Algorithmen und Datenstrukturen

Martin Hofmann

Institut für Informatik

LMU München

Sommersemester 2006

Literatur

- T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 2. Aufl. 2001. Seit 2004 existiert dt. Üb. bei Oldenbourg. Standardwerk und Vorlesungstext.
- Weitere Lehrbücher: [Aho, Hopcroft, Ullman], [Sedgewick], [Mehlhorn], [Ottmann-Widmayer]. Können zur Vertiefung hinzugezogen werden.

Was sind Algorithmen, wieso Algorithmen?

Ein Algorithmus ist eine genau festgelegte Berechnungsvorschrift, die zu einer Eingabe oder Problemstellung eine wohldefinierte Ausgabe oder Antwort berechnet.

Ein Algorithmus kann durch ein Programm in einer Programmiersprache angegeben werden, aber typischerweise lässt sich ein und derselbe Algorithmus auf verschiedene Arten und in verschiedenen Programmiersprachen implementieren.

Neben Programmiermethodiken (wie Modularisierung, Objektorientierung, etc.) bilden Algorithmen den Schlüssel zu erfolgreicher Softwareentwicklung. Durch Verwendung professioneller Algorithmen lassen sich oft spektakuläre Laufzeitgewinne realisieren.

Zwar stellen Sprachen wie Java viele Algorithmen fertig implementiert in Form von Bibliotheken bereit; viele Projekte werden jedoch nach wie vor in C implementiert, außerdem ist es oft der Fall, dass eine Bibliotheksfunktion nicht genau passt.

Die Algorithmik ist eine lebendige Disziplin. Neue Problemstellungen (Bioinformatik, Internet, Mobilität) erfordern zum Teil neuartige Algorithmen. Für alte Problemstellungen werden bessere Algorithmen entwickelt (kombinatorische Optimierung, Planung). Schnellere Hardware rückt bisher unlösbare Probleme in den Bereich des Machbaren (automatische Programmverif., Grafik, Simulation).

Was sind Datenstrukturen, wieso Datenstrukturen?

Methoden, Daten so anzuordnen, dass Zugriffe möglichst schnell erfolgen.

Konkurrierende Anforderungen: Platzbedarf, verschiedene Arten von Zugriffen.

Beispiel: Stapel (Keller): Ist die Größe von vornherein beschränkt, so bietet sich ein Array an. Anderenfalls eine verkettete Liste. Diese verbraucht aber mehr Platz und Zeit. Will man nicht nur auf das oberste Element zugreifen, so benötigt man eine doppelt verkettete Liste oder gar einen Baum, ...

Datenstrukturen und Algorithmen gehen Hand in Hand. Oft bildet eine geeignete Datenstruktur den Kern eines Algorithmus. Beispiel: Heapsort (nächste Woche).

Zusammenfassung

Kapitel I	Einführung: Sortieren durch Einfügen, Sortieren durch Mischen, Laufzeitabschätzungen, Lösen von Rekurrenzen.
Kapitel II	Sortieren und Suchen: Heapsort, Quicksort, Median.
Kapitel III	Professionelle Datenstrukturen: Balancierte Binärbäume, Hashtabellen, *union find*, Prioritätsschlangen.
Kapitel IV	Allgemeine Entwurfsmethoden: Greedy, Backtracking, Branch & Bound, dynamische Programmierung.
Kapitel V	Graphalgorithmen: transitive Hülle, Spannbäume, kürzester Pfad, Fluss in Netzwerken.
Kapitel VI	Schnelle Fouriertransformation: Grundlagen der FT, FFT Algorithmus, Anwendungen aus der Signal- und Bildverarbeitung
Kapitel VII	Suche in Zeichenketten mit Hashing, mit Automaten, mit dynamischer Programmierung (approximative Z.); Anwendung in der Bioinformatik
Kapitel VIII	Geometrische Algorithmen und Entwurfsverfahren

I. Einführung

- Sortieren durch Einfügen
- Sortieren durch Mischen
- Laufzeitabschätzungen
- Lösen von Rekurrenzen

Sortieren

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$.

Ausgabe: Eine Umordnung $\langle a_{\pi 1}, a_{\pi 2}, \ldots, a_{\pi n} \rangle$, sodass $a_{\pi 1} \leq a_{\pi 2} \leq \cdots \leq a_{\pi n}$.
Hierbei muss $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Bijektion sein.

Beispiel: $\langle 31, 41, 59, 26, 41, 58 \rangle \mapsto \langle 26, 31, 41, 41, 58, 59 \rangle$.

Allgemeiner: „Zahlen“ $\mapsto$ „Objekte“, „$\leq$“ $\mapsto$ „Ordnungsrelation“.

Beispiele:

- Zeichenketten, lexikographische Ordnung.
- Autos, Baujahr.
- Geometrische Objekte, Entfernung.

Sortieren durch Einfügen

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Einfügen von A[j] in die sortierte Folge A[1..j − 1]
4          i ← j − 1
5          while i > 0 und A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Pseudocode

Um von Implementierungsdetails zu abstrahieren, verwenden wir zur Angabe von Algorithmen Pseudocode: Eine PASCAL oder C ähnliche Sprache, welche

- Neben formellen Anweisungen und Kontrollstrukturen auch Umgangssprache enthalten kann,
- Verallgemeinerte Datentypen (wie Mengen, Graphen, etc.) bereitstellt,
- Blockstruktur auch durch Einrückung kennzeichnet (wie in Python)

Laufzeitanalyse

Wir wollen die Laufzeit eines Algorithmus als Funktion der Eingabegröße ausdrücken.

Manchmal auch den Verbrauch an anderen Ressourcen wie Speicherplatz, Bandbreite, Prozessorzahl.

Laufzeit bezieht sich auf ein bestimmtes Maschinenmodell, hier RAM (*random access machine*).

Laufzeit kann neben der Größe der Eingabe auch von deren Art abhängen (*worst case*, *best case*, *average case*). Meistens *worst case*.

Laufzeit von Insertion-Sort

Insertion-Sort(A)	Zeit	Wie oft?
1 **for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$		
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

t_j = Anzahl der Durchläufe der *while*-Schleife im j-ten Durchgang.

$c_1 - c_8$ = unspezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

Bester Fall: Array bereits sortiert

Ist das Array bereits aufsteigend sortiert, so wird die *while*-Schleife jeweils nur einmal durchlaufen: $t_j = 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist $T(n)$ eine lineare Funktion der Eingabegröße n.

Schlechtester Fall: Array absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen: $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 (\frac{n(n+1)}{2} - 1) + (c_6 + c_7) \frac{n(n-1)}{2}$$

Also ist $T(n)$ eine quadratische Funktion der Eingabegröße n.

worst case und *average case*

Meistens geht man bei der Analyse von Algorithmen vom (*worst case*) aus.

- *worst case* Analyse liefert obere Schranken
- In vielen Fällen ist der *worst case* die Regel
- Der aussagekräftigere (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (*average case*) ist oft bis auf eine multiplikative Konstante nicht besser als der *worst case*.
- Belastbare Annahmen über die mittlere Verteilung von Eingaben sind oft nicht verfügbar.
- Manchmal muss man aber eine *average case* Analyse durchführen (Beispiel: Quicksort)
- Manchmal zeigt sich auch, dass in der Praxis der *worst case* selten auftritt (Beispiel: Simplexverfahren, Typinferenz in ML). Abhilfe *parametrisierte Komplexität*, *smoothed analysis*.

Größenordnungen

Um Rechnungen zu vereinfachen und da Konstanten wie $c_1, \ldots, c_8$ sowieso willkürlich sind, beschränkt man sich oft darauf, die Größenordnung der Laufzeit anzugeben:

$$an^2 + bn + c = O(n^2)$$

$$an + b = O(n)$$

$$a2^n + bn^{10000} = O(2^n)$$

$O(f(n))$ bezeichnet alle Funktionen der Größenordnung höchstens $f(n)$. Dies wird später formal definiert.

Laufzeit von Insertion-Sort im schlechtesten Fall ist $O(n^2)$.

Es gibt auch gemischte Notationen wie $2n^2 + O(n)$.

Teile und herrsche

Das Entwurfsverfahren *divide-and-conquer* (Teile und Herrsche, *divide et impera*) dient dem Entwurf rekursiver Algorithmen.

Die Idee ist es, ein Problem der Größe n in mehrere gleichartige aber kleinere Teilprobleme zu zerlegen (*divide*).

Aus rekursiv gewonnenen Lösungen der Teilprobleme gilt es dann, eine Gesamtlösung für das ursprüngliche Problem zusammenzusetzen (*conquer*).

Beispiel: Sortieren durch Mischen (*merge sort*):

Teile n-elementige Folge in zwei Teilfolgen der Größe $n/2 \pm 1$.

Sortiere die beiden Teilfolgen rekursiv.

Füge die nunmehr sortierten Teilfolgen zusammen durch Reißverschlussverfahren.

Sortieren durch Mischen

```
MERGE-SORT(A, p, r)
    ▷ Sortiere A[p..r]
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3       MERGE-SORT(A, p, q)
4       MERGE-SORT(A, q + 1, r)
5       MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
    ▷ Sortiere A[p..r] unter der Annahme, dass A[p..q] und A[q + 1..r] sortiert sind.
1   i ← p; j ← q + 1
2   for k ← 1 to r − p + 1 do
3       if j > r or (i ≤ q and A[i] ≤ A[j])
4       then B[k] ← A[i]; i ← i + 1 else B[k] ← A[j]; j ← j + 1
5   for k ← 1 to r − p + 1 do A[k + p − 1] ← B[k]
```

Analyse von Merge-Sort

Sei $T(n)$ die Laufzeit von Merge-Sort.

Das Aufteilen braucht $O(1)$ Schritte.

Die rekursiven Aufrufe brauchen $2T(n/2)$ Schritte.

Das Mischen braucht $O(n)$ Schritte.

Also:

$$T(n) = 2T(n/2) + O(n), \text{ wenn } n > 1$$

NB $T(1)$ ist irgendein fester Wert.

Die Lösung dieser Rekurrenz ist $T(n) = O(n \log(n))$.

Für große n ist das besser als $O(n^2)$ trotz des Aufwandes für die Verwaltung der Rekursion.

NB: $\log(n)$ bezeichnet den Zweierlogarithmus ($\log(64) = 6$). Bei O-Notation spielt die Basis des Logarithmus keine Rolle, da alle Logarithmen proportional sind. Z.B.: $\log(n) = \ln(n)/\ln(2)$.

Motivation der Lösung

Intuitiv: Rekursionstiefe: $\log(n)$, auf dem Tiefenniveau k hat man 2^k Teilprobleme der Größe jeweils $g_k := n/2^k$, jedes verlangt einen Mischaufwand von $O(g_k) = O(n/2^k)$. Macht also $O(n)$ auf jedem Niveau: $O(n \log(n))$ insgesamt.

Durch Formalismus: Sei $T(n)$ für Zweierpotenzen n definiert durch

$$T(n) = \begin{cases} 0, \text{ wenn } n = 1 \\ 2T(n/2) + n, \text{ wenn } n = 2^k, k \geq 1 \end{cases}$$

Dann gilt $T(2^k) = k2^k$, also $T(n) = n \log(n)$ für $n = 2^k$.

Beweis durch Induktion über k (oder n).

Dass es dann für Nicht-Zweierpotenzen auch gilt, kann man beweisen.

Asymptotik: Definition von O

Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ Funktionen. (NB Geht auch für andere Definitionsbereiche.)

$$O(g) = \{f \mid \text{es ex. } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$$

$O(g)$ ist also eine Menge von Funktionen: die Funktionen der Größenordnung höchstens g.

In Ermangelung guter Notation für Funktionen schreibt man z.B. $O(n^2)$ und meint damit $O(g)$ wobei $g(n) = n^2$.

Bei $O(m + n^2)$ wird's schwierig. Da muss aus dem Zusammenhang klar werden, auf welche Variable sich das O bezieht.

Noch schlimmer sind Notationen wie $O(1 + \alpha)$, wobei $\alpha = n/m$, kommt aber vor...

Man schreibt $f = O(g)$ anstelle von $f \in O(g)$.

Asymptotik: Definition von O

Beispiele:

$$4n^2 + 10n + 3 = O(n^2)$$
$$an + b = O(n)$$
$$a2^n + bn^{10000} = O(2^n)$$
$$n = O(2^n)$$

$$3n^4 + 5n^3 + 7\log(n) \leq 3n^4 + 5n^4 + 7n^4 = 15n^4 = O(n^4)$$

Asymptotik: Ausdrücke mit O

Kommt $O(g)$ in einem Ausdruck vor, so bezeichnet dieser die Menge aller Funktionen, die man erhält, wenn man die Elemente von $O(g)$ für $O(g)$ einsetzt.

Z.B. ist $n^2 + 100n + 2\log(n) = n^2 + O(n)$ aber $2n^2 \neq n^2 + O(n)$.

Manchmal kommt O sowohl links, als auch rechts des Gleichheitszeichens vor. Man meint dann eine Inklusion der entsprechenden Mengen. M.a.W. jede Funktion links kommt auch rechts vor.

Z.B.

$$2n^2 + O(n) = O(n^2)$$

Die Verwendung von „=“ statt „$\in$“ oder „$\subseteq$“ ist zugegebenermaßen etwas unbefriedigend, z.B. nicht symmetrisch, hat sich aber aus praktischen Gründen durchgesetzt.

Asymptotik: $\Theta, \Omega, o, \omega$

Seien wieder $f, g : \mathbb{N} \to \mathbb{R}$. (NB Geht auch für andere Definitionsbereiche.)

$$\Omega(g) = \{f \mid g \in O(f)\} = \{f \mid \text{es ex. } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } cg(n) \leq f(n) \text{ für alle } n \geq n_0\}$$
$$\Theta(g) = O(g) \cap \Omega(g)$$

$f = \Omega(g)$ heißt: g ist eine asymptotische untere Schranke für f.

$f = \Theta(g)$ heißt: f ist von derselben Größenordnung wie g.

Oft wird O im Sinne von Θ gebraucht.

Kleines o und ω

$$o(g) = \{f \mid \text{für alle } c > 0 \text{ gibt es } n_0 \in \mathbb{N} \text{ so dass } 0 \leq f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$$

$f = o(g)$ heisst: g ist eine asymptotische obere Schranke für f und f ist nicht asymptotisch proportional zu g. f ist gegenüber g verschwindend gering.

Beachte: $f = o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Man schreibt auch $f \in \omega(g)$ für $g \in o(f)$, also $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Beispiele

$2n^2 = O(n^2)$, aber$2n^2 \neq o(n^2)$

$1000n = O(n^2)$ und sogar $1000n = o(n^2)$

Beispiele für Asymptotik

- $n^2 = o(n^3)$
- $10^{10}n^2 = O(10^{-10}n^2)$ aber nicht mit o statt O.
- $f = o(g)$ impliziert $f = O(g)$
- $\Theta(\log_b(n)) = \Theta(\log_c(n))$. NB Cormen schreibt $\lg(x)$ für $\log_2(x) = \log(x)$. Allgemein schreibt man $\ln(x)$ für $\log_e(x)$.
- $f = \Theta(g)$ und $g = \Theta(h)$ impliziert $f = \Theta(h)$. Ähnliches gilt für O, Ω, o, ω.
- $\log n = o(n^\epsilon)$ für jedes $\epsilon > 0$.
- $n! = O(n^n)$
- $\log(n!) = \Omega(n \log n)$
- $\log(n!) = \Theta(n \log n)$
- $\Theta(f) + \Theta(g) = \Theta(f + g)$ und umgekehrt und auch für O, Ω, o, ω.

Asymptotik und Induktion

Will man eine asymptotische Beziehung durch Induktion beweisen, so muss man die Konstante c ein für alle Mal wählen und darf sie nicht während des Induktionsschritts abändern:

Sei $S(n) = \sum_{i=1}^{n} i$, also $S(n) = n(n+1)/2$.

Falscher Beweis von $S(n) = O(n)$:

$S(1) = O(1)$.

Sei $S(n) = O(n)$, dann gilt $S(n+1) = S(n) + (n+1) = O(n) + O(n) = O(n)$.

Asymptotik und Induktion

Richtiger Beweis von $S(n) = \Omega(n^2)$:

Wir versuchen die Induktion mit einem noch zu bestimmenden c durchzukriegen und leiten Bedingungen an diese Konstante ab.

$S(1) = 1$, daraus $c \leq 1$.

Sei $S(n) \geq cn^2$. Es ist
$S(n+1) = S(n) + n + 1 \geq cn^2 + n + 1 \geq cn^2 + 2cn + c = c(n+1)^2$ falls $c \leq 1/2$.

Also funktioniert's mit $c \leq 1/2$, insbesondere $c = 1/2$.

Die Induktion fing bei $n = 1$ an, also können wir $n_0 = 1$ nehmen. □

Bemerkung: Man kann im Prinzip immer $n_0 = 1$ wählen und eventuelle Sprünge der zu beschränkenden Funktion durch Vergrößern/Verkleinern von c auffangen. Die Definition mit n_0 ist aber in der Praxis flexibler und auf Funktionen mit negativen Werten verallgemeinerbar.

Lösen von Rekurrenzen bei *divide and conquer*

Bei der Analyse von *divide-and-conquer* Algorithmen stößt man auf Rekurrenzen der Form:

$$T(n) = aT(n/b) + f(n)$$

Das passiert dann, wenn ein Problem der Größe n in a Teilprobleme der Größe n/b zerlegt wird und der Aufwand für das Aufteilen und Zusammenfassen der Teilresultate Aufwand $f(n)$ erfordert.

Bemerkung: n/b steht hier für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$.

Mehr noch: $aT(n/b)$ kann man hier sogar als $a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil)$ mit $a_1 + a_2 = a$ lesen.

Die *Master-Methode* liefert eine kochrezeptartige Lösung für derartige Rekurrenzen.

Hauptsatz der Master-Methode

Satz: Seien $a \geq 1, b > 1$ Konstanten, $f, T : \mathbb{N} \to \mathbb{R}$ Funktionen und gelte

$$T(n) = aT(n/b) + f(n)$$

Dann erfüllt T die folgenden Größenordnungsbeziehungen:

1. Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$, so gilt $T(n) = \Theta(n^{\log_b a})$.
2. Wenn $f(n) = \Theta(n^{\log_b a})$, so gilt $T(n) = \Theta(n^{\log_b a} \log(n))$.
3. Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und außerdem $af(n/b) \leq cf(n)$ für ein $c < 1$ und genügend großes n, so gilt $T(n) = \Theta(f(n))$.

Zum Beweis, siehe CORMEN Abschnitt 4.4.

Beachte: Ist $f(n) = n^{\log_b(a)+\epsilon}$, so gilt
$af(n/b) = an^{log_b(a)+\epsilon}/b^{\log_b(a)+\epsilon} = af(n)/a/b^\epsilon \leq cf(n)$ mit $c = b^{-\epsilon} < 1$.

Beachte: $\log_b(a) = \log(a)/\log(b)$.

Wichtigster Spezialfall: Wenn $T(n) = aT(n/a) + O(n)$, dann $T(n) = O(n \log n)$.

Beispiele für die Master Methode

- Die Laufzeit $T(n)$ von MERGE-SORT genügt der Beziehung:
  $T(n) = 2T(n/2) + \Theta(n)$ somit $T(n) = \Theta(n^{\log_2(2)} \log(n)) = \Theta(n \log(n))$.
- Wenn $T(n) = 2T(n/3) + n$ dann $T(n) = \Theta(n)$.
- Wenn $T(n) = 2T(n/2) + n^2$ dann $T(n) = \Theta(n^2)$.
- Wenn $T(n) = 4T(n/2) + n$ dann $T(n) = \Theta(n^2)$.
- Wenn $T(n) = aT(n/a) + O(n)$ dann $T(n) = O(n \log n)$. Analog für Θ.
- Die Rekurrenz $2T(n/2) + n \log n$ kann man mit der Master-Methode nicht lösen. Die Lösung ist hier $T(n) = \Theta(n \log^2(n))$.

Matrizenmultiplikation

Seien $A = (a_{ik}), B = (b_{ik})$ zwei $n \times n$ Matrizen. Das Produkt $C = AB$ ist definiert durch

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

Das macht $\Theta(n^3)$ Additionen und Multiplikationen.

Matrizen kann man blockweise multiplizieren:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Die "kleinen" Produkte AB sind hier selbst wieder Matrizenprodukte.

Die blockweise Multiplikation führt auf dasselbe Ergebnis, wie die direkte Verwendung der Definition.

Matrizenmultiplikation mit *divide-and-conquer*

$T(n) =$ Anzahl der Operationen erforderlich zur Multiplikation zweier $n \times n$ Matrizen

Es ist mit *divide-and-conquer*:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Es ist $\log_2(8) = 3$ und $\Theta(n^2) = O(n^{3-\epsilon})$, z.B. mit $\epsilon = 1$, also $T(n) = \Theta(n^3)$.

Keine Verbesserung, da auch die direkte Verwendung der Definition $\Theta(n^3)$ Rechenoperationen ($\Theta(n^3)$ Multiplikationen, $\Theta(n^2)$ Additionen.)

Strassens erstaunlicher Algorithmus

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} R & S \\ T & U \end{pmatrix}$$

wobei (V. Strassens geniale Idee):

$$R = P_5 + P_4 - P_2 + P_6 \qquad S = P_1 + P_2$$
$$T = P_3 + P_4 \qquad U = P_5 + P_1 - P_3 - P_7$$
$$P_1 = A(F - H) \qquad P_2 = (A + B)H$$
$$P_3 = (C + D)E \qquad P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H) \qquad P_6 = (B - D)(G + H)$$
$$P_7 = (A - C)(E + F)$$

Nunmehr ist

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Also: $T(n) = \Theta(n^{\log_2(7)}) = O(n^{2,81})$.

II. Sortieren und Suchen

- Heapsort
- Quicksort
- Vergleichskomplexität
- Maximum und Minimum
- Median und Selektion

Heapsort

Heapsort ist ein Verfahren, welches ein Array der Größe n ohne zusätzlichen Speicherplatz in Zeit $O(n \log n)$ sortiert.

Dies geschieht unter gedanklicher Verwendung einer baumartigen Datenstruktur, dem *heap*.

Aus einem *heap* kann man in logarithmischer Zeit das größte Element entfernen. Sukzessives Entfernen der größten Elemente liefert die gewünschte Sortierung.

Man kann auch neue Elemente in logarithmischer Zeit einfügen, was alternativ eine Vorgehensweise wie bei Insertion-Sort erlaubt.

Heaps

Ein *heap* (dt. „Halde“) ist ein binärer Baum mit den folgenden Eigenschaften

H1 Die Knoten und die Blätter des Baums sind mit Objekten beschriftet (hier Zahlen).

H2 Alle Schichten sind gefüllt bis auf den rechten Teil der Untersten. M.a.W. alle Pfade haben die Länge d oder $d-1$,; hat ein Pfad die Länge d, so auch alle Pfade zu weiter links liegenden Blättern.

H3 Die Beschriftungen der Nachfolger eines Knotens sind kleiner oder gleich den Beschriftungen des Knotens.

