Datenstrukturen & Algorithmen  Solution of Sheet 4  FS 14

Solution 4.1  Questions about Sorting Algorithms.

a) A sequence $f_1, f_2, \ldots, f_n$ is called a Min-Heap, if $f_i \leq f_{2i}$ for every $i$ with $2i \leq n$ and $f_i \leq f_{2i+1}$ for every $i$ with $2i + 1 \leq n$. For the sorted sequence, we have $f_i \leq f_j$ for every $j \geq i$, therefore the sorted sequence is indeed a Min-Heap.

b) Because of the heap property $f_i \leq f_{2i}$ for every $i$ with $2i \leq n$, the largest element can only be stored in positions $i$ where $2i > n$, i.e. it is located in the second half of the array.

c) Even naive implementations of insertion sort and bubble sort are already stable. Merge sort can easily be made stable, if we remember to take the leftmost element when a tie is encountered while merging. There is no easy way to make selection sort, quicksort and heapsort stable.

d) Selection sort, insertion sort, bubble sort and heapsort work directly on the array to be sorted, and are therefore in-situ. Quicksort requires between $\Omega(\log n)$ and $O(\log n)$ additional space for storing the recursive function calls. This additional space is not used for elements in the sequence, therefore Quicksort is also in-situ. For merge sort, parts of the array must be copied for the merging. There are (complicated) methods to perform the merging in-situ, but no such methods can be implemented as simple modifications of the standard algorithm.

e) If no two numbers in the sequence are equal, we need $\Omega(\log n)$ digits (depending on the number system, e.g. bits) for the representation of numbers. This means, among other things, that Radixsort requires $\Omega(\log n)$ iterations and therefore its overall running time is $\Omega(n \log n)$. In general it hence makes no sense to refer to Radixsort as a “linear-time sorting algorithm”. Only when we restrict the number of digits to a constant we get a running time of $\Theta(n)$ for Radixsort, because the constant “disappears” in the asymptotic notation. However, this means that there have to be numbers in the input sequence that appear very often.

Solution 4.2  Various topics.

a) $x = 1572, y = 8687$ or $x = 8786, y = 7215$

b) Sequence: 5, 4, 3, 2, 1

c) Depending on the implementation of insertion sort, there exist two possible solutions: (16, 9), (8, 16), (8, 9), (13, 16) or (16, 9), (8, 9), (13, 8), (13, 9).

Solution 4.3  Extended Heaps.

The function Sift-Down was presented in lecture. It exchanges the node containing a key $k$ with the smaller of its successors until both successors contain a key greater or equal than $k$, or $k$ does not have any successors any more. Analogously we define a function Bottom-Up that exchanges a node of a key $k$ with its predecessor until the predecessor contains a key smaller or
equal than $k$, or $k$ is stored in the root of the heap. Let $A$ be the array storing the heap, and $i$ be the initial position of $k$ in $A$. In pseudocode we get the following function:

\[ \text{Bottom-Up}(A, i) \]

\begin{verbatim}
1 while $i \geq 2$ do
   \hspace{1em} $j \leftarrow \lfloor i/2 \rfloor$
   \hspace{1em} if $A[j] \leq A[i]$ then STOP
   \hspace{2em} $A[i]$ has a predecessor
   \hspace{1em} else Vertausche $A[i]$ und $A[j]$; $i \leftarrow j$
\end{verbatim}

Additionally we maintain a variable $n$ that indicates the number of keys stored in the heap, and that is initialized with 0. Since in the above algorithm we have $i \in \{1, \ldots, n\}$, its running time is bounded by $O(\log n)$.

a) **Min**: $A$ is a Min-Heap. Thus, **Min** simply returns $A[1]$ (or an error message if $n = 0$). The running time is constant, i.e. in $O(1)$.

b) **REPLACE**($i, k$): Let $k' = A[i]$ be the key to be replaced. We distinguish two cases:

1. **Case**: $k < k'$. Then the subtree rooted at $A[i]$ is still a (Min-)Heap after the substitution (the smallest element just got smaller). However, $A[1..n]$ might not be a heap any more because $k$ might be smaller than its predecessor. To restore the heap property, we invoke **Bottom-Up**($A, i$).

2. **Case**: $k > k'$. Then, after the substitution, $A[i]$ might store a key that is larger than its predecessor, thus the subtree rooted at $A[i]$ might not be a heap any more ($k$ might be larger than its successors). To restore the heap property, we invoke **Sift-Down**($A, i$).

