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: $W(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i \le M$.

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.

$$P(n,M) = \max\{P(n-1,M), p_n + P(n-1, M - w_n)\} \text{ if } w_n \le M$$

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

$$\begin{aligned} P(j,m) &= P(j-1,m) \text{ if } w_j > m \\ P(j,m) &= \max\{P(j-1,m), p_j + P(j-1, m - w_j)\} \text{ if } w_j \le m \end{aligned}$$

with

$$\begin{aligned} P(1,m) &= p_1 \text{ if } w_1 \le m \\ P(1,m) &= 0 \text{ if } w_1 > m \end{aligned}$$

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	01	6	7	7	10	11	16	17	17	18	19	19	22	23	28	29	29	29	31	32
5	01	6	7	7	10	11	16	17	17	18	19	19	22	23	28	29	29	29	31	32
4	01	6	7	7	10	11	16	17	17	18	19	19	22	23	28	29	29	29	29	29
3	01	6	7	7	10	11	16	17	17	17	17	17	17	17	17	17	17	17	17	17
2	01	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
1	01	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	01	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.

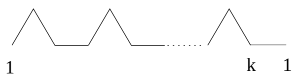
□

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)

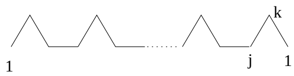
If k is the last vertex in HC before returning to vertex 1, then:

$$\textit{opt } HC = \text{ optimum path from 1 to } k \text{ passing through all vertices } + w(k,1) :$$



Similarly:

$$\textit{opt } HC = \text{ optimum path from 1 to } j + w(j,k) + w(k,1), \ j \in \{2, \ldots, n\} \backslash \{k\}$$



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 $S \backslash \{k\}$.

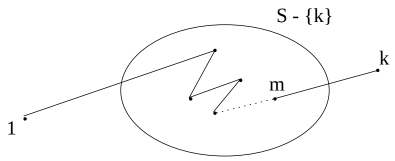
The cost of optimum HC is:

$$\min\{P(\{2, \ldots, n\}, k) + w(k,1) \text{ for } 2 \leq k \leq n\}$$

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.

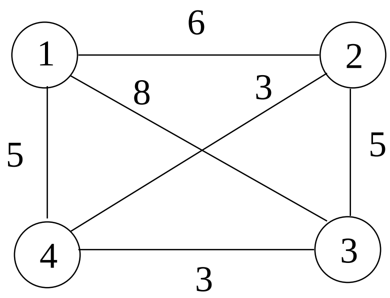


Figure 4.1:

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 $k = 2$		

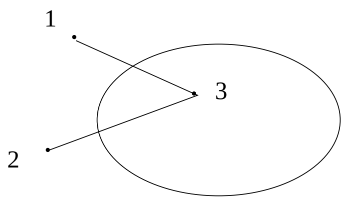


Figure 4.2:

The whole bottom-up process is shown in the diagram given by Figure 4.4.

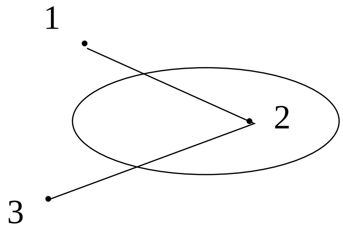


Figure 4.3:

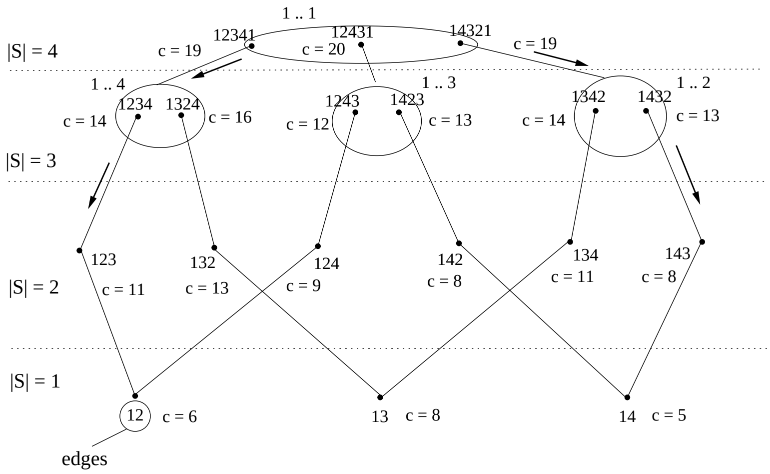


Figure 4.4:

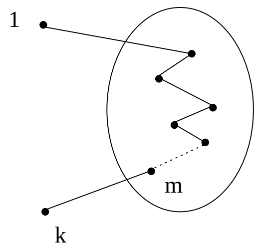
Remarks on the diagram

See Figure 4.4.