Repräsentation von Heaps

Ein *heap* A wird im Rechner als Array A zusammen mit einer Zahl *heap-size*$[A]$ repräsentiert (und nicht als verzeigerter Baum)

- Der Eintrag 1 bildet die Wurzel des *heaps*.
- Der Elternknoten des Eintrags i ist der Eintrag $\lfloor i/2 \rfloor$.
- Die linken und rechten Nachfolger des Eintrags i sind die Einträge $2i$ und $2i+1$. Übersteigt dieser Wert die Größe *heap-size*$[A]$, so existiert der entsprechende Nachfolger nicht.

Die Eigenschaften H1 und H2 sind für ein Array von Objekten automatisch gegeben. H3 bedeutet, dass $A[\lfloor i/2 \rfloor] \geq A[i]$ für alle $i \leq$ *heap-size*$[A]$.

Beispiel:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Die Höhe eines *heaps* der Größe n ist $\Theta(\log(n))$.

NB: $\log_b =$ „Anzahl der Male, die man durch b dividieren kann, bevor man 1 erreicht.“

Prozedur HEAPIFY: Spezifikation

Spezifikation von HEAPIFY(A, i):

- Wir sagen, der Teilbaum mit Wurzel i erfülle die Heapeigenschaft wenn gilt $A[j] \leq A[j/2]$ für alle von i aus erreichbaren Knoten j.
- Vor Aufruf mögen die Teilbäume mit Wurzeln $2i$ und $2i + 1$ die Heapeigenschaft erfüllen.
- Dann erfüllt i nach Aufruf von HEAPIFY(A, i) die Heapeigenschaft.
- Die *Menge* der Knoten des Teilbaums mit Wurzel i ändert sich dabei nicht; die Knoten können aber umstrukturiert werden. Der Heap außerhalb des Teilbaums bleibt unverändert.

NB Erfüllt ein Knoten die Heapeigenschaft so auch alle seine „Kinder“. Verletzt ein Knoten die Heapeigenschaft so auch alle seine „Vorfahren“.

Prozedur HEAPIFY: Implementierung

```
HEAPIFY(A, i)
1   l ← 2i
2   r ← 2i + 1
3   if l ≤ heap-size[A] und A[l] > A[i]
4       then largest ← l
5       else largest ← i
6   if r ≤ heap-size[A] und A[r] > A[largest]
7       then largest ← r
8   if largest ≠ i
9       then exchange A[i] ↔ A[largest]
10      HEAPIFY(A, largest)
```

Prozedur HEAPIFY: Laufzeitanalyse

Sei $h(i)$ die Höhe des Knotens i, also die Länge des längsten Pfades von i zu einem Blatt.

NB: $h(i) = O(\log(\mathit{heap\text{-}size}[A]))$.

Sei $T(h)$ die maximale Laufzeit von HEAPIFY(A, i) wenn $h(i) = h$.

Es gilt $T(h) = O(h)$, also $T(h) = O(\log(\mathit{heap\text{-}size}[A]))$.

Prozedur Build-Heap

Wir wollen die Einträge eines beliebigen Arrays so permutieren, dass ein *heap* entsteht.

```
Build-Heap(A)
1  heap-size[A] ← length[A]
2  for i ← heap-size[A]/2 downto 1 do
3        Heapify(A, i)
4        ▷ Alle Teilbäume mit Wurzel ≥ i erfüllen die Heapeigenschaft
```

Nach Aufruf von Build-Heap(A) enthält A dieselben Einträge wie zuvor, aber nunmehr bildet A einen *heap* der Größe n.

Prozedur BUILD-HEAP: Laufzeitanalyse

Ein *heap* der Größe n enthält maximal $\lceil n/2^{h+1} \rceil$ Knoten der Höhe h.

Die Laufzeit von BUILD-HEAP(A) ist somit:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

Hier haben wir verwendet:

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x} \quad \Big| \quad \frac{\mathrm{d}}{\mathrm{d}x}$$

$$\sum_{h=0}^{\infty} h x^{h-1} = \frac{1}{(1-x)^2} \quad \Big| \quad x$$

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

Also gilt $\sum_{h=0}^{\infty} h/2^h = (1/2)/(1/2)^2 = 2$ mit $x = 1/2$.

Merke: BUILD-HEAP(A) läuft in $O(\mathit{length}[A])$.

Prozedur Heap-Sort

```
Heap-Sort(A)
1  Build-Heap(A)
2  for i ← length[A] downto 2 do
3       exchange A[1] ↔ A[i]
4       heap-size[A] ← heap-size[A] − 1
5       Heapify(A, 1)
```

Laufzeit von Heap-Sort(A) mit $\mathit{length}[A] = n$ ist $O(n \log n)$.

Prioritätsschlangen

Eine Prioritätsschlange (*priority queue*) ist eine Datenstruktur zur Verwaltung einer Menge von Objekten, die linear geordnete Schlüssel als Attribute besitzen. Eine Prioritätsschlange unterstützt die folgenden Operationen:

- INSERT(S, x): Einfügen des Elements x in die Schlange S.
- MAXIMUM(S) liefert das (ein) Element von S mit dem größten Schlüssel.
- EXTRACT-MAX(S) liefert das (ein) Element von S mit dem größten Schlüssel und löscht es aus S.

Wird S als *heap* organisiert, so laufen alle drei Operationen jeweils in Zeit $O(\log(|S|))$, außerdem erfordert das einmalige Aufbauen eines *heaps* nur lineare Zeit.

Prozedur Heap-Insert

```
Heap-Insert(A, key)
1   heap-size[A] ← heap-size[A] + 1
2   i ← heap-size[A]
3   while i > 1 and A[i/2] < key do
4         A[i] ← A[i/2]
5         i ← i/2
5   A[i] ← key
```

Verfolgt den Pfad vom ersten freien Blatt ($heap\text{-}size + 1$) zur Wurzel bis der Platz für key gefunden ist. Laufzeit $O(\log n)$.

Quicksort

```
QUICKSORT(A, p, r)
    ▷ Sortiere A[p..r]
1   if p < r then
2       q ← PARTITION(A, p, r)
3       QUICKSORT(A, p, q)
4       QUICKSORT(A, q + 1, r)
```

Die Prozedur PARTITION(A, p, r) arbeitet wie folgt: Man setzt $x = A[p]$ (Pivot) und gruppiert die Elemente von $A[p..r]$ um und bestimmt einen Index $q \in \{p, \ldots, r-1\}$ sodass nach der Umgruppierung gilt: $A[p..q] \leq k \leq A[q+1..r]$.

Prozedur Partition

```
PARTITION(A, p, r)
1  x ← A[p]
2  i ← p − 1
3  j ← r + 1
4  while TRUE do
5        repeat j ← j − 1 until A[j] ≤ x
6        repeat i ← i + 1 until A[i] ≥ x
7        if i < j
8              then exchange A[i] ↔ A[j]
9              else return j
```

Beachte:

- In Zeile 4 gilt die Invariante $A[p..i] \leq x \leq A[j..r]$
- $q = r$ ist nicht möglich
- $q = r$ wäre möglich wenn man Zeile 1 durch $A[r]$ ersetzte

Laufzeit von Quicksort

Sei $n = r - p + 1$ die Größe des zu bearbeitenden Arrays.

Der Aufruf Partition(A, p, r) hat Laufzeit $\Theta(n)$.

Sei $T(n)$ die Laufzeit von Quicksort(A, p, r).

Es gilt $T(n) = T(n_1) + T(n_2) + \Theta(n)$ wobei $n_1 + n_2 = n$.

Bester Fall: $n_1, n_2 = O(n)$, z.B., $= n/2$. Dann ist $T(n) = \Theta(n \log n)$.

Schlechtester Fall: $n_1 = O(n), n_2 = O(1)$ oder umgekehrt. Dann ist $T(n) = \Theta(n^2)$.

Randomisiertes Quicksort

Der schlechteste Fall tritt tatsächlich auf wenn das Array schon mehr oder weniger sortiert ist.

Beispiel: Buchungen sind nach Eingangsdatum sortiert, sollen nach Buchungsdatum sortiert werden.

Um diesen Effekt zu vermeiden, wählt man das Pivotelement zufällig:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[p] ↔ A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r then
2     q ← RANDOMIZED-PARTITION(A, p, r)
3     RANDOMIZED-QUICKSORT(A, p, q)
4     RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Erwartete Laufzeit

Die Laufzeit von RANDOMIZED-QUICKSORT ist nunmehr eine Zufallsvariable ohne festen Wert.

Wir bestimmen ihren Erwartungswert. Wir nehmen vereinfachend an, dass alle Elemente verschieden sind.

Wie groß wird der Teil $n_1 = q - p + 1$ der Partition?

Das hängt vom Rang des Pivotelements ab.

Ist das Pivotelement das Kleinste, so ist $n_1 = 1$.

Ist das Pivotelement das Zweitkleinste, so ist $n_1 = 1$.

Ist das Pivotelement das Drittkleinste, so ist $n_1 = 2$.

Ist das Pivotelement das $n-1$kleinste, so ist $n_1 = n - 2$.

Ist das Pivotelement das Größte, so ist $n_1 = n - 1$.

Also gilt für den Erwartungswert der Laufzeit $T(n)$:

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1} T(q) + T(n-q)\right) + \Theta(n)$$

Explizite Bestimmung der erwarteten Laufzeit

Wir wissen bereits, dass $T(n) = O(n^2)$ somit $T(n)/n = O(n)$ und somit können die ersten beiden Summanden durch $\Theta(n)$ absorbiert werden:

$$T(n) = \frac{1}{n}(\sum_{q=1}^{n-1} T(q) + T(n-q)) + \Theta(n) = \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n)$$

Wir raten $T(n) = O(n \log n)$ und probieren durch Induktion über n zu zeigen $T(n) \leq cn \ln n$ für ein noch zu bestimmendes $c > 0$.

Sei n groß genug und fest gewählt. Es gelte $T(q) \leq cq \ln q$ für alle $q < n$.

$$T(n) \leq \frac{2c}{n}\sum_{q=1}^{n-1} q \ln q + dn \text{ (das “}d\text{” kommt vom } \Theta(n))$$

Es ist $\sum_{q=1}^{n-1} q \ln q \leq \int_{q=1}^{n} q \ln q \, \mathrm{d}q = [\frac{1}{2}q^2 \ln q - \frac{1}{4}q^2]_1^n \leq \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2$.

Mit $c \geq 2d$ bekommen wir also $T(n) \leq cn \ln n$ somit ist $T(n) = O(n \log n)$ erwiesen.

Untere Schranke für vergleichsbasiertes Sortieren

Sei $\mathcal{A}$ irgendein Algorithmus, welcher n Objekte sortiert, indem auf die Objekte nur über binäre Vergleiche der Form "$o_1 \leq o_2$?" zugegriffen wird.

Also nicht durch explizites Lesen der Objekte.

Beispiel aus `stdlib.h`

```
void qsort(void A[], int size, int compare(void *,void *))
```

Anderes Beispiel: das Java Interface `Comparable`.

Wir behaupten: $\mathcal{A}$ erfordert $\Omega(n \log n)$ solche Vergleiche.

Beweis der unteren Schranke

Nehmen wir an, $\mathcal{A}$ führe V Vergleiche durch.

Welche Elemente hier verglichen werden, hängt i.a. vom Ausgang vorhergehender Vergleiche ab!

Die möglichen Ausgänge dieser Vergleiche partitionieren die möglichen Eingaben in 2^V Klassen.

Eingaben, die in die gleiche Klasse fallen, werden gleich behandelt.

Jede der $n!$ Permutationen von n verschiedenen Objekten erfordert unterschiedliche Behandlung.

Also muss gelten $2^V \geq n!$ oder $V \geq \log_2 n! = \Omega(n \log n)$.

[Beweis, dass $\log_2 n! = \Omega(n \log n)$.
$\log n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 i \geq \frac{n}{2} \log_2(\frac{n}{2}) = \frac{n}{2}(\log_2(n) - 1) = \Omega(n \log n)$.]

Bestimmung des Maximums

Das Array A enthalte n verschiedene Zahlen.

Folgender Algorithmus bestimmt das Maximum der $A[i]$.

```
MAXIMUM(A, n)
▷  Bestimmt das größte Element von A, wenn n ≥ 0
1  kandidat ← A[1]
2  for i ← 2 to n do
▷  kandidat ≥ A[1..i − 1]
3      if A[i] > kandidat then kandidat ← A[i]
4  return kandidat
```

Die Vergleichskomplexität dieses Verfahrens beträgt $V(n) = n - 1$.

Soll heißen, $n - 1$ Größenvergleiche werden durchgeführt.

Ganz analog haben wir ein Verfahren MINIMUM, das das kleinste Element bestimmt.

Vergleichskomplexität des Maximums

Die Vergleichskomplexität des Problems „Maximumbestimmung“ ist die minimale Zahl von Vergleichen, die im schlechtesten Fall erforderlich sind, um das Maximum zu bestimmen.

Die Existenz des Verfahrens MAXIMUM belegt $V(n) \leq n - 1$.

Es gilt tatsächlich $V(n) = n - 1$.

Vergleichskomplexität des Maximums

- Sei M die Menge der Positionen im Array, an denen aufgrund der bis dato gemachten Vergleiche noch das Maximum stehen könnte.
- Am Anfang ist $M = \{1, \ldots, n\}$. Am Ende muss $|M| = 1$ sein.
- Aus M entfernen können wir eine Position i nur dann, wenn ein Vergleich stattgefunden hat, in dem $A[i]$ das kleinere Element ist.
- Ein Vergleich entfernt also höchstens ein Element aus M.
- $n - 1$ Vergleiche sind erforderlich.
- Das gilt ganz gleich wie die Vergleiche ausgehen, also auch im besten Fall.

Maximum und Minimum gleichzeitig

Es gelte, simultan das größte und das kleinste Element in einem Array zu bestimmen.

Anwendung: Skalierung von Messwerten.

Durch Aufruf von MAXIMUM und dann MINIMUM erhalten wir einen Algorithmus für dieses Problem mit Vergleichskomplexität $2n - 2$.

Somit gilt für die Vergleichskomplexität $V(n)$ des Problems „Maximum und Minimum“

$$V(n) \leq 2n - 2$$

Ist das optimal?

Maximum und Minimum gleichzeitig

```
MAXIMUM-MINIMUM(A, n)
▷ Bestimmt das Maximum und das Minimum in A[1..n]
1  for i ← 1 to ⌊n/2⌋ do
2      if A[2i − 1] < A[2i]
3          then B[i] ← A[2i − 1]; C[i] ← A[2i]
4          else C[i] ← A[2i − 1]; B[i] ← A[2i]
5  if n ungerade
6      then B[⌊n/2⌋ + 1] ← A[n]; C[⌊n/2⌋ + 1] ← A[n]
7  return (MINIMUM(B, ⌈n/2⌉), MAXIMUM(C, ⌈n/2⌉))
```

- Die Elemente werden zunächst in $\lceil n/2 \rceil$ verschiedenen Paaren verglichen. Das letzte Paar besteht aus zwei identischen Elementen, falls n ungerade.
- Das Maximum ist unter den $\lceil n/2 \rceil$ „Siegern“; diese befinden sich in C.
- Das Minimum ist unter den $\lceil n/2 \rceil$ „Verlierern“; diese befinden sich in B.

Es gilt also $V(n) \leq \lceil \frac{3n}{2} \rceil - 2$. Das ist optimal.

Untere Schranke für Maximum und Minimum

Satz: Jeder Algorithmus, der simultan Minimum und Maximum eines Arrays A mit $\mathit{length}[A] = n$ bestimmt, benötigt mindestens $\lceil \frac{3n}{2} \rceil - 2$ Vergleiche.

Seien $K_{\max}$ und $K_{\min}$ die Mengen der Indizes i, für die $A[i]$ noch als Maximum bzw. Minimum in Frage kommen.

$K_{\max}$ enthält diejenigen i, für die $A[i]$ noch bei keinem Vergleich "verloren" hat.

Bei jedem Vergleich werden $K_{\min}$ und $K_{\max}$ jeweils höchstens um ein Element kleiner. Ein Vergleich ist ein

- Volltreffer, falls $K_{\min}$ und $K_{\max}$ kleiner werden,
- Treffer, falls nur eines von $K_{\min}$ und $K_{\max}$ kleiner wird,
- Fehlschuss sonst.

Untere Schranke für Maximum und Minimum

Wir werden zeigen:

Lemma: Für jeden Algorithmus gibt es eine Eingabe, bei der er nur $\lfloor \frac{n}{2} \rfloor$ Volltreffer landet.

Beweis des Satzes:

Am Anfang ist $|K_{\max}| = |K_{\min}| = n$, am Ende soll $|K_{\max}| = |K_{\min}| = 1$ sein.

Es sind also $2n - 2$ Elemente aus $K_{\min}$ und $K_{\max}$ zu entfernen.

Werden nur $\lfloor \frac{n}{2} \rfloor$ Volltreffer gelandet, muss es also noch

$$2n - 2 - 2\lfloor \frac{n}{2} \rfloor \;=\; 2\lceil \frac{n}{2} \rceil - 2$$

Treffer geben, also ist die Zahl der Vergleiche mindestens

$$\lfloor \frac{n}{2} \rfloor + 2\lceil \frac{n}{2} \rceil - 2 \;=\; \lceil \frac{3n}{2} \rceil - 2\,.$$

Beweis des Lemmas über Volltreffer

Idee: Eingabe wird von einem Gegenspieler $\mathbb{A}$ (engl. adversary) während des Ablaufs konstruiert.

$\mathbb{A}$ merkt sich seine Antworten zu den Vergleichsanfragen "$A[i] \leqslant A[j]$?", und antwortet stets so, dass

(1) die Antwort mit der erzeugten partiellen Ordnung konsistent ist.

(2) möglichst kein Volltreffer erzielt wird.

Falls $i, j \in K_{\max} \cap K_{\min}$, sind i und j noch völlig frei.

$\rightsquigarrow$ $\mathbb{A}$ kann beliebig antworten, in jedem Fall ein Volltreffer.

In jedem anderen Fall kann $\mathbb{A}$ so antworten, dass nur ein Treffer oder Fehlschuss erzielt wird.

Ist z.B. $i \in K_{\max}$ und $j \in K_{\min}$, aber $i \notin K_{\min}$, so antwortet $\mathbb{A}$ mit $A[j] < A[i]$

$\rightsquigarrow$ Treffer, falls $j \in K_{\max}$, sonst Fehlschuss.

Also: Volltreffer nur, falls $i, j \in K_{\max} \cap K_{\min}$, das kann nur $\lfloor \frac{n}{2} \rfloor$ mal vorkommen.

Die Selektionsaufgabe

Die Selektionsaufgabe besteht darin, von n verschiedenen Elementen das i-kleinste (sprich: [ihtkleinste]) zu ermitteln.

Das i-kleinste Element ist dasjenige, welches nach aufsteigender Sortierung an i-ter Stelle steht.

Englisch: i kleinstes Element = ith order statistic.

Das 1-kleinste Element ist das Minimum.

Das n-kleinste Element ist das Maximum.

Das $\lfloor \frac{n+1}{2} \rfloor$-kleinste und das $\lceil \frac{n+1}{2} \rceil$-kleinste Element bezeichnet man als Median.

Ist n gerade, so gibt es zwei Mediane, ist n ungerade so gibt es nur einen.

Anwendung des Medians

Fakt: Sei $x_1, \ldots, x_n$ eine Folge von Zahlen. Der Ausdruck $S(x) = \sum_{i=1}^{n} |x - x_i|$ nimmt sein Minimum am Median der x_i an.

Beispiele

- n Messwerte x_i seien so zu interpolieren, dass die Summe der absoluten Fehler minimiert wird. Lösung: Median der x_i.
- n Städte liegen auf einer Geraden an den Positionen x_i. Ein Zentrallager sollte am Median der x_i errichtet werden um die mittlere Wegstrecke zu minimieren (unter der Annahme, dass jede Stadt gleich oft angefahren wird.)
- Analoges gilt auch in 2D bei Zugrundelegung der Manhattandistanz.

Vergleichskomplexität der Selektionsaufgabe

- Durch Sortieren kann die Selektionsaufgabe mit Vergleichskomplexität $\Theta(n \log n)$ gelöst werden, somit gilt für die Vergleichskomplexität $V(n)$ der Selektionsaufgabe: $V(n) = O(n \log n)$.
- $V(n) = \Omega(n)$ ergibt sich wie beim Maximum. Mit weniger als $n - 1$ Vergleichen kann ein Element nicht als das i-kleinste bestätigt werden.
- Tatsächlich hat man $V(n) = \Theta(n)$.

Selektion mit mittlerer Laufzeit $\Theta(n)$

RANDOMIZED-SELECT(A, p, r, i)

▷ Ordnet $A[p..r]$ irgendwie um und bestimmt den Index des i-kleinsten Elements in $A[p..r]$

1 **if** $p = r$ **then return** p

2 $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

3 $k \leftarrow q - p + 1$

4 **if** $i \leq k$

5 **then return** RANDOMIZED-SELECT(A, p, q, i)

6 **else return** RANDOMIZED-SELECT$(A, q + 1, r, i - k)$

Laufzeit (und Vergleichskomplexität) im schlechtesten Falle: $\Theta(n^2)$.

Mittlere Laufzeit von Randomized-Select

Für den Erwartungswert $V(n)$ der Laufzeit von Randomized-Select(A, p, r, i), wobei $n = r - p + 1$, gilt die Rekurrenz:

$$T(n) \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

Diese Rekurrenz hat die Lösung $T(n) = O(n)$ wie man durch Einsetzen und Induktion bestätigt.

Lineare Laufzeit im schlechtesten Fall

SELECT(A, p, r, i)

▷ Bestimmt den Index des i-kleinsten Elements in $A[p..r]$

1 **if** $p = r$ **then return** p

2 Teile die $A[i]$ in Fünfergruppen auf (plus eventuell eine kleinere Gruppe)

3 Bestimme den Median jeder Gruppe durch festverdrahtete Vergleiche

4 Bestimme durch rekursiven Aufruf von SELECT den Median dieser Mediane.

5 Vertausche in A diesen Median mit $A[r]$

6 $q \leftarrow$ PARTITION(A, p, r)

7 $k \leftarrow q - p + 1$

8 **if** $i \leq k$

9 **then return** SELECT(A, p, q, i)

10 **else return** SELECT$(A, q + 1, r, i - k)$

Worst-case Laufzeit von SELECT

Sei $T(n)$ die *worst case* Laufzeit von SELECT.

- Gruppenbildung und individuelle Mediane: $O(n)$.
- Bestimmung des Medians der Mediane: $T(n/5)$.
- Der Median der Mediane liegt oberhalb und unterhalb von jeweils mindestens $\frac{3n}{10}$ Elementen.
- Die größere der beiden „Partitionen“ hat also weniger als $\frac{7}{10}$ Elemente.
- Der rekursive Aufruf auf einer der beiden „Partitionen“ erfordert also $T(\frac{7n}{10})$.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

Die Lösung ist $T(n) = O(n)$ wie man durch Einsetzen bestätigt.

NB Die Lösung von $T(n) = T(n/5) + T(8n/10) + O(n)$ ist $O(n \log n)$.

III. Datenstrukturen

- Dynamische Mengen
- Binäre Suchbäume, Realisierung durch Rot-Schwarz-Bäume
- Hashtabellen. Kollisionsauflösung: Verkettung, offene Adressierung
- Amortisierte Komplexität
- Prioritätsschlangen, Realisierung durch Binomial Queues und Fibonacci Heaps.
- B-Bäume

Dynamische Mengen

Eine dynamische Menge ist eine Datenstruktur, die Objekte verwaltet, welche einen Schlüssel tragen, und zumindest die folgenden Operationen unterstützt:

- SEARCH(S, k): liefert (einen Zeiger auf) ein Element in S mit Schlüssel k, falls ein solches existiert; NIL sonst.
- INSERT(S, x): fügt das Element (bezeichnet durch Zeiger) x in die Menge S ein.

