Department of Computer Science — FCUP

Algorithms (CC4010) — 2021/2022

Test (26.11.2021) — *duration: 2h*

1. [1.5] Is $\Theta(\log_2 n) = \Theta(\log_{100000} n)$? Explain.

The two sets are equal due to the fact that $\log_b n = \log_b a \ \log_a n$. The proof goes as follows:

- $\Theta(\log_2 n) \subseteq \Theta(\log_{100000} n)$:
  Let $f(n) \in \Theta(\log_2 n)$. Then, $\exists c_1, c_2 \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{Z}^+ \, \forall n \geq n_0 \;\; c_1 \log_2 n \leq f(n) \leq c_2 \log_2 n$.
  We want to show that $f(n) \in \Theta(\log_{100000} n)$.
  Since $\log_2 n \geq \log_{100000} n$, for all $n \in \mathbb{Z}^+$ and $\log_{100000} n = \log_{100000} 2 \ \log_2 n$, we have
  $$c_1 \log_{100000} n \leq c_1 \log_2 n \leq f(n) \leq c_2 \log_2 n = \frac{c_2}{\log_{100000} 2} \log_{100000} n$$
  This means that $\exists c_1', c_2' \in \mathbb{R}^+ \, \exists n_0' \in \mathbb{Z}^+ \, \forall n \geq n_0' \;\; c_1' \log_{100000} n \leq f(n) \leq c_2' \log_{100000} n$, because we can take $c_1' = c_1$, $c_2' = \frac{c_2}{\log_{100000} 2}$ and $n_0' = n_0$. Therefore, $f(n) \in \Theta(\log_{100000} n)$. ■
- $\Theta(\log_2 n) \supseteq \Theta(\log_{100000} n)$:
  Let $f(n) \in \Theta(\log_{100000} n)$. We want to show that $f(n) \in \Theta(\log_2 n)$.
  As above, if $\exists c_1', c_2' \in \mathbb{R}^+ \, \exists n_0' \in \mathbb{Z}^+ \, \forall n \geq n_0' \;\; c_1' \log_{100000} n \leq f(n) \leq c_2' \log_{100000} n$ then we can conclude that $\exists c_1, c_2 \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{Z}^+ \, \forall n \geq n_0 \;\; c_1 \log_2 n \leq f(n) \leq c_2 \log_2 n$. We take $c_1 = \frac{c_1'}{\log_2 100000}$, $c_2 = c_2'$ and $n_0 = n_0'$. ■

  Note that, in both cases, the constants we take depend on the ones whose existence is ensured by the hypothesis on f.

2. [2.5] Which is the difference between *quicksort* and *randomized quicksort*? Which is the worst case time complexity of *quicksort* and the expected time complexity of *randomized quicksort*? Why for *randomized quicksort* no specific input elicits the worst-case behavior?

In *quicksort*, the pivot is selected deterministically (for instance, as the first element in the sequence) whereas in *randomized quicksort* the pivot is selected at random among the elements in the sequence.

The worst case time complexity of *quicksort* is $\Theta(N^2)$, where N is the size of the sequence we have to sort.

The expected time complexity of *randomized quicksort* is $\Theta(N \log N)$.

For *randomized quicksort* no specific input elicits the worst-case behavior because each time we run the algorithm on a given instance, the pivots are selected randomly, which implies that, in the limit, if all elements are distinct, all splits $k : (N-k)$ have the same probability, $\frac{1}{N}$, for $1 \leq k \leq N$, and the expected runtime would be $\Theta(N \log N)$.

3. [2.5] Explain why the asymptotic time complexity of Insertion Sort is $O(n^2)$ and of Selection Sort is $\Theta(n^2)$. Start by presenting the main ideas of the two methods.

