Universität Bremen

FB3: Mathematik/Informatik

Prof. Dr. Daniel Neuen
Jens Schlöter

Wintersemester 2023/2024

Algorithmentheorie

Übungsblatt 1

Übung 1.1 (7 Punkte)

Ihr arbeitet als Analysten in einer Investmentfirma. Mit Hilfe von Simulationen seid Ihr in der Lage, die Preisentwicklung der Aktie eines Unternehmens für die nächsten n Tage zu berechnen. Ihr habt also Daten zur Verfügung, welche beispielhaft die folgende Form haben:

Tag	0	1	2	3	4	5	6	7	...
Änderung (in \$)	3	7	-4	-8	10	-2	4	-3	...

Das Array enthält einen Eintrag der für jeden Tag den Gewinn bzw. Verlust zur Zeit des Börsenschlusses des zugehörigen Tages angibt. Wir nehmen an, dass das Array nicht nur negative Zahlen enthält. Eure Aufgabe ist es, auf Basis dieser Daten, genau einen der nächsten n Tage auszuwählen, um morgens die Aktie des Unternehmens zu kaufen. Außerdem sollt Ihr genau einen dieser n Tage auswählen, um die Aktie am Abend wieder zu verkaufen. Dabei sollt Ihr einen möglichst hohen Profit erzielen. Euer Ziel ist es also, zwei Indizes $i \leq j$ mit dem bestmöglichen Profit zu finden, d.h., Ihr sucht i und j, so dass $\sum_{k=i}^{j} \texttt{A}[k]$ maximiert wird.

Gebt den Pseudocode eines Algorithmus an, der das Problem löst. Argumentiert kurz, warum Euer Algorithmus korrekt ist und welche Laufzeit dieser erzielt.

Lösung Wir betrachten zunächst einen naiven Algorithmus, der durch alle Paare (i, j) mit $i \leq j$ iteriert, den Profit $p_{ij} = \sum_{k=i}^{j} A[k]$ des aktuellen Paares berechnet und das Paar mit dem bisher höchsten Profit speichert. Am Ende des Algorithmus wird das aktuell gespeicherte Paar zurückgegeben. Algorithmus 1 zeigt den zugehörigen Pseudocode.

Laufzeit Alle einzelnen Operationen des Algorithmus sind entweder Vergleiche, simple arithmetische Operationen, Zuweisungen oder Array-Zugriffe und haben damit eine Laufzeit von $\mathcal{O}(1)$. Das bedeutet insbesondere auch, dass eine einzelne Iteration der inneren Schleife eine Worst-Case Laufzeit von $\mathcal{O}(1)$ hat. Die asymptotische Worst-Case Laufzeit des Algorithmus wird also von der *Anzahl* der inneren Schleifenausführungen dominiert.

Die k-te Iteration der äußeren Schleife führt insgesamt $n-(k-1) = n-k+1$ Iterationen der inneren Schleife aus. Der Grund dafür ist, dass die Variable i in der k-ten Iteration den Wert $k-1$ hat und die innere Schleife über die Indizes von (einschließlich) $k-1$ bis $n-1$ iteriert. Die äußere Schleife wird insgesamt n mal ausgeführt, da über die Indizes von 0 bis $n-1$ iteriert wird. Die Anzahl der inneren Schleifenausführungen, und damit auch die Gesamtlaufzeit, ist also höchstens

$$\sum_{k=1}^{n} n-k+1 = \sum_{k=1}^{n} k = \frac{n^2+n}{2} \in \mathcal{O}(n^2).$$

Korrektheit Der Algorithmus terminiert: Alle Schleifen haben eine fixe obere Schranke von $n-1$ und die Schleifen-Variablen i und j werden in jeder Schleifen-Iteration um eins erhöht und nie verringert.

Es bleibt also zu argumentieren, dass der Algorithmus die korrekte Antwort zurückgibt. Der Algorithmus iteriert über alle möglichen Lösungen (i, j) mit $i \leq j$. Jede solche Lösung wird einmal in Zeile 10 des Algorithmus betrachtet. Die Variable p enthält an dieser Stelle den Profit $p_{ij} = \sum_{k=i}^{j} A[k]$ der möglichen Lösung (i, j). Die if-Anweisung von Zeile 10-13 stellt sicher, dass (b_1, b_2) immer die profitabelste bisher betrachtete Lösung ist. Der Algorithmus gibt also am Ende die profitabelste Lösung zurück.

