CSCI-570 Spring 2023

Practice Midterm 1

INSTRUCTIONS

- The duration of the exam is 140 minutes, closed book and notes.
- No space other than the pages on the exam booklet will be scanned for grading! Do not write your solutions on the back of the pages.
- If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.

1. True/False Questions

a) (T/F) Dynamic Programming approach only works on problems with non-overlapping sub problems.
False, Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub problems using recursion and storing the results of sub problems to avoid computing the same results again.

b) (T/F) The difference between dynamic programming and divide and conquer techniques is that in divide and conquer sub-problems are independent.
True, divide and conquer sub-problems are independent and DP uses solutions from the previous sub problems.

c) In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.
False, Consider the Dynamic Programming solution for Fibonacci Sequence, it just requires two variables to store the recently computed sub problem values.

d) (T/F) Suppose that there is an algorithm that merges two sorted arrays in $\sqrt{n}$ then the merge sort algorithm runs in $O(n)$
True. The time complexity of merge sort is $T(n) = 2T(\frac{n}{2}) + \sqrt{n}$. Using the master theorem $T(n) = O(n)$

e) (T/F) To determine whether two binary search trees on the same set of keys have identical tree structures, one could perform an inorder tree walk on both and compare the output lists.
False. An inorder tree walk will simply output the elements of a tree in sorted order. Thus, an inorder tree walk on any binary search tree of the same elements will produce the same output.

f) (T/F) If graph G has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a minimum spanning tree.
False. Any unique heaviest edge that is not part of a cycle must be in the MST. A graph with one edge is a counterexample.

g) (T/F) If the lightest edge in a graph is unique, then it must be part of every MST.
True. If the lightest edge is unique, then it is the lightest edge of any cut that separates its endpoints.

h) (T/F) If $f(n) = \Omega(n \log n)$ and $g(n) = O(n^2 \log n)$, then $f(n) = O(g(n))$.
False. Example, $f(n) = n^2, g(n) = n$.

i) (T/F) The shortest path in a weighted directed acyclic graph can be found in linear time.
True. Do not run Dijkstra's. See lecture 5, discussion problem3.

2. Multiple Choice Questions

a) The quicksort algorithm is a divide and conquer method as follows:
divide step: select a random element x from the array. Divide the array to two sub-arrays such that all elements of one of them are less than x and all elements of the other are greater than x.
Conquer step: attach the two sorted subarrays

What is the worst case complexity of the quicksort for an array of size n:
a) n
b) $n \log n$
c) $c^{\frac{2}{3}}$
d) n^2
d. The worst case scenario could happen when $T(n) = T(n-1) + O(n)$.

b) If a binomial heap contains these three trees in the root list: B_0, B_1, and B_3, after 2 DeletetMin operations it will have the following trees in the root list.
a) B_0 and B_3
b) B_1 and B_2
c) B_0, B_1 and B_2
d) None of the above
a. B_0 and B_3

c) Consider a complete graph G with 4 vertices. How many spanning tree does graph G have?
a) 15
b) 8
c) 16
d) 13
c. A graph can have many spanning trees. And a complete graph with n vertices has $n^{(n-2)}$ spanning trees. So, the complete graph with 4 vertices has $4^{(4-2)} = 16$ spanning trees.

d) Which of the following is false about Prim's algorithm?
a) It is a greedy algorithm.
b) It constructs MST by selecting edges in increasing order of their weights.
c) It never accepts cycles in the MST

d) It can be implemented using the Fibonacci heap.

b. Prim's algorithm can be implemented using Fibonacci heap and it never accepts cycles. And Prim's algorithm follows a greedy approach. Prim's algorithms span from one vertex to another.

e) The solution to the recurrence relation $T(n) = 8T(n/4)+O(n^{1.5}\log n)$ by the Master theorem is

a) $O(n^2)$

b) $O(n^2 \log n)$

c) $O(n^{1.5} \log n)$

d) $O(n^{1.5} \log^2 n)$

d.

3. Dynamic Programming Algorithm

USC students get a lot of free food at various events. Suppose you have a schedule of the next n days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the EVK Dining Hall for \$7 . Alternatively, you can purchase one week's groceries for \$21, which will provide dinner for each day that week. However, as you don't own a fridge, the groceries will go bad after seven days and any leftovers must be discarded. Due to your very busy schedule (midterms), these are your only three options for dinner each night.

Write a dynamic programming algorithm to determine, given the schedule of free meals, the minimum amount of money you must spend to make sure you have dinner each night. The iterative version of your algorithm must have a polