Technische Universität München
Institut für Informatik
Lehrstuhl für Effiziente Algorithmen
Prof. Dr. Angelika Steger

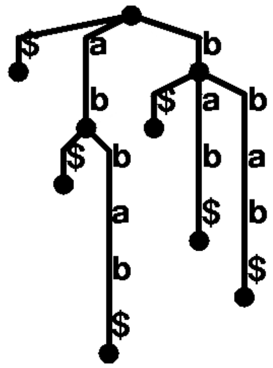
Wintersemester 2001/02
Semestralklausur
7. Februar 2002

Effiziente Algorithmen und Datenstrukturen I

Aufgabe 1 (4 Punkte)

Zeichnen Sie einen Suffix-Tree für die Zeichenkette *abbab*\$

Lösungsvorschlag



Aufgabe 2 (6 Punkte)

Betrachten Sie die folgenden Probleme:

Eingabe: Eine positive ganze Zahl x
Ausgabe: a) $\lfloor\sqrt{x}\rfloor$
b) 2^x
c) $x!$

Sind diese Probleme in polynomieller Zeit lösbar? Bezüglich welchen Kostenmaßes ? Begründen Sie Ihre Antworten.

Lösungsvorschlag

Zunächst einmal ist die Eingabelänge $n = \log x$, da man $\log x$ Bits braucht, um x darzustellen.
zu a)
Bezüglich des Einheitskostenmaßes hat die Berechnung von $\sqrt{x}$ Laufzeitkosten von $O(n)$, wenn man die Wurzel mit Hilfe einer binären Suche findet. Man benötigt in diesem Fall $\log x$ Multiplikationen, die im Einheitskostenmaß $O(1)$ Kosten haben.
Wenn man im logarithmischen Kostenmaß rechnet, dann braucht man für jede Multiplikation $O((\log x)^2)$ Kosten (evtl auch weniger wenn man den Algorithmus von Strassen verwendet.) D.h. man hat eine gesamt Laufzeit von $O(n^3)$, was immer noch polynomiell ist.
zu b)

Hier hängt es davon ab wie geschickt man sich anstellt: Angenommen wir nehmen die bitweise Darstellung von x, d.h.

$$x = x_0 \cdot 2^0 + x_1 \cdot 2^1 + \ldots + x_n \cdot 2^n = x_0 + (x_1 + (x_2 + \ldots + (x_{n-1} + x_n \cdot 2) \cdot 2 \ldots) \cdot 2) \cdot 2$$

Dann kann man 2^x so berechnen:

$$2^x = 2^{x_0} \cdot \left(2^{x_1} \cdot \left(2^{x_2} \cdot \ldots \left(2^{x_{n-1}} \cdot \left(2^{x_n}\right)^2\right)^2 \ldots\right)^2\right)^2$$

D.h. man braucht $\log x$ Quadrierungen und $\log x$ Multiplikationen. Im Einheitskostenmaß kann man also wieder eine lineare Laufzeit $O(n)$ erreichen.
Im logarithmischen Kostenmaß kann man anders argumentieren; um 2^x darstellen zu können braucht man mindestens x Bits. Da jedes Bit mit Laufzeit $\Omega(1)$ geschrieben werden muss, hat man auf jeden Fall eine gesamt Laufzeit von $\Omega(x)$.
Die Eingabelänge ist nur $n = \log x$, also erhält man eine exponentielle Laufzeit von $\Omega(2^n)$ im logarithmischen Kostenmaß!
zu c)
In diesem Fall benötigt man $x-1$ Multiplikationen. Da die Eingabelänge auch im Einheitskostenmaß nur $\log x$ beträgt, erhält man auch im Einheitskostenmaß einen exponentiellen Algorithmus mit $O(2^n)$ Laufzeit.
Im logarithmischen Kostenmaß schaut man einfach wieder auf die Bits, die geschrieben werden müssen. Es ist bekannt, dass $log(x!) = \Theta(x \log x)$. Also braucht man $x \log x$ Bits zur Darstellung und damit hat man eine Laufzeit von mindestens $\Omega(n \cdot 2^n)$.

Aufgabe 3 (5 Punkte)

Gegeben Sei folgende Methode in Java

```
static public void increment(int a[])
{
  int i;
  i=0;
  while (i<a.length && a[i] == 1)
  {
    a[i]=0;
    i++;
  }
  if (i<a.length) a[i]=1;
}
```

Das Array $a[]$ enthält bei Aufruf der Methode immer nur Nullen und Einsen.
Beschreiben Sie die genaue Funktionalität der Methode.
Sei $n = a.length$. Die Worst-Case Laufzeit der Methode beträgt sicher $O(n)$. Definieren Sie eine Potenzialfunktion und führen Sie eine amortisierte Analyse durch, um zu zeigen, dass die amortisierte Laufzeit der Methode nur $O(1)$ ist.

