technische universität dortmund

SoSe 2018

M. Buchin
A. Krivošija, D. Mezlaf
L. Pfahler, A. Rey

DAP2 – Heimübung 5

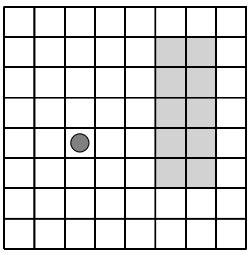
Ausgabedatum: 04.05.2018 — Abgabedatum: Mo. 14.05.2018 bis 12 Uhr

Abgabe:

Schreiben Sie unbedingt immer Ihren vollständigen Namen, Ihre Matrikelnummer und Ihre Gruppennummer auf Ihre Abgaben! Beweise sind nur dort notwendig, wo explizit danach gefragt wird. Eine Begründung der Antwort wird allerdings *immer* verlangt.

Aufgabe 5.1 (5 Punkte): (Teile und Herrsche)

Wir spielen Topfschlagen, aber ohne vom Sofa aufzustehen. Der Topf liegt in einem quadratischen Gebiet, das in kleine Quadrate aufgeteilt ist. Wir suchen ihn mit verbundenen Augen. Wir nennen die Seitenlänge n und nehmen an, dass n eine Zweierpotenz ist. Im folgenden Beispiel ist $n = 8$:



Bei der Suche hilft uns ein Orakel, das die Position des Topfes kennt, uns diese aber nicht direkt verrät. Das Orakel beantwortet aber Fragen, ob der Topf in einem bestimmten Rechteck liegt. Oben ist ein Rechteck eingezeichnet, das wir anfragen könnten, und in dem der Topf nicht liegt.

Ein Rechteck kann man zum Beispiel durch die linke untere und rechte obere Ecke beschreiben. Eine Anfage an das Orakel kann man also durch zwei Ecken (i_1, j_1) und (i_2, j_2) mit $1 \leq i_1 \leq i_2 \leq n$ und $1 \leq j_1 \leq j_2 \leq n$ ausdrücken. Die Antwort ist dann ja, wenn sich der Topf an einer Position (i, j) mit $i \in \{i_1, \ldots, i_2\}$ und $j \in \{j_1, \ldots, j_2\}$ befindet, und nein sonst.

Wir möchten den Topf jetzt mit möglichst wenigen Anfragen an das Orakel finden.

a) Entwerfen Sie einen Teile-und-Herrsche Algorithmus mit der Name `Topfschlagen`, der den Topf findet, und beschreiben Sie ihn mit eigenen Worten. Die volle Punktzahl erreicht ein Algorithmus, der mit $\mathcal{O}(\log n)$ Anfragen an das Orakel auskommt und den Topf immer findet.

b) Geben Sie eine Implementierung Ihres Algorithmus in Pseudocode an. Das Orakel kann dabei mit `Orakel`(i_1, j_1, i_2, j_2) aufgerufen werden und antwortet mit 1, wenn der Topf im angegebenen Rechteck liegt, und mit 0, wenn der Topf dort nicht liegt.

c) Beweisen Sie die Korrektheit Ihres Algorithmus.

Lösung:

a) Da von uns ein Algorithmus mit der Laufzeit $O(\log n)$ erwartet wird, bedeutet das dass unsere Teile und Herrsche Algorithmus auf jeder von $\log n$ Ebenen der Rekursion nur konstante Anzahl der Aufrufe von `Orakel`. Es gibt mehrere möglichen Aufteilungen des Eingabearrays der Größe $n \times n$. Wegen Einfachkeit werden wir immer unseres zweidimensionalen Eingabearray ins vier Teilarrays aufteilen (die Länge und Breite von Eingabearrays halbieren), auf jedem davon `Orakel` aufrufen, und weitere rekursive Aufrufe nur am Array fortfahren, aud dem das `Orakel` als Rückgabe `1` zurückgibt. Die Rekursion bricht beim Aufruf eines 1×1-Arrays. Wenn wir als Eingabearray ein 1×1-Array erhalten, sollten wir den Wert `Orakel`$(1, 1, 1, 1)$ zurückgeben, was in konstanter Laufzeit erfolgt.

