Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Michael Gatto

Datenstrukturen & Algorithmen SS 05

Aufgabe 1:

a) Um das schönste Paar zu finden, kann man wie folgt vorgehen. Man traversiert beide sortierten Arrays gleichzeitig, beginnend beim kleinsten Index. Wenn wir den i-ten Mann und die j-te Dame im Array untersuchen, so berechnen wir die Differenz ihrer Grössen. Ist diese kleiner als das aktuelle Minimum, so merken wir uns das Paar und setzen das Minimum auf deren Grössenunterschied. Dann inkrementiert man der Index der Liste, wo die betrachtete Person kleiner ist. Dies darf gemacht werden, da man durch das Inkrementieren des Index im Array mit der grösseren Person den Grössenunterschied nur vergrössert. Deshalb muss man dieses Paar nicht anschauen. Intuitiv sieht man das auch, indem man sich überlegt, wie das schönste Paar mit einer fixen Dame aussieht. Es gibt nur eine kleine Auswahl von Männern, die mit der gegebenen Dame ein schönes Paar bilden können. Diese sind derjenige Mann, der unmittelbar kleiner oder gleich gross ist wie Dame, oder derjenige Mann, der unmittelbar grösser ist als die Dame. Das oben beschriebene Verfahren testet diese zwei Möglichkeiten auf eine effiziente Art.

b)
```
paar(dam: ARRAY [PERSON]; her: ARRAY [PERSON]) is
    require
             damen_not_void: dam /= void
             herren_not_void: her /= void
             damen_not_empty: dam.count /= 0
             herren_not_empty: her.count /= 0
    local
            d: INTEGER -- Damen Index
            h: INTEGER -- Herren Index
            g_min: INTEGER -- Minimalen Groessenunterschied
            h_min: INTEGER -- Index des Herren, der das schoene Paar bildet.
            d_min: INTEGER -- Index der Dame, die das schoene Paar bildet.
            h_temp: INTEGER -- Zwischenspeicher fuer den Groessenunterschied
    do
            from
                    d := dam.lower
                    h := her.lower
                    g_min := (dam.item(d).height - her.item(h).height).abs()
                    --Die maximale Groessendifferenz ist sicher kleiner gleich
                    --wie die des ersten Paars
                    h_min := h
                    d_min := d
            variant
                    dam.upper + her.upper - d - h + 1
            until
                    d > dam.upper or h > her.upper
            loop
```

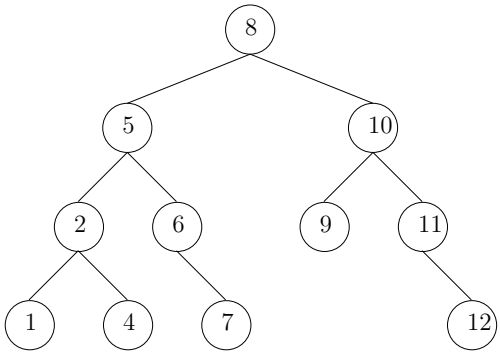
```
                    h_temp := (dam.item(d).height - her.item(h).height).abs()
                    if (h_temp < g_min) then
                            -- neues Minimum
                            g_min := h_temp
                            h_min := h
                            d_min := d
                    end
                    if (dam.item(d).height < her.item(h).height) then
                            d:= d + 1
                    else
                            h:= h + 1
                    end
            end
            -- Paar gefunden, ausgeben
            io.put_string("Das schoenste Paar bilden ")
            io.put_string(dam.item(d_min).name)
            io.put_string(" und ")
            io.put_string(her.item(h_min).name)
            io.put_new_line
    end
```

c) Die Komplexität des Verfahren ist $O(N + M)$, da man beide Arrays genau einmal traversiert, und der Aufwand pro betrachtetem Paar konstant ist.