Oft werden weitere Operationen unterstützt, wie etwa

- DELETE(S, x): löscht das Element (bezeichnet durch Zeiger) x.
- Maximum, Minimum, Sortieren, etc. bei geordneten Schlüsseln.

Typische Anwendung: Symboltabelle in einem Compiler. Schlüssel = Bezeichner, Objekte = (Typ, Adresse, Größe, ...)

Realisierungen durch Bäume

Dynamische Mengen können durch binäre Suchbäume realisiert werden.

Notation für binäre Bäume:

- Wurzel des Baumes T gespeichert in $root[T]$.
- Jeder Knoten x hat Zeiger auf die Söhne $left[x]$ und $right[x]$, sowie auf den Vater $p[x]$ (ggf. NIL).

Die Operationen

- TREE-SEARCH(T, k)
- TREE-INSERT(T, x)
- TREE-DELETE(T, x)
- TREE-MINIMUM(T) und TREE-MAXIMUM(T)
- TREE-SUCCESSOR(T, x) und TREE-PREDECESSOR(T, x)

haben sämtlich eine Laufzeit von $O(h)$, wobei h die Tiefe des Baumes T ist.

Balancierte binäre Suchbäume

Damit die Operationen auf binären Suchbäumen effizient sind, sollten diese balanciert sein, d.h. die Höhe eines Baumes mit n Knoten ist $O(\log n)$.

Bekanntes Beispiel balancierter Suchbäume: AVL-Bäume

Balancieren erfolgt durch Rotationsoperationen:

Operation Rotate-Right(T, x)

Dual dazu ($\leftarrow$) wirkt Rotate-Left(T, y).

Rot-Schwarz-Bäume

Rot-Schwarz-Bäume: binäre Suchbäume mit zusätzlicher Struktur, die dafür sorgt, dass diese balanciert sind, also die Höhe höchstens $O(\log n)$ ist.

Jeder Knoten wird als innerer Knoten aufgefasst, Blätter sind zusätzliche, leere Knoten (Wächter).

Knoten haben ein zusätzliches Feld $color[x]$, mit den Werten RED und BLACK. Wir sprechen von roten und schwarzen Knoten.

Rot-Schwarz-Eigenschaft:

1. Die Wurzel und alle Blätter sind schwarz.
2. Beide Söhne eines roten Knotens sind schwarz.
3. Für jeden Knoten x gilt:

   jeder Pfad von x zu einem Blatt enthält gleich viele schwarze Knoten.

Eigenschaften von Rot-Schwarz-Bäumen

Wir zeigen, dass Rot-Schwarz-Bäume balanciert sind:

Satz:

> Die Höhe eines Rot-Schwarz-Baums mit n inneren Knoten ist höchstens $2\log_2(n+1)$.

Daraus folgt:

> Die Operationen Tree-Search, Tree-Minimum, Tree-Maximum, Tree-Successor und Tree-Predecessor benötigen für Rot-Schwarz-Bäume $O(\log n)$ Operationen.

Auch Tree-Insert und Tree-Delete laufen auf Rot-Schwarz-Bäumen in Zeit $O(\log n)$, erhalten aber nicht die Rot-Schwarz-Eigenschaft.

$\rightsquigarrow$ Für diese benötigen wir spezielle Einfüge- und Löschoperationen.

Einfügen in einen Rot-Schwarz-Baum

Neuer Knoten x wird mit TREE-INSERT eingefügt und rot gefärbt.

Problem: Vater $p[x]$ kann auch rot sein $\quad\leadsto\quad$ Eigenschaft 2 verletzt.

Diese Inkonsistenz wird in einer Schleife durch lokale Änderungen im Baum nach oben geschoben:

- Fall 1: $p[x] = \mathit{left}[p[p[x]]]$:
  - Betrachte x' Onkel $y = \mathit{right}[p[p[x]]]$.
  - Fall 1.1: y ist rot. Dann färbe $p[x]$ und y schwarz, und $p[p[x]]$ rot.
    Aber: $p[p[p[x]]]$ kann rot sein $\quad\leadsto\quad$ Inkonsistenz weiter oben.
  - Fall 1.2: y ist schwarz, und $x = \mathit{left}[p[x]]$. Dann wird $p[x]$ schwarz und $p[p[x]]$ rot gefärbt, und anschliessend um $p[p[x]]$ nach rechts rotiert.
    $\leadsto\quad$ keine Inkonsistenz mehr.
  - Fall 1.3: y ist schwarz, und $x = \mathit{right}[p[x]]$. Wird durch eine Linksrotation um $p[x]$ auf Fall 1.2 zurückgeführt.
- Fall 2: $p[x] = \mathit{right}[p[p[x]]]$ ist symmetrisch.

Einfügen in einen Rot-Schwarz-Baum

Am Ende wird die Wurzel $root[T]$ schwarz gefärbt.

Bemerke:

- In den Fällen 1.2 und 1.3 bricht die Schleife sofort ab.
- Schleife wird nur im Fall 1.1 wiederholt, dann ist die Tiefe von x kleiner geworden.
- Rotationen werden nur in 1.2 und 1.3 durchgeführt.

$\rightsquigarrow$ Laufzeit ist $O(\log n)$, und es werden höchstens 2 Rotationen durchgeführt.

Löschen aus einem Rot-Schwarz-Baum

Zu löschender Knoten z wird mit TREE-DELETE(T, z) gelöscht.

Erinnerung: TREE-DELETE(T, z) entfernt einen Knoten y, der höchstens einen (inneren) Sohn hat.

- Hat z höchstens einen Sohn, so ist $y = z$.
- Andernfalls $y \leftarrow$ TREE-SUCCESSOR(T, z).

Problem: Wenn der entfernte Knoten y schwarz war, ist Eigenschaft 3 verletzt.

Intuition: der (einzige) Sohn x von y erbt dessen schwarze Farbe, und ist jetzt "doppelt schwarz".

Das zusätzliche Schwarz wird in einer Schleife durch lokale Änderungen im Baum nach oben geschoben.

Löschen aus einem Rot-Schwarz-Baum

- Ist x rot, so wird es schwarz gefärbt, und die Schleife bricht ab.
  Andernfalls unterscheide zwei Fälle:
- Fall 1: $x = \mathit{left}[p[x]]$.
  - Betrachte x' Bruder $w = \mathit{right}[p[x]] \neq \textsc{nil}$, und unterscheide 2 Fälle:
  - Fall 1.1: w ist rot. Dann muss $p[x]$ schwarz sein, also färbe $p[x]$ rot,
    w schwarz und rotiere links um $p[x]$
    $\leadsto$ reduziert auf Fall 1.2.
  - Fall 1.2: w ist schwarz. Es gibt drei weitere Fälle:
    - Fall 1.2.1: Beide Kinder von w sind schwarz. Also können wir w rot
      färben, und das zusätzliche Schwarz von x zu $p[x]$ versetzen
      $\leadsto$ nächste Iteration.
    - Fall 1.2.2: $\mathit{left}[w]$ ist rot, $\mathit{right}[w]$ ist schwarz. Dann färbe w rot, $\mathit{left}[w]$
      schwarz und rotiere rechts um w
      $\leadsto$ reduziert auf Fall 1.2.3.
    - Fall 1.2.3: $\mathit{right}[w]$ ist rot. Dann vertausche die Farben von w und $p[x]$,
      färbe $\mathit{right}[w]$ schwarz, und rotiere links um $p[x]$
      $\leadsto$ Zusätzliches Schwarz ist verbraucht.

Löschen aus einem Rot-Schwarz-Baum

- Fall 2: $x = \mathit{right}[p[x]]$ ist symmetrisch.

Auch beim Löschen wird am Ende die Wurzel $\mathit{root}[T]$ schwarz gefärbt.

Bemerke:

- In den Fällen 1.2.2 und 1.2.3 bricht die Schleife sofort ab.
- Schleife wird nur im Fall 1.2.1 wiederholt, dann ist die Tiefe von x kleiner geworden.
- Im Fall 1.1 wird die Tiefe von x größer, aber die Schleife bricht danach im Fall 1.2 sofort ab.

$\rightsquigarrow$ Laufzeit ist $O(\log n)$, es werden höchstens 3 Rotationen durchgeführt.

Zusammengefasst: die Operationen SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, INSERT und DELETE können für Rot-Schwarz-Bäume mit Laufzeit $O(\log n)$ realisiert werden.

B-Bäume

B-Bäume verallgemeinern binäre Suchbäume dahingehend, dass in einem Knoten mehrere Schlüssel stehen und ein Knoten mehrere Kinder hat. (Typischerweise jeweils 500-1000).

Dadurch sinkt die Zahl der Knoten, die bei einem Suchvorgang besucht werden müssen, dafür ist aber jedes einzelne Besuchen aufwendiger.

Das ist sinnvoll, wenn die Knoten auf einem Massenspeichermedium abgelegt sind (Plattenstapel o.ä.) wo einzelne Zugriffe recht lange dauern, dafür aber gleich eine ganze *Seite* (z.B.: 1024Byte) auslesen, bzw. schreiben.

B-Bäume: Definition

Die Definition von B-Bäumen bezieht sich auf ein festes $t \in \mathbb{N}$, z.B.: $t = 512$.

Ein B-Baum ist ein Baum mit den folgenden Eigenschaften:

- Jeder Knoten x enthält die folgenden Einträge:
  - die Anzahl $n[x]$ der im Knoten gespeicherten Schlüssel, wobei $n[x] \leq 2t - 1$.
  - die in aufsteigender Reihenfolge gespeicherten Schlüssel: $key_1[x] \leq key_2[x], \ldots, key_{n[x]}[x]$
- Jeder Knoten x ist entweder ein Blatt oder ein innerer Knoten und hat dann gerade $n[x] + 1$ Kinder.
- Ist x innerer Knoten und nicht die Wurzel, so gilt $n[x] \geq t - 1$.
- Alle Schlüssel, die sich im oder unterhalb des i-ten Kindes eines Knotens x befinden, liegen bzgl. der Ordnung zwischen $key_{i-1}[x]$ und $key_i[x]$. Grenzfall $i = 1$: die Schlüssel sind kleiner als $key_1[x]$; Grenzfall $i = n[x]$: die Schlüssel sind größer als $key_{n[x]}[x]$.

Suchen in B-Bäumen

Beim Suchen vergleicht man den gesuchten Schlüssel k mit den Einträgen der Wurzel. Entweder ist k unter den Wurzeleinträgen, oder man ermittelt durch Vergleich mit den Wurzeleinträgen denjenigen Unterbaum, in welchem k, wenn überhaupt, zu finden ist und sucht in diesem rekursiv weiter.

Natürliche Verallgemeinerung des Suchens in BST.

Einfügen in B-Bäumen

1. Bestimme durch Suche das Blatt x, in welches der neue Schlüssel k aufgrund seiner Ordnungsposition gehört.
2. Falls dort noch nicht zuviele Einträge (also $n[x] < 2t - 1$) vorhanden sind, so füge den neuen Schlüssel dort ein.
3. Andernfalls füge k trotzdem in das Blatt ein (es enthält dann $2t$ Schlüssel) und teile es in zwei Blätter der Größe t und $t - 1$ auf. Der übrige Schlüssel wird dem Elternknoten als Trenner der beiden neuen Kinder hinzugefügt.
4. Führt dies zum Überlauf im Elternknoten, so teile man diesen ebenso auf, etc. bis ggf. zur Wurzel.

Löschen aus B-Bäumen

1. Zunächst ersetzt man ähnlich wie bei BST den zu löschenden Schlüssel durch einen passenden Schlüssel aus einem Blatt. Dadurch kann man sich auf das Entfernen von Schlüsseln aus Blättern beschränken.
2. Kommt es beim Entfernen eines Schlüssels aus einem Blatt b zum Unterlauf (weniger als $t-1$ Schlüssel), hat aber der rechte Nachbar b' mehr als $t-1$ Schlüssel, so gibt man den Schlüssel aus dem Elternknoten, der b von b' trennt, zu b hinzu und nimmt den kleinsten Schlüssel in b' als neuen Trenner.
3. Hat der rechte Nachbar b' von b auch nur $t-1$ Schlüssel, so verschmilzt man b und b' samt dem Trenner im Elternknoten zu einem neuen Knoten mit nunmehr $t-2+t-1+1=2t-2$ Schlüsseln. Dem Elternknoten geht dadurch ein Schlüssel verloren, was wiederum zum Unterlauf führen kann. Ein solcher ist durch rekursive Behandlung nach demselben Schema zu beheben.

Mögliche Optimierung

Cormen und auch die Wikipedia empfehlen, beim Einfügen bereits beim Aufsuchen der entsprechenden Blattposition solche Knoten, die das Höchstmaß $2t - 1$ erreicht haben, vorsorglich zu teilen, damit nachher ein Überlauf lokal behandelt werden kann und nicht wieder rekursiv nach oben gereicht werden muss.

Analog würde man beim Löschen Knoten der Mindestgröße $t - 1$ durch Ausgleichen vom Nachbarn oder, falls das nicht möglich, Verschmelzen, vorsorglich verdicken.

Man nimmt hierdurch mehr Spaltungen / Verschmelzungen vor, als unbedingt nötig, spart sich dafür aber Knotenzugriffe. Was im Einzelfall besser ist, ist unklar.

Dynamische Mengen als Hashtabelle

- Direkte Adressierung
- Hashing und Hashfunktionen
- Kollisionsauflösung durch Verkettung
- Offene Adressierung
- Analyse der erwarteten Laufzeiten

In diesem Abschnitt werden Arrayfächer beginnend mit 0 nummeriert.

Direkte Adressierung

- Sind die Schlüssel ganze Zahlen im Bereich $0 \ldots N-1$, so kann eine dynamische Menge durch ein Array A der Größe N implementiert werden.
- Der Eintrag $A[k]$ ist x falls ein Element x mit Schlüssel k eingetragen wurde.
- Der Eintrag $A[k]$ ist NIL, falls die dynamische Menge kein Element mit Schlüssel k enthält.
- Die Operationen SEARCH, INSERT, DELETE werden unterstützt und haben Laufzeit $\Theta(1)$.
- Nachteile: Enormer Speicherplatz bei großem N. Nicht möglich, falls keine obere Schranke an Schlüssel vorliegt.

Hash-Tabelle

- Sei U die Menge der Schlüssel, z.B.: $U = \mathbb{N}$.
- Gegeben eine Funktion $h : U \to \{0, 1, 2, 3, \ldots, m-1\}$, die „Hashfunktion“.
- Die dynamische Menge wird implementiert durch ein Array der Größe m.
- Das Element mit Schlüssel k wird an der Stelle $A[h(k)]$ abgespeichert.

```
SEARCH(A, k)
1  return A[h(k)]
INSERT(A, x)
1  A[h(key[x])] ← x
DELETE(A, k)
1  A[h(k)] ← NIL
```

- Zum Beispiel: $U = \mathbb{N}$, $h(k) = k \bmod m$.
- Problem: $h(k_1) = h(k_2)$ obwohl $k_1 \neq k_2$ (Kollision). Kommt es zu Kollisionen so ist dieses Verfahren inkorrekt (also gar kein Verfahren!).

Häufigkeit von Kollisionen

- Alle m Hashwerte seien gleichwahrscheinlich.
- Die Wahrscheinlichkeit, dass k zufällig gewählte Schlüssel paarweise verschiedene Hashwerte haben ist dann:

$$\begin{aligned} & 1 \cdot \tfrac{m-1}{m} \cdot \tfrac{m-2}{m} \cdots \tfrac{m-k+1}{m} = \textstyle\prod_{i=0}^{k-1}(1 - \tfrac{i}{m}) \\ \leq & \quad \textstyle\prod_{i=0}^{k-1} e^{-i/m} \\ = & \quad e^{-\sum_{i=0}^{k-1} i/m} = e^{-k(k-1)/2m} \end{aligned}$$

- Diese Wahrscheinlichkeit wird kleiner als 50% wenn $k \geq 1 + \frac{1}{2}\sqrt{1 + 8m \ln 2}$.
- Beispiel $m = 365$, $h(k) =$„Geburtstag von k“. Bei mehr als 23 Menschen ist es wahrscheinlicher, dass zwei am selben Tag Geburtstag haben, als umgekehrt.
- Kollisionen sind häufiger als man denkt.

Kollisionsauflösung durch Verkettung

Um Kollisionen zu begegnen, hält man in jeder Arrayposition eine verkettete Liste von Objekten.

- Suchen geschieht durch Suchen in der jeweiligen Liste,
- Einfügen geschieht durch Anhängen an die jeweilige Liste,
- Löschen geschieht durch Entfernen aus der jeweiligen Liste.

SEARCH(A, k)

1 Suche in der Liste $A[h(k)]$ nach Element mit Schlüssel k

INSERT(A, x)

1 Hänge x am Anfang der Liste $A[h(k)]$ ein.

DELETE(A, k)

1 Entferne das Objekt mit Schlüssel k aus der Liste $A[h(k)]$.

Leider ist die Laufzeit jetzt nicht mehr $\Theta(1)$.

Lastfaktor

Die Hashtabelle habe m Plätze und enthalte n Einträge.

Der Quotient $\alpha := n/m$ heißt Lastfaktor.

Beachte: $\alpha > 1$ ist möglich.

Der Lastfaktor heißt auch Belegungsfaktor

Eine Hashtabelle heißt auch Streuspeichertabelle.

Analyse von Hashing mit Verkettung

Die Hashwerte seien wiederum uniform verteilt.

Dann werden die Listen im Mittel Länge α besitzen.

Die Operationen SEARCH, INSERT, DELETE haben also jeweils erwartete (=mittlere) Laufzeit

$$T \leq c(1 + \alpha)$$

für geeignete Konstante $c > 0$.

Der Summand „1“ bezeichnet den Aufwand für das Berechnen der Hashfunktion und die Indizierung.

Der Summand α bezeichnet die lineare Laufzeit des Durchsuchens einer verketteten Liste.

Hashfunktionen

Seien die einzutragenden Objekte x irgendwie zufällig verteilt.

Die Hashfunktion sollte so beschaffen sein, dass die Zufallsvariable $h(key[x])$ uniform verteilt ist (da ja sonst manche Fächer leer bleiben, während andere überfüllt sind.)

Sind z.B. die Schlüssel in $\{0, \ldots, N-1\}$ uniform verteilt, so ist $h(k) = k \bmod m$ eine gute Hashfunktion.

Sind z.B. die Schlüssel in $[0, 1[$ uniform verteilt, so ist $h(k) = \lfloor mk \rfloor$ eine gute Hashfunktion.

m wie immer die Größe der Hashtabelle.

Die Schlüssel sind meist nicht uniform verteilt:

Bezeichner in einer Programmiersprache: `count`, `i`, `max_zahl` häufiger als `zu6fgp98qq`. Wenn `kli` dann oft auch `kli1`, `kli2`, etc.

Nichtnumerische Schlüssel

. . . müssen vor Anwendung einer „Lehrbuch-Hashfunktion“ zunächst in Zahlen konvertiert werden.

Zeichenketten etwa unter Verwendung der Basis 256:

`'p'` $= 112$, `'q'` $= 116$, also `"pq"` $= 112 \cdot 256 + 116 = 28788$.

Divisionsmethode

Wie immer: Schlüssel: $0 \ldots N-1$, Hashwerte: $0 \ldots m-1$.

Hashfunktion: $h(k) = k \bmod m$.

- m sollte keine Zweierpotenz sein, da sonst $h(k)$ nicht von allen Bits (von k) abhängt.
- Ist k eine Kodierung eines Strings im 256er System, so bildet h bei $m = 2^p - 1$ zwei Strings, die sich nur durch eine Transposition unterscheiden, auf denselben Wert ab.
- Eine gute Wahl für m ist eine Primzahl die nicht nahe bei einer Zweierpotenz liegt. Z.B. $n = 2000$, vorauss. Lastfaktor $\alpha = 3$: Tabellengröße $m = 701$ bietet sich an.
- Bei professionellen Anwendungen empfiehlt sich ein Test mit „realen Daten“.

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m(kA \bmod 1) \rfloor$ für $A \in]0,1[$.

Hier $x \bmod 1$ =„gebrochener Teil von x“, z.B.: $\pi \bmod 1 = 0,14159\ldots$.

Rationale Zahlen A mit kleinem Nenner führen zu Ungleichverteilungen, daher empfiehlt sich die Wahl $A = (\sqrt{5}-1)/2$ („Goldener Schnitt“)[a].

Vorteile der Multiplikationsmethode:

- Arithmetische Progressionen von Schlüsseln $k = k_0, k_0 + d, k_0 + 2d, k_0 + 3d, \ldots$ werden ebenmäßig verstreut.
- Leicht zu implementieren wenn $m = 2^p$ (hier unproblematisch) und $N < 2^w$, wobei w die Wortlänge ist: Multipliziere k mit $\lfloor A \cdot 2^w \rfloor$. Dies ergibt zwei w-bit Wörter. Vom Niederwertigen der beiden Wörter bilden die p höchstwertigen Bits den Hashwert $h(k)$.

[a] Diese Zahl ist die “irrationalste” überhaupt, da der Nenner als Funktion der Approximationsgüte bei dieser Zahl am stärksten wächst.

Weiterführendes

- Universelles Hashing: Zufällige Wahl der Hashfunktion bei Initialisierung der Tabelle, dadurch Vermeidung systematischer Kollisionen, z.B. Provokation schlechter Laufzeit durch bösartig konstruierte Benchmarks.
- Gute Hashfunktionen können zur Authentizierung verwendet werden, z.B., MD5 *message digest*.

Offene Adressierung

Man kann auf verkettete Listen verzichten, indem man bei Auftreten einer Kollision eine andere Arrayposition benutzt.

Dazu braucht man eine zweistellige Hashfunktion
$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$.

```
INSERT(T, x)
1  i ← 0
2  while i ≤ m and h(key[x], i) ≠ NIL do
3        i ← i + 1
4  if i ≤ m
5        then T[h(key[x], i)] = x
6        else error "hash table overflow"
```

Für jeden Schlüssel k sollte die Sondierungsfolge (*probe sequence*)

$$h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)$$

eine Permutation von $0, 1, 2, \ldots, m-1$ sein, damit jede Position irgendwann sondiert wird.

Offene Adressierung

```
SEARCH(T, k)
1  i ← 0
2  repeat
3        j ← h(k, i); i ← i + 1
4  until i = m or T[j] = NIL or key[T[j]] = k
5  if i < m and key[T[j]] = k
6        then return T[j]
7        else return NIL
```

NB: Tabelleneinträge sind Objekte zuzüglich des speziellen Wertes NIL.

Z.B. Zeiger auf Objekte oder Nullzeiger.

Einschränkung: Bei offener Adressierung ist Löschen etwas umständlich (etwa durch explizites Markieren von Einträgen als "gelöscht").

Hashfunktionen für offene Adressierung

- Lineares Sondieren (*linear probing*):

  $$h(k,i) = (h'(k) + i) \bmod m$$

  Problem: Lange zusammenhängende Blöcke besetzter Plätze entstehen (*primary clustering*), dadurch oft lange Sondierdauer.