Insertion sort: at iteration i, for $1 \leq i < n$, we have $v[1] \leq v[2] \leq \ldots \leq v[i]$ and look for a correct position to $v[i+1]$ to get the first $i+1$ elements sorted. For that purpose, starting from position i, we shift $v[j]$, for $j \leq i$, to the right until we find the correct slot for the initial value $v[i+1]$. Iteration i takes $\Theta(i)$ time in the worst case and $\Theta(1)$ in the best case.

Selection sort: at iteration i, for $1 \leq i < n$, we have $v[1] \leq \ldots \leq v[i-1]$ and swap the minimum of $v[i], v[i+1] \ldots, v[n]$ with $v[i]$ if it is not $v[i]$. Iteration i takes $\Theta(i)$ time always.

Insertion sort runs in $\Theta(n^2)$ in the worst case, which occurs when the elements are all distinct and the sequence is in the reverse order (i.e, it is strictly decreasing). In that case, the asymptotic time complexity is of the order of the number of comparisons/shifts, which is $\sum_{i=1}^{n-1} i$. But, in the best case, when the sequence is already sorted, the time complexity of each iteration is $\Theta(1)$, and the total time is $\Theta(n)$. This means that Insertion Sort is a $O(n^2)$ algorithm.

Selection sort runs in $\Theta(n^2)$, in the worst case and in the best case, because the complexity is dominated by the time spent finding the successive minimum values, which is of the order of $\sum_{i=1}^{n-1} i$.

Note that $\sum_{i=1}^{n-1} i$ is not n^2. But, it is true that $\sum_{i=1}^{n-1} i \in \Theta(n^2)$.

4. [6.0] Given an array x of n integers, SELECT(x, a, b, k) returns the k^{th} **largest** element of $x[a], \ldots, x[b]$. Suppose that all elements of x are distinct, $x[1]$ is the first element, $1 \leq a \leq b \leq n$ and $1 \leq k \leq b - a + 1$. Assume that SELECT(x, a, b, k) is correct and MYPARTITION **uses** $x[a]$ **as pivot and runs deterministically**, in-place, and in time $\Theta(b - a + 1)$.

```
SELECT($x, a, b, k$)
1. if $a = b$ then return $x[a]$
2. $t$ = MYPARTITION($x, a, b$)
3. $p = t - a + 1$
4. if $p = k$ then return $x[t]$
5. if $p < k$ then
6.     return SELECT($x, t + 1, b, k - p$)
7. return SELECT($x, a, t - 1, k$)
```

a) Let $x[i] = s_i$, for $1 \leq i \leq n$, be the state of x in line 1. Which is the state of x in **line 4** and the value of t? Recall that SELECT(x, a, b, k) returns the k^{th} **largest** element of $x[a], \ldots, x[b]$.

t is the index of the final position of the pivot (that is of s_a);
$x[a], \ldots, x[t-1]$ contain all $s_i > s_a$, for $a \leq i \leq b$;
$x[t+1], \ldots, x[b]$ contain all $s_i < s_a$, for $a \leq i \leq b$;
(note that the problem statement says that all elements of x are distinct)
For all $i \notin [a, b]$, the value $x[i]$ remains unchanged.

b) Could the time complexity of SELECT$(x, 1, n, n)$ be $\Omega(n^2)$ in the worst case? Which is the time complexity of SELECT(x, a, b, k) in the worst case and in the best case? Explain.

SELECT$(x, 1, n, n)$ returns the minimum value of the array x. When x is in strict decreasing order, MYPARTITION(x, a, b) returns a, which means that $x[a], \ldots, x[t-1]$ will be empty and $x[t+1], \ldots, x[b]$ has all elements except the pivot, and SELECT$(x, 1, n, n)$ will recursively call SELECT$(x, 2, n, n-1)$ (line 6).

Since MYPARTITION(x, a, b) works in $\Theta(b - a + 1)$, which is $\Theta(n)$ if $a = 1$ and $b = n$, we conclude that in that case, the time complexity of SELECT$(x, 1, n, n)$ is $\Theta(n^2)$ and, therefore, it is in $\Omega(n^2)$, because $\Theta(n^2) = \Omega(n^2) \cap O(n^2)$.

