- - - b - - - -
7(-2) + 1 = -13

Figure 4.12: Two optimal solutions for $S = acdbcdbc$ and $T = b$

a) and b) $\Rightarrow$

$$A(n,m) = \max_{0 \le k \le m} (A(\frac{n}{2},k) + A^R(\frac{n}{2}, m-k))$$

$\square$

Let k^* be a position k that maximizes

$$A(\frac{n}{2},k) + A^R(\frac{n}{2}, m-k).$$

By the theorem we just proved, there is an optimal alignment of S and T such that if we trace back the alignment in the $(n+1) \times (m+1)$ matrix, the path i from (n,m) to $(0,0)$ will go through entry $(\frac{n}{2}, k^*)$ in the matrix.

Let $L_{\frac{n}{2}}$ be the subpath of L that starts with the last node of L in row $\frac{n}{2} - 1$ and ends with the first node of L in row $\frac{n}{2} + 1$.

Theorem 4.4.2 *A position k^* in row $\frac{n}{2}$ can be found in $O(nm)$ time and $O(m)$ space. Additionally, a subpath $L_{\frac{n}{2}}$ can be found in $O(nm)$ time and stored in $O(m)$ space.*

Proof:

a) Use the dynamic programming algorithm to compute the optimal alignment of S and T until we hit row $\frac{n}{2}$. When processing row $\frac{n}{2}$, save the traceback pointers for the entries in that row. At this point, $A(\frac{n}{2}, k)$ is known for every position k, $0 \leq k \leq m$, of T. Note that only $O(m)$ space is needed to save the traceback pointers and values of that row.

b) Use the dynamic programming algorithm again to compute the optimal alignment of S^R and T^R and stop at row $\frac{n}{2}$. S^R and T^R are the reverse strings of S and T, respectively. Once again, save the traceback pointers and values of each entry of that row. Again, only $O(m)$ space is needed for storing the values and pointers, and $A^R(\frac{n}{2}, m-k)$ is known for each k. Add $A(\frac{n}{2}, k)$ to $A^R(\frac{n}{2}, m-k)$ for each k, and let k^* be the index that gives the largest sum.

Note that the addition and comparisons take $O(m)$ time. We use the first set of saved traceback pointers to go from entry $(\frac{n}{2}, k^*)$ to an entry $(\frac{n}{2}-1, k_1)$ in row $\frac{n}{2}-1$. This identifies a subpath that is on an optimal path from $(0,0)$ to $(\frac{n}{2}, k^*)$.
Similarly. we use the second set of saved traceback pointers to go from $(\frac{n}{2}, k^*)$ to entry $\frac{n}{2}+1, k_2)$ in row $\frac{n}{2}+1$. Once again, that path identifies a subpath of an optimal path from $(\frac{n}{2}, k^*)$ to (n, m). The two subpaths from $(\frac{n}{2}-1, k_1)$ to $(\frac{n}{2}, k^*)$ and from $(\frac{n}{2}, k^*)$ to $(\frac{n}{2}+1, k_2)$ form the subpath $L_{\frac{n}{2}}$ which is part of an optimal path L (that passes through $(\frac{n}{2}, k^*)$ and goes from $(0,0)$ to (n,m)). Computing and finding k^*, k_1, k_2 and $L_{\frac{n}{2}}$ is done in $O(nm)$ time and $O(m)$ space. □

The problem of finding the optimal alignment of S and T is reduced to two smaller alignment problems:

- one for strings $S_1 S_2 \dots S_{\frac{n}{2}-1}$ and $T_1 T_2 \dots T_{k_1}$ and
- one for strings $S_{\frac{n}{2}+1} S_{\frac{n}{2}+2} \dots S_n$ and $T_{k_2} T_{k_2+1} \dots T_m$.

Aligning $S_1 \dots S_{\frac{n}{2}-1}$ and $T_1 \dots T_{k_1}$ is called the top problem (see section X of Figure 4.13). Aligning $S_{\frac{n}{2}+1} \dots S_n$ and $T_{k_2} \dots T_m$ is called the bottom problem (see section Y of Figure 4.13), the rest of the matrix can be ignored.

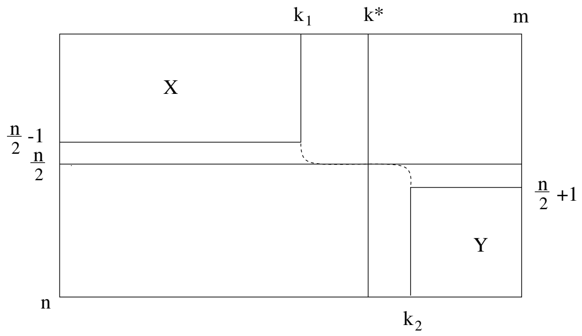


Figure 4.13:

We can recursively determine the middle row of X (or Y) in time proportional to the total size of X (or Y). So the middle row in the top problem can be determined in at most $\frac{ck^*n}{2}$ time, and likewise, the middle row in the bottom problem can be determined in at most $\frac{c(m-k^*)n}{2}$. The sum of these two bounds is $\frac{cnm}{2}$.
We are now in a position to discuss Hirschberg's linear-space optimal alignment algorithm.

4.4.3 Hirschberg's linear-space optimal alignment algorithm

As seen previously, we first reduce the original n by m alignment problem to two smaller alignment problems: the top and bottom subproblems. It can be done in $O(nm)$ time and $O(m)$ space. Then, we solve the top and bottom problems by a recursive application of this reduction. We have two alignment problems, one on a matrix of size at most $\frac{n}{2}$ by k^* and another on a matrix of size at most $\frac{n}{2}$ by $m - k^*$. We can therefore find the best column in the middle row of each of the two subproblems in at most

$$\frac{cnk^*}{2} + \frac{cn(m - k^*)}{2} = \frac{cnm}{2} \text{ time,}$$

and recurse from there with four subproblems.

Continuing in this recursive way, we can find an optimal alignment of S and T with $\log_2 n$ levels of recursion always using at most $O(m)$ space. At each recursive call, we find and store a subpath of an optimal path L. Since the subpaths are edge disjoint, the total length is $O(n + m)$. Procedure OPTA takes as parameters the starting and ending indexes of the matrix, i.e. OPTA$(1, n, 1, m)$ for the first call.

It processes the top subproblem first, then the bottom one. The subpath is output between the two OPTA calls. The concatenation of the various outputs for the subpaths will give an optimal path L from matrix entry $(0, 0)$ to (n, m).

In other words, the concatenation of the subpaths will give an optimal alignment of S and T.

All the main steps of OPTA have already been explained in the previous paragraphs of this section.

OPTA(l, l', r, r')

---

begin
$k := \frac{l'+l}{2}$
In $O(r' - r) = O(m)$ space, find an index k^* between
l and l', inclusively, such that there is an optimal
alignment of $S_l \dots S_{l'}$ and $T_r \dots T_{r'}$
consisting of an optimal alignment of $S_l \dots S_k$
and $T_r \dots T_{k^*}$ followed by an optimal alignment
of $S_{k+1} \dots S_{l'}$ and $T_{k^*+1} \dots T_{r'}$.
Also find and store subpath L_k that is part of an
optimal path L' from entry (l, r) to (l', r') and that
begins with the last entry k_1 on L' in row $k - 1$ and
ends with the first entry k_2 on L' in row $k + 1$.
All this is done as described earlier.
OPTA$(l, k - 1, r, k_1)$ *new top problem*
output L_k
OPTA$(k + 1, l', k_2, r')$ *new bottom problem*
end

Theorem 4.4.3 *Using Hirschberg's procedure* OPTA*, an optimal alignment of two strings of length n and m can be found in $O(nm)$ time and $O(m)$ space.*

Proof:
We have already seen that the first level of recursion uses cnm time and that the second level of recursion uses at most $\frac{cnm}{2}$ time. At the i^{th} level of recursion, we have 2^{i-1} subproblems, each of which has $\frac{n}{2^{i-1}}$ rows and a variable number of columns. Since the columns in the subproblems (top and bottom) are distinct, the

total size of all problems is at most the total number of columns m times $\frac{n}{2^{i-1}}$. In other words, the total size $\leq \frac{mn}{2^{i-1}}$. The total time used at the i^{th} level of recursion is at most $\frac{cnm}{2^{i-1}}$. If n is a power of 2, we have total time

$$= \sum_{i=1}^{\log_2 n} \frac{cnm}{2^{i-1}} \leq 2cnm.$$

Even if n is not a power of 2, we still have that total time is at most $2cnm$. $\square$

Conclusion: The time complexity of Opta is $O(nm)$ (but Opta performs twice as many computations as the original dynamic programming algorithm), the space complexity of Opta is $O(m)$ (which is better than $O(nm)$ of the original dynamic programming algorithm).

In the next section, we show how dynamic programming can be used to solve another string-based problem called the edit distance problem [CLR90].

4.5 The Edit Distance Problem

When a smart terminal updates a line of text, replacing an existing "source" string with a new "target" string, there are several ways in which the changes can be made.

A single character of the source string can be deleted, replaced by another character, or copied to the target string; characters can be inserted; or two adjacent characters of the source string can be interchanged ("twiddled") while being copied to the target string. After all the other operations have occurred, an entire suffix of the source string can be deleted, an operation know as "kill to eol".

Assume we have three operations only:

character insertion

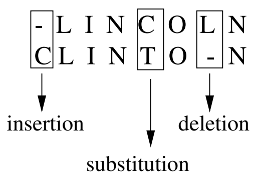
character deletion and

character substitution

Given two sequences X and Y and the three operations, the edit distance from X to Y is the cost of the least expensive transformation sequence that converts X to Y.

We use dynamic programming to find the edit distance from X to Y and print an optimal transformation sequence.

Example:



The edit distance between LINCOLN and CLINTON is 3.

Suppose that X has n characters and Y has m characters. $E(i,j)$ is the least edit distance between the first i characters of X and the first j characters of Y. So the edit distance between X and Y is given by $E(n,m)$.

$E(i,0) = i$	since we need i deletions to convert a string with i characters to the empty string
$E(0,j) = j$	since we need j insertions to convert an empty string into a j - character string

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

```
AL-TRUISTIC              ALTRUISTIC
ALGOR-I-THM      or      ALGORI-THM
```

Figure 4.14: Edit distance between ALTRUISTIC and ALGORITHM

In general

$$E(i,j) = min \begin{cases} E(i-1,j)+1 & \text{deletion} \\ E(i,j-1)+1 & \text{insertion} \\ E(i-1,j-1) + \begin{cases} 0 \text{ if } & X[i] = Y[j] \\ 1 \text{ if } & X[i] \neq Y[j] \end{cases} & \text{substitution} \end{cases}$$

Draw an $(n+1) \times (m+1)$ matrix by filling it, top to bottom, left to right. We could fill the first row and first column first (as in the procedure) then proceed row by row to fill in the remaining entries of the matrix.

```
EditDistance(X, Y)
begin
for i := 1 to n
      E(i,0) := i
for i := 1 to m
      E(0,j) := j
for i := 1 to n
      for j := 1 to m
            if X[i] = Y[i]
                  then E(i,j) = min { E(i-1,j) + 1
                                      E(i,j-1) + 1
                                      E(i-1,j-1)
                  else E(i,j) = min { E(i-1,j) + 1
                                      E(i,j-1) + 1
                                      E(i-1,j-1) + 1
end
```

The matrix shown in Figure 4.14 gives the edit distance between ALTRUISTIC and ALGORITHM [CLR90]. There are $O(nm)$ entries in the matrix and the computation of each takes $O(1)$ time. So the total running time (and also space) is $O(nm)$.
In the next section, we deviate from string problems and tackle the single-source single-sink shortest path problem in layered networks that can be found in [PS82].

4.6 The Single-source Single-sink Shortest Path Problem in Layered Networks

In the single-source single-sink shortest path problem in layered networks we are given a weighted, directed, connected graph with two special vertices: the source vertex and the sink vertex. Moreover, the vertices are ordered in layers with the source in the first layer and the sink, in the last layer. The direction of the edges is always from the first layer to the last. We would like to find the cheapest cost of going from the source to the sink.
The following is a formal definition of the problem.

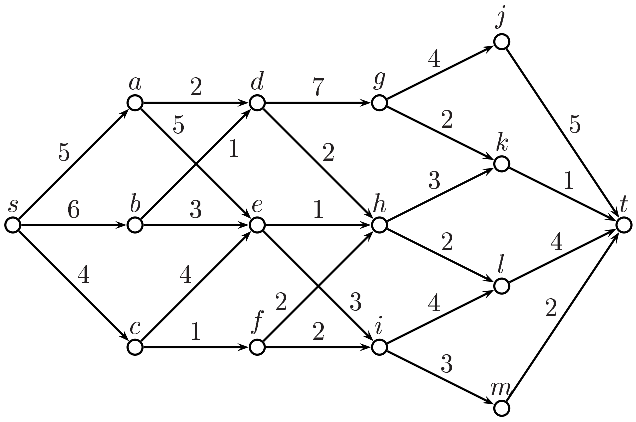
Problem instance:
A weighted graph (positive weights) with source vertex s and sink vertex t.

Feasible solution:
A path from s to t.

Objective function:
For each feasible solution, associate the total weight of the path.

Optimal solution:
A minimum path from s to t.

Example:



table(i): The optimal way to continue a shortest path when we are in the i^{th} layer from t, i.e. table(i) contains the best decision for each node in the layer i arcs from t.

node	j	k	l	m
next node	t	t	t	t
total cost	5	1	4	2

table(1)

node	g	h	i
next node	k	k	m
total cost	3	4	5

table(2)

node	d	e	f
next node	h	h	h
total cost	6	5	6

table(3)

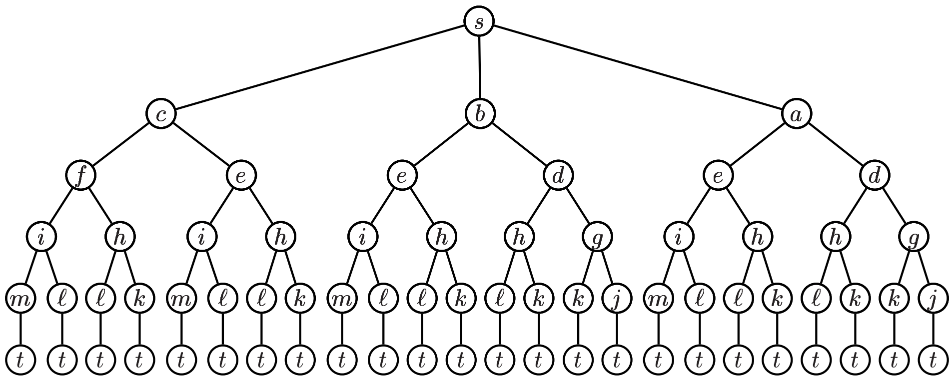
node	a	b	c
next node	d	d	f
total cost	8	7	7

table(4)

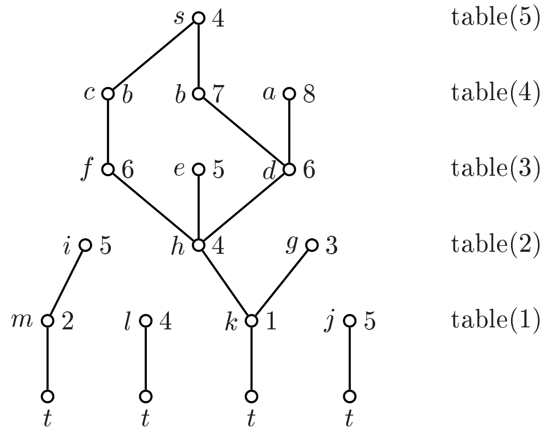
node	s
next node	c
total cost	11

table(5)

We can reconstruct an optimal path by backtracking through tables 5, 4, 3, 2, and 1 to obtain path s, c, f, h, k, t with cost 11. Note that if we use a top-down approach by drawing a tree of all paths from the root node s to leaf t, we would be enumerating the same subtree more than once (see the following figure).



With the bottom-up dynamic programming approach, each subtree is computed only **once**.



Remarks:

- We note that "all paths" were not enumerated, for instance, the trajectory along the top was **not** evaluated.

- Whenever we were determining the best choice of an edge by which to leave any particular node, we already knew the best choice to make at all nodes to the right of the one under consideration.

In the next section, we consider another graph problem that has many real-world applications: the all pairs shortest path problem [CLR90].

4.7 The All Pairs Shortest Path Problem

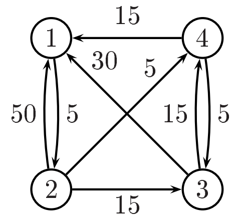
Given a weighted graph, we want to compute the shortest path between each pair of nodes. The weights of the edges are nonnegative. The vertices are numbered from 1 to n, and matrix L gives the length (weight) of each edge, with $L(i,j) = \infty$ if edge (i,j) does not exist, $L(i,i) = 0$ and $L(i,j) = t$ if t is the weight of edge (i,j).

The principle of optimality applies here: If k is a node on the shortest path from i to j, then that part of the path from i to k, and that from k to j, must also be optimal. Construct a matrix D that gives the length of the shortest path between each pair of nodes. Initially, D is equal to L. At the k^{th} iteration, D gives the length of the shortest paths that only use nodes in $\{1, 2, \ldots, k\}$ as intermediate nodes [BB88]. One has to check for each pair of nodes and see if there exists a path passing through node k that is shorter than the present optimal path passing only through nodes in $\{1, 2, \ldots, k-1\}$. If D_k denotes the matrix after the k^{th} iteration, then

$$D_k(i,j) = \min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$$

```
Floyd(L)
  begin
  D ← L
  for k := 1 to n
    for i := 1 to n
      for j := 1 to n
         D(i,j) := min(D(i,j), D(i,k) + D(k,j))
  return D
  end
```

Consider the following EXAMPLE found in [BB88]:



$$D_0 = L = \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \qquad D_1 = \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \mathbf{35} & 0 & 15 \\ 15 & \mathbf{20} & 5 & 0 \end{bmatrix}$$

$$D_2 = \begin{bmatrix} 0 & 5 & \mathbf{20} & \mathbf{10} \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \qquad D_3 = \begin{bmatrix} 0 & 5 & 20 & 10 \\ \mathbf{45} & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$

$$D_4 = \begin{bmatrix} 0 & 5 & \mathbf{15} & 10 \\ \mathbf{20} & 0 & \mathbf{10} & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$

Procedure FLOYD's running time is $\Theta(n^3)$. D_n will give the value of the shortest path for all pairs of vertices. To get the actual shortest paths, we could add the following code:

> **if** $D(i,k) + D(k,j) < D(i,j)$
>   **then** $D(i,j) := D(i,k) + D(k,j)$
> $P(i,j) := k$

to the innermost loop of procedure FLOYD. P is an $n \times n$ matrix initialized to 0. At the end of the procedure's execution, $P(i,j)$ contains the number of the last iteration that caused a change in $D(i,j)$. Look at $P(i,j)$ to recover the shortest path from i to j. If $P(i,j) = 0$, the shortest path is directly along the edge (i,j) (since that entry in P has not changed since its initialization). If $P(i,j) = k$, the shortest path from i to j passes through k (the last time $P(i,j)$ was updated was at iteration k). Check out $P(i,k)$ and $P(k,j)$ to see if there are more intermediate nodes along the shortest path from node i to node j.

For the above example, P is

$$P = \begin{bmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

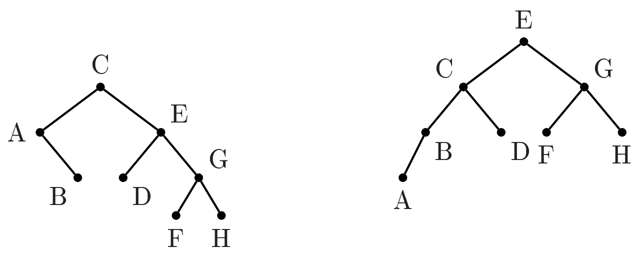
For vertices 1 and 3 we have $P(1,3) = 4$ so the shortest path from 1 to 3 passes through 4. Now we have to look at $P(1,4) = 2$ and $P(4,3) = 0$; so we have to check $P(1,2) = 0$ and $P(2,4) = 0$. Hence the shortest path from 1 to 3 is $1, 2, 4, 3$ and is of length $D_4(1,3) = 15$.

Next we consider the problem of building optimal search trees [Sti87].

4.8 The Optimal Search Tree Problem

The value contained in every internal node of a search tree is greater than or equal to the values contained in its left-hand descendants, and less than or equal to the values contained in its right-hand descendants.

We can have more than one search tree for a given set of keys



If all keys are sought with the same probability it takes

$$\frac{1 + 2(2) + 3(3) + 2(4)}{8} = \frac{22}{8} \text{ comparisons}$$

on the average to find a key in the left tree and

$$\frac{1 + 2(2) + 4(3) + 4}{8} = \frac{21}{8} \text{ comparisons}$$

on the average to find a key in the right tree.

