Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department of Computer Science
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

3rd November 2016

Data Structures & Algorithm — Solutions to Sheet 6 — AS 16

Lösung 6.1 *Comparison of Sorting Algorithms.*

	`bubbleSort`		`insertionSort`	
	min	max	min	max
Comparisons	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Input sequence	every	every	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Swaps	0	$\Theta(n^2)$	0	$\Theta(n^2)$
Input sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Additional memory	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Input sequence	every	every	every	every

	`selectionSort`		`quicksort`	
	min	max	min	max
Comparisons	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$
Input sequence	every	every	$(\star)$	$1, 2, \ldots, n$
Swaps	0	$n-1$	$\Theta(n)$	$\Theta(n \log n)$
Input sequence	$1, 2, \ldots, n$	$n, 1, 2, \ldots, n-1$	$1, 2, \ldots, n$	$(\star)$
Additional memory	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
Input sequence	every	every	$(\star)$	$1, 2, \ldots, n$

$(\star)$: An appropriate sequence is not easy to write. The sequence must be designed such that every chosen pivot halves the area that will be stored. For $n = 7$, the sequence will be $4, 5, 7, 6, 2, 1, 3$.

Solution 6.2 *Questions About Sorting Algorithms.*

a) Because of the heap property $A[\lfloor i/2 \rfloor] > A[i]$ for all $i \geq 2$, the smallest key can only be stored in positions i with $i > n/2$, i.e. it is located in the second half of the array (more precisely, in the last $\lceil n/2 \rceil$ positions).

b) 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.

c) *Selection sort*, *insertion sort*, *bubble sort* and *Heapsort* work directly on the array to be

sorted, and are therefore in-situ. *Quicksort* needs only $\mathcal{O}(\log n)$ additional space for storing the recursive function calls if we continue with sorting the smaller part recursively after a pivoting step. 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.

d) Even though *Quicksort* has a worst-case running time of $\Theta(n^2)$, with a random selection of the pivot the probability to get the quadratic running time is extremely small. The expected running time of *Quicksort* is $\mathcal{O}(n \log n)$. In addition, we know from the previous part of this exercise that *Quicksort* works in-situ (in contrast to *merge sort*). Furthermore, a much smaller constant is "hidden" in the expected running time of *Quicksort* than in the one of *Merge sort*. However, *Merge sort* is also used in practice, because it is stable (in contrast to *Quicksort*). Therefore, the `sort`-Method of the Java standard library uses a variant of *Quicksort* for sorting arrays of primitive data types (e.g. integer, float, and double), whereas it uses a combination of *Merge sort* and *insertion sort* for arrays of general objects.

Solution 6.3 *Median-of-three Quicksort.*

The basic idea in the construction of such an instance is to place the the largest and the second largest element in each recursive call in such a way that the next recursive call will need to process all the remaining elements. Let $n \in \mathbb{N}$ be an arbitrary integer. We construct an array $A^{(n)}$ of length n, on which Median-of-three Quicksort has a quadratic running time.

Let be $A^{(n)} = (a_1, \ldots, a_n)$, such that $a_n < a_1 < a_2 < \cdots < a_{n-1}$. A possible array that fulfills this condition is $(2, 3, 4, \ldots, n, 1)$ The smallest element is located at the last position and the second smallest element at the second-to-last position. We show the following: The pivoting step of Quicksort partitions $A^{(n)}$ in one array of length 1 and one array $A' = (a'_1, \ldots, a'_{n-2})$ of length $n-2$ that has exactly the same structure as A, i.e. $a'_{n-2} < a'_1 < a'_2 < \cdots < a'_{n-1}$ holds. Especially, the first $n-3$ elements of A' are sorted in ascending order, the smallest element is on the first position and the second smallest element is at the last position.

We consider the pivoting step of $A^{(n)}$. First, we calculate the median of the first element a_1, the middle element $a_{\lceil n/2 \rceil}$ and the last element a_n. As a_n is the smallest element and a_1 the second smallest element of $A^{(n)}$, a_1 will be the pivot element and swapped with a_n. The pass from left stops at position $i = 2$ (that contains the key a_2 that is larger than a_1). The pass from right does not stop before position $j = 1$ (that contains key a_n that is smaller than the pivot element a_1). As $j < i$, the elements a_i and a_j are not swapped. Instead, the pivot element is swapped with a_2 and we get the array

$$(a_n, a_1, a_3, a_4, \ldots, a_{n-1}, a_2),$$

that is partitioned in the array (a_n) of length 1 and the array $A' = (a_3, a_4, \ldots, a_{n-1}, a_2)$. We note that A' has the same structure as $A^{(n)}$. It holds that $a_2 < a_3 < \cdots < a_{n-1}$, that is the first $n-3$ elements are in strictly ascending order, the smallest element is at the last position and the second smallest element at the beginning of the array.

