University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, A. Mályusz

Algorithm Theory
Sample Solution Exercise Sheet 1

Due: Friday, 25th of October, 2024, 10:00 am

Exercise 1: Recurrence Relation *(4 Points)*

a) Guess the solution of the following recurrence relation by repeated substitution.

$$T(n) \leq 3 \cdot T\left(\frac{n}{3}\right) + c \cdot n \log_3 n, \quad T(1) \leq c$$

where $c > 0$ is a constant. *(2 Points)*

b) Use induction to show that your guess is correct. *(2 Points)*

Remark: You can assume that n equals 3^j for some $j \in \mathbb{N}$.

Sample Solution

a) First, by using substitution, we achieve a guess for the answer of the inequality.

$$\begin{aligned}
T(n) &\leq 3T(n/3) + cn \log_3 n \\
&\leq 3\big(3T(n/9) + cn/3(\log_3 n - 1)\big) + cn \log_3 n \\
&\leq 9T(n/9) + 2cn \log_3 n - cn \\
&\leq 9\big(3T(n/27) + cn/9(\log_3 n - 2)\big) + 2cn \log_3 n - cn \\
&\leq 27T(n/27) + 3cn \log_3 n - 3cn \\
&\vdots \\
&\leq 3^i T(n/3^i) + icn \log_3 n - \sum_{j=1}^{i-1} j \cdot cn
\end{aligned}$$

By considering $i = \log_3 n$,

$$T(n) \leq cn + \frac{1}{2}cn \log_3^2 n + \frac{1}{2}cn \log_3 n$$

b) Here we use induction to prove our guess achieved by induction.

Induction base: $T(1) \leq c \cdot (1) + \frac{1}{2}c \cdot (1) \log_3^2 1 + \frac{1}{2}c \cdot (1) \log_3 1 = c \leq c$ ✓

Induction hypothesis: $\forall n' < n : T(n') \leq cn' + \frac{1}{2}cn' \log_3^2 n' + \frac{1}{2}cn' \log_3 n'$

Induction step:

$$\begin{aligned}
T(n) &\leq 3T(n/3) + cn \log_3 n \\
&\leq 3\left(c\frac{n}{3} + \frac{1}{2}c\frac{n}{3} \log_3^2 \frac{n}{3} + \frac{1}{2}c\frac{n}{3} \log_3(\frac{n}{3})\right) + cn \log_3 n \\
&\leq cn + \frac{1}{2}cn \log_3^2 n + \frac{1}{2}cn \log_3 n \text{ ✓}
\end{aligned}$$

From the induction we have $T(n) \leq cn + \frac{1}{2}cn \log_3^2 n + \frac{1}{2}cn \log_3 n$, for all $n \geq 1$.

Exercise 2: Fraud Detection *(6 Points)*

Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account. It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Devise an algorithm that answers the following question: *Let $0 < k \leq n$ be an integer, among the collection of n cards, is there a set of more than n/k of them that are all equivalent to one another and if so how many are they?* Your algorithm should use only $O(kn \log n)$ invocations of the equivalence tester. Argue correctness and running time. *(6 Points)*
Remark: Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester.

Bonus Question: Let $k = 2$. What happens in this case in part (a)? Can you speed things up by giving a linear time algorithm that solves the problem for this case? If yes, then argue correctness, and analyze its running time. *(You might get extra Points)*
Hint: try reducing the size of the array to at most half via pairing up elements.

Sample Solution

(a) We will use a divide and conquer approach to solve the problem. For convineance, we assume n to be a power of 2 (the solution can be slightly and easily modified to work with the cases were n is any even or odd number). We also assume our n bank cards input is given in an array A of size n. Moreover, let's define a $k - majority$ element to be an element that appears more than n/k times in the corresponding array (that we are working with) of size n.

Algorithm idea: If $n = 1$, then return $A[1]$. If $n \geq 2$, then divide the input array A into two halves and solve each half recursively to check how many elements appear more than $n/(2k)$ times in that half (ie we are searching for all k-majority elements in each subarray of size $n/2$). If any of the recursive solutions return a k-majority element, then scan each such candidate in the other half to see if it actually appears more than n/k overall. If yes, then the answer to the question is yes and we can output all the surviving candidates as the k-majority elements of our original array of size n, else the answer to the question is no.

Correctness: We need to show that if indeed there is an element in A that appears more than n/k times then the algorithm returns it, else it returns that such an element doesn't exist. We observe two things, firstly *"if A has a k-majority element, then this element will also be a k-majority element in either one of the two halves of A i.e. it will appear more than $n/(2k)$ times in that half."* and secondly, *"there can be at most $k-1$ k-majority elements in any array"*.