Suppose we have a set of n distinct keys: $c_1 < c_2 < c_3 < \ldots < c_n$. Let p_i be the probability that a request refers to key c_i. Assume that all requests refer to keys that are present in the search tree. In other words,

$$\sum_{i=1}^{n} p_i = 1$$

Note that $d_i + 1$ comparisons are necessary to find a key in a node at depth d_i where the depth of the root is 0, its children are at depth 1, and so on. So the average number of comparisons needed to find a key in a given search tree is

$$C = \sum_{i=1}^{n} p_i(d_i + 1)$$

and we would like to minimize C.

Suppose that the $j - i + 1$ successive keys $c_i, c_{i+1}, \ldots, c_j$ occupy nodes in subtree U. If key $c_k, i \leq k \leq j$ is held in a node at depth d_k^* in U, the average number of comparisons carried out in U when looking for a key c_q (not necessarily in U) is:

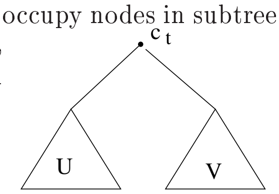


Figure 4.15:

$$C^* = \sum_{k=i}^{j} p_k(d_k^* + 1)$$

Note that:

a) this expression has the same form as the one for C, and

b) a change in the subtree does not affect the contribution made to C by other subtrees of the main tree disjoint from the one being considered.

By the previous remarks, we see that the principle of optimality applies here: In an optimal tree all the subtrees must also be optimal with respect to the keys they contain.

Let $m_{ij} = \sum_{k=i}^{j} p_k$, and c_{ij} be the average number of comparisons carried out in an optimal subtree containing keys $c_i, c_{i+1}, c_{i+2} \ldots, c_j$ when a key is sought in the main tree. Let $c_{ij} = 0$ for $j = i-1$.

One of these keys, k for example, must occupy the root node of the subtree. L is an optimal subtree containing $c_i, c_{i+1}, c_{i+2} \ldots, c_{k-1}$. R is an optimal subtree containing $c_{k+1}, c_{k+2}, \ldots, c_j$. When we look for a key in the main tree, the probability that it is in the sequence $c_i, c_{i+1}, \ldots, c_j$ is m_{ij}. In this case, one comparison is made with c_k, then depending on the result of the comparison, other comparisons may then be made in L or R. The average number of comparisons carried out is

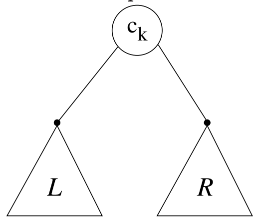


Figure 4.16:

$$c_{ij}^k = m_{ij} + c_{i\,k-1} + c_{k+1\,j}$$

where m_{ij} is the contribution of the root, $c_{i\,k-1}$ that of L and $c_{k+1\,j}$ that of R. So the recurrence relation is

$$c_{ij} = m_{ij} + \min_k \{c_{i\,k-1} + c_{k+1\,j}\}$$

Note that $c_{ii} = p_i$ and (by convention) $c_{i\,i-1} = 0$.
Consider the following example that can be found in [Sti87].

EXAMPLE: Find the optimal search tree for

i	1	2	3	4	5
keys	c_1	c_2	c_3	c_4	c_5
p_i	0.3	0.05	0.08	0.45	0.12

To solve this problem, first compute the m matrix where $m_{ij} = \sum_{t=i}^{j} p_t$.

$$m = \begin{bmatrix} 0.3 & 0.35 & 0.43 & 0.88 & 1.00 \\ & 0.05 & 0.13 & 0.58 & 0.70 \\ & & 0.08 & 0.53 & 0.65 \\ & & & 0.45 & 0.57 \\ & & & & 0.12 \end{bmatrix}$$

Second, we compute c_{ij} using

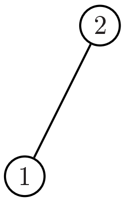
$$c_{ij} = m_{ij} + \min_k \{c_{i\,k-1} + c_{k+1\,j}\}$$

Note that $c_{ii} = p_i$ for $i = 1, 2, 3, 4, 5$. We can build a matrix for the dynamic programming algorithm.

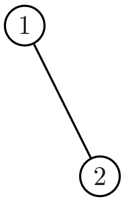
	1	2	3	4	5
1	0.3	0.4			
2		0.05			
3			0.08		
4				0.45	
5					0.12

$$
\begin{aligned}
c_{12} &= m_{12} + \min\{c_{10} + c_{22}, c_{11} + c_{32}\} \\
&= 0.35 + \min\{0.05, 0.3\} = 0.4
\end{aligned}
$$

The choices correspond to



and



Continue in a diagonal fashion:

$$
c_{23} = 0.18 \qquad c_{34} = 0.61 \qquad c_{45} = 0.69
$$

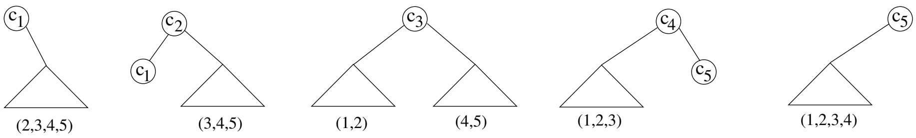
then

$$
\begin{aligned}
c_{13} &= m_{13} + \min\{c_{10} + c_{23}, c_{11} + c_{33}, c_{12} + c_{43}\} \\
&= 0.43 + \min\{0.18, 0.38, 0.4\} = 0.61 \\
c_{24} &= m_{24} + \min\{c_{21} + c_{34}, c_{22} + c_{44}, c_{23} + c_{54}\} \\
&= 0.58 + \min\{0.61, 0.5, 0.18\} = 0.76 \\
c_{35} &= m_{35} + \min\{c_{32} + c_{45}, c_{33} + c_{55}, c_{34} + c_{65}\} \\
&= 0.65 + \min\{0.69, 0.2, 0.61\} = 0.85 \\
c_{14} &= m_{14} + \min\{c_{10} + c_{24}, c_{11} + c_{34}, c_{12} + c_{44}, c_{13} + c_{54}\} \\
&= 0.88 + \min\{0.76, 0.91, 0.85, 0.61\} = 1.49 \\
c_{25} &= m_{25} + \min\{c_{21} + c_{35}, c_{22} + c_{45}, c_{23} + c_{55}, c_{24} + c_{65}\} \\
&= 0.7 + \min\{0.85, 0.74, 0.3, 0.76\} = 1.00
\end{aligned}
$$

and finally get

$$
\begin{aligned}
c_{15} &= m_{15} + \min\{c_{10} + c_{25}, c_{11} + c_{35}, c_{12} + c_{45}, c_{13} + c_{55}, c_{14} + c_{65}\} \\
&= 1.73
\end{aligned}
$$

where the choices correspond to



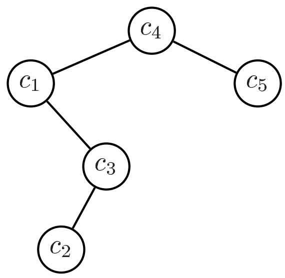
Finally we get as a resulting matrix:

	1	2	3	4	5
1	0.3	0.4	0.61	1.49	1.73
2		0.05	0.18	0.76	1.00
3			0.08	0.61	0.85
4				0.45	0.69
5					0.12

By actually keeping track of where the minimum occurred in

$$c_{ij} = m_{ij} + \min_k \{c_{i\,k-1} + c_{k+1\,j}\}$$

we can reconstruct the tree



4.8.1 Time complexity of the optimal search tree algorithm

The number of c_{ij} we computed is $n - m$ where $m = j - i$. For each c_{ij} we had a choice among $m + 1$ possibilities. So the total work is

$$\sum_{m=1}^{n-1} (n-m)(m+1)$$

and therefore the time complexity is $\Theta(n^3)$.

Chapter 5

Backtracking Algorithms

A backtracking algorithm is an organized exhaustive search which often avoids searching all possibilities. Thus, all **feasible** solutions are checked, though it is possible, via **pruning**, to avoid generating all the feasible solutions. We represent the whole process by a **state space tree** in which we try to recognize *good* and *bad* branches and prune the tree by abandoning the bad branches (thus cutting down the number of feasible solutions to be checked). Pruning is often done by using a **bounding function**. Backtracking algorithms generally make use of a bounding function to create a **depth first** state space tree. **Branch and bound** is a backtracking technique.
We revisit the traveling salesperson problem and show how branch and bound can be used to solve this combinatorial optimization problem.

5.1 The Traveling Salesperson Problem

Consider the following EXAMPLE:

	1	2	3	4	5
1	∞	25	40	31	27
2	5	∞	17	30	25
3	19	15	∞	6	1
4	9	50	24	∞	6
5	22	8	7	10	∞

The above table represents the cost matrix of visiting a certain city from any other of the remaining 4 cities. For example, the entry in row 3 and column 5 is the cost of going from city 3 to city 5. Recall that with the traveling salesman problem, we want to find the cheapest Hamiltonian circuit. Let us represent the above information by a cost matrix M.

$$M = \begin{bmatrix} \infty & 25 & 40 & 31 & 27 \\ 5 & \infty & 17 & 30 & 25 \\ 19 & 15 & \infty & 6 & 1 \\ 9 & 50 & 24 & \infty & 6 \\ 22 & 8 & 7 & 10 & \infty \end{bmatrix}$$

Next, we search for a lower bound on the cheapest Hamiltonian tour. Reduce M to a matrix that has a 0 in every row and in every column.

$$\begin{array}{cc}
 & \text{row subtractions} \\
\begin{bmatrix}
\infty & \mathbf{0} & 15 & 6 & 2 \\
\mathbf{0} & \infty & 12 & 25 & 20 \\
18 & 14 & \infty & 5 & \mathbf{0} \\
3 & 44 & 18 & \infty & \mathbf{0} \\
15 & 1 & \mathbf{0} & 3 & \infty
\end{bmatrix} &
\begin{array}{c} 25 \\ 5 \\ 1 \\ 6 \\ 7 \\ \hline 44 \end{array}
\end{array}$$

$$\begin{array}{cc}
 & \begin{bmatrix}
\infty & 0 & 15 & 3 & 2 \\
0 & \infty & 12 & 22 & 20 \\
18 & 14 & \infty & 2 & 0 \\
3 & 44 & 18 & \infty & 0 \\
15 & 1 & 7 & \mathbf{0} & \infty
\end{bmatrix} \\
\text{column subtractions} & \begin{array}{ccccc} & & & 3 & \end{array}
\end{array}$$

Hence, the total amount subtracted is: $44 + 3 = 47$.

Fact: Any Hamiltonian circuit will have a cost greater or equal to 47. Therefore 47 is a lower bound and we can use it as our bounding function.

Next, we are interested in finding one Hamiltonian path. One of the ways of finding such a path is demonstrated in the following steps. The Hamiltonian path will constitute the leftmost path in the state space tree. Recall that the construction of the state space tree is a depth-first construction.

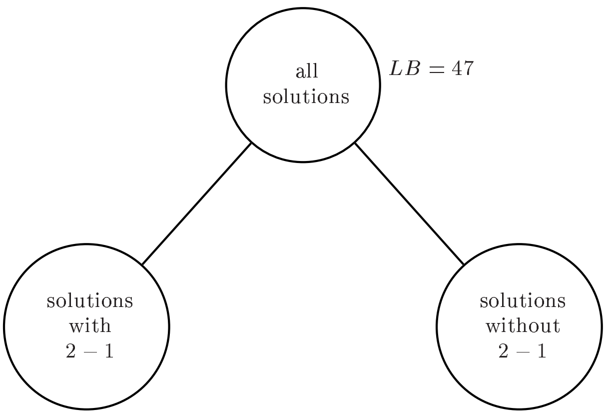
Find the minima (diferrent from the first zero) of each row and column.

$$\begin{array}{ccl}
 & & \text{minima} \\
 & \begin{bmatrix}
\infty & \mathbf{0} & 15 & 3 & 2 \\
\mathbf{0} & \infty & 12 & 22 & 20 \\
18 & 14 & \infty & 2 & \mathbf{0} \\
3 & 44 & 18 & \infty & \mathbf{0} \\
15 & 1 & \mathbf{0} & 0 & \infty
\end{bmatrix} &
\begin{array}{l} 2 \\ 12 \text{ highest!} \\ 2 \\ 3 \\ 0 \end{array} \\
\text{minima} & \begin{array}{ccccc} 3 & 1 & 12 & 0 & 0 \end{array} &
\end{array}$$

Suppose that we choose edge 2-1 as 1^{st} edge (since 12 is highest).

We start building the state space tree. The root node represents **all** possible solutions.

The left child of the root node represents all solutions that include the edge 2-1 while the right child represents all solutions that do not include the edge 2-1. So far we have constructed:



We will continue by considering the left child since this is a depth-first procedure.

Left child of the root node:

Delete row 2 and column 1 since by choosing 2-1 we do not want to leave from 2 anymore and we do not want to get to 1 from anywhere. Moreover, entry $(1, 2)$, i.e. cost from going from 1 to 2 is now ∞ since it is an impossible path (no cycles). Therefore, the correspinding cost matrix is:

	2	3	4	5
1	∞	15	3	2
3	14	∞	2	0
4	44	18	∞	0
5	1	0	0	∞

Again, we will reduce the new cost matrix to a matrix, where every row and every column has a 0 (by subtracting 1 in the first column and 2 in the first row). So we obtain:

	2	3	4	5
1	∞	13	1	0
3	13	∞	2	0
4	43	18	∞	0
5	0	0	0	∞

Since we subtracted $1 + 2 = 3$, the lower bound at node solution with $2 - 1$ is $LB = 47 + 3 = 50$. We choose edge $4 - 5$. Again we get a new cost matrix:

	2	3	4
1	∞	13	1
3	13	∞	2
5	0	0	∞

The reduction yields

	2	3	4
1	∞	12	0
3	11	∞	0
5	0	0	∞

So lower bound at solution with $4 - 5$ is $LB = 50 + 3 = 53$. Now we choose edge $1 - 4$, with a cost matrix

	2	3
3	11	∞
5	0	0

and a lower bound $LB = 53 + 11 = 64$, since we reduce this matrix to the matrix below by subtracing 11 from the first row. Hence the matrix at solution with $1-4$ is

	2	3
3	0	∞
5	0	0

Since the remaining table has 0's and ∞'s only, we can deduce the circuit corresponding to this path (left most) in the state space tree.
In the process, we chose the following edges: $2-1$, $4-5$, $1-4$, $3-2$, $5-3$, so we have $1-4-5-3-2-1$ as circuit with cost 64.
So 64 is the cost of the current best circuit.
At this point, we are done with the leftmost path in the state space tree and have found a Hamiltonian path of cost 64.

So now we backtrack to node solution without $1-4$ and we get as new cost matrix

	2	3	4
1	∞	12	∞
3	11	∞	0
5	0	0	∞

So the lower bound at node solution without $1-4$ is $LB = 53 + 12 = 65$, but $65 \geq 64$, therefore *prune* tree at this node, i.e. **stop** this path in the state space tree. So backtrack to solution without $4-5$. The cost matrix is

	2	3	4	5
1	∞	13	1	0
3	13	∞	2	0
4	43	18	∞	∞
5	0	0	0	∞

and after the reduction (subtract 18 from the elements in the third row) we have:

	2	3	4	5
1	∞	13	1	0
3	13	∞	2	0
4	25	0	∞	∞
5	0	0	0	∞

Therefore lower bound at solution without $4-5$ is $LB = 50 + 18 = 68$ and again $68 \geq 64$, and backtrack to the right child of the root node.

Right child of the root node:

Backtracking to the right child of the root node take us to: solution without $2-1$.

The cost matrix is

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	12	22	20
3	18	14	∞	2	0
4	3	44	18	∞	0
5	15	1	0	0	∞

and after subtracting 12 in the second row and 3 in the first column, we have

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	0	10	8
3	15	14	∞	2	0
4	0	44	18	∞	0
5	12	1	0	0	∞

with a lower bound of $LB = 47 + 15 = 62 \leq 64$. So continue working along this path in the tree. We choose edge 4-1, and the new cost matrix is already reduced:

	2	3	4	5
1	0	15	∞	2
2	∞	0	10	8
3	14	∞	2	0
5	1	0	0	∞

so at node solution with $4-1$ $LB = 62 + 0 = 62$. We continue and now we choose edge $2-3$, obtaining the cost matrix

	2	4	5
1	0	∞	2
3	∞	2	0
5	1	0	∞

that is again already reduced.
After choosing edge $3-5$ the cost matrix still remains reduced:

	2	4
1	0	∞
5	1	0

So choose edges $1-2$ and $5-4$ with lower bound $LB = 62 + 0$. Stop this branch with current optimal solution $= 62$ and circuit $4-1-2-3-5-4$. Forget about the previous $LB = 64$ and now use the current optimal $LB = 62$.

Backtrack to solution without $5-4$ with cost matrix

	2	4
1	0	∞
5	1	∞

But here we have to subtract 1 (second row), so $LB = 62 + 1 = 63 > 62$. Therefore stop here and backtrack to solution without $3-5$. But the cost matrix

	2	4	5
1	0	∞	2
3	∞	2	∞
5	1	0	∞

has to be recuded (subtract 2 in second row and in third column), so $LB = 62 + 4 = 66 > 62$, again stop this path and backtrack to solution without $2-3$.

	2	3	4	5
1	0	15	∞	2
2	∞	∞	10	8
3	14	∞	2	0
5	1	0	0	∞

Reduce by subtracting 8 in 2^{nd} row. Therefore $LB = 62 + 8 = 70$, so stop this path and backtrack to solution without $4-1$. The cost matrix is

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	0	10	8
3	15	14	∞	2	0
4	∞	44	18	∞	0
5	12	1	0	0	∞

that has to be reduced by subtracting 12 in the first column, therefore $LB = 62 + 12 = 74$. So stop (prune).

The best solution is $4-1-2-3-5-4$ with cost $= 62$.
Note that we could have stopped at the node corresponding to *solutions with edge* $3-5$ of cost$= 62$, since its lower bound is 52 which is equal to the lower bound of the node corresponding to *solutions without edge* $2-1$. There is no way one can find a cheaper cost (than 62) by considering the right subtree of the node corrresponding to *solutions without arc* $2-1$ since

$$\begin{array}{c}\text{lower bound}\\ \text{of node}\end{array} \leq \begin{array}{c}\text{lower bound}\\ \text{of any descendant}\\ \text{of that node}\end{array}$$

The state space tree is given by Figure 5.1.

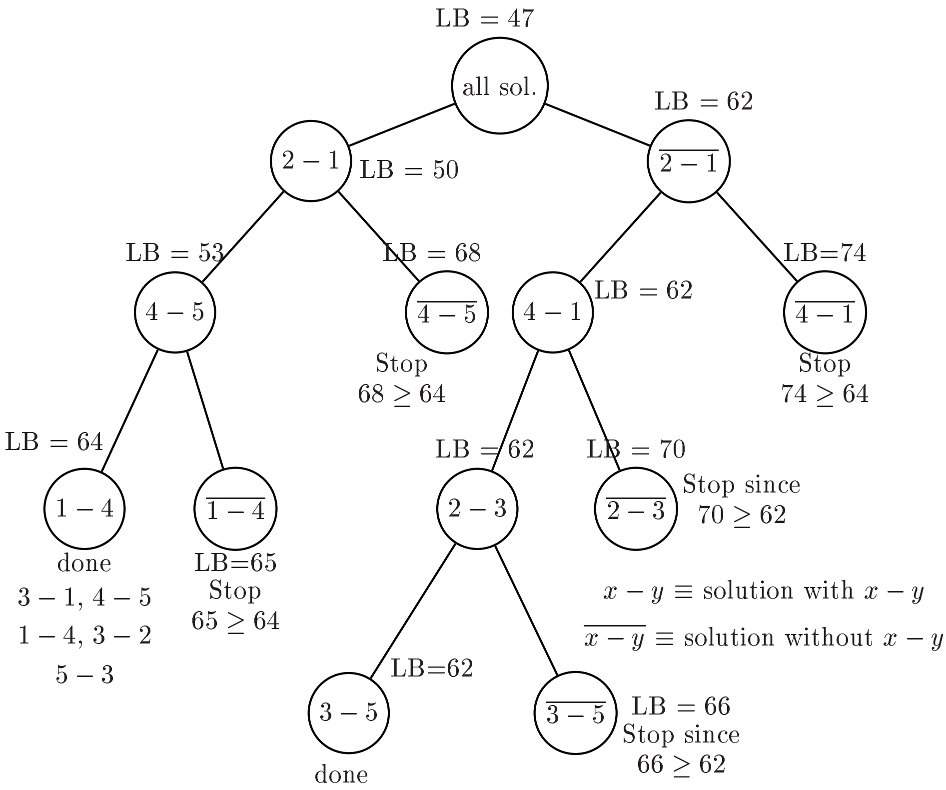


Figure 5.1: The State Space Tree.

Next, we revisit the 0-1 knapsack problem and give two backtracking algorithms to sole the problem.

5.2 Backtracking and the 0-1 Knapsack Problem

In this section, we discuss two methods for solving the 0-1 knapsack problem. While the first is very intuitive and easy to understand, the second is more efficient [Sti87].

First method: Let $\mathbf{x} = (x_1, x_2, \ldots, x_{lev}, -, -, \ldots, -)$ denote a partial solution during the construction of a feasible solutions. Recall that

$$x \text{ is feasible } \Leftrightarrow \sum_{i=1}^{lev} w_i x_i \leq M \qquad 1 \leq lev \leq n \quad \begin{array}{c}\text{where } M \text{ is the}\\ \text{knapsack capacity}\end{array}$$

$\sum_{i=1}^{lev} w_i x_i$ can thus be used to prune the state space tree.

EXAMPLE:

objects	1	2	3	4	5
weights	11	12	8	7	9
profits	23	24	15	13	16

$M = 26$

Note that the ordering of the objects $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$ is not important here.

Second method: Use a bounding function $B(x)$ where $\mathbf{x} = \{x_1, x_2 \ldots, x_{lev}, \ldots\}$. Let $currw := \sum_{i=1}^{lev} w_i x_i$ and define $RK(lev, M')$ to be the optimal profit of the rational knapsack problem using objects $lev+1, lev+2, \ldots, n$ and capacity $M' = M - currw$. Define the bounding function $B(\mathbf{x})$ where $\mathbf{x}$ is a feasible partial solution:

$$B(\mathbf{x}) = \sum_{i=1}^{lev} p_i w_i + RK(lev, M - currw).$$

In other words, $B(\mathbf{x})$ is the sum of

- the profit obtained from objects $1, 2, \ldots, lev$, plus
- the profit from the remaining objects, using the remaining capacity $M - currw$, but allowing rational x_i's.

EXAMPLE: Consider the previous one.

1. $\mathbf{x} = (-,-,-,-,-)$; the optimal solution using objects $1,2,3,4,5$ is given by $RK(0, M)$ ($M' = M$, since $currw = 0$), where

   $$RK(0, M) = 23(1) + 24(1) + 15(\frac{3}{8}) = 52.625,$$

   since $(1, 1, \frac{3}{8}, 0, 0)$ is the solution vector $\Rightarrow$ $B(\mathbf{x}) = 0 + 52.625$

2. $\mathbf{x} = (1,-,-,-,-)$ with $currw = 11$; optimal solution using $2,3,4,5$ and $M' = 26 - 11 = 15$ is

   $$RK(1, M') = 24(1) + 15(\frac{3}{8}) = 29.625$$

   Therefore

   $$B(\mathbf{x}) = 23 \cdot (1) + 29.625$$

3. $\mathbf{x} = (1,1,-,-,-)$, $currw = 23$ and $M' = 26 - 23 = 3$

   $$RK(2, M') = \frac{3}{8} \cdot 15 = 5.625$$

   Therefore

   $$B(\mathbf{x}) = [23(1) + 24(1)] + [5.625] = 52.625$$

4. $\mathbf{x} = (1, 1, 1, -, -)$, is not a feasible solution since $currw = 31 > 26$; so backtrack and consider:

5. $\mathbf{x} = (1, 1, 0, -, -)$, $currw = 23$ and $M' = 26 - 23 = 3$; now look for optimal solution using objects 4 and 5 to fill knapsack of capacity 3

$$RK(3, M') = \frac{3}{7} \cdot 13 = 5.5705$$

$$\Rightarrow B(\mathbf{x}) = [23(1) + 24(1) + 15(0)] + 5.5705 = 52.5705$$

6. $\mathbf{x} = (1, 1, 0, 1, -)$ is not a feasible solution, therefore backtrack and consider:

7. $\mathbf{x} = (1, 1, 0, 0, -)$ with $currw = 23$ and $M' = 26 - 23 = 3$; so

$$RK(4, M') = \frac{3}{9} \cdot 16 = 5.3333$$

Therefore

$$B(\mathbf{x}) = 47 + 5.3333 = 52.3333$$

8. $\mathbf{x} = (1, 1, 0, 0, 1)$ is not a feasible solution; backtrack and consider:

9. $\mathbf{x} = (1, 1, 0, 0, 0)$; profit $= 23(1) + 24(1) = 47$; therefore let $opt = 47$ (current optimum profit); now backtrack almost to the beginning:

10. $\mathbf{x} = (1, 0, -, -, -)$, $currw = 11$ and $M' = 26 - 11 = 15$

$$RK(2, M') = 1(15) + 1(13) = 28$$

Therefore

$$B(\mathbf{x}) = 23 + 28 = 51$$

11. $\mathbf{x} = (1, 0, 1, -, -)$, $currw = 19$ and $M' = 26 - 19 = 7$

$$RK(3, M') = 1(13) = 13$$

Therefore

$$B(\mathbf{x}) = (23 + 15) + 13 = 51$$

12. $\mathbf{x} = (1, 0, 1, 1, -)$ with $currw = 26$ and $M' = 26 - 26 = 0$

$$RK(4, M') = 0$$

So

$$B(\mathbf{x}) = (23 + 15 + 13) + 0 = 51$$

13. $\mathbf{x} = (1, 0, 1, 1, 1)$ is not a feasible solution, therefore backtrack and consider

14. $\mathbf{x} = (1, 0, 1, 1, 0)$ with $currw = 26$, $B(\mathbf{x}) = P = 51$; since $51 > 47$, let $opt = 51$ (that is the new current optimal profit); backtrack and consider

15. $\mathbf{x} = (1, 0, 1, 0, -)$, $currw = 19$ and $M' = 26 - 19 = 9$

$$RK(4, M') = \frac{7}{9} \cdot 16$$

$$B(\mathbf{x}) = (23 + 15) + \frac{7}{9} \cdot 16 = 50.52 < 51 \text{ so prune}$$

Again backtrack and consider

16. $\mathbf{x} = (1, 0, 0, -, -)$ with $currw = 11$ and $M' = 26 - 11 = 15$

$$RK(3, M') = 1(13) + \frac{8}{9} \cdot 16 = 13 + 14.08$$

$$B(\mathbf{x}) = 23 + 27.08 < 51 \text{ so again prune}$$

backtrack and consider

17. $\mathbf{x} = (0, -, -, -, -)$, $currw = 0$ and $M' = 26 - 0 = 26$; so

$$RK(1, M') = 1(24) + 1(15) + \frac{6}{7} \cdot 13 = 50.141$$

and

$$B(\mathbf{x}) = 0 + 50.141 < 51 \text{ so prune}$$

Finally, we are done with the solution $(1, 0, 1, 1, 0)$ and an optimal profit $opt = 51$.

Remark: We know that

$$\text{optimal solution of 0-1 knapsack} \quad \leq B(\mathbf{x})$$

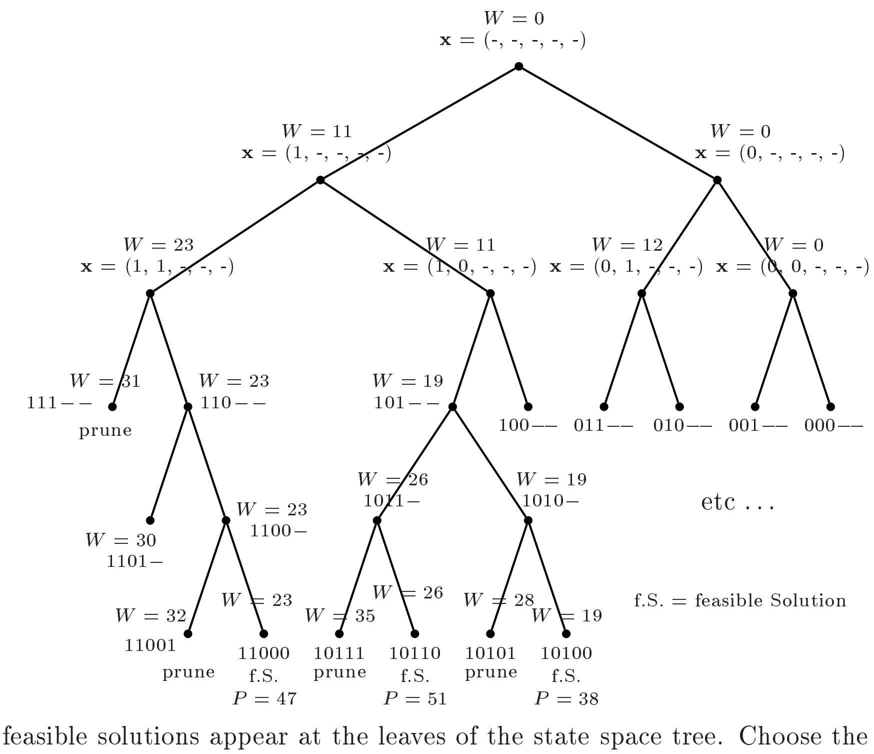
So if at some stage during the construction of the state space tree we have

$$B(\mathbf{x}) \leq opt$$

(where *opt* is the optimal profit found so far) then

$$\text{optimal solution of 0-1 knapsack} \ \leq B(\mathbf{x}) \leq opt$$

we can therefore ignore **all** descendants of $\mathbf{x}$ (i.e. prune tree) since none of them will produce a profit higher than *opt*.



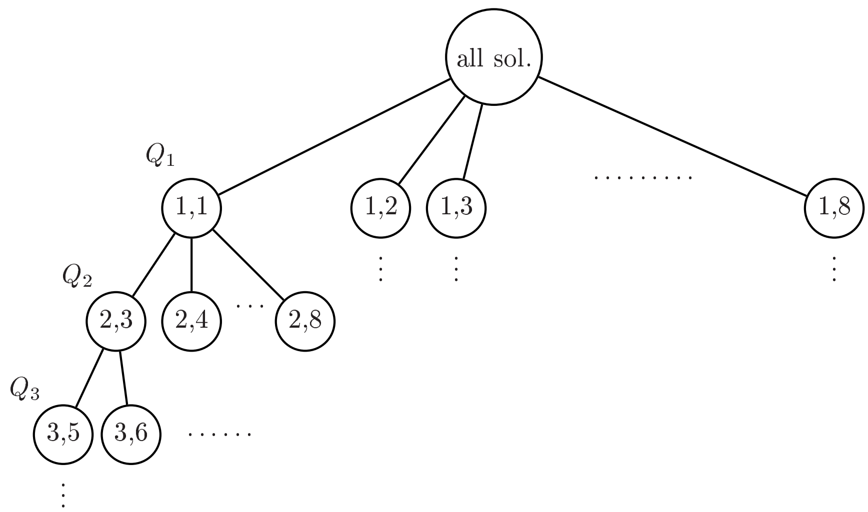
All feasible solutions appear at the leaves of the state space tree. Choose the one with highest profit.
We now consider three more problems that can be solved by using backtracking.

5.3 More Backtracking Examples

There are problems that can be solved only by performing an exhaustive search of all possible solutions. One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution. Recall that to find a solution via backtracking, first make a sequence of decisions in an attempt to reach a solution as long as this is possible. The sequence of decisions can one represent by a path in the decision tree (state space tree). Once it is known that no solution can result from any further sequence of decisions, backtrack to the parent of the current vertex and work toward a solution with another series of decisions, if this is possible. The procedure continues until a solution is found, or it is established that no solution exists. The problems discussed in this section are taken from [Ros95, pp. 585-588].

5.3.1 The Eight Queens Problem

The n-queens problem asks how n queens can be placed on an $n \times n$ chessboard so that no two queens can attack one another. We have to find n positions on an $n \times n$ chessboard so that no two of these positions are in the same row, same column, or in the same diagonal. We will use backtracking to solve the n-queens problem.



Positions on the board are represented by (a, b); a: row, b: column. So the levels in the tree are the rows. The 1^{st} node has 8 children but subsequent nodes will have less than 8 children since we know that if we place a queen on $(1, 1)$ then we don't need to consider $(2, x)$ for $x = 1, 2$. So now place a queen on $(1, 1)$. Consider the children of $(1, 1)$. Place the second queen Q_2 on $(2, 3)$. Consider the children of $(2, 3)$. Place the 3^{rd} queen Q_3 on $(3, 5)$ etc. until we are stuck. Backtrack to the most recent parent, consider the next child and go on until stuck again or all 8 queens are placed on the board. Note that this method will yield a solution with Q_1 on $(1, 1)$. If we are interested in more than one solution, we can finally backtrack to $(1, 2)$, place Q_1 on $(1, 2)$ and use the same method to generate solutions with Q_1 on $(1, 2)$.

Possible solutions:

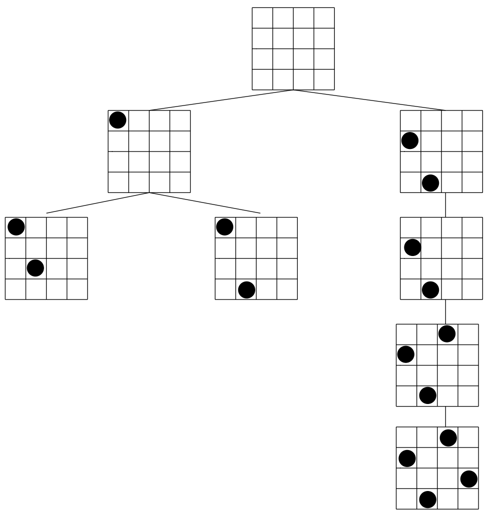
1. (1, 3), (2, 5), (3, 8), (4, 4), (5, 1), (6, 7), (7, 2), (8, 6)
2. (1, 1), (2, 7), (3, 4), (4, 6), (5, 8), (6, 2), (7, 5), (8, 3)

There are $\binom{64}{8} = 4,426,165,368$ ways of choosing 8 out of (8×8). By allowing the placement of queens only on distinct rows and columns, we require the examination of at most 8! 8-tuples.

Another way of seeing the 8! is by observing that in the (a, b) notation we can omit **a** (since we have $a = 1$, $a = 2$, ..., $a = 8$ in that order) and represent the solution with $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ where the x_i are the b's i.e. the columns. So we have a permutation of 8 queens. As a matter of fact, Sahni and Horowitz used this fact and have an elegant solution in "Fundamentals of Computer Algorithms" [HS78].

As a second example, Rosen considers the Four-Queens problem. He uses backtracking to solve the problem. At stage $(k + 1)$ we attempt putting an additional queen on the board in the $(k + 1)$st column, where there are already queens in the first k columns. We examine squares in the $(k + 1)$st column starting with the square in the first row, looking for a position to place this queen so that it is not in the same row or in the same diagonal as a queen already on the board. If it is impossible to place the queen in the $(k + 1)$st column, backtrack to the placement of the queen in the kth column, and place this queen in the next allowable row in this column, if such a row exists. If no such row exists, backtrack further.

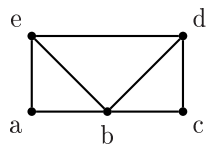
The following is a solution to the 4-queens problem, that works in the way described above.

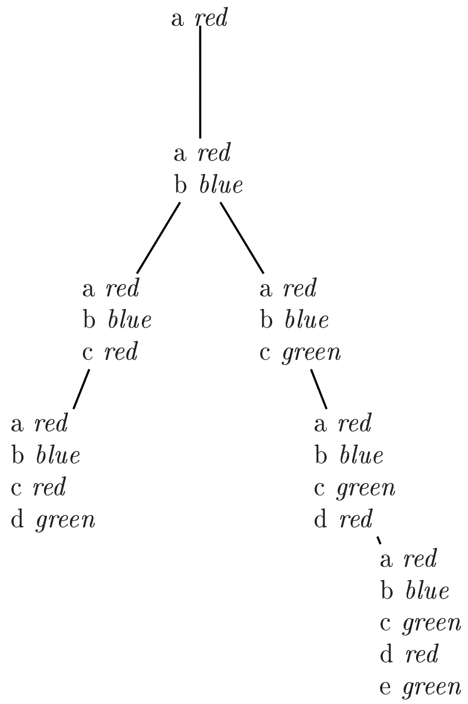


5.3.2 Vertex coloring problem

Find the minimum number of colors needed to color the vertices of a graph such that no two adjacent nodes have the same color. In this example, we would like to know if we can color the graph with 3 colors only. The tree shown in the figure illustrates how backtracking can be used to construct a 3-coloring. In this procedure, red is used first, then blue, and finally green.

In this tree, the initial path from the root, which represents the assignment of red to a, leads to a coloring with a red, b blue, c red, and d green. It is impossible to color e using any of the three colors when a, b, c, and d are colored in this way. So, backtrack to the parent of the vertex representing this coloring. Since no other color can be used for d, backtrack one more level. Then change the color of c to green. We obtain a coloring of the graph by then assigning red to d and green to e. □

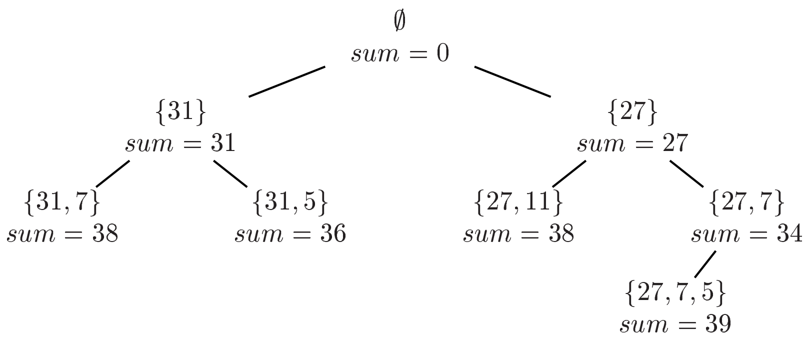




5.3.3 Sum of subsets

Given a set of positive integers $x_1, x_2, \ldots, x_n$ find a subset that has M as its sum.

EXAMPLE: $S = \{31, 27, 15, 11, 7, 5\}, M = 39$



To find a solution, we start with a sum with no terms. We build up the sum by successively adding terms. An integer in the sequence is included if the sum remains less than M when this integer is added to the sum. If a sum is reached so that the addition of any term is greater than M, backtrack by dropping the last term of the sum. The above figure displays a backtracking solution to the problem of finding a subset of $\{31, 27, 15, 11, 7, 5\}$ with sum equal to 39. $\square$

This concludes the algorithmic techniques we wanted to study; namely: greedy, divide and conquer, dynamic programming and backtracking. Before continuing and studying topics, we end the introduction to algrithmic techniques by giving a very brief classification of problems.

5.4 Classification of Problems

Roughly speaking, problems can be classified in three categories:

tractable: solvable in polynomial-time
intractable: solvable but not in polynomial-time
unsolvable: problems that **cannot** be solved (example: halting problem)

NP-Complete Problems: Problems for which no polynomial-time algorithm is known, but no one has been able to show that such algorithms do not exit. NP-complete problems are equivalent to the sense that if any one of them has a polynomial-time solution, then all of them do. One of the major open questions in computer science is whether there is a polynomial-time solution for any of the NP-complete problems. Almost all experts conjecture strongly that the answer is **no**. The evidence supporting this conjecture, that $P \subset NP$ and not $P = NP$, is only the failure to find such a polynomial-time solution to any of the NP-complete problems, in over thirty years of research. Steven Cook defined the notion of NP-completeness and showed the NP-completeness of a collection of problems, mostly from the field of mathematical logic. The first problem he tracked is the Satisfiability Problem. In 1970's, he proved that this problem is NP-complete.

Roughly speaking, he defined NP-complete problems to be problems that have the property that we can verify in polynomial-time whether a supplied solution is correct. For example, if we did not have to find an optimal filling of the 0-1 Knapsack Problem but simply to verify that a given filling has a profit value of at least k, where k is a given integer, it would be easy to write a polynomial-time algorithm to do so.

Since then, many hundreds of natural problems from many areas of computer science and mathematics such as graph theory, combinatorial optimization, scheduling and symbolic computation have been proven to be NP-complete.

There are several reasons for which polynomial-time problems are considered tractable:

a) Most practical problems can be solved in $O(n^4)$ and do not require higher-degree polynomial-time algorithms.

b) Portability of polynomial-time problems (from one model to another).

c) The class of polynomial-time problems has nice closure properties because polynomials are closed under $+$, $\times$ and $\circ$ (polynomial composition).

In practice, we still need to solve many problems that are known to be NP-complete. There are several ways of dealing with NP-complete problems:

1. We can sometimes use dynamic programming or branch and bound techniques to cut down the running time from the brute force exhaustive search. This might work only if the problem size is not too big.

2. In some cases, algorithms have non-polynomial behavior in the worst-case, but are efficient in some cases (with certain input data).

3. Give up the exact algorithm and settle for an approximate algorithm that will generally produce a suboptimal solution. Approximation algorithms (unlike heuristics) produce solutions with some guarantee of how close they are to optimal solutions.

4. Heuristics such as local search, simulated annealing, genetic algorithms, neural networks, and many others can be used to improve the solution quality or the running time. However, the solutions produced cannot be guaranteed to be within a certain distance from the optimal solution. Rigorous analysis of heuristic algorithms is generally very difficult.

5. The problem at hand may be more specialized than the general one that was proven to be NP-complete.

Chapter 6

Topics

In this chapter we study three topics: shortest paths, splay trees and the coloring of graphs [CLR90].

6.1 Shortest Paths

The basic problem is to find the "best" way of getting from point s to point t where s and t are vertices in a graph

Best is measured by summing the length (or weight) of the edges that are along the path from s to t. In shortest path problems, we have a weighted directed graph $G = (V, E)$ with a weight function that maps edges to real-valued weights. We will assume that weights are positive, at least for the first algorithms we study. There are three variants:

single-source shortest path: we want to find a shortest path from a given source $s \in V$ to every vertex $v \in V$

single-pair shortest path: find shortest path from $u \in V$ to $v \in V$, i.e. two fixed vertices u and v

all-pairs shortest paths: find shortest path for every pair $u \in V$ and $v \in V$

6.1.1 Single-Source Shortest Path Problem

Dijkstra's algorithm is based on the greedy technique. The algorithm maintains a set S of vertices whose shortest path from the source s is already known and the length $dis[u]$ of the shortest path from the source s to each vertex u that passes only through vertices in S.

Initially, S contains no vertices. At each stage, we add to S a remaining vertex x whose distance value $dis[x]$ is minimum. A tie is broken arbitrarily. At this point, $dis[u]$ for vertex u not in S may decrease, since there may be a shorter path through x of length $dis[x] + d(x, w)$. So at each iteration, we find a shortest path among paths from the source s to a vertex that passes only through vertices in S. The process is repeated until S contains all vertices of V.

Procedure DIJKSTRA finds a shortest path from the source s to every other vertex $v \in V \backslash \{s\}$ in a directed graph $G = (V, E)$. For each vertex $u \in V \backslash \{s\}$, $pred[u]$ denotes the predecessor of u in a shortest path tree. $dis[u]$ denotes the length of a shortest path from s to u. Procedure DIJKSTRA assumes that graph G is represented by adjacency lists. It also assumes that all edges have nonnegative weights.

The procedure maintains a priority queue Q that initially contains all the vertices in V by their *dis* value. More precisely, *dis* is used as a key where *dis* is initialized in the following fashion:

$$dis[v] := \infty \qquad \text{for all } v \in V\backslash\{s\} \text{ and}$$

$$dis[s] := 0$$

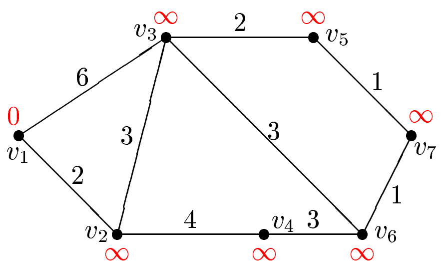
which forces s to be chosen in the first iteration of the while loop.

For a given graph $G = (V, E)$, a starting vertex s and a weight function $d : E \to \mathbf{R}^+$ procedure DIJKSTRA(G, s) is as follows:

DIJKSTRA(G, s)

begin
  $S \leftarrow \emptyset$
  for all $v \in V\backslash\{s\}$ **do**
    $pred[v] \leftarrow s$
initialize priority queue Q containing all vertices $v \in V\backslash\{s\}$ with key $dis[v] \leftarrow \infty$ and containing s with key $dis[s] \leftarrow 0$
  while $S \neq V$ **do**
    $v \leftarrow$ EXTRACTMIN(Q)
    $S \leftarrow S \cup \{v\}$
    for each $w \in adj[v]$ **do**
      if $dis[v] + d(v, w) < dis[w]$
        then
          DECREASEKEY$(w, dis[v] + d(v, w))$
          /* DECREASEKEY changes $dis[w]$ to $dis[v] + d(v, w)$ */
          $pred[w] \leftarrow v$
end

EXAMPLE:

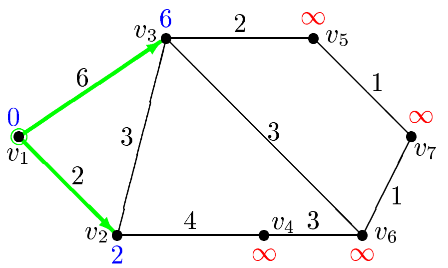


input graph G;

starting node v_1;

dis$[v_1]$=0;

dis$[u]$=$+\infty$
for all $u \in V - \{v_1\}$;

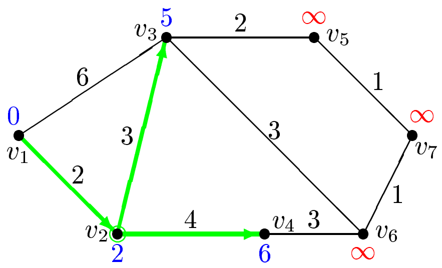


process v_1:

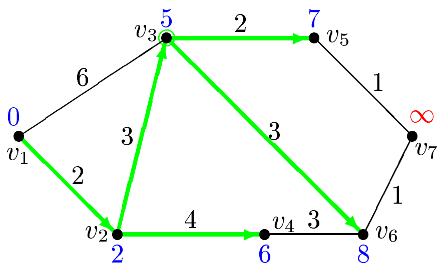
$adj[v_1] = \{v_2, v_3\}$

mark (v_1, v_2);
set dis$[v_2]$=2;

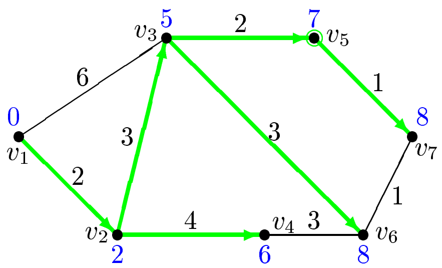
mark (v_1, v_3);
set dis$[v_3]$=6;



process v_2:
mark (v_2, v_3) and
unmark (v_1, v_3), because
$\text{dis}[v_2] + d(v_2, v_3) < \text{dis}[v_3]$;
set $\text{dis}[v_3]=2+3=5$;
mark (v_2, v_4);
set $\text{dis}[v_4]=2+4=6$;



process v_3;
mark (v_3, v_5);
set $\text{dis}[v_5]=5+2=7$;
mark (v_3, v_6);
set $\text{dis}[v_6]=5+3=8$;
(no change when
v_4 is processed)



process v_5;
mark (v_5, v_7);
set $\text{dis}[v_7]=7+1=8$;
(no more changes when
v_6 and v_7 are processed)

Note that the strategy of Dijkstra's algorithm is similar to the one used for PRIM's minimum spanning tree algorithm. We grow a "dijkstra tree", starting at vertex s, by adding at each iteration a "cheap" edge whose non-tree endpoint is as close as possible to s. In PRIM's algorithm, the non-tree endpoint is as close as possible to any vertex in the current prim tree.

In both algorithms we insert one vertex at a time (build the tree by adding one edge at a time) and adjust the weights of the remaining vertices outside the "set" accordingly. Note that procedure DIJKSTRA also produces the actual minimum-weight paths themselves (not only the value of the paths stored in $dis[u]$). The edges in the "dijkstra tree" are of the form $(pred[u], u)$ for all $u \in V \backslash \{s\}$.

Proof of Dijkstra's algorithm

The key is to show that each time a vertex u is inserted into S, we have $dis[u] = \delta(s, u)$, where $\delta(s, u)$ denotes the distance of the shortest path from s to u.

The proof is based on the following facts.

Fact 1: *Shortest paths are composed of shortest subpaths*
Note that if this were not true and there exists a shorter path than any subpath, then the shorter path should replace that subpath to make the whole path shorter

Fact 2: *If $s \to \ \ldots \ \to u \to v$ is a shortest path from s to v, then after the insertion of u in S, we have $dis[u] = \delta(s, v)$, and $dis[v]$ does not change thereafter.* Note that at all times we have $dis[v] \geq \delta(s, v)$.

Theorem 6.1.1 *After the execution of procedure* Dijkstra*, we have* $dis[u] = \delta(s,u)$ *for all* $u \in V$.

Proof: The proof that $dis[u] = \delta(s,u)$ for all $u \in V$, after running the procedure, is by contradiction. Suppose that v is the first vertex added to S for which $dis[v] \neq \delta(s,v)$, i.e. $dis[v] > \delta(s,v)$. Since $dis[s] = 0$, v cannot be s. Since $dis[v] < \infty$ (v is added to S) then there must be a path from s to v. Also, since there is a path from s to v, there must be a shortest path from s to v.

Let $s \cdots x \to y \cdots v$ be a shortest path from s to v, where x is in S and y is the first vertex not in S. We know that $dis[x] = \delta(s,x)$ since we assume that v is the **first** vertex added to S for which $dis[v] \neq \delta(s,v)$. Since $y \in adj[x]$, when x is inserted in S, we perform the **if** statement, after which we have:

$$dis[y] = dis[x] + d(x,y)$$

So

$$\begin{aligned} dis[y] &= \delta(s,y) \\ &\leq \delta(s,v) \\ &\leq dis[v] \end{aligned}$$

So $dis[y] \leq dis[v]$.

But right before choosing v to insert in S, we had $v \in V \backslash S$ and $y \in V \backslash S$, and v was chosen over y. The choice was dictated by the fact that:

$$dis[v] \leq dis[y]$$

$$\left.\begin{array}{c} dis[y] \leq dis[v] \\ \text{and} \\ dis[v] \leq dis[y] \end{array}\right\} \Rightarrow \quad dis[v] = dis[y]$$

In other words,

$$\left.\begin{array}{rcl} dis[y] & = & \delta(s,y) \\ & \leq & \delta(s,x) \\ & \leq & dis[v] \\ & \text{and} & \\ dis[y] & = & dis[v] \end{array}\right\} \Rightarrow \quad dis[v] = \delta(s,v)$$

which contradicts our hypothesis.

So whenever a node u is chosen to be inserted in S, we have $dis[u] = \delta(s,u)$.

Analysis of Dijkstra's algorithm

Note that the vertices are scanned in increasing order of their shortest-path distance from s. Each vertex is scanned at most once. Its *dis*-value is updated at most once for each incoming edge, so DecreaseKey is performed at most $|E|$ times. By implementing the priority queue Q with a fibonacci heap, we know that each call to DecreaseKey will take only $O(1)$ amortized time to be executed.

Procedure Dijkstra performs ExtractMin*(Q)* for each vertex in V. So if Q is implemented with a fibonacci heap, then we have $|V| \log_2 |V|$ work since each ExtractMin*(Q)* operation requires $O(\log_2 |V|)$ amortized time. The total running time is therefore $O(|E| + |V| \log_2 |V|)$.

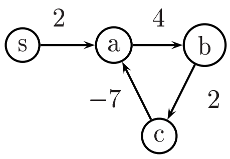
The fastest known implementation of Dijkstra's algorithm uses fibonacci heaps. The running times using other implementations are summarized in the following table:

heap implementation	ExtractMin	Insert	total
linked list	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
binary heap	$O(\log_2 \|V\|)$	$O(\log_2 \|V\|)$	$O(\|E\| \log_2 \|V\|)$
d-ary heap	$O\left(\frac{d \log_2 \|V\|}{\log_2 d}\right)$	$O\left(\frac{\log_2 \|V\|}{\log_2 d}\right)$	$O\left((\|V\|d + \|E\|)\frac{\log_2 \|V\|}{\log_2 d}\right)$
fibonacci	$O(\log_2 \|V\|)$ *amortized*	$O(1)$ *amortized*	$O(\|V\| \log_2 \|V\| + \|E\|)$

A d-ary heap is similar to a binary heap, except that each node has d children instead of just two.

Negative-weight edges

In some instances of the single-source shortest-paths problem, there may be edges whose weights are negative. As long as the graph $G = (V, E)$ does not contain a negative-weight cycle reachable from the source s, then for all $v \in V$, the shortest path weight $\delta(s, v)$, defined in the previous section, remains well defined, even if some of the edges have a negative value.



Vertices a, b and c form a negative-weight cycle. It does not make any sense to ask for the shortest path from s to a for instance. Is it 2, going directly from s to a, or 1, after one cycle through a, b and c or 0 after two cycles, or -1 after three cycles ...?

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have nonnegative weights. The Bellmann-Ford procedure will solve the problem even if some edges have negative weights as long as G contains no negative-weight cycles.

The Bellmann-Ford algorithm

The Bellmann-Ford algorithm solves the single-source shortest-path problem in the more general case in which edge weights can be negative. But the graph should not contain negative-weight cycles that can be reached from the source s. Procedure Bellman-Ford is based on the fact that if $s, v_1, v_2, \ldots, v_k = a$ is the shortest path between s and a for $1 \leq k \leq |V| - 1$, then if we perform

for $(v, w) \in E$
  if $dis[w] > dis[v] + d(v, w)$
  then
    $dis[w] := dis[v] + d(v, w)$
    $prev[w] := v$

first with $(v, w) = (s, v_1)$
then with $(v, w) = (v_1, v_2)$
then with $(v, w) = (v_2, v_3)$ and so on till $(v, w) = (v_{k-1}, a)$,

then we are sure that $dis[a]$ will contain the value of the shortest path between s and a and that the shortest path itself is enclosed in $prev$.

In essence, as was the case with Dijkstra's algorithm, $dis[v]$ is an upper bound on the value of the shortest distance between s and v, that we progressively decrease (with each iteration of the outer loop of the algorithm) until we achieve $dis[v] = \delta(s, v)$.

```
BELLMANN-FORD(G, s)
begin
  dis[v] := 0
  for all v ∈ V\{s} do
    dis[v] := ∞
    prev[v] := nil
  for i = 1 to |V| − 1 do
    for each (v, w) ∈ E do
      if dis[w] > dis[v] + d(v, w)
      then
        dis[w] := dis[v] + d(v, w)
        prev[w] := v
end
```

Note that by demanding that all edges go through the inner **for** loop $|V| - 1$ times, we are guaranteeing that edges $(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, a)$, on the shortest path between s and a, got through the inner loop in the desired order (i.e. first (s, v_1), followed by (v_1, v_2) and so on till (v_{k-1}, a)).

The Bellman-Ford algorithm runs in time $O(|V||E|)$, since the initialization takes $\Theta(|V|)$ time, and each of the $|V| - 1$ passes over the edges in the main **for** loop takes $O(|E|)$ time.

Single-source shortest path in DAG's

A DAG is a **d**irected **a**cyclic **g**raph. For this important class of graphs, shortest paths can be computed more quickly, in linear time.

Shortest paths are always well defined in a DAG, since those graphs can never contain negative weight cycles, even if they have negative-weight edges. Procedure DAG-SHORTEST-PATHS first topologically sorts the DAG. A topological ordering of a DAG is such that if we have an edge from x to y, then x precedes y in the topological sort. A topological sort of a DAG can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. The topological sort can be performed in $\Theta(|V| + |E|)$ time.

Procedure DAG-SHORTEST-PATHS then considers each vertex (in the topologically sorted order) and processes each adjacent vertex of the chosen vertex. Processing consists in updating $dis[w]$ for each $w \in adj[v]$ when v is the chosen vertex.

DAG-SHORTEST-PATHS$(G,\ s)$

begin
  $dis[s] := 0$
  for $v \in V \backslash \{s\}$
    $dis[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  topologically sort the vertices of G
  for each vertex v taken in topologically sorted order
    for each $w \in adj[v]$
      if $dis[v] + d(v,w) < dis[w]$
      then
        $dis[w] \leftarrow dis[v] + d(v,w)$
        $prev[w] \leftarrow v$
end

The topological sort – as mentioned already above – can be done in $\Theta(|V|+|E|)$ time. The outer **for** loop considers each vertex once and so the outer loop is performed $|V|$ times. The work done on each edge leaving w is constant. The total running time of the DAG-SHORTEST-PATHS procedure is $\Theta(|V|+|E|)$.

Topological sort

A topological sort of a directed acyclic graph $G = (V,E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering.

Results:

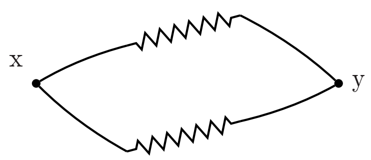
$$\begin{array}{c} G = (V,E) \text{ has a topological ordering} \\ \Updownarrow \\ G \text{ is a directed acyclic graph} \end{array}$$

Instead of the above, we show the equivalent result:

$$G \text{ has no topological ordering} \Leftrightarrow G \text{ contains a cycle}$$

Proof:

($\Leftarrow$): G contains a cycle, i.e.



x appears before y in the upper path
and
x appears after y in the lower path

$\Rightarrow G$ does not have a topological order

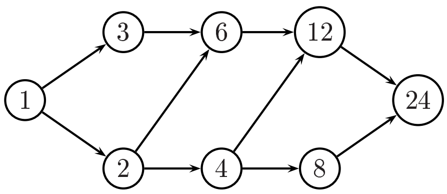
(⇒): consider the following procedure:

```
TOPOLOGICAL-SORT(G)
begin
  L := ∅;                                holds the vertices in topological order
  S := ∅;                                holds vertices without incoming edges
  for each vertex v ∈ V do
    I[v] := indegree[v];                 number of incoming edges to v
    if I[v] = 0
    then
      insert v in S;                     v is removed from V
  while S ≠ ∅ do
    remove a vertex x from S
    for each outgoing edge (x, w) do
      I[w] := I[w] − 1
      if I[w] = 0
      then
        insert w in S;                   w is removed from V
    insert x in L
end
```

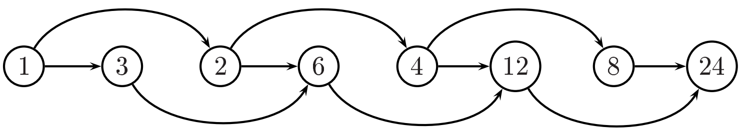
For example, TOPOLOGICAL-SORT produces the following list L, in that order

$$L = 1\ 2\ 3\ 4\ 6\ 8\ 12\ 24$$

when the procedure is applied to the DAG shown below.



Many other solutions, including $L := 1\ 3\ 2\ 6\ 4\ 12\ 8\ 24$ are possible. Recall that a topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.



The diagram shows the graph topologically sorted ($L := 1\ 3\ 2\ 6\ 4\ 12\ 8\ 24$).

Back to the proof: If the procedure executes to completion (and does not get stuck), L will contain the elements of V topologically sorted since $u \in V$ is inserted in L only after all its incoming edges have been deleted (in other words, all predecessors of u are already in L). If procedure TOPOLOGICAL-SORT does not execute to completion, it means that it cannot find a vertex with no incoming edges; in other words, $indegree[u] > 0$ for all u not in L. We want to show that in this case G has a cycle.

Pick one of the remaining nodes, w let us say, and find an incoming edge (t, w). We know that it exists since $indegree[w] > 0$. Likewise, let (m, t)

be an incoming edge of t. By continuing this process we are bound to have the following situation, where we eventually run into a vertex x we have seen before. This happens since there are only finitely many vertices (and all have indegrees greater than zero). So we have a cycle from x to itself. Therefore

$$G \text{ has no topological order} \Rightarrow G \text{ contains a cycle}$$

In summary, we have shown that

$$G \text{ has a topological ordering} \Leftrightarrow G \text{ is a directed acyclic graph}$$

Note that procedure TOPOLOGICAL-SORT finds a topological sorting of the graph when it exists, and can be used to find cycles when G is not a directed acyclic graph. The time complexity of the procedure is $\Theta(|V| + |E|)$.

6.1.2 Single-Pair Shortest-Path

To solve this problem, simply solve the single-source shortest-path problem from s to all vertices in V, and this will of course yield the shortest path from s to t. There are no algorithms that will find the shortest path from s to t without also finding the shortest path from s to all the other vertices in the worst case.

6.1.3 All-Pair Shortest Path

We have already studied procedure FLOYD that computes all-pairs shortest paths in $\Theta(|V|^3)$ time. The algorithm uses the dynamic programming technique.

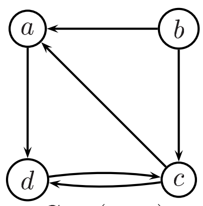
Note that this problem, as well as the other variants of the shortest path problem, has many applications including building a table of distances between all pairs of cities for a road atlas. One way of solving this problem is by running a single-source shortest paths algorithm $|V|$ times, one for each vertex as the source s. If the edges of G are nonnegative, we can use Dijkstra's algorithm. By using the fibonnaci heap implementation (as priority queue), we get a running time of $O(|V|^2 \log_2 |V| + |V||E|)$.

If on the other hand we have negative-weight edges, we could use BELLMAN-FORD procedure $|V|$ times, once from each vertex. The running time is then $O(|V|^2|E|)$.

Transitive closure of a directed graph

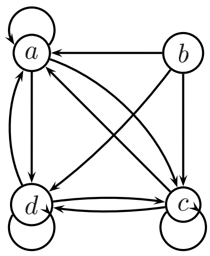
Given a directed Graph $G = (V, E)$, we would like to know whether there is a path in G from $u \in V$ to $v \in V$ for all vertex pairs $u, v \in V$. The transitive closure answers the question. The transitive closure of G is the graph $G^* = (V, E^*)$, where

$$E^* = \{(u, v)| \text{ there is a path from } u \text{ to } v \text{ in } G.\}$$



$G = (V, E)$

Procedure TRANSITIVE CLOSURE is similar to procedure FLOYD for all pairs shortest paths that uses the dynamic programming technique. The main difference is that here the edges do not have weights.