Aufgabe 2:

a) Die Tabelle sieht wie folgt aus: [5, 10, 7, 15, 12, 8, 20, 16, 20, 20, 13].

b) Je nachdem, ob man den symmetrischen Vorgänger benutzt oder der symmetrischen Nachfolger, gibts zwei oder eine Doppelrotation. Der Baum am Ende sieht wie folgt aus:



c) Nach der Zugriffsfolge ist die selbstanordnende Liste:

$$\mathrm{O} \longrightarrow \mathrm{L} \longrightarrow \mathrm{A} \longrightarrow \mathrm{H} \longrightarrow \mathrm{G} \longrightarrow \mathrm{R} \longrightarrow \mathrm{I} \longrightarrow \mathrm{T} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U} \longrightarrow \mathrm{S}$$

d) Die gefüllte Hashtabelle ist:

```
H = __ 26 16 03 __ 12 22
```

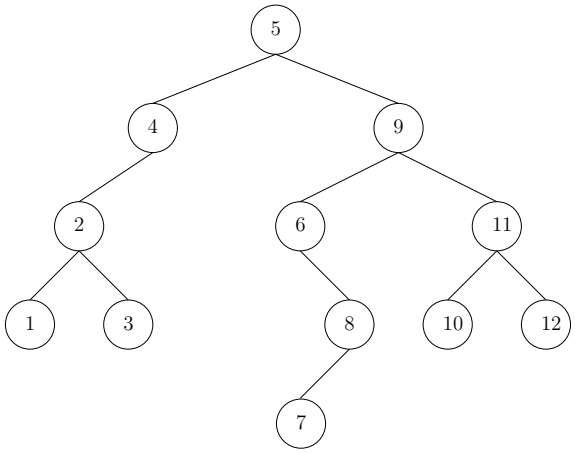
e) Auf Niveau 0: 1. Auf Nievau 1 : 3. Die Anzahl Knoten verdreifacht sich jedes mal, wo man im Baum von Niveau i zu Niveau $i + 1$ geht. Auf Niveau h sind deshalb 3^h Knoten vorhanden.

f)

	Vergleiche	Schreib-Operationen
sortierter Array	$\Theta(\log(n))$	$\Theta(n)$
sortierte lineare Liste	$\Theta(n)$	$\Theta(1)$
AVL-Baum	$\Theta(\log(n))$	$\Theta(\log(n))$
Heap	$\Theta(\log(n))$	$\Theta(\log(n))$

Beim AVL-Baum kann es nötig sein, die Balance bis ganz oben auf die Wurzel anpassen zu müssen. Deshalb entstehen $\Theta(\log(n))$ Schreibeoperationen.

g)



Aufgabe 3:

a) Mit Teleskopieren erkennt man auf die Form $T(n) = \frac{4}{3}n^2 + \frac{5}{3}$.
Die Induktionsverankerung ist: $T(1) = \frac{4}{3} + \frac{5}{3} = 3$.
Induktionsschritt: $T(n) = 4T(\frac{n}{4}) - 5 = 4(\frac{4}{3}\left(\frac{n}{2}\right)^2 + \frac{5}{3}) - 5 = \frac{4}{3}n^2 + \frac{20}{3} - 5 = \frac{4}{3}n^2 + \frac{5}{3}$.

b) Die Reihenfolge ist:
$$7n^2 + 3n, n^2\log(n), \frac{n^3}{\log(n)}, \frac{n^3}{7} - 7n^2, 2^{\sqrt{n}}, 2^n, n!$$

c)
- ☐ $n^5 - n \in O(n^4)$: Falsch, da $n^5 - n = n \cdot (n^4 - 1)$.
- ☐ $\log_3 n + \log_2 n \in O(\log_2 n)$: Wahr, da $\log_3 n = \frac{\log_2 n}{\log_2 3}$.
- ☐ $\sum_{i=1}^{n}(ni) \in \Omega(n^3)$: Wahr, da $\sum_{i=1}^{n}(ni) = n\sum_{i=1}^{n} i = n \cdot \frac{n^2-n}{2} = n^3 \in \Theta(n^3)$.
- ☐ $(n+1)! \in \Theta(n!)$: Falsch, da $(n+1)! = (n+1) \cdot n!$.

