Assignment #6

Due: March 16, 2022 (11:59 pm)
(Total 40 marks)

CMPUT 204
Department of Computing Science
University of Alberta

Most questions here are based on Problem Set #4. Possible solutions to some other problems may also be found online. You may consult, adopt, or revise any of these solutions to compose your answers. But recall that you must understand what you submitted and be able to explain your answers. If your algorithm is the same as the one given in the Problem Set, just indicate so; otherwise please specify.

Problem 1. (8 marks)

a. Consider the algorithm of the expoentiation problem discussed in class.

(i) Show the sequence of recursive calls to compute `power(`$b, 79$`)`.
Solution: $Power(b, 79) \rightarrow Power(b, 78) \rightarrow Power(b, 39) \rightarrow Power(b, 38) \rightarrow Power(b, 19) \rightarrow Power(b, 18) \rightarrow Power(b, 9) \rightarrow Power(b, 8) \rightarrow Power(b, 4) \rightarrow Power(b, 2) \rightarrow Power(b, 1) \rightarrow Power(b, 0)$

(ii) To compute `power(`b, n`)` for an arbitrary integer $n > 0$, *at most* how many recursive calls (in terms of n) to the procedure `power(.)` are needed? Justify your answer briefly, and give an instance of $n > 50$ for which the worst-case occurs.
Solution: $T(n) \in O(logn)$; 12 calls for n=50

b. Consider Exercise 4.2-3 (CLRS p.82, which is also given in Problem Set #4): How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Explain how your algorithm works for the case where $n = 30$.

Solution: padding the operands with 0's

For $n = 30$: We pad zero's by adding on 2 extra rows and 2 extra columns. Apply Strassens Algorithm since the matrix is now a power of 2 and since matrix multiplication involves multiplying rows by columns, the extra padded rows will not have an effect on the (30×30) matrix and thus the $A \times B[(30 \times 30) \times (30 \times 30)] = A' \times B'[(32 \times 32) \times (32 \times 32)]$ and Strassens algorithm produces the correct result and then we can just decrease the matrix back down to 30×30

Problem 2. (12 marks)

a. Consider the following problem from Problem Set #4: Given a set P of n teams in some sport, a Round-Robin tournament is a collection. of games in which each team plays each other team exactly once. Design an efficient algorithm for constructing a Round-Robin tournament assuming n is a power of 2. Note that every team can play at most one game per day and the goal is to schedule all the games in $n - 1$ days.

Solution:
Problem Set 4, Problem 5.

You are asked to answer the following questions:

(i) Suppose there are 8 teams, named $\{t_1, t_2, \ldots, t_8\}$. What will be the schedule produced by your algorithm? Show a concrete schedule among the 8 teams.

Solution:
Day 1: $(t_1, t_2)(t_3, t_4)(t_5, t_6)(t_7, t_8)$
Day 2: $(t_1, t_3)(t_2, t_4)(t_5, t_7)(t_6, t_8)$
Day 3: $(t_1, t_4)(t_2, t_3)(t_5, t_8)(t_6, t_7)$
Day 4: $(t_1, t_5)(t_2, t_6)(t_3, t_7)(t_4, t_8)$
Day 5: $(t_1, t_6)(t_2, t_7)(t_3, t_8)(t_4, t_5)$
Day 6: $(t_1, t_7)(t_2, t_8)(t_3, t_5)(t_4, t_6)$
Day 7: $(t_1, t_8)(t_2, t_5)(t_3, t_6)(t_4, t_7)$

(ii) What is the running time of your algorithm in terms of the number of games played (i.e., making the arrangement for one game takes $O(1)$ time)? Give a recurrence relation and briefly explain what it solves to.

Solution:
$T(n) = \begin{cases} 2T(n/2) + \Theta(n^2) & if n > 2 \\ \Theta(1) & if n \leq 2 \end{cases}$
by case 3 of the Master Theorem (the regularity condition clearly holds for $f(n) = n^2) \rightarrow T(n) \in \Theta(n^2)$.

(iii) Let us assume n is an even number which may not be a power of 2. Can your algorithm be generalized so that every team plays at most one game per day and all the games are to be completed in $n - 1$ days? If yes, show how, and if no, argue by a counterexample.

Solution:
This is not possible to generalize our algorithm to schedule in only $n - 1$ days. For example $n = 6$
After dividing once we get $Group1 = t_1, t_2, t_3$ and $Group2 = t_4, t_5, t_6$ which based on the claim should be solved in $n - 1 = 3 - 1 = 2$ days. However, it is obvious to see that for n = 3, it takes 3 days at least to play each other and a contradiction is raised.

b. Consider the following problem from Problem Set #4: You have a manufacturing company who produces some form of glass that is more resistant than a typical glass. In your quality control section you do testing of samples and the setup for a particular type of cup produced is as follows. You have a ladder with n steps and you want to find the highest step from which you can drop a sample cup and not have it break. We call this the highest safe step. A natural approach to find that highest safe step is binary search: drop a cup from step $n/2$ and depending on whether it breaks or not try step $n/4$ or $3n/4$ (and of course if it broke you take another sample), and so on. Of course you can find the answer quickly (in $O(\log n)$ drops) but the drawback is you may break many cups.

You are asked to answer the following questions.

(i) Suppose $k = 3$. Describe a strategy that uses at most 3 cups to find the highest safe step among the n steps. If $T_3(n)$ is the function that upper bounds the number of experiments performed by your algorithm (i.e. how many times you drop a cup) for $k = 3$, it must be the case that $T_3(n) \in o(\sqrt{n})$ since for $k = 2$ we can have $T_2(n) \in O(\sqrt{n})$.

