Algo Zusammenfassung

Tim Nogga

Contents

Chapter 1

Einleitung

1.1 Insertionsort

Algorithm 1: Insertion-Sort

Input: int[] a
Output: a sortiert

```
for int j = 1; j < a.length; j++ do
    int x = a[j];
    int i = j - 1;
    while i ⩾ 0 a[i] > x do
        a[i + 1] = a[i];
        i = i - 1;
    a[i + 1] = x;
return a;
```

Theorem 1.1.1
Die Ausgabe von Insertionsort ist stets eine aufsteigend sortierte Permutation der Eingabe.

Theorem 1.1.2
Die Laufzeit von Insertionsort ist in $O(n^2)$

1.2 Größenordnungen

Definition 1.2.1: Größenordnungen

Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ zwei Funktionen

1. $f(n) \in O(g(n))$ falls $\exists c > 0, n_0 \in \mathbb{N}$ mit $f(n) \leqslant c \cdot g(n) \forall n \geqslant n_0$
2. $f(n) \in \Omega(g(n))$ falls $\exists c > 0, n_0 \in \mathbb{N}$ mit $f(n) \geqslant c \cdot g(n) \forall n \geqslant n_0$
3. $f(n) \in \Theta(g(n))$ falls $f(n) \in O(g(n))$ und $f(n) \in \Omega(g(n))$
4. $f(n) \in o(g(n))$ falls $\forall c > 0 \exists n_0 \in \mathbb{N}$ mit $f(n) \leqslant c \cdot g(n) \forall n \geqslant n_0$
5. $f(n) \in \omega(g(n))$ falls $\forall c > 0 \exists n_0 \in \mathbb{N}$ mit $f(n) \geqslant c \cdot g(n) \forall n \geqslant n_0$

Note:-

Es gilt für die exakten schranken $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, wenn $f(n) \in o(g(n))$ und umgekehrt, wenn $f(n) \in \omega(g(n))$

Chapter 2

Methoden zum Entwurf von Algorithmen

2.1 Divide and Conquer

2.1.1 Binary Search

Algorithm 2: Binary Search

Input: int[] a, int x, int l, int r
Output: Bool

if $l > r$ **then**
  return false;
int m = $\lfloor \frac{l+r}{2} \rfloor$;
if $a[m] < x$ **then**
  return binarySearch(a, x, m + 1, r);
if $a[m] > x$ **then**
  return binarySearch(a, x, l, m - 1);

Theorem 2.1.1
Die Laufzeit von Binary Search ist in $O(\log n)$

Note:-
Erklärung:

- $l > r$ bedeutet, dass das gesuchte Element nicht in der Liste ist
- $a[m] < x$ bedeutet, dass das gesuchte Element rechts von m ist
- $a[m] > x$ bedeutet, dass das gesuchte Element links von m ist

Geht rekurisv weiter, bis das Element gefunden ist.

2.1.2 Merge Sort

Algorithm 3: Merge Sort

```
Input: int[] a, int l, int r
Output: a sortiert
if l < r then
    int m = ⌊(l+r)/2⌋;
    mergeSort(a, l, m);
    mergeSort(a, m + 1, r);
    merge(a, l, m, r);
```

Algorithm 4: Merge

```
Input: int[] a, int l, int m, int r
Output: a sortiert
int [] l = new int[m - l + 1];
int [] r = new int[r - m];
for int i = 0; i < l.length; i++ do
    l[i] = a[l + i];
for int i = 0; i < r.length; i++ do
    r[i] = a[m + 1 + i];
int iL = 0;
int iR = 0;
int iA = l;
while iL < m-l + 1 and iR < r - m do
    if l[iL] ⩽ r[iR] then
        a[iA] = l[iL];
        iL++;
        iA++;
    else
        a[iA] = r[iR];
        iR++;
        iA++;
    while iL < m-l + 1 do
        a[iA] = l[iL];
        iL++;
        iA++;
    while iR < r - m do
        a[iA] = r[iR];
        iR++;
        iA++;
```

Theorem 2.1.2

Die Laufzeit von Merge Sort ist in $O(n \log n)$

2.1.3 Divide-and-Conquer am Beispiel von Strassen

Note:-

Wenn man die Laufzeit von Strassen zu Simpleren Divide and Conquer Algorithmen vergleicht, wie Simple Product, ist die Laufzeit von Strassen besser

