Aufgabensammlung zum Buch „Algorithmen und Datenstrukturen“ (Kapitel 2)

1 Aufgaben aus dem Buch

Zu folgenden Aufgaben, die direkt aus dem Buch entnommen sind, gibt es an der Universität Freiburg am Lehrstuhl Ottmann Musterlösungen. In der Version mit Lösungen sind diese angegeben. Hinter der fortlaufenden Aufgabennummer steht in Klammern die Nummer der Aufgabe im Buch.

Aufgabe 1 (Aufgabe 2.1):

Sortieren Sie die unten angegebene, in einem Feld a gespeicherte Schlüsselfolge mit dem jeweils angegebenen Verfahren und geben Sie jede neue Belegung des Feldes nach einem Schlüsseltausch an.

1. 40–15–31–8–26–22; Auswahlsort
2. 35–22–10–51–48; Bubblesort.

Aufgabe 2 (Aufgabe 2.3):

Geben Sie für die unten angegebenen Zahlenfolgen jeweils mit Begründung die Laufzeit der Sortierverfahren Auswahlsort, Einfügesort und Bubblesort in Groß-Oh-Notation an.

1. $1, \frac{N}{2}+1, 2, \frac{N}{2}+2, \ldots, \frac{N}{2}, N$ (N gerade)
2. $N, 1, N-1, 2, N-2, 3, \ldots, N-\frac{N}{2}+1, \frac{N}{2}$ (N gerade)
3. $N, 1, 2, 3, \ldots, N-1$
4. $2, 3, 4, \ldots, N, 1$

Aufgabe 3 (Aufgabe 2.5):

Gegeben sei das Array a von neun Elementen mit den Schlüsseln

$$41 \quad 62 \quad 13 \quad 84 \quad 35 \quad 96 \quad 57 \quad 28 \quad 79.$$

Geben Sie alle Aufrufe der Prozedur *quicksort* und die Reihenfolge ihrer Abarbeitung an, die als Folge eines Aufrufs von *quicksort*$(a, 1, 9)$ im Hauptprogramm für obiges Array auftreten.

Aufgabe 4 (Aufgabe 2.11):

Sortieren Sie die angegebene Zahlenfolge durch Fachverteilung. Geben Sie dabei die Belegung der einzelnen Fächer nach jeder Verteilphase an und jeweils die Folge, die nach einer Sammelphase entstanden ist.

$$1234, 2479, 7321, 4128, 5111, 4009, 6088, 9999, 7899, 6123,$$
$$3130, 4142, 7000, 0318, 8732, 3038, 5259, 4300, 8748, 6200$$

2 Ähnlich Aufgaben

Bei den folgenden Aufgaben handelt es sich um Aufgaben die an der ETH Zürich, am Institut für Theoretische Informatik und an der Universität Freiburg im Institut für Informatik in diversen Vorlesungen gestellt wurden. Inhaltlich sind diese Aufgaben mit dem behandelten Stoff im Buch verwandt. Zu allen Aufgaben gibt es Musterlösungen, die allerdings nur in der Version mit Lösungen enthalten sind.

Aufgabe 5:

McDiarmid and Reeds Variante des BOTTOM-UP-HEAP-SORT, kurz MDR-HEAPSORT, ermöglicht es, weitere Vergleiche einzusparen, indem alte Informationen verwendet werden. Es wird allerdings ein zusätzliches Array `Info` benötigt, das diese alten Informationen verwaltet. Die möglichen Belegungen von `Info[j]` sind $unbekannt$, $links$ und $rechts$ (festgelegt durch die Konstanten `UNKNOWN`, `LEFT` und `RIGHT`) und können mit zwei Bits codiert werden. Ist `Info[j] = LEFT`, dann enthält das linke Kind ein kleineres Element als das rechte, im Falle `Info[j] = RIGHT` ist es entsprechend genau umgekeht. Wenn `Info[j] = UNKNOWN` gilt, ist keine Dominanz der Kinder untereinander bekannt.

