Midterm Exam, Algorithms 2019-2020

- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- Your explanations should be clear enough and in sufficient detail that a fellow student can understand them. In particular, do not only give pseudo-code without explanations. A good guideline is that a description of an algorithm should be such that a fellow student can easily implement the algorithm following the description.
- You are allowed to refer to algorithms covered in class without reproving their properties.
- **Do not touch until the start of the exam.**

Good luck!

Name: ______________________ **N° Sciper:** ______________

Problem 1 / 10 points	Problem 2 / 34 points	Problem 3 / 26 points	Problem 4 / 16 points	Problem 5 / 14 points

Total / 100

1 *(10 pts)* **Basic questions.**

1a *(4 pts)* Answer whether the following statements are **true** or **false**.

The k-th largest element of an array A can be found in $O(n + k \log n)$ time by running the first k iterations of HeapSort. True or false? TRUE

$n^{1+O(1/\log n)} = O(n \log n)$. True or false? TRUE

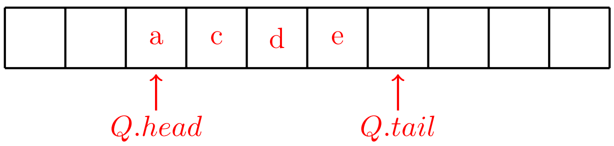
The best case runtime of InsertionSort is $\Omega(n^2)$. True or false? FALSE

If $f(n) = O(g(n))$, then $10^{f(n)} = O(10^{g(n)})$ True or false? FALSE

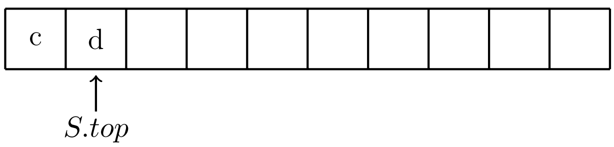
1b *(3 pts)* Consider the queue Q below (assume the implementation shown in class):

What is the resulting queue Q after the following operations: Enqueue(Q, c), Enqueue(Q, d), Dequeue(Q), Enqueue(Q, e)? Specify the content of the array used to implement the queue as well as the values of the head and tail pointers.

Solution:

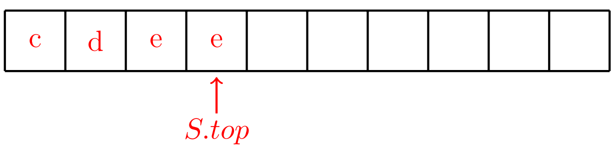


1c *(3 pts)* Consider the following stack S implemented in the same way as seen in class:



What is the resulting stack S after the following operations: Push(S, e), Push(S, f), Pop(S), Push(S, e), Push(S, g), Pop(S)?

Solution:



2 *(34 pts)* **Recurrences.** Consider the following algorithms XYZ and ABC that take as input an array A and two indices low and $high$ in the array:

```
Xyz(A, low, high)
if low ≥ high
     return
else
    for i = low to high
        print A[i]
    mid = ⌊(low + high)/2⌋
    Xyz(A, low, mid)
return

Abc(A, low, high)
Xyz(A, low, high)
if low ≥ high
     return
else
     p = low + ⌊(high − low)/4⌋
     q = ⌊(low + high)/2⌋
     Abc(A, low, p)
     Abc(A, p + 1, high)
     Abc(A, p + 1, q)
     Abc(A, low, p)
     Abc(A, q + 1, high)
return
```

2a *(10 pts)* Let $S(n)$ be the time it takes to execute ABC$(A, low, high)$ with $n = high - low + 1$, and let $T(n)$ denote the time it takes to execute XYZ$(A, low, high)$ with $n = high - low + 1$. **Give the recurrence relations** for $S(n)$ and $T(n)$. To simplify notation, you may ignore floors and ceilings in your recurrence.

Solution: For $T(n)$ we have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 10, \\ T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

and for $S(n)$ we have

$$S(n) = \begin{cases} \Theta(1) & \text{if } n \leq 10, \\ S(3n/4) + S(n/2) + 3S(n/4) + T(n) + \Theta(1) & \text{otherwise.} \end{cases}$$

2b *(24 pts)* **Prove** tight asymptotic bounds on $S(n)$ and $T(n)$.