Transitive closure of G

We are not interested in finding the shortest paths, but rather if paths exist. We define a matrix D such that entry $d_{ij} = 1$ if there is a path from i to j, and $d_{ij} = 0$ if no such path exists. We are assuming that $V = \{1, 2, \ldots, n\}$. Procedure TRANSITIVE-CLOSURE*(G)* will progressively build D.

Initially, D = incidence matrix of G with ones in main diagonal. At the k^{th} iteration of procedure TRANSITIVE-CLOSURE*(G)*, the entry $d_{ij}^{(k)}$ of D will be

one if there is a path from vertex i to vertex j with all intermediate vertices in $\{1, 2, \ldots, k\}$, and

zero otherwise

$G^* = (V, E^*)$ is thus constructed by putting edge (i, j) in E^* if and only if $d_{ij}^{(n)} = 1$. The recursive definition of d_{ij}^{k} is :

$$d_{ij}^{(0)} = \begin{cases} 0 \text{ if } i \neq j \text{ and } (i,j) \notin E \\ 1 \text{ if } i = j \text{ or } (i,j) \in E \end{cases}$$

and

$$d_{ij}^{(k)} = d_{ij}^{(k-1)} \vee (d_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)}) \quad \text{for } k \geq 1$$

As in the FLOYD algorithm, procedure TRANSITIVE-CLOSURE *(G)* computes the matrices D with entries $d_{ij}^{(k)}$ in order of increasing k.

TRANSITIVE-CLOSURE*(G)*

---

begin
  $n := |V|$
  for $i := 1$ **to** n
    for $j := 1$ **to** n
      if $i = j$ **or** $(i, j) \in E$
        then $d_{ij}^{(0)} := 1$
        else $d_{ij}^{(0)} := 0$
  for $k := 1$ **to** n
    for $i := 1$ **to** n
      for $j := 1$ **to** n
        $d_{ij}^{(k)} := d_{ij}^{(k-1)} \vee (d_{ik}^{(k-10)} \wedge d_{kj}^{(k-1)})$
  return D *entries of D are $d_{ij}^{(n)}$*
end

The time complexity of the procedure is $\Theta(n^3)$, where $|V| = n$.

Johnson's all-pair shortest-paths algorithm

Johnson's algorithm finds shortest paths between all pairs in $O(|V|^2 \log_2 V + |V||E|)$ time. It makes use of Dijkstra's algorithm and Bellman-Ford's algorithm.

Assume that $G = (V, E)$ is a weighted, directed graph and that $w : E \to \mathbf{R}$ is the weight function. If G has negative weight-edges, we use a new weight function $\overline{w}$ to recompute the weights of the edges of G.

Ideally we would like to define $\overline{w} : E \to \mathbf{R}$ such that:

1. for all pairs of vertices $u, v \in V$, a shortest path from u to v using weight function w is also a shortest path from u to v using $\overline{w}$.

2. for all edges (u, v), the new weight $\overline{w}(u, v)$ is nonnegative.

Let $\overline{\delta}$ denote the shortest-path weights derived from $\overline{w}$. The process of recomputing the weights (according to $\overline{w}$) is called reweighting. In other words, we would like to reweight the edges such that:

1. shortest paths are preserved, and

2. we don't have any negative weight edges.

Recall, $G = (V, E)$ is a weighted, directed graph with weight function $w : E \to \mathbf{R}$. Let $h : V \to \mathbf{R}$ be any function mapping, and define:

$$\overline{w}(u, v) = w(u, v) + h(u) - h(v)$$

for all edges (u, v) in G.

Claim: $\overline{w} : E \to \mathbf{R}$ as defined above preserves shortest paths.
In other words, if $v_0 \to v_1 \cdots \to v_k$ is a path P in G then

$$w(P) = \delta(v_0, u_k) \Leftrightarrow \overline{w}(P) = \overline{\delta}(v_0, v_k)$$

First, let us write $\overline{w}(P)$ as a function of $w(P)$.

$$\begin{aligned}
\overline{w}(P) &= \overline{w}(v_0, v_1) + \overline{w}(v_1, v_2) + \cdots + \overline{w}(v_{k-1}, v_k) = \\
&= [w(v_0, v_1) + h(v_0) - h(v_1)] + [w(v_1, v_2) + h(v_1) - h(v_2)] + \\
&\quad + \cdots + [w(v_{k-1}, v_k) + h(v_{k-1}) - h(v_k)] \\
&= [w(v_0, v_1) + h(v_0)] + [w(v_1, v_2)] + \cdots + [w(v_{k-1}, v_k) - h(v_k)] \\
&= w(P) + h(v_0) - h(v_k)
\end{aligned}$$

Now we show that reweighting does not change shortest paths, that is

$$w(P) = \delta(v_0, v_k) \Leftrightarrow \overline{w}(P) = \overline{\delta}(v_0, v_k)$$

Proof

($\Rightarrow$) the proof is by contradiction. Suppose that $\overline{w}(P) = \overline{\delta)}(v_0, v_k)$ is not the shortest path from v_0 to v_k under function $\overline{w}$.

So there is a shorter path P' from v_0 to v_k using $\overline{w}$. That is $\overline{w}(P') < \overline{w}(P)$. But with $\overline{w}(P') < \overline{P}$

$$\begin{aligned}
&\Rightarrow \quad w(P') + h(v_0) - h(v_k) < w(P) + h(v_0) - h(v_k) \\
&\Rightarrow \quad w(P') < w(P)
\end{aligned}$$

which contradicts the assumption that P is a shortest path from v_0 to v_k using w.

($\Leftarrow$) similar argument to the one above $\square$

Remark: We can use a similar argument to show that

$$\begin{array}{c} G \text{ has a negative-weight} \\ \text{cycle using } w \end{array} \Leftrightarrow \begin{array}{c} G \text{ has a} \\ \text{negative-weight} \\ \text{cycle using } \overline{w} \end{array}$$

Reweighting starts by augmenting V. Given a weighted, directed graph $G = (V, E)$, and $w : E \to \mathbf{R}$, we construct $G' = (V', E')$ where:

$$V' = V \cup \{s\}, S \notin V \text{ and } E' = E \cup \{(s, v) | v \in V\}$$

with $w(s, v) = 0$ for all $v \in V$.

Note that:

$$\begin{array}{c} G' \text{ has no negative–} \\ \text{weight cycles} \end{array} \Leftrightarrow \begin{array}{c} G \text{ has no} \\ \text{negative-weight} \\ \text{cycles} \end{array}$$

Let $h(v) = \delta(s, v)$ for all $v \in V'$. Recall that for all edges $(u, v) \in E$ in a directed graph, we have:

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

So $h(v) \leq h(u) + w(u, v)$

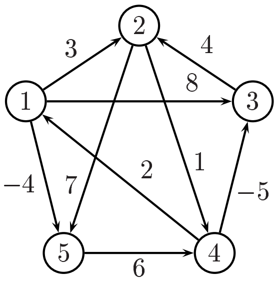
$$\begin{aligned} &\Rightarrow \quad w(u, v) + h(u) - h(v) \geq 0 \\ &\Rightarrow \quad \overline{w}(u, v) \geq 0 \end{aligned}$$

And so the second property (of not having negative-weight edges) is satisfied. Thus, $\overline{w} : E \to \mathbf{R}$, where $\overline{w}(u, v) = w(u, v) + h(u) - h(v)$ satisfies the two conditions mentioned earlier.

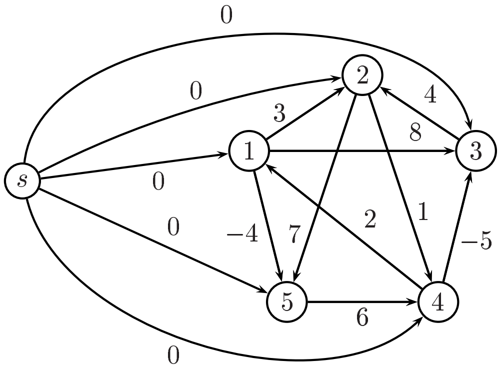
Procedure Johnson assumes that the edges are stored in an adjacency lists. The procedure returns the $|V| \times |V|$ matrix D where entry $d_{ij} = \delta(i, j)$ for $i, j = 1, 2, \ldots, |V|$. The procedure assumes, that the vertices are numbered from 1 to $|V|$.

```
JOHNSON(G)
begin
  construct G' = (V', E'), where V' = V ∪ {s} ,
  E' = E ∪ {(s,v)|v ∈ V} and w(s,v) = 0 for v ∈ V
  for each vertex v ∈ V'
    run the Bellman-Ford algorithm           since we might have
    to compute δ(s,v)                        negative-weight edges
    h(v) ← δ(s,v)
  for each edge (u,v) ∈ E'
    w̄(u,v) ← w(u,v) + h(u) − h(v)
  for each vertex u ∈ V
    run DIJKSTRA(G,u)                        with weights as
    to compute δ̄(u,v)                        computed by w̄
    for all v ∈ V
    for each vertex v ∈ V
      d_uv ← δ̄(u,v) + h(v) − h(u)
  return D
end
```

Example: Consider the following graph $G = (V, E)$.



First we augment the graph $G = (V, E)$ by adding one vertex s.

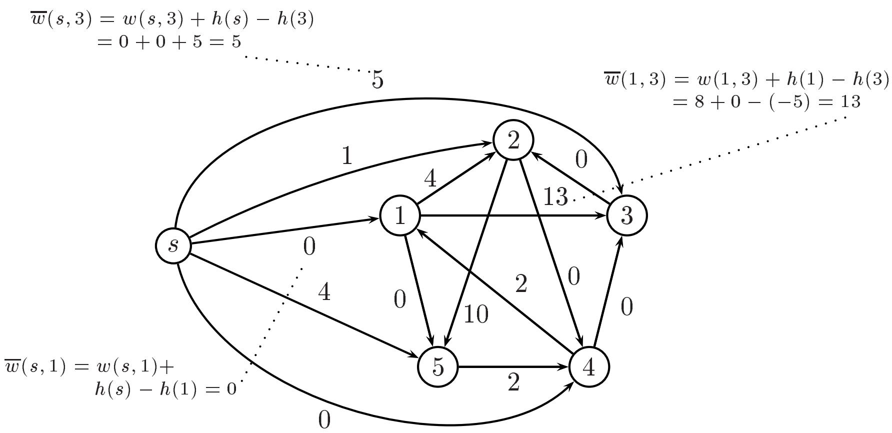


With procedure BELLMAN-FORD we obtain the following table

i	$h(i)$ $= \delta(s, i)$
1	0
2	-1
3	-5
4	0
5	-4

and $h(s) = 0$.

The next figure shows G' after reweighting with weight function $\overline{w}$, where $\overline{w}(u, v) = w(u, v) + h(u) - h(v)$.



The matrix of all shortest paths produced by procedure JOHNSON with above example is

$$D = \begin{bmatrix} 0 & 2 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 & 2 \\ 2 & 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 & 2 \\ 4 & 2 & 2 & 2 & 0 \end{bmatrix}$$

If the priority queue in Dijkstra's algorithm is implemented by a Fibonacci-Heap, then the running time of Johnson's algorithm is

$$O(|V|^2 \log_2 |V| + |V||E|).$$

Note that Johnson's algorithm is efficient for sparse matrices.

In the next section, we study splay trees.

6.2 Splay Trees

A splay tree is a self-adjusting binary search tree. With self-adjusting data structures, we allow them to be in an arbitrary state, but during each operation, we apply a simple restructuring rule intended to improve the efficiency of future operations. Recall path compression for instance.

Self -adjusting data structures have several possible advantages over balanced or explicitly constrained data structures:

a) in an amortized sense, ignoring constant factors, they are never much worse than constrained structures, and since they adjust according to usage, they can be much more efficient if the usage pattern is skewed

b) they need less space, since no balance or other constraint information need to be stored

c) their access and update algorithms are easy to implement

Self-adjusting data structures have two potential disadvantages

a) they require more local adjustment, for example look-up operations during accesses; other structures need adjusting only during updates, not during accesses

b) individual operations within a sequence can be expensive, which is a problem for real time applications

Many variants of the simple binary search tree were developed to try and improve on the poor worst case behavior of search trees. The basic operations on binary search trees take $O(h)$ where h is the height of the tree. A complete binary tree with n elements has $h = \log_2 n$. But this is not always maintained as the tree undergoes various operations. One needs to balance the tree (and thus change the height of the tree) to achieve good performances so that the binary search tree can be an effective data structure.

AVL trees, $2-3$ trees and red-black trees are among a host of binary search trees that were developed so as to improve the performance of binary search trees. Splay trees were designed for that purpose, too. The goal of splay trees is to have the tree maintain a logarithmic depth in an amortized sense by adjusting its structure with each access.

Splay trees are reorganized whenever they are accessed or updated. The basic reorganizing operation (splaying) moves the accessed node towards the root by a sequence of rotations. Therefore, frequently accessed keys tend to be near the root. For the worst sequence of splaying, the number of operations is $O(\log_2 n)$ per node in the tree where n is the number of nodes.

While other variants of binary search trees often aggressively pursue maintaining a balanced tree, splay trees deal with the problem lazily, doing a little bit of work with each tree operation, and making little obvious effort to maintain a nicely balanced tree.

If we do an amount of work $c > O(\log_2 n)$, we show that there were a lot of operations in the recent past where the work done was less than allowed and so that we can amortize c by spreading it around to previous performed operations. In other words, any additional work that we do, beyond $O(\log_2 n)$, can be accounted for by our past laziness – a typical amortized example.

The two basic elements of splay trees are self-adjustment and rotation. Splay trees use a double rotation to distribute some of the weight below the queried node over its sibling. The total amount of imbalance, which is measured by the sum of ranks, is a potential function against which we charge the cost of long searches.

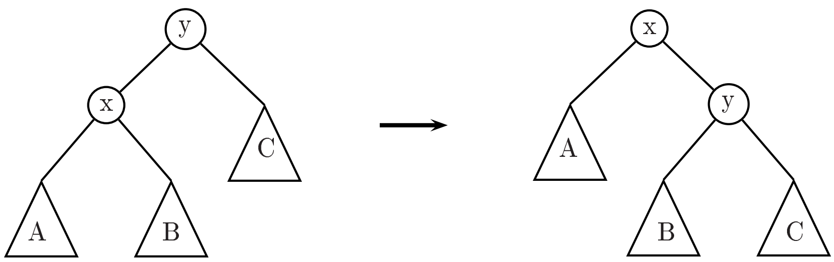
A splay tree is a binary search tree with the following tree splaying operations:

1. single rotation: ZIG
2. double rotations: ZIG-ZIG and ZIG-ZAG

6.2.1 Tree Rotations

Tree rotations can be thought of as a way to move a node to a higher position in a binary search tree without affecting the ordering properties.

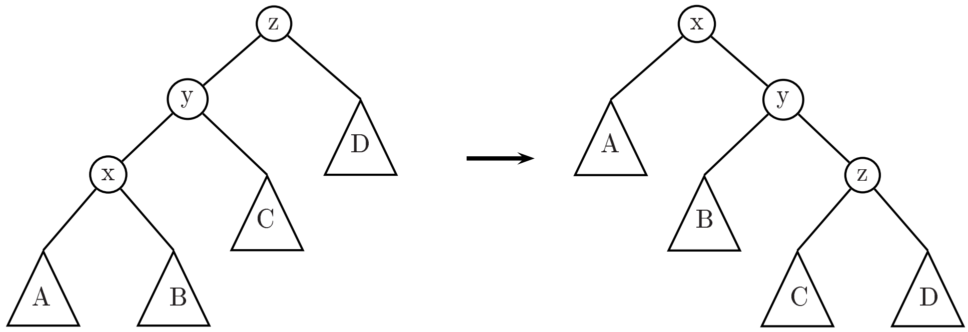
1. Single rotation: ZIG
   Splay step at x where x is the child of the rood node



   Single rotations involve two nodes and their corresponding subtrees. Note that the rotation brings node x to the top of the tree (and the resulting tree is still a binary search tree).

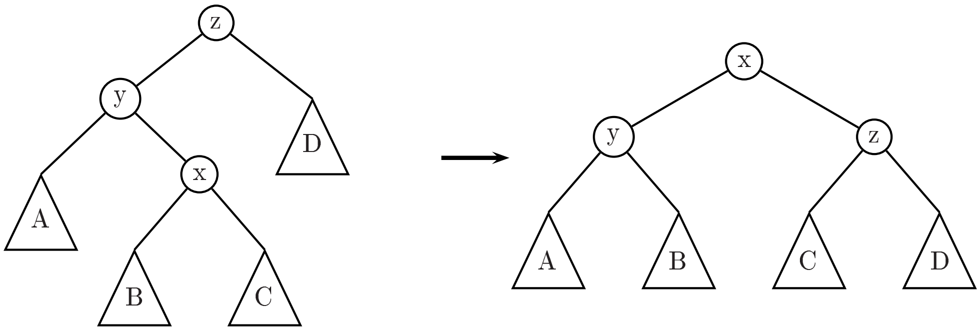
2. Double rotation
   Three nodes and their subtrees are involved with double rotations. A double rotation essentially performs single rotations in sequence.

   (a) ZIG-ZIG
   The parent of x is not the root node, and x and its parent are **left** children.



The node x is brought up two levels to the top of the tree (or subtree).

(b) ZIG-ZAG
The parent of x is the left child of the root node. x is the right child of its parent.



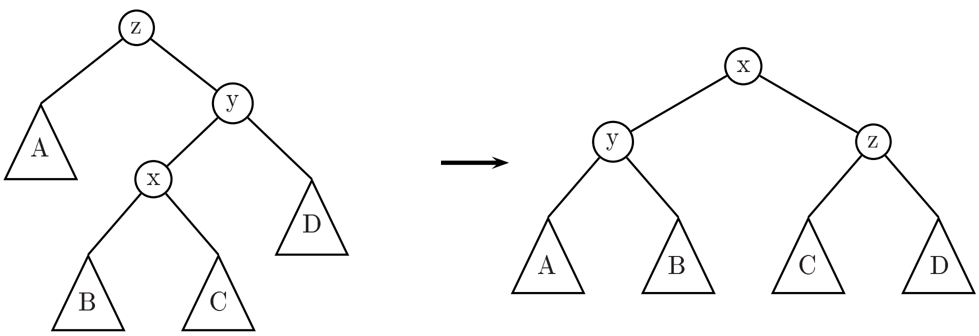
In this case, any weight below x is spread more evenly as x is brought to the root of the tree (subtree).

The double rotation is one of the key elements that make splay trees successful. While single rotations can move a queried element to the top of a tree without affecting key ordering, only double rotations produce the balancing properties of trees.

Remarks:

1. it is sufficient to look at the local structure of the tree defined by x, x's parent and x's grandparent to decide on one of the tree rules to follow. One has to continue applying the rules until x eventually reaches the top of the tree.