Solution:
Let $l_1 = n^{2/3} \rightarrow$ drop cups from $l_1, 2l_1, 3l_1, ...$ until the cup breaks on $(i + 1) \times l1 \rightarrow$ the safest step that the cup won't break is at $i \times l_1 \rightarrow$ maximum drop times is $n/n^{2/3} = n^{1/3}$
The range of interest is between $i \times l_1 and (i + 1) \times l_1 \rightarrow$ Set $l_2 = (n^{2/3})^{1/2} = n^{1/3} \rightarrow$ Drop cups

from $(l_1 \times i) + l_2, (l_1 \times i) + 2l_2, (l_1 \times i) + 3l_2, ...$ until the cup breaks on $(l_1 \times i) + (j+1) \times l_2 \rightarrow$ maximum drop times is $n^{1/3}$
The range of interest is between $(l_1 \times i) + (j \times l_2)$ and $(l_1 \times i) + ((j+1) \times l_2) \rightarrow$ there are l_2 steps in between which we should drop cups in each step $\rightarrow$ maximum drop times is $n^{1/3}$
Overall drop times $= 3 \times n^{1/3} \rightarrow T(n) \in \Theta(n^{1/3}) \in o(\sqrt{n})$

(ii) What if $k = 4$? Describe your solution. You can describe it on top of your solution for $k = 3$.

Solution:
$l_1 = n^{3/4} \rightarrow$ drop times: $n/n^{3/4} = n^{1/4}$
$l_2 = (n^{3/4})^{2/3} = n^{1/2} \rightarrow$ drop times: $n^{1/4}$
$l_3 = (n^{1/2})^{1/2} = n^{1/4} \rightarrow$ drop times: $n^{1/4}$
$l_4 = 1 \rightarrow$ drop times: $n^{1/4}$
Overall drop times $= 4 \times n^{1/4} \rightarrow T(n) \in \Theta(n^{1/4}) \in o(\sqrt{n})$

Note: The question here is a generalization of the "magic wand" question we studied earlier.

Problem 3. (8 marks)

Consider the Maximum Subarray Sum Problem given in Problem Set #4: You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

You are asked to answer the following questions.

a. Consider the input array $B = [2, -3, 2, -1, 3, -1, 2, -1]$, with the index to the first array element being 1 and the index to the last array element being 8. What is the output of running your algorithm?

Solution:
Largest sub-array: [2, -1, 3, -1, 2] , sum = 2 -1 + 3 -1 + 2 = 5

b. Answer the same question as above for the input array $C = [1, -3, 1, 3, -4, 5, -2, -2, 3]$.

Solution:
Largest sub-array: [5] , sum = 5

c. If the given array consists of only positive numbers, then the solution is the sum of all numbers in the array. Now consider the the maximum subarray problem where an input array has exactly **two negative numbers in consecutive positions**. Describe an algorithm for this problem. Your algorithm must be asymptotically faster than your algorithm for the original problem. You do not need to write pseudocode but the description of your algorithm should be clear. What is the running time of your algorithm? Justify your answer briefly.

Solution:
Sequentially loop through array once. Sum all positive numbers before the two negative numbers (sumB), sum the two negative numbers (sumN), sum all the positive numbers after the two negative numbers (sumA). The solution will be the max(sumB + sumN + sumA, sumB, sumA).

Since we only loop through the array once, our algorithm's running time is O(n) which is less than original problem O(nlogn).

Problem 4. (12 marks) Write an efficient algorithm (in pseudocode) that searches for a value in an $n \times m$ table (two-dimensional array). The table is sorted along the rows and columns, i.e., for table T $(1 \leq i \leq n, 1 \leq j \leq m)$

$$T[i, j] \leq T[i, j + 1]$$
$$T[i, j] \leq T[i + 1, j]$$

Analyze the running time of your algorithm in terms of n and m (i.e., your asymptotic bound should be a function of n and m). Consider both the best-case and worst-case running times. The following is an example of such a table.

1	4	7	11
2	5	8	12
3	6	9	16
10	13	14	17
18	21	23	26

Hint: Imagine a robot that starts from the top right position and walks, repeatedly if necessary, each time either to the left or to the row below depending on the result of comparing the value of the box with the target value; eventually the robot reaches the box that holds the target value, or report that the target value is not in the table.

Before you give your algorithm in pseudocode, please first describe your algorithm in plain words.

Solution:

Since we know that the last element of each row, is its largest element, and elements of each column are sorted in ascending order, we should compare K with those last elements of each row (last column from top to bottom) iteratively. For each element, if it is smaller than K, then we know that K will not be in its corresponding row, thus we skip that row. We continue the iteration until we find such element that is greater than our K. This means that K, if exists, should be in the corresponding row, because K is smaller than the last element and greater than the current element. In that case, we iterate over the elements of that row to see if K exists in the row or not.

Start search with top right element of given matrix Loop: compare this element with Key and do the following

1. If current value is equal to K return its position
2. If current element < Key then move downward and search the element in next row (if out of bound of matrix then break return false)
3. If current element > Key then move to the left (if out of bound of matrix then break return false)
4. repeat the 2nd step till you find element or returned false

Time complexity: Best case: The first value is equal to k: O(1).
Worst case: The K is the last element of the last row: O(n+m).