Solution: First, we have $T(n) = \Theta(n)$, by the Master Theorem with $a = 1$, $b = 2$, and $f(n) = \Theta(n)$. This simplifies the recursion of $S(n)$ to

$$S(n) = S(3n/4) + S(n/2) + 3S(n/4) + \Theta(n).$$

To solve for $S(n)$ we will guess that $S(n) \approx n^\alpha$ for some constant $\alpha > 0$. To find α we must solve the equation

$$1 = (3/4)^\alpha + (1/2)^\alpha + 3 \cdot (1/4)^\alpha.$$

It turns out $\alpha = 2$ solves this exactly, so we will guess that $S(n) = \Theta(n^2)$ (also note that the non-recursive work done in Abc is $o(n^2)$).

Proof of lower bound: We will prove by induction that $S(n) \geq an^2$ for some constant a.

Base case ($n \leq 10$): We know that $S(n) = \Theta(1)$, so $S(1) \geq a$ for sufficiently small a.

Inductive step: We know by induction that

$$\begin{aligned} S(n) &\geq S(3n/4) + S(n/2) + 3S(n/4) \\ &\geq a(3n/4)^2 + a(n/2)^2 + 3a(n/4)^2 \\ &= an^2. \end{aligned}$$

Proof of upper bound: We will prove by induction that $S(n) \leq an^2 - bn$ for some constants a and b. We may assume that the non-recursive part of the formula, denoted by $\Theta(n)$ is at most cn for some constant c.

Base case ($n \leq 10$): We know that $S(n) = \Theta(1)$, so $S(1) \geq a - b$ for large enough difference $a - b$.

Inductive step: We know by induction that

$$\begin{aligned} S(n) &\leq S(3n/4) + S(n/2) + 3S(n/4) + cn \\ &\leq a(3n/4)^2 - b(3n/4) + a(n/2)^2 - b(n/2) + 3a(n/4)^2 - 3b(n/4) + cn \\ &= an^2 - 2bn + cn. \end{aligned}$$

This is indeed less than $an^2 - bn$ as long as $b \geq c$. Note that these too conditions, $a - b$ large enough and $b \geq c$ can be satisfied at the same time. This completes the proof.

3 *(26 pts)* **Can you find the palindrome?** In this problem your task is to find the longest palindromic subsequence of a given sequence of characters. Recall that a sequence is called a palindrome if it does not change when reversed. For example, `ABBA` and `ABDBA` are palindromes, whereas `AABB` is not.

Consider a sequence $s = (s_1, s_2, \ldots, s_n)$ of n characters. We say that a sub-sequence $s' = (s[i_1], s[i_2], \ldots, s[i_k])$, where $1 \le i_1 < i_2 < \cdots < i_k \le n$, is a palindromic subsequence if the subsequence does not change if we reverse it. Formally, s' is palindromic if $s[i_j] = s[i_{k-j+1}]$ for every $j = 1, \ldots, k$. In this problem your task is to design a dynamic programming algorithm that given a sequence s of n characters as input, finds a longest palindromic subsequence of s. For example, if $s =$ `ACBDBA`, the longest palindromic subsequence is `ABDBA`.

> **Input:** Sequence $s = (s_1, \ldots, s_n)$ of n characters.
>
> **Output:** A longest palindromic subsequence of s.

Design and analyze a dynamic programming solution for the problem. For full credit your algorithm should run in time $O(n^2)$.

3a For $1 \le i \le j \le n$ let $d(i, j)$ denote the length of the largest palindromic subsequence in the substring $s[i], s[i+1], \ldots, s[j]$. Write a recursive formula for $d(i, j)$.

Solution:

$$d(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ 2 + d(i+1, j-1) & \text{if } i < j \text{ and } s[i] = s[j] \\ \max\left(d(i, j-1), d(i+1, j)\right) & \text{if } i < j \text{ and } s[i] \neq s[j] \end{cases}$$

3b Give a bottom up implementation of your recursion from **3a** and analyze its running time.

