Midterm Exam, CS-250: Algorithms I, 2025

Do not turn the page before the start of the exam. This document is double-sided and has 8 pages. Do not unstaple.

- The exam consists of two parts. The first part consists of multiple-choice questions (Problem 1) and the second part consists of three open-ended questions (Problems 2, 3, 4).
- For the open-ended questions, your explanations should be clear enough and in sufficient detail that a fellow student can understand them. In particular, do not only give pseudocode 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 material covered in the lectures including algorithms and theorems (without reproving them). You are however *not* allowed to simply refer to material covered in exercises.

Good luck!

Problem 1: Multiple Choice Questions (34 points)

For each question, select the correct alternative. Note that each question has **exactly one** correct answer. Wrong answers are **not penalized** with negative points.

1a. Binary Search Trees *(7 points).* Consider a binary search tree with n nodes whose values are distinct. Which one of the following answers is correct?

A. Finding the maximum value in the tree takes $O(1)$ time.

B. Inserting any element in the tree takes $\Theta(\log n)$ time.

C. Finding an element in the tree takes $O(n)$ time.

D. Deleting an element from the tree takes $\Omega(\log n)$ time.

Solution. The answer is C. $\square$

1b. Asymptotics *(9 points).* Which of the following correctly orders the complexities from smallest to largest?

A. $O(n!), O(2^n), O(n^2), O(\sqrt{n}), O(\log n)$

B. $O(\log n), O(\sqrt{n}), O(n^2), O(2^n), O(n!)$

C. $O(\sqrt{n}), O(\log n), O(n^2), O(2^n), O(n!)$

D. $O(\log n), O(\sqrt{n}), O(n^2), O(n!), O(2^n)$

E. $O(2^n), O(n!), O(n^2), O(\log n), O(\sqrt{n})$

Solution. The answer is B. $\square$

1c. Heaps *(9 points).* Consider the array $A = \boxed{11 \mid 9 \mid 5 \mid 7 \mid 3 \mid 1 \mid 2}$ indexed from 1 to 7. Which one of the following arrays is the result of calling Heap-Extract-Max$(A, 7)$?

A. $\boxed{9 \mid 7 \mid 5 \mid 2 \mid 3 \mid 1}$

B. $\boxed{9 \mid 2 \mid 5 \mid 7 \mid 3 \mid 1}$

C. $\boxed{5 \mid 9 \mid 2 \mid 7 \mid 3 \mid 1}$

D. $\boxed{9 \mid 7 \mid 5 \mid 3 \mid 2 \mid 1}$

E. $\boxed{9 \mid 7 \mid 5 \mid 3 \mid 1 \mid 2}$

Solution. The answer is A. $\square$

1d. Data Structures *(9 points).* Let S be a stack and Q be a queue, initially empty. Consider the following sequence of operations on S and Q:

$\textsc{Push}(S, 2)$,
$\textsc{Enqueue}(Q, 2)$,
$\textsc{Push}(S, 1)$,
$\textsc{Enqueue}(Q, \textsc{Pop}(S) + 1)$,
$\textsc{Enqueue}(Q, 1)$,
$\textsc{Push}(S, \textsc{Dequeue}(Q))$,
$\textsc{Enqueue}(Q, 1)$,
$\textsc{Pop}(S)$,
$\textsc{Push}(S, \textsc{Dequeue}(Q))$.

We recall that $\textsc{Pop}$ and $\textsc{Dequeue}$ also return the item that they removed from the stack and queue respectively. Which of the following statements about S and Q holds true after having run the sequence of operations above?

A. The output of $\textsc{Dequeue}(Q)$ is the same as $\textsc{Pop}(S)$.

B. The queue Q contains both 1's and 2's.

C. The stack S contains 3 elements.

D. The output of $\textsc{Pop}(S)$ is 2.

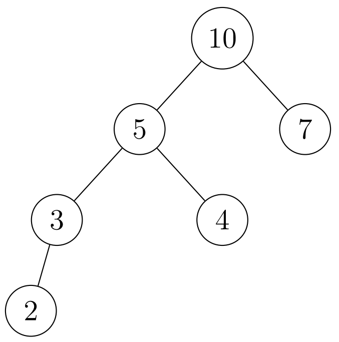
E. The output of $\textsc{Dequeue}(Q)$ is 2.