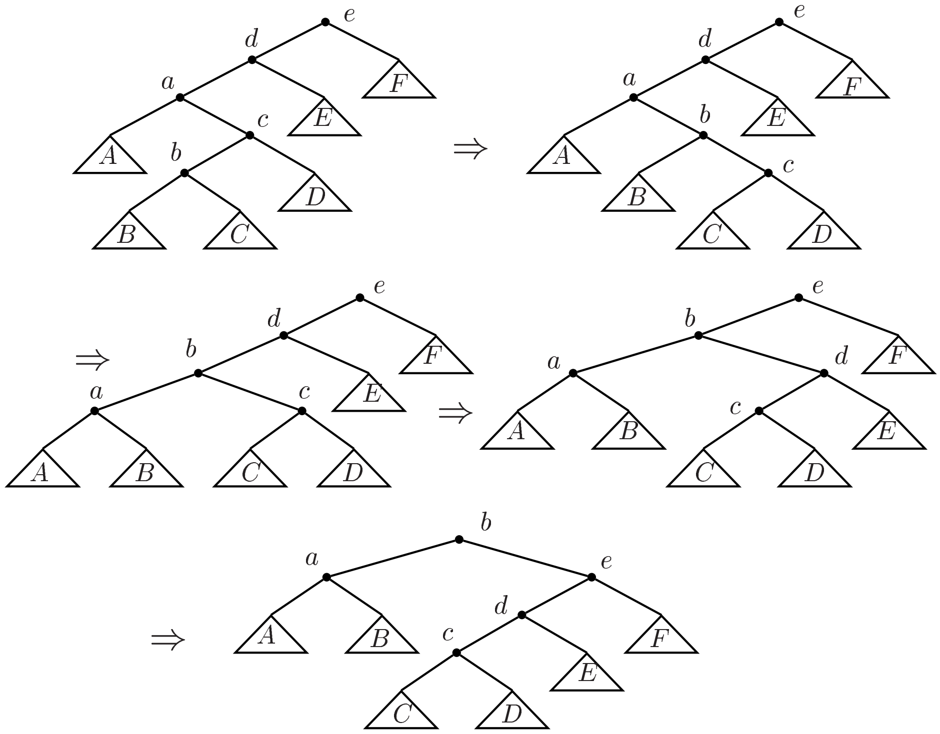
2. each rule has a mirror image variant, which covers all the cases. ("read" the figures from right to left for ZIG and ZIG-ZIG; for the ZIG-ZAG consider the following figure)



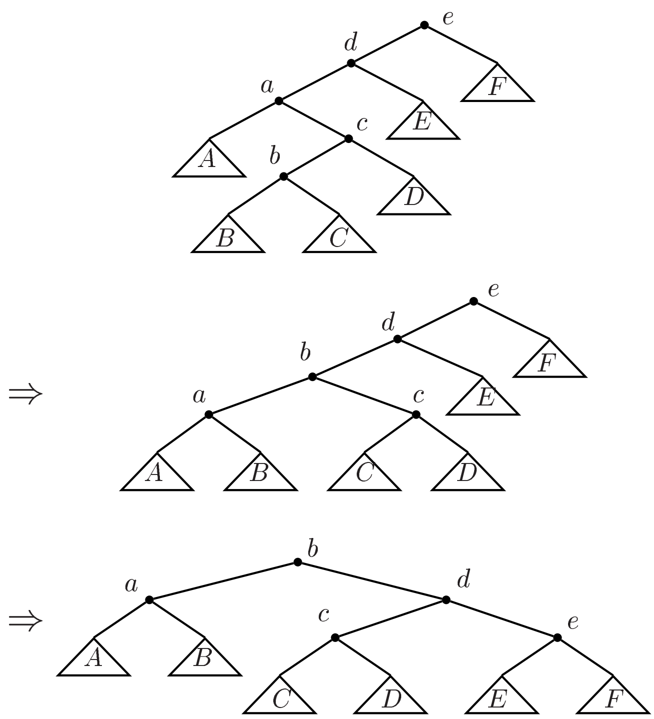
3. the ZIG-ZIG rule is the one that distinguishes splaying from just rotating x to the root of the tree

4. the three rules just studied are all bottom-up splaying rules

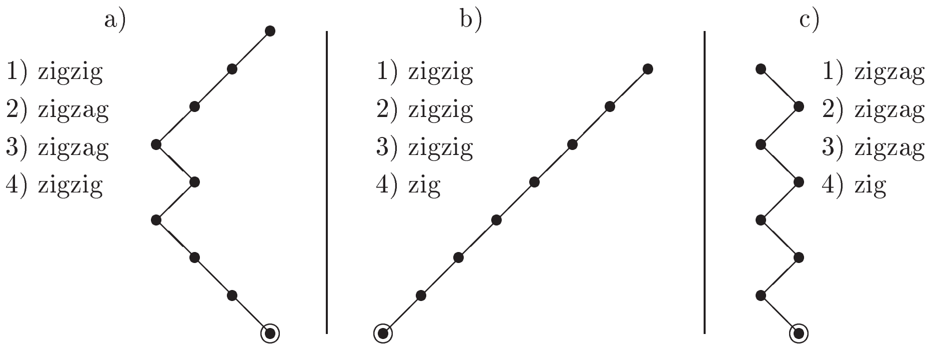
There is a big different between splaying and move–to–root.
EXAMPLE: Move–to–root. Simple rotations are used to make node b the root node.



EXAMPLE: Splaying. To splay a node b, repeat the splay step on b until it is the root.



EXAMPLE: Performing the following operations on each of the three trees (as indicated) will splay the encircled element to the root node.



6.2.2 Analysis of Splay Tree Operations

Splay trees run all basic operations in $\log_2 n$ time. Splaying at a node of depth d takes $\Theta(d)$ time. We prove this through the manipulation of weights and a potential function. Let

c_k	denote the actual cost of operation k
$\overline{c_k}$	denote the amortized cost of the k^{th} operation
D_k	denote the state of the data structure (the tree) after applying the k^{th} operation on D_{k-1}
$\Phi(D_k)$	denote the potential associated with D_k

We know that

$$\begin{aligned} \overline{c_k} &= c_k + \Phi(D_k) - \Phi(D_{k-1}) \\ \Rightarrow\ c_k &= \overline{c_k} - \Phi(D_k) + \Phi(D_{k-1}) \end{aligned}$$

and the actual amount of work required is given by

$$\begin{aligned} \sum_{k=1}^{m} c_k &= \sum_{k=1}^{m} (\overline{c_k} - \Phi(D_k) + \Phi(D_{k-1})) \\ &= \Phi(D_0) - \Phi(D_m) + \sum_{k=1}^{m} \overline{c_k} \end{aligned}$$

and similarly

$$\sum_{k=1}^{m} \overline{c_k} = \Phi(D_m) - \Phi(D_0) + \sum_{k=1}^{m} c_k$$

Potential function for splay trees: Let

$w(x)$	weight of node x, a fixed but arbitrary value
T_x	tree rooted at x
$y \in T_x$	y is a node in tree T_x
$s(x) = \sum_{y \in T_x} w(y)$	s is the size of the weights of all descendants of x, including the weight of x itself
$r(x) = \log_2(s(x))$	r is the rank of x
$\Phi(T) = \sum_{x \in T} r(x)$	T is the data structure D and Φ is the potential function we use for proving properties about splay trees

Remarks:

1. performing a rotation between a pair of nodes x and y only affects the ranks of the nodes x and y, and no other nodes in the tree
2. if $w(y) = 1$ for all $y \in T$, then the potential of a balanced tree is $O(n)$, and at the other extreme, the potential of a long chain (most unbalanced tree) is $O(n \log_2 n)$

We shall need the following lemma later.

Lemma 1: *If b is a node in the tree with children a and c, then*

$$r(a) + r(c) - 2r(b) + 2 \leq 0$$

Proof: By definition,

$$s(b) = s(a) + s(c) + w(b).$$

Weights are positive, hence

$$\begin{aligned} & s(b) \geq s(a) + s(c) \\ \Rightarrow\quad & [s(b)]^2 \geq (s(a) + s(b))^2 \\ \Rightarrow\quad & [s(b)]^2 - 4s(a)s(c) \geq (s(a) - s(c))^2 \\ \Rightarrow\quad & [s(b)]^2 - 4s(a)s(c) \geq 0 \quad \text{since } (s(a) - s(b))^2 \geq 0 \\ \Rightarrow\quad & [s(b)]^2 \geq 4s(a)s(c) \\ \Rightarrow\quad & \log_2 [s(b)]^2 \geq \log_2 (4s(a)s(c)) \\ \Rightarrow\quad & 2\log_2 (s(b)) \geq 2\log_2 2 + \log_2 (s(a)) + \log_2 (s(c)) \end{aligned}$$

So by definition of $r(a)$, we get

$$2r(b) \geq 2 + r(a) + r(c)$$

so we have

$$r(a) + r(c) - 2r(b) + 2 \leq 0$$

$\square$

We consider the number of rotations. Let s and r denote the size and rank before a splay, and s' and r' denote the size and rank after the splay.

Lemma 2:(*access theorem*) *The amortized time for each rotation involved in the splaying of node x is at most*

$$3(r'(x) - r(x)) \ ,$$

and the time for a single rotation is at most

$$1 + 3(r'(x) - r(x)).$$

Proof:

Case 1: ZIG

The amortized time:

$$\begin{aligned} \overline{c_k} &= c_k + \Phi(T_k) - \Phi(T_{k-1}) \\ &= 1 + [r'(x) + r'(y)] - r(x) - r(y) \end{aligned}$$

since only x and y change ranks. But $r'(x) = r(y)$, hence

$$\overline{c_k} = 1 + r'(y) - r(x)$$

since y is below x in T_k, we have $r'(y) < r'(x)$. So $\overline{c_k} = 1 + r'(y) - r(x)$ with $r'(y) = r'(x) \Rightarrow \ \overline{c_k} < 1 + r'(x) - r(x)$. Since $r'(x) > r(x)$, we have

$$\begin{aligned} \overline{c_k} &< 1 + (r'(x) - r(x)) < 1 + 3(r'(x) - r(x)) \\ &\Rightarrow \ \overline{c_k} < 1 + 3(r'(x) - r(x)) \end{aligned}$$

Case 2: ZIG-ZIG

Here we have two rotations. So $c_k = 2$ and only x, y and z change ranks.

$$\begin{aligned} \overline{c_k} &= c_k + \Phi(T_k) - \Phi(T_{k-1}) = \\ &= 2 + [r'(x) + r'(y) + r'(z)] - [r(x) + r(y) + r(z)] \end{aligned}$$

We have $r'(x) = r(z)$, so $\overline{c_k} = 2 + r'(y) + r'(z) - r'(x) - r(y)$, but $r'(x) < r'(y)$ in T_k and $r(y) > r(x)$ in T_{k-1}, therefore

$$\begin{aligned} \Rightarrow \ \overline{c_k} &= 2 + r'(y) + r'(z) - r(x) - r(y) \\ &< 2 + r'(x) + r'(z) - r(x) - r(x) \\ \Rightarrow \ \overline{c_k} &< 2 + r'(x) + r'(y) - 2r(x) \end{aligned}$$

Add and subtract $2r'(x)$ and $r(x)$

$$\begin{aligned} \overline{c_k} &< 2 + r'(x) + r'(z) - 2r(x) + 2r'(x) - 2r'(x) + r(x) - r(x) \\ \overline{c_k} &< 2 + r(x) + r'(z) - 2r'(x) + 3r'(x) - 3r(x) \\ \Rightarrow \ \overline{c_k} &< [r(x) + r'(x) - 2r'(x) + 2] + 3\,[r'(x) - r(x)] \end{aligned}$$

but $r(x) + r'(x) - 2r'(x) + 2 \leq 0$ by Lemma 1. So

$$\overline{c_k} < 3(r'(x) - r(x))$$

Case 3: ZIG-ZAG

Here too $c_k = 2$, since we have 2 rotations. Only x, y and z change ranks and $r'(x) = r(z)$. So

$$\begin{aligned}
\overline{c_k} &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) = \\
&= 2 + r'(y) + r'(z) - r(x) - r(y) \\
\overline{c_k} &< 2 + r'(y) + r'(z) - r(x) - r(x) \text{ since } r(x) < r(y) \\
\Rightarrow \overline{c_k} &< 2 + r'(y) + r'(z) - 2r(x)
\end{aligned}$$

Add and subtract $2r'(x)$. Hence

$$\begin{aligned}
\overline{c_k} &< 2 + r'(x) + r'(x) - 2r(x) + 2r'(x) - 2r'(x) \\
&< [2 + r'(y) + r'(z) - 2r'(x)] + 2r'(x) - 2r(x) \\
&< [r'(y) + r'(z) - 2r'(x) + 2] + 2(r'(x) - r(x))
\end{aligned}$$

Again, by Lemma 1 we have:

$$r'(y) + r'(z) - 2r'(x) + 2 \leq 0$$

So $c_k < 2(r'(x) - r(x))$ and thus $\overline{c_k} < 3(r'(x) - r(x))$ □

We showed that the amortized cost of

- the ZIG step is bounded by

$$3(r'(x) - r(x)) + 1 \text{ , and of}$$

- the ZIG-ZIG and ZIG-ZAG steps by

$$3(r'(x) - r(x)).$$

We can now sum over all the rotations that are needed to splay x to the root.

Lemma 3: *The amortized time to splay a node x to the root is*

$$O\left(\log_2 \frac{s(t)}{w(x)}\right)$$

where $s(t)$ is the sum of all the weights in the tree rooted at t and $w(x)$ is the weight of x.

Proof: The amortized cost of splaying x to the root t is

$$1+3(r(t)-r(x_{k-1}))+3(r(x_{k-1}-r(x_{k-2}))+3(r(x_{k-3})-r(x_{k-4})+\cdots+3(r(x_1-r(x)) =$$

$$= 1 + 3(r(t) - r(x))$$

Note that $+1$ comes from the ZIG step that can happen only **once**. Here x, x_1, x_2, $x_3, \ldots, x_i, \ldots, x_{k-1}, t$ are roots of subtrees that are rotated. Recall $r(t) = \log_2(s(t))$ and $r(x) = \log_2(s(x))$. So

$$\begin{aligned}
3(r(t) - r(x)) &= O(r(t) - r(x)) = \\
&= O(\log_2(s(t)) - \log_2(s(x))) = \\
&= O\left(\log_2 \frac{s(t)}{s(x)}\right)
\end{aligned}$$

Since $w(x) \leq s(x)$ we have

$$\log_2 \frac{s(t)}{w(x)} < \log_2 \frac{s(t)}{w(x)}$$

$$\Rightarrow\ 3(r(t) - r(x)) = O\left(\log_2 \frac{s(t)}{w(x)}\right)$$

Let $W = s(t)$, i.e., the sum of all the weights of the nodes rooted at t. So the amortized time to splay x to the root is

$$O\left(\log_2 \frac{W}{w(x)}\right).$$

Lemma 4:(*balance theorem*) *A sequence of m splays in a tree of n nodes takes time*

$$O((m+n)\log_2 n).$$

Proof: From Lemma 3 we know that the amortized cost of splaying x to the root t is $1+3(r(t)-r(x))$. Let $w(y) = 1$ for all nodes $y \in T$. For a given splay, $r(t) \leq \log_2 n$ and $r(x) \geq 0$. So the amortized cost of the splay is at most $1 + 3\log_2 n$.

$$\sum_{k=1}^{m} \overline{c_k} = \Phi(T_m - \Phi(T_0) + \sum_{k=1}^{m} c_k$$

But

$$\begin{aligned} \Phi(T_0) &\leq n\log_2 n \quad \text{in case tree is skewed} \\ \Phi(T_m) &\geq 0 \\ \Rightarrow \sum_{k=1}^{m} c_k &\leq n\log_2 n + m(3\log_2 n + 1) \end{aligned}$$

Therefore $\sum_{k=1}^{m} c_k$ is $O(m\log_2 n + n\log_2 n)$. So the sequence of m splays in a tree of n nodes takes time

$$O((m+n)\log_2 n)$$

Lemma 4, the balance theorem, states that on a sufficiently long sequence of accesses a splay tree is as efficient as any form of an uniformly balanced tree.

Remarks:

1. Note that by Lemma 3 we know that the amortized time to splay x to the root is $O\left(\log_2 \frac{W}{w(x)}\right)$. So if $w(y) = 1$ for all nodes in the tree, then $W = n$. Therefore in a tree with n nodes, the access time per operation is

$$O\left(\log_2 \frac{W}{w(x)}\right) = O\left(\log_2 \frac{n}{1}\right) = O(\log_2 n)$$

2. If we now let $w(y) = \frac{1}{3^d}$ for all $y \in T$ where d is the depth of node y in the tree, then

$$W = \sum_{y \in T} w(y) \leq \sum_{i=1}^{n} 2^i 3^{-d} \leq 1$$

So the amortized cost to access a node that is found in depth d of a splay tree is

$$O\left(\log_2 \frac{W}{w(x)}\right) = O\left(\log_2 \frac{1}{3^{-d}}\right) = O(d).$$

This shows that splaying is competitive with any fixed binary search tree.

3. As a matter of fact, splaying is competitive with the best possible tree. Let m denote the total number of accesses of a tree. Let p_x denote the fraction of times x will be accessed (compared to all other accesses of the rest of the nodes in the tree). Note $p_x m$ denotes the access frequency of node x. It is known that the optimal access time is

$$\Omega(m \sum_{xinT} p_x \log_2 \frac{1}{p_x})$$

Now, if we go back to Lemma 3 and let $w(y) = p_y$, then the amortized cost for x is

$$O\left(\log_2 \frac{W}{w(x)}\right) = O\left(\frac{1}{p_x}\right).$$

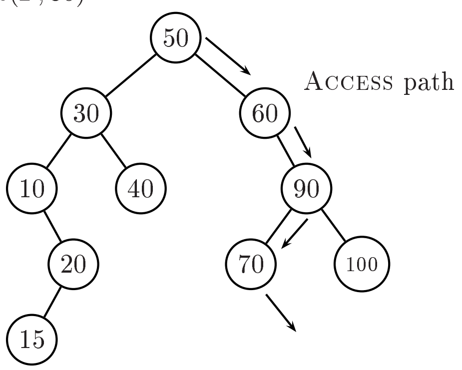
This sum is the entropy of a probability distribution. If we think of code lengths of items instead of node depths, then the above equation is the average code length required to send information about m items given the underlying probability distribution. This result, known as the static optimality theorem, implies that a splay tree is as efficient as any fixed search tree.

6.2.3 Update Operations on Splay Trees

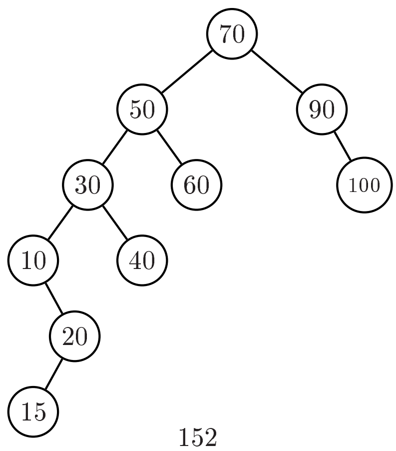
Using splaying, we can implement all the standard update operations on binary search trees.

access(T, i) If item i is in the tree, return a pointer to its location, otherwise return a pointer to the null node. Search down the tree for i. If the search reaches a node x containing i, we complete the access by splaying at x and returning a pointer to x. If i is not in T, we splay the last nonnull node reached during the search and return a pointer to null. Splaying's effect of moving a designated node to the root considerably simplifies the updating of splay trees.

EXAMPLE: ACCESS$(T, 80)$

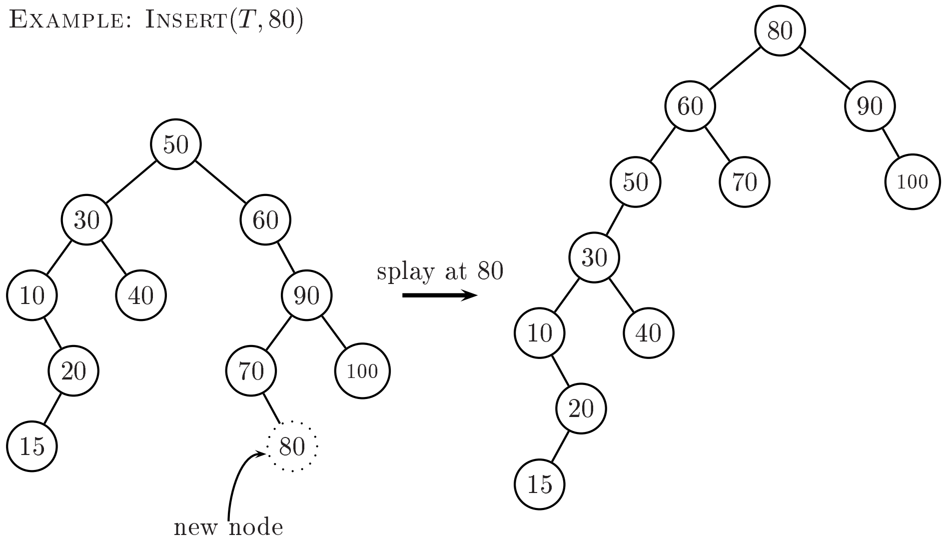


70 is the last nonnull node in the unsucessful search, so splay at 70.



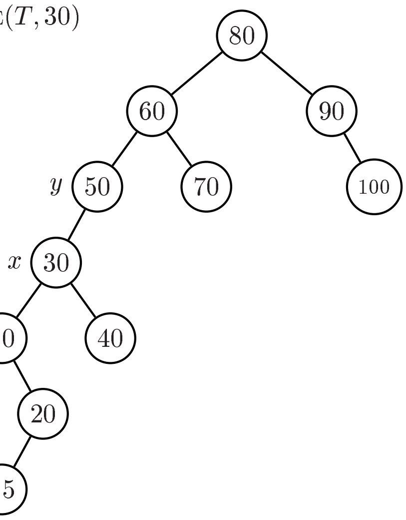
insert(T, i) Insert item i in tree T, assuming that it is not there already. One way of implementing $\text{INSERT}(T, i)$ is to search for i, then replace the pointer to null reached during the search by a pointer to a new node containing i, and finally splay the tree at the new node.

EXAMPLE: $\text{INSERT}(T, 80)$

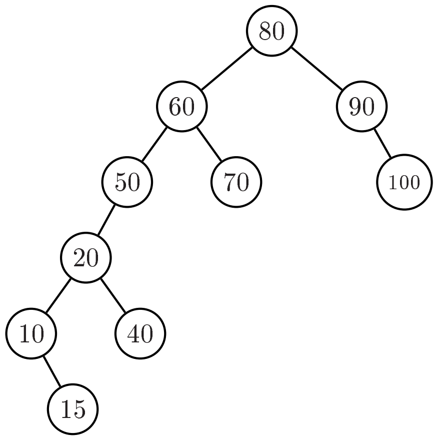


delete(T, i) Delete item i from T, assuming that it is present in T. One way of implementing $\text{DELETE}(T, i)$ is to search for the node containing i. If x is that node and y its parent, we replace x as a child of y by the join of the left and right subtrees of x, and then we splay at y.

