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
Thomas Tschager
Antonis Thomas

16. März 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 3 FS 16

Lösung 3.1 *Vergleich von Sortieralgorithmen.*

	`bubbleSort`		`insertionSort`	
	min	max	min	max
Vergleiche	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Eingabefolge	jede	jede	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Vertauschungen	0	$\Theta(n^2)$	0	$\Theta(n^2)$
Eingabefolge	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

	`selectionSort`		`quicksort`	
	min	max	min	max
Vergleiche	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$
Eingabefolge	jede	jede	$(\star)$	$1, 2, \ldots, n$
Vertauschungen	0	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
Eingabefolge	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$ $(\star\star)$	$1, 2, \ldots, n$	$(\star)$

$(\star)$: Eine entsprechende Folge ist nicht leicht hinzuschreiben. Die Folge muss so konstruiert sein, dass jedes Pivotelement den zu sortierenden Bereich halbiert. Für $n = 7$ etwa ist eine solche Folge durch $4, 5, 7, 6, 2, 1, 3$ gegeben.
$(\star\star)$: Noch mehr Vertauschungen – genauer $n - 1$ und damit die höchstmögliche Anzahl an Vertauschungen – benötigt `selectionSort` für die Eingabefolge $n, 1, 2, 3, ..., n - 1$.

Lösung 3.2 *Algorithmenentwurf: Zahlensummen.*

a) Zuerst sortieren wir A in aufsteigender Reihenfolge. Dies geht in $\mathcal{O}(n \log n)$ Schritten (z.B. mit Mergesort/Heapsort). Danach prüfen wir für jede mögliche Wahl von a (es gibt n Kandidaten), ob die Zahl $z - a$ im Array vorkommt (in diesem Fall hätten wir zwei Zahlen mit $a + b = z$ gefunden). Da A sortiert ist, können wir mittels binärer Suche in $\mathcal{O}(\log n)$ Schritten feststellen, ob $z - a$ in A vorkommt. Falls die Zahl vorkommt, haben wir eine Lösung gefunden, ansonsten probieren wir die nächste Wahl von a aus, usw. Insgesamt ergibt sich eine Laufzeit von $\mathcal{O}(n \log n)$.

Bemerkung: Das obige Verfahren ist asymptotisch optimal. Man kann zeigen, dass *jeder* Algorithmus für dieses Problem $\Omega(n \log n)$ Schritte benötigt (Details finden sich in "Refined Upper and Lower Bounds for 2-SUM", A.C. Chan, W.I. Gasarch, C.P. Kruskal, 1997).

b) Natürlich könnten wir hier genau wie in a) vorgehen, ausser dass wir nicht mehr sortieren müssen. Allerdings wäre so die Laufzeit immer noch $\mathcal{O}(n \log n)$, da n Mal nach der Zahl $z - a$ binär gesucht werden muss.

Eine Laufzeit von $\mathcal{O}(n)$ erreichen wir mit folgender Überlegung: Seien l, r die Indizes des

linken bzw. rechten Endes des Arrays. Falls $A[l] + A[r] = z$, geben wir $A[l]$ und $A[r]$ aus und beenden das Verfahren. Wenn nun $A[l] + A[r] > z$, dann gilt sicher $A[k] + A[r] > z$ für jedes k mit $l \leq k \leq r$ (es gilt $A[k] \geq A[l]$, da A sortiert ist). Folglich gibt es im Array von Position l bis Position r keine Zahl, die zu $A[r]$ zusammengezählt genau z ergibt. Es genügt also, im Array nur Elemente mit Index $\leq r - 1$ zu berücksichtigen. Daher setzen wir in diesem Fall $r := r - 1$ und wiederholen das Verfahren.

Ist dagegen $A[l]+A[r] < z$, dann gilt sicher $A[l]+A[k] < z$ für jedes k mit $l \leq k \leq r$. Folglich gibt es im Array von Position l bis Position r keine Zahl, die zu $A[l]$ zusammengezählt genau z ergibt (es gilt $A[k] \leq A[r]$, da A sortiert ist). Also genügt es, im Array nur Elemente mit Index $\geq l+1$ zu berücksichtigen. Daher setzen wir in diesem Fall $l := l+1$ und wiederholen das Verfahren.

Wir brechen ab, falls entweder ein Paar gefunden wurde, oder falls $l = r$. Wir finden so immer ein Paar a, b mit $a + b = z$, falls ein solches existiert. Die Laufzeit von $\mathcal{O}(n)$ können wir beweisen, indem wir den Verlauf von $r - l$ betrachten: In jedem Schritt, in dem das Verfahren nicht abbricht, wird entweder r um eins erniedrigt oder l um eins erhöht (aber niemals beides gleichzeitig). Das heisst, $r - l$ wird in jedem Schritt eins kleiner. Ursprünglich ist $r - l = n - 1$, und somit terminiert das Verfahren nach höchstens $n - 1$ Schritten.