1 :

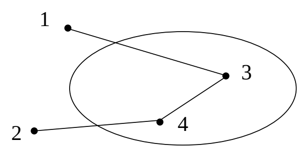
At the bottom level: $S = \{2\}, S = \{3\}, S = \{4\}$
At the first level: $S = \{2, 3\}, S = \{2, 4\}, S = \{3, 4\}$
At the last level: $S = \{2, 3, 4\}$

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:

$$(1 + x)^{n-1} = \sum_{j=0}^{n-1} \binom{n-1}{j} x^j$$

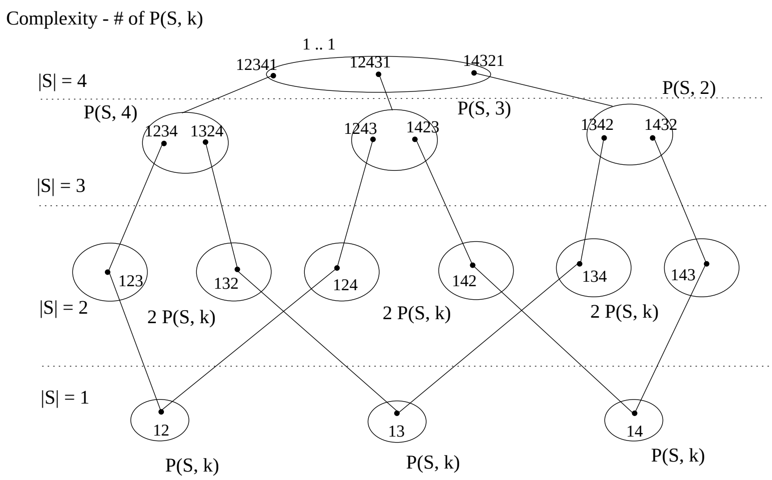
By taking the derivative on both sides we obtain:

$$(n-1)(1+x)^{n-2} = \sum_{j=0}^{n-1} j\binom{n-1}{j} x^{j-1} = \sum_{j=1}^{n-1} j\binom{n-1}{j} x^{j-1}$$

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)