Erweitern Sie `Bottom-Up-Heapsort` um diesen Ansatz, indem Sie die Prozeduren *LeafSearch* und *Interchange* entsprechend modifizieren.

Aufgabe 6:

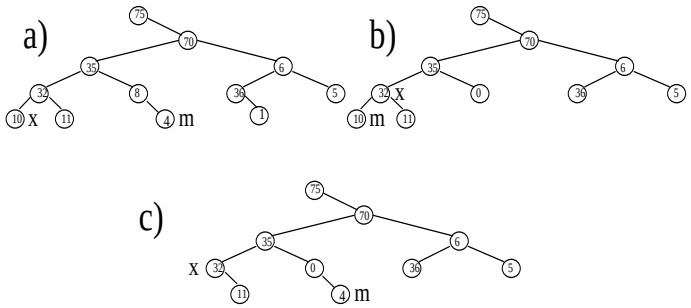
WEAK-HEAP-SORT benötigt im *MergeForest* Schritt im ungünstigen Fall $\lceil \log(m+1) \rceil$ und im günstigen Fall $\lceil \log(m+1) \rceil - 1$ viele Vergleiche. Die Gesamtanzahl der Vergleiche ist durch $n-1$ plus die Schlüsselvergleiche in der Sortierphase durch *MergeForest*(m), für $m = n-1, \ldots, 2$, bestimmt.

In dieser Aufgabe soll EXTERNAL-WEAK-HEAP-SORT in Java programmiert werden, ein Ansatz, der jeweils den best case erzwingt (und demnach mit $n \log n - 0.9n$ Vergleichen) auskommt. Dabei gehen wir davon aus, daß wir einen zusätzlichen Datenbereich $A[n \ldots 2n-1]$ besitzen, in dem wir die entnommenen Wurzelelemente schreiben können. Dazu nutzen wir ein Boolesches Array `Active`, das die derzeit aktiven Index-Positionen verwaltet. Falls `Active[i]` wahr ist, dann wurde der Schlüssel an Position i noch nicht in den Bereich $A[n \ldots 2n-1]$ geschrieben und ist demanch noch aktiv für den betrachteten Weak-Heap. Die Positionen $A[n \ldots 2n-1]$ werden in einer Schleifenvariablen $call$ verwaltet.

In Pseudo-Code gestaltet sich die äußere Schleife nun wie folgt

```
procedure EXTERNAL-WEAK-HEAPSORT
  i := n-1, call := 0
  WHILE i > 1
    WHILE (Active[i] = 0) i := i - 1
    call := call + 1
    MergeForest(i)
```

Wir betrachten die sich ergebenen Fälle in der Sortierphase nun genauer. In den Fällen b) und c) der folgenden Abbildung liegen m und x auf unterschiedlichen Leveln, d.h. $depth(x) = depth(m) - 1 = \lceil \log(m+1) \rceil - 1$. In Fall a) gilt hingegen $depth(x) = depth(m) = \lceil \log(m+1) \rceil$.



Demnach können wir immer dann ein Vergleich einsparen, wenn wir anstatt dem m-Element am Ende des Arrays, ein Element am Ende des speziellen Pfades wählen und den entsprechende Index nach dem *MergeForest*-Schritt *deaktivieren*.

Zusatzaufgabe (keine Bewertung): Überlegen Sie weiterhin, ob sich EXTERNAL-WEAK-HEAP-SORT analog zu QUICK-HEAPSORT in eine QUICKSORT Variante wandeln läßt und wie sich der zusätzliche Speicherbedarf reduzieren läßt.

Aufgabe 7:

Auswahlsort ist ein einfacher Vorläufer von Heapsort. Gegeben sei ein Elementarray A von n vergleichbaren Objekten. Der Algorithmus sucht für $j = 1, \ldots, n-1$ in dem Array von der Stelle j aus das kleinste Element und tauscht es an die Stelle j.

1. Bestimmen Sie die Anzahl der Vergleiche im best, worst und average case.
2. Besimmen Sie die Anzahl von Bewegungen im best, worst und average case.

Aufgabe 8:

Countingsort ist ein Sortierverfahren ohne Vergleiche. Gegeben ist ein Array `A[0..n-1]` mit `A[i]` $\in \{0, \ldots, k-1\}$. Ausgabe ist ein sortiertes Array `B[0..n-1]`. Genutzt wird ein Array `C[0..k-1]`, in dem die Anzahlen der Elemente der Größe `A[i]` innerhalb gezählt werden. Diese Information wird dann zur Extraktion der sortierten Reihenfolge genutzt.

Schreiben Sie ein Pseudocodeprogramm für *Countingsort* und zeigen Sie, daß die Laufzeit in $O(n + k)$ liegt.

Aufgabe 9:

Sortieren Sie die Elemente 4,7,8,9,0,1,5,3,2,6

1. mittels Quicksort (nutzen Sie dabei die erste Zahl als Pivotelement).
2. mittels Heapsort.

Zur Aufgabenlösung zugelasse Arbeitsmaterialien sind nur Zettel und Stift.

Aufgabe 10:

Sei S eine Menge von n paarweise verschiedenen Elementen. Ein α-*Goodsplitter* liefert für $\alpha \in [1/2, 1)$ (unabhängig von S) ein Element x in der Menge S mit
$|\{y \in S |\ y < x\}| \geq (1-\alpha)n$ und $|\{y \in S |\ y > x\}| \geq (1-\alpha)n$.

In der Vorlesung haben Sie gesehen, daß der Median-der-Mediane ein deterministischer α-*Goodsplitter* mit $\alpha \approx 7/10$ ist. Wir stellen nun einen randomisierten α-*Goodsplitter* vor:

Goodsplitter$_\alpha(S)$
repeat
Wähle $x \in S$ zufällig
until x ist ein α-*Goodsplitter*

Zu zeigen ist, daß *Goodsplitter*$_\alpha$ im Mittel $1/(2\alpha - 1)$ Versuche benötigt, um ein gutes Splitelement I zu finden. Die Wahrscheinlichkeit $p(I = i)$, daß man dazu i Versuche benötigt, ist $[2(1-\alpha)]^{i-1} \cdot (1-2(1-\alpha))$.

1. Erklären Sie, wie sich dieser Wert zusammensetzt.
2. Berechnen Sie den Erwartungswert $E[I] = \sum_{i \geq 1}(i \cdot p(I = i))$ für die Anzahl der Versuche. Es ist dabei nützlich, $\beta = 2(1 - \alpha)$ zu setzen und sich der Berechung von $\sum_{i \geq 1} i/2^i$ zu erinnern (HEAPSORT-Generierungsphase).
3. Folgern Sie: *Goodsplitter*$_\alpha(S)$ benötigt eine erwartete Zahl von höchstens $n/(2\alpha - 1)$ Vergleichen.

Aufgabe 11:

Die folgende Prozedur *Splitsort* nutzt die Bestimmung des Medians durch den Aufruf *Select*$(S, \lceil n/2 \rceil)$ aus der vorigen Aufgabe.

Splitsort(S)
$x \leftarrow$ *Select*$(S, \lceil n/2 \rceil)$
$S_< \leftarrow \{y \in S |\ y < x\}$
$S_> \leftarrow \{y \in S |\ y > x\}$
return Splitsort$(S_<, x, S_>)$

Zeigen Sie: Falls *Select* bei der Eingabegröße n höchstens dn Vergleiche braucht (d konstant), dann braucht *Splitsort* höchstens $O(n \log n)$ Vergleiche.
Zusatzaufgabe: (ohne Bewertung) Zeigen Sie, daß für $n = 2^k$ ist die Anzahl der Vergleiche in Splitsort genauer durch $(d + 1) \cdot n \cdot (1 + \log n)$ beschränkt ist. Folgern Sie, daß Splitsort mit dem randomisierten *Goodsplitter* aus Aufgabe 3 im Mittel maximal $\frac{5\alpha - 2\alpha^2 - 1}{(1-\alpha)(2\alpha-2)} \cdot n \cdot (1 + \log n)$ viele Vergleiche benötigt.