- Quadratisches Sondieren (*quadratic probing*):

  $$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

  für passende c_1, c_2 sodass $h(k,\cdot)$ Permutation ist.

Quadratisches Sondieren ist wesentlich besser als lineares Sondieren, aber beide Verfahren haben folgenden Nachteil:

Wenn $h(k_1,0) = h(k_2,0)$, dann $h(k_1,i) = h(k_2,i)$, d.h., kollidierende Schlüssel haben dieselbe Sondierungsfolge.

Insgesamt gibt es nur m verschiedene Sondierungsfolgen (von $m!$ Möglichen!); das führt auch zu Clusterbildung (*secondary clustering*).

Hashfunktionen für offene Adressierung

- Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

Jede Sondierungsfolge ist eine arithmetische Progression, Startwert und Schrittweite sind durch Hashfunktionen bestimmt.

Damit alle Positionen sondiert werden, muss natürlich $\mathrm{ggT}(h_2(k), m) = 1$ sein. Z.B. m Zweierpotenz und $h_2(k)$ immer ungerade.

Es gibt dann $\Theta(m^2)$ Sondierungsfolgen.

Analyse der Offenen Adressierung

Vereinfachende Annahme: Die Schlüssel seien so verteilt, dass jede der $m!$ Sondierungsfolgen gleichwahrscheinlich ist.

Diese Annahme wird durch *double hashing* approximiert aber nicht erreicht.

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartende Dauer einer erfolglosen Suche beschränkt durch $1/(1-\alpha)$.

Beispiel: Lastfaktor $\alpha = 0,9$ (Tabelle zu neunzig Prozent gefüllt): Eine erfolglose Suche erfordert im Mittel weniger als 10 Versuche (unabhängig von m, n).

Bemerkung: Dies ist auch die erwartete Laufzeit für eine Insertion.

Beweis des Satzes

Sei X eine Zufallsvariable mit Werten aus $\mathbb{N}$.

Dann ist

$$E[X] := \sum_{i=0}^{\infty} i\Pr\{X = i\} = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

Dies deshalb, weil $\Pr\{X \geq i\} = \sum_{j=i}^{\infty} \Pr\{X = j\}$.

Daher ergibt sich für die erwartete Suchdauer D:

$$D = \sum_{i=1}^{\infty} \Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\}$$

$$\Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \alpha^i$$

Also, $D \leq \sum_{i=1}^{\infty} \alpha^i = 1/(1-\alpha)$.

Analyse der Offenen Adressierung

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartetende Dauer einer erfolgreichen Suche beschränkt durch $(1 - \ln(1 - \alpha))/\alpha$.

Beispiel: Lastfaktor $\alpha = 0,9$: Eine erfolgreiche Suche erfordert im Mittel weniger als 3,67 Versuche (unabhängig von m, n).

Lastfaktor $\alpha = 0,5$: mittlere Suchdauer $\leq 3,39$.

Achtung: All das gilt natürlich nur unter der idealisierenden Annahme von *uniform hashing*.

Beweis

Die beim Aufsuchen des Schlüssels durchlaufene Sondierungsfolge ist dieselbe wie die beim Einfügen durchlaufene.

Die Länge dieser Folge für den als $i+1$-ter eingefügten Schlüssel ist im Mittel beschränkt durch $1/(1-i/m) = m/(m-i)$. (Wg. vorherigen Satzes!)

Gemittelt über alle Schlüssel, die eingefügt wurden, erhält man also

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$$

als obere Schranke für den Erwartungswert der Suchdauer.

Hier ist $H_k = \sum_{i=1}^{k} 1/i$ die k-te harmonische Zahl. Es gilt mit Integralabschätzung $\ln k \leq H_k \leq 1 + \ln k$.

Daraus ergibt sich der Satz.

Zusammenfassung

- Hashing = Speichern von Objekten an Arraypositionen, die aus ihrem Schlüssel berechnet werden.
- Die Zahl der Arraypositionen ist i.a. wesentlich kleiner als die der Schlüssel.
- Kollisionsauflösung durch Verkettung: Jede Arrayposition enthält eine verkettete Liste.
- Offene Adressierung: Bei Kollision wird eine andere Arrayposition sondiert.
- Hashfunktionen für einfaches Hashing: Multiplikationsmethode, Divisionsmethode.
- Hashfunktionen für offene Adressierung: Lineares, quadratisches Sondieren. Double Hashing.
- Analyse unter Uniformitätsannahme, Komplexität jeweils als Funktion des Lastfaktors α (Auslastungsgrad):
  - Suchen und Einfügen bei einfachem Hashing: $c(1+\alpha)$
  - Einfügen und erfolgloses Suchen bei offener Adressierung: $\leq 1/(1-\alpha)$ Versuche
  - Erfolgreiches Suchen bei offener Adressierung: $\leq (1-\ln(1-\alpha))/\alpha$ Versuche

IV. Allgemeine Entwurfs- und Optimieru

- Greedy-Algorithmen: die Lösung wird Schritt für Schritt verbessert, zu jedem Zeitpunkt wird die lokal optimale Verbesserung durchgeführt.
- Dynamische Programmierung: die optimale Lösung wird aus rekursiv gewonnenen optimalen Lösungen für Teilprobleme berechnet. Der Zeitaufwand bei der Rekursion wird dadurch verbessert, dass Ergebnisse für die Teillösungen tabelliert werden und ggf. aus der Tabelle genommen werden, statt neu berechnet zu werden.
- Amortisierte Analyse: Erlaubt es, den Aufwand für eine ganze Folge von Operationen besser abzuschätzen, als durch einfaches Aufsummieren. Sinnvoll, wenn es teure Operationen gibt, die sich durch eine "Verbesserung" der Datenstruktur "amortisieren". Anwendung: *union find* Datenstruktur.
- Backtracking: Man exploriert systematisch (Tiefensuche) alle Möglichkeiten, verwirft aber einen Teilbaum, sofern festgestellt werden kann, dass dieser keine (optimale) Lösung enthalten kann.

Greedy-Algorithmen

Entwurfsprinzip für Algorithmen, speziell für Optimierungsprobleme.

- Grundsätzlich anwendbar, wenn Lösung eines Problems sich als Folge von Einzelentscheidungen / partiellen Lösungen ergibt.

  Beispiel: Finden einer Fahrtroute: Einzelentscheidungen = jeweils nächstes Ziel.

- Lösung L wird als Folge von partiellen Lösungen $L_0, L_1, L_2, \ldots, L_n = L$ konstruiert. Die partielle Lösung L_{i+1} ergibt sich aus L_i, indem unter allen Ein-Schritt-Erweiterungen von L_i diejenige gewählt wird, die den Nutzen oder Gewinn maximiert. Als Anfang L_0 nimmt man meist die leere partielle Lösung.

- Ist die optimale Lösung nur auf dem Umweg über schlechte partielle Lösungen erreichbar, so verfehlt diese Heuristik das Ziel

- Ein Greedy Algorithmus liegt vor, wenn diese lokal optimierende greedy Strategie tatsächlich zur beweisbar besten Lösung führt.

Einfaches Beispiel: ein Auswahlproblem

Gegeben: Menge $A = \{(s_1, e_1), \ldots, (s_n, e_n)\}$ mit $s_i \leq e_i \in \mathbb{R}^+$ für alle i.

Ziel: Finde $B \subseteq \{1, \ldots, n\}$ maximaler Größe mit $s_i \geq e_j$ oder $s_j \geq e_i$ für alle $i \neq j \in B$.

Annahme: A nach den e_i sortiert: $e_1 \leq e_2 \leq \cdots \leq e_n$.

Greedy-Algorithmus:

$B \leftarrow \emptyset$
$m \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  if $s_i \geq m$
    then $B \leftarrow B \cup \{i\}$
      $m \leftarrow e_i$
return B

Satz: Der Greedy-Algorithmus findet hier stets eine optimale Lösung für das Auswahlproblem.

Korrektheitsbeweis

Dass solch ein Greedy Algorithmus korrekt ist, kann man (wenn es überhaupt der Fall ist!) mit der folgenden Strategie zu beweisen versuchen.

- Versuche als Invariante zu zeigen: L_i lässt sich zu einer optimalen Lösung erweitern. Dies gilt sicher am Anfang wo noch keine Entscheidungen getroffen worden sind. Wichtig ist also, dies beim Übergang von L_i auf L_{i+1} zu erhalten. Dabei hilft das folgende Austauschprinzip.
- Sei L_i eine partielle Lösung und L eine sie erweiternde optimale Lösung. Sei L_{i+1} diejenige partielle Teillösung, die vom Greedy Algorithmus ausgewählt wird. Dann lässt sich L zu einer optimalen Lösung L' umbauen, die dann auch L_{i+1} erweitert.

Anwendung im Beispiel

Die triviale Lösung $B = \emptyset$ lässt sich zu einer optimalen Lösung erweitern (nur kennen wir sie nicht...).

Sei B_i zu optimaler Lösung B erweiterbar. Nehmen wir an, diese optimale Lösung enthalte nicht das vom Greedy Algorithmus gewählte Intervall (s_j, e_j). Sei (s, e) das Intervall in $B \setminus B_i$ mit dem kleinsten Endzeitpunkt e. Ersetzen wir es durch die vom Greedy Algorithmus getroffene Wahl, so liegt immer noch eine optimale Lösung vor.

Klassische Anwendung: Steinitzscher Austauschsatz

Satz: Sei $v_1, \ldots, v_k$ eine Menge linear unabhängiger Vektoren und B eine Basis, die $\{v_1, \ldots, v_k\}$ erweitert. Sei $v_1, \ldots, v_k, u\}$ linear unabhängig. Dann existiert ein Vektor $u' \in B \setminus \{v_1, \ldots, v_k\}$ sodass $B \setminus \{u'\} \cup \{u\}$ auch eine Basis ist.

Beweis: Sei $B = \{v_1, \ldots, v_k, w_1, \ldots, w_l\}$ Repräsentiere u in der Basis B als $u = \lambda_1 v_1 + \cdots + \lambda_k v_k + \mu_1 w_1 + \cdots + \mu_l w_l$. Die μ_i können nicht alle Null sein wegen der Annahme an u. O.B.d.A. sei $\mu_1 \neq 0$. Dann ist $\{v_1, \ldots, v_k, u, w_2, \ldots, w_l\}$ auch eine Basis.

Daher findet folgender Greedyalgorithmus immer eine Basis: Beginne mit beliebigem Vektor $v_1 \neq 0$. Sind $v_1, \ldots, v_k$ schon gefunden, so wähle für v_{k+1} irgendeinen Vektor, der von $v_1, \ldots, v_k$ linear unabhängig ist. Gibt es keinen solchen mehr, so liegt Basis vor.

Huffman-Codes

Definition:

Für Wörter $v, w \in \{0,1\}^*$ heißt v ein Präfix von w, falls $w = vu$ für ein $u \in \{0,1\}^*$.

Ein Präfixcode ist eine Menge von Wörtern $C \subseteq \{0,1\}^*$ mit: für alle $v, w \in C$ ist v nicht Präfix von w.

Das Huffman-Code-Problem:

Gegeben: Alphabet $A = \{a_1, \ldots, a_n\}$ mit Häufigkeiten $h(a_i) \in \mathbb{R}_+$

Ziel: Finde Präfixcode $C = \{c_1, \ldots, c_n\}$ derart dass $\sum_{i=1}^{n} h(a_i) \cdot |c_i|$ minimal wird.

Ein solches C heißt optimaler Präfixcode, oder Huffman-Code; diese werden bei der Datenkompression verwendet.

Der Huffman-Algorithmus

- Erzeuge Knoten $z_1, \ldots, z_n$ mit Feldern $B[z_i] \leftarrow a_i$ und $H[z_i] \leftarrow h(a_i)$.
- Speichere diese in einer Priority Queue Q mit Operationen INSERT und EXTRACT-MIN (z.B. reaslisiert als Heap).
- Baue sukzessive einen binären Baum auf: Blätter sind die Knoten z_i, innere Knoten x haben nur ein Feld $H[x]$.
- Codewort c_i entspricht Pfad im Baum zu z_i, wobei $\mathit{left} \mathrel{\hat{=}} 0$ und $\mathit{right} \mathrel{\hat{=}} 1$.
- Aufbau des Baumes nach dem folgenden Algorithmus:

  for $i \leftarrow 1$ **to** $n-1$ **do**
  erzeuge neuen Knoten x
  $\mathit{left}[x] \leftarrow$ EXTRACT-MIN(Q)
  $\mathit{right}[x] \leftarrow$ EXTRACT-MIN(Q)
  $H[x] \leftarrow H[\mathit{left}[x]] + H[\mathit{right}[x]]$
  INSERT(Q, x)
  return EXTRACT-MIN(Q)

- Laufzeit: $O(n) + 3(n-1) \cdot O(\log n) = O(n \log n)$.

Korrektheit des Huffman-Algorithmus

Satz: Der Huffman-Algorithmus findet stets einen optimalen Präfixcode.

Beweis: Identifiziere Präfixcodes mit Codebäumen.

Für Codebaum T ist $K(T) := \sum_{x \in A} h(x) d_T(x)$

Reduktion R: Durch Auswahl von a, b wird A reduziert zu $A' := A \setminus \{a, b\} \cup \{d\}$, mit $h(d) = h(a) + h(b)$.

Erweiterung E: Ersetze in Codebaum T für A' das Blatt z mit $B[z] = d$ durch Knoten x mit $B[\mathit{left}[x]] = a$ und $B[\mathit{right}[x]] = b$.

- Greedy Choice-Prinzip:

  Seien $a, b \in A$ mit $h(a), h(b) \leq h(c)$ für alle $c \in A \setminus \{a, b\}$.
  Dann gibt es einen optimalen Codebaum, in dem a und b in benachbarten Blättern maximaler Tiefe liegen.

- Reflektion der Kostenfunktion:

  Sind T und T' Codebäume für A', und ist $K(E(T)) < K(E(T'))$, so auch $K(T) < K(T')$.

Dynamische Programmierung

Entwurfsprinzip, hauptsächlich für Algorithmen für Optimierungsprobleme:

- Zerlegung des Problems in kleinere, gleichartige Teilprobleme.
- Lösung ist zusammengesetzt aus den Lösungen der Teilprobleme.
- Aber: Teilprobleme sind nicht unabhängig, sondern haben ihrerseits gemeinsame Teilprobleme.

$\rightsquigarrow$ rekursives Divide-and-Conquer-Verfahren würde die gleichen Teilprobleme immer wieder lösen.

Lösung: Rekursion mit Memoisierung, oder besser:

Dynamische Programmierung:

Bottom-Up-Berechnung der Lösungen größerer Probleme aus denen kleinerer, die dabei in einer Tabelle gespeichert werden.

Einfaches Beispiel: Matrizen-Kettenmultiplikation

Problem: n Matrizen $M_1, \ldots, M_n$ sollen multipliziert werden, wobei M_i eine $n_i \times n_{i+1}$-Matrix ist. Aufwand hängt von der Klammerung ab.

Bezeichne $M_{i..j}$ das Produkt $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$, und $m[i,j]$ die minimale Zahl von Multiplikationen zum Berechnen von $M_{i..j}$.

1. Optimale Klammerung von $M_{1..n}$ ist von der Form $M_{1..k} \cdot M_{k+1..n}$, für optimale Klammerungen von $M_{1..k}$ und $M_{k+1..n}$.
2. Deshalb gilt:

$$m[i,j] = \begin{cases} 0 & \text{falls } i = j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1} & \text{sonst.} \end{cases} \qquad (*)$$

3. Fülle eine Tabelle, in der unten die $m[i,i] = 0$ stehen, und der h-ten Ebene die Werte $m[i,j]$ mit $j - i = h$. An der Spitze steht schließlich $m[1,n]$.
4. Speichere in jedem Schritt auch einen Wert $s[i,j]$, nämlich dasjenige k, für das in $(*)$ das Minimum angenommen wird.

Längste gemeinsame Teilfolge (lgT)

Definition: $Z = \langle z_1, z_2, \ldots, z_m \rangle$ ist Teilfolge von $X = \langle x_1, x_2, \ldots, x_n \rangle$,
falls es Indizes $i_1 < i_2 < \ldots < i_m$ gibt mit $z_j = x_{i_j}$ für $j \leq m$.

Problem: Gegeben $X = \langle x_1, x_2, \ldots, x_m \rangle$ und $Y = \langle y_1, y_2, \ldots, y_n \rangle$

Ziel: finde $Z = \langle z_1, z_2, \ldots, z_k \rangle$ maximaler Länge k, das Teilfolge von X und Y ist.

Abkürzung: $X_i := \langle x_1, x_2, \ldots, x_i \rangle$ ist das i-te Präfix von X.

Lösung mit dynamischer Programmierung:

1. Sei Z lgT von X und Y. Dann gilt:
   (a) Ist $x_m = y_n$, so ist $z_k = x_m = y_n$, und Z_{k-1} ist lgT von X_{m-1} und Y_{n-1}.
   (b) Ist $x_m \neq y_n$, so gilt:
   - Ist $z_k \neq x_m$, dann ist Z lgT von X_{m-1} und Y.
   - Ist $z_k \neq y_n$, dann ist Z lgT von X und Y_{n-1}.

Längste gemeinsame Teilfolge (lgT)

2. Sei $c[i,j]$ die Länge einer lgT von X_i und Y_j. Es gilt:

$$c[i,j] = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0, \\ c[i-1,j-1]+1 & \text{falls } i,j > 0 \text{ und } x_i = y_j \\ \max(c[i-1,j], c[i,j-1]) & \text{falls } i,j > 0 \text{ und } x_i \neq y_j \end{cases}$$

3. Fülle die Tabelle der $c[i,j]$ zeilenweise, jede Zeile von links nach rechts. Am Ende erhält man $c[m,n]$, die Länge einer lgT von X und Y.

4. Für $i,j \geq 1$, speichere einen Wert $b[i,j] \in \{\uparrow, \nwarrow, \leftarrow\}$, der anzeigt, wie $c[i,j]$ berechnet wurde:

$$\begin{aligned} \uparrow &: c[i,j] = c[i-1,j] \\ \nwarrow &: c[i,j] = c[i-1,j-1]+1 \\ \leftarrow &: c[i,j] = c[i,j-1] \end{aligned}$$

   Damit läßt sich eine lgT aus der Tabelle ablesen.

Zusammenfassung Dynamische Programmierung

- Ausgangspunkt ist immer eine rekursive Lösung des gegebenen Problems.
- Dynamische Programmierung bietet sich an, wenn die Zahl der rekursiven Aufrufe bei naiver Implementierung größer ist, als die Zahl der überhaupt möglichen voneinander verschiedenen Aufrufe der rekurive definierten Funktion.
- Man studiert dann die Reihenfolge, in der die Ergebnisse der Aufrufe benötigt werden und berechnet diese iterativ von unten her.

Amortisierungs-Analyse

Prinzip zur Analyse von Operationen auf Datenstrukturen.

- Betrachte nicht einzelne Operationen, sondern Folgen von Operationen.
- Statt der (*worst-case*-) Kosten jeder einzelnen Operation werden die mittleren Kosten der Operationen in der Folge bestimmt.

  $\rightsquigarrow$ teure Operationen können sich durch nachfolgende billigere amortisieren.
- Kosten einiger Operationen werden zu hoch, andere zu niedrig veranschlagt.
- Drei Methoden:
  1. Aggregat-Methode:
     Berechne worst-case Kosten $T(n)$ einer Folge von n Operationen.
     Amortisierte Kosten: $T(n)/n$.
  2. Konto-Methode:
     Elemente der Datenstruktur haben ein Konto, auf dem zuviel berechnete Kosten gutgeschrieben werden. Zu niedrig veranschlagte Operationen werden aus dem Konto bezahlt.
  3. Potenzial-Methode:
     Ähnlich der Konto-Methode, aber gesamte zuviel berechnete Kosten werden als *potenzielle Energie* der Datenstruktur aufgefasst.

Einfaches Beispiel: Binärzähler

Datenstruktur: Array A von $length[A] = k$ Bits.

Einzige Operation:

INCREMENT(A)

```
1  i ← 0
2  while i < length[A]  und  A[i] = 1
3        do A[i] ← 0
4           i ← i + 1
5  if i < length[A]
6    then A[i] ← 1
```

Worst-case-Kosten eines INCREMENT: $\lceil \log_2(n+1) \rceil$ Wertzuweisungen.

Aggregat-Methode

Betrachte Folge von n INCREMENT-Operationen (Annahme: $k \geq \lceil \log_2(n+1) \rceil$).

Berechne Kosten $T(n)$ der ganzen Folge von n INCREMENT-Operationen:

- Beobachtung:

  $A[0]$ wird bei jedem INCREMENT verändert,
  $A[1]$ nur bei jedem zweiten,
  $A[2]$ nur bei jedem vierten,
  etc.

- Gesamtzahl der Wertzuweisungen bei n INCREMENT-Operationen:

$$T(n) \leq \sum_{i=0}^{\lceil \log_2(n+1) \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2n$$

- Also: $T(n) = O(n)$, somit amortisierte Kosten jedes INCREMENT:

$$O(n)/n = O(1)$$

Konto-Methode

Für die i-te Operation:

tatsächliche Kosten c_i amortisierte Kosten $\hat{c}_i$.

- Elemente der Datenstruktur haben Konten.
- Ist $\hat{c}_i > c_i$, so wird die Differenz $\hat{c}_i - c_i$ auf ein Konto gutgeschrieben.
- Ist $\hat{c}_i < c_i$, so muss die Differenz $c_i - \hat{c}_i$ aus einem Konto bezahlt werden.

  $\rightsquigarrow$ $\hat{c}_i$ müssen so gewählt sein, dass genügend Guthaben auf den Konten ist.

Im Beispiel:

- Kosten einer Wertzuweisung seien 1 EUR.
- Jede Array-Position $j \leq k$ hat ein Konto.
- Veranschlage amortisierte Kosten eines INCREMENT: 2 EUR.
- Jedes INCREMENT setzt nur ein Bit $A[i] \leftarrow 1$ (Zeile 6):
  Zahle dafür 1 EUR, und schreibe 1EUR auf dessen Konto gut.

  $\rightsquigarrow$ Jedes j mit $A[j] = 1$ hat ein Guthaben von 1 EUR.
  Zahle damit die Wertzuweisungen $A[i] \leftarrow 0$ in Zeile 3.

Potenzial-Methode

Sei D_i die Datenstruktur nach der i-ten Operation.

- Definiere eine Potenzialfunktion Φ, die jedem D_i ein $\Phi(D_i) \in \mathbb{R}$ zuordnet.
- Amortisierte Kosten $\hat{c}_i$ definiert durch $\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$.
- Gesamte amortisierte Kosten:

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n} \big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big) \;=\; \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- Ist $\Phi(D_n) \geq \Phi(D_0)$, so sind tatsächliche Gesamtkosten höchstens die amortisierten Gesamtkosten.
- Meist ist n variabel, und man setzt $\Phi(D_0) = 0$, und $\Phi(D_i) \geq 0$ für $i > 0$.

