FTP_Alg 2020–2021: Final Exam

Jung Kyu Canci – Draft

1. **Algorithm complexity (2P)**

   We consider the running time of a recursive algorithm $y(n)$. Suppose that $y(n)$ verifies the following:

   $$\begin{cases} y(1) = 0 \\ y(n) = y\left(\frac{n}{2}\right) + 1 \quad n \geq 1 \end{cases}$$

   The running time is

   (a) $\Theta\left(n^5\right)$
   (b) $\Theta\left(n^{\log_3 5}\right)$
   (c) $\Theta\left(\log n\right)$ ✓
   (d) $\Theta\left(n^2 \log^2 n\right)$
   (e) $\Theta\left(n^2 \log n\right)$
   (f) $\Theta\left(1\right)$

2. **Algorithm complexity (2P)**

   We consider the running time of a recursive algorithm $y(n)$. Suppose that $y(n)$ verifies the following:

   $$\begin{cases} y(1) = 0 \\ y(n) = 3y\left(\frac{n}{4}\right) + n^2 \log_2 n \quad n \geq 1 \end{cases}$$

   The running time is

   (a) $\Theta\left(n^5\right)$
   (b) $\Theta\left(n^{\log_3 5}\right)$
   (c) $\Theta\left(n \log^2 n\right)$
   (d) $\Theta\left(n^2 \log^2 n\right)$
   (e) $\Theta\left(n^2 \log n\right)$ ✓
   (f) $\Theta\left(1\right)$

3. **Algorithm complexity (2P)**

We consider the running time of a recursive algorithm $y(n)$. Suppose that $y(n)$ verifies the following:

$$\begin{cases} y(1) = 0 \\ y(n) = 5y\left(\frac{n}{3}\right) + \log_2 n \quad n \geq 1 \end{cases}$$

The running time is

(a) $\Theta\left(n^5\right)$
(b) $\Theta\left(n^{\log_3 5}\right)$ ✓
(c) $\Theta\left(n \log^2 n\right)$
(d) $\Theta\left(n^2 \log^2 n\right)$
(e) $\Theta\left(n^2 \log n\right)$
(f) $\Theta(1)$

4. **BUILD–MAX–HEAP (1+2+2 P)**

We apply BUILD–MAX–HEAP (Lecture 3 and Lecture 4) with the input vector

$$(12, 8, 1, 128, 7, 64, 30, 3, 72, 6)$$

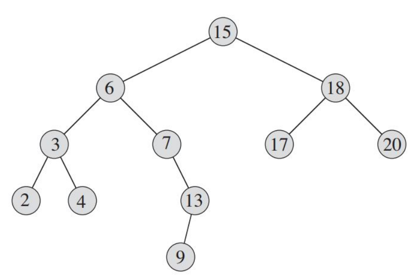
The height h of the output tree is $\boxed{3 \quad ✓}$

The number of leaves is $\boxed{5 \quad ✓}$

The left child of 72 is $\boxed{12 \quad ✓}$

5. **Binary Search Tree (1+2+2 P)**

We consider the following Binary Search Tree



We operate the following operations (in the precise listed order):

(a) Delete the 18.
(b) Insert 1.
(c) Insert 0.
(d) Delete the Maximum of the tree.

The height h of the output tree is $\boxed{5 \quad \checkmark}$

The number of leaves of the output tree is $\boxed{4 \quad \checkmark}$

The the left child of 3 in the output tree is $\boxed{2 \quad \checkmark}$

6. **Counting Sort (1+2+2 P)**

We apply COUNTING–SORT (Lecture 7) with the input the vector A

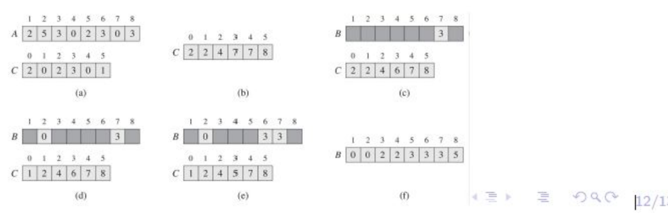
$$A = (5, 6, 5, 3, 3, 7, 4, 4, 4, 5, 3, 8, 8)$$

Let C and B be the arrays mentioned in the pseudocode of COUNTING–SORT (below is the original algorithm from our lecture slides).

```
COUNTING-SORT(A, B, k)
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 for j = A.length downto 1
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] − 1
```

The procedure starts by creating an array C that tell us the number of elements having key i for each $0 \leq i \leq k$. The step after is to change C, which now tells us the number of elements having key $\leq i$ for each $0 \leq i \leq k$.