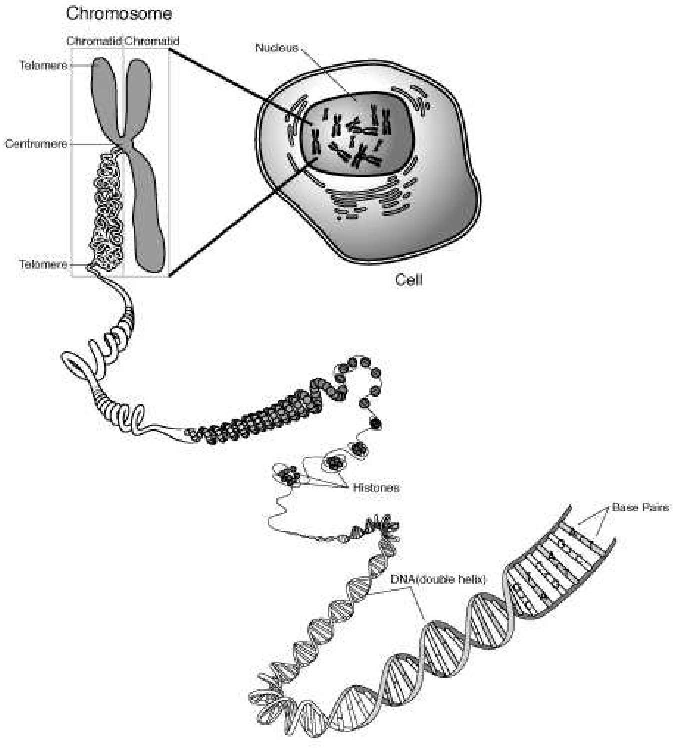


Figure 4.5: The human genome

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

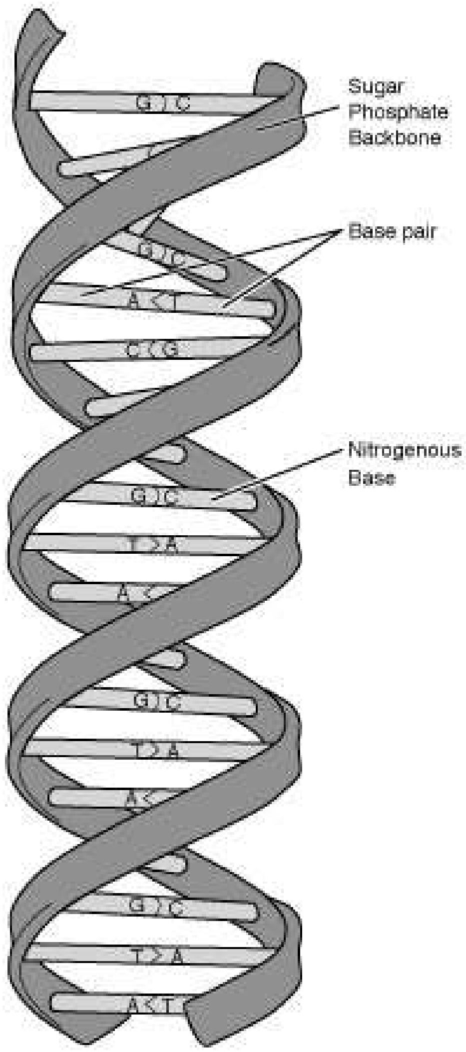


Figure 4.6: The DNA structure

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.

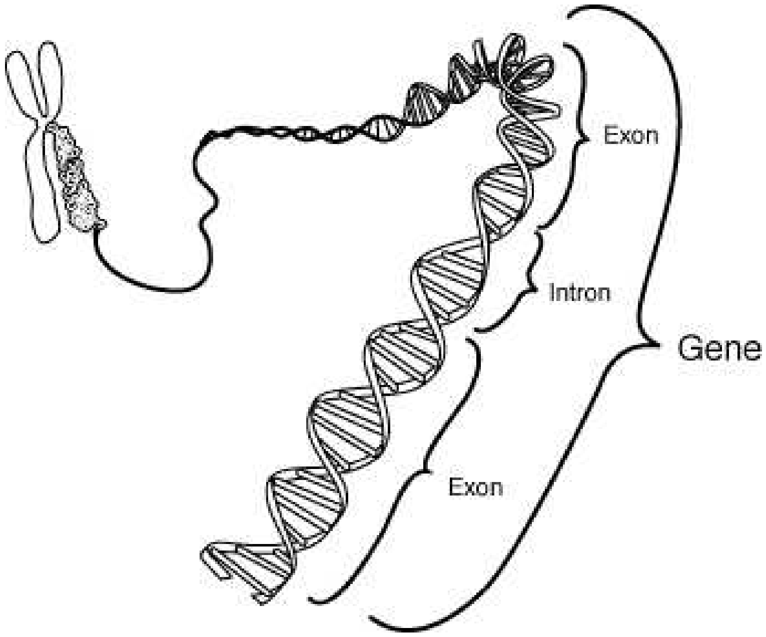


Figure 4.7: The gene of a DNA molecule

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.