Aufgabe 12:

Erweitern Sie Quicksort um

1. die *3-Median-Strategie* (Anleitung dazu im Buch).
2. die Randomisierung zur Auswahl des Pivotelementes.

Aufgabe 13:

Gegeben sei eine Studenten-Datei mit folgenden Einträgen: `Vorname`, `Nachname`, `Hauptfach` und `eMail`.

1. Schreiben Sie eine Klasse `Studi` bestehend aus eben jenen Einträgen und lesen Sie die Datei in einer Funktion der Klasse `InformatikII` ein. Tragen Sie Daten in ein Array `A` vom Typ `Studi` ein.

2. Sortieren Sie aufbauend auf den Sortierrahmen der Vorlesung die Liste nach den Nachnamen mit einem Sortierverfahren Ihrer Wahl. Sie benötigen dabei den lexikographischen Vergleich aus der Klasse `String`.

3. Durch mehrfache Anmeldung zu den Übungen haben sich einige *Doppelte* eingemogelt. Finden Sie diese und geben Sie die bereinigte Liste aus.

Aufgabe 14:

Ein *Weakheap* ist eine Datenstruktur, die durch die Relaxierung der Heapbedingung entsteht. Auch der Weakheap kann als ein binärer Baum aufgefaßt und durch zwei Arrays a und r beschrieben werden. In a_i ist der Schlüssel des i-ten Elementes abgelegt und r_i ist ein einzelnes Bit, dessen Funktion unten erläutert wird. Wir wollen *Weakheaps* zum Sortieren nutzen. Dabei stehen die n zu sortierenden Elemente auf den Positionen $a_0, \ldots, a_{n-1}$.

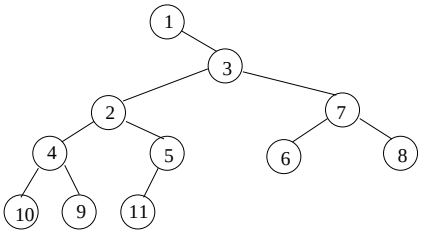


Abbildung 1: Beispiel eines Weak-Heaps.

In einem Weakheap gelten die folgenden Bedingungen: $i)$ Jeder Schlüssel im rechten Teilbaum eines Knotens ist größer als der Schlüssel an der Wurzel des Teilbaumes selbst. $ii)$ Die Wurzel hat kein linkes Kind. $iii)$ Blätter finden sich nur auf den letzten beiden Schichten des Baumes. In der Arrayrepräsentation definieren wir das linke Kind eines Index i als $2i + 1 - r_i$ und das rechte Kind als $2i + r_i$. Durch das Negieren der Boole'schen Werte r_i (hier als Ganzzahlen 0 und 1 verwaltet) ändern wir die Adressierung des rechten und linken Kindes. Der durch i beschriebene Teilbaum wird demnach rotiert.

Die Nachfolger eines Knotens auf dem linken Schenkel des unterliegenden rechten Teilbaumes werden für den Algorithmus als *Enkel* definiert. Im Beispiel von Abbildung 1 bestehen die Enkel von der Wurzel aus den Knoten mit der Beschriftung 3,2,4 und 10. Die inverse Funktion *Großelternteil* (`gp`) zum Arrayindex j läßt sich algorithmisch wie folgt beschreiben: `while(odd(`j`) ==` $r_{j/2}$`) j/=2; return j/2;` Dabei prüft die Bedingung `odd(`j`) ==` $r_{j/2}$, ob j linkes Kind seines Elternteils $j/2$ ist, wobei `odd(`j`)` bestimmt, ob j ungerade ist.

