German University in Cairo
Media Engineering and Technology
Dr. Nourhan Ehab

November 5, 2023

CSEN 703: Analysis and Design of Algorithms
Winter 2023
Midterm Exam

Bar Code

Instructions: Read carefully before proceeding.

1) Duration of the exam: 2 hours (120 minutes).
2) (Non-programmable) Calculators are allowed.
3) No books or other aids are permitted for this test.
4) This exam booklet contains 10 pages, excluding this one. Two extra sheets of scratch paper and a formula sheet are attached and have to be kept attached. **Note that if one or more pages are missing, you will lose their points. Thus, you must check that your exam booklet is complete**.
5) Write your solutions in the space provided. If you need more space, write the answer on the three extra sheets and make an arrow indicating that. **Scratch sheets will not be graded unless an arrow on the problem page indicates that the solution extends to the scratch sheets**.
6) When you are told that time is up, stop working on the test.

Good Luck!

Don't write anything below ;-)

Exercise	1	2	3	4	5	$\sum$
Possible Marks	12	16	14	16	16	74
Final Marks						

Exercise 1 True/False Questions (12 Marks)

Decide whether each of the following statements is True or False. Justify your answer.

a) $f(n) = O(f(n)^2)$ for any positive function $f(n)$.

b) It is guaranteed that designing any algorithm using the divide and conquer technique will result in a worst case time complexity faster than the brute force solution.

c) Changing the divide time of the Merge Sort algorithm from $O(n)$ to $O(1)$ while keeping everything else the same improves the worst-case time complexity of the algorithm.

d) For a recurrence $T(n) = aT(\frac{n}{b}) + f(n)$ in the form of the master theorem, it is possible that $f(n) = \Omega(n^{log_b(a)+\varepsilon})$ for $\varepsilon > 0$ but the recurrence can still not be solved using the master theorem.

e) The Quick Sort algorithm does not have a combine step.

f) The average running time of Quick Sort is as bad as its worst case running time.

Exercise 2 Pancake Sorting (3+3+2+2+6=16 Marks)

Pancake sorting is an in-place sorting algorithm inspired by sorting a stack of pancakes so that the pancakes appear in sorted order with the smallest pancake at the top of the stack and the largest pancake at the bottom of the stack. The primary operation in pancake sorting is the `flip` operation which flips (reverses) a chosen prefix of the stack. In what follows, a stack of pancakes is represented as a 0-indexed array of integers `arr` where `arr[i]` is the size of pancake `i`. The top of the stack is the left-most element in the array and the bottom of the stack is the right-most element in the array. In order to sort `arr` of size `n`, pancake sorting starts from current size equal to `n` and reduces the current size by one while it is greater than 1. Let the current size be `curSize` and do the following for every `curSize`.

a) Find index of the maximum element in `arr[0..curSize-1]`. Let the index be `maxIndex`.

b) Call `flip(arr, maxIndex)` to reverse `arr` from index 0 to `maxIndex` and bring the maximum element to the beginning of the array.

c) Call `flip(arr, curSize-1)` reverses `arr` from index 0 to `curSize-1` and bring the maximum element to its correct position from the end of the array.

Below is the pseudo code implementing the above steps.

```
function pancakeSort(arr){
    n = length of arr

    for curSize = n  down to 1 {
        # Find the index of the maximum element in the first 'curSize' pancakes
        maxIndex = findMaxIndex(arr, curSize-1)

        # If the maximum element is already at the correct position, no flip needed
        if maxIndex != curSize - 1 {
            # Flip the 'maxIndex' pancakes to bring the maximum element to the top
            flip(arr, maxIndex)

            # Flip the first 'curSize' pancakes to bring the maximum element
           #to its correct position
           flip(arr, curSize-1)
        }
    }
}

# Find the index of the maximum element in the first 'size' pancakes
function findMaxIndex(arr, size){
    maxIndex = 0
    for i from 1 to size {
        if arr[i] > arr[maxIndex]
            maxIndex = i
    }
    return maxIndex
}

# Reverse the order of the first 'k' pancakes
function flip(arr, k){
    start = 0
    while start <= k{
        Swap arr[start] and arr[k]
        start++
        k--
    }}
```

a) Trace the above algorithm on `arr=[1,5,3,2]`.

b) Prove that pancake sorting is correct using a loop invariant.

c) Give an example of `arr` on which pancake sorting runs in its best case complexity.

d) Give an example of `arr` on which pancake sorting runs in its worst case complexity.

e) Give the best case and worst case time complexities of pancake sorting in appropriate asymptotic notations. Justify your answer. You do not have to analyze the algorithm using summations, a logical argument is enough.

Exercise 3 Recurrence Trees (14 Marks)

a) Get a **tight bound** on the running time of the following recurrence by using the recursion tree method. Draw the recursion tree and show all of your workout.

$$T(n) = 2T(\frac{n}{2}) + \frac{1}{n^2}$$

b) Can the recurrence in part a) be solved using the master method? If yes, solve it. If not, justify your answer.

Exercise 4 Master Theorem (4+4+4+4=16 Marks)

Can the following recurrences be solved using the master theorem? If yes, solve them. If not, explain why. Show all of your workout.

a) $T(n) = 3T(\frac{n}{2}) + n^2$

b) $T(n) = 4T(\frac{n}{2}) + n^2$

c) $T(n) = 16T(\frac{n}{4}) + nlog(n)$

d) $T(n) = 2T(\frac{n}{2}) + T(\frac{n}{4}) + n$

Exercise 5 Divide and Conquer Algorithms (6+10=16 Marks)

a) A sorted array of size `n` contains distinct integers between `1` and `n+1`, with one missing integer. Consider the below pseudo code of an algorithm to find the missing number.

```
function findMissingNumber(arr, low, high){
    if(low == high)
        return low + 1

    mid = floor((low + high) / 2)

    if(arr[mid] == mid + 1)
        # middle is in its correct position, so the missing number is in the right
        return findMissingNumber(arr, mid + 1, high)
    else
        # middle is not in its correct position, so the missing number is in the left
        return findMissingNumber(arr, low, mid)
}
```

1. Write the recurrence representing the running time of the above algorithm.

2. Solve the above recurrence to obtain the worst-case time complexity of the above algorithm.

b) You are playing a strategy game in which there are n available heroes. The heroes have different health measures which can be positive or negative. The higher the health of a hero, the better performing the hero will be in your game. Your task is to pick a number $k \leq n$ of heroes with the best overall performance (summation of healths is maximum) to promote your chances of winning the game. However, there is a twist. You are only allowed to pick k **consecutive** heroes in the list of available heroes. Given an array of the healths of `n` available heroes, write a $O(nlog(n))$ divide and conquer algorithm to return the summation of healths of the picked team of k consecutive heroes. You can describe your algorithm in concise English statements or Pseudo Code.

Useful Formulas

Summations:

- Constant series: $\sum_{i=j}^{k} a = a(k-j+1)$
- Arithmetic series: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Finite Geometric series: $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$

Logarithms:

- $ln(n) = log_e(n)$
- $log^k(n) = (log(n))^k$
- $log_c(ab) = log_c(a) + log_c(b)$
- $log_c(\frac{a}{b}) = log_c(a) - log_c(b)$
- $log_b(a^n) = nlog_b(a)$
- $a^{log_b(c)} = c^{log_b(a)}$
- Logarithmic change of base: $log_b(a) = \frac{log_c(a)}{log_c(b)}$
- $\frac{d}{dn} ln(n) = \frac{1}{n}$.
- $\frac{d}{dn} log_b(n) = \frac{1}{n\ ln(b)}$.

Scratch Paper

Scratch Paper