Potenzial-Methode

Im Beispiel:

- Definiere $\Phi(D_i) = b_i :=$ Anzahl der 1 im Zähler nach i INCREMENT-Operationen .
- Damit ist $\Phi(D_0) = 0$, und $\Phi(D_i) > 0$ für alle $i > 0$.
- Sei t_i die Zahl der Durchläufe der **while**-Schleife beim i-ten INCREMENT.
- Damit ist $c_i = t_i + 1$: Es werden t_i bits auf 0 gesetzt, und eines auf 1.
- Ausserdem gilt: $b_i = b_{i-1} - t_i + 1$.
- Daher ist die Potenzialdifferenz:

$$\Phi(D_i) - \Phi(D_{i-1}) \;=\; b_i - b_{i-1} \;=\; (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i \;,$$

  also sind die amortisierten Kosten:

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; t_i + 1 + (1 - t_i) \;=\; 2$$

Queue als zwei stacks

Eine Queue Q kann mittels zweier Keller S_{in} und S_{out} realisiert werden:

PUT(Q, x)

  PUSH(S_{in}, x)

GET(Q)

  ifEMPTY(S_{out})
  then **while** nicht EMPTY(S_{in}) **do**
    PUSH$(S_{\text{out}}, \text{POP}(S_{\text{in}}))$
  returnPOP(S_{out})

Wieviele Kelleroperationen benötigen die Queue-Operationen im worst-case?

PUT(Q, x): 1
GET(Q): $2n + 1$

Amortisierungsanalyse der Queue

Definiere Potenzial $\Phi(Q) := 2 \cdot |S_{\text{in}}|$.

Sei Q' die Queue nach Anwendung der Operation auf Q.

PUT(Q, x): es ist $\Phi(Q') = \Phi(Q) + 2$.

$$c_{\text{PUT}} = 1 \qquad \hat{c}_{\text{PUT}} = 1 + \Phi(Q') - \Phi(Q) = 1 + 2 = 3$$

GET(Q), Fall 1: S_{out} nicht leer. Dann ist $\Phi(Q') = \Phi(Q)$.

$$c_{\text{GET}} = 1 \qquad \hat{c}_{\text{GET}} = 1 + \Phi(Q') - \Phi(Q) = 1$$

GET(Q), Fall 2: S_{out} leer. Dann ist $\Phi(Q') = 0$ und $\Phi(Q) = 2 \cdot |S_{\text{in}}| = 2n$.

$$c_{\text{GET}} = 2n + 1 \qquad \hat{c}_{\text{GET}} = 1 + \Phi(Q') - \Phi(Q) = 2n + 1 + 0 - 2n = 2$$

Union-Find: Datenstruktur für disjunkte Mengen

Es soll eine Familie von disjunkten dynamischen Mengen

$$M_1, \ldots, M_k \quad \text{mit} \quad M_i \cap M_j = \emptyset \text{ für } i \neq j$$

verwaltet werden.

Dabei sollen folgende Operationen zur Verfügung stehen:

- MAKE-SET(x) fügt die Menge $\{x\}$ zur Familie hinzu. Es ist Aufgabe des Benutzers, sicherzustellen dass x vorher in keiner der M_i vorkommt.
- FIND-SET(x) liefert ein kanonisches Element der Menge M_i mit $x \in M_i$, also FIND-SET(x) = FIND-SET(y), falls x und y in der gleichen Menge M_i liegen.
- UNION(x, y) ersetzt die Mengen M_i und M_j mit $x \in M_i$, $y \in M_j$ durch ihre Vereinigung $M_i \cup M_j$.

Typische Anwendungen: Äquivalenzklassen einer Äquivalenzrelation, Zusammenhangskomponenten eines Graphen , ...

Union-Find: Realisierung als Wälder

Jede Menge wird durch einen Baum dargestellt. Knoten x haben Zeiger $p[x]$ auf ihren Vater. Kanonische Elemente sind an der Wurzel und haben $p[x] = x$.

Optimierungen:

- Rangfunktion: Jedes Element hat Rang $\approx \log\Big(\big|\text{Teilbaum ab } x\big|\Big)$
- Pfadkompression: Hänge jeden besuchten Knoten direkt an die Wurzel.

MAKE-SET(x)
  $p[x] \leftarrow x$
  $rank[x] \leftarrow 0$

FIND-SET(x)
  if $x \neq p[x]$
    then $p[x] \leftarrow$ FIND-SET($p[x]$)
  return $p[x]$

UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  if $rank[x] > rank[y]$
    then $p[y] \leftarrow x$
    else $p[x] \leftarrow y$
      if $rank[x] = rank[y]$
        then $rank[y] \leftarrow rank[y] + 1$

Analyse der *Union-Find*-Wälder

Sei $size(x)$ die Größe des Teilbaumes unter x.

Lemma: Für jede Wurzel x gilt: $size(x) \geq 2^{rank[x]}$

Lemma: Für jedes $r \geq 1$ gibt es höchstens $n/2^r$ Knoten x mit $rank[x] = r$.

Korollar: Jeder Knoten hat $rank[x] \leq \log_2 n$.

Lemma: Für jede Wurzel x ist die Höhe $h(x)$ des Baumes unter x höchstens $h(x) \leq rank[x]$.

Satz: Jede Folge von m MAKE-SET-, UNION- und FIND-SET-Operationen, von denen n MAKE-SET sind, hat für Wälder mit Rangfunktion eine Laufzeit von $O(m \log n)$.

Komplexitätsverbesserung durch Pfadkompression

Definition: Der iterierte Logarithmus $\log^* n$ ist gegeben durch:

$$\log^{(0)} n := n$$

$$\log^{(i+1)} n := \begin{cases} \log \log^{(i)} n & \text{falls } \log^{(i)} n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

$$\log^* n := \min\{i \,;\, \log^{(i)} n \leq 1\}$$

Satz: Jede Folge von m Make-Set-, Union- und Find-Set-Operationen, von denen n Make-Set sind, hat für Wälder mit Rangfunktion und Pfadkompression eine Laufzeit von $O(m \log^* n)$.

Bemerkung: Für $n \leq 10^{19.728}$ ist $\log^* n \leq 5$.

Backtracking

Mögliche Lösungen erscheinen als Blätter eines (immensen) Baumes.

Teillösungen entsprechen den inneren Knoten.

Der Baum wird gemäß der Tiefensuche durchgearbeitet.

Sieht man einer Teillösung bereits an, dass sie nicht zu einer Lösung vervollständigt werden kann, so wird der entsprechende Teilbaum nicht weiter verfolgt.

Damenproblem

Man soll n Damen auf einem $n \times n$ Schachbrett so platzieren, dass keine Dame eine andere schlägt.

Wir platzieren die Damen zeilenweise von unten her und repräsentieren Teillösungen als Listen.

Beispiel: `[0;1;6]` bedeutet:

Die unterste Zeile enthält eine Dame auf Position 6 (=siebtes Feld von links).

Die zweite Zeile von unten enthält eine Dame auf Position 1 (=zweites Feld von links).

Die dritte Zeile von unten enthält eine Dame auf Position 0 (=erstes Feld von links).

Solch eine Teillösung heißt konsistent, wenn keine Dame eine andere schlägt.

Testen auf Inkonsistenz

```
let inkonsistent i l =
  List.mem i l ||
  let rec test j = if j >= List.length l then false else
    List.nth l j  = i+j+1 || List.nth l j  = i-j-1 || test (j+1)
  in test 0
```

Ist `l` selbst konsistent, so ist `i::l` konsistent genau dann, wenn `inkonsistent i l = false`.

Lösung mit Backtracking

```
let rec finde_damen l =
  if List.length l = anzahl_damen then l else
  let rec probiere i =
    if i>=anzahl_damen then [] else
    if not (inkonsistent i l) then
    let result = finde_damen (i::l) in
         if result = [] then probiere (i+1)
         else result
    else probiere (i+1)
  in probiere 0
```

Ist `l` konsistent, so liefert `finde_damen l` eine Erweiterung von `l` auf `anzahl_damen` Damen, falls es eine gibt. Es wird `[]` zurückgegeben, falls es keine gibt.

Eine Lösung: `[3; 1; 6; 2; 5; 7; 4; 0]`

Iterative Deepening

Oft ist die Problemstellung so, dass eine Lösung mit steigender Tiefe im Lösungsbaum immer unwahrscheinlicher oder weniger brauchbar wird.

Im Prinzip würde es sich dann anbieten, die Tiefensuche durch eine Breitensuche zu ersetzen, was allerdings meist zu aufwendig ist (Speicherplatzbedarf steigt exponentiell mit der Tiefe).

Man kann dann der Reihe nach Tiefensuche bis maximale Tiefe 1, dann bis zur maximalen Tiefe 2, dann 3, dann 4, etc. durchführen.

Zwar macht man dann beim nächsten Tiefenschritt die gesamte bisher geleistete Arbeit nochmals; dennoch erzielt man so eine höhere Effizienz als bei der Breitensuche. Diese Technik heißt *iterative deepening*.

Branch and bound

Oft sind die Lösungen eines Baum-Suchproblems angeordnet gemäß ihrer Güte und man sucht entweder die beste Lösung, oder aber eine deren Güte eine bestimmte Schranke überschreitet.

Manchmal kann man die Güte von Lösungen schon anhand einer Teillösung abschätzen. Zeigt sich aufgrund dieser Abschätzung, dass keine Lösung in einem Teilbaum besser ist, als eine bereits gefundene, so braucht man diesen Teilbaum nicht weiter zu verfolgen.

Diese Verbesserung von Backtracking bezeichnet man als *Branch-and-Bound*.

Abstraktes Beispiel

Sei `f : bool list -> int` eine Funktion mit der Eigenschaft, dass `f(l)` $\geq$ `f(l@l')` und `f(l)` ≥ 0.

Man bestimme $\max\{$`f(l)` $: |$`l`$| = n\}$.

Folgendes Programm kann das tun:

```
let rec finde_optimum f n =
   let rec finde_erweiterung teilloesung bisherbester tiefe =
       if bisherbester >= f(teilloesung) then bisherbester else
       if tiefe = 0 then f(teilloesung) else
       let optimum_links = finde_erweiterung (teilloesung@[false])
                                  bisherbester (tiefe-1) in
       let optimum_rechts = finde_erweiterung (teilloesung@[true])
                                  optimum_links (tiefe-1) in
       optimum_rechts
   in finde_erweiterung [] 0 n
```

In der Praxis verwendet man B&B gern bei der linearen Optimierung mit Ganzzahligkeitsbedingungen (ILP).

V. Graphalgorithmen

- Grundlegendes
  - Repräsentation von Graphen
  - Breiten- und Tiefensuche
  - Anwendungen der Tiefensuche
- Minimale Spannbäume
  - Algorithmus von Prim
  - Algorithmus von Kruskal
- Kürzeste Wege
  - Algorithmus von Dijkstra
  - Bellman-Ford-Algorithmus
  - Floyd-Warshall-Algorithmus
- Flüsse in Netzwerken

Repräsentation von Graphen

Graph $G = (V, E)$, wobei $E \subseteq V \times V$ (gerichteter Graph)
bzw. $E \subseteq \binom{V}{2}$ (ungerichteter Graph).

Adjazenzlisten

Für jeden Knoten $v \in V$ werden in einer verketteten Liste $Adj[v]$
alle Nachbarn u mit $(v, u) \in E$ (bzw. mit $\{v, u\} \in E$) gespeichert.

Platzbedarf: $O(|V| + |E| \log |V|)$

Adjazenzmatrix

Es wird eine $|V| \times |V|$-Matrix $A = (a_{ij})$ gespeichert, mit

$$a_{ij} = 1 \quad \text{genau dann, wenn} \quad (v_i, v_j) \in E \; .$$

Platzbedarf: $O(|V|^2)$

Breitensuche

Gegeben: Graph G (als Adjazenzlisten), ausgezeichneter Startknoten $s \in V$.

Gesucht für jeden Knoten $v \in V$: kürzester Pfad zu s und Distanz $d[v]$.

Speichere für jedes $v \in V$ den Vorgänger $\pi[v]$ auf kürzestem Pfad zu s.

Initialisiere $\pi[v] = \text{NIL}$ und $d[v] = \infty$ für alle $v \in V$.

Setze $d[s] = 0$ und speichere s in einer *FIFO-queue* Q.

while $Q \neq \emptyset$
  do $v \leftarrow get(Q)$
    foreach $u \in Adj[v]$ with $d[u] = \infty$
      do $d[u] \leftarrow d[v] + 1$
        $\pi[u] \leftarrow v$
        $put(Q, u)$

Tiefensuche

Depth-First-Search (DFS):

> Sucht jeden Knoten einmal auf, sondert eine Teilmenge der Kanten aus, die einen Wald (den DFS-Wald) bilden.

Hilfsmittel Färbung:

- Weiß $\hat{=}$ noch nicht besucht.
- Grau $\hat{=}$ schon besucht, aber noch nicht abgefertigt
- Schwarz $\hat{=}$ abgefertigt, d.h. der gesamte von hier erreichbare Teil wurde durchsucht.

Speichert außerdem für jeden Knoten v:

- Zeitpunkt des ersten Besuchs $d[v]$ (*discovery time*)
- Zeitpunkt der Abfertigung $f[v]$ (*finishing time*)

Tiefensuche: Pseudocode

DFS(G)

initialisiere $color[v] \leftarrow white$ und $\pi[v] \leftarrow$ NIL für alle $v \in V$
$time \leftarrow 0$
foreach $v \in V$
  doif$color[v] = white$
    thenDFS-VISIT(G, v)

DFS-VISIT(G, v)

$color[v] \leftarrow grey$
$d[v] \leftarrow ++time$
foreach $u \in Adj[v]$ with $color[u] = white$
  do$\pi[u] \leftarrow v$
    DFS-VISIT(G, u)
$color[v] \leftarrow black$
$f[v] \leftarrow ++time$

Klammerungseigenschaft

Seien $u, v \in V$ und $u \neq v$. Die folgenden drei Fälle sind möglich:

- $d[u] < d[v] < f[v] < f[u]$ und v ist Nachfahre von u im DFS-Wald.
- $d[v] < d[u] < f[u] < f[v]$ und u ist Nachfahre von v im DFS-Wald.
- $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$ und weder ist u Nachfahre von v im DFS-Wald noch umgekehrt.

Insbesondere ist die Konstellation $d[u] < d[v] < f[u] < f[v]$ unmöglich und der DFS-Wald lässt sich aus den Aktivitätsintervallen $[d[v], f[v]]$ eindeutig rekonstruieren.

Klassifikation der Kanten

Durch DFS werden die Kanten eines Graphen in die folgenden vier Typen klassifiziert.

- Baumkanten sind die Kanten des DFS-Waldes, also (u, v) mit $\pi[v] = u$.

  Kennzeichen: Beim ersten Durchlaufen ist v weiß.

- Rückwärtskanten (u, v), wo v Vorfahre von u im DFS-Wald ist.

  Kennzeichen: Beim ersten Durchlaufen ist v grau.

- Vorwärtskanten (u, v), wo v Nachkomme von u im DFS-Wald ist.

  Kennzeichen: Beim ersten Durchlaufen ist v schwarz, und $d[u] < d[v]$.

- Querkanten sind alle übrigen Kanten (u, v).

  Kennzeichen: Beim ersten Durchlaufen ist v schwarz, und $d[u] > d[v]$.

Bei ungerichteten Graphen kommen nur Baum- und Rückwärtskanten vor.

Topologische Sortierung

Eine topologische Ordnung eines gerichteten, azyklischen Graphen (dag) ist eine lineare Ordnung der Knoten

$$v_1 \prec v_2 \prec \ldots \prec v_n$$

so dass für jede Kante $(u, v) \in E$ gilt $u \prec v$.

Lemma: Ein gerichteter Graph ist genau dann azyklisch, wenn bei DFS keine Rückwärtskanten entstehen.

Satz: Eine topologische Ordnung auf einem dag G erhält man durch absteigende Sortierung nach den *finishing times* $f[v]$ nach Ausführung von DFS(G).

Zusammenhang

Weg (oder Pfad) von v_0 nach v_k:

$$p = \langle v_0, \ldots, v_k \rangle \text{ mit } (v_i, v_{i+1}) \in E \text{ für alle } i < k \ .$$

Schreibweise: $p : v_0 \rightsquigarrow v_k$.

Für ungerichtete Graphen:

Zwei Knoten u und v heißen zusammenhängend, wenn es einen Weg $p : u \rightsquigarrow v$ gibt.

Für gerichtete Graphen:

Zwei Knoten u und v heißen stark zusammenhängend, wenn es Wege $p : u \rightsquigarrow v$ und $q : v \rightsquigarrow u$ gibt.

Die Äquivalenzklassen bzgl. dieser Äquivalenzrelation heißen (starke) Zusammenhangskomponenten.

Starke Zusammenhangskomponenten

Definition: der zu $G = (V, E)$ transponierte Graph ist $G^T := (V, E^T)$,
wobei $(u, v) \in E^T$ gdw. $(v, u) \in E$.

Folgender Algorithmus zerlegt G in seine starken Zusammenhangskomponenten:

- Zuerst wird DFS(G) aufgerufen.
- Sortiere die Knoten nach absteigender *finishing time*.
- Berechne G^T.
- Rufe DFS(G^T) auf, wobei die Knoten im Hauptprogramm in der Reihenfolge der obigen Sortierung behandelt werden.
- Starke Zusammenhangskomponenten von G sind die Bäume des im zweiten DFS berechneten DFS-Waldes.

Minimale Spannbäume

Gegeben: Zusammenhängender, ungerichteter Graph $G = (V, E)$,
Gewichtsfunktion $w : E \to \mathbb{R}$.

Gesucht: Minimaler Spannbaum $T \subseteq E$ mit:

- T ist azyklisch.
- T spannt den Graphen auf:

  je zwei Knoten $u, v \in V$ sind durch einen Pfad in T verbunden.
- Gewicht $\sum_{e \in T} w(e)$ ist minimal.

Definition: Sei $A \subseteq E$ Teilmenge eines Minimalen Spannbaumes.
Kante e heißt sicher für A, falls $A \cup \{e\}$ Teilmenge eines Minimalen Spannbaumes ist.

Vorgehensweise: Beginne mit $A = \emptyset$, füge dann sukzessive Kanten hinzu, die sicher für A sind.

Finden sicherer Kanten

Definition: Für $S \subseteq V$ und $e \in E$ sagen wir "e kreuzt S",
falls $e = \{u, v\}$ mit $u \in S$ und $v \in V \setminus S$.

Satz:

> Sei A Teilmenge eines minimalen Spannbaumes,
> sei $S \subseteq V$ mit der Eigenschaft: keine Kante in A kreuzt S,
> und sei e eine Kante minimalen Gewichtes, die S kreuzt.
>
> Dann ist e sicher für A.

Insbesondere: Sei Z eine Zusammenhangskomponente von A. Ist e eine Kante minimalen Gewichts, die Z mit einer anderen Zusammenhangskomponente verbindet, so ist e sicher für A.

Der Algorithmus von Prim

Benutzt eine Priority-Queue Q.

- Eingabe: G mit Gewichtsfunktion w und Anfangsknoten $r \in V$.
- Initialisiere Q mit allen $v \in V$, mit $key[v] = \infty$.
- Setze $key[r] \leftarrow 0$ und $\pi[r] \leftarrow$ NIL.
- Solange Q nicht leer ist, setze $v \leftarrow$ EXTRACT-MIN(Q):

  Für alle Nachbarn $u \in Adj[v]$, die noch in Q sind, und für die $w(\{u, v\}) < key[u]$ ist, setze $\pi[u] \leftarrow v$ und $key[u] \leftarrow w(\{u, v\})$.

Der Baum A ist implizit gegeben durch $\big\{ \{v, \pi[v]\} \,;\, v \in (V \setminus Q \setminus \{r\} \big\}$.

Komplexität mit Q als Heap:

- Initialisierung (BUILD-HEAP) $O(|V|)$
- $|V|$ Durchläufe der äußeren Schleife mit EXTRACT-MIN ($O(\log |V|)$).
- $|E|$ Durchläufe der inneren Schleife mit DECREASE-KEY ($O(\log |V|)$).

Also insgesamt: $O(|V| \log |V|) + O(|E| \log |V|) = O(|E| \log |V|)$.

Der Algorithmus von Kruskal

Benutzt eine UNION-FIND-Datenstruktur.

Erinnerung: Diese Datenstruktur verwaltet ein System disjunkter Mengen von "Objekten" und bietet folgende Operationen an:

- INIT Initialisieren
- $Make - Set(x)$ Fügt eine neue Einermenge mit Inhalt x hinzu. Ist x schon in einer vorhandenen Menge enthalten, so passiert nichts.
- $Find(x)$ Ist x in einer Menge enthalten, so liefere diese in Form eines kanonischen Elementes zurück. Anderenfalls liefere NIL o.ä. zurück. Insbesondere kann man durch den Test $Find(x) = Find(y)$ feststellen, ob zwei bereits eingefügte Elemente in derselben Menge liegen.
- $Union(x, y)$: Sind x und y in zwei verschiedenen Mengen enthalten, so vereinige diese zu einer einzigen. Anschließend gilt also insbesondere $Find(x) = Find(y)$.

Beachte: man kann Mengen und Elemente nicht wieder entfernen oder auseinanderreißen.

Der Algorithmus von Kruskal

Kruskal's Algorithmus

- Setze $A := \emptyset$.
- Rufe MAKE-SET(v) für jeden Knoten $v \in V$ auf.
- Sortiere die Kanten aufsteigend nach Gewicht.
- Für jede Kante $e = \{u, v\}$, in der sortierten Reihenfolge, prüfe ob FIND-SET(u) $\neq$ FIND-SET(v).
- Falls ja, füge e zu A hinzu, und rufe UNION(u, v) auf, sonst weiter.

Komplexität bei geschickter Implementierung der Union-Find Struktur: $O(|E| \log |E|)$.

Kürzeste Wege

Gegeben: gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$.

Für einen Weg $p : v_0 \rightsquigarrow v_k$

$$p = \langle v_0, \ldots, v_k \rangle \text{ mit } (v_i, v_{i+1}) \in E \text{ für alle } i < k \,.$$

sei das Gewicht des Weges p definiert als:

$$w(p) = \sum_{i=1}^{k} w\big((v_{i-1}, v_i)\big)$$

Minimaldistanz von u nach v:

$$\delta(u, v) = \begin{cases} \min\{w(p) \;;\; p : u \rightsquigarrow v\} & \text{falls } v \text{ von } u \text{ erreichbar ist,} \\ \infty & \text{sonst.} \end{cases}$$

Kürzester Weg von u nach v:

Pfad $p : u \rightsquigarrow v$ mit $w(p) = \delta(u, v)$.

Eigenschaften kürzester Wege

Problem: Gibt es einen negativen Zyklus $p : v \rightsquigarrow v$ mit $w(p) < 0$, so ist $\delta(u, u')$ nicht wohldefiniert, falls es einen Weg von u nach u' über v gibt.

Algorithmen für kürzeste Wege von einem Startpunkt s:

Dijkstra: nimmt an, dass $w(e) \geq 0$ für alle $e \in E$.

Bellman-Ford: Entdeckt die Präsenz negativer Zyklen, und liefert korrekte kürzeste Wege, falls es keinen gibt.

Optimale Teillösungen: Ist $p = \langle v_0, \ldots, v_k \rangle$ ein kürzester Weg von v_0 nach v_k, so ist für alle $0 \leq i < j \leq k$ der Pfad

$$p_{ij} = \langle v_i, \ldots, v_j \rangle$$

ein kürzester Weg von v_i nach v_j.

Daher reicht es zur Angabe eines kürzesten Weges von s zu v für alle $v \in V$, für jeden Knoten $v \neq s$ einen Vorgänger $\pi[v]$ anzugeben.