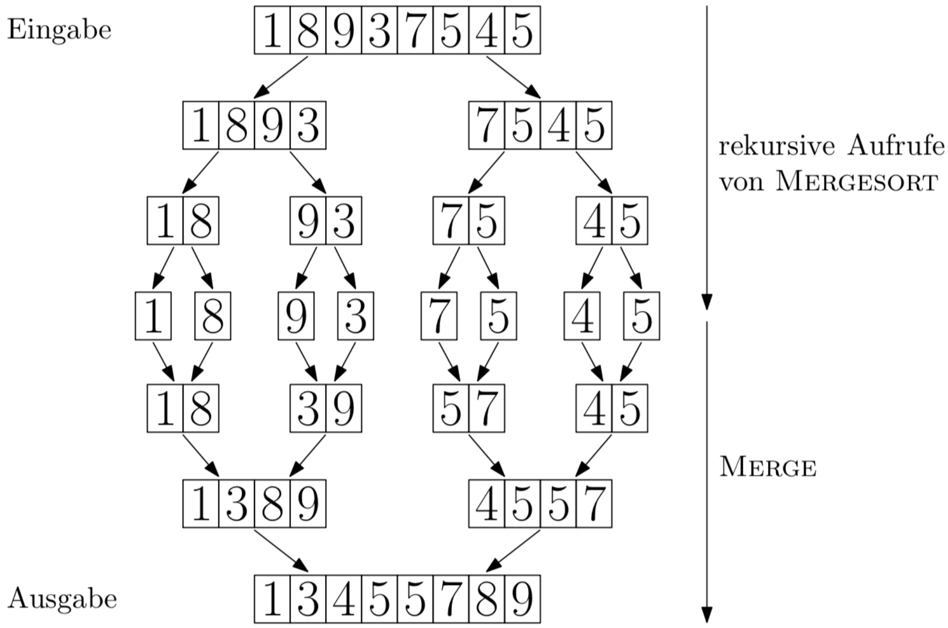

Figure 2.1: Mergesort

Theorem 2.1.3
Die Laufzeit von Strassen ist in $O(n^{\log_2 7})$

Proof: Siehe Seite 25 skript(Kein Bock das zu Texen) folgt aber aus $T(n) = 7T(n/2) + O(n^2)$ ☺

2.1.4 Master Theorem

Theorem 2.1.4 Mastertheorem
Seien $a \geqslant 1, b > 1$, Konstanten und sei f: $\mathbb{N} \to \mathbb{R}_{\geqslant 0}$ eine Funktion. Ferner sei $T : \mathbb{N} \to \mathbb{R}$ definiert durch $T(1) = \Theta(1)$ und

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

für alle $n > 1$. Die Funktion T kann wie folgt beschrieben werden:

- $T(n) = O(n^{\log_b a})$ falls $f(n) = O(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0$
- $T(n) = \Theta(n^{\log_b a} \log n)$ falls $f(n) = \Theta(n^{\log_b a})$
- Falls $T(n) = \Omega(n^{\log_b a + \epsilon})$ für eine Konstante $\epsilon > 0$ und falls $af(\frac{n}{b}) \leqslant cf(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n, dann gilt $T(n) = \Theta(f(n))$

Note:-
Im Prinzip werden nur die Funktionen n und $n^{\log_b a}$ verglichen. Im ersten Fall wächst f langsamer im zweiten gleich schnell. Verglichen zum ersten Fall führt das zu einem zusätzlichen $\log n$ führt. Im dritten Fall wächst f schneller als $n^{\log_b a}$, also ist die Lösung $\Theta(f(n))$

Lemma 1. *Seien $a \geqslant 1, b > 1$, Konstanten und sei f: $\mathbb{N} \to \mathbb{R}_{\geqslant 0}$ eine Funktion. Sei $T(n) = \Theta(O(n^{\log_b(a)})) + g(n)$ mit $g(n) \overset{\text{def}}{=} \sum_{i=0}^{\log_b(n)-1} a^i f(\frac{n}{b^i})$ Betrachte die Funktion beschränkt auf Werte von n mit $n = b^j$ für $j \in \mathbb{N}$.*

- *Falls f(n) $= O(n^{\log_b(a)-\epsilon})$ für eine Konstante $\epsilon > 0$, dann $g(n) = O(n^{\log_b(a)})$*
- *Falls f(n) $= \Theta(n^{\log_b(a)})$, dann $g(n) = \Theta(n^{\log_b(a)} \log n)$*

- *Falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und falls $af(\frac{n}{b}) \leqslant cf(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n, dann gilt $g(n) = \Theta(f(n))$*