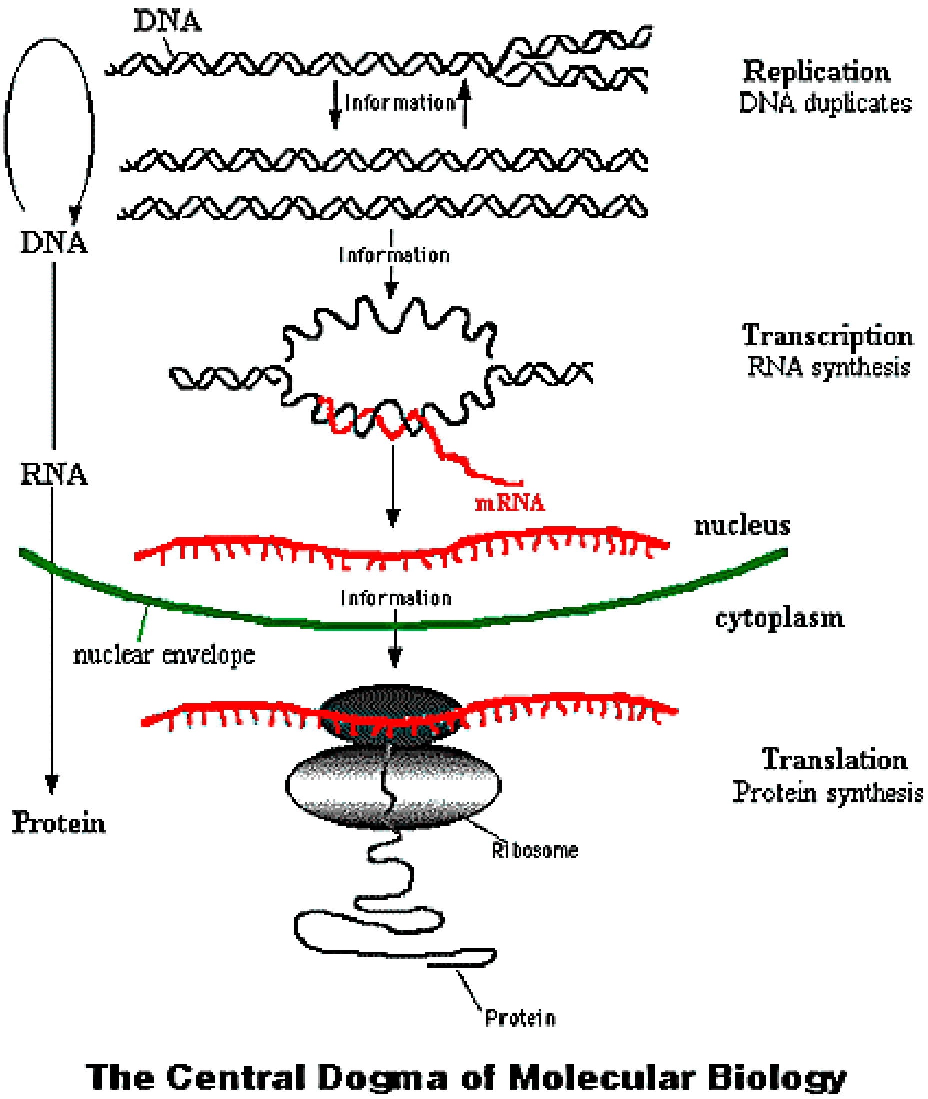


Figure 4.8:

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 $A(i,j)$, $0 \leq i \leq n$ and $1 \leq j \leq m$ 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:

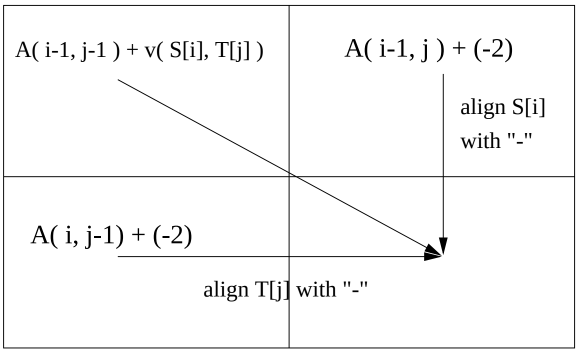
$$v(S[i],T[j]) = \begin{cases} +1 \text{ if } & S[i] = T[j] \\ -1 \text{ if } & S[i] \neq T[j] \end{cases}$$

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 $A(n,m) = \max_{0 \le k \le m}[A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m-k)]$.

Proof:

a) *Show that* $\max_{0 \le k \le m}[A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m-k)] \le A(n,m)$.
For any fixed position k' in T, we can find an alignment of S and T consisting of

$$S_1 S_2 \ldots S_{\frac{n}{2}} \text{ and } T_1 T_2 \ldots T_{k'}$$

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

$$S_{\frac{n}{2}+1} \ldots S_n \text{ and } T_{k'+1} \ldots T_m.$$

		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

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

$$S_{\frac{n}{2}} S_{\frac{n}{2}+1} \ldots S_n \text{ and } T_{k'+1} T_{k'+2} \ldots T_m.$$

		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	$\boxed{-4}$	-3	0	3	$\boxed{1}$	-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

```
a  c  d  b  c
-  -  -  b  c
-2 -2 -2 +1 +1 = -4
```

```
a  c  d  b  c  -
b  c  d  b  c  b
-1 +1 +1 +1 +1 -2 = +1
```