Wir können annehmen, dass n eine 2er Potenz ist, d.h. in jedem rekursiven Aufruf wird die Länge und die Breite des zweidimensionalen Arrays halbiert. Das kann man höchstens $\log_2 n$ mal tun und in jeder Rekursionsebene hat man höchstens vier Orakel-Aufrufe. Die Rekursionsabbruchbehandlung eine konstante Anzahl der Rechenschritte verlangt. Deswegen ist die genutzte Laufzeit dieses Algorithmus mit $O(\log n)$ begrenzt.

b) Jetzt ist die obige Beschreibung einfach in der Pseudocode umzuwandeln, wobei der erste Aufruf **Topfschlagen(1,1,n,n)** ist.

Topfschlagen(**int** i_1, **int** j_1, **int** i_2, **int** j_2):

```
1.  if i_1 = i_2 und j_1 = j_2 then
2.      if Orakel (i_1, j_1, i_1, j_1) then
3.          return (i_1, j_1)
4.      else
5.          return Topf nicht gefunden
6.  else
7.      k ← ⌊(i_1 + i_2)/2⌋
8.      ℓ ← ⌊(j_1 + j_2)/2⌋
9.      if Orakel(i_1, j_1, k, ℓ)= 1 then
10.         return Topfschlagen(i_1, j_1, k, ℓ)
11.     if Orakel(k + 1, j_1, i_2, ℓ)= 1 then
12.         return Topfschlagen(k + 1, j_1, i_2, ℓ)
13.     if Orakel(i_1, ℓ + 1, k, j_2)= 1 then
14.         return Topfschlagen(i_1, ℓ + 1, k, j_2)
15.     if Orakel(k + 1, ℓ + 1, i_2, j_2)= 1 then
16.         return Topfschlagen(k + 1, ℓ + 1, i_2, j_2)
```

c) **Behauptung:** Algorithmus **Topfschlagen** findet den Topf im zweidimensionalen gegebenen Array der Größe $n \times n$ korrekt, für alle $n \in \mathbb{N}$.

(I.A.) Wenn $n = 1$ ist, sind die beide Koordinaten der linken unteren und der rechten oberen Ecke gleich ($i_1 = i_2$ und $j_1 = j_2$). Wenn das der Fall ist, ist die Bedingung in der Zeile 1 korrekt. Wenn im Feld (i_1, j_1) der Topf liegt, wird dieser Feld in der

Zeile 3 zurückgegeben, sonst wird es berichtet, dass der Topf nicht vorhanden ist. Die Behauptung ist deswegen für Induktionsanfang korrekt.

(I.V.) Es gelte dass der Algorithmus **Topfschlagen** den Topf korrekt findet, wenn das Eingabearray der Größe $n \times n$ ist, für alle natürlichen Zahlen n die kleiner als beliebige aber feste $n_0 \in \mathbb{N}$, $n_0 > 1$.

(I.S.) Da $n_0 > 1$ ist, ist die Bedingung in der Zeile 1 nicht erfüllt. Seien $k = \lfloor (i_1 + i_2)/2 \rfloor$ und $\ell = \lfloor (j_1 + j_2)/2 \rfloor$ die Koordinaten der mittleren Felder im Array. Wir teilen das Array der Größe $n_0 \times n_0 = (i_2 - i_1 + 1) \times (j_2 - j_1 + 1)$ in vier Subarrays, die mit linken unteren und rechten oberen Feld beschrieben sind, wie folgt:

- (i_1, j_1) und (k, ℓ), der Größe $(k - i_1 + 1) \times (\ell - j_1 + 1)$
- $(k + 1, j_1)$ und (i_2, ℓ), der Größe $(i_2 - k - 1 + 1) \times (\ell - j_1 + 1)$
- $(i_1, \ell + 1)$ und (k, j_2), der Größe $(k - i_1 + 1) \times (j_2 - \ell - 1 + 1)$
- $(k + 1, \ell + 1)$ und (i_2, j_2), der Größe $(j_2 - k - 1 + 1) \times (j_2 - \ell - 1 + 1)$.