d) Die externe Schlaufe wird $\log n$ mal durchgelaufen. Die interne Schalaufe läuft jeweils von i-mal durch. Am Anfangs also n mal, bei der zweiten Iteration mit $i = \frac{n}{2}$ nur noch $\frac{n}{2}$ mal, dann $\frac{n}{4}$, $\frac{n}{8}$, $\frac{n}{16}$, und so weiter. Daraus folgt:
$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} = n \sum_{i=0}^{\log(n)} \frac{1}{2^i} = n \cdot \frac{1 - \frac{1}{2^{\log(n)+1}}}{1 - \frac{1}{2}} < 2n$$

e) Für i zwischen 1 und $\frac{n}{3}$, j zwischen $\frac{2}{3}n$ und n, erfolgen mindestens $\frac{n}{3}$ Iterationen im Loop, das mit k indexiert ist. Somit ist die Laufzeit $\Omega(n^3)$. Es wird auch nicht mehr als $O(n^3)$ Aufwand getrieben, denn das Loop mit k macht für jedes fixe (i, j)-Paar auch höchstens n Iterationen, und es gibt n^2 solche Paare. Deshalb ist der Aufwand $\Theta(n^3)$.

Aufgabe 4:

a) Für dieses Problem verwendet man einen Scan-Line Ansatz. Die vertikale Scanline geht von links nach rechts durch das Feld und stoppt bei jedem Anfangs- und Endpunkt eines Segments. An diesen Punkten kann sich etwas über die Sichtbarkeit ändern. Beginnt ein Segment, so bildet es mit den zwei Nachbarsegmente auf der Scanline jeweils ein sichtbares Paar. Endet ein Segment, so sind seine zwei Nachbarn jetzt ein sichtbares Paar. Damit das effizient läuft, speichert man die Segmente, die die Scanline kreuzen, in die Blätter eines AVL Baum mit Blatt-Verkettung. Als Schlüssel verwendet man die y-Koordinaten der Segmente. Die Blatt-Verkettung erlaubt somit den effizienten Zugriff auf das unmittelbar kleinere und grössere Element, was hier erlaubt, die Nachbarschaft des betrachteten Segment aufzufinden. Um duplikate zu verhindern, fügt man jedes gefundene sichtbare Paar in einen AVL-Baum ein, wo man das Einfügen so abändert, das vorhandene Elemente nicht nochmals eingefügt werden.

b) *Zusätzliche Erklärungen zum Vorgehen* Um den Scanline-Algorithmus auszuführen, muss man die $2n$ Endpunkte der Segmente in aufsteigender x-Koordinate durchgehen. Dafür muss man die Endpunkte sortieren, oder sie aus einer Priority-Queue für Minimumsuche der Reihe nach entfernen.

Die Scanline geht dann die Endpunkte der Reihe nach durch. Findet sie einen neuen Anfangspunkt, so wird diese in den AVL-Baum eingefügt. Dies braucht $O(log(n))$ Zeit. Danach wird die Nachbarschaft angeschaut. Jedes dieser Segmente bildet mit dem gerade eingefügte Segment ein sichtbares Paar.

Um sichtbare Paare nicht doppelt zu berichten, fügt man jedes sichtbare Paar in einen zweiten AVL-Baum ein. Zur Erinnerung kann man den AVL-Baum so implementieren, dass falls ein Element schon vorhanden ist, man es nicht ein zweites mal einfügt. Dies erfolgt zum Beispiel indem man zuerst nach dem Element sucht, und es nur dann einfügt, wenn man es nicht gefunden hat. Da sowohl das Einfügen als auch das Suchen in einem AVL-Baum in $O(\log n)$ Zeit erfolgt, hat dies keinen negativen Einfluss zuf die Asymptotische Laufzeit des Algorithmus.

Als Schlüssel für den AVL-Baum der die Resultate speichert benutzt man das Paar von x-Koordinaten der zwei sichtbaren Segmente: Man sortiert die zwei x Koordinaten so, dass $x_1 < x_2$, und fügt dann das Paar (x_1, x_2) in den AVL Baum ein. Somit erreicht man, das jedes Segmentpaar ein eindeutiger (x_1, x_2)-Schlüssel hat. Im AVL-Baum sind diese Schlüssel lexikographisch geordnet.

Findet die Scanline hingegen ein Endpunkt eines Segments, so wird dieses von AVL-Baum entfernt. Auch dies passiert in $O(\log n)$ Zeit. Nach dem Entfernen entsteht ein neues sichtbares Paar, und zwar die zwei Segmente, die vor dem Entfernen Nachbarn vom gelöschten Segment waren. Diese fügt man wieder in den zweiten AVL-Baum ein, um Duplikate zu verhindern.

