## Aufgabe 1

Wir betrachten den folgenden Algorithmus, dem als Eingabe ein Array a aus n natürlichen Zahlen übergeben wird und der als Ausgabe zwei Zahlen x und y liefert.

```
var x,y: int; a : array 1..n of int;
input a;

x := a[0];
y := a[0];
for i=1 to n-1 do
        if x > a[i] then
                x := a[i];
        fi
        if y < a[i] then
                y := a[i];
        fi
od
output: x, y
```

**a)** Was berechnet der obige Algorithmus? (2P)

L: Minimum (x) und Maximum (y) der Folge a.

**b)** Implementieren Sie den Algorithmus in Java und geben Sie bei jedem Schritt der For-Schleife den aktuellen Wert von x und y an. (2P)

**c)** Wie viele Vergleiche, abhängig von n, führt der obige Algorithmus durch? Hierbei sollen nur Vergleiche mit Elementen aus a berücksichtigt werden. (1P)

L: $2(n-1)$

**d)** Geben Sie einen Algorithmus in Pseudocode an, der dasselbe Problem wie der obige Algorithmus löst und dafür höchstens $\frac{3}{2}n$ Vergleiche von Elementen aus a benötigt. Beschreiben Sie zusätzlich die Grundidee Ihres Algorithmus in wenigen Sätzen. (4P)

L: Idee: Vergleiche zwei nebeneinanderliegende (adjazente) Elemente der Folge. Sind diese verschieden und das erste Element kleiner als das zweite, so kann das erste nur Minimum und das zweite Element nur Maximum sein. So benötigt man für die Bearbeitung von zwei Elementen nur drei statt vier Vergleiche im ursprünglichen Algorithmus.

Eine einfachere Optimierung besteht darin, in obigem Algorithmus den zweiten Vergleich nur dann auszuführen, wenn der erste kein neues Minimum ergeben hat. Allerdings bleibt diese Variante im Worstcase bei $2(n-1)$ Vergleichen.

**e)** Kann man das Problem, das der obige Algorithmus löst, auch mit weniger als $n-1$ Vergleichen lösen? Begründen Sie Ihre Antwort. (2P)

L: Nein, da sonst Elemente unberücksichtigt.

### Aufgabe 2

Ein Einbrecher hat ein Problem: Seine Tasche ist zu klein für alle Wertsachen im Tresor. Im Tresor befinden sich nämlich $n$ Gegenstände, die alle jeweils ein Gewicht $w_i$ und einen Wert $v_i$ ($i \in R = [1, n]$) haben. Praktischerweise hat der Einbrecher aber seinen Computer dabei, so dass er schnell ein Programm schreibt, das ihm die optimale Auswahl an Diebesgut berechnet, so dass

- das Gesamtgewicht der Gegenstände in der Tasche deren maximale Traglast $W_{max}$ nicht übersteigt ($\sum_{i \in R} w_i \leq W_{max}$), und
- der Wert der Gegenstände in der Tasche ( $\sum_{i \in R} v_i$ ) maximal ist, d. h. es gibt keine andere Bepackung der Tasche, so dass der Gesamtwert der Gegenstände größer ist.

Ihre Aufgabe ist es, das Programm des Einbrechers selbst zu schreiben. Benutzen Sie dabei ein Verfahren, das alle Möglichkeiten berechnet und testet. Das beste Ergebnis soll dann ausgegeben werden (Brute-Force). Welche durchschnittliche Laufzeit hat Ihr Algorithmus? Begründen Sie ihre Antwort kurz. (8P)

Hinweis: Das Programm erhält eine Folge von Argumenten, zuerst die maximale Traglast der Tasche, danach jeweils das Gewicht und den Wert eines Gegenstands. Ein Aufruf des Programms soll folgende Ausgabe liefern:

```
java Einbrecher 15 2 6 2 3 6 5 5 4 3 5 7 8 1 3 6 3 4 2
110011100
Value 25
Time needed: 2 ms
```

L: Im Brute-Force-Fall müssen einfach alle möglichen ($2^n$) Kombinationen ermittelt und berechnet werden. Implementierung mittels vollständigem Berechnen zB über ein Bitfeld oder ein Backtracking.