Solution 6.4 *Algorithm design and Lower bounds.*

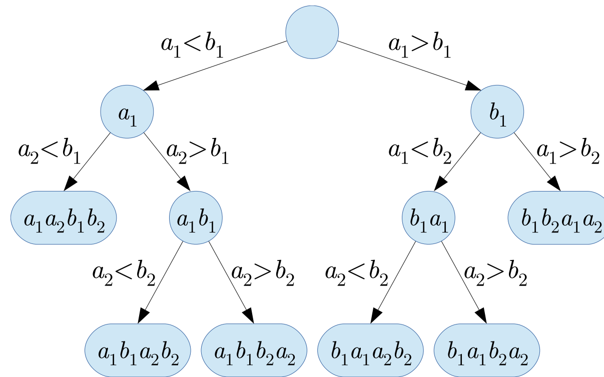
a) The merging routine of Mergesort uses two numbers i and j (initially $i = j = 1$) that point at the next key in A, respectively in B. While $i \leq n$ and $j \leq n$, $A[i]$ is compared to $B[j]$. If $A[i] \leq B[j]$, then $A[i]$ is added to the sorted sequence (and i incremented by 1). Otherwise, $B[j]$ is added to the sorted sequence and j is incremented. With every

comparison either i or j is increased. After some steps either $i > n$ or $j > n$. The keys from A, respectively from B, that have still not been considered are then added to the sorted sequence without further comparisons.

Both i and j are incremented at most $n-1$ times until $i = n$ or $j = n$ (or both). If $i = j = n$ after $2(n-1) = 2n-2$ comparisons, one additional comparison of $A[n]$ and $B[n]$ is sufficient to decide upon the penultimate key of the sorted sequence. On the other hand, if after $2n-2$ comparisons either $i > n$ or $j > n$, the procedure terminates even earlier. Hence, the merging routine of Mergesort compares at most $2n-2+1 = 2n-1$ many keys.

The arrays $A = (1, 3, 5, \ldots, 2n-1)$ and $B = (2, 4, 6, \ldots, 2n)$ are an example for the worst case, where $2n-1$ comparisons are necessary. The upper bound of $2n-1$ comparisons in the worst case is therefore tight.

b) Let $A = (a_1, a_2)$ and $B = (b_1, b_2)$ be the given arrays. In every inner vertex of the decision tree two keys a_i and b_j are compared. For simplicity, we assume that the sorted sequence contains no key twice, i.e. we have either $a_i < b_j$ or $a_i > b_j$ in every comparison. The label of each vertex shows keys that are already added to the complete sequence. The leaves of the decision tree contain the resulting sorted sequences. We get the following decision tree:



c) The resulting sorted array contains $2n$ keys. Exactly n keys are from A and exactly n keys are from B. Every possible array can be encoded by a sequence of 0 and 1, where 0 means that the next key is taken from A and 1 means that the next key is taken from B. The sequence has $2n$ positions and exactly n 0's. There are exactly $\binom{2n}{n}$ possibilities to distribute n 0's to $2n$ positions. Hence, merging two arrays can result in $\binom{2n}{n}$ different arrays of length $2n$.

A correct algorithm for merging two arrays has to compute every such sequence correctly. If we represent such an algorithm by a decision tree as shown in b), the tree has to contain one vertex for every possible resulting array. As there exist $\binom{2n}{n}$ possible resulting arrays, the decision tree of a correct algorithm must have at least that many vertices.

The height of this tree (i.e. the number of comparisons in the worst case plus 1) is at least

$$\log \binom{2n}{n} = \log \left(\frac{(2n)!}{(n!)((2n-n)!)} \right) = \log \Big((2n)! \Big) - 2\log(n!). \tag{1}$$

Considering the estimate from exercise 2.2

$$n \ln n - n \leq \ln(n!) \leq n \ln n - n + \mathcal{O}(\ln n)$$

we get

$$\log\binom{2n}{n} \geq \frac{1}{\ln 2}\Big(2n\ln(2n) - 2n - 2(n\ln n - n + \mathcal{O}(\ln n))\Big) \tag{2}$$

$$= \frac{1}{\ln 2}\Big(2n\ln 2 + 2n\ln n - 2n - 2n\ln n + 2n - \mathcal{O}(\ln n)\Big) = 2n - \mathcal{O}(\ln n). \tag{3}$$

d) We consider the input $A = (a_1, \ldots, a_n) = (1, 3, \ldots, 2n-1)$ and $B = (b_1, \ldots, b_n) = (2, 4, \ldots, 2n)$, and show that *every* correct, general, and deterministic algorithm needs to compare at least $2n-1$ keys.

We consider that every correct algorithm has to compare the keys a_1 and b_1. If an algorithm would not compare these keys, we could swap the keys in a and b and would get the input $A'_1 = (b_1, a_2, a_3, \ldots, a_n)$ and $B'_1 = (a_1, b_2, b_3, \ldots, b_n)$. Note that both A'_1 and B'_1 are still sorted. As a_1 and b_1 are not compared to each other, the algorithm cannot distinguish the inputs (A, B) and (A', B') and outputs an incorrect result for at least one of both inputs.

Similarly, the key a_k needs to be compared both to b_{k-1} and to b_k for every $k \in \{2, \ldots, n\}$. Assume that a_k would not be compared to b_k. As discussed above, we could swap a_k and b_k in A and B and would still have to sorted sequences $A'_k = (a_1, \ldots, a_{k-1}, b_k, a_{k+1}, \ldots, a_n)$ and $B'_k = (b_1, \ldots, b_{k-1}, a_k, b_{k+1}, \ldots, b_n)$, that cannot be distinguished from A and B by the algorithm. Assume that a_k is not compared to b_{k-1}. We could again swap a_k and b_{k-1} in A and B and the algorithm would again not be able to distinguish the resulting sequences from A and B.