Then the procedure creates the array B containing the sorted list by using the info in C.



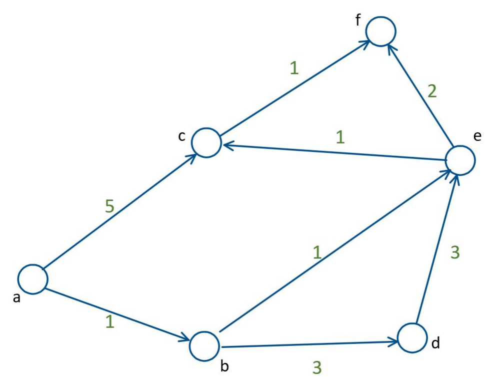
Let C be the array obtained after the for loop at line 7 and 8 of the pseudocode (so the array at the step (b) of the figure at slide 12 of Lecture 7). The entry $C[7]$ is the number $\boxed{11 \quad \checkmark}$

We consider the first cycle **for** at line 10 of the pseudocode (see (c) of the figure at slide 12 of Lecture 7) in the figure. The number $B[13]$ is $\boxed{8 \quad \checkmark}$

We consider the first cycle **for** at line 10 of the pseudocode (see (c) of the figure at slide 12 of Lecture 7). The number $C[8]$ is $\boxed{12 \quad \checkmark}$

7. **Dijkstra's algorithm (2+1+2 P)**

We apply DIJKSTRA (Lecture 13 and Lecture 14) to the graph (G, V) represented in the following picture:



Dijkstra is an iterative procedure, which update at each step the value $v.d$, that is the distance of the node v to the root a. After INITIAL-SINGLE-SOURCE(G,a) we have $a.d = 0$ and $v.d = \infty$ for each vertex $v \neq a$. We consider the situation after two iterations (line 4 to line 8) of Dijkstra with starting node a(Look out! We consider only two iterations and not the whole Dijkstra's procedure).

What is $c.d$? $\boxed{5 \quad \checkmark}$

What is $f.d$? $\boxed{100 \quad \checkmark}$

What is $d.d$? $\boxed{4 \quad \checkmark}$

In your answers, instead of ∞ write the number 100.

8. **KD–Tree (1+1+3 P)**

We apply BUILDKDTREE$(P, 0)$ (the pseudocode given in Lecture 8 and explanations in Lecture 9) to the following set P of points of the plane:

$$P = \{(1, 3), (12, 1), (4, 5), (5, 4), (10, 11), (8, 2), (2, 7)\}$$

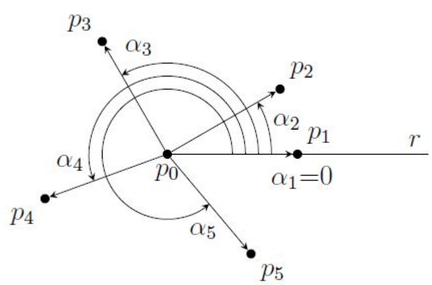
Give the height of the tree $\boxed{3 \quad \checkmark}$

How many leafs there are? $\boxed{7 \quad \checkmark}$

The second leaf (starting from left) is the point with first coordinate the number $\boxed{5 \quad \checkmark}$

9. **Essay: Polar angle (10P)**

The polar angle of a point p_i with respect of an origin p_0 is the angle from the semi horizontal straight line r (see picture below) and the vector $\vec{p_0 p_i}$. The positive direction of the angle is the counterclockwise one. Furthermore angle amplitude are taken in the interval $[0, 2\pi)$. In the picture below you find some examples of polar angle.



Write a pseudocode, which orders n points $q_1, \ldots, q_n$ according their polar angles, in increasing order. The algorithm should have $O(n \log n)$ running time.

A possible answer. Strategy: Let $p_0 = [x_0, y_0]$ and $q_i = [x_i, y_i]$ express in coordinates. We divide the set of points in $A = \{q_1, \ldots, q_n\}$ in two groups. A_1 is the subset of A containing the points with coordinates $[x, y]$ such that $x \geq x_0$ and A_2 the one such that $x < x_0$. (this costs $O(n)$). The polar angles of the points in A_1 are in the interval $[0, \pi]$ and the polar angles of the points in A_2 are in the interval $(\pi, 2\pi)$. Concretely we assume that A is a vector whose entries are the points q_i. So

$$A = [q_1, \ldots, q_n]$$

We denote by $x.A[k] = x_i$ for all index $1 \leq k \leq n$. The following Algorithm give a partition of A where the fist i elements are the points in A_1 and the remianing are elements in A_2. It returns also the first index where we have an element of A_2.

Algorithm 1

```
procedure PARTITIONANGLE(A, x_0)
    i = 0
    for j = 1 to n do
        if x.A[j] ≥ x_0 then
            i = i + 1
            exchange A[i] with A[j]
    return i + 1
```

Now we consider a procedure similar to MERGE(A, p, q, r) (see slide 7/28 of Lecture 3 and 4) where the comparison $L[i] \leq R[j]$ is not the comparison of number but if $L[i] = (x_i, y_i)$ and $R[j] = (x_j, y_j)$, then we replace the condition $L[i] \leq R[j]$ with the condition about the cross product

$$(x_i - x_0, y_i - y_0) \times (x_j - x_0, y_j - y_0) \geq 0$$

Let's denote this procedure ANGLEMERGE(A, p, q, r)

Nun we define the following algorithm (the similar recursive one like MERGE-SORT)

Algorithm 2

procedure ANGLEMERGE-SORT(A, p, r)
  if $p < r$ **then**
    $q = \lfloor (p + r)/2 \rfloor$
    ANGLEMERGE–SORT(A, p, q)
    ANGLEMERGE-SORT$(A, q + 1, r)$
    ANGLEMERGE(A, p, q, r)

Finally the last algorithm give the solution to the problem

Algorithm 3

procedure ANGLE-SORT(A)
  if $A \neq NIL$ **then**
    $n = length.A$
    $q =$PARTITIONANGLE(A, x_0)
    ANGLEMERGE–SORT$(A, 1, q - 1)$
    ANGLEMERGE–SORT(A, q, n)

Algorithm 1 costs $O(n \log n)$. Algorithm 2 costs $O(m \log m)$ with $m = r - p$. Thus Algorithm 3 costs $O(n \log n)$.

10. **Operation on data structures (4P)**

Describe the most time-efficient way to implement the operation written below. With N we denote the number of values currently stored in the underlying data structure. We assume that no duplicate values occur. At the end of your explanation write the Big-O running time (so in the worst case).

Operation: Pushing a value onto a stack implemented as an array. Assume the array is of size 2N

A possible answer. Assuming the array is big enough to hold the values we are inserting, Have bottom of stack be array[0], top value on stack is at array[top-1] O(1) - write new value in location array[top] O(1) - top++ Overall run time is $O(1)$

11. **Operation on data structures (4P)**

    Describe the most time-efficient way to implement the operation written below. With N we denote the number of values currently stored in the underlying data structure. We assume that no duplicate values occur. At the end of your explanation write the Big-O running time (so in the worst case).

    Operation: Popping a value in a stack implemented as linked list. Be specific in explaining how you get the runtime you provide

    A possible answer. Have a pointer point to top of stack. When you push: top = new node(value, top); So that pointers point towards the bottom of stack. Pop is then: (all constant time operations) temp = top; top = top.next; return temp.value; $O(1)$

12. **Operation on data structures (4P)**

    Describe the most time-efficient way to implement the operation written below. With N we denote the number of values currently stored in the underlying data structure. We assume that no duplicate values occur. At the end of your explanation write the Big-O running time (so in the worst case).

    Operation: Given a FIFO queue, find which value is the minimum value and delete it. When you finish, the rest of the values should be left in their original order.

    A possible answer. Go through all the values in the queue, maintaining their current order, keep track of the smallest value you have seen so far: dequeue(), compare to current min, enqueue() size times. This pass takes time O(N) assuming you have O(1) for enqueue and dequeue operations. At the end of this pass the values are in original order, but you know what the min value is. Then cycle through a second time, dequeueing and re-enqueueing as you go. Except when you find the min value, don't re-enqueue it. This pass also takes O(N) time. O(N) + O(N) = O(N) so overall run time is O(N).

13. **Operation on data structures (4P)**

    Describe the most time-efficient way to implement the operation written below. With N we denote the number of values currently stored in the underlying data structure. We assume that no duplicate values occur. At the end of your explanation write the Big-O running time (so in the worst case).

    Operation: Given a binary search tree, find which value is the median value, and delete that value

    A possible answer. There is not really a "worst case", regardless of the shape of the tree, you must go through N/2 values until you get to the median ("middle") value. We need to describe a general algorithm that will solve the problem in all cases. Do an inorder traversal, counting number of values visited as you go. When you have seen $N/2$ values, we will call the next value the median. So delete that value. Deleting will cause us to find the smallest successor (or largest predecessor) which could take $O(N)$ since we have no guarantee on height of the tree. Total time is $O(N/2) + O(N) = O(N)$.