1. Der Aufbau des Weakheaps geschieht dadurch, daß für jeden Index i mit $n - 1 \geq i \geq 1$ der Knoten i mit seinem *Großelternteil* verglichen wird. Ist der Schlüssel dort größer, so wird ein Tausch durchgeführt, d. h. die Funktion `Merge(`i,j`){ if(`a_i`>`a_j`) {` r_j`=1 -` r_j`; swap(`i,j`);} }` rufen wir mit `Merge(gp(i),i)` auf. Schreiben Sie ein Java-Programm `heapify`, das ein ungeordnetes Array in einen Weakheap überführt. Wieviele Schlüsselvergleiche sind dazu notwendig?

2. Zur Sortierung entnehmen wir analog zu Heapsort ein Element an der Wurzel und stellen die Weakheapbedingungen $i) - iii)$ wieder her. Dazu tauschen wir das letzte Element a_m für $n - 1 \geq m \geq 2$ an die Wurzel und laufen die Enkel x der Wurzel hinunter. Sukzessive erfüllen wir im Hinaufgehen durch `Merge(`$0, x$`)` die geforderten Bedingungen, d. h. wir führen für alle m die folgende Funktion `combine` aus:
`combine(`m`){`x`=1;`

`while(`$2x+r_x<m$`)`$x=2x+r_x$`;`
`while(`x`>0){Merge(0,`x`);`$x/=2$`;}}`
Schreiben Sie eine Java-Klasse `WeakHeapSort` und leiten Sie diese von dem in der Vorlesung vorgegebenen Sortierrahmen ab. Vergleichen Sie die Anzahl von Schlüsselvergleichen mit denen von *Quicksort* (ohne bzw. mit 3-Median Strategie).

Aufgabe 15:

Heaps sind einfache Datenstrukturen mit einem hohen Freiheitsgrad bezüglich der Anordnung der Elemente.

1. Zeigen Sie, daß und wie der Pfad zur Stelle n in einem Heap durch die Binärdarstellung von n bestimmt ist.

2. Heaps lassen sich als Vorrangwarteschlangen nutzen. Überlegen Sie, wie man die Datenstruktur anreichern kann, um eine beidseitige Vorrangwarteschlange zu erhalten, die sowohl die Extraktion des Minimum als auch des Maximums in logarithmischer Zeit erlaubt.

3. Die harte Nuß (freiwillige Zusatzaufgabe): Sei $f(n)$ die Anzahl der Heaps mit n paarweise verschiedenen Schlüsseln und s_i die Göße des Teilbaumes zur Wurzel i, $1 \leq i \leq n$. Zeigen Sie, daß $f(n) = n!/\prod_{i=1}^{n} s_i$ gilt. Nutzen Sie dabei die Rekursion $f(n) = \binom{n-1}{|T_1|} f(|T_1|) f(|T_2|)$ mit T_1 bzw. T_2 als der rechte bzw. linke Teilbaum der Wurzel.

Aufgabe 16:

Das in der Vorlesung vorgestellte Bottom-Up-Heapsort Verfahren läßt durch eine geschickte Organisation der Tauschoperationen in der Anzahl der Schlüsselbewegungen noch verbessern. Ändern Sie das in der Vorlesung vorgestellte Java-Programm so ab, daß keine Schlüsselbewegungen während der Sinkphase oder in der Methode `bubbleUp` ausgeführt werden. Die Schlüsselbewegungen sollen in einem dritten Schritt erfolgen und keine weiteren Schlüsselvergleiche erfordern. Die Anzahl der Schlüsselbewegungen soll proportional zur Tiefe der für den Schlüssel gefundenen Position sein. (Hinweis: Merken Sie sich die Indizes der während der Sinkphase betrachteten Knoten in einem Hilfsarray.)

Aufgabe 17:

Jeder kennt die Schwierigkeit, in einer Werkbank zu einer Schraube die passende Mutter zu finden. Wir gehen hierbei von n Schrauben unterschiedlicher Größe und von n korrespondierenden Muttern aus. Der Vergleich der Muttern und der Schrauben untereinander bringt keine neue Information ein. Vergleicht man hingegen eine Schraube mit einer Mutter, so stellt sich heraus, daß die Schraube $i)$ entweder paßt, $ii)$ zu klein, oder $iii)$ zu groß für die ausgewählte Mutter ist.