Bemerkung: Das gerade beschriebene Vorgehen wäre natürlich auch in a) anwendbar und auch effizienter als die dort verwendete wiederholte binäre Suche. Weil dort aber allein für das Sortieren bereits $\Omega(n \log n)$ Schritte benötigt werden, wäre die asymptotische Gesamtlaufzeit nicht besser.

c) Eine Laufzeit von $\mathcal{O}(n^2)$ ergibt sich, indem das Array zuerst sortiert und danach die Idee aus b) verwendet wird: Wir probieren jeden der n möglichen Werte in A als Kandidaten für c aus, und wenden nacheinander für jeden davon den Algorithmus aus b) an. Zusätzlich müssen wir noch sicherstellen, dass a und b verschieden sind.

Bemerkung: Es ist unbekannt, ob mit einem anderen Algorithmus eine bessere Laufzeit erreicht werden kann.

Lösung 3.3 *Median nach Blum (**Programmieraufgabe**).*

In dieser Aufgabe sollte der *Algorithmus von Blum* zur Medianberechnung implementiert werden. Wir beschreiben die Implementierung schrittweise. Zwei Variablen werden innerhalb der Klasse global benutzt: Die Variable n speichert die Anzahl der Elemente, während die Elemente selbst im Array v gespeichert werden. Die konkrete Aufgabe besteht in der Berechnung des Medians des Arrays v, also des Elements auf der Position $\lceil n/2 \rceil$, wenn v sortiert wäre.

```
static int n;
static int v[];
```

Wir definieren eine Hilfsfunktion `swap`, die zwei Elemente des Arrays v vertauscht.

```
static void swap(int i, int j) {
    int aux = v[i];
    v[i] = v[j];
    v[j] = aux;
}
```

Berechnung des Medians in jeder Gruppe

Zunächst teilen wir die Elemente in $\lfloor n/5 \rfloor$ Gruppen mit je fünf Elementen und höchstens eine Gruppe mit n mod 5 Elementen auf. Danach muss der Median jeder Gruppe gefunden werden. Dies geschieht durch die Funktion `posMedian5`, die zwei Parameter besitzt: Den linken Index l (`indexFirst`) und den rechten Index r (`indexLast`) der Gruppe. Üblicherweise setzen wir $r = l + 4$, ausser für die letzte Gruppe. Die Implementierung benutzt einen elementaren Sortieralgorithmus (hier: Bubblesort) zur Sortierung von $v[l..r]$, und nach der Sortierung wird die Position $\lfloor (l + r)/2 \rfloor$ des Medians zurückgeliefert.

```
static int posMedian5(int indexFirst, int indexLast) {
    int i, rep;
    /* Use bubble sort */
    for(rep = 0; rep < 4; rep++)
        for(i = indexFirst; i < indexLast-rep; i++)
            if(v[i] > v[i+1]) swap(i, i+1);
    /* Median is located at the middle position */
    return (indexFirst+indexLast)/2;
}
```

Aufteilungsschritt von Quickselect

Wir implementieren den Aufteilungsschritt von Quickselect auf die folgende Art. Nachdem ein Pivotelement p gewählt wurde, wird es mit dem letzten Element der aufzuteilenden Sequenz vertauscht. Danach durchläuft die Variable i die Werte `left` bis `right` -1, und in jeder Iteration wird $v[i]$ betrachtet. Weiterhin benutzen wir eine Variable `storePos` mit der Eigenschaft, dass $v[\texttt{left}..(\texttt{storePos}) - 1]$ alle Elemente kleiner als p enthält, während $v[\texttt{storePos}..(i - 1)]$ nur Elemente enthält, die mindestens den Wert p besitzen. Ist nun $v[i]$ kleiner als p, dann wird es es mit dem Element $v[\texttt{storePos}]$ vertauscht und `storePos` inkrementiert. Am Ende wird das Pivotelement p rechts mit $v[\texttt{storePos}]$ vertauscht (da dieses Element einen Wert von mindestens p hat). Dann befindet sich p auf der korrekten Position: Alle Elemente links von p haben einen kleineren Wert und alle Elemente rechts von p haben einen Wert von mindestens p.

Betrachten wir zum Beispiel das Array $[1, 9, 6, 5, 3, 7, 4]$ und benutzen 5 als Pivotelement. Zunächst wird das Pivotelement mit dem Element ganz nach rechts im Array vertauscht, was zum Array $[1, 9, 6, 4, 3, 7, 5]$ führt. Danach iterieren wir von links nach rechts und prüfen, ob ein Element kleiner ist als das Pivotelement. In diesem Fall wird es auf die linke Seite verschoben. Nachdem wir alle Elemente, die kleiner als das Pivotelement sind, nach links verschoben haben, erhalten wir das Array $[1, 4, 3, 9, 6, 7, 5]$. Schliesslich wird das Pivotelement hinter die letzte Position verschoben, die ein Element enthält, welches kleiner als das Pivotelement ist. Damit ergibt sich das Array $[1, 4, 3, 5, 6, 7, 9]$.

```
static int partition(int left, int right, int pivotPos) {
    int storeIndex = left, i;
    int pivotVal = v[pivotPos];
    swap(pivotPos, right); // Move pivot to the end
    // Move element less than pivot at the beginning
    for(i = left; i < right; i++)
      if(v[i] < pivotVal)
          // Swap v[i] with v[storeIndex]
          swap(i, storeIndex++);
    // Move pivot on it's final position
    swap(storeIndex, right);
    return storeIndex; // Return new pivot position
}
```