Therefore combining the two observations, if A contains a k-majority element, then one of the recursive calls will indeed return it and the linear scan on the other half will verify that if it appears more than a k-th fraction of the size of A overall. Otherwise, the algorithm returns that there is no k-majority element in A (i.e. this is true even if the recursive calls return the k-majority elements to their corresponding halves, eventually the linear scan will confirm that none of them is a k-majority element of A).

Running time: We get the recurrence relation $T(n) \leq 2T(\frac{n}{2}) + O(nk)$ and $T(1) \leq c$, where $c > 0$ is a constant. Note that the $O(nk)$ comes from the merge part and using the second observation above, indeed in the merge step of the two halves where each half is of size $n/2$, we will have at most $2k - 2$ linear scans to do.
(NB the student who pointed out in the tutorial that we will have at most $k-1$ k-majority elements

is each half and not $2k-1$ *was ofcourse correct).*
Finally, if we apply the repeated substitution method like in exercise 1, we notice that in its i-th iteration step we will have $T(n) \leq 2^i T(\frac{n}{2^i}) + i \cdot cnk$. Taking $i = \log_2 n$, we can show that by induction $T(n) \leq cn + cnk \log n$.

Bonus question:

For convineance, we will call the 2-majority element directly the majority element. Moreover, we notice that there cannot be two distinct majority elements in the same array.

Algorithm idea: First, we start from the beginning of our input array A and pair up every two consecutive elements until the end of the array (notice that at the end of this pairing up procedure if n happens to be even, then all ellements are paired up, otherwise $A[n]$ will be left unpaired and at this point we test whether it is the majority element via a linear scan of the array; if yes, then return $A[n]$ as the majority element, else we discard it and continue in the following). Also, while doing this pairing up procedure above we simultaneously do a filtering step as follows: for each pair if they are different, then discard both, else they are the same and we can keep only one of the duplicates.
Finally, we repeat doing this filtering step until *either* only one element remains, at which point we can test via a linear scan of the array if it is indeed a majority element and return it, *or* no elements remain and we can directly return that no majority element exists in A.

Correctness: The correctness of the algorithm relies on the following claim *if A has a majority element, then it remains a majority element at the end of the algorithm in the final array* (which will only be one element) and to see why this is, notice the only two cases we have at the end of the algorithm:

- Case 1: No elements remain, thus we can say that in particular no majority element exists, and by the contraposition of the claim we know that no majority element in A should also exist, which is what the algorithm returns.
- Case 2: Only one element remains, thus we check if it is a majority element. If yes, we are done, else by the claim we know that A shouldn't have a majority element, which is what the algorithm outputs.

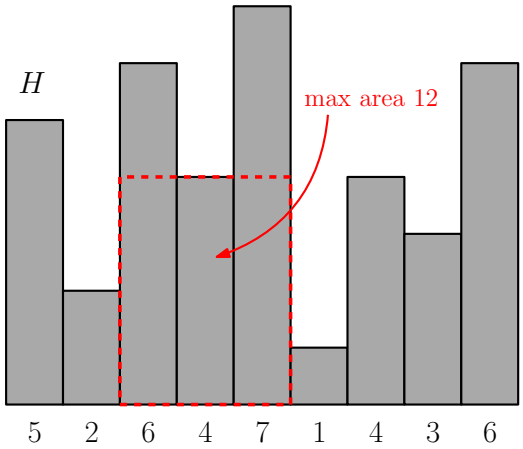
Now, notice that the claim is a direct deduction of the following observation *if an element is a majority element in the original array, it remains a majority element in the filtered array i.e. the remaining array after one filtering step.* Thus we are left to show the observation. Indeed, let X_1, X_2 define the number of appearances of the majority element x in the original array, filtered array respectively. Let Y_1, Y_2 also define the number of appearances of the remaining elements in the original array, filtered array respectively. We initially have $X_1 > Y_1$. Now, let's pay a closer look to what is happening in the filtering step: if we think of all the pairs (a, b) where $a \neq b$, then both will be discarded during the filtering process, hence, the same amount that is reduced from X_1 will also be reduced from Y_1 (if anything even more). Thus we are left with more of pairs of the form (x, x) than (a, a) where $a \neq x$. And hence when keeping one of the duplicates for each same element pair in the filtering step, we will end up with $X_2 > Y_2$ and thus X_2 is more than half the size of the filtered array (i.e. x remains a majority element in the filtered array), which proves what we want.

Running time: Notice that the size of the array is always reduced to at most half of its size after each filtering step and since the filtering can be carried out in linear time and the size of the problem at most halves after each filtering step, we get the recurrence relation $T(n) \leq T(\frac{n}{2}) + O(n)$ for $n > 2$ and $T(n) = O(1)$ for $n \leq 2$, hence we obtain an algorithm of running time $O(n)$.

Exercise 3: Maximum Rectangle in a Histogram *(10 Points)*

Consider a sequence $h_1, \ldots, h_n$ of positive, integer numbers. This sequence represents a histogram H consisting of n horizontally aligned bars each of width 1, where h_i represents the height of the i^{th} bar.

