**ICS 311: Algorithms** Spring 2025

# Problem Set 8

*Prof. Nodari Sitchinava* *Due: Friday, Mar 7, 2025 at 4pm*

You may discuss the problems with your classmates, however **you must write up the solutions on your own** and **list the names** of every person with whom you discussed each problem.

Start **every** problem on a separate sheet of paper, with the exception of Problems 1 (Peer credit assignment) – Problem 1 can be on the same page as any other problem. Any problem that starts on the same sheet of paper as some other problem will receive 0 points!

## 1 Peer Credit Assignment (1 point extra credit for replying)

Please list the names of the other members of your peer group for this week and the number of extra credit points you think they deserve for their participation in group work.

- You have a total of 60 points to allocate across all of your peers.
- You can distribute the points equally, give them all to one person, or do something in between.
- You need not allocate all the points available to you.
- ***You cannot allocate any points to yourself!*** Points allocated to yourself will not be recorded.

## 2 Maximum Palindrome Subsequence (50 pts)

*Before solving this problem, you might want to try solving the bonus problem first as a warm-up exercise to understand palindromes better.*

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, `noon`, and `aibohphobia` (fear of palindromes).

Given a string $s$, a *palindrome subsequence* is a subsequence of the characters in $s$ that forms a palindrome. For example, `aba` and `aacaa` are palindrome subsequences of the string `abracadabra`. The first one contains 3 charactes, the second one contains 5 characters. These are not the only palindrome subsequences of `abracadabra`. For example `abaaaba` and `araaara` are also palidrome subsequences of `abracadabra` and contain 7 characters.

In this problem you will find the **length** of the **longest** palindrome subsequence of a given string.

Design a ***recursive backtracking*** algorithm that takes a string as an argument and computes the *length* of the longest palindrome subsequence. Make your algorithm as simple as possible.

You may assume that an input string $s$ is given as an array of characters. For example, input string `noon` is given as an array $s[1..4]$, where $s[1] =$ `n`, $s[2] =$ `o`, $s[3] =$ `o`, and $s[4] =$ `n`.

Write down the pseudocode and prove that your algorithm is correct. *Hint: use induction.* ***No points will be given for an algorithm without the proof.*** If you are struggling to prove the correctness of your algorithm, chances are it is either too complicated, is incorrect, or both. *Try solving the bonus problem first to understand palindromes.*

## 3 Treasure Hunt (50 pts)

You are provided a treasure map represented as an 2-dimensional grid of size $m \times n$. Each cell in the treasure map can be:

1. Land (`.`) – passable terrain
2. Water (`#`) – impassable terrain
3. Treasure (`X`) – exactly one exists on the map

For example, a $5 \times 5$ treasure map can be represented as:

```
. . # # .
. # . . .
. . # X .
. # . # .
. . . . .
```

Design a **recursive backtracking** algorithm that determines whether a path from a given starting position to the treasure exists or not. Valid moves within the map are to adjacent land cells (up, down, left, and right – diagonal moves are not allowed). For example, using the $5 \times 5$ treasure map provided above and starting at a position of $(1, 1)$ (i.e., top-left corner), a path to the treasure exists and can be represented by marking each cell used in the path with '`v`':

```
v . # # .
v # . . .
v . # X v
v # . # v
v v v v v
```

Write down the pseudocode and prove that your algorithm is correct (*Hint: use induction on the number of land cells left to traverse in the treasure map*). **No points will be given for an algorithm without a proof of correctness.**

# 4 Almost Palindromes (OPTIONAL - 0 pts)

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, `noon`, and `aibohphobia` (fear of palindromes).

You may assume that in the problems below, an input string is given as an array of characters. For example, input string `noon` is given as an array $s[1..4]$, where $s[1] =$ `n`, $s[2] =$ `o`, $s[3] =$ `o`, and $s[4] =$ `n`.

(a) Give a ***recursive*** algorithm that tests if the input string is a palindrome. Write down the pseudocode and use induction to prove that your algorithm correctly identifies palindromes. Analyze your algorithm's running time. Make your algorithm as simple as possible.

(b) Sometimes removing a single character from the string, turns it into a palindrome. For example, if we remove `l` from the `revolver`, we will get a palindrome. We call such strings ***almost palindromes***. Write down the pseudocode for an algorithm that tests if the input string is an almost palindrome. Analyze your algorithm's running time.

# 5 Permutation in-place (OPTIONAL - 0 pts)

Design a ***recursive backtracking*** algorithm that outputs all possible permutations of an array of $n$ elements in-place (i.e., your algorithm should not use more than $O(1)$ additional space). Write down the pseudocode, and prove that your algorithm correctly prints out all permutations (*Hint: use induction*). ***No points will be given for an algorithm without the correctness proof.***

You may call the following procedures from within your algorithm:

- PRINT$(A, p, r)$ – to print the subarray $A[p..r]$
- SWAP$(A, i, j)$ – to exchange the items $A[i]$ and $A[j]$ within the array $A$