Note:-

Die Funktion g(n) ist die Summe der Kosten der rekursiven Aufrufe.
Das Lemma beschreibt das Mastertheorem für den Fall, dass n eine Potenz von b ist. Die höhe des Rekursionbaumes beträgt $\log_b(n)$ Logischerweise ist die gesammte Laufzeit die Summe über alle Knoten für jedes Level, wobei sich die anzahl an Knoten mit der höhe Skaliert mit a^h hinzu kommt dannn noch das die Kosten pro level noch mit der Summe $\sum_{i=0}^{h-1} f(\frac{n}{b^i})$ gegeben ist. Also ensteht daher die Formel für die gesammt Laufzeit mit $\Theta(a^h) + \sum_{i=0}^{h-1} f(\frac{n}{b^i})$

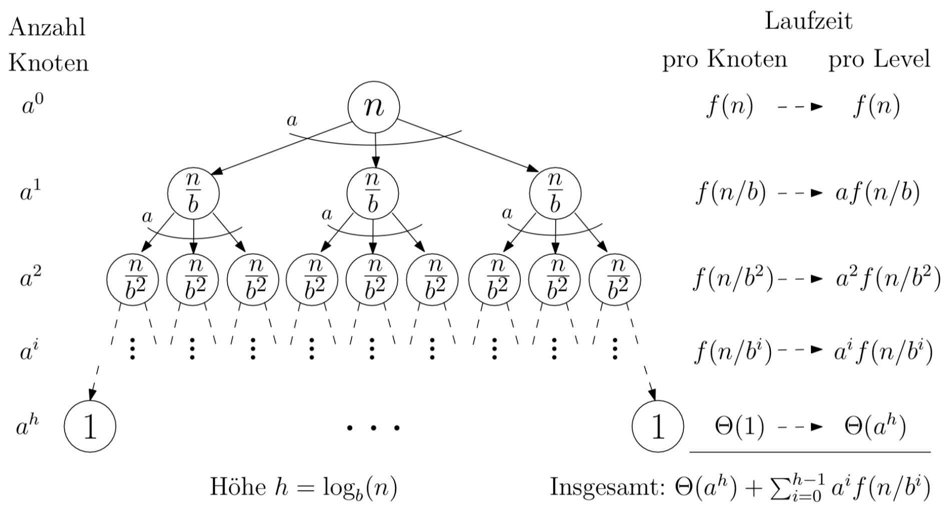

Figure 2.2: Visualized Lemma 1

2.2 Greedy Algorithmen

Definition 2.2.1: Greedy Algorithmen

Greedy Algorithmen sind Algorithmen die immer den besten lokalen Schritt wählen.

Example 2.2.1 (Wechselgeldproblem)

Das Wechselgeldproblem stellt die Münzen schritt für schritt zusammen, die am nächsten an den Restbetrag herankommen. Wie viel noch fehlt wird sich in der Variable z' gemerkt.

Theorem 2.2.1

Der Greedy Algorithmus löst das Wechselgeldproblem optimal.(Was ist wenn die Währung doof ist, siehe nächste Note)

Note:-

Für richtige Währungen wenn man Quatsch Währungen wählt lässt sich schnell ein gegenbeispiel Konstruieren z.B. 1,3,4 jezt 6 als Betrag. Also wird halt die 4 gewählt weil die am größten ist dementsprechend wird in den nächsten 2 Schritten 2 mal die 1 gewählt, aber die 3 und 3 wären besser gewesen.

Ich mach einfach mal nen beweis für die Vibes dazu, keine Ahnung gerade Bock drauf steht aber auch so im Skript(in Schöner siehe Seite 29).

Proof. Sei $z \in \mathbb{N}$ ein beliebiger Betrag, welcher genau erreicht werden soll. Für $i \in M := \{1, 2, 5, 10, 20, 50, 100, 200\}$ mit $x_i \in \mathbb{N}0$ Wegen der Def von Greedy Algorithmen, das immer der Lokal beste Schritt gewählt wird folgt das die ungleichung mit $i \in M, i > 1$ gilt

$$\sum_{j \in M, j < i} j x_j < i$$

Das j ist hierbei die Münze die gerade betrachtet wird, diese ist immer kleiner als i, da sonst das i gewählt werden würde, was gegen die Defenition von Greedy Algorithmen verstößt, da es eine bessere Lösung gibt.
Zusammen mit der Ausgangsbedingung das der zu erreichende Betrag z immer $\sum_{i \in M} i x_i$ ist. Nun lässt sich hierdrüber folgende Induktion aufbauen.
$z = 1$ trivialerweise gilt die Aussage.
Die Aussage gilt für alle $z \in \mathbb{N}$ mit $z > 1$ betrachte