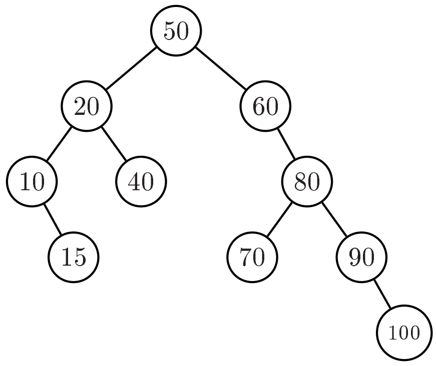
EXAMPLE: $\text{DELETE}(T, 30)$



We join the children of 30. Thus, we get



Now splay at 50 to obtain



join(T_1, i, T_2) Returns a tree where i is the root with T_1 as its left subtree and T_2 as its right subtree. It assumes that i is such that $t_1 < i < t_2$ for all $t_1 \in T_1$ and $t_2 \in T_2$. $\text{JOIN}(T_1, i, T_2)$ has amortized cost $O(\log_2 n)$. Note that we can also join two trees $T_1 < T_2$ by splaying the rightmost element of T_1 making T_2 its right subtree.

split(T, i) Returns two trees $T_1 \leq i < T_2$. Note that i simply defines a boundary between the two trees; i is not necessarily contained in T_1. $\text{SPLIT}(T, i)$ is implemented by splaying the greatest element less than i to the root. We rename the right subtree and call it T_3. The remaining tree is named T_1.

□

For any item i in a tree, let i^- and i^+, respectively, denote the item preceding i and the item following i in T. If i^- is undefined, $w(i^-) = \infty$ and if i^+ is undefined, let $w(i^+) = \infty$. The amortized times of the splay tree operations have the following upper bounds:

$$\begin{aligned} i^- &:= \text{largest element in T smaller than } i \\ i^+ &:= \text{smallest element in T larger than } i \end{aligned}$$

$$\text{ACCESS}(T, i) : \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + O(1), & \text{if } i \in T \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right) + O(1), & \text{if } i \notin T \end{cases}$$

$$\text{INSERT}_{i \notin T}(T,i) : 3\log\left(\frac{W - w(i)}{min\{w(i^-), w(i^+)\}}\right) + \log\left(\frac{W}{w(i)}\right) + O(1)$$

$$\text{DELETE}_{i \in T}(T,i) : 3\log\left(\frac{W}{w(i)}\right) + 3\log\left(\frac{W - w(i)}{w(i^-)}\right) + O(1), \text{ if } i \text{ not minimal in } T$$

$$\text{JOIN}(T_1, i, T_2) : 3\log\left(\frac{W}{w(i)}\right) + O(1), \ i \text{ maximal in } T_1$$

$$\text{SPLIT}(T,i) : \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + O(1), & \text{if } i \in T \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right) + O(1), & \text{if } i \notin T \end{cases}$$

Remarks:

1. We defined $\text{JOIN}(T_1, x, T_2)$. If we use $\text{JOIN}(T_1, T_2)$, we begin by accessing the largest element, say i in T_1. After the access, the root of T_1 contains i and thus has a null right child. The join is completed by making T_2 the right subtree of this root and returning the resulting tree.

2. Alternative implementation of DELETE and INSERT using SPLIT and JOIN.

   delete(T,i) We first perform $\text{SPLIT}(T,i)$ to obtain two subtrees T_1 and T_2. Note that since i was an element of T_1, it must now be the root of T_1. Since the root of T_1 does not have a left subtree, we may remove i in constant time. We then join the two trees T_1 and T_2. Since we have done a constant number of $O(\log_2 n)$ operations, the amortized cost for DELETE is $O(\log_2 n)$.

   insert(T,i) We first perform $\text{SPLIT}(T,i)$ to obtain two subtrees, T_1 and T_2. Then we $\text{JOIN}(T_1, i, T_2)$ to obtain the resultant tree. The amortized cost for INSERT is $O(\log_2 n)$.

□

In the next section, we study the problem of colosing the vertices and edges of graphs.

6.3 Graph Coloring

6.3.1 Vertex Coloring

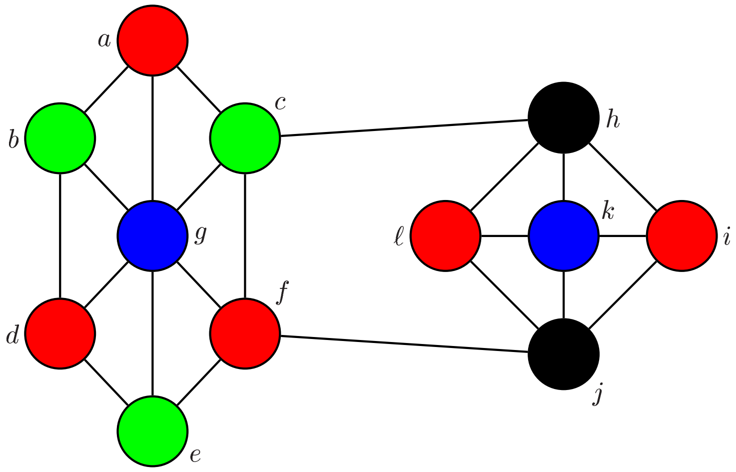
$G = (V, E)$ is an undirected graph. We are going to assign colors to vertices. The subset comprising all the vertices of a given color is a cell of a partition of V. Drawing the graph with colors assigned to the vertices is an intuitive way to represent such a partition.

Minimization problem of vertex coloring

Most applications involving vertex coloring are concerned with determining the minimum number of colors required under the condition that the endpoints of an edge of E cannot have the same color.

A vertex-coloring of G is an assignment $f : V \to C$ where C is the set of colors (usually denoted by $\{1, 2, \ldots, m\}$. G is said to be vertex k–colorable if it has a vertex-coloring using exactly k colors and such that the endpoints of each edge in E are assigned two different colors.

EXAMPLE: The following is an example of a vertex-coloring of a graph G.



In this example, we colored $\{a, d, f, l, i\}$ in red, $\{b, c, e\}$ in green, $\{g, k\}$ in blue and $\{h, j\}$ in black. But we can do better. By assigning green to k and the blue color to h and j we get a 3–coloring of G. $\square$

Note that in the literature, the term "coloring" is used instead of "vertex coloring" since vertex coloring is more frequently encountered than edge coloring (or map-coloring).

The chromatic number of a graph, denoted by $\chi(G)$, is the minimum number of different colors required for the vertex-coloring of G such that 2 endpoints of edges have different colors. In other word, the chromatic number of G is the smallest k for which G is vertex k–colorable. So $\chi(G) = k$ if G is k–colorable but not $(k-1)$–colorable. A graph is k–chromatic if $\chi(G) = k$.

Remarks:

1. A graph G has $\chi(G) = 1 \Leftrightarrow$ G has no edges
2. The study of vertex-coloring of graphs is usually restricted to simple graphs, since:
   (a) If G has a self-loop, then it is uncolorable since the endpoint of the self-loop is adjacent to itself, and
   (b) A multiple adjacency has no more effect on the color of its endpoints than a single adjacency

A color class in a vertex-coloring of a graph G is a subset of V containing all the vertices of some color.

Modelling applications as vertex-coloring problems

When an application is modelled as a vertex-coloring problem, the vertices sharing the same color (in the same color class) generally represent items (objects or individuals) that do not compete (or conflict) with each other.

EXAMPLES:

1. **Assignment of radio frequencies**
   The vertices represent transmitters for radio stations. Two stations are considered adjacent when their broadcast areas overlap, which results in interference if they broadcast at the same frequency. So the adjacent stations should be assigned different transmission frequencies (in other words, different colors). In this application, each color class contains vertices representing stations with

no overlap, and $\chi(G)$ is the minimum number of transmission frequencies required to avoid broadcast interference.

2. **Combustible chemicals**
   The vertices represent different kinds of chemicals needed in some manufacturing process. But some chemicals cannot be stored together since the might explode. So each pair of chemicals that might explode if combined, has an edge between them. The chromatic number here is the minimum number of different storage areas required so that no two chemicals that mix explosively are stored together.

3. **University course scheduling**
   The vertices represent the courses at a university. Two courses are adjacent if at least one student preregisters for both courses. These two courses should not be scheduled at the same time. So the chromatic number here is the minimum number of time periods in which to schedule the classes so that no student has a conflict between two courses.

4. **Fast register allocation**
   Some computers have a limited number of special registers that permit faster execution of arithmetic operations than ordinary memory locations (or even other ordinary registers). The program variables that are frequently used should be "register" storage class variables. But we should not declare more "register" variables than the actual number of fast registers in hardware, otherwise we will waste time due to swapping.

   One solution is to have the programmer control the register designation, so that variables that are going to be simultaneously active are assigned to different registers. So the graph model has one vertex for each variable, and two vertices are adjacent if the endpoint variables can be simultaneously active in some computation (or operation). The chromatic number is the minimum number of fast registers needed to avoid the overswapping phenomenon.

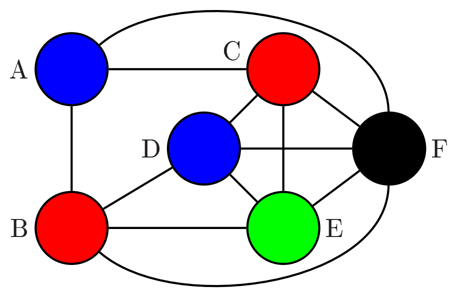
Lemma 1: *If G contains k mutually adjacent vertices, then $\chi(G) \geq k$.*
Proof: Using fewer than k colors would result in a pair from the mutually adjacent set of k vertices being assigned the same color. □

A clique in a graph $G = (V, E)$ is a maximal subset $S \subseteq V$ of mutually adjacent vertices. A clique number $\omega(G)$ of a graph $G = (V, E)$ is the number of vertices in a largest clique in G.

Lemma 2: *Let $G = (V, E)$ be a graph. Then $\chi(G) \geq \omega(G)$.*

Lemma 3: *If H is a subgraph of G then $\chi(G) \geq \chi(H)$.*

Example:



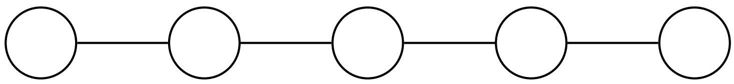
$S = \{C, D, E, F\}$ is a clique. $\omega(G) = 4$. So $\chi(G) \geq 4$. The graph is vertex 4–colorable since we can assign red to vertices $\{B, C\}$, blue to $\{A, D\}$, green to $\{E\}$ and black to $\{F\}$. So the graph is 4–chromatic and $\chi(G) = 4$. □

Chromatic numbers for common graph families

1. Complete graphs
   A complete graph with n vertices, K_n is a simple graph such that every pair of vertices is joined by an edge. By Lemma 1, $\chi(K_n) \geq n$ since K_n has n mutually adjacent vertices. On the other hand, K_n is vertex n–colorable. So $\chi(K_n) = n$.

2. Bipartite graphs
   A bipartite graph $G = (V, E)$ is a graph in which V can be partitioned into two subsets U_1 and U_2, such that each edge $e \in E$ has one endpoint in U_1 and one endpoint in U_2. The pair U_1, U_2 is the bipartition of G, and are also known as the bipartition subsets. By Lemma 1, $\chi(G) \geq 2$, and G is 2–colorable since we can use one color for the vertices of U_1 and another color for the vertices of U_2. So if G is a bipartite graph, then $\chi(G) = 2$.

3. Path graphs
   A path graph $G = (V, E)$, denoted by P_n, is a simple graph with $|V| = |E| + 1$ $(|V| = n)$ that can be drawn so that all of its vertices and edges lie on a single straight line. Since path graphs are bipartite graphs, $\chi(P_n) = 2$ for $n \geq 2$.

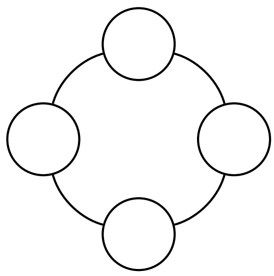
   

   path graph P_5

4. Trees
   Any nontrivial tree T is bipartite. So $\chi(T) = 2$.

5. Cycle graphs
   A cycle graph is a single vertex with a self-loop or a simple connected graph C with $|V| = |E|$ that can be drawn so that all of its vertices and edges lie on a circle. An n–vertex cycle graph is denoted by C_n.

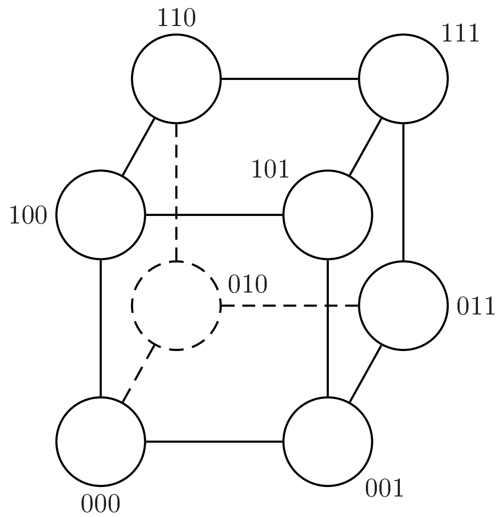
   (a) Even-cycle graphs
   Since an even-cycle graph, C_{2n}, is bipartite, $\chi(C_{2n}) = 2$.

   cycle graph C_4

   (b) Odd-cycle graphs
   If $v_1, v_2, \ldots, v_{2n+1}$ are the vertices of C_{2n+1} we would need 3 colors. If we use one color for $\{v_1, v_3, \ldots, v_{2n+1}\}$ and a second color for $\{v_2, v_4, \ldots, v_{2n}\}$, we would not have a 2–colorable graph, since v_1 and v_{2n+1} are adjacent. So we need 3 colors. Thus, $\chi(C_{2n+1}) = 3$.

6. Cube graphs
   A cube graph, Q_n, is a n–regular graph $Q_n = (V, E)$ (i.e. all vertices of V have degree n) whose vertex-set is the set of bitstrings of length n, and such that there is a n edge between two vertices if they differ in exactly one bit. Since the cube graph Q_n is bipartite, we have $\chi(Q_n) = 2$.

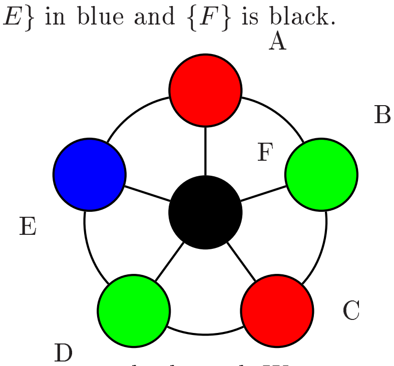


cube graph Q_3

7. Wheel graphs
   A wheel graph W_n, $n \geq 4$ is a cycle graph, C_{n-1}, to which we add one more vertex in the "center" and make it adjacent to all the vertices of C_{n-1}.

   (a) Odd-order wheel graph
   An odd-order wheel, W_{2n+1}, needs two colors for the even cycle graph C_{2n}, and then a third color to color the "center" vertex (since it is adjacent to every vertex of C_{2n}). Thus, $\chi(W_{n+1}) = 3$ for all $n \geq 1$.

   (b) Even-order wheel graph
   On the other hand, an even-order wheel, W_{2n}, needs three colors for its odd-cycle graph C_{2n-1}, plus a fourth color for the center vertex. So, $\chi(W_{2n}) = 4$.
   EXAMPLE: The vertices of W_6 in the figure are colored: $\{A, C\}$ in red, $\{B, D\}$ in green, $\{E\}$ in blue and $\{F\}$ is black.



wheel graph W_6

The results are summarized in the following table:

Graph G	$\chi(G)$
Complete graph K_n	n
Bipartite graph	2
Path graph P_n	2
Tree	2
Cycle graphs (even)	2
Cycle graphs (odd)	3
Cube graphs Q_n	2
Wheel graphs (odd order)	3
Wheel graphs (even order)	4

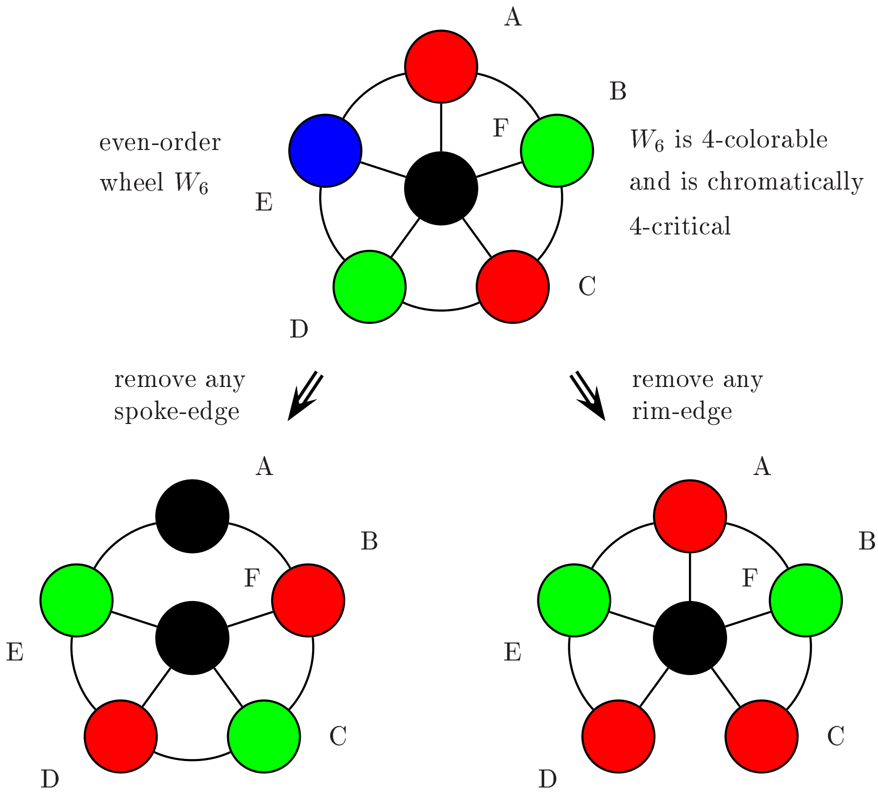
The rest of this section deals with non-common graph families, where the aim is to find good and tight bounds for $\chi(G)$.

If $G = (V, E)$ is a graph, then an edge-deletion subgraph G' is obtained by removing an edge from E. So, $G' = (V, E')$ where $E' = E \backslash \{e\}, e \in E$. A connected graph $G = (V, E)$ is chromatically k–critical if $\chi(G) = k$ and the edge-deletion subgraph G' is $(k-1)$–colorable, for every edge $e \in E$.

Examples:

1. An odd cycle is chromatically 3–critical, since the deletion of any edge will give a path graph and so the chromatic value goes from $\chi(G) = 3$ to $\chi(P) = 2$.
2. The even-order wheel W_6 is chromatically 4–critical. Recall $\chi(G) = 4$.

   even-order wheel W_6 W_6 is 4–colorable ($\{A, C\}$, $\{B, D\}$, $\{E\}$, $\{F\}$) and is chromatically 4–critical



G' is 3-colorable, ($\{A, F\}$, $\{B, D\}$, $\{E, C\}$) $\chi(G') = 3$

G'' is 3-colorable, ($\{A, D, C\}$, $\{E, B\}$, $\{F\}$) $\chi(G'') = 3$

$\square$

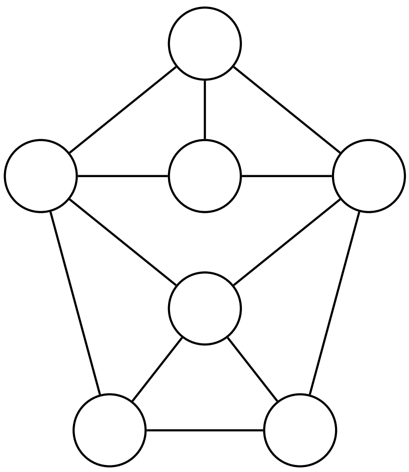
Similarly for vertex-deletion subgraphs. If $G = (V, E)$ is a graph, then a vertex-deletion subgraph G' is obtained by removing a vertex v from V, and consequently all edges in E that have v as endpoint. So $G' = (V', E')$ where $V' = V \backslash \{v\}$ and $E' = \{e \in E | v \text{ is not an endpoint of } e\}$. Since the deletion of a vertex will necessarily entail the removal of an edge, we have: If G is a chromatically k-critical graph and G' is a vertex-deletion subgraph obtained by removing any vertex of G, then G' is $(k-1)$-colorable.

Lemma 4: *If $G = (V, E)$ is a chromatically k-critical graph, then $deg(v) \geq k-1$ for all $v \in V$.*

Proof: The proof is by contradiction. Suppose that $\deg(v_1) < k-1$ for some $v_1 \in V$. Then the vertex-deletion subgraph G', obtained by removing v_1 from G is $k-1$-colorable. Since $\deg(v_1) < k-1$, v_1 has fewer than $k-1$ adjacent vertices and thus not all $k-1$ colors will be used on the adjacent vertices of v_1 in G'. Let c be a color not used on the adjacent vertices of v_1 in G'. Restore back G by adding v_1 to G' and color v_1 with color c. This means, that G is $(k-1)$-colorable which contradicts the fact that G is chromatically k-critical (and therefore $\chi(G) = k$). So no vertex of G has degree less than $k-1$. $\square$

An independent set of vertices in a graph is a set of mutually nonadjacent vertices. The vertex independent number of a graph G is the maximum cardinality of an independent set of vertices. It is denoted by $ind(G)$.