In worst case, SELECT(x, a, b, k) runs in $\Theta((b - a + 1)^2)$ time if $k = b - a + 1$.

In the best case, SELECT(x, a, b, k) runs in $\Theta(b - a + 1)$. That happens when $p = k$, that is, the pivot is the kth largest element (there is no need for a recursive call). In that case, we do not have to assume any order for the remaining elements.

Note that $\Theta(b - a + 1)$ applies also to the case when $a = b$ (and $k = 1$), since it gives $\Theta(1)$, which is the complexity for line 1.

c) Under the condition stated on **a)**, justify the correctness of SELECT.

- If $a = b$, we have $k = 1$, and the function returns $x[a]$ which is the correct value.
- Assume now that $b - a + 1 > 1$. Under the condition in 4a), $x[t] = s_a$ is the pth largest value, there are exactly $p - 1$ elements greater than $x[t] = s_a$ and MYPARTITION moved them to the positions a to $t - 1$ of x and moved the elements that are smaller than s_a to the positions $t + 1$ to b.

  If $k = p$, the function returns $x[t]$ in line 4, which is correct. In line 5, $p < k$ means that we look for a value that is smaller than $x[t] = s_a$ (since its rank is greater than p). Therefore, it belongs to the segment $[t + 1, b]$ of x and its rank must be updated to $k - p$ because we have to take into account the p elements in $[a, t]$. If $p > k$ (line 7), the element we look for is greater than $x[t]$. It remains the kth largest in x, but now we know that it belongs to the segment $[a, t - 1]$ of x. This means that the arguments in the recursive calls are correct. So, if we assume that the function returns the correct value for all segments whose size is smaller than $b - a + 1$, for all k, we can conclude that it returns the correct value for segment $[a, b]$, for all k.

d) Which are the differences to *quickselect* and *median of five medians*?

The version of *quickselect* described in class returns the kth smallest element, but the approach is similar to that of finding the kth largest element. The main difference to SELECT is that the pivot is selected randomlyin *quickselect* (as for randomized quicksort) whereas in SELECT we are assuming that MYPARTITION uses a deterministic rule.

Median of medians (that was called median of 5 medians) follows a very different approach to find the pivot. First it splits the elements in groups of five elements and computes the median of each group, by some (likely brute-force) procedure. *Median of medians* is recursively applied to select the median of the n/5 medians. That median is then used as the pivot by PARTITION. The remaining steps are similar to the ones followed by *quickselect* (and, essentially, correspond to lines 3-7 above, with the necessary adjustments if we want the kth smallest element, instead of the kth largest)

Quickselect runs in $\Theta(N)$ expected time, for $N = b - a + 1$, whereas *Median of medians* runs deterministically in $\Theta(N)$ time. So, *Median of medians* runs in linear time even in the worst case, whereas SELECT runs in $\Theta(N^2)$ time in that case.

5. [2.0] Given the sequence $v_1, \ldots, v_n$ of the vertices of a n-vertex convex polygon P, in **counterclockwise order** (CCW), we want to check whether a point q belongs to the interior of P. Assume v_1 is the vertex with the smallest y-value. Explain the main steps of the $O(\log_2 n)$ algorithm for solving the problem.

The algorithm is based on *binary search*, and exploits a (rotational) decomposition of the plane determined by the sequence of $n-2$ wedges defined by $[v_1, v_k, v_{k+1}]$, for $2 \leq k < n$, which is already sorted in CCW order (by construction), and the open wedge $[v_1, v_n, v_2]$.