Solution:

Initialize $d(i, j) = 1$ if $i = j$ and 0 if $i > j$. Then for all i, j such that $i < j$, fill $d(i, j)$ according to the recursion in the increasing order of $j - i$. Since we have $\Theta(n^2)$ entries to fill and each entry can be found in $\Theta(1)$ time given that we have already computed d values of smaller $j - i$ values, the total running time is $\Theta(n^2)$.

To find the palindromic subsequence, maintain another 2-d array p to keep track of the start and end indices of the longest palindromic subsequences of substrings of s: To elaborate, $p(i, j) = (a, b)$ if the longest palindromic subsequence in $s[i], s[i+1], \ldots, s[j]$ begins at $s[a]$ and ends at $s[b]$. We can update p whenever we update d, and finally use p to reconstruct the longest palindromic subsequence of s. Note that we can still update each entry in p in constant time if we update $p(i, j)$ the same time we update $d(i, j)$.

4 *(16 pts)* **Merging Binary Search Trees.** Suppose that you are given k (not necessarily balanced) binary search trees $T_1, T_2 \cdots, T_k$ each containing n/k integers. Give an $O(n \log k)$ time algorithm for merging the trees $T_1, \ldots, T_k$ into a single *balanced* binary search tree T^*: the height of T^* must be $O(\log n)$. See Fig. 1 for an example.

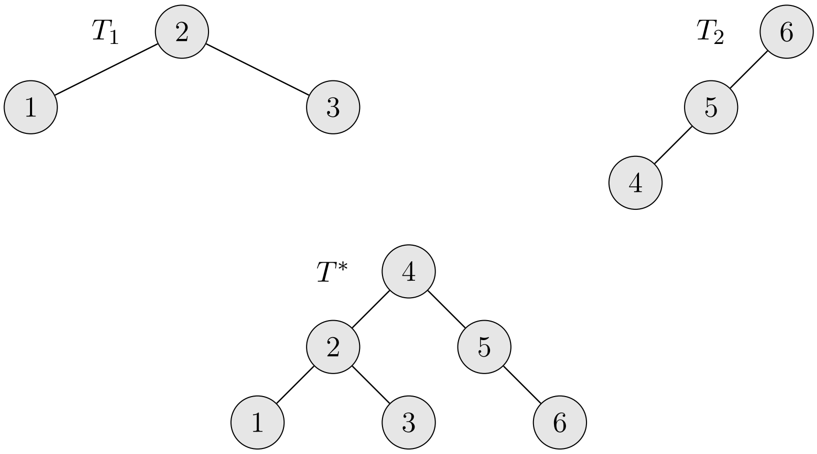


Figure 1. Input instance with $k = 2$ and $n = 6$. The tree T^* is the result of merging T_1 and T_2.

> **Input:** Binary search trees T_i, $i = 1, \ldots, k$ containing n/k integers each. The T_i's are not necessarily balanced.
>
> **Output:** A binary search tree T^* containing all n integers. The height of T^* must be bounded by $O(\log n)$.

Solution:

The algorithm is as follows:

1. Perform an inorder traversal on each T_i to get arrays A_i for $i \in [k]$. This step can be performed in $O(n/k)$ time for each tree and hence $O(n)$ time in total.
2. Merge A_i for $i \in [k]$ into a single array A^*. This step can be performed in $O(n \log k)$ time using min-heaps (see solutions of the fourth problem of exercise set 4).
3. Output Balanced-BST$(A^*, 0, n)$

Balanced-BST$(A^*, low, high)$
1. **if** $high \leq low$ return $Null$;
2. $mid = \lfloor \frac{high-low}{2} \rfloor$
3. $root = A^*[mid]$
4. $root.left =$Balanced-BST(A^*, low, mid)
5. $root.right =$Balanced-BST$(A^*, mid + 1, high)$
2. **return** $root$

The recurrence relation is given by $T(n) = 2T(n/2) + O(1)$. Using the Master theorem, $T(n) = O(n)$. It's also easy to see that the depth of the recursion tree and hence the height of the tree outputted is $O(\log n)$.