Relaxierung

Algorithmen halten für jedes $v \in V$ eine Abschätzung $d[v] \geq \delta(s,v)$ und einen vorläufigen Vorgänger $\pi[v]$.

INITIALISE(G,s) :

for $v \in V$ **do**

$d[v] \leftarrow \infty; \pi[v] \leftarrow$ NIL

$d[s] \leftarrow 0$

RELAX(u,v,w) : ▷testet, ob der bisher gefundene kürzeste Pfad zu v

▷ durch die Kante (u,v) verbessert werden kann

if $d[v] > d[u] + w\big((u,v)\big)$

then $d[v] \leftarrow d[u] + w\big((u,v)\big)$

$\pi[v] \leftarrow u$

Eigenschaften der Relaxierung

Lemma: Wird für einen Graphen G und $s \in V$ erst INITIALIZE(G, s), und dann eine beliebige Folge von RELAX(u, v, w) für Kanten (u, v) ausgeführt, so gelten die folgenden Invarianten:

1. $d[v] \geq \delta(s, v)$ für alle $v \in V$.
2. Ist irgendwann $d[v] = \delta(s, v)$, so ändert sich $d[v]$ nicht mehr.
3. Ist v nicht errreichbar von s, so ist $d[v] = \delta(s, v) = \infty$.
4. Gibt es einen kürzesten Pfad von s zu v, der in der Kante (u, v) endet, und ist $d[u] = \delta(s, u)$ vor dem Aufruf RELAX(u, v, w), so ist danach $d[v] = \delta(s, v)$.
5. Enthält G keinen negativen Zyklus, so ist der Teilgraph

   $$G_\pi = (V, E_\pi) \qquad \text{wobei} \quad E_\pi := \{(\pi[v], v), v \in V\}$$

   ein Baum mit Wurzel s.

Lemma: Gilt nach einer Folge von RELAX(u, v, w), dass $d[v] = \delta(s, v)$ für alle $v \in V$ ist, so enthält der Baum G_π für jeden Knoten v einen kürzesten Pfad zu s.

Der Algorithmus von Dijkstra

Benutzt eine *priority queue* Q, die Knoten $v \in V$ mit Schlüssel $d[v]$ hält,
und eine dynamische Menge S.

DIJKSTRA(G, w, s)

- Rufe INITIALIZE(G, s) auf, setze $S \leftarrow \emptyset$ und $Q \leftarrow V$.
- Solange $Q \neq \emptyset$ ist, setze $u \leftarrow$ EXTRACT-MIN(Q) und füge u zu S hinzu.
- Für jedes $v \in Adj[u]$ führe Relax(u, v, w) aus.
  (Bemerke: dies beinhaltet DECREASE-KEY-Operationen.)
  Anschliessend nächste Iteration.

Korrektheit:

Nach Ausführung von DIJKSTRA(G, w, s) ist $d[v] = \delta(s, v)$ für alle $v \in V$.

Komplexität: Hängt von der Realisierung der queue Q ab. (Vgl. Prim)

Als Liste: $O(|V|^2)$ Als Heap: $O(|E| \log |V|)$
Als Fibonacci-Heap (s. CORMEN): $O(|V| \log |V| + |E|)$.

Der Algorithmus von Bellman-Ford

Bellman-Ford(G, w, s)

- Rufe Initialize(G, s) auf.
- Wiederhole $|V| - 1$ mal:

  Für jede Kante $(u, v) \in E$ rufe Relax(u, v, w) auf.
- Für jede Kante $(u, v) \in E$, teste ob $d[v] > d[u] + w(u, v)$ ist.
- Falls ja für eine Kante, drucke "*negativer Zyklus vorhanden*",
  sonst brich mit Erfolg ab.

Korrektheit: Nach Ausführung von Bellman-Ford(G, w, s) gilt:

Ist kein negativer Zyklus von s erreichbar, dann ist
$d[v] = \delta(s, v)$ für alle $v \in V$, und der Algorithmus terminiert erfolgreich.

Andernfalls ist wird der negative Zyklus durch den Test entdeckt.

Komplexität ist offenbar $O(|V| \cdot |E|)$.

Kürzeste Wege zwischen allen Paaren

Aufgabe: Berechne $\delta(i,j)$ für alle Paare $i,j \in V = \{1,\ldots,n\}$.
Kantengewichte in Matrix $W = (w_{i,j})$, mit $w_{i,i} = 0$.

Dynamische Programmierung: Berechne rekursiv die Werte

$d_{i,j}^{(m)}$ = minimales Gewicht eines Weges von i zu j, der m Kanten lang ist.

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

$$d_{i,j}^{(m)} = \min\Big(d_{i,j}^{(m-1)}, \min_{k\neq j}\big(d_{i,k}^{(m-1)} + w_{k,j}\big)\Big)$$

$$= \min_{1\leq k\leq n}\big(d_{i,k}^{(m-1)} + w_{k,j}\big)$$

Keine negativen Zyklen $\quad\rightsquigarrow\quad \delta(i,j) = d_{i,j}^{(n-1)} = d_{i,j}^{(m)}$ für alle $m \geq n-1$.

Kürzeste Wege und Matrizenmultiplikation

Betrachte Matrizen $D^{(m)} = \left(d_{i,j}^{(m)}\right)$. Es gilt

$$D^{(m)} = D^{(m-1)} \odot W$$

wobei $\odot$ eine Art Multiplikation ist mit $\min \hat{=} \sum$ und $+ \hat{=} \times$.

Matrix $D^{(n-1)} = \left(\delta(i,j)\right)$ kann ausgerechnet werden in Zeit $\Theta(n^4)$.

Bessere Methode durch iteriertes Quadrieren:

Da für $m \geq n-1$ gilt $D^{(m)} = D^{(n-1)}$, und $\odot$ assoziativ ist,
berechne $D^m = D^{(n-1)}$ für $m = 2^{\lceil \log(n-1) \rceil}$ mittels

$$D^{(1)} = W$$
$$D^{(2k)} = D^{(k)} \odot D^{(k)}$$

Zeitkomplexität: nur $\Theta(n^3 \log n)$.

Der Algorithmus von Floyd-Warshall

Betrachte Weg von i nach j:

$$\langle i = v_0, v_1, \ldots, v_{\ell-1}, v_\ell = j \rangle$$

Knoten $v_1, \ldots, v_{\ell-1}$ sind die Zwischenknoten.

Dynamische Programmierung: Berechne rekursiv die Werte

$d_{i,j}^{(k)}$ = minimales Gewicht eines Weges von i zu j, der nur Zwischenknoten $\{1, \ldots, k\}$ verwendet.

$$d_{i,j}^{(0)} = w_{i,j}$$
$$d_{i,j}^{(k)} = \min\big(d_{i,j}^{(k-1)} , d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \big)$$

Klar: $\delta(i,j) = d_{i,j}^{(n)}$.

Matrix $D^{(n)} = \big(d_{i,j}^{(n)}\big) = \big(\delta(i,j)\big)$ kann in Zeit $\Theta(n^3)$ berechnet werden.

Flüsse in Netzwerken

Gegeben: gerichteter Graph $G = (V, E)$ mit Quelle $s \in V$ und Senke $t \in V$,
für $(u, v) \in E$ Kapazität $c(u, v) \geq 0$. Für $(u, v) \notin E$ sei $c(u, v) = 0$.

Gesucht: Ein Fluss durch G: Funktion $f : V \times V \to \mathbb{R}$ mit

1. $f(u, v) \leq c(u, v)$
2. $f(u, v) = -f(v, u)$
3. Für alle $u \in V \setminus \{s, t\}$:

$$\sum_{v \in V} f(u, v) = 0$$

Wert des Flusses f

$$|f| := \sum_{v \in V} f(s, v)$$

soll maximiert weden.

Eigenschaften von Flüssen

Für $X, Y \subseteq V$ sei $f(X,Y) := \sum_{x \in X} \sum_{y \in Y} f(x,y)$.

Abkürzung: $f(v, X) = f(\{v\}, X)$.

Eigenschaft 3 lautet damit: $f(u, V) = 0$.

Lemma: Für alle $X, Y, Z \subseteq V$ mit $Y \cap Z = \emptyset$ gilt:

- $f(X,X) = 0$
- $f(X,Y) = -f(Y,X)$
- $f(X, Y \cup Z) = f(X,Y) + f(X,Z)$
- $f(Y \cup Z, X) = f(Y,X) + f(Z,X)$

Restnetzwerke und Erweiterungspfade

Sei f ein Fluss in einem Netzwerk $G = (V, E)$ mit Kapazität c.

Für $u, v \in V$ ist die Restkapazität $c_f(u, v) = c(u, v) - f(u, v)$.

Das Restnetzwerk $G_f = (V, E_f)$ ist gegeben durch

$$E_f := \{(u, v) \; ; \; c_f(u, v) > 0\} \; .$$

Lemma: Ist f' ein Fluss in G_f, so ist $f + f'$ ein Fluss in G mit Wert $|f| + |f|$.

Ein Weg $p : s \rightsquigarrow t$ in G_f ist ein Erweiterungspfad, seine Restkapazität ist

$$c_f(p) = \min\{c_f(u, v) \; ; \; (u, v) \text{ Kante in } p\}$$

Für einen Erweiterungspfad p definiere

$$f_p(u, v) = \begin{cases} c_f(p) & (u, v) \text{ in } p \\ -c_f(p) & (v, u) \text{ in } p \\ 0 & \text{sonst} \end{cases}$$

Dann ist f_p ein Fluss in G_f.

Das Max-Flow-Min-Cut Theorem

Ein Schnitt in G ist eine Zerlegung (S, T) mit $s \in S \subseteq V$ und $t \in T = V \setminus S$.

Lemma: Ist (S, T) ein Schnitt, so ist $f(S, T) = |f|$.

Satz: Die folgenden Aussagen sind äquivalent:

1. f ist ein maximaler Fluss in G.
2. Im Restnetzwerk G_f gibt es keinen Erweiterungspfad.
3. Es gibt einen Schnitt (S, T) mit $|f| = c(S, T)$.

Die Ford-Fulkerson-Methode

FORD-FULKERSON(G, s, t, c)

- Initialisiere $f(u, v) = 0$ für alle $u, v \in V$.
- Solange es einen Erweiterungspfad p in G_f gibt:

  Setze für jede Kante (u, v) in p

$$f(u, v) \leftarrow f(u, v) + c_f(p) \quad ; \quad f(v, u) \leftarrow -f(u, v)$$

Korrektheit folgt aus dem Max-Flow-Min-Cut-Theorem.

Komplexität hängt davon ab, wie man nach Erweiterungspfaden sucht.

Ist $c(x, y) \in \mathbb{N}$ für alle $(x, y) \in E$, so ist die Laufzeit $O(|E| \cdot |f^*|)$, für einen maximalen Fluss f^*.

Der Algorithmus von Edmonds-Karp

Algorithmus von Edmonds-Karp:

Suche bei Ford-Fulkerson Erweiterungspfade mittels Breitensuche in G_f.

Für $v \in V$, sei $\delta_f(s, v)$ die Distanz von s zu v in G_f.

Lemma: Beim Ablauf des Algorithmus von Edmonds-Karp steigt $\delta_f(s, v)$ für jeden Knoten $v \in V \setminus \{s, t\}$ monoton an.

Satz: Die Zahl der Iterationen der äußeren Schleife beim Algorithmus von Edmonds-Karp ist $O(|V| \cdot |E|)$.

Damit: Laufzeit ist $O(|V| \cdot |E|^2)$.

Anwendung: Maximale Matchings

Sei $G = (V, E)$ ein ungerichteter Graph. Ein Matching in G ist $M \subseteq E$ mit

$$e_1 \cap e_2 = \emptyset \qquad \text{für alle } e_1, e_2 \in M \ .$$

Aufgabe: Gegeben ein bipartiter Graph $G = (V, E)$ mit $V = L \cup R$ und $E \subseteq L \times R$, finde ein Matching maximaler Größe.

Idee: Betrachte $G' = (V', E')$, wobei $V' = V \cup \{s, t\}$, und

$$E' = E \cup \{(s, \ell)\,;\, \ell \in L\} \cup \{(r, t)\,;\, r \in R\}$$

mit Kapazität $c(e) = 1$ für alle $e \in E'$.

Beobachtung: Jedes Matching M in G entspricht einem ganzzahligen Fluss in G' mit $|f| = |M|$, und umgekehrt.

Satz: Ist die Kapazitätsfunktion c ganzzahlig, so ist auch der mit der Ford-Fulkerson-Methode gefundene maximale Fluss ganzzahlig.

Zusammenfassung: Graphenalgorithmen

- L

VI. Schnelle Fouriertransformation

- Motivation: Gleichung der schwingenden Saite
- Crashkurs "Komplexe Zahlen"
- Anwendung: Mustererkennung, Signalverarbeitung (skizziert)
- Anwendung: Multiplikation von Polynomen
- Schnelle Implementierung mit *divide-and-conquer* (FFT)

Schwingende Saite als Motivation

Eine Saite der Länge 1 sei an ihren Enden fest eingespannt. Ihre Auslenkung an der Stelle x (wobei $0 \leq x \leq 1$) zum Zeitpunkt t (wobei $t \geq 0$) bezeichnen wir mit $u(x,t)$.

Unter vereinfachenden Annahmen gilt:

$$c^2 \frac{\partial^2}{\partial x^2} u(x,t) = \frac{\partial^2}{\partial t^2} u(x,t)$$

- c ist eine Konstante, in die Spannung und Masse der Saite eingehen.
- Die linke Seite entspricht der Krümmung der Saite an der Stelle x zum Zeitpunkt t; daraus resultiert eine Rückstellkraft.
- Die rechte Seite entspricht der Beschleunigung, also nichts anderes als Newtons Gesetz "Kraft ist Masse mal Beschleunigung".

Theorie der partiellen DGL $\Rightarrow$: ist $u(x,0)$ vorgegeben (Anfangsauslenkung), gilt $\frac{\partial u}{\partial t}(x,0) = 0$ (Saite wird nicht angeschoben) und gilt $u(0,t) = u(1,t) = 0$ (Saite ist an den Enden eingespannt), so ist $u(x,t)$ für alle $0 \leq x \leq 1$ und $t \geq 0$ eindeutig festgelegt (entsprechend der Anschauung!)

Aber wie rechnet man $u(x,t)$ aus $v(x) := u(x,0)$ aus??

Dieses Problem löste Fourier im 19.Jh.

Fouriers Lösung

Wenn $v(x) = u(x,0) = \sin(\pi k x)$, dann ist $u(x,t) = \sin(\pi k x)\cos(\pi c k t)$.

Beweis durch Nachrechnen:

$$c^2 \frac{\partial^2}{\partial x^2} u(x,t) = -c^2\pi^2 k^2 u(x,t) = \frac{\partial^2}{\partial t^2} u(x,t)$$

Falls $v(x) = \sum_{k=1}^{\infty} \alpha_k \sin(\pi k x)$ dann $u(x,t) = \sum_{k=1}^{\infty} c_k \sin(\pi k x)\cos(\pi c k t)$.

Fouriers geniale Feststellung: Jedes stetige v ist von dieser Form:

$$\alpha_k = \frac{2}{\pi k} \int_0^{\pi} v(x)\sin(\pi k x)\mathrm{d}x$$

Die Folge α_k ist die *Fouriertransformation* von v (genauer: Folge der Fourierkoeffizienten).

Diskrete Fouriertransformation

Gegeben sei eine Folge $a_0, a_1, \ldots, a_{n-1}$ reeller Zahlen. Z.B. Stützstellen einer Funktion.

Die *diskrete Fouriertransformation* (DFT) dieser Folge ist die Folge $y_0, y_1, \ldots, y_{n-1}$ *komplexer Zahlen* definiert durch

$$y_k = \sum_{j=0}^{n-1} e^{\frac{2\pi i}{n} jk} a_j$$

Zusammenhang mit Motivation: $e^{i\phi} = \cos(\phi) + i \sin(\phi)$ (EULERsche Formel). Die α_k können aus Imaginärteil der DFT näherungsweise bestimmt werden.

NB: Die Motivation mit „schwingender Saite“ ist nicht prüfungsrelevant.

Wir interessieren uns für die DFT als eigenständige Operation aus anderen (aber verwandten) Gründen.

Komplexe Zahlen

Menge der komplexen Zahlen $\mathbb{C}$ entsteht aus $\mathbb{R}$ durch formale Hinzunahme von $i = \sqrt{-1}$, also $i^2 = -1$.

Eine komplexe Zahl ist ein Ausdruck der Form $a + ib$ wobei $a, b \in \mathbb{R}$.

Addition: $(a_1 + ib_1) + (a_2 + ib_2) = (a_1 + a_2) + i(b_1 + b_2)$.

Multiplikation:
$(a_1 + ib_1)(a_2 + ib_2) = a_1a_2 + i(a_1b_2 + a_2b_1) + i^2b_1b_2 = (a_1a_2 - b_1b_2) + i(a_1b_2 + a_2b_1)$.

Division: $\frac{a_1+ib_1}{a_2+ib_2} = \frac{(a_1+ib_1)(a_2-ib_2)}{(a_2+ib_2)(a_2-ib_2)} = \frac{a_1a_2+b_1b_2+i(a_2b_1-a_1b_2)}{a_2^2+b_2^2}$

Bemerkung: a ist der Realteil der komplexen Zahl $a + ib$, die Zahl b ist ihr Imaginärteil. Man schreibt $a = \mathrm{Re}(a + ib)$ und $b = \mathrm{Im}(a + ib)$.

Zahlenebene

Man visualisiert komplexe Zahlen als Punkte oder Ortsvektoren. $a + ib$ entspricht dem Punkt (a, b).

Addition entspricht dann der Vektoraddition.

Betrag: $|a + ib| = \sqrt{a^2 + b^2}$. Es gilt $|wz| = |w||z|$.

Man kann einer komplexen Zahl $z = a + ib$ ihren Winkel im Gegenuhrzeigersinn von der reellen Achse aus gemessen, zuordnen. Dieser Winkel ϕ heißt Argument von z, i.Z. $\arg(z)$. Es gilt $\phi = \texttt{atan2}(a, b)$. D.h. $\tan(\phi) = b/a$ und $b \leq 0 \Leftrightarrow |\phi| \geq \pi/2$.

Aus Betrag $r = |z|$ und Argument $\phi = \arg(z)$ kann man z rekonstruieren:

$$z = r\cos(\phi) + ir\sin(\phi)$$

Mit $e^{i\phi} = \cos(\phi) + i\sin(\phi)$ erhält man

$$z = re^{i\phi}$$

Dies ist die Polarform der komplexen Zahl z.

Es gilt: $(r_1 e^{i\phi_1})(r_2 e^{i\phi_2}) = r_1 r_2 e^{i(\phi_1+\phi_2)}$.

Beim Multiplizieren multiplizieren sich die Beträge und addieren sich die Argumente.

Einheitswurzeln

Sei $n \in \mathbb{N}$. Wir schreiben $\omega_n := e^{2\pi i/n}$.

Es gilt $\omega_n^n = 1$ und allgemeiner $(\omega_n^k)^n = 1$.

Die n (verschiedenen) Zahlen $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}$ heißen daher n-te Einheitswurzeln.

Sie liegen auf den Ecken eines regelmäßigen n-Ecks mit Mittelpunkt Null und einer Ecke bei der Eins.

Satz: Es gilt für $k = 0, \ldots, n-1$:

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \begin{cases} n, \text{ falls } k = 0 \\ 0, \text{sonst} \end{cases}$$

Beweis mit geometrischer Reihe oder durch Veranschaulichung in der Zahlenebene.

Satz: Ist n gerade, so sind die $n/2$-ten Einheitswurzeln gerade die Quadrate der n-ten Einheitswurzeln.

Beweis: Eine Richtung: $((\omega_n^k)^2)^{n/2} = 1$. Andere Richtung: $\omega_{n/2}^k = (\omega_n^2)^k$.

Einheitswurzeln und DFT

Definition der DFT: $y_k = \sum_{j=0}^{n-1} e^{\frac{2\pi i}{n} jk} a_j$.

Es ist $y_k = \sum_{j=0}^{n-1} \omega_n^{jk} a_j$.

Die DFT ergibt sich also als die Folge $A(1), A(\omega_n), A(\omega_n^2), \ldots, A(\omega_n^{n-1})$ wobei $A(x) = \sum_{j=0}^{n-1} a_j x^j$.

Translationsinvarianz

Satz: Sei $(a_j)_{j=0}^{n-1}$ eine Folge reeller Zahlen und $b_j = a_{(j+d) \bmod n}$. Sei $(y_k)_k$ die DFT von $(a_j)_j$ und $(z_k)_k$ die DFT von $(b_j)_j$. Es gilt $|y_k| = |z_k|$. D.h. der Betrag der DFT, das sog. Intensitätsspektrum, ist translationsinvariant.

Beweis: $z_k = \sum_j \omega_n^{jk} a_{j+d \bmod n} = \sum_{j'} \omega_n^{(j'-d)k} a_{j'k} = \omega_n^{-dk} y_k$. Und $|\omega_n^x| = 1$.

Anwendung: Um festzustellen, ob ein bestimmtes Muster irgendwo in der Folge a vorhanden ist, vergleiche man die DFT von a mit der DFT des Musters.
Insbesondere interessant bei 2D-DFT. Z.B. Schrifterkennung.

Inverse DFT

Gegeben sei eine Folge $y_0, \ldots, y_{n-1}$ komplexer Zahlen.

Die inverse DFT ist die Folge $a_0, \ldots, a_{n-1}$ definiert durch

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{-jk} y_k$$

Satz: Die inverse DFT ist tatsächlich die zur DFT inverse Operation.

Beweis: Wir verwenden das Kroneckersymbol $\delta_{jl} = \texttt{if } j = l \texttt{ then } 1 \texttt{ else } 0$.

Es genügt, die Behauptung für $a_j = \delta_{jl}$ für alle $l = 0, \ldots, n-1$ zu zeigen, da DFT, und inverse DFT linear sind und diese a's eine Basis bilden.

Sei also $a_j = \delta_{jl}$. Anwendung der DFT liefert

$$y_k = \omega_n^{kl}$$

Anwendung der inversen DFT liefert dann

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{-jk} \omega_n^{kl} = \frac{1}{n} \sum_{k=0}^{n-1} (\omega_n^k)^{l-j} = \delta_{lj}$$

DFT und Polynome

Ein Polynom mit Gradschranke n ist ein formaler Ausdruck der Form

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

Die a_j heißen Koeffizienten des Polynoms A.

Man kann ein Polynom als Funktion auffassen (Polynomfunktion). Z.B.:
$A(x) = 1 + 2x + 3x^2$ und $A(2) = 17$.

Hornerschema

Sei $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ ein Polynom mit Gradschranke n und x_0 eine Zahl.

Bestimmung von $A(x_0)$ nach HORNER mit $\Theta(n)$ Rechenoperationen:

$$A(x_0) = a_0 + x_0(a_1 + x_1(a_2 + \cdots + x_0(a_{n-2} + x_0 a_{n-1}) \dots)$$

Interpolation

Ein Polynom ist durch seine Polynomfunktion eindeutig bestimmt, d.h., man kann die Koeffizienten aus der Polynomfunktion ausrechnen.

Genauer: Sei $A(x_k) = y_k$ für n verschiedene Stützstellen $x_0, \ldots, x_{n-1}$.