Die Größen alle Subarrays sind kleiner als $n_0 = i_2 - i_1 + 1 = j_2 - j_1 + 1$. Deswegen findet nach der Induktionsvoraussetzung der Algorithmus den Topf im ausgewählten Untergebiet korrekt. Das wird korrekt in den Zeilen 10, 12, 14 oder 16 zurückgegeben.,

Dann gilt unsere Behauptung für alle natürliche Zahlen n, d.h. die Korrektheit des Algorithmus ist damit bewiesen.

□

Aufgabe 5.2 (5 Punkte): (Teile und Herrsche)

Bob steht im Lernraum 4.029 und blickt über die Häuser Dortmunds. Er möchte die Skyline, also den Umriss oder die Kontur der Stadt, zeichnen, doch er ist kein Künstler, sondern Informatiker.

Er abstrahiert: Die Stadt besteht aus einer Menge von n rechteckigen, sich teils überlappenden Häusern, wobei das i-te Haus durch das Tupel (x_i, b_i, h_i) beschrieben ist. Dabei ist x_i die horizontale Position des linken Rands des Hauses, $b_i > 0$ die Breite und $h_i > 0$ die Höhe. Ein Punkt (x, y) mit $y > 0$ liegt genau dann unter der Skyline, wenn (x,y) in mindestens einem der n Häuser liegt.

Bob verwendet eine Zeichen-Funktion, die zwei Punkte (x, h) und (x', h') mit $x' > x$, verbindet, indem zunächst eine Horizontale von (x, h) nach (x', h) gezeichnet wird und dann eine Vertikale von (x', h) nach (x', h'). Er braucht also einen Algorithmus, der, gegeben eine Liste von n Häusern, eine Liste mit den Eckpunkten der Skyline ermittelt. Mit der Laufzeit seines ersten Algorithmus, der ein Haus nach dem anderen in die Skyline integriert, ist er nicht zufrieden. Kann ein Teile-und-Herrsche Ansatz helfen?

Wir nehmen an, dass keine Positionen doppelt vorkommen, also $x_i \neq x_j$ und $x_i + b_i \neq x_j$ für alle $i \neq j$ und dass alle Häuser unterschiedlich hoch sind. Weiterhin gilt, dass $n = 2^k$ mit $k \in \mathbb{N}_0$.

a) Entwickeln Sie einen Teile-und-Herrsche Algorithmus `Skyline`(X, B, H, i, n), der die Skyline berechnet. Beschreiben Sie die Funktionsweise und geben Sie eine Implementierung in Pseudocode an. Die volle Punktzahl erreicht ein Algorithmus, der Worst-Case-Laufzeit in $\mathcal{O}(n \log n)$ erreicht.
Tipp: Orientieren Sie sich an Merge-Sort.

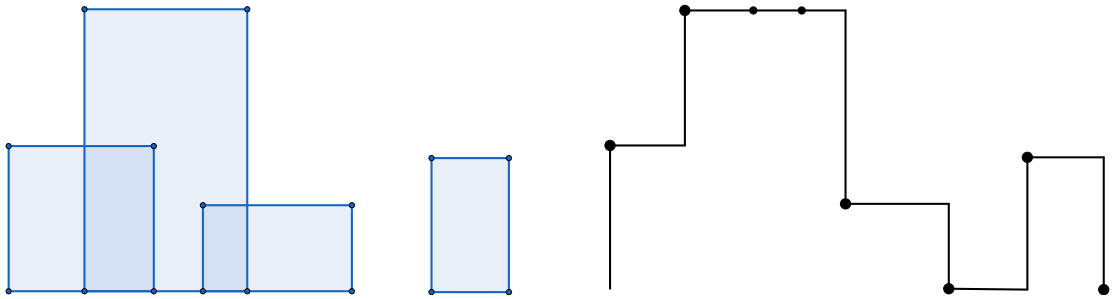


Abbildung 1: Beispiel: Links eine Menge von $n = 4$ Häusern, rechts die dazugehörige Skyline. Die dick markierten Punkte sind die ausgegebenen Ecken der Skyline (von links nach rechts). Die dünn markierten Punkte dürfen auch ausgegeben werden.

b) Beweisen Sie, dass Ihr Algorithmus die geforderte Laufzeit erfüllt.

c) Beweisen Sie induktiv die Korrektheit des Algorithmus.