Der Algorithmus von Blum

Wir implementieren nun den Algorithmus von Blum zur Bestimmung des Elements, das sich in der sortierten Sequenz auf Position k befindet. Man beachte, dass das erste Element den Index 0 und das letzte den Index $n-1$ besitzt. Zur Berechnung des Medians von v muss daher $k = \lfloor (n-1)/2 \rfloor$ benutzt werden. Der Algorithmus ist in der Funktion `medianPos` implementiert und besitzt neben k drei weitere Parameter: `left` und `right` geben die linke bzw. die rechte Grenze der Sequenz an, während `display` eine boolesche Variable ist, die angibt, ob der Algorithmus eine Ausgabe erzeugt oder nicht. Da eine Ausgabe nur im ersten Schritt des Algorithmus erfordert wird, ist der Wert von `display` initial `true`, wird aber für jeden weiteren rekursiven Aufruf des Algorithmus auf `false` geändert.

Wie bereits erklärt wird das Array in $\lceil n/5 \rceil$ Gruppen aufgeteilt. Wir sortieren die Elemente innerhalb jeder dieser Gruppen und berechnen so die Mediane $M_1, ..., M_{\lceil n/5 \rceil}$ der Gruppen. Danach verschieben wir diese Mediane an den Anfang von v[`left..right`]. Konkret vertauschen wir die Inhalte von v[`left`$+i-1$] mit dem Median M_i der i-ten Gruppe für $1 \leq i \leq \lceil n/5 \rceil$. Da die Mediane nun in einem zusammenhängenden Teilarray gespeichert sind, kann der *Median-der-Mediane* rekursiv berechnet werden, denn dieser ist genau der Median von v[`left`$..($`left`$+\lceil n/5 \rceil - 1)$]. Beträgt die Anzahl der Mediane höchstens 5, dann wird die vorher definierte Funktion `posMedian5` anstelle des Algorithmus von Blum benutzt, um den Median zu berechnen.

Nach dem rekursiven Aufruf haben wir die Position des Medians-der-Mediane berechnet. Dieses Element wird nun als Pivotelement für den Aufteilungsschritt benutzt. Wenn k genau der Position des Pivotelements *nach* dem Aufteilungsschritt entspricht, dann sind wir fertig (denn jedes Element links vom Pivotelement ist kleiner, während der Wert der Elemente rechts vom Pivotelement mindestens so gross wie das Pivotelement selbst ist). Ist dagegen die Position des Pivotelements grösser als k, dann führen wir den rekursiven Aufruf auf dem linken Teilbereich durch, ansonsten auf dem rechten.

```
static int medianPos(int first, int last, int k, boolean display) {
    int i, j, posMedian, pivotPos, nMedians = 1 + (last - first)/5;
    // Find medians of groups of 5
    for(i = first, j = 0; i <= last; i += 5, j++) {
        // Last group may have less than 5 elements
        posMedian = posMedian5(i, Math.min(i+4, last));

        // Display median of each group of 5 elements
        if(display) System.out.print(v[posMedian] + " ");

        // Store medians in the beginning of the array
        swap(first + j, posMedian);
    }
    // Find median of medians recursively
    // If we have less than 5 medians we can use posMedian5 to find the median
    if(nMedians > 5)
        pivotPos = medianPos(first, first+nMedians-1, first+(nMedians-1)/2, false);
    else
        pivotPos = posMedian5(first, first+nMedians-1);

    // Display median of medians
    if(display) System.out.print(v[pivotPos] + " ");

    // Partition with the median of medians as pivot
    pivotPos = partition(first, last, pivotPos);

    // Recurse on the left or right side of the pivot if necessary
    if(pivotPos == k) return pivotPos;
    else if(pivotPos > k) return medianPos(first, pivotPos-1, k, false);
    else return medianPos(pivotPos+1, last, k, false);
}
```

Hauptprogramm

```
import java.util.Scanner;

public class Main2 {

    static int n;
    static int v[];

    static void swap(int i, int j){...}
    static int posMedian5(int indexFirst, int indexLast) {...}
    static int partition(int left, int right, int pivotPos) {...}
    static int medianPos(int first, int last, int k, boolean display) {...}

    public static void main(String[] args) {
        int test, ntest, i;
        Scanner sc = new Scanner(System.in);
        ntest = sc.nextInt();
        for(test = 1; test <= ntest; test++) {
            n = sc.nextInt();
            v = new int[n];
            for(i = 0; i < n; ++i)
                v[i] = sc.nextInt();

            // Compute the position of the median
            int posMedian = medianPos(0, n-1, (n-1)/2, true);
            System.out.println(v[posMedian]);
        }
    }

}
```