Dann gilt nach Lagrange:

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

Ausmultiplizieren liefert das gewünschte Polynom.

Bemerkung: Dies gilt nur bei den reellen Zahlen, oder genauer bei Körpern der Charakteristik 0. Im zweielementigen Körper induzieren x^2 und x dieselben Polynomfunktionen.

Addition und Multiplikation von Polynomen

Polynome kann man addieren und multiplizieren:

Die Summe zweier Polynome $A(x) = \sum_{j=0}^{n-1} a_j x^j$ und $B(x) = \sum_{j=0}^{n-1} b_j x^j$ mit Gradschranke n ist das Polynom $C(x) = \sum_{j=0}^{n-1} c_j x^j$ wobei $c_j = a_j + b_j$.

Das Produkt zweier Polynome $A(x) = \sum_{j=0}^{n-1} a_j x^j$ und $B(x) = \sum_{j=0}^{n-1} b_j x^j$ mit Gradschranke n ist das Polynom $C(x) = \sum_{j=0}^{2n-1} c_j x^j$ mit Gradschranke $2n$ wobei $c_j = \sum_{l=0}^{j} a_l b_{j-l}$.

Z.B.: $(1 + 2x + 3x^2)(4 + 5x + 6x^2) = 1 + 13x + 25x^2 + 27x^3 + 18x^4$.

Sind Polynome als Koeffizientenliste repräentiert, so erfordert das Ausrechnen der Summe $O(n)$ Operationen; das Ausrechnen des Produkts erfordert $O(n^2)$ Operationen.

Punkt-Wert Repräsentation

Man kann Polynome auch als Punkt-Wert Liste repräsentieren, also als Liste von Paaren

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

wobei die x_k paarweise verschieden sind.

Z.B. $A(x) = 1 + 2x + 3x^2$ repräsentiert durch $((0, 1), (1, 6), (-1, 2))$.

Punkt-Wert Repräsentation

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

und $((x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1}))$

zwei solche Repräsentationen zu denselben Stützstellen, so erhält man Repräsentationen für Summe und Produkt als

$$((x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1}))$$

und

$$((x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{n-1}, y_{n-1} y'_{n-1}))$$

D.h. Addition und Multiplikation von Polynomen in Punkt-Wert Repräsentation erfordert $O(n)$ Operationen.

Aber die Auswertung von Polynomen in Punkt-Wert Repräsentation erfordert $O(n^2)$ Operationen (*there ain't no such thing as a free lunch*).

Multiplikation mit DFT

Erinnerung: Die DFT einer Liste $(a_0, \ldots, a_{n-1})$ ist die Liste $(y_0, \ldots, y_{n-1})$ wobei

$$y_k = A(\omega_n^k)$$

und

$$A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

Die DFT von $(a_0, \ldots, a_{n-1})$ ist die Punkt-Wert Repräsentation des Polynoms $\sum_{j=0}^{n-1} a_j x^j$ zu den Stützstellen $\omega_n^0, \ldots, \omega_n^{n-1}$, also den n-ten Einheitswurzeln.

Um zwei Polynome in Koeffizientenform zu multiplizieren kann man ihre DFTs punktweise multiplizieren und dann die inverse DFT anwenden.

Multiplikation mit DFT

- Gegeben zwei Polynome A, B mit Gradschranke n als Koeffizientenlisten
- Fülle mit Nullen auf um formell Polynome mit Gradschranke $2n$ zu erhalten.

$$a = (a_0, \ldots, a_{2n-1}), \quad b = (b_0, \ldots, b_{2n-1})$$

- Bilde jeweils DFT der beiden expandierten Koeffizientenlisten:
  $(y_0, \ldots, y_{2n-1}) = \mathrm{DFT}_{2n}(a)$, $(z_0, \ldots, z_{2n-1}) = \mathrm{DFT}_{2n}(b)$,
- Multipliziere die DFTs punktweise: $w_k = y_k z_k$ für $k = 0, \ldots, 2n-1$,
- Transformiere zurück um Koeffizientenliste von $C = AB$ zu erhalten:
  $(c_0, \ldots, c_{2n-1}) = \mathrm{DFT}^{-1}(w_0, \ldots, w_{2n-1})$.

Leider braucht das Ausrechnen der DFT $\Theta(n^2)$ Rechenoperationen, sodass keine unmittelbare Verbesserung eintritt.

Mit *divide and conquer* kann man aber die DFT in $\Theta(n \log n)$ ausrechnen!

Schnelle Fouriertransformation (FFT)

Sei n bis auf weiteres eine Zweierpotenz.

$A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$.

Es ist $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$ wobei

$$A^{[0]}(z) = a_0 + a_2 z + a_4 z^2 + \cdots + a_{n-2} z^{n/2-1}$$

$$A^{[1]}(z) = a_1 + a_3 z + a_5 z^2 + \cdots + a_{n-1} z^{n/2-1}$$

Beispiel:

$$A(x) = 1 + 2x + 3x^2 + 4x^3$$
$$A^{[0]}(z) = 1 + 3z$$
$$A^{[1]}(z) = 2z + 4z^2$$

Schnelle Fouriertransformation (FFT)

Also gilt unter Beachtung von $\omega_n^{2k} = \omega_{n/2}^k$:

$$A(\omega_n^k) = A^{[0]}(\omega_{n/2}^k) + \omega_n^k A^{[1]}(\omega_{n/2}^k)$$

Für $k < n/2$ sind die beiden rechtsstehenden Polynomauswertungen selbst DFTs, aber zur halben Größe.

Für $k \geq n/2$ passiert nichts Neues, da $\omega_{n/2}^{k+n/2} = \omega_{n/2}^k$.

Nur der Vorfaktor ω_n^k ändert sich: $\omega_n^{k+n/2} = -\omega_n^k$.

Schnelle Fouriertransformation (FFT)

Schnelle Fouriertransformation (FFT)

RECURSIVE-FFT(a)

1 $n \leftarrow \mathit{length}[a]$

2 **if** $n = 1$

3 **then return** a

4 $\omega_n \leftarrow e^{2\pi i/n}$

5 $\omega \leftarrow 1$

6 $a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n/2-2})$

7 $a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n/2-1})$

8 $y^{[0]} \leftarrow$ RECURSIVE-FFT$(a_0, a_2, \ldots, a_{n/2-2})$

9 $y^{[1]} \leftarrow$ RECURSIVE-FFT$(a_1, a_3, \ldots, a_{n/2-1})$

10 **for** $k \leftarrow n/2 - 1$ **do**

11 $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$

12 $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$

13 $\omega \leftarrow \omega\omega_n$

14 **return** y

Schnelle Fouriertransformation (FFT)

- Nach dem Vorhergesagten ist klar, dass y die DFT von a ist.
- Die Laufzeit $T(n)$ von RECURSIVE-FFT erfüllt $T(n) = 2T(n) + \Theta(n)$, also $T(n) = \Theta(n \log n)$.
- In der Praxis rechnet man die DFT iterativ von unten her aus (vgl. dynamische Programmierung), indem man sich überlegt, für welche Arrays rekursive Aufrufe stattfinden.
- Man kann die FFT auch gut parallelisieren.
- Zur Multiplikation von Polynomen mit ganzen Koeffizienten empfiehlt sich die modulare DFT.

DFT in 2D

Sei $a_{jk}, j = 0, \ldots, n-1, k = 0, \ldots, n-1$ eine Matrix von Zahlen. Die DFT von a ist die Matrix $y_{uv}, u = 0, \ldots, n-1, v = 0, \ldots, n-1$ definiert durch

$$y_{uv} = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \omega_{2n}^{ju+kv} a_{jk}$$

Translationsinvarianz des Intensitätsspektrums gilt jetzt bezüglich Verschiebung in j und k Richtung.

Schnelle inverse DFT

Erinnerung: Die inverse DFT einer Liste $(y_0, \ldots, y_{n-1})$ ist die Liste $(a_0, \ldots, a_{n-1})$ wobei

$$a_j = \frac{1}{n} Y(\omega_n^{-j})$$

und

$$Y(x) = y_0 + y_1 x + \cdots + y_{n-1} x^{n-1}$$

Aber $\omega_n^{-j} = \omega_n^{n-j}$. Man erhält die inverse DFT von y aus der DFT von y durch Umgruppieren und Dividieren durch n:

Schnelle inverse DFT

INVERSE-FFT(y)

1 $n \leftarrow \mathit{length}[y]$

2 $a \leftarrow$ RECURSIVE-FFT(y)

3 **for** $j \leftarrow 1$ **to** $n/2 - 1$ **do**

4 $\quad$ *Swap* $a_j \leftrightarrow a_{n-j}$

5 **for** $j \leftarrow 0$ **to** $n - 1$ **do**

6 $\quad$ $a_j \leftarrow a_j/n$

7 **return** a

Faltung

Seien $a = (a_0, \ldots, a_{n-1})$ und $b = (b_0, \ldots, b_{n-1})$ zwei Folgen von je $2n$ Zahlen wobei $a_k = b_k = 0$ für $k \geq n$.

Die Faltung (engl.: *convolution*) $a \star b$ ist die Folge von $2n$ Zahlen definiert durch

$$(a \star b)_k = \sum_{j=0}^{k} a_j b_{k-j}$$

Beachte: $a \star b$ ist die Koeffizientenfolge des Produkts der durch a und b repräsentierten Polynome.

Satz: $a \star b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a)\mathrm{DFT}_{2n}(b))$ wobei das Produkt komponentenweise zu verstehen ist.

Anwendung der Faltung

Faltung mit $b = (-1, 2, -1, 0, 0, 0, 0, \ldots, 0)$ hebt Sprünge hervor.

In 2D: *edge detection*.

Faltung mit $b = (1, 2, 1, 0, 0, 0, 0, 0, \ldots, 0)$ verwischt.

In 2D: *blurring*.

Die Operation $c \mapsto \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(c)/\mathrm{DFT}_{2n}(b))$ macht die Faltung mit b rückgängig.

In 2D: *sharpening*

Die Faltung hat auch zahlreiche Anwendungen bei der Audiosignalverarbeitung.

VII. Suche in Zeichenketten

- Problemstellung & Anwendungen
- Grundlagen
- Rabin-Karp Verfahren (basierend auf hashing)
- Suche in Zeichenketten mit endlichen Automaten
- Knuth-Morris-Pratt Verfahren (Verfeinerung)
- Boyer-Moore Algorithmus (leistungsfähiges heuristisches Verfahren)
- Angenäherte Suche durch dynamische Programmierung.

Suche in Zeichenketten: Problemstellung

Σ ein endliches Alphabet (Menge) von Zeichen (characters).

Σ^* = Menge der Zeichenketten (strings) über Σ. Eine Zeichenkette ist ein Array von Zeichen.

$T \in \Sigma^*$ eine Zeichenkette der Länge n: der Text.

$P \in \Sigma^*$ eine Zeichenkette der Länge m: das Muster.

Frage: Kommt P in T vor, d.h. existiert $0 \leq s \leq n - m$ sodass $T[s + 1..s + m] = P$?

Z.B. $P =$ `abaa` kommt in $T =$ `abcabaabcabac` vor und zwar mit $s = 3$.

Anwendungen: Textverarbeitung, Suchen in Dokumenten oder Dateien, Molekularbiologie.

Notation und Grundlagen

$x, y, z, w \in \Sigma^*$.

$\epsilon =$ leeres Wort.

$|x| =$ Länge von X.

$w \sqsubset x \Leftrightarrow \exists y.x = wy$. Man sagt: w ist Präfix von x.

$w \sqsupset x \Leftrightarrow \exists y.x = yw$. Man sagt: w ist Suffix von x.

Beachte: $\epsilon \sqsubset x, \epsilon \sqsupset x$.

$x \sqsubset y \Leftrightarrow zx \sqsubset zy$.

$x \sqsupset y \Leftrightarrow xz \sqsupset yz$. **Lemma:** Gelte $x \sqsupset z$ und $y \sqsupset z$. Wenn $|x| \leq |y|$ dann $x \sqsupset y$. Wenn $|x| \geq |y|$ dann $y \sqsupset y$. Wenn $|x| = |y|$ dann $x = y$.

Abkürzung: $x_k = x[1..k]$.

P kommt in T vor, wenn s existiert mit $P \sqsupset T_{s+m}$.

String matching: naives Verfahren

NAIVE-STRING-MATCHER(T, P)

```
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n − m do
4        if P[1..m] = T[s + 1..s + m]
5              then print „Muster gefunden bei“ s
```

Worst-case Laufzeit: $\Theta(nm)$. Genauer: $(n - m + 1)m$ Zeichenvergleiche.

Rabin-Karp Verfahren

Berechne Hashwert des Musters $h(P)$ und vergleiche sukzessive mit Hashwert $h(T[s+1..s+m])$. Bei Übereinstimmung genau vergleichen.

Die Hashfunktion wird so gewählt werden, dass $h(T[s+2..s+1+m])$ aus $h(T[s+1..s+m])$ in Zeit $O(1)$ berechnet werden kann.

Die Laufzeit ist dann $\Theta(n+pm)$ wobei p die Zahl der Hashwertübereinstimmungen ist.

Schlimmstenfalls ist $p = n - m$ also keine Verbesserung. Meist ist aber p klein.

Hashfunktion bei Rabin-Karp

Es bietet sich die schon bekannte Hashfunktion an:

$h(x) = \sum_{i=1}^{|x|} d^{i-1} \ulcorner x_{|x|+1-i} \urcorner \bmod q$

Hier ist $d = |\Sigma|$.

Und $\ulcorner c \urcorner$ die Kodierung des Zeichens c als Zahl $1..|\Sigma|$ ist. Z.B.
$\Sigma =$ ASCII Zeichen, $d = 256$. Diese „Kodierungsecken“ $\ulcorner \cdot \urcorner$ lassen wir meist weg.

Und q ist irgendeine Zahl, günstigerweise eine Primzahl nicht nahe bei einer Zweierpotenz, derart dass $dq < 2^{\text{Wortlänge}}$.

Beachte:

$$h(T[s+2..s+1+m]) = d \cdot (h(T[s+1..s+m]) - d^{m-1} T[s+1]) + T[s+1+m] \bmod q$$

Natürlich reduziert man schon während der Berechnung ständig mod q.

Pseudocode: Initialisierungsphase

RABIN-KARP-MATCHER(T, P)

1 $n \leftarrow length[T]$

2 $m \leftarrow length[P]$

3 $D \leftarrow d^{m-1} \bmod q$

4 $p \leftarrow 0$

5 $t \leftarrow 0$

6 **for** $i \leftarrow 1$ **to** m **do**

7 $\quad p \leftarrow (dp + P[i]) \bmod q$

8 $\quad t \leftarrow (dt + T[i]) \bmod q$

Es ist $p = h(P)$ und $t = h(T[1..m])$ mit Hornerschema.

Pseudocode: Hauptteil

```
9   for s ← 0 to m − n do
10      if p = t
11          then if P = T[s + 1..s + m]
12                  print „Muster gefunden bei“ s
13      if s < n − m
14          then t ← (d(t − T[s + 1]D) + T[s + m + 1]) mod q
```

Invariante in Zeile 10: $t = h(T[s + 1..s + m])$.

Endliche Automaten

Ein endlicher Automat M ist ein 5-Tupel $M = (Q, q_0, A, \Sigma, \delta)$, wobei

- Q eine endliche Menge ist, die Zustände,
- $q_0 \in Q$, der Anfangszustand,
- $A \subseteq Q$, die Menge der akzeptierenden Zustände,
- Σ eine endliche Menge ist, das Eingabealphabet
- δ eine Funktion von $Q \times \Sigma$ nach Q ist, die Zustandsübergangsfunktion.

Beispiel: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $Q = \{0, 1\}$, $q_0 = 0$, $A = \{1\}$,
$\delta(0, \mathtt{a}) = 1, \delta(0, \mathtt{b}) = 0, \delta(1, x) = 0$.

Endliche Automaten

Der Automat arbeitet eine Zeichenkette w ausgehend von q_0 gemäß δ ab und erreicht dadurch einen Endzustand $\phi(w)$.

Formal:

$$\phi(\epsilon) = q_0$$
$$\phi(wa) = \delta(\phi(w), a), \text{für } w \in \Sigma^* \text{ und } a \in \Sigma$$

Die akzeptierte Sprache $L(M) \subseteq \Sigma^*$ ist definiert durch $L(M) = \{w \mid \phi(w) \in A\}$.

Im Beispiel: $L(M) = \{w \mid w = u\mathtt{ba}^k, k \text{ ungerade}\}$.

String-matching mit Automaten

Zu gegebenem Muster $P \in \Sigma^*$ definiere Automaten $M(P)$ wie folgt:

- $Q = \{0, 1, \ldots, m\}$ (wie immer $m = |P|$)
- $q_0 = 0$
- $A = \{m\}$
- $\delta(q, x) =$ „das größte k sodass $P_k \sqsupset P[1..q]x$“

Suffixfunktion: Wir schreiben $\sigma(x) = \max\{k \mid P_k \sqsupset x\}$, also $\delta(q, x) = \sigma(P_q x)$.

Theorem: $\phi(T_i) = \sigma(T_i)$. (Wird später bewiesen.)

D.h. arbeitet man den Text gemäß $M(P)$ ab, so gibt der gerade erreichte Zustand an, welches Anfangsstück des Musters man gerade „gematcht“ hat. Wird dieser Zustand gleich m, so hat man ganz P „gematcht“ und das Muster kommt vor.

String-matching mit Automaten

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n ← length[T]
2  q → 0
3  for i → 1 to n do
4        q → δ(q, T[i])
5        if q = m
6              then s → i − m
7                   print „Muster gefunden bei“ s
```

Korrektheit und Laufzeit

Lemma: Für $x \in \Sigma^*$ und $a \in \Sigma$ gilt $\sigma(xa) \leq \sigma(x) + 1$.

Lemma: Für $x \in \Sigma^*$ und $a \in \Sigma$ gilt: Wenn $q = \sigma(x)$ dass $\sigma(xa) = \sigma(P_q a)$.

Das Theorem $\phi(T_i) = \sigma(T_i)$ folgt nun mit Induktion über Textlänge.

Theorem: $\phi(T_i) = \sigma(T_i)$.

Laufzeit: $O(n) +$ „Zeit für die Konstruktion von $M(P)$“

Berechnung von $M(P)$

Die Zustandsübergangsfunktion wird zu Beginn in einem 2D-Array der Größe $(m+1) \times |\Sigma|$ abgespeichert.

Die Berechnung jedes dieser Einträge gemäß der Definition

$$\delta(q, x) = \text{„das größte } k \text{ sodass } P_k \sqsupset P[1..q]x\text{“}$$

erfordert Zeit $O(m^2)$.

Daher Gesamtlaufzeit : $O(m^3|\Sigma|)$.

Das beste Verfahren zur Berechnung von $M(P)$ braucht nur $O(m|\Sigma|)$.

Knuth-Morris-Pratt (KMP) Verfahren

Beim Finite-Automaton-Matcher verwenden wir

$$\delta : \{0, \ldots, m\} \times \Sigma \to \{0, \ldots, m\}$$

um aus dem alten „gematchten“ Präfix ein neues, d.h., welches das nächste Zeichen einschließt, zu berechnen.

Redundanzen der Funktion δ:

- Ist $q \neq m \wedge P[q+1] = a$ so ist $\delta(q, a) = q + 1$,
- Ist $k := \delta(q, a) > 0$, so ist $P[k] = a$

Idee von KMP: Nutze diese Redundanzen um die Abhängigkeit der Übergangsfunktion von a ganz zu vermeiden.

Knuth-Morris-Pratt (KMP) Verfahren

Finde $\pi : \{0, \ldots, m\} \to \{0, \ldots, m\}$ sodass

- $\pi(q) < q$ oder $\pi(q) = 0$,
- $q = m \vee P[q+1] \neq a \Rightarrow \delta(q, a) = \delta(\pi(q), a)$

Mit so einem π (wenn wir es haben) können wir δ wie folgt rekonstruieren.