Lösungsvorschlag

Die Methode inkrementiert eine Zahl mit n Bits, die durch $a[0], \ldots, a[n-1]$ dargestellt ist, um eins.

Wenn bei Aufruf der Methode alle Bits auf 1 gesetzt sind, dann setzt die Methode alle Bits wieder auf 0.
Wir definieren die Potenzialfunktion, als die Anzahl der Einsen, die $a[]$ enthält.
Beispiel: $a[] := \{1,1,1,0,1,0,1\}$ hat Potenzial 5.
Wenn die Schleife einmal durchlaufen wird, dann wird eine Eins durch eine Null ersetzt. Im Letzten Schritt wird eine Null zu einer Eins gemacht, falls nicht alle Bits 1 waren. D.h. bei k Schleifendurchläufen hat man tatsächliche Kosten von $k+1$ und die Anzahl der Einsen hat um mindestens $k-1$ abgenommen. Also ergibt sich:

$$\begin{aligned} \text{amort. Kosten} &= \text{tats. Kosten} + \text{Potenzialdifferenz} \\ \text{amort. Kosten} &\leq k+1-(k-1) = 2 \end{aligned}$$

D.h. die amortisierte Laufzeit beträgt $O(1)$.

Aufgabe 4 (6 Punkte)

Gegeben sei ein Graph $G(V,E)$. Jede Kante habe ein positives Gewicht.
Angenommen das Gewicht eines Pfades ist nicht die Summe der Gewichte der benutzten Kanten, sondern nur das Maximum der Gewichte der benutzten Kanten.
Beschreiben Sie einen modifizierten Dijkstra Algorithmus, der unter diesen Bedingungen den kürzesten Pfad findet. (Benutzen Sie kommentierten Pseudo-Code zur Beschreibung)
Erklären Sie warum der Dijkstra Algorithmus unter diesen Bedingungen funktioniert.

Lösungsvorschlag

Damit Dijkstra in diesem Fall funktioniert, muss man lediglich die Art und Weise, wie Pfadlängen berechnet werden und wie Pfadlängen aktualisiert angepasst werden.
Dazu implementiert man Dijkstra wie folgt (im Pseudo-Code). Gegeben sei ein Graph $G = (V,E)$ mit Gewichtsfunktion $l : E \to R^+$ und der Startknoten s und Zielknoten z:

$\forall\, v \in V$ do $pqElement[v] := nil$	Noch kein Knoten in der PQ
$dist[s] := 0$	
$pqElement[s] := PQInsert(s, 0)$	Füge s mit Entfernung 0 in PQ ein
do	
$v := PQExtractMin()$	Hole nächsten Knoten aus der PQ
$\forall\, u$: u is neighbour of v do	Besuche alle Nachbarn von v
$d := \max\{\, dist[v], l(\{v,u\})\,\}$	Distanz eines neuen Pfades zu u
if $(pqElement[u] = nil)$ do	Wenn u noch nicht in der PQ ist
$dist[u] := d$	Dann füge u in die PQ ein
$pqElement[u] := PQInsert(u, d)$	
else if $(dist[u] > d)$	Sonst: Wenn neuer Weg kürzer
$dist[u] := d$	Erniedrige Distanz in PQ
$PQDecreaseKey(pqElement[u], d)$	
while $(v! = z)$	Widerhole bis Zielknoten gefunden

Der Algorithmus funktioniert, weil die entscheidende Voraussetzung gegeben ist: Ein Pfad wird durch Verlängerung um eine Kante nie kürzer. (Genau diese Eigenschaft gilt bei negativen Kantengewichten *nicht*.)

Genauer: Sobald ein Knoten durch $ExtractMin$ aus der Priority-Queue geholt wird, gibt es nur noch Knoten in der Priority-Queue die weiter weg sind. Da zusätzliche Kanten die Entfernung nicht verkürzen können, ist jeder Pfad der über einen anderen Knoten der Priority-Queue läuft länger, d.h. es kann keinen kürzeren Pfad für den Knoten geben, der durch $ExtractMin$ aus der Priority-Queue geholt wurde.

Aufgabe 5 (7 Punkte)

Angenommen Sie haben eine boolesche Formel in konjunktiver Normalform gegeben, in der jede Klausel genau drei Literale von verschiedenen Variablen enthält. Angenommen es gibt m Klauseln und jeder Klausel ist ein positives Gewicht $w_i, 1 \leq i \leq m$ zugewiesen. Das Gewicht einer Belegung sei durch die Summe der Gewichte der Klauseln definiert, die durch die Belegung erfüllt werden.