Der Pseudocode sieht z.B. wie folgt aus.

Procedure `get_smaller(`y`)`
Input: y: Blatt im AVL-Baum mit Blattverkettung
Output: y_s: Das unmittelbar kleinere Blatt im AVL-Baum mit Blattverkettung. Void, falls keins vorhanden.
Procedure `get_bigger(`y`)`
Input: y: Blatt im AVL-Baum mit Blattverkettung
Output: y_b: Das unmittelbar grössere Blatt im AVL-Baum mit Blattverkettung. Void, falls keins vorhanden.

```
Procedure sichtbarkeit
    Output: res: Der AVL-Baum mit den sichtbaren Paaren
    scan: AVL-Baum mit Blattverkettung, für die Scanline
    queue: Priority Queue für (x-Koordinate, Segment)
    forall s in segments do
        //Anfangs und Enpunkte der Segmente in einer Priority Queue
        queue.add(s.x,s)
        queue.add(s.x+s.l,s)
    while not queue.empty do
        el := queue.get_min
        key := queue.minimum_key
        queue.remove_minimum
        if key = el.x then
            //Startpunkt
            el mit Schlüssel el.y in scan einfügen.
            smaller := get_smaller(el)
            bigger := get_bigger(el)
            if smaller exists then
                (smaller,el) ohne Duplikate in res Einfügen, als Schlüssel (smaller.x, el,x) verwenden.
            if bigger exists then
                (bigger,el) ohne Duplikate in res Einfügen, als Schlüssel (bigger.x, el,x) verwenden.
        else
            //Endpunkt
            smaller := get_smaller(el)
            bigger := get_bigger(el)
            if smaller exists and bigger exists then
                (smaller, bigger) ohne Duplikate in res Einfügen, als Schlüssel das Paar von aufsteigend
                sortierten x-Koordinaten verwenden
            scan.remove(el)
```

c) Im Ganzen gibt es $2n$ Endpunkte. Die Endpunkte müssen zuerst der Grösse nach sortiert werden, was $O(n \log n)$ Zeit braucht. Pro Endpunkt muss man das Element in den AVL-Baum mit Blattverkettung einfügen oder entfernen. Dies braucht $O(\log n)$ Zeit. Das Auffinden der Nachbarn zu einem gegebenen Segment erfolgt dank der Blattverkettung in $O(1)$ Zeit.

Danach muss man das Resultat in den AVL-Baum einfügen. Da es höchstens $3n$ sichtbare Paare geben kann (höchstens zwei für jeder Startpunkt eines Segments, und höchstens ein Paar für jedes Segmentende), hat auch der AVL-Baum zum speichern des Resultats $O(n)$ Knoten, und somit lauft die Einfügeoperation in $O(\log n)$ Zeit. Dieser Algorithmus hat also eine Laufzeit von $O(n \log n)$.

Der benötigte Speicherplatz ist $O(n)$: der AVL-Baum mit Blattverkettung hat $O(n)$ Blätter maximal, und pro Blatt einen konstanten Speicherbedarf. Der Baum zum speichern der Resultate hat auch $O(n)$ Blätter mit konstanten Speicherbedarf pro Blatt.

d) Es genügt, die Scanline einmal horizontal für die horizontalen Linien durchlaufen zu lassen, und einmal vertikal für die vertikalen Linien. Beim horizontalen Scan benutzt man nur die horizontalen Linien, beim vertikalen Scan nur die vertikalen. Der Grund dafuer ist, dass sich die gegebenen horizontalen und vertikalen Linien gemäss der gegeben Definition von Sichtbarkeit nicht beeinflussen. Sogar der pathologische Fall, dass man ein Schar von horizontalen Linien zwischen zwei vertikalen Linien hat, die

die Sichtbarkeit verhindern könnten, ist nicht moeglich, da alle Segmente unterschiedliche ganzzahlige (x, y)-Koordinaten haben. Um das Verfahren nicht nochmals für vertikale Segmente implementieren zu müssen, kann man die vertikalen Linien um 90 Grad drehen, und das Verfahren (beschränkt auf die gedrehten Segmente) wie gewohnt ausführen.