### Aufgabe 3

Gegeben sei das folgende Programm.

```
Eingabe: reelle Zahl x und natürliche Zahl n
Ausgabe: ???

Algorithmus P(x, n):
var n,x,temp: int
if n = 1 then
        output x
fi

if n gerade then
        temp = P(x, n/2);
```

```
        output temp * temp
else
        temp = P(x, (n-1)/2 );
        output temp * temp * x
fi
```

**a)** Was berechnet der Algorithmus? Geben Sie hierzu eine Formel an. (4P)

L: $x^n$

**b)** Geben Sie eine möglichst gute obere Schranke für die Anzahl der Multiplikationen des Algorithmus an und begründen Sie diese. (2P)

L: $\frac{3}{2}\log_2 n$. Begründung: Rekursion halbiert n fortlaufend und wird beendet bei n=1, das heisst wir brauchen so viele Rekursive Aufrufe r, so dass $2^r = n$.

### Aufgabe 4

Implementieren Sie eine rekursive Variante der Binären Suche. (8P)

### Aufgabe 5

Welche der im Buch, Kapitel 5, genannten Sortieralgorithmen läuft für eine bereits sortierte Menge am schnellsten ab? Begründen Sie Ihre Antwort. (2P)

L: Insertion Sort und Bubble Sort laufen nur einmal durch. (n Vergleiche). Siehe auch Saake/Sattler, S. 139 ff.

### Aufgabe 6

Welche der im Buch, Kapitel 5, genannten Sortieralgorithmen läuft für eine in umgekehrter Reihenfolge sortierte Menge am schnellsten ab? Begründen Sie Ihre Antwort. (2P)

L: Merge Sort. Der Algorithmus läuft in allen Fällen gleich ab, nämlich mit $n \log_n$ Vergleichen.

Siehe auch Saake/Sattler, S. 139 ff. und http://www.ddeville.me/2010/10/sorting-algorithms-comparison/

### Aufgabe 7

Implementieren Sie eine rekursive Variante von Quicksort, die für Teildateien von weniger als M Elementen zu Insertion Sort übergeht, und bestimmen Sie empirisch den Wert von M, für den er bei einer zufälligen Datei mit 1000 Elementen am schnellsten abgearbeitet wird. (8P)

L: Die beiden Algorithmen stehen fertig im Buch. Es ist nur beim rekursiven Aufruf zu testen, ob die Grenze M bereits erreicht ist. In diesem Fall muss statt qSort(Teilmenge) insertionSort(Teilmenge) aufgerufen werden. Ermittlung des Optimums durch Iteration über alle Werte von 2 bis 1000. Offenbar liegt dieses Optimum im Bereich von etwa 50-100.

## Aufgabe 8

Implementieren Sie eine rekursive Variante von Mergesort für ein Feld aus N ganzen Zahlen unter Verwendung eines Hilfsfeldes, dessen Grösse weniger als $N/2$ beträgt. (8P)

L: Nach dem Sortieren der beiden Hälften genügt es im Merge-Schritt, nur die erste Hälfte ins Hilfsarray auszulagern. Der Merge erfolgt dann zur Hälfte aus dem Hilfsarray, zur anderen Hälfte aus der zweiten Hälfte des Original-Arrays. Es besteht keine Gefahr, dass die zweite Hälfte beim Merge überschrieben wird mit Daten aus dem Hilfsarray, da dieses ja gleich gross ist.

Siehe auch http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm

## Aufgabe 9

Welches ist die kleinste Anzahl von Schritten, mit denen Mergesort im optimalen Fall auskommen könnte? (2P)

L: Merge Sort benötigt immer $n \log n$ Schritte.