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

$$A(\frac{n}{2}, k') + A^R(\frac{n}{2}, m - k') \le \max_{0 \le k \le m} (A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m - k))$$

Therefore

$$A(n, m) \le \max_{0 \le k \le m} (A(\frac{n}{2}, k) + A^R(\frac{n}{2}, m - k))$$

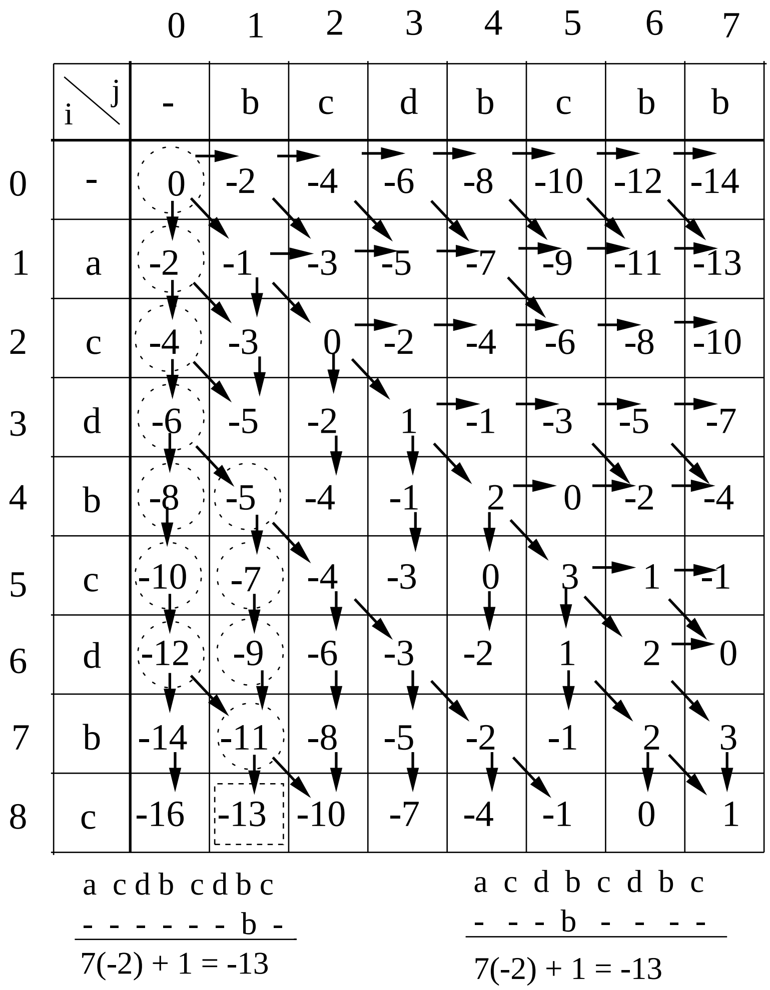


Figure 4.12: Two optimal solutions for $S = acdbcdbc$ and $T = b$

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 $S_1 S_2 \ldots S_{\frac{n}{2}-1}$ and $T_1 T_2 \ldots T_{k_1}$ and
- one for strings $S_{\frac{n}{2}+1} S_{\frac{n}{2}+2} \ldots S_n$ and $T_{k_2} T_{k_2+1} \ldots T_m$.

Aligning $S_1 \ldots S_{\frac{n}{2}-1}$ and $T_1 \ldots T_{k_1}$ is called the top problem (see section X of Figure 4.13). Aligning $S_{\frac{n}{2}+1} \ldots S_n$ and $T_{k_2} \ldots T_m$ is called the bottom problem (see section Y of Figure 4.13), the rest of the matrix can be ignored.