1. Zeigen Sie, daß jeder Algorithmus zur Lösung des Problems $\Omega(n \log n)$ viele Vergleiche benötigt. Orientieren Sie sich hierbei an dem Beweis der unteren Schranke für allgemeine Sortierverfahren.

2. Geben Sie einen Algorithmus an, der für die Suche die kleinsten Schraube und ihrer korrespondierenden Mutter $2n - 2$ Vergleiche benötigt.

Aufgabe 18:

Kategorisieren Sie die vorgestellten Sortierverfahren gemäß der folgenden 7 Kriterien:

1. Das Sortierverfahren sollte allgemein sein. Objekte aus beliebig geordneten Mengen sollten sortiert werden können. 2. Die Implementation des Sortierverfahrens soll einfach sein. 3. Das Sortierverfahren soll internes Sortieren ermöglichen. Dabei steht neben den Arrayplätzen nur sehr wenig Platz zur Verfügung. 4. Die durchschnittliche Zahl von wesentlichen Vergleichen, d.h. Vergleichen zwischen zu sortierenden Objekten soll klein sein. Sie soll für ein möglichst kleines c durch $n \log n + cn$ beschränkt sein. 5. Die worst-case Zahl von wesentlichen Vergleichen soll klein sein. Sie soll für ein möglichst kleines c' durch $n \log n + c'n$ beschränkt sein. 6. Die Zahl der übrigen Operationen wie Vertauschungen, Zuweisungen und unwesentliche Vergleiche soll höchstens um ein konstanten Faktor größer als die Zahl der wesentlichen Vergleiche sein. 7. Die CPU-Zeit für jede übrige Operation soll klein gegenüber der CPU-Zeit für einen Vergleich zweier komplexer Objekte sein.

Aufgabe 19:

Wir wollen den folgenden Sachverhalt zeigen:

> Sei $k = \lceil \log n \rceil$. Die maximale Anzahl von Schlüsselvergleichen von WEAK-HEAPSORT ist $nk - 2^k + n - 1 \leq n \log n + 0.086013n$.

Der Beweis läuft wie folgt:

> Es gibt durch die Aufrufe `combine`(i) für $i = n-1, \ldots, 2$ höchstens
>
> $$\sum_{i=2}^{n-1} \lceil \log(i+1) \rceil = nk - 2^k \qquad (1)$$
>
> Vergleiche. Zusammen mit den $n-1$ Vergleichen, die zum Aufbau des Heaps benötigt werden, sind dies höchstens $nk - 2^k + n - 1$ Vergleiche für den gesamten Sortieralgorithmus.
>
> Für alle $n \in N$ existiert ein $x \in [0, 1[$, so daß:
>
> $$\begin{aligned} nk - 2^k + n - 1 &= n \log n + nx - n2^x + n - 1 \\ &= n \log n + n(x - 2^x + 1) - 1. \end{aligned}$$
>
> Die reelle Funktion $f(x) = x - 2^x + 1$ ist nach oben durch 0.086013 beschränkt. Damit ist die Anzahl an Vergleichen geringer als $n \log n + 0.086013n$.

Zeigen Sie:

1. $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$ durch vollständige Induktion.
2. $\sum_{i=1}^{n} \lceil \log i \rceil = nk - 2^k + 1$ durch eine geschickte Aufteilung der Summe nach jeweils 2^i Elementen. Sie benötigen dann zur Lösung die Gleichung aus dem ersten Aufgabenteil.
3. Die reelle Funktion $f(x) = x - 2^x + 1$ ist für $x \in [0, 1[$ nach oben durch 0.086013 beschränkt.

