Assignment 3 – Selected Model Answers

EXERCISE 1.

(i) Show that any comparison-based algorithm for determining the smallest of n elements requires $n-1$ comparisons.

(ii) Show also that any comparison-based algorithm for determining the smallest and second smallest elements of n elements requires at least $n - 1 + \log n$ comparisons.

Note. You must consider that an arbitrary *algorithm* contains these number of comparisons, whereas on a specific input the number may still be lower (see the proof on the number of comparisons in a sorting algorithm).

(iii) Give an algorithm with this performance.

SOLUTION.

(i) Let $\ell = [x_1, \dots, x_n]$. We build a binary decision tree, where the leaves are labelled with one element x_k and the non-leaf vertices are labelled by comparisons $x_i \leq x_j$. The edge from a non-leaf vertex to its left successor is labelled with x_j, and the edge to its right successor is labelled with x_i. If we have a path $v_0, \dots, v_d$ from the root to a leaf, then all edges on such a path are labelled by x_j that have been discarded for being the minimum. That is, we must have $d \geq n-1$.

(ii) As in the proof of the lower bound of comparison-based sorting algortithms let $\ell = [x_1, \dots, x_n]$. Build a binary decision tree with non-leaf-vertices labelled by comparisons. A leaf is labelled by a pair (x_i, X) with $1 \leq i \leq n$ and $X \subseteq \{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n\}$.

If $v_0, \dots, v_d$ is a path from the root to a leaf, where the label of v_k is $x_{k_i} \leq x_{k_j}$ (for $0 \leq k \leq d-1$), then v_k corresponds to

$$\varphi_k = \begin{cases} x_{k_i} \leq x_{k_j} & \text{if } v_{k+1} \text{ is the left successor of } v_k \\ x_{k_j} < x_{k_i} & \text{if } v_{k+1} \text{ is the right successor of } v_k \end{cases} .$$

If v_d is labelled by (x_i, X), then $\{\varphi_0, \dots, \varphi_{d-1}\}$ imply $x_i = \min\{x_1, \dots, x_n\}$ and $x_i \leq x_j < x$ for all $x \in X$, where $j \neq i$ and x_j is the second smallest element of $\{x_1, \dots, x_n\}$.

Then there are $n \cdot 2^{n-1}$ leaves. There can be at most 2^d leaves in a binary tree of depth d. This implies $2^d \geq n \cdot 2^{n-1}$, i.e. $d \geq \log_2(n \cdot 2^{n-1}) = \log_2 n + n - 1$ as claimed.

(iii) If ℓ is the input list, then we should assume $|\ell| \geq 2$. For $|\ell| = 2$ a single comparison suffices to determine the smallest and second smallest element, for $|\ell| = 3$ three comparisons are required.

For $|\ell| > 3$ split ℓ into two sublists ℓ_1 and ℓ_2, where $|\ell_1|$ is a power of 2 and $|\ell_2| \geq 2$. Then apply the algorithm recursively to ℓ_1 and ℓ_2. Given $x_1^1 \leq x_2^1$ and $x_1^2 \leq x_2^2$ for the resulting smallest and second smallest elements of ℓ_1 and ℓ_2, respectively, two comparisons—first x_1^1 with x_1^2, then the larger of these two with one of the remaining elements—suffice to determine the smallest and second smallest elements of ℓ.

In this way we obtain a linear recurrence equation, and then we can proceed to show that the number of comparisons is $n - 1 + \log n$. We omit further details.

Exercise 2. Design an algorithm to find both the largest and the smallest elements in a list with n elements such that at most $2n-3$ comparisons are needed. You may assume that n is a power of 2.

(i) Determine the exact number of comparisons of your algorithm.

(ii) How does the number of comparisons change, if n is not a power of 2?

Solution.

(i) A straightforward algorithm first scans the list to find the maximum M and the corresponding index i_M, i.e. M is the i_m'th list element. This requires $n-1$ comparisons. Then scan the list again to find the minimum, but ignore the position i_M. This requires additional $n-2$ comparisons. In total, the algorithm requires $2n-3$ comparisons.

A better divide-and-conquer algorithm divides the list into two almost equally sized sublists. Then determine the minimum m_i and the maximum M_i in both sublists. Finally, compare m_1 with m_2 to determine the minimum, and compare M_1 with M_2 to determine the maximum. If $f(n)$ is the number of comparisons needed for a list of length n, then we obtain the recurrence equation $f(n) = f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + 2$.