EXAMPLE: The following graph has $ind(G) = 2$. This means that the maximum size of a color class of vertices is two.



Since G has 7 vertices, there must be at least 4 color classes. $\square$

Lemma 5: *If G is any graph, then*

$$\chi(G) \geq \left\lceil \frac{|V|}{ind(G)} \right\rceil$$

Proof: Since each color class contains at most $ind(G)$ vertices, then the number of different color classes must be at least $\frac{|V|}{ind(G)}$ $\square$

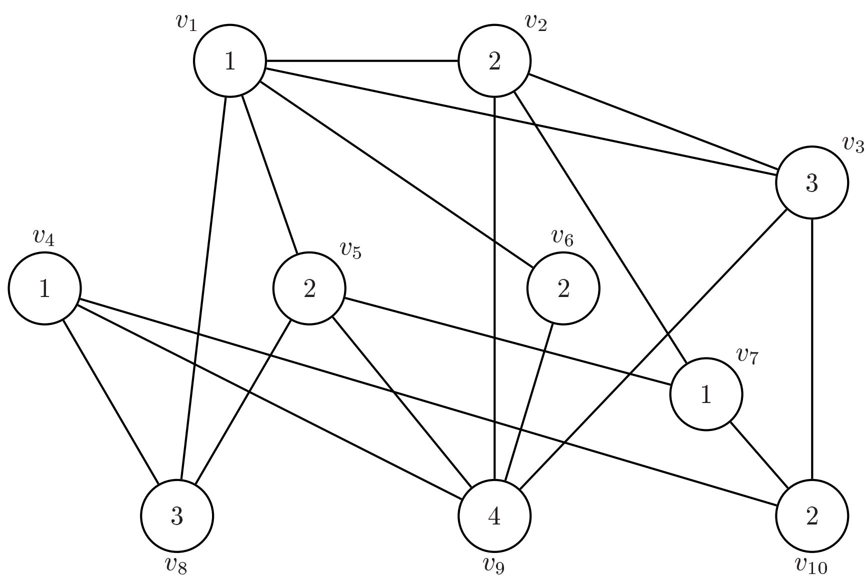
Sequential vertex-coloring algorithm

As mentioned in the chapter on Backtracking, the problem of calculating the chromatic number of a graph is known to be NP-hard. Deciding whether a graph is vertex 3-colorable is NP-complete.

SEQUENTIAL VERTEX-COLORING is a procedure that colors the vertices of a graph by always assigning the smallest color number that has still not been assigned to previously processed endpoints.

```
SEQUENTIAL VERTEX-COLORING
begin
  n ← |V|
  for i ← 1 to n
      let f(v_i) ← smallest color number
      not used on any of the smaller subscripted
      adjacent vertices of v_i
  return vertex-coloring f
end
```

The procedure assumes that $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices.
EXAMPLE:



By applying SEQUENTIAL VERTEX-COLORING to the above graph, we obtain a 4-coloring using colors 1, 2, 3 and 4. □

The color assignment in the example is not optimal. We could have used color 2 for v_4 and color 3 for v_7 (and thus not necessarily giving the smaller color number each time through the **for** loop). Then a 3-coloring can be obtained with the following color sets: $\{v_1, v_9, v_{10}\}$, $\{v_2, v_4, v_5, v_6\}$ and $\{v_3, v_7, v_8\}$.

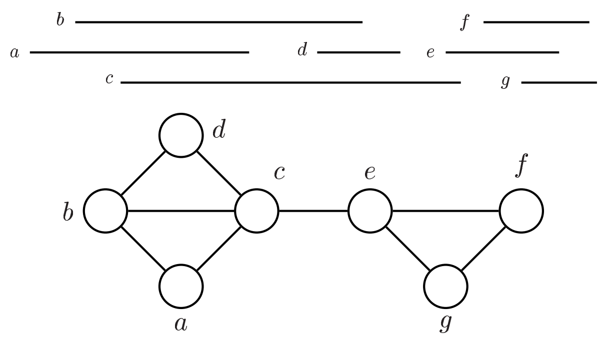
Remarks:

1. Let us rename the vertices in $\{v_1, v_9, v_{10}\} \to \{v_1, v_2, v_3\}$ and $\{v_2, v_4, v_5, v_6\}$ to $\{v_4, v_5, v_6, v_7\}$ and $\{v_3, v_7, v_8\}$ to $\{v_8, v_9, v_{10}\}$. If the new graph with the renaming as described is used as input to SEQUENTIAL VERTEX-COLORING, the procedure will produce a minimum coloring.

2. Remark 1 implies that if we use all $n!$ possible labellings of $G = (V, E)$ where $|V| = n$, the procedure will output a minimum coloring (for one of the input data), but of course it is extremely inefficient.

3. Note that SEQUENTIAL VERTEX-COLORING is a greedy algorithm since at each iteration of the **for** loop we assign the "smallest" color available.

4. SEQUENTIAL VERTEX-COLORING yields an optimal coloring of interval graphs. Recall that with the fast register allocation example, we can allocate two variables not used at the same time, to the same register. For each variable, we compute the first and last time where it is used. A variable is active during the interval between this times. We define a graph with the variables as vertices, in which two vertices are adjacent if they are active at a common time. Recall that the number of registers needed is the chromatic number of the graph. Given a family of intervals, define a graph whose vertices are the intervals, and such that two vertices are adjacent when the intervals intersect. A graph formed in this way is an interval graph, and the family of intervals is an interval representation of the graph.



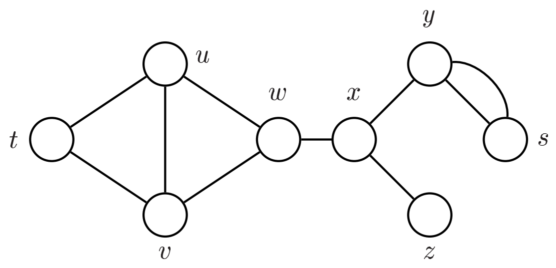
If the vertices are renamed $a \to v_1$, $b \to v_2$, $c \to v_3$, $d \to v_4$, $e \to v_5$, $f \to v_6$, $g \to v_7$ then SEQUENTIAL VERTEX-COLORING will yield an optimal coloring of 1, 2, 3, 1, 1, 2, 3 to v_1, v_2, $\dots v_7$ respectively.

The maximum degree of a vertex in a graph G is denoted by $\delta_{max}(G)$ (or $\Delta(G)$).

Lemma 6: *For any graph,* $\chi(G) \leq \delta_{max}(G) + 1$
Proof: Just apply SEQUENTIAL VERTEX-COLORING to the given graph, regardless of the labelling used. The procedure will not use more than $\delta_{max}(G) + 1$ colors since every vertex does not have more than $\delta_{max}(G)$ adjacent vertices □

We can tighten up the upper bound to only $\delta_{max}(G)$, according to Brook's theorem. We need to define a few terms first.

A component of G is a maximal connected subgraph of G. In other words, a connected subgraph H is a component of a graph G if H is not a proper subgraph of any connected subgraph G. The only component of a connected graph is the entire graph. One can think of the components of a non-connected graph G as the "pieces" that make up G. If the vertex-deletion subgraph H, obtained by removing vertex x from V, is such that H has more components than G, then x is called a cut-vertex (or cut point). A block of a loopless graph is a maximal connected subgraph H such that no vertex of H is a cut-vertex of H.
EXAMPLE: The following graph has four blocks. Each block has the following set of vertices: $\{t, u, w, v\}$, $\{w, x\}$, $\{x, y, z\}$ and $\{y, s\}$



□

Note that a block H of a graph G has no cut-vertices (of H), but H may contain vertices that are cut-vertices of G. In the above example, w, x and y are cut-vertices of G.

Remarks:

1. Two different blocks of a graph can have at most one vertex in common.

2. $$\begin{array}{c} x \in G \text{ is a cut-vertex of } G \\ \Updownarrow \\ x \text{ is in two different blocks of } G \end{array}$$

Result 1: Let B_1 and B_2 be distinct blocks of a connected graph G. Let $y_1 \in B_1$ and $y_2 \in B_2$ such that neither y_1 nor y_2 is a cut-vertex of G. Then vertex y_1 is not adjacent to vertex y_2. For example u in $\{t, u, w, v\}$ and z in $\{x, y, z\}$ are not adjacent (see figure of previous example).

A leaf block of a graph G is a block that contains exactly one cut-vertex of G.

Result 2: Let G be a connected graph with at least one cut-vertex. Then G has at least two leaf blocks. A graph G is k-connected if G is connected and it takes the removal of at least k vertices to either disconnect G or reduce G to a 1-vertex graph.

Lemma 7 *G is a non-complete, k-regular 2-connected graph with $k \geq 3$. Then G has a vertex x with two non-adjacent vertices y and z such that $G' = (V', E')$ with $V' = V \backslash \{y, z\}$ is a connected graph.*

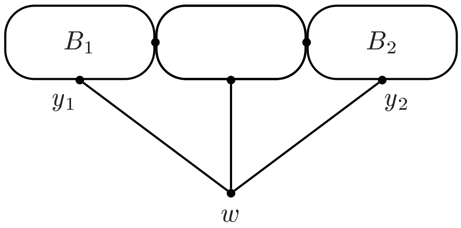
Proof: Let w be any vertex in G. $G \backslash w$ is either 2-connected or $G \backslash w$ is 1-connected.

Case 1: $G \backslash w$ is 2-connected

Let z be any vertex at distance 2 from w (which exists since G is regular and non-complete). If x is the vertex between w and z and y is taken to be w, we are done. In other words, G has 2 vertices, y and z that are non-adjacent to one another, but both adjacent to x such that $G \backslash \{y, z\}$ is connected

Case 2: $G \backslash w$ is 1-connected

By result 2, we know that we can find 2 leaf blocks, B_1 and B_2, of $G \backslash w$. Since G has no cut-vertices, it follows that w is adjacent to some vertex $y_1 \in B_1$ that is not a cut-vertex of $G \backslash w$, and likewise w is adjacent to some vertex $y_2 \in B_2$ that is not a cut-vertex of $G \backslash w$ (Note: $G \backslash \{w, y_1, y_2\}$ is connected since blocks have no cut-vertices).



By result 1, y_1 is not adjacent to y_2. If we let $w = x$, $y_1 = y$ and $y_2 = z$, we are done □

Brook's Theorem: *If G is a non-complete, simple connected graph with* $\delta_{max} \geq 3$ *then* $\chi(G) \leq \delta_{max}(G)$.
Proof: The proof makes use of SEQUENTIAL VERTEX-COLORING, and is divided into 3 cases. Suppose $|V| = n$.

Case 1: G is not regular. Choose $v_n \in V$ such that its degree is less than $\delta_{max}(G)$. Since G is connected, we can grow a spanning tree of G from v_n, assigning indices in decreasing order as we reach vertices. Each vertex other than v_n in the resulting ordering $v_1, v_2, \ldots, v_n$ has a higher-indexed adjacent vertex along the path to v_n in the tree. Hence, each vertex has at most $\delta_{max}(G) - 1$ lower-indexed adjacent vertices. Therefore, SEQUENTIAL VERTEX-COLORING uses at most $\delta_{max}(G)$ colors.

In the remaining two cases, G is regular.

Case 2: G is regular and has a cut-vertex x
Let G' be a component of the graph $G\backslash x$ together with its edges to x. The degree of x in G' is less than $\delta_{max}(G)$. Thus, we can use the procedure of case 1 to color the vertices of G' with at most $\delta_{max}(G)$ colors. The same can be done for all other components. By permuting the names of colors in each subgraph so that vertex x always gets color c_1 (or 1) we can construct a coloring of the vertices of G with $\delta_{max}(G)$ colors.

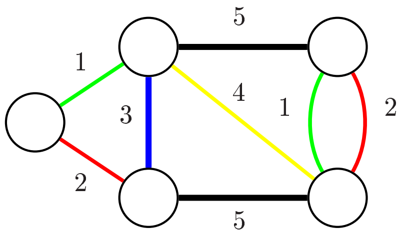
Case 3: G is regular and 2-connected by lemma 7, G has a vertex, which we call v_n with two non-adjacent vertices, which we denote by v_1 and v_2 such that $G\backslash\{v_1, v_2\}$ is connected.
Grow a spanning tree from v_n in $G\backslash\{v_1, v_2\}$ using numbers for colors $(3, \ldots, n)$ such that labels increase along paths to the root v_n. As in case 1, each vertex except v_n has at most $\delta_{max}(G) - 1$ lower-indexed adjacent vertices. So SEQUENTIAL VERTEX-COLORING uses at most $\delta_{max}(G)$ colors on all vertices except v_n. Moreover, the sequential coloring algorithm assigns the same color to v_1 and v_2 and at most $k-2$ colors to the other $k-2$ adjacent vertices of v_n. So, one of the $\delta_{max}(G)$ colors is available for coloring v_n. □

6.3.2 Edge Coloring

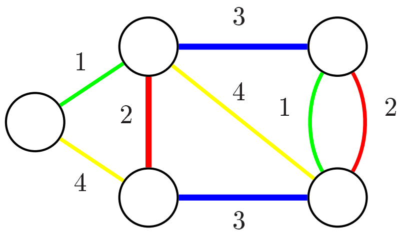
An edge-coloring of a graph $G = (V, E)$ is a mapping $f : E \to C$ where C is the set of colors usually denoted by $\{1, 2, \ldots, n\}$.

Two different edges are adjacent edges if they have at least one endpoint in common. G is said to be edge k-colorable if it has an edge-coloring using exactly k colors and such that adjacent edges are assigned different colors. The edge-chromatic number or chromatic index denoted by $\chi'(G)$ is the minimum k such

that G is edge k-colorable. Whereas multi-edges have no bearing on the vertex-colorings of a graph, they have an effect on edge colorings and should thus be taken into consideration. But graphs with self-loops will not be studied here.
EXAMPLE: The following is an example of an edge-coloring of a graph.



But we can do better



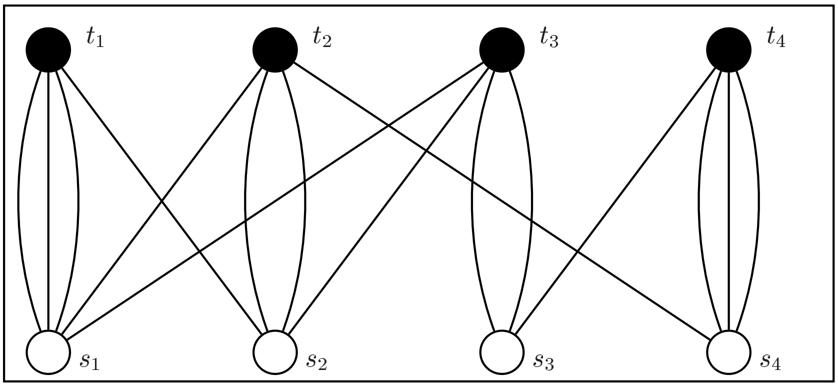
The graph is not edge 3-colorable since it contains 4 mutually adjacent edges. Therefore $\chi'(G) = 4$ □

Remark:

$$\text{a graph } G \text{ has } \chi'(G) = 1$$
$$\Updownarrow$$
$$\delta_{max}(G) = 1$$

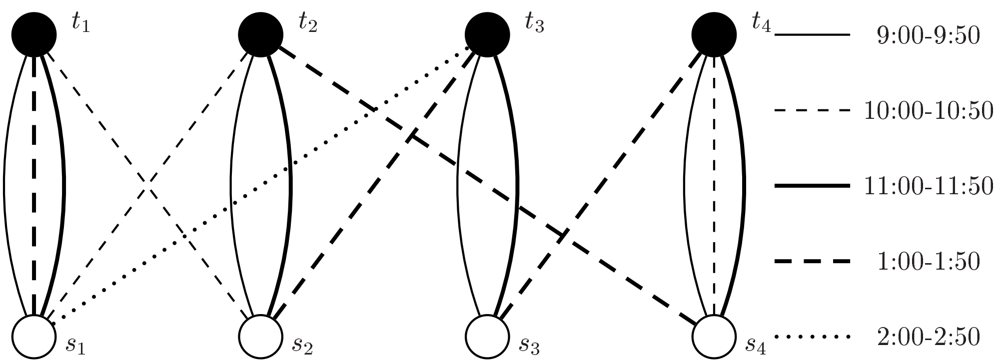
Modelling applications as edge-coloring problems

Example 1: Scheduling classes in High Schools.
$\{t_1, \ldots, t_m\}$ set of teachers
$\{s_1, \ldots, l_n\}$ set of courses.
Teacher t_j must teach $s_{j,k}$ sections of s_k.
Problem: Find a schedule with the minimum number of time periods such that no two sections of the same course are taught at the same time.
Solution: Construct a bipartite graph so that there are $s_{j,k}$ edges joining t_j and s_k for all j and k.



Representing a scheduling problem by a bipartite graph

Each edge-color represents a time slot in the schedule. The chromatic index represents the smallest number of time periods, which is equal to five in this example.



A minimum edge-coloring for the above graph

Example 2: Circuit boards.
The electronic devices $x_1, x_2, \ldots, x_n$ are on a board. The connecting wires emerging from each device must be colored differently, so that they can be distinguished. The least number of colors required is the chromatic index of the associated network. Since the edges incident to one vertex need different colors, we have

Result 1: For any graph,

$$\chi'(G) \geq \delta_{max}(G)$$

Result 2: If G has k mutually adjacent edges, then $\chi'(G) \geq k$.

Result 3: If H is a subgraph of G, then $\chi'(G) \geq \chi'(H)$.

Chromatic index for common graph families

Complete graph K_n, n even	$\chi'(G) = n - 1$
Complete graph K_n, n odd	$\chi'(G) = n$
Bipartite Graph G	$\chi'(G) = \delta_{max}(G)$
Path graph, P_n, $n \geq 3$	$\chi'(G) = 2$
Tree T	$\chi'(T) = \delta_{max}(T)$
Cube graph Q_n	$\chi'(Q_n) = n$
Cycle graph C_n, n even	$\chi'(C_n) = 2$
Cycle graph C_n, n odd	$\chi'(C_n) = 3$
Wheel graph W_n, $n > 4$	$\chi'(W_n) = n - 1$

Sequential edge-coloring algorithm

SEQUENTIAL EDGE-COLORING is analogous to sequential vertex-coloring and assumes that the input is the edge list: $e_1, e_2, \ldots, e_m$ where $e_i \in E$.

SEQUENTIAL EDGE-COLORING

---

begin
  $m \leftarrow |E|$
  for $i \leftarrow$ **to** m
    Let $f(e_i) \leftarrow$ smallest color number not used
    on any of the smaller-subscripted adjacent edges of e_i
  return edge-coloring f
end

As in the case with vertex-coloring, edge-coloring is an NP-hard problem. SEQUENTIAL EDGE-COLORING is a greedy procedure that does not necessarily deliver the chromatic index.

Vizing Theorem: *Let G be a simple graph; then either $\chi'(G) = \delta_{max}(G)$ or $\delta_{max}(G) = \delta_{max}(G) + 1$.* □

Multigraphs

The multiplicity $\mu(G)$ of a graph G is the maximum number of edges joining two vertices of G.

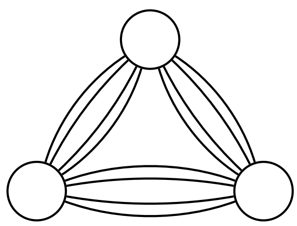
Result 4: For any loopless graph G, we have:

$$\delta_{max}(G) \leq \chi'(G) \leq \delta_{max}(G) + \mu(G)$$

Result 5: For any loopless graph G

$$\chi'(G) \leq \left\lfloor \frac{3\delta_{max}(G)}{2} \right\rfloor.$$

This result, due to Claude Shannon, is in terms of $\delta_{max}(G)$ only.
EXAMPLE:



In the multigraph, the edges are pairwise intersecting, and must receive distinct colors. In this example,

$$\chi'(G) = \frac{3\delta_{max}(G)}{2} = \delta_{max}(G) + \mu(G) = 12$$

□

Bibliography

[BB88] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

[CCS92] D. Casey, C. Cantor, and S. Spengler. Primer on Molecular Genetics. Technical report, U.S. Department of Energy, 1992.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W.H. Freeman and Company, 1979.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[Liu68] C.L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, 1968.

[Liu85] C.L. Liu. *Elements of Discrete Mathematics.* McGraw-Hill, 1985.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[Ros95] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill, 1995.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[Sti87] D.R. Stinson. *An Introduction to the Design and Analysis of Algorithms.* Charles Babbage Research Centre, Manitoba, Canada, 1987.