$$\sum_{j \in M, j < i} j x_j < i$$

für ein $i\ maximal \leqslant z$ muss mindestens eine Münze vom Wert i enthalten sein, da sonst der Betrag nicht erreicht werden kann.
Die Behauptung folgt auch für $z' = z - i$ da $z' \leqslant z$ gilt.
Also ist die Lösung vom Greedy Algorithmus mit der Ungleichung erfüllt.
Nun lässt sich Annehmen, das sich eine optimale Lösung finden lässt. Wenn man alle Münzen in M durchgeht lässt sich immer jede Münze durch eine Kombination von Münzen mit kleinerem Wert ersetzen.
Formaler sei $y_i \in \mathbb{N}_0$ mit $\sum_{i \in M} i y_i = z$ und kleinstmöglicher Zahl $\sum i \in M y_i$
Sei die Lösung in Optimalerweise $x_i \in \mathbb{N}_0$ Zum Beispiel gilt $y_1 \leqslant 1$, da zwei 1 Cent Münzen durch eine 2 Cent Münze ersetzt werden können.
Daraus folgt die ungleichung $1 \cdot y_1 < 2$
Analog folgt für $y_2, y_2 \leqslant 2$, da sich 3 2 Cent Münzen durch eine 5 Cent Münze und eine 1 Cent Münze ersetzen lassen.
Daraus folgt die ungleichung $1 \cdot y_1 + 2 \cdot y_2 < 5$
$\longrightarrow$ Dies gilt für alle $i \in M$ ☺

2.2.1 Optimale Auswahl von Aufgaben

Note:-

Intervall Scheduling: Jede Aufgabe hat eine Startzeit $s_i \geqslant 0$ und eine Endzeit f_i, wobei $s_i < f_i$ gilt.
Zudem steht ein Prozesor zur verfügung der nur eine Aufgabe gleichzeitig bearbeiten kann.
Die Aufgabe ist es nun eine möglichst große Teilmenge von Aufgaben zu finden, die sich nicht überschneiden.

Definition 2.2.2: Intervall Scheduling Formales Ziel

Gescucht ist eine Teilmenge $S' \subseteq S$, sodass für jedes $i, j \in S'$ gilt $i \neq j$ und $[s_i, f_i) \cap [s_j, f_j) = \emptyset$

Note:-

Es liegt nahe das es einen Greedy Algorithmus gibt welcher das Problem löst(Im Skript kommen noch Algorithmen die nicht klappen)

Algorithm 5: Greedy Ende

$S^* = \emptyset$
while $S \neq \emptyset$ **do**
  Wähle die Aufgabe i mit dem frühesten Ende
  $S^* = S^* \cup \{i\}$
  $S = S \setminus \{i\}$ Keine Ahnung nicht im Skript aber kommt mir logisch vor
  Entferne alle Aufgaben die sich mit i überschneiden

Für den Beweis von den Algorithmus wird folgendes Lemma benötigt.

Lenma 2.2.1

Es sei S eine Menge von Aufgaben und es sei $i \in S$ eine Aufgabe mit dem frühesten Ende. Dann gibt es eine Optimale Auswahl $S' \subseteq S$ von paarweise nicht kollidierenden Aufgaben mit $i \in S'$

Note:-

Das Lenma sagt basically nur aus das es eine optimale Lösung gibt.

Proof. Genauer Beweis im Skript seite 32. Aber man definiert sich eine Optimale Menge S^* in diese fügt man dann eine Aufgabe i ein. Die Menge S^* ist dann immer noch optimal und es gibt keine Überschneidungen, da i vor oder gleich mit j der vorher kürzesten Aufgabe aufhört. j hat sich halt obviously auch nicht überschnitten, weil dann wär es nicht optimal gewesen. 😄

Theorem 2.2.2

Der Algorithmus GreedyEnde wählt für jede Instanz eine größtmög- liche Menge von paarweise nicht kollidierenden Aufgaben aus.