1. Was ist der Erwartungswert des Gewichtes einer zufälligen Belegung.
2. Beschreiben Sie einen deterministischen Algorithmus, um eine Belegung zu erhalten, deren Gewicht mindestens so groß ist wie der Erwartungswert. Begründen Sie, warum der Algorithmus funktioniert.

Lösungsvorschlag

Zu 1)

Sei $W = \sum w_i$. Eine Klausel wird von einer zufälligen Belegung mit Wahrscheinlichkeit $p = \frac{7}{8}$ erfüllt. Sei X_i die Zufallsvariable, die das Gewicht beschreibt, das eine Klausel zum Gesamtgewicht einer zufälligen Belegung beiträgt. Bezeichne X das Gesamtgewicht einer zufälligen Belegung. Der Erwartungswert $E[X_i]$ ist $\frac{7}{8}w_i$. Daraus folgt:

$$\begin{aligned}
E[X] &= E[\sum X_i] \\
E[X] &= \sum E[X_i] \quad \text{wegen Linearität des Erwartungswertes} \\
E[X] &= \sum \frac{7}{8} w_i \\
E[X] &= \frac{7}{8} W
\end{aligned}$$

Zu 2)

Sei k_i die Anzahl der Literale die in einer Klausel vorkommen. Allgemein ist $E[X_i] = \frac{2^{k_i}-1}{2^{k_i}} w_i$. D.h. wenn man eine Formel vorgegeben hat, dann kann den Erwartungswert einer zufälligen Belegung für diese Formel leicht ausrechnen.

Aus 1) wissen wir, dass es eine Belegung geben muss, die ein Gewicht von $\frac{7}{8}W$ hat.

Man fängt nun bei der ersten Variable v_1 an und stellt sich vor, dass diese Variable auf 0 gesetzt wird. Diese Belegung von v_1 führt dazu, dass ein Teil der Klauseln (in denen v_1 negiert vorkommt) erfüllt wird und ein anderer Teil (in denen die v_1 nicht negiert vorkommt) kürzer wird. Für die neu entstehende Formel kann man den Erwartungswert wie oben beschrieben berechnen. Addiert man diesen Wert zu dem Gewicht der bereits erfüllten Klauseln, dann erhält man den Erwartungswert einer zufälligen Belegung für die restlichen Variablen, *unter der Bedingung*, dass v_1 auf 0 gesetzt wird.

Genauso kann man den Erwartungswert unter der Bedingung, dass v_1 auf 1 gesetzt wird, berechnen.

Man wählt für v_1 den Wert, bei dem der Erwartungswert höher ist. Da gilt

$$E[X] = \frac{1}{2}E[X|v_1 = 0] + \frac{1}{2}E[X|v_1 = 1]$$

muss einer der beiden Erwartungswerte größer oder gleich $E[X]$ sein; man fährt sukzessive mit den restlichen Variablen fort, bis man alle Variablen belegt hat. Da man in jedem Schritt einen Erwartungswert von mindestens $\frac{7}{8}W$ hat, erhält man am Schluss auch eine Belegung die ein Gewicht von mindestens $\frac{7}{8}W$ hat.

Aufgabe 6 (4 Punkte)

Geben Sie einen Algorithmus an, der von zwei Texten mit jeweils n Länge feststellt, ob der zweite Text eine zyklische Rotation des ersten Textes ist. Ihr Algorithmus soll $O(n)$ Laufzeit haben. Begründen Sie warum Ihr Algorithmus lineare Laufzeit hat.
Beispiel: *otto* ist eine zyklische Rotation von *toot*
Illustrieren Sie Ihren Algorithmus an dem angegebenen Beispiel.
(**Hinweis:** geht mit KMP)

Lösungsvorschlag

Wir „verdoppeln“ den ersten Text, d.h. aus „otto“ wird „ottootto“. Dann lassen wir den KMP Algorithmus den zweiten Text im verdoppelten ersten Text suchen. Dazu benötigt KMP höchstens $O(2n+n) = O(3n) = O(n)$ Laufzeit. Wenn KMP ein Vorkommen findet, dann ist der zweite Text eine zyklische Rotation des ersten Textes.

Illustration:

o	t	t	o	o	t	t	o
t	o	o	t				
	t	o	o	t			
		t	o	o	t		

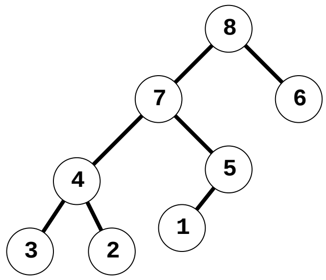
⇒ „toot“ ist eine zyklische Rotation von „otto“.

