1. Übung

1. (a) Selection Sort ist ein einfacher Sortieralgorithmus: Sei A eine Liste mit n Elementen, so sucht man zunächst das kleinste Element und bringt es an die erste Position. Anschließend sucht man das zweitkleinste Element und bringt es an die zweite Position etc.
   Überlegen Sie sich einen Pseudocode (siehe z.B. http://de.wikipedia.org/wiki/Pseudocode für eine Definition/Erklärung von Pseudocode) für diesen Algorithmus und bestimmen Sie die Anzahl der notwendigen Schritte, die (in Ihrem Pseudocode) nötig sind, um eine n–elementige Menge zu sortieren.

   (b) Sequentielle Suche: Überlegen Sie sich einen Pseudocode, welcher in einem $n-1$–elementigen Datensatz $A[1], \ldots, A[n-1]$ ein Element x sucht und $j \in \{1, \ldots, n-1\}$ ausgibt, falls $x = A[j]$ und n sonst. Machen Sie bei Ihrem Algorithmus eine

      i. best–case Analyse (Es werden Eingaben betrachtet, die minimale Laufzeiten (wie groß sind diese?) liefern).

      ii. worst–case Analyse (Es werden Eingaben betrachtet, die maximale Laufzeiten (wie groß sind diese?) liefern).

      iii. average–case Analyse (Es wird der Mittelwert der Laufzeiten für alle Eingaben berechnet).

2. Das Horner–Schema dient zur Auswertung von Polynomen. Grundidee ist die Umformung

$$p(x) = \sum_{k=0}^{n} a_n x^n = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + xa_n)\ldots)).$$

   (a) Wiederholen Sie die genaue Funktionsweise des Horner–Schemas und überlegen Sie sich einen Pseudocode für das Horner–Schema.

   (b) Überlegen Sie sich einen Pseudocode für die direkte Auswertung von Polynomen (direktes Einsetzen).

   (c) Vergleichen Sie die Schrittzahlen der beiden Codes.

3. (a) Vergleichen Sie das asymptotische Verhalten von $f(n) = n!$ und $g(n) = (n+2)!$, d.h. überlegen Sie sich ob eine (welche) der Funktionen ein o, O, ω, Ω, Θ der anderen Funktion ist.

   (b) Vergleichen Sie das asymptotische Verhalten von $f(n) = n^{\log_2 4}$ und $g(n) = 3^{\log_2 n}$, d.h. überlegen Sie sich ob eine (welche) der Funktionen ein o, O, ω, Ω, Θ der anderen Funktion ist.

   (c) Zeigen Sie anhand der Definition, dass für asymptotisch nichtnegative Funktionen f und g die Beziehung

$$\Theta(f(n) + g(n)) = \max(f(n), g(n))$$

   gilt.

   (d) Folgt aus $f(n) = O(g(n))$, dass $2^{f(n)} = O(2^{g(n)})$?

   (e) Ist $f(n) = O(f(n)^2)$?

   (f) Finden Sie eine Funktion f, sodass weder $f(n) = O(n)$ noch $f(n) = \Omega(n)$.

   (g) Weisen Sie nach, dass $\sum_{j=0}^{n} a^j = \Theta(a^n)$.

4. Weisen Sie nach, dass $\sum_{k=1}^{n} 1/k = O(\ln(n))$, indem Sie $\sum_{k=1}^{n} 1/k$

   (a) abschätzen durch in $N = \lfloor \ln(n) \rfloor$ Blöcke der Gestalt $\sum_{j=0}^{2^i-1} \frac{1}{2^i+j}$, $i = 1, \ldots, N$ und anhand dieser Aufteilung $\sum_{k=1}^{n} 1/k \leq \ln(n) + 1$ verifizieren

   (b) mit dem Cauchy'schen Integralkriterium abschätzen.

5. Bekanntlich sind die Fibonacci Zahlen rekursiv definiert durch $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$.

   (a) Zeigen Sie mit vollständiger Induktion, dass

$$F_n = \frac{\Phi^n - \Phi'^n}{\sqrt{5}}, \tag{1}$$

   wobei $\Phi = \frac{1+\sqrt{5}}{2}$ und $\Phi' = \frac{1-\sqrt{5}}{2}$.

(b) Schließen Sie aus (a) auf das asymptotische Wachstum von F_n.

(c) Angenommen, man würde die Fibonacci Zahlen anhand eines Algorithmus berechnen, welcher auf der rekursiven Definition beruht, d.h. einem Algorithmus der Gestalt
$define\ F_n$:
$if\ n \leq 2$:
$return\ 1$
$else$:
$return\ F_{n-1} + F_{n-2}$
Offenbar hätte man das Problem, dass in der Rekursion niedrigere Fibonacci Zahlen oft (neu) berechnet werden müssten.
Überlegen Sie sich, wie oft beispielsweise für F_{20} die Zahl F_3 berechnet werden müsste. Verallgemeinern Sie diese Beobachtung und folgern Sie daraus, dass allgemein die Schrittzahl zur rekursiven Berechnung von F_n exponentiell wächst.

6. (Fortsetzung Fibonacci): Neben der direkten Formel (1) zur Berechnung von F_n (in welcher Rundungsfehler problematisch werden können), existiert die auf Matrixmultiplikation beruhende Formel:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n .$$

(a) Verifizieren Sie diese Formel mit vollständiger Induktion.

(b) Überlegen Sie sich einen rekursiven Algorithmus, der zur Berechnung von $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ nur logarithmisch viele Schritte benötigt.

7. (a) Zeigen Sie mit vollständiger Induktion, dass ein Algorithmus, dessen Laufzeit $T(n)$ (in Abhängigkeit von n, der Anzahl der Elemente der Eingabemenge, welche eine Zweierpotenz sei) der Rekursion

$$T(n) = \begin{cases} 2 & \text{für } n = 2, \\ 2T(n/2) + n & \text{für } n = 2^k, k = 2, 3, 4, \ldots . \end{cases}$$

genügt, $T(n) = n \log_2 n$ erfüllt.

(b) Insert Sort kann folgendermaßen rekursiv definiert werden. Um das n–elementige Array $A[1, \ldots, n]$ (d.h., das n–Tupel $(A[1], A[2], \ldots, A[n])$) zu sortieren, sortiert man (rekursiv) das $n-1$–elementige Array $A[1, \ldots, n-1]$ und sortiert in diese das n–te Element $A[n]$ ein usw. Überlegen Sie sich einen Pseudocode für diese Prozedur und bestimmen Sie eine Rekursion für die Laufzeit dieses Algorithmus.

8. Unimodal Search: Ein n–elementiges Array $A[1, \ldots, n]$ heißt unimodal, falls es mit einem streng monoton wachsenden Abschnitt $A[1, \ldots m]$ beginnt, an welchen ein streng monoton fallender Abschnitt $A[m+1, \ldots, n]$ anschließt. Speziell ist das Element $A[m]$ das maximale Element des Arrays.
Geben Sie einen Divide and Conquer Algorithmus an, welcher in $O(n \log_2 n)$ Schritten das maximale Element eines unimodalen Arrays liefert.