Therefore the operation **REPLACE**($i, k$) can be implemented to run in time $O(\log n)$.

c) **INSERT**($k$): We increase $n$ by 1 and store the new key $k$ in the entry $A[n]$. Now it might happen that $A$ is not a Min-Heap any more, because the new key might be smaller than its predecessor. To restore the heap property it suffices to invoke **Bottom-Up**($A, n$). The running time is in $O(\log n)$.

d) **DELETE**($i$): We invoke **REPLACE**($i, A[n]$), i.e., the key stored at position $i$ is replaced by $A[n]$. Now the key stored in $A[n]$ occurs once too often, thus we decrease $n$ by 1 (the size of the heap shrinks by 1 and the redundant elements gets cut off). The operation has the same running time than **REPLACE**($i, k$), thus it is in $O(\log n)$.

**Solution 4.4**  \textit{Lower bounds / Algorithm design.}

a) Let the given coins be $M_1, \ldots, M_9$. The following decision tree describes a strategy that determines the false coin with exactly two weighings. Each node describes which coins lie in the left and in the right scalepan. After each weighing there are three possible outcomes: the coins on the left are either lighter ($<$), have the same weight ($=$) or are heavier ($>$) than the coins on the right. After each step the false coin is on the side with highest weight.
b) Let the given coins be $M_1, \ldots, M_n$. For $n = 1$ we found the false coin and we are finished. Otherwise, we split the coins into three groups $G_1 := \{M_1, \ldots, M_{n/3}\}$, $G_2 := \{M_{n/3+1}, \ldots, M_{2n/3}\}$ and $G_3 := \{M_{2n/3+1}, \ldots, M_n\}$, and we put $G_1$ in the left pan and $G_2$ in the right pan. If the coins on the left are heavier, then the false coin is in $G_1$ and we continue with this group. Similarly we continue with $G_2$ if the right side is heavier, or with $G_3$, if both sides are equally heavy.

It remains to show that in this way we really have only $\log_3(n)$ weighings. We show by induction over $m \in \mathbb{N}$ that the following statement is correct: For $n = 3^m$, the above procedure performs exactly $m$ weighings to find the false coin.

**Base step** ($m = 1$): For $m = 1$ we have $n = 3$, and exactly one weighing is required to find the false coin.

**Inductive hypothesis:** Let the statement be true for $m$, i.e. for $n = 3^m$, the above procedure needs exactly $m$ weighings to find the false coin.

**Inductive step** ($m \rightarrow m + 1$): Let $n = 3^{m+1}$. We divide the coins into three equally-sized groups and continue with the heaviest group. This group contains exactly

$$n/3 = 3^{m+1}/3 = 3^m$$

many coins and by the inductive hypothesis, $m$ weighings are sufficient to find the false coin. Thus the method requires a total of $m + 1$ weighings.

We now observe that for $m = \log_3(n)$, we have $n = 3^m$ and therefore $\log_3(n)$ weighings are sufficient.

c) Let the given coins be $M_1, \ldots, M_n$. Every possible algorithm can be described by a decision tree like the one on the previous page. The maximum depth $T$ of this tree is the maximum number of weighings in the worst case. Therefore we must show that the depth $T$ of every valid decision tree is at least $\log_3(n) - 1$.

First, we observe that each weighing possesses only three possible outcomes. Therefore any valid decision tree is a ternary tree, i.e. each node has at most three children. We use induction to show that there exist at most $3^k$ nodes at depth $k$.

**Base step** ($k = 0$): At depth 0 there exists a single node, namely, the root of the tree.
**Inductive hypothesis:** We assume that the assertion is true for $k$, i.e., there exist at most $3^k$ nodes at depth $k$.

**Inductive step** ($k \rightarrow k + 1$): Consider the nodes at depth $k + 1$. Such a node can be reached by an edge only from a node at depth $k$. Each node has at most three children and by the inductive hypothesis, there are at most $3^k$ nodes at depth $k$. So there exist at most $3 \cdot 3^k = 3^{k+1}$ nodes at depth $k + 1$.

In some nodes of the tree the algorithm outputs a result, since the available information is sufficient to identify the false coin. Since each result $M_1, \ldots, M_n$ is possible, there are $n$ possible situations that the algorithm has to distinguish. For a correctly working algorithm there must exist a node for each situation. Therefore we have

$$n \leq \text{Number of nodes} \leq \sum_{k=0}^{T} 3^k = \frac{3^{T+1} - 1}{2} < 3^{T+1},$$

and thus we get

$$\log_3(n) < T + 1 \iff T > \log_3(n) - 1.$$  \hspace{1cm} (3)

Thus, the depth of each decision tree is at least $\log_3(n) - 1$, and therefore every correct algorithm must perform at least these many weighings in the worst case.

**Note:** It is easy to see that every result of the algorithm must be stored in a leaf (and not in an inner node). Furthermore it can easily be shown that every ternary tree of depth $T$ has at most $3^T$ many leaves, thus we get $n \leq 3^T \iff T \geq \log_3(n)$. Therefore the best algorithm must perform at least $\log_3(n)$ many weighings, i.e., the algorithm presented in b) is optimal.