(Solution to problem 4 continued)

5 *(14 pts)* **Median of two sorted arrays.** In this problem you are given two sorted arrays with n distinct integers $a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$, and your task is to find the median in the union of the arrays. Let $c = \{a_1, a_2, \ldots, a_n, b_1, \ldots, b_n\}$ denote the union of the two arrays and let $c_1 \leq c_2 \leq \ldots \leq c_{2n-1} \leq c_{2n}$ denote the elements of c in sorted order. Recall that the median of c is $(c_n + c_{n+1})/2$. For example, suppose that $n = 4$, $a_1 = 1, a_2 = 3, a_3 = 5, a_4 = 7$ and $b_1 = 2, b_2 = 4, b_3 = 6, b_4 = 8$. Then $c_i = i$ for $i = 1, \ldots, 8$, and the median is $(c_4 + c_5)/2 = 4.5$.

> **Input:** Two sorted arrays $a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$ of n distinct elements each.
>
> **Output:** The median of the union of a and b.

For simplicity you may assume that the two arrays do not share any elements, i.e. all elements in c are distinct.

Design and analyze an algorithm for the problem. For full credit your solution should run in $O(\log^2 n)$ time.

Solution:

Solution 1: For any index $i \in [n]$ let g_i denote the number of elements in a that are smaller than b_i. More formally let

$$g_i := |\{j \in [n] : a_j < b_i\}|\,.$$

Since b is increasing, for any $1 \leq i < n$ we have $g_i \leq g_{i+1}$. Suppose that we would like to find the k'th element of the union of two arrays. Then if this element is the i'th element of b, then

$$g_i + i = k.$$

Since g_i is increasing in i, $g_i + i$ also is, and thus, given k, we can find i using binary search in time $O(\log n)$ times the time it takes to evaluate g_j for a given $j \in [n]$. We note, however, that g_j can be evaluated by a binary search over the array a in time $O(\log n)$. This gives total time $O(\log^2 n)$, for finding the k-th element of the union of the two arrays, so we can find the n-th, the $(n+1)$-th, and output their average.

Solution 2: Suppose that we want to solve a more general version of the problem. Given two sorted arrays A_1 and A_2 each of size n and m respectively. We want to find the k'th smallest element of the union of two arrays (in the recursive calls we have variables (s_1, e_1) and (s_2, e_2) specifying the range of the two arrays that we are currently looking at; the outer call starts with $s_1 = 1, e_1 = n, s_2 = 1, e_2 = m$). By using a divide and conquer approach, similar to the one used in binary search, we can find the k-th element in a more efficient way. We compare the middle elements of arrays A_1 and A_2, let us call these indices mid_1 and mid_2 respectively. Let us assume $k \leq mid_1 + mid_2$ and $A_1[mid_1] < A_2[mid_2]$, then clearly the elements after mid_2 cannot be the required element. We then set the last element of A_2 to be mid_2. If $k > mid_1 + mid_2$ and $A_1[mid_1] > A_2[mid_2]$ then clearly the elements before mid_2 cannot be the required element, so we set the first element of A_2 to be $mid_2 + 1$, and we need to find $k - (mid_2 - s_2 + 1)$'th element in the subproblem (see algorithm below). In this way, we define a new subproblem with the size of one of the arrays reduced by a factor of two. Thus, the number of recursive calls to the subproblems is at most $\log n + \log m$ and in each call we run constant number of operations. Hence, the runtime when $m = n$ is $O(\log n)$.

Algorithm 1 FIND$(A_1, s_1, e_1, A_2, s_2, e_2, k)$

if $s_1 == e_1$ **then return** $A_2[k]$
if $s_2 == e_2$ **then return** $A_1[k]$
$mid_1 = (s_1 + e_1)/2$
$mid_2 = (s_2 + e_2)/2$
if $mid_1 + mid_2 \geq k$ **then**
    if $A_1[mid_1] < A_2[mid_2]$ **then**
        return FIND$(A_1, s_1, e_1, A_2, s_2, mid_2, k)$
    else
        return FIND$(A_1, s_1, mid_1, A_2, s_2, e_2, k)$
else
    if $A_1[mid_1] < A_2[mid_2]$ **then**
        return FIND$(A_1, mid_1 + 1, e_1, A_2, s_2, e_2, k - (mid_1 - s_1 + 1))$
    else
        return FIND$(A_1, s_1, mid_1, A_2, s_2, e_2, k - (mid_2 - s_2 + 1))$