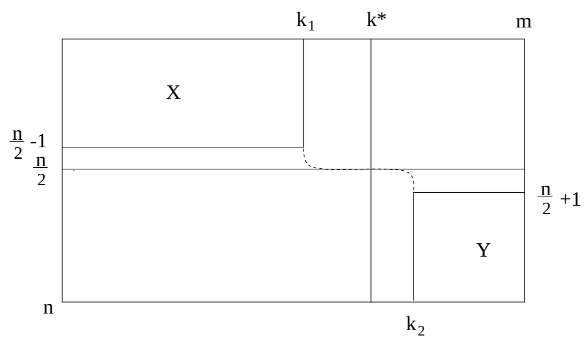


Figure 4.13:

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 $2cnm$. □

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

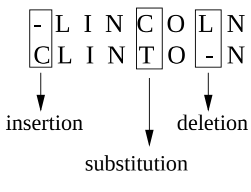
character deletion and

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

A L - T R U I S T I C
A L G O R - I - T H M

or

A L T R U I S T I C
A L G O R I - T H M

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]
```

$$\textbf{then } E(i,j) = min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1) \end{cases}$$

$$\textbf{else } E(i,j) = min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+1 \end{cases}$$

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.

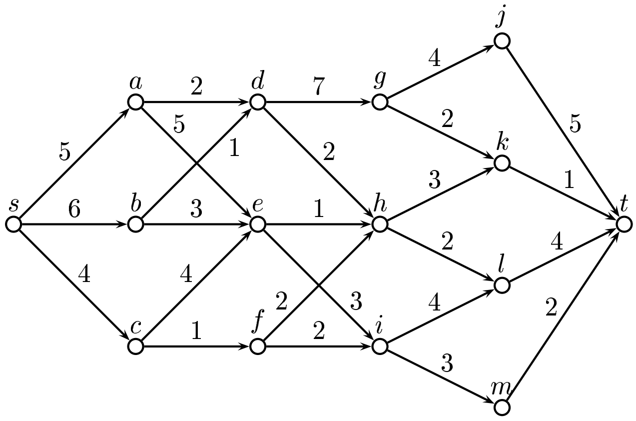
Problem instance:
A weighted graph (positive weights) with source vertex s and sink vertex t.

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)

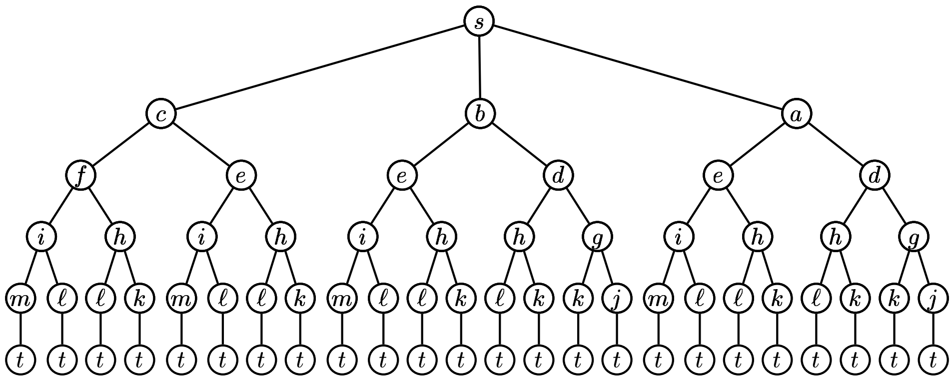
node	a	b	c
next node	d	d	f
total cost	8	7	7

table(4)

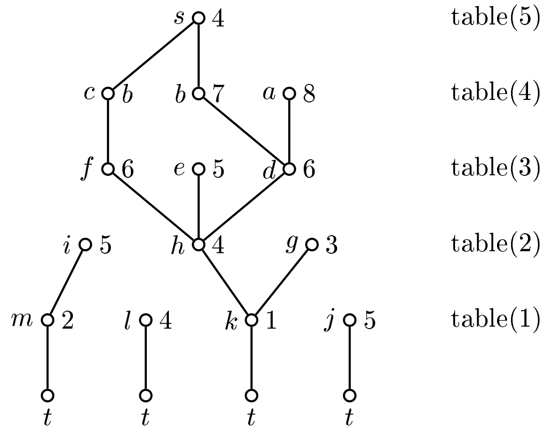
node	s
next node	c
total cost	11

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

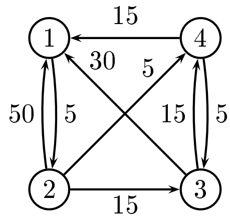
Given a weighted graph, we want to compute the shortest path between each pair of nodes. The weights of the edges are nonnegative. The vertices are numbered from 1 to n, and matrix L gives the length (weight) of each edge, with $L(i, j) = \infty$ if edge (i, j) does not exist, $L(i, i) = 0$ and $L(i, j) = t$ 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

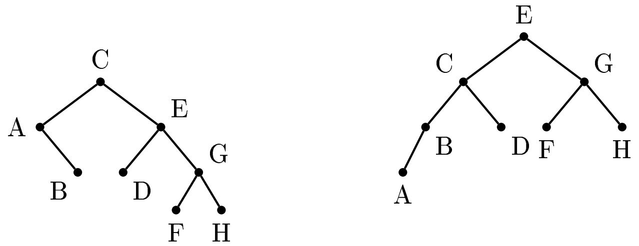
$$P = \begin{bmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

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:

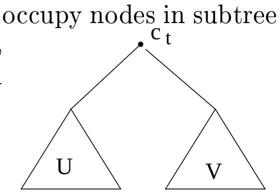


Figure 4.15:

$$C^* = \sum_{k=i}^{j} p_k(d_k^* + 1)$$