Solution. The answer is D. □

Problem 2: Magical Path (22 points)

Your task is to design and analyze an algorithm that, given a binary tree and a target number, determines whether there exists a root-to-leaf path whose sum of node values is equal to the target number. The algorithm should return **True** if such a path exists and **False** otherwise.

For example, consider the following tree. For this tree there are exactly three target numbers for which your algorithm should return **True**: 20 (path $10 \to 5 \to 3 \to 2$), 19 (path $10 \to 5 \to 4$) and 17 (path $10 \to 7$). For any other target number your algorithm should return **False**.



You may assume that every node `u` of the tree has the following attributes:

- `u.value`: the value stored in this node.
- `u.left`: the node corresponding to its left child or NULL if there is no such child.
- `u.right`: the node corresponding to its right child or NULL if there is no such child.

Your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree.

Solution. We will write an algorithm that recursively solves a similar problem in the subtrees of a node and then uses those solutions to solve the initial problem.

Specifically, let's say that we are at a node u and we want to determine whether there exists a path from u to a leaf whose sum of values equals x.

To solve this, we observe the following:

1. If u is a **leaf node** (i.e., it has no children), then the only path available is the single-node path consisting of u itself. In this case, we check whether u's value is exactly equal to x. If so, we return **True**; otherwise, we return **False**.
2. If u is **not** a leaf, the problem reduces to its left and right subtrees. We subtract u's value from x, since we have already accounted for u, and then recursively check whether either subtree has a root-to-leaf path that sums to the new target $x -$ value of u.
3. If **either** the left or right subtree has a valid path sum, we return **True**; otherwise, we return **False**.

To obtain the solution to our original problem, we simply need to call the above algorithm with the root of the tree and the given target number.

To analyze the time complexity, we define $T(n)$ as the time required to solve the problem for a tree with n nodes. Our recursive function visits each node exactly once, performing a constant amount of work per node (checking conditions, subtracting values, and making recursive calls).

Thus, the recurrence relation can be written as:

$$T(n) = T(n_L) + T(n_R) + O(1)$$

where n_L and n_R are the sizes of the left and right subtrees of the current node. By iteratively applying this recurrence until we reach the leaf nodes (where $T(1) = O(1)$), we get that $T(n) = O(n)$.

Algorithm 1 Has-Path-Sum

```
procedure HasPathSum($node, x$)
    if $node$ = null then
        return False
    end if
    $x \leftarrow x - node.value$
    if $node.left$ = NULL and $node.right$ = null then          ▷ Check if it's a leaf node
        return ($x == 0$)
    end if
    return (HasPathSum($node.left, x$) or HasPathSum($node.right, x$))
end procedure
```

□

Problem 3: Can the frog reach the end? (22 points)

A frog lives in a one-dimensional world which consists of $n \geq 2$ positions, labeled $0, \ldots, n-1$. Initially, the frog sits at position 0, and its objective is to reach position $n-1$. For every position $i \in \{0, \ldots, n-2\}$, the frog is given access to a non-empty array PosJumps(i), which contains non-negative integers and has length at most n.

The frog can make jumps according to the following rule: If the frog is currently positioned at position $i \neq n-1$, it may jump to position $i+k$, for any $k \in$ PosJumps(i) such that $i+k \leq n-1$. Then, the same rule is applied at the new position and so on.

Your task is to design an algorithm which decides, given the number of positions n, and the arrays PosJumps(i) for $i \in \{0, \ldots, n-2\}$, whether it is possible for the frog to reach position $n-1$ by any valid sequence of jumps. Your algorithm should return True if this is the case, and False if this is not the case.

Your task is to design and analyze an algorithm that solves the above problem and runs in time $O(n^2)$.

Examples. To simplify the notation for the following examples, we write the arrays PosJumps(i) compactly as an array of $n-1$ arrays. For example, if the input is given by $n = 4$ and PosJumps$(0) = [0, 1]$, PosJumps$(1) = [1]$, PosJumps$(2) = [4, 8, 10]$, we will instead simply write PosJumps $= [[0, 1], [1], [4, 8, 10]]$.

a) If $n = 6$ and PosJumps $= [[1, 2], [4], [1], [0], [0]]$, the answer is **True**. First, the frog makes a jump of length 1 from position 0 to position 1. Then, it makes a jump of length 4 to the final position 5. Notice that, even though $2 \in$ PosJumps(0), allowing the frog to jump from position 0 to position 2 at the beginning, this strategy is not optimal, since from position 2 the frog cannot reach position 5.

b) If $n = 5$ and PosJumps $= [[1], [1], [1], [1]]$, the answer is **True**, since the frog can perform 4 consecutive jumps of length 1 each. If instead PosJumps(1) were $[0]$, the answer would be **False**.

c) If $n = 5$ and PosJumps $= [[1, 2, 3], [1, 2], [1], [0]]$, the answer is **False**. This is because reaching position 3 is unavoidable, and the frog cannot make any further jump from that position.