Aufgabe 7 (8 Punkte)

Ein kartesischer Baum ist ein binärer Baum, der auf einem Array $a_1, \ldots, a_n$ mit n paarweise verschiedenen Elementen definiert ist. Die Definition ist rekursiv:

- Die Wurzel ist $a_{max} : a_{max} = \max\{a_1, \ldots, a_n\}$.
- Der linke Teilbaum besteht aus dem kartesischen Baum für die Elemente $a_1, \ldots, a_{max-1}$.
  Der rechte Teilbaum besteht aus dem kartesischen Baum für die Elemente $a_{max+1}, \ldots, a_n$.

Beispiel für ein Array mit den Einträgen $3, 4, 2, 7, 1, 5, 8, 6$:



1. Beschreiben Sie einen Algorithmus der einen kartesischen Baum konstruiert. Illustrieren Sie obiges Beispiel, indem Sie die Bäume zeichnen, die beim Ablauf des Algorithmus entstehen.

   Hinweis: Fangen Sie mit a_1 an und erweitern Sie den Baum sukzessiv um $a_2, a_3, \ldots, a_n$.

2. Definieren Sie eine geeignete Potenzialfunktion und führen Sie damit eine amortisierte Analyse durch, um zu zeigen, dass die Laufzeit Ihres Algorithmus $O(n)$ beträgt.

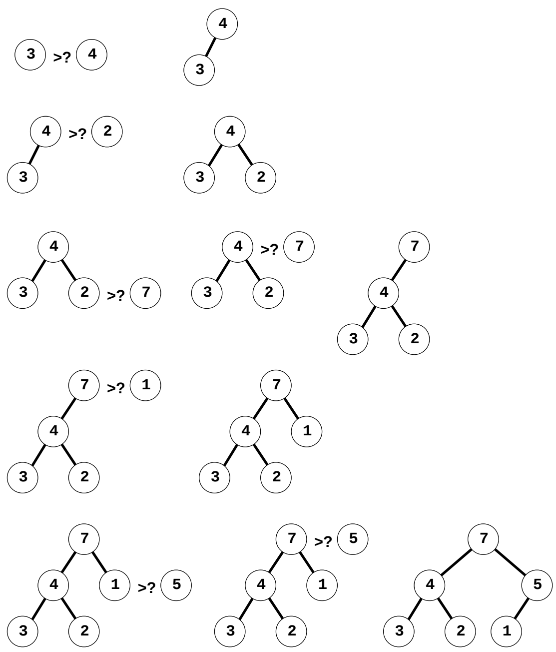
Lösungsvorschlag

zu 1)

Der Algorithmus funktioniert wie folgt:

- Man fängt bei a_1 an, d.h. man erzeugt den kartesischen Baum der nur aus dem Knoten a_1 besteht.
- In jedem Schritt merkt man sich den am weitesten rechts liegenden Knoten, d.h. den Knoten, den man von der Wurzel aus erreicht, wenn man immer das rechte Kind hinabsteigt, bis man einen Knoten erreicht, der kein rechtes Kind hat.
- Wenn jetzt ein neuer Knoten a_i hinzukommt, dann vergleicht man den Wert von a_i mit dem Wert des am weitesten rechts liegenden Knoten. Solange a_i einen größeren Wert hat, macht man mit dem jeweiligen Vater des verglichenen Knotens weiter, bis man einen Knoten erreicht hat, der größer als a_i ist, oder es keinen Vater mehr gibt.
- Der zuletzt verglichene Knoten wird durch a_i ersetzt und wird gleichzeitig linkes Kind von a_i. a_i selber wird der neue am weitesten rechts liegende Knoten.

Illustration:



usw.

zu 2)

Wir definieren die Potenzialfunktion, als die Entfernung des am weitesten rechts liegenden Knoten zur Wurzel. (Ist die Wurzel selber der am weitesten rechts liegende Knoten, hat man also Entfernung 0.)

Wenn wir k Vergleiche ausführen, um einen neuen Knoten einzufügen, dann haben wir k tatsächliche Kosten für die Vergleiche und 1 tatsächliche Kosten für das Umhängen der Knoten.
Bei k Vergleichen verkürzt sich die Entfernung des am weitesten rechts liegenden Knoten um genau $k-1$. (Da die $k-1$ kleineren Knoten wegfallen.) Also beträgt die Potenzialdifferenz $-(k-1)$. Es ergibt sich also:

$$\begin{aligned} \text{amort. Kosten} &= \text{tats. Kosten} + \text{Potenzialdifferenz} \\ \text{amort. Kosten} &= k+1-(k-1)=2 \end{aligned}$$

D.h. jeder neue Knoten erzeugt $O(1)$ amort. Kosten. Bei n Knoten haben wir also eine Laufzeit von $O(n)$.