We can check in $O(1)$ if q belongs to this last wedge, by checking whether (v_1, v_n, q) makes a left turn (or if (v_1, v_2, p) makes a right turn). If that is true, then $q \in Ext(P)$, the exterior of P. Otherwise, we apply the following algorithm to decide whether q belongs to $Int(P)$, the interior of P.

```
1   $a \leftarrow 0$
2   $b \leftarrow n - 3$
3   while ($a \leq b$) do
4       $m \leftarrow \lfloor (a+b)/2 \rfloor$
5       if $(v_1, v_{k+m}, p)$ is a left turn then
6           $a \leftarrow m + 1$
7       else if $(v_1, v_k, p)$ is a right turn then
8           $b \leftarrow m - 1$
9       else decide whether $p$ is in the intersection of $Int(P)$ with the $m$th wedge and return True or False accordingly
10  return False
```

For any given points r, s, t in the plane, the sequence (r, s, t) makes a left-turn (in s) iff the cross-product $\vec{rs} \times \vec{rt}$ is positive (i.e., the z-coordinate of this vector is positive). If it is negative, (r, s, t) makes a right-turn and if it is null, the three points are collinear.

If p belongs to the mth wedge (line 9), which is defined by $[v_1, v_{m+2}, v_{m+3}]$, the answer in line 9 is True if $p \neq v_1$ and (v_{m+2}, v_{m+3}, p) makes a left turn. Otherwise, it is False.

6. [2.5] Explain why radix sort requires the auxiliary sorting algorithm to be a stable sorting algorithm. Is radix sort itself a stable sorting algorithm?

A sorting algorithm is called *stable* if it preserves the initial relative order of the elements that have the same key. Let us assume that the keys are positive integers. In that case, for radix sort, the keys are represented in a base B by a sequence $x_{k-1}x_{k-2}\ldots x_0$ of k digits, where x_0 denotes the least significant digit.

In its basic form, in iteration i, for $i \geq 0$, radix sort sorts the elements by considering only the ith digit. For that purpose, it uses an auxiliary algorithm (for instance, *counting sort*, that can perform each iteration in $\Theta(B+n)$ time, for an array of size n).

If the auxiliary sorting algorithm is stable, all elements that have the same digit in position i will keep their relative order in iteration i. This invariant means that they remain sorted according to the value of the suffix keys $x_i x_{i-1} \ldots x_0$. If the algorithm was not stable, there would be instances for which the work done previously would be destroyed and in the end the keys were not correctly sorted. Stability is required for the correctness of radix sort.

As a side effect, it makes radix sort stable too. Any two elements with the same key keep their relative initial positions in every iteration of radix sort.

7. [3.0] Let $P = \{p_1, p_2, p_3, \dots, p_n\}$ be a set of n points in the plane. Assume that p_1 is the point with the smallest y-value, P is sorted in **strictly decreasing order** of polar angle w.r.t. p_1 (there are no ties), and we want to report the vertices of the convex hull in **clockwise order (CW)**, starting from p_1.

Using pseudocode, write the algorithm (by adapting Graham-scan). Explain why it is correct and which is the time complexity in this case.

If we move along the boundary of a convex polygon in counter-clockwise direction we always turn left. This property ensures the correctness of the version of Graham-scan described in class. If we move in clockwise direction, we always turn right. Therefore, under the constraints stated above, using the fact that P is sorted in clockwise direction around p_1, we can compute the convex hull by adapting Graham-scan as follows, where t denotes the top of the stack and t^- the element just below it at each step.

```
1  Push $p_1, p_2$ in an empty stack $S$, with $p_2$ on the top
2  for $i$ from 3 to $n$ do
3      while $(t^-, t, p_i)$ is a left turn do
4          pop $t$
5      push $p_i$
6  return $S$
```

If the stack S is supported by an array, the sequence of vertices of the convex hull in CW order is $S[1], S[2], \dots, S[h]$, where h is its size, with $S[1] = p_1$.

The time complexity is $\Theta(n)$, because it performs n push operations and the amortized time complexity of the loop 2-5 is $\Theta(1)$ per each point p_i, as the overall number of push and pop operations in this loop is at most $2(n-2)-1$.

Master theorem:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.