Solution. We solve the problem by a dynamic programming approach. We design an algorithm which constructs an array dp$[0 \ldots n-1]$, such that dp$[i]$ = True if and only if position $n-1$ can be reached by the frog from position i. Assuming access to such an array, we only have to return dp$[0]$.

To construct dp, we go from right to left. First, notice that dp$[n-1]$ = True trivially. Then, for any position $0 \leq i < n-1$, we have the following identity:

$$\text{dp}[i] = \begin{cases} \text{True}, & \text{if } \exists k \in \text{PosJumps(i), such that } i < i+k \leq n-1 \text{ and dp}[i+k] = \text{True.} \\ \text{False}, & \text{else.} \end{cases}$$

Indeed, position $n-1$ can be reached from position i only if there exists a valid (non-zero) jump from index i to another position $i + k > i$, such that position $n - 1$ is reachable from position $i + k$. Hence, assuming we already computed $\mathrm{dp}[j]$ for all $i < j \leq n - 1$, we can compute $\mathrm{dp}[i]$ using $O(n)$ time. This follows because the array $\mathrm{PosJumps}(i)$ has length at most n, and so there are at most n potential jumps $k \in \mathrm{PosJumps}(i)$ which need to be checked.

Since the array dp has n entries, and each entry takes at most $O(n)$ time to compute, the total runtime complexity is given by $O(n^2)$.

□

Problem 4: Almost sorted array (22 points)

Consider an array A containing n distinct integers. In this problem, we will design an algorithm to completely sort an almost sorted array.

The array has the following structure: every element in the given array A is at most k positions away from its sorted position. In other words, if we sort the array A, an element in index i of A will end up in the range $[\max\{1, i-k\}, \ \ldots, \ \min\{n, i+k\}]$.

For example, the array $A = \boxed{6 \mid 5 \mid 3 \mid 2 \mid 8 \mid 10 \mid 9}$ is an almost sorted array with $k = 3$. If we completely sort A, we will get $\boxed{2 \mid 3 \mid 5 \mid 6 \mid 8 \mid 9 \mid 10}$.

Originally, 2 was at position $i = 4$, and after sorting, it moved to position 1. Clearly, $1 \in [\max\{0, 4-3\}, \min\{7, 4+3\}] = [1, 7]$. The number 5, initially at position $i = 2$, moved to position 3 after sorting. Again, $3 \in [\max\{0, 2-3\}, \min\{7, 2+3\}] = [0, 5]$. Similarly, if we pick any i in A, we will observe that it moves to a position in the range $[\max\{0, i-k\}, \ \min\{n, i+k\}]$.

Given the value k and a k-sorted array A containing n distinct integers, you need to design and analyze an algorithm that completely sorts A.

Your algorithm must run in $O(n \log k)$ time.

Solution. The key idea is to modify heap-sort to be more efficient. Since we know that the array is $k-$ sorted, for any index i in A, we need to find the minimum between i to $i + k$ with the help of a min-heap. Note that the numbers on the left of i are already in the correct sorted position. Since we maintain a min-heap of length k, and the INSERT operation is worst case $O(\log k)$ with n insertions, the algorithm runs in $O(n \log k)$ time. Each POP operation is $O(1)$.

```
Require: An array $A$ of size $n$ that is $k$-sorted
 1: Create a min-heap $H$                        ▷ Insert the first $k+1$ elements into the heap
 2: for $i = 1$ to $k+1$ do
 3:     INSERT $A[i]$ into $H$
 4: end for
 5: $index \leftarrow 0$          ▷ Extract the min element and insert the next element from the array
 6: for $i = k+2$ to $n$ do
 7:     $A[index] \leftarrow$ POP($H$)
 8:     INSERT $A[i]$ into $H$
 9:     $index \leftarrow index + 1$
10: end for                                      ▷ Extract remaining elements from heap
11: while H is not empty do
12:     $A[index] \leftarrow$ POP($H$)
13:     $index \leftarrow index + 1$
14: end while
```

□