Lösung:

a) Die Skyline für ein einzelnes Haus zu berechnen ist einfach, sie entspricht dem Umriss des Hauses:

$$\text{Skyline}(X, B, H, i, 1) = [(x_i, h_i), (x_i + b_i, 0)].$$

Für unseren Teile-und-Herrsche Algorithmus benötigen wir nun ein Verfahren, dass zwei Skylines zu einer kombiniert. Dabei orientieren wir uns an der bekannten Merge-Funktion aus der Vorlesung und passen sie für unsere Anwendung an. Die Merge-Funktion nimmt zwei nach x-Koordinate aufsteigend-sortierte Skylines und kombiniert diese zu einer Liste, die alle x-Koordinaten der beiden Skylines in aufsteigender Sortierung enthält. Die korrespondierenden y-Koordinaten entsprechen dabei dem Maximum der Höhen der beiden Skylines an der entsprechenden x-Koordinate.

`merge`(A, n, B, m):

1. $S \leftarrow []$
2. $i \leftarrow 1$
3. $j \leftarrow 1$
4. **while** $i \leq n$ *oder* $j \leq m$ **do**
5. **if** $i \leq n$ *und* $(j > m$ *oder* $A[i].x < B[j].x)$ **then**
6. $S \leftarrow S \cup \{(A[i].x, \max(A[i].y, B[j-1].y)\}$ // Annahme: $B[0].y = 0$
7. $i \leftarrow i + 1$
8. **else**
9. $S = S \cup \{(B[j].x, \max(B[j].y, A[i-1].y)\}$ // Annahme: $A[0].y = 0$
10. $j \leftarrow j + 1$
11. **return** S

Der vollständige Algorithmus teilt die Menge der Häuer in zwei Teile auf, berechnet rekursiv die Skyline der kleineren Probleme und verwendet die oben entwickelte Merge-Funktion, um die kombinierte Skyline zu berechnen. Optional können redundante Punkte noch entfernt werden.

`Skyline`(X, B, H, i, n):

1. **if** $n == 1$ **then**
2. **return** $[(x_i, h_i), (x_i + b_i, 0)]$
3. **else**
4. $A \leftarrow Skyline(X, B, H, i, \frac{n}{2})$
5. $B \leftarrow Skyline(X, B, H, i + \frac{n}{2}, \frac{n}{2})$
6. **return** $merge(A, n, B, n)$

b) Die Laufzeit der Merge-Funktion merge(A, n, B, m) liegt offensichtlich in $\mathcal{O}(n + m)$. Aus jedem Haus entstehen zwei Listeneinträge, somit können wir die exakte Laufzeit von merge(A, n, B, n) abschätzen für alle $n \geq 1$ mit $2cn$. Wir verwenden die folgende Rekursionsgleichung für die Worst-Case-Gesamtlaufzeit:

$$T(n) = \begin{cases} 1 & \text{für } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + 2cn & \text{für } n > 1 \end{cases}$$

Mit der Vorlesungsmethode ergibt sich eine Laufzeit von $2^{i+1} c \frac{n}{2^i} = 2cn$ auf der i-ten Rekursionsebene. Auf der untersten Ebene k ergibt sich die Laufzeit 2^k. Die Höhe des Baums beträgt $\log n$, somit ergibt sich Gesamtlaufzeit

$$T(n) = (\log n + \frac{1}{2})2n \in \mathcal{O}(n \log n)$$

c) Wir beweisen zunächst, dass die Merge-Funktion korrekt ist. Genauer beweisen wir: Wenn

- A die Skyline der Menge der Häuser $a, ..., a + n - 1$ ist,
- B die Skyline der Häuer $b, ..., b + m - 1$ ist,
- $n, m \geq 1$
- A und B nach x-Koordinate aufsteigend sortiert sind

dann berechnet merge(A, n, B, m) die aufsteigend sortierte Skyline, die an jeder x-Koordinate genau so hoch ist wie das Maximum von A und B an der Stelle.