Proof. Lässt sich recht entspannt über ne invariante Zeigen:
In Zeile 2 gilt immer das S^* sich zu einer optimalen Lösung erweitert.
In der ersten Iteration gilt die Invariante, da S^* leer ist und S noch alle Aufgaben enthält. Sei nun die Invariante zu Beginn eines Schleifendurchlaufs erfüllt. Dann gibt es eine optimale Auswahl $\hat{S} \subseteq S^* \cup S$ von Aufgaben, die alle Aufgaben aus S^* und gegebenenfalls zusätzliche Aufgaben aus S enthält. Außerdem kollidieren Aufgaben aus S^* nicht mit Aufgaben aus S. Das heißt, bei der Frage, welche Aufgaben aus S den Aufgaben aus S^* noch hinzugefügt werden müssen, um eine optimale Auswahl $\hat{S}$ zu erhalten, spielen die Aufgaben aus S^* keine Rolle. Die Menge $\hat{S} \setminus S^*$ dieser Aufgaben ist eine größtmögliche Teilmenge von paarweise nicht kollidierenden Aufgaben aus S. Gemäß Lenma 2.2.1 existiert eine solche Menge, die die Aufgabe $i \in S$ mit dem frühesten Fertigstellungszeitpunkt f_i aller Aufgaben aus S enthält. Genau diese Aufgabe fügt der Greedy-Algorithmus am Ende der Menge S^* hinzu. Anschließend werden aus S alle Aufgaben entfernt, die mit dieser Aufgabe i kollidieren. Dies garantiert zusammen, dass auch am Anfang der nächsten Iteration die Invariante wieder erfüllt ist. 😄

Definition 2.2.3: Laufzeit Greedy Ende

Die Laufzeit des Algorithmus GreedyEnde beträgt $O(n \log n)$, wobei $n = |S|$ die Anzahl der Aufgaben in der Eingabe bezeichnet. Sind die Aufgaben bereits aufsteigend nach ihrem Fertigstellungszeitpunkt sortiert, so beträgt die Laufzeit O(n).

Note:-

Es läuft halt n mal durch und man kann es in $O(n \log n)$ sortieren.

2.2.2 Rucksackproblem mit Teilbaren Aufgaben

Das Rucksack, wie wir es Behandeln (anscheinend kommt mehr in Algo 2) besteht aus einer Menge von Objekten $O = \{1, \ldots, n\}$, wobei jedes Objekt $i \in O$ ein Gewicht $w_i \in \mathbb{N}$ und einen Nutzen $p_i \in \mathbb{N}$ besitzt.
Es soll eine Teilmenge der Objekte gefunden werden, die in den Rucksack passt und unter dieser Bedingung maximalen Nutzen besitzt. Jetzt wo es ein bisschen anders ist:
Wir dürfen das Objekt Teilen für den maximalen Nutzen.

Algorithm 6: Greedy Rucksack

Sortiere die Objekte nach dem Nutzen pro Gewicht p_i/w_i in absteigender Reihenfolge.
for *int i = 1; i ⩽ n; i++* **do**
  x[i] = 0;
int i = 1;
while $t > 0$ *and* $i \leqslant n$ **do**
  if $t \geqslant w_i$ **then**
    $x_i = 1$
    $t = t - w_i$
  else
    x[i] = t/w[i], t = 0
  i++;
return $x_1, \ldots, x_n$

Theorem 2.2.3

Der Algorithmus GreedyRucksack liefert für jede Instanz eine optimale Lösung in $O(n \log n)$ Zeit.

Proof. Zur Laufzeit: Greedy Rucksack wird von den Sortieren Dominiert, dies läuft über Merge-Sort in $O(n \log n)$. Die Schleifen können jeweils nur n mal durchlaufen, was die laufzeit von $O(n \log n)$ zeigt.
Zur Korrektheit: Da die Lösung trivial ist, wenn alle Objekte in den Rucksack passen wird im folgenden nur der andere Fall betrachtet mit $t < \sum_{i=1}^{n} w_i$.
Es gibt nun ein $i \in \{1, \ldots, n\}$ mit $x_1 = \cdots = x_{i-1} = 0$ $x < 1$ und $x_{i+1} = \cdots = x_n = 0$
Greedy Rucksack füllt natürlich auch den Rucksack. Jede Optimale Lösung muss den Rucksack auch ausfüllen, das heißt es gilt $\sum_{i=1}^{n} x_i^* w_i = t$.
Es wird nun gezeigt, dass sich die Lösung von x^* in die Lösung x von Greedy Rucksack umwandeln lässt ohne die Nutzen zur veringern.
Jetzt wird der kleinste Index i genommen $x_i^* < 1$ und $x_{i+1}^* = \cdots = x_n^* = 0$. Hier gilt jetzt $x^* = x$ muss mir nochmal anschauen warum.TODO Es gibt einen Index $j > i$ mit $x_j^* > 0$. Sei j der größte solcher Indexe. Die j lassen sich jetzt reduzieren während man das i gegenscaled. Wegen der Effizienz von i welche mindestens die von j ist verändert sich nicht der Nutzen 😄