Algorithmus 1 : Naiver Algorithmus für Aufgabe 1.1

```
   Input : Array A = [a_0, ..., a_n] mit a_j ∈ ℤ für alle j ∈ {1, ..., n}.
 1 i := 0;
 2 b_1 := −1;              /* Erster Index des bisher profitabelsten Paares */
 3 b_2 := −1;             /* Zweiter Index des bisher profitabelsten Paares */
 4 p* := −∞;                    /* Profit des bisher profitabelsten Paares */
 5 while i ≤ n − 1 do
 6 |   p := 0;                     /* Speichert Profit des aktuellen Paares */
 7 |   j := i;
 8 |   while j ≤ n − 1 do
 9 |   |   p := p + A[j];
10 |   |   if p > p* then
11 |   |   |   b_1 := i;
12 |   |   |   b_2 := j;
13 |   |   |   p* := p;
14 |   |   j := j + 1;
15 |   i := i + 1;
16 return (b_1, b_2)
```

Anmerkung: Mit einem Divide-and-Conquer Algorithmus ist es möglich, die Aufgabe mit einer Worst-Case Laufzeit von $\mathcal{O}(n \log n)$ zu lösen. Diese Laufzeit kann weiter verbessert werden auf $\mathcal{O}(n)$.

Übung 1.2 (7 Punkte)

Gegeben sei ein sortiertes n-elementiges Array $\texttt{A} = [a_0, a_1, ..., a_{n-1}]$ von natürlichen Zahlen, sowie eine natürliche Zahl k. Gesucht ist ein Indexpaar, dessen Elemente summiert k ergeben. Formal: Gib ein Indexpaar (i, j) aus, sodass $0 \leq i < j \leq n-1$ und $a_i + a_j = k$. Falls kein solches Indexpaar existiert, gib -1 aus.

Gebt den Pseudocode eines Algorithmus an, der das Problem mit linearer Worst-Case Laufzeit löst, d.h., die Worst-Case Laufzeit ist $an + b$ für geeignete Konstanten a, b. Argumentiert, warum Euer Algorithmus korrekt ist und tatsächlich die geforderte Laufzeit erzielt. Was ist die Best-Case Laufzeit?

Lösung Betrachte Algorithmus 2. Wir argumentieren zunächst über die Terminierung und Worst-Case Laufzeit des Algorithmus.

Laufzeit und Terminierung. In jeder Iteration der while-Schleife mit $A[i] + A[j] \neq k$ wird entweder i um eins erhöht oder j um eins dekrementiert. Das heißt, die Differenz $i - j$ wird in jeder solchen Iteration um eins kleiner. Initial gilt $i - j = n - 1$ und der Algorithmus terminiert

spätestens wenn $i - j = 0$. Der Algorithmus terminiert also entweder nach spätestens $n - 1$ Iterationen wegen $i - j = 0$ oder bereits vorher in Zeile 5 wenn das aktuelle Paar i, j die Bedingung $A[i] + A[j] = k$ erfüllt.

Da alle einzelnen Operationen des Algorithmus eine Laufzeit von $\mathcal{O}(1)$ haben, wird die Worst-Case Laufzeit von der Anzahl der Schleifendurchläufe dominiert. Wir haben bereits argumentiert, dass diese höchstens $n - 1$ ist. Damit ist die Worst-Case Laufzeit in $\mathcal{O}(n)$.

Korrektheit. Betrachte den Anfang einer beliebigen Iteration der while-Schleife und die aktuellen Variablen i und j. Die beiden Kernargumente für die Korrektheit sind:

(i) Wenn $A[i] + A[j] < k$, dann gibt es keine Lösung (ℓ, r) mit $A[\ell] + A[r] = k$ und $\ell \leq i$.

(ii) Wenn $A[i] + A[j] > k$, dann gibt es keine Lösung (ℓ, r) mit $A[\ell] + A[r] = k$ und $r \geq j$.

Wenn dies für alle Iterationen gilt, dann folgt, dass die while-Schleife entweder ein Tuple (ℓ, r) mit $A[\ell] + A[r] = k$ findet oder, dass kein solches Tuple existiert. Dies impliziert dann auch die Korrektheit des Algorithmus.

Es bleibt also zu argumentieren, dass (i) und (ii) für jede Iteration der while-Schleife gelten. Für die erste Iteration ist dies leicht zu sehen. Wenn $A[0] + A[n-1] < k$, dann muss eine Lösung (ℓ, r) mit $A[\ell] + A[r] = k$ auch $\ell > 0$ erfüllen, da $A[0]$ selbst zusammen mit dem größten Element $A[n-1]$ noch zu klein ist. Wenn $A[0] + A[n-1] > k$, dann muss eine Lösung (ℓ, r) mit $A[\ell] + A[r] = k$ auch $r < n - 1$ erfüllen, da $A[n-1]$ selbst zusammen mit dem kleinsten Element $A[0]$ noch zu groß ist.