The goal is to find a rectangle of maximum area completely within H (i.e., within the union of bars).



(a) Describe an algorithm that computes a maximum area rectangle in H in time $\mathcal{O}(n^2)$. *(3 Points)*

(b) Describe an algorithm that computes a maximum area rectangle that is within H and also intersects the i^{th} bar in time $\mathcal{O}(n)$ and prove the running time. *(5 Points)*

Remark: Correct solutions in $o(n^2)$ grant partial points.

(c) Give an algorithm that uses the divide and conquer principle to compute a maximum area rectangle in H in time $\mathcal{O}(n \log n)$ and prove the running time. *(2 Points)*

Remark: You can use part (b), even if you did not succeed.

Sample Solution

(a) One idea go linearly thru the h_i's and check the max unique rectangle in H that can include the full h_i in it as the minimum (this takes n time). and then take max of all. Idea the max rectangle must have a minimum full h_i bar in it, else we can extend it to cover the full min h_i and get a better rectangle, so we loook at all possiblities...

Another idea is to compute for any pair i, j the largest rectangle that that starts and ends with the i^{th} and j^{th} bar respectively.

Algorithm 1 `max-area`$(h_1, \ldots, h_n)$

```
max-area ← 0
for i ← 1 to n do
    min-height ← h_i
    for j ← i to n do
        min-height ← min(min-height, h_j)
        max-area ← max(max-area, min-height · (j − i + 1))
return max-area
```

(b) The idea is to start at bar i and extend the rectangle in the direction with the higher bar.

Algorithm 2 `max-area-bar`$(i, h_1, \ldots, h_n)$

```
max-area ← 0
min-height ← h_i
ℓ, r ← i
h_0, h_{n+1} ← −∞                                   ▷ Sentinel elements
while ℓ > 1 or r < n do
    if h_{ℓ−1} ≥ h_{r+1} then
        ℓ ← ℓ − 1
        min-height ← min(min-height, h_ℓ)
    else
        r ← r + 1
        min-height ← min(min-height, h_r)
    max-area ← max(max-area, min-height · (r − ℓ + 1))
return max-area
```

Running time: The operations inside the while loop have constant running time. The while loop has at most n iterations since in each iteration we either increase r or decrease ℓ and both variables together can be increased or decrease at most n times, until they hit 1 or n. Therefore, the running time is $\mathcal{O}(n)$.

Correctness: (Not required though) For completeness sake we give an argument why the algorithm produces the correct result, i.e., the maximum rectangle in H that intersects bar i. This is not so clear (as in part (a)), since we are not checking every possibility, i.e., the area of every rectangle extending from some bar $\ell' \leq i$ to some bar $r' \geq i$, so we can not simply argue that we get the correct result by "brute force".

Instead, we will argue that we always consider a rectangle R that intersects i with maximal area. Assume that R spans from $\ell' \leq i$ to $r' \geq i$. Further assume (w.l.o.g.) that the index ℓ in the algorithm reaches ℓ' before index r reaches r' (the other case is symmetric). That means, we now have $\ell = \ell'$ but $r < r'$. We need to show that r moves to r', before ℓ moves on, so that we consider the area of R.

For a contradiction, assume that $h_{\ell-1} \geq h_{r+1}$ (i.e., $\ell \leftarrow \ell' - 1$ in the algorithm). But then $h_{\ell-1}$ is larger than the height of R, which means we could fit a rectangle R' into H with the same height as R but spanning from $\ell' - 1$ to r' (i.e., larger width). That means R' has strictly larger area than R, a contradiction. That means we always move the right pointer next, until eventually $r \leftarrow r'$ and the area of R will be computed and recorded in max-area.

(c) The idea is to split the Histogram H at the middle bar $m := \lfloor \frac{n}{2} \rfloor$ and take the maximum area of three partial results. The first is the maximum area of a rectangle in $h_1, \ldots, h_{m-1}$ the second is the maximum area intersecting bar m and the third is the maximum area rectangle in $h_{m+1}, \ldots, h_n$. As the partial results cover all possible positions of rectangles, the overall result is correct if the partial results are. The first and third result are obtained recursively, the second with our result from part (b).

Algorithm 3 `max-area-dc`$(h_1, \ldots, h_n)$

```
if n = 0 then
    return 0                                          ▷ check base cases
if n = 1 then
    return h_1
m ← ⌊n/2⌋
area1 ← max-area-dc(h_1, ..., h_{m-1})
area2 ← max-area-bar(m, h_1, ..., h_n)
area3 ← max-area-dc(h_{m+1}, ..., h_n)
return max(area1, area2, area3)
```

Running time: Our recursive function for the running time is $T(n) \leq 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$. Applying the Master Theorem given in the lecture shows that $T(n) \in \mathcal{O}(n \log n)$.