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 $c_i, c_{i+1}, c_{i+2} \dots, c_j$ 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 $c_i, c_{i+1}, c_{i+2} \dots, c_{k-1}$. R is an optimal subtree containing $c_{k+1}, c_{k+2}, \dots, c_j$. When we look for a key in the main tree, the probability that it is in the sequence $c_i, c_{i+1}, \dots, c_j$ 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

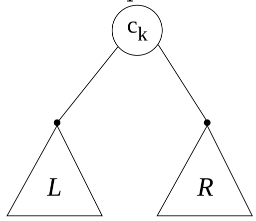


Figure 4.16:

$$c_{ij}^k = m_{ij} + c_{i\,k-1} + c_{k+1\,j}$$

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

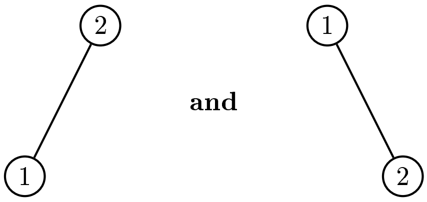
$$c_{ij} = m_{ij} + \min_k \{c_{i\,k-1} + c_{k+1\,j}\}$$

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



Continue in a diagonal fashion:

$$c_{23} = 0.18 \qquad c_{34} = 0.61 \qquad c_{45} = 0.69$$

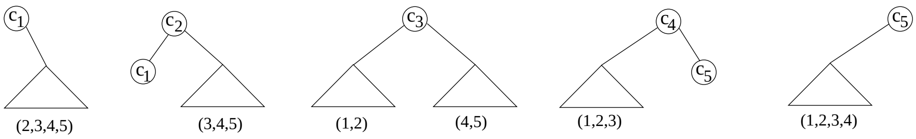
then

$$\begin{aligned} c_{13} &= m_{13} + \min\{c_{10} + c_{23}, c_{11} + c_{33}, c_{12} + c_{43}\} \\ &= 0.43 + \min\{0.18, 0.38, 0.4\} = 0.61 \\ c_{24} &= m_{24} + \min\{c_{21} + c_{34}, c_{22} + c_{44}, c_{23} + c_{54}\} \\ &= 0.58 + \min\{0.61, 0.5, 0.18\} = 0.76 \\ c_{35} &= m_{35} + \min\{c_{32} + c_{45}, c_{33} + c_{55}, c_{34} + c_{65}\} \\ &= 0.65 + \min\{0.69, 0.2, 0.61\} = 0.85 \\ c_{14} &= m_{14} + \min\{c_{10} + c_{24}, c_{11} + c_{34}, c_{12} + c_{44}, c_{13} + c_{54}\} \\ &= 0.88 + \min\{0.76, 0.91, 0.85, 0.61\} = 1.49 \\ c_{25} &= m_{25} + \min\{c_{21} + c_{35}, c_{22} + c_{45}, c_{23} + c_{55}, c_{24} + c_{65}\} \\ &= 0.7 + \min\{0.85, 0.74, 0.3, 0.76\} = 1.00 \end{aligned}$$

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



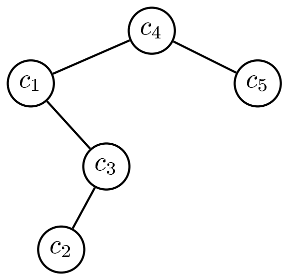
Finally we get as a resulting matrix:

	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 $\Theta(n^3)$.

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.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[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.