Für spätere Iterationen können wir dann induktiv argumentieren. Wir wissen bereits, dass es keine Lösungen (ℓ, r) mit $A[\ell] + A[r] = k$ und $\ell < i$ oder $r > j$ für die aktuellen Werte i und j gibt. Wenn $A[i] + A[j]$ also zu klein ist, dann muss i erhöht werden, da j bereits den größtmöglichen Wert hat, der noch für eine Lösung in Frage kommt. Wenn $A[i] + A[j]$ zu groß ist, dann muss j kleiner werden, da i bereits den kleinstmöglichen Wert hat, der noch für eine Lösung in Frage kommt. Es folgt also, dass (i) und (ii) für alle Iterationen gelten und damit auch, dass der Algorithmus korrekt ist.

Best-Case Laufzeit. Wenn $A[0] + A[n-1] = k$, dann terminiert der Algorithmus bereits in der ersten Ausführung der Schleife in Zeile 5. Damit ist die Best-Case Laufzeit in $\mathcal{O}(1)$.

Algorithmus 2 : Algorithmus für Aufgabe 1.2

Input : Soriertes Array $A = [a_0, \ldots, a_n]$ mit $a_j \in \mathbb{N}$ für alle $j \in \{1, \ldots, n\}$. Gesuchte Zahl $k \in \mathbb{N}$.

```
1  i := 0;
2  j := n − 1;
3  while i < j do
4  |  if A[i] + A[j] = k then
5  |  |  return (i, j);
6  |  else if A[i] + A[j] < k then
7  |  |  i := i + 1;
8  |  else
9  |  |  j := j − 1;
10 return -1;
```

Übung 1.3 (6 Punkte)

Gegeben sei ein n-elementiges Array $\texttt{A} = [a_0, a_1, ..., a_{n-1}]$ von natürlichen Zahlen. Wir möchten analysieren wie viele Vergleiche (siehe Zeile 4 im Pseudocode aus der Vorlesung) Insertion Sort benötigt um `A` zu sortieren:

(a) Wie viele Vergleiche führt Insertion Sort *mindestens* aus um `A` zu sortieren? Begründe warum deine Antwort korrekt ist und beschreibe wie das Eingabearray aussehen muss, damit dieser Best Case erreicht wird.

(b) Wie viele Vergleiche führt Insertion Sort *maximal* aus um `A` zu sortieren? Begründe warum deine Antwort korrekt ist und beschreibe wie das Eingabearray aussehen muss, damit dieser Worst Case erreicht wird.

Lösung (a) In jeder Iteration der äußeren Schleife mit $i \leq n-1$ führt Zeile 4 mindestens einen Vergleich $A[j-1] > A[j]$ aus, unabhängig davon, wie das Eingabearray aussieht. Die äußere Schleife wird insgesamt $n-1$ mal ausgeführt (von $i = 1$ bis $i = n-1$). Die Anzahl der Vergleiche ist also mindestens $n-1$. Wenn das Array bereits aufsteigend sortiert ist, dann gibt der Vergleich $A[j-1] > A[j]$ immer `false` zurück. Die innere Schleife wird also nie ausgeführt und die Anzahl an Vergleichen pro Iteration der äußeren Schleife ist genau 1. Der Best-Case von $n-1$ Vergleichen wird also erreicht wenn die Eingabe bereits sortiert ist.

Lösung (b) Betrachte Iteration i der äußeren while-Schleife. Zu Beginn dieser Iteration gilt $j = i$ (vgl. Zeile 3 im Pseudocode). Wegen des Abbruchkriteriums $j > 0$ (und weil j in jeder Iteration der inneren Schleife um eins dekrementiert wird), wird die innere Schleife während Iteration i der äußeren Schleife höchstens i mal ausgeführt. Jede Iteration der inneren Schleife führt höchstens einen Vergleich in Zeile 4 durch. Damit ist die Anzahl der Vergleiche in Iteration i der äußeren Schleife höchstens i.

Die äußere Schleife wird höchstens $n-1$ mal ausgeführt. Damit ergibt sich, dass die Gesamtanzahl der Vergleiche höchstens

$$\sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

ist. Wenn das Eingabearray (strikt) absteigend sortiert ist, dann wird diese Anzahl an Vergleichen auch tatsächlich erreicht, weil der Vergleich $A[j-1] > A[j]$ in Zeile 4 dann immer `true` zurück gibt und die innere Schleife deshalb entsprechend oft ausgeführt wird.