Zunächst erkennen wir, dass sich die Skyline S =merge(A, n, B, m) nur an x-Koordinaten ändert, die in A oder B vorkommen. Analog zur merge-Funktion vom letzten Übungsblatt berechnet unsere Funktion korrekt eine Liste von Punkten, die alle x-Koordinaten aus A und B in aufsteigender Reihenfolge enthält. Einiger dieser Punkte sind möglicherweise redundant, dies ist aber laut Aufgabenstellung und zu zeigender Aussage erlaubt.

Es bleibt zu zeigen, dass die korrekten y-Koordinaten gewählt werden. Um die Notation zu vereinwachen, definieren wir $A[0] = (-\infty, 0)$ und $B[0] = (-\infty, 0)$. Betrachten wir den k-ten Punkt in S. Wir müssen zeigen, dass $S[k].y$ das Maximum der beiden Höhen ist. $S[k].x$ ist entweder $A[i].x$ für ein i oder $B[j].x$ für ein j. Wir nehmen an, der erste Fall gilt. Wir wissen also, dass A an der Stelle $S[k].x$ die Höhe $A[i].y$ hat. Sei j der Wert von j in der Iteration k. Da $A[i].x$ in S hinzugefügt wurde, muss gelten, dass $A[i].x < B[j].x$. Wir wissen also, dass die Skyline B auf dem Intervall zwischen $B[j-1].x$ und $B[j].x$ konstante Höhe hat. Die Höhe auf diesem Intervall beträgt genau die Höhe am linken Rand, also $B[j-1].y$. Der Algorithmus merge vergleicht also korrekt die Höhen von A und B an der Stelle $S[k].x$ und wählt den richtigen Wert.

Der Fall dass $S[k].x = B[j].x$ wird analog gezeigt.

Folglich gilt, dass merge(A, n, B, m) korrekt ist.

Nun beweisen wir per Induktion die Korrektheit von Skyline().
I.A.: $n = 1$. Sei i beliebig. Der Aufruf Skyline$(X, B, H, i, 1)$ berechnet korrekt die Kontur des rechteckigen Hauses i beschrieben durch die Ecke oben links (x_i, h_i) und die Ecke unten rechts $(x_i + b_i, 0)$. Jeder Punkt (x, y) mit $y > 0$ liegt genau dann unter dem Liniensegment $(x_i, h_i) \to (x_i + b_i, h_i)$, wenn (x, y) im Haus i liegt.

I.V.: Für ein beliebiges n und für alle i berechnet Skyline(X, B, H, i, n) die korrekte Skyline der Häuser $i, i + 1, ...i + n - 1$.

I.S.: Wir betrachten den Aufruf Skyline$(X, B, H, i, 2n)$. Da $2n > 1$ sind wir im Else-Fall. Laut IV enthalten A und B die korrekten Skylines der Häuser $i, ..., i + n - 1$ bzw. $i + n, ..., i + 2n - 1$. Wir haben bereits gezeigt, dass S =merge(A, n, B, n) korrekt das Maximum von A und B berechnet. Sei nun (x, y) mit $y > 0$ ein beliebiger Punkt. Falls (x, y) unter S liegt, so liegt es auch unter A oder B. Da A und B nach IV korrekte Skylines sind, liegt (x, y) also in einem Haus.

Falls (x, y) über der Skyline liegt, liegt (x, y) über A und über B, da merge() korrekt das Maximum berechnet. Da A und B nach IV korrekte Skylines sind, liegt (x, y) in keinem Haus. Somit ist S nach Definition die Skyline der Häuser $i, i + 1, ..., i + 2n - 1$.

Somit haben wir die Korrektheit von Skyline() gezeigt.