```
DELTA-FROM-PI(q, a)
1  if q = m then q ← π(q)
2  while q > 0 ∧ P[q + 1] ≠ a do
3        q ← π(q)
4  if P[q + 1] = a then return q + 1 else return 0
```

Die Präfixfunktion

Definiere die Präfixfunktion als

$$\pi(q) = \max\{k \mid k < q \wedge P_k \sqsupset P_q\}$$

Klar gilt $\pi(q) < q$ oder $\pi(q) = 0$ (beachte $\max \emptyset = 0$).

Theorem: $q = m \vee P[q+1] \neq a \Rightarrow \delta(q,a) = \delta(\pi(q), a)$.

Beweis des Theorems

Lemma: Wenn $k < q$ und $P_k \sqsupset P_q$, dann $P_k \sqsupset P_{\pi(q)}$.

Beweis: Es gilt nach Definition $\pi(q) \geq k$. Die Behauptung folgt mit dem Lemma auf Folie 202.

Beweis des Theorems: Sei $q = m$ oder $P[q+1] \neq a$. Sei $0 \leq k \leq m$ beliebig. Wir zeigen $P_k \sqsupset P_q a \Leftrightarrow P_k \sqsupset P_{\pi(q)} a$: Wenn $P_k \sqsupset P_q a$, so kann muss unter den Voraussetzungen $k \leq q$ sein. Entweder ist $k = 0$ und trivialerweise $P_k = \epsilon \sqsupset P_{\pi(q)} a$ oder $P[k] = a$ und $P_{k-1} \sqsupset P_q$. Mit dem Lemma gilt $P_{k-1} \sqsupset P_{\pi(q)}$ und $P_k \sqsupset P_{\pi(q)} a$. Die umgekehrte Richtung ist klar, da $P_{\pi(q)} \sqsupset P_q$.

Das KMP-Verfahren

KMP-Matcher(T, P, π)

1 $n \leftarrow \mathit{length}[T]$; $m \leftarrow \mathit{length}[P]$; $q \rightarrow 0$

3 **for** $i \leftarrow 1$ **to** n **do**

7 **while** $q > 0 \wedge P[q+1] \neq T[i]$ **do** $q \leftarrow \pi(q)$

8 **if** $P[q+1] = T[i]$ **then** $q \leftarrow q+1$

9 **if** $q = m$ **then**

10 print „Muster gefunden bei“ $i - m$

12 $q \leftarrow \pi(q)$

Invariante nach Zeile 3: $q < m$.

Invariante nach Zeile 8: $q = \delta(q_0, T[i])$, wobei q_0 Wert von q nach Zeile 8 beim vorherigen Durchlauf.

KMP-Matcher ist äquivalent zu Finite-Automaton-Matcher.

Berechung der Präfixfunktion

Es ist $\pi(0) = \pi(1) = 0$ und für $q > 1$ gilt

$$\pi(q) = \delta(\pi(q-1), P[q])$$

Aber dieses δ lässt sich mit DELTA-FROM-PI aus $\pi(k)$ für $k < q$ berechnen.

Zur Erleichterung der Analyse expandieren wir den Code von DELTA-FROM-PI und lassen die erste Zeile weg, die ja nicht gebraucht wird, da $k < q \leq m$.

Berechung der Präfixfunktion

```
COMPUTE-PREFIX-FUNCTION(P)
1  m ← length[P] ; π[1] ← 0 ; k ← 0
2  for q ← 2 to m do
3       while k > 0 ∧ P[k + 1] ≠ P[q] do
4            k ← π[k]
5       if P[k + 1] = P[q] then k ← k + 1
6       π[q] ← k
7  return π
```

Berechnung der Zahl der Ausführungen von Zeilen 4 und 5 mit Potentialmethode.

Potential: k.

Amortisierte Kosten von Zeile 4: ≤ 0, von Zeile 5: ≤ 2.

Laufzeit

Das Potential liegt zwischen 0 und m:

Die tatsächlichen Kosten sind beschränkt durch die Summe der amortisierten Kosten:

Die Zeilen 4 und 5 werden zusammen $\leq 2m$ Mal durchlaufen.

Die Laufzeit von Compute-Prefix-Function ist also $O(m)$.

Laufzeit von KMP-Matcher

Wir zählen die Durchläufe von Zeilen 7 und 8.

Als Potential nehmen wir q.

Zeile 7 hat amortisierte Kosten ≤ 0

Zeile 8 hat amortisierte Kosten ≤ 2

Also sind die tatsächlichen Kosten $\leq 2n$

Die Laufzeit ist $O(n)$.

Rechnet man noch die Zeit für die Berechnung der Präfixfunktion hinzu, so ergibt sich eine Laufzeit von $O(m+n)$ für das KMP-Verfahren.

Boyer-Moore Verfahren

```
BOYER-MOORE-MATCHER(T, P)
1   n ← length[T] ; m ← length[P] ; s ← 0
4   while s ≤ n − m do
5       j ← m
6       while j > 0 ∧ P[j] = T[s + j] do j ← j − 1
7       if j = 0
8       print „Muster gefunden bei“ s ; ADVANCE
10      else ADVANCE
```

Wählen wir ADVANCE zu $s \leftarrow s + 1$ so haben wir den NAIVE-STRING-MATCHER.

Beim Boyer-Moore-Verfahren wird der Index s gemäß zweier Heuristiken im allgemeinen um **mehr als eins** heraufgesetzt.

Heuristik „Falsches Zeichen“

Nehmen wir an, dass der Vergleich von P mit $T[s+1..s+m]$ abgebrochen wurde, da $P[j] \neq T[s+j]$.

$T[s+j]$ ist das „falsche Zeichen“.

Kommt das falsche Zeichen überhaupt nicht in P vor, so können wir s um j erhöhen.

Kommt es (von rechts gelesen) erstmals an der Stelle i in P vor, so können wir immerhin um $j - i$ erhöhen (falls das positiv ist).

Heuristik „Gutes Suffix“

Nehmen wir an, dass der Vergleich von P mit $T[s+1..s+m]$ abgebrochen wurde, da $P[j] \neq T[s+j]$.

Wenn $j < m$, so wissen wir immerhin, dass $P[j+1..m] \sqsupset T[s+1..s+m]$. Das ist das „gute Suffix“.

Wir schreiben $x \sim y$ für $x \sqsupset y \vee y \sqsupset x$.

Wir können s um $m - \max\{k \mid 0 \leq k \leq m \wedge P[j+1..m] \sim P_k\}$ erhöhen.

Das Boyer-Moore Verfahren

Man kann die möglichen Shifts bei den beiden Heuristiken im Vorhinein ähnlich wie π berechnen.

ADVANCE wird als das Maximum der beiden Heuristiken implementiert.

In Zeile 8 bei erfolgreichem Match kommt nur die Heuristik „Gutes Suffix“ zum Einsatz, da es kein „falsches Zeichen“ gibt.

Es gibt m.W. keine rigorose Laufzeitanalyse des Boyer-Moore Verfahrens.

Bei grossen Alphabeten und langen Mustern verhält es sich in der Praxis sehr gut.

Emacs Ctrl-S arbeitet mit dem Boyer-Moore Verfahren

VIII. Geometrische Algorithmen

- Entscheidungsprobleme:
  - Schneiden sich zwei Geraden / Strecken / Kreise / Objekte?
  - Liegt ein Punkt im Inneren eines Dreiecks / Polygons / Kugel / Objekts?
  - Bilden n Punkte ein konvexes Polygon / Polyeder?
  - Liegen Punkte auf einer Geraden / in einer Ebene ?
  - Ist ein Winkel spitz oder stumpf?
- Berechnungsprobleme:
  - Konvexe Hülle
  - Projektion
  - Triangulierung
- Anwendungen:
  - Grafik (Spiele, VR, ...)
  - CAD/CAM
  - Robotik

Punkte, Vektoren, Strecken, Geraden

Ein Punkt in der Ebene ist durch sein Koordinatenpaar gegeben: $p = (x, y)$.

Wir identifizieren Punkte mit ihren Ortsvektoren.

Vektoren (und somit Punkte) kann man (komponentenweise) addieren, subtrahieren und mit Skalaren multiplizieren.

Gerade durch zwei Punkte in der Ebene:

$$p_1 p_2 = \{\alpha p_1 + (1 - \alpha) p_2 \mid \alpha \in \mathbb{R}\}$$

Strecke zwischen zwei Punkten in der Ebene:

$$\overline{p_1 p_2} = \{\alpha p_1 + (1 - \alpha) p_2 \mid 0 \leq \alpha \leq 1\}$$

Ein Punkt in $\overline{p_1 p_2}$ heißt auch konvexe Kombination von p_1 und p_2.

Eine Punktmenge M ist konvex, wenn für je zwei Punkte $p_1, p_2 \in M$ die gesamte Verbindungsstrecke in M liegt, also $\overline{p_1 p_2} \subseteq M$.

Kreuzprodukt

Das Kreuzprodukt zweier Vektoren $p_1 = (x_1, y_1)$ und $p_2 = (x_2, y_2)$ ist der Skalar

$$p_1 \times p_2 = x_1 y_2 - x_2 y_1$$

Es gilt:

$$p_1 \times p_2 = - p_2 \times p_1$$

Es gilt auch:

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$

Der Betrag von $p_1 \times p_2$ ist die Fläche des von p_1 und p_2 aufgespannten Parallelogramms.

Das Vorzeichen von $p_1 \times p_2$ ist positiv, wenn p_2 aus p_1 durch Drehen nach links (und Skalieren) entsteht.

Das Vorzeichen von $p_1 \times p_2$ ist negativ, wenn p_2 aus p_1 durch Drehen nach rechts (und Skalieren) entsteht.

Das Kreuzprodukt $p_1 \times p_2$ ist Null, wenn p_1 und p_2 linear abhängig sind.

Nach links oder nach rechts?

Gegeben sind drei Punkte p_0, p_1, p_2.

Man bewegt sich entlang $\overline{p_0 p_1}$ von p_0 nach p_1 und dann entlang $\overline{p_1 p_2}$ nach p_2.

Man soll feststellen ob man bei p_1 nach links oder nach rechts geht.

Man geht nach links, wenn $(p_1 - p_0) \times (p_2 - p_0) > 0$.

Man geht nach rechts, wenn $(p_1 - p_0) \times (p_2 - p_0) < 0$.

Man geht geradeaus oder rückwärts, wenn $(p_1 - p_0) \times (p_2 - p_0) = 0$.

Feststellen, ob sich zwei Strecken schneiden

Wir untersuchen jetzt die Frage, ob sich zwei gegebene Strecken in einem Punkt schneiden.

Zu einer Strecke $\overline{p_1 p_2}$ bilden wir die *bounding box* (begrenzendes achsenparalleles Rechteck):

Linke untere Ecke der *bounding box*: $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$, wobei
$\hat{x}_1 = \min(x_1, x_2), \hat{y}_1 = \min(y_1, y_2)$.

Rechte obere Ecke der *bounding box*: $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$, wobei
$\hat{x}_2 = \max(x_1, x_2), \hat{y}_2 = \max(y_1, y_2)$.

Oberflächlicher Test

Zwei Strecken $\overline{p_1p_2}$ und $\overline{p_3p_4}$ haben sicher keinen Punkt gemeinsam, wenn ihre *bounding boxes* disjunkt sind, d.h., wenn

$$(\hat{x}_2 \geq \hat{x}_3) \wedge (\hat{x}_4 \geq \hat{x}_1) \wedge (\hat{y}_2 \geq \hat{y}_3) \wedge (\hat{y}_4 \geq \hat{y}_1) \qquad (\star)$$

falsch ist.

Hier sind $(\hat{x}_3, \hat{y}_3)$ und $(\hat{x}_4, \hat{y}_4)$ die linke untere / rechte obere Ecke der *bounding box* von $\overline{p_3p_4}$.

Ist die Aussage $(\star)$ wahr so schneiden sich die *bounding boxes*; die Strecken können natürlich trotzdem disjunkt sein.

Entwurfsprinzip: *bounding box*

Oft muss man entscheiden, ob in einer Menge geometrischer Objekte welche überlappen, oder sich überdecken.

Beispiel: Szenenwiedergabe (*rendering*).

Es bietet sich an, für alle Objekte *bounding boxes* mitzuführen, oder zu berechnen. Durch schnelle Tests auf den *bounding boxes* kann man die meisten Fälle von vornherein ausschließen und muss (möglicherweise aufwendige) exakte Tests nur auf wenigen Objekten durchführen.

Feststellen, ob sich zwei Strecken schneiden

Gegeben sind zwei Strecken $\overline{p_1p_2}$ und $\overline{p_3p_4}$. Wir wollen überprüfen, ob $\overline{p_1p_2} \cap \overline{p_3p_4} \neq \emptyset$.

- Zunächst überprüft man, ob sich die *bounding boxes* schneiden.
- Falls nein, so schneiden sich die Strecken auch nicht.
- Falls ja, so schneiden sich die Strecken genau dann, wenn $\overline{p_1p_2}$ die Gerade p_3p_4 schneidet und $\overline{p_3p_4}$ die Gerade p_1p_2 schneidet.
- Die Strecke $\overline{p_3p_4}$ schneidet p_1p_2 genau dann, wenn sich die Vorzeichen von $(p_3 - p_1) \times (p_2 - p_1)$ und $(p_4 - p_1) \times (p_2 - p_1)$ unterscheiden, oder eines oder beide Null ist (sind).

Beachte: Der Fall dass alle vier Punkte auf einer Geraden liegen, die Strecken aber nicht überlappen, wird durch den „oberflächlichen Test“ ausgeschlossen.

Überstreichen

Gegeben ist eine Menge S von n Strecken. Wir wollen feststellen, ob $a \cap b \neq \emptyset$ für zwei $a, b \in S$ mit $a \neq b$.

Wir verwenden das Entwurfsprinzip „Überstreichung“ (engl.: *sweeping*).

Eine gedachte senkrechte Gerade überstreicht dabei die gegebenen Daten von links nach rechts.

Dabei wird an Ereignispunkten nach Maßgabe des bisher Gesehenen eine dynamische Datenstruktur verändert.

Das Endresultat ergibt sich nach Überstreichen der gesamten Eingabe aus dem erreichten Zustand der Datenstruktur.

Anordnung der Strecken

Vereinfachende Annahme: Keine Strecke senkrecht, höchstens zwei Strecken schneiden sich in einem Punkt.

Seien a, b Strecken in S und x eine Position der Überstreichungsgeraden. Wir schreiben $a <_x b$ wenn sowohl a, als auch b, die Gerade durch x schneiden und zwar a weiter unten als b.

$$a <_x b \Leftrightarrow \exists y_1, y_2 . y_1 < y_2 \wedge (x, y_1) \in a \wedge (x, y_2) \in b$$

Schneiden sich a und b, so gibt es eine Position der Überstreichungsgeraden x_0, sodass a, b in x_0 unmittelbar aufeinanderfolgen.

Wie stellt man fest, ob $a <_x b$?

- Zunächst schauen wir, ob a und b von der Geraden durch x getroffen werden. Das erfordert nur Größenvergleiche reeller Zahlen. Wenn nicht, so sind a und b bzgl. $<_x$ unvergleichbar.
- Nun können wir die Schnittpunkte der duch a, b definierten Geraden mit der Geraden durch x bestimmen und die y-Koordinaten vergleichen. Durchmultiplizieren mit Hauptnenner vermeidet Divisionen.
- Alternativ können Kreuzprodukte verwendet werden, siehe Übung.

Feststellen, ob sich irgendzwei Strecken in S schneiden

Um festzustellen, ob sich irgendzwei Strecken in S schneiden...

- ...halten wir in einer Datenstruktur die Ordnung $<_x$ in der aktuellen Position x der Überstreichungsgeraden fest;
- der Zustand der Datenstruktur wird nur verändert an x-Koordinaten von Anfangs- oder Endpunkten von Strecken in S; die Ereignispunkte sind also diese x-Koordinaten;
- werden an einem Ereignispunkt x zwei Strecken unmittelbar aufeinanderfolgend (in der Ordnung $<_x$), so überprüfen wir, ob sie sich schneiden;
- falls das passiert, so gibt es zwei sich schneidende Strecken in S,
- passiert es nie, so schneiden sich keine zwei Strecken in S.

Verwendete Datenstruktur

- An den Ereignispunkten verändert sich die Ordnung $<_x$ nur dadurch, dass Strecken eingefügt oder entfernt werden.
- Die relative Position von Strecken zueinander ändert sich nicht, denn das könnte nur passieren, wenn sie sich schneiden und das würden wir (hoffentlich) rechtzeitig merken.
- Wir können daher die Ordnung $<_x$ in einem Rot-Schwarz-Baum speichern, der die folgenden Operationen unterstützen muss:
  - Insert(a, x): Einfügen der Strecke a gemäß $<_x$, falls a von der Geraden durch x geschnitten wird.
  - Above(a): Liefert den unmittelbaren Nachfolger von a in der durch den Baum definierten Ordnung (falls er existiert; Nil sonst.)
  - Below(a): Liefert den unmittelbaren Vorgänger von a in der durch den Baum definierten Ordnung.
  - Delete(a): Löscht die Strecke a aus der Datenstruktur.

Pseudocode für das Verfahren

```
TEST-INTERSECT(S)
1   (p_1, ..., p_k) ← Sortierung der Endpunkte in S nach x-Koordinaten und
                      bei gleicher x-Koordinate nach y-Koordinaten.
2   Erzeuge leeren Rot-Schwarz-Baum für Strecken
3   for i ← 1 to n/2 − 1 do
4       x ← x-Koordinate von p_i
5       if p_i = Linker Endpunkt der Strecke a then
6           INSERT(a, x)
7           if INTERSECT(a, ABOVE(a)) ∨ INTERSECT(a, BELOW(a)) then return TRUE
8       else
9           if INTERSECT(ABOVE(a), BELOW(a)) then return TRUE
10          DELETE(a)
11  return FALSE
```

Pseudocode für das Verfahren

Beachte: In Zeile 1 müssen Punkte, die mehrfach als Endpunkte auftreten, auch mehrfach aufgeführt werden.

Am besten erreicht man das, indem man p als Paar eines Punktes und einer Strecke implementiert.

Nur dann geht auch das Auffinden von a in $O(1)$.

Die Prozedur INTERSECT(a, b) soll entscheiden, ob sich a und b schneiden. Ist eines der beiden Argumente NIL, so soll FALSE herauskommen.

Verifikation

Die folgenden Invarianten gelten jeweils nach Zeile 10:

- Die Datenstruktur enthält genau die Ordnung $<_x$ eingeschränkt auf die bisher gesehenen Punkte und abzüglich derer, deren rechter Endpunkt x ist.
- Keine zwei Strecken schneiden sich in einem Punkt mit x-Koordinate kleiner oder gleich x
- Schneiden sich zwei im Baum befindliche Strecken nach x, so folgen sie nicht unmittelbar aufeinander.

Wir liefern nur dann TRUE zurück, wenn tatsächlich eine Überschneidung vorliegt.

Wir liefern nur dann FALSE zurück, wenn wir alle Punkte in der for-Schleife verarbeitet haben. in diesem Falle implizieren die Invarianten, dass keine Überschneidungen vorliegen.

Konvexe Hülle

Die konvexe Hülle einer Punktmenge M ist die kleinste konvexe Punktmenge, die M umfasst.

Zur Erinnerung: X ist konvex, wenn mit $x, y \in X$ auch $\overline{xy} \subseteq X$.

Ist M eine endliche Menge, so ist die konvexe Hülle ein (konvexes) Polygon, dessen Ecken eine Teilmenge von M bilden.

Das Problem „Bestimmung der konvexen Hülle“ besteht darin, zu gegebener (endlicher) Punktmenge M der Größe n eine Liste von Punkten $p_1, \ldots, p_h$ zu berechnen, sodass die p_i die Ecken der konvexen Hülle von M in der richtigen Reihenfolge sind.

Manchmal verzichtet man auf die Reihenfolge.

Anwendungen der komplexen Hülle

- Größter Durchmesser. Um die am weitesten entfernt liegenden Punkte einer Liste von Punkten zu finden, kann man die Suche auf die konvexe Hülle beschränken. Den größten Durchmesser eines konvexen Polygons kann man in Zeit $O(n)$ bestimmen.
- Oberflächlicher Test für Überlappung. Überlappen die konvexen Hüllen nicht, so auch nicht die Punktmengen.

Bestimmung der konvexen Hülle

- **Inkrementelles Verfahren.** Sukzessives Hinzufügen von Punkten zur konvexen Hülle der bisher gesehenen Punkte. Wenn naiv implementiert: $\Theta(n^2)$.
- **Graham's scan.** Man ordnet die Punkte nach Polarwinkel und überstreicht dann mit einer rotierenden Halbgeraden. Als Datenstruktur kommt ein Keller zur Anwendung: $O(n \log n)$.
- **Jarvis' march.** (auch *gift wrapping*) man „umwickelt" die Punktmenge von unten her links und rechts: $O(nh)$ wobei h die Größe der konvexen Hülle ist.
- **Divide-and-conquer.** Komposition der konvexen Hülle zweier konvexer Polygone: $\Theta(n \log n)$.
- **Bestes Verfahren:** Ähnlich wie Medianfindung in Linearzeit: $O(n \log h)$.

Graham's scan

- Wir wählen irgendeinen Punkt p_0 aus M und sortieren die übrigen nach Polarwinkel: $p_1, \ldots, p_{n-1}$.
- Wir verarbeiten die Punkte in dieser Reihenfolge, also $p_0, p_1, p_2, \ldots$.
- Wir verwenden einen Keller (*stack*) dessen Inhalt immer gerade die konvexe Hülle der bisher gesehenen Punkte ist.
- Bildet ein neuer Punkt p mit den obersten beiden Punkten auf dem Keller eine Linkskurve, so wird p auf den Keller gelegt.
- Das gleiche gilt, wenn der Keller weniger als zwei Elemente enthält.
- Bildet ein neuer Punkt p mit den obersten beiden Punkten auf dem Keller eine Rechtskurve oder liegt geradeaus, so entfernen wir Punkte vom Keller bis eine Linkskurve vorliegt.
- Sind alle Punkte verarbeitet, so enthält der Keller die Ecken der konvexen Hülle.

Zur Verifikation muss man sich nur überlegen, dass die angekündigte Invariante tatsächlich erhalten wird.

Laufzeit von Graham's scan

- Das Verarbeiten des i-ten Punktes kann bis zu $i-2$ Operationen erfordern. Laufzeit $\Theta(n^2)$??
- Natürlich nicht. Jeder Punkt wird höchstens einmal auf den Keller gelegt und höchstens einmal wieder entfernt. Es finden also insgesamt nur $O(n)$ Kelleroperationen statt. (Aggregatmethode)
- Darüberhinaus sind $O(n)$ Kreuzprodukte und Vergleiche vorzunehmen.
- Die Laufzeit wird vom Sortierprozess dominiert: $O(n \log n)$.

Jarvis' march

- Wir befestigen einen Faden am Punkt mit der niedrigsten y Koordinate.
- Wir wickeln das eine Ende des Fadens rechts herum, das andere links herum, solange bis sich die beiden Enden oben treffen.
- Der nächste Eckpunkt ist jeweils der mit dem kleinsten Polarwinkel in Bezug auf den als letztes Hinzugefügten.
- Der Polarwinkelvergleich kann allein mit Kreuzprodukten ermittelt werden.

Laufzeit $O(nh)$.

Besser als Graham's scan wenn $h = o(\log n)$.

Verallgemeinerbar auf 3D.

Punkte mit kleinstem Abstand

Wir wollen aus einer Menge P von n Punkten die zwei Punkte mit minimalem Abstand bestimmen.

Abstand: von (x_1, y_1) und (x_2, y_2):

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Interessiert man sich nur für Abstandsvergleiche, dann kann man auch das Abstandsquadrat verwenden. Man braucht dann keine Quadratwurzel.

Anwendungsbeispiel (in 3D): Flugsicherung. Die jeweils am nächsten beieinanderliegenden Flugzeuge seien rot zu kennzeichnen.

Mit *divide and conquer* können wir diese Aufgabe mit $O(n \log n)$ Operationen lösen.

Divide and conquer Verfahren für minimalen Abstand

- Ist $n \leq 3$ so bestimmen wir die nächstliegenden Punkte direkt.
- Ist $n > 3$ so teilen wir P durch eine senkrechte Gerade l in zwei Teile P_L und P_R sodass gilt: $|P_L| = \lceil n/2 \rceil$ und $|P_R| = \lfloor n/2 \rfloor$ und $P = P_L \cup P_R$ und P_L, bzw. P_R enthält nur Punkte, die links von l oder auf l, bzw. rechts von l oder auf l liegen.
- Seien δ_L und δ_R die minimalen Punktabstände in P_L, bzw. P_R bestimmt durch rekursive Anwendung des Verfahrens. Weiter sei $\delta = \min(\delta_L, \delta_R)$.
- Ist der minimale Punktabstand in P selbst kleiner als δ, so wird er realisiert durch Punkte $p_L \in P_L$ und $p_R \in P_R$. Die Punkte p_L und p_R müssen daher in einem Streifen der Breite 2δ um l liegen.

Durchsuchen des Streifens

- Die y-Koordinaten von p_L und p_R unterscheiden sich auch um höchstens δ, d.h. p_L und p_R (so es sie gibt) müssen in einem $2\delta \times \delta$ Rechteck um l liegen.
- Solch ein Rechteck enthält höchstens 8 Punkte: Die $\delta \times \delta$-Hälfte, die links von l liegt enthält höchstens vier Punkte (alle Punkte in P_L haben ja Abstand $\geq \delta$), ebenso die $\delta \times \delta$-Hälfte rechts von l. Also 8 Punkte insgesamt.
- Sortiert man die Punkte im 2δ-Streifen um l nach aufsteigenden y-Koordinaten, so muss man jeden Punkt immer nur mit den 7 nächstgrößeren vergleichen. Findet man unter diesen keinen, der näher als δ liegt, dann auch nicht später.

Die Laufzeit dieses Verfahrens erfüllt $T(n) = 2T(n/2) + O(n \log n)$.

Da ja für das Aufteilen nach x-Koordinaten sortiert werden muss und für das Mischen nach y-Koordinaten.

Also $T(n) = O(n(\log n)^2)$

Optimierung der Laufzeit

Man sortiert die Punkte am Anfang nach x und nach y Koordinaten.

D.h. man erzeugt zwei Listen X und Y jeweils der Länge $|P|$. Die Liste X (bzw. Y enthält die Punkte von P nach X-Koordinaten (bzw. y-Koordinaten) sortiert.

Aus diesen kann man in linearer Zeit ebensolche Listen gewinnen, die sich auf Teilmengen von P beziehen.

Damit erhält man die Laufzeit $O(n \log n)$ für das Verfahren.

*** ENDE DER VORLESUNG ***