If n is a power of 2, this equation becomes $a_n = 2a_{n-1} + 2$ for $a_n = f(2^n)$ with $a_0 = 0$ and $a_1 = 1$. As the characteristic polynomial of this recurrence equation is $(x-2)(x-1)$, the general solution becomes $a_n = a \cdot 2^n + b$. The initial conditions give rise to the linear equations $1 = 2a + b$ and $4 = 4a + b$ with the unique solution $a = 3/2$ and $b = -2$. That is, we have $a_n = 32^{n-1} - 2$ and $f(n) = 3/2n - 2$ under the constraint that n is a power of 2 and ≥ 2.

(ii) If n is not a power of 2, we have $2^\ell < n < 2^{\ell+1}$ with $\ell = \lfloor \log_2 n \rfloor$. Let $k = \min\{n - 2^\ell, 2^{\ell+1} - n\}$. Then k is the number of times we descend to a list of length 3, which requires 3 comparisons, two more than a list of length 2. So we have to add $2k$ more comparisons.

Exercise 3. Consider a list L with n elements from a totally ordered set T. A *majority element* is an element $x \in T$ such that there are more than $n/2$ list elements x.

Design an algorithm to decide in linear time, whether L contains a majority element. If such an element exists, the algorithm shall return it as the result.

Solution.

Attempt 1. For a straightforward divide-and-conquer strategy the list is split into two sublists of almost equal length. Then apply the algorithm to both sublists returning a majority value $x \in T$ (if it exists, otherwise $\perp$) and the number c of times x appears in the sublist.

Clearly, for a list with only one element x at position i we can immediately return x and $c = 1$. For a list $[x, y]$ we either return x and $c = 2$ if $x = y$, or $\perp$ (i.e., there is no majority element) and $c = 0$ otherwise. These base cases require time bounded by some constant.

Assume that we have split a list of length n into two sublists of length $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Let the results of the recursive calls of the algorithm for these sublists be (x_1, c_1) and (x_2, c_2).

If $x_1 = x_2$, we can immediately return $(x_1, c_1 + c_2)$, in which case the combination requires constant time.

Otherwise for $x_1 \neq \perp$ we check if $c_1 + \lceil n/2 \rceil - c_2 > n/2$ holds. If not, x_1 cannot be a majority element of the combined list. If yes, we scan the second list. Each time we find x_1, the counter c_1 is incremented. Finally, check if $c_1 > n/2$ holds. If yes, the result is (x_1, c_1) after the modifications of the counter. We proceed analogously for $x_2 \neq \perp$, if $c_2 + \lfloor n/2 \rfloor - c_1 > n/2$ holds. It is impossible that both conditions hold at the same time.

This strategy leads to a recurrence equation, and from the solution of this recurrence we see that the straightforward algorithm has time complexity in $\Theta(n \log n)$. However, we are required to find a linear time algorithm.

Attempt 2. By scanning the list we always couple the i'th and $(i+1)$'th element for all odd i, Replace a pair (x, x) by x—also keep the last element, if the length of the list is odd. Pairs (x, y) with $x \neq y$ are discarded. So the length of the resulting list is at most $\lceil n/2 \rceil$. Then proceed recursively and return $\perp$, if the procedure produces the empty list. If it produces a list with only one element x, then scan the list again to check, if x is really a majority element.

If n is a power of 2, the reduction step is executed at most $\log_2 n$ times, and the time complexity is

$$\sum_{i=0}^{\log_2 n} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\log_2 n} \frac{1}{2^i} = 2n \cdot (1 - (1/2)^{\log_2 n + 1}) \leq 2n \ ,$$

i.e., the algorithm works in linear time.

We show that if L contains a majority element x, then it is also a majority element in the reduced list L'.

If n is even, we can write $n = 2(p+q)$, where p is the number of pairs of the form (y, y), while q is the number of pairs (y, z) with $y \neq z$. If x is a majority element, it occurs more than $(p+q)$ times in L. Hence it occurs at most q times in pairs of the second kind, and more than p times in pairs of the first kind. This implis that there are more than $\lceil p/2 \rceil$ pairs (x, x) if p is even, and at least $\lceil p/2 \rceil$ pairs (x, x) if p is odd, i.e. in both cases we have more than $p/2$ pairs (x, x). Consequently x appears more than $p/2$ times in L', and p is the length of L'.

If n is odd, we append the majority element x to L to get an even length list with x as majority element. Then the same arguments apply.

Exercise 4.

(i) Implement a rotation operation *rotate*(m) by m positions (with $m \in \mathbb{N}$) on lists. Use a divide & conquer algorithm that works in-place.

(ii) Implement a selection operation *select*(k) (with $k \in \mathbb{N}$) on lists to find the k'th smallest element of the list. Use a divide & conquer algorithm.

Solution. See the C++ header and program files in `Ass3_Ex4solution.zip`.