Aufgabe 20:

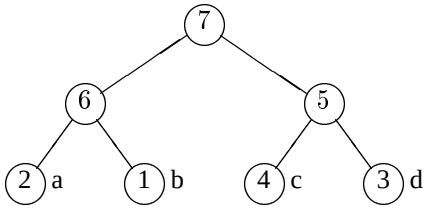
1. Erstellen Sie aus der Folge
   $$1, 3, 5, 6, 2, 7, 4$$
   einen Max-Heap in Array-Darstellung. Verwenden Sie das in der Vorlesung vorgestellte Verfahren durch iteriertes Versickern. Geben Sie die dabei entstehenden Folgen an. Wieviel Schlüsselvergleiche sind ingesamt erforderlich?

2. Entfernen Sie das Maximum des Max-Heaps

$$7, 4, 6, 3, 2, 5, 1$$

und stellen Sie erneut die Heapbedingung durch Versickern her. Geben Sie den entstehenden Heap an. Wieviele Schlüsselvergleiche werden durchgeführt?

3. Entfernen Sie das Maximum des Max-Heaps (in Baum-Darstellung)



und stellen Sie die Heapbedingung nach dem Verfahren Bottom-up-Heapsort her. Welches Blatt des Baumes wird während des Versickerns besucht? Welche Schlüssel werden beim Versickern miteinander verglichen?

Aufgabe 21:

1. Das Quicksort-Verfahren läßt sich rekursiv formulieren. Jeder Rekursionsschritt besteht dann aus der Wahl des Pivotelementes und einem Aufteilungsschritt. Führen Sie einen solchen Aufteilungsschritt konkret aus. Nehmen Sie dazu an, daß der aufzuteilende Bereich durch die in einem Array gespeicherten Zahlen

$$41, 62, 13, 84, 35, 96, 57, 28, 79, 43$$

gegeben ist. Pivotelement sei das Element am rechten Ende des aufzuteilenden Bereiches.

Geben Sie Zwischenschritte an, die unmittelbar vor und nach einem Tausch zweier Arrayelemente entstehen, und geben Sie dabei die Positionen der Zeiger an.

2. Geben Sie Laufzeit von Quicksort im Worst-Case und im Best-Case in Abhängigkeit der Anzahl der zu sortierenden Zahlen an.

3. Geben Sie die Laufzeit von Quicksort im Average-Case an, wenn man als Pivotelement stets das rechte Element des zu unterteilenden Bereiches wählt und annimmt, daß jede möglicher Eingabesequenz der Schlüssel $k_1, \ldots, k_n$ gleich wahrscheinlich ist.

4. Was unterliegt beim randomisierten Quicksort dem Zufall? Geben Sie die erwartete Laufzeit des randomisierten Quicksorts an.

5. Worin besteht der prinzipielle Unterschied in den beiden Komplexitätsresultate von c) und d).

Aufgabe 22:

Seien $L_1, \ldots, L_k$ aufsteigend sortierte Listen der Längen $n_1, \ldots, n_k$. Sei $n = \sum_{i=1}^{k} n_i$.

1. Geben Sie ein Verfahren an, daß die Listen L_i zu einer einzigen sortierten Liste vereinigt und in Zeit $O(nk)$ läuft. Begründen Sie die Laufzeitkomplexität Ihres Verfahrens.

2. Geben Sie ein Verfahren an, daß die Listen L_i zu einer einzigen sortierten Liste vereinigt und in Zeit $O(n \log k)$ läuft. Begründen Sie die Laufzeitkomplexität Ihres Verfahrens. Welches Sortierverfahren ergibt sich, falls jede Liste aus nur einem Element besteht?

Hinweise:

Verwenden Sie zur Beschreibung Ihres Verfahrens Variablen, Zuweisungen, die Standard-Kontrollstrukturen (wie z.B. **while, for, if** etc.) und in der Vorlesung behandelte Datenstrukturen (wie Listen, Heaps, Hashtabellen, balancierte Suchbäume etc.) mit zugehörigen Datenstruktur-Operationen (wie *insert, append, delete, delete-min, is-empty, search* etc.). Für eine Liste L stehen die Methoden

L.remove-first(), die das erste Element von L liefert und aus L entfernt (Zeit $O(1)$),

L.is-empty(), die testet, ob L leer ist (Zeit $O(1)$),

L.append(e), die das Element e an das Ende der Liste L anfügt (Zeit $O(1)$), und

L.merge(L'), die die Elemente der sortierten Liste L' in die sortierte Liste L einfügt (Zeit $O(|L| + |L'|)$),

zur Verfügung. Das Verfahren muß nachvollziehbar, aber nicht syntaktisch korrekt sein.

Aufgabe 23:

Mergesort ist ein rekursives Sortierverfahren, mit dem eine Menge A von n Zahlen aufsteigend sortiert werden kann. Wir nehmen an, dass n eine Potenz von 2 ist.

Falls die Menge A höchstens ein Element enthält, stoppt der Algorithmus sofort, da diese Folge bereits sortiert ist. Andernfalls zerlegt Mergesort die Menge A in zwei gleichgrosse[1] disjunkte Teilmengen A_l und A_r. Diese beiden Mengen werden rekursiv mit Mergesort sortiert. Anschliessend werden die beiden Mengen A_l und A_r zu einer sortierten Folge verschmolzen, indem sie beide von vorne durchlaufen werden und das jeweils kleinere Element in die Ergebnis-Folge geschrieben wird.

Analysiere die Worst-Case-Laufzeit für Mergesort. Stelle dazu zunächst eine Rekursionsgleichung für die Anzahl der Vergleiche auf, die Mergesort durchführt. Löse anschliessend diese Rekursionsformel, um die Laufzeit zu bestimmen.

Aufgabe 24:

Eine aufsteigend sortierte Folge $(1, 2, 3, \ldots, 27)$ ist eine Worst-Case-Eingabe für Quicksort. Gib eine Best-Case-Eingabe für Quicksort mit den Zahlen $1, \ldots, 32$ an.

Aufgabe 25:

Quicksort führt im Mittel nur $O(n \log n)$ Vergleiche durch, um n Zahlen aufsteigend zu sortieren. Im Worst-Case werden jedoch $\Theta(n^2)$ Vergleiche benötigt.

Eine Worst-Case-Eingabe ist z.B. eine Folge von Zahlen, die bereits aufsteigend sortiert ist. Könnte man Quicksort so verändern, dass es manche Eingaben, die bereits teilweise sortiert sind, geschickter verarbeitet?

[1] Hier verwenden wir, dass n eine Potenz von 2 ist. Andernfalls müsste man hier zwei etwa gleichgrosse Mengen A_l und A_r bilden. Dadurch wird dann die Analyse etwas komplizierter, was wir uns ersparen wollen.

Aufgabe 26:

Gegeben sei die Schlüsselfolge $\sigma :=< X, C, B, D, M, S, Z, U, F >$, die in einem Array a gespeichert ist. Die Folge soll alphabetisch aufsteigend sortiert werden.

1. Zeichne den Heap, der sich in Heapsort nach der Heap-Creation-Phase für die Schlüsselfolge σ ergibt.
2. Sortiere die Schlüsselfolge σ aufsteigend mit Heapsort. Gib dazu für jeden Zwischenschritt der Heap-Selection-Phase den jeweiligen Inhalt des Arrays a an.
3. Berechne für die Schlüsselfolge $\sigma :=< X, C, B, D, M, S, Z, U, F >$ die Anzahl der Inversionen und die Anzahl der Runs.

Aufgabe 27:

Geben Sie für das folgende Programmstück, das eine Zahlenfolge der Länge n sortiert (gegeben im Integer-Array A), den Zeitaufwand in Abhängigkeit von n im besten und im schlechtesten Fall in O-Notation an. Geben Sie jeweils eine Beispielfolge und eine kurze Begründung dafür an.

```
boolean vertauschung = false;
int temp;
do {
     vertauschung = false;
     for (int i=0; i<n-1; i++)
     {
       if (A[i] > A[i+1])
       {
         temp = A[i+1];
         A[i+1] = A[i];
         A[i] = temp;
         vertauschung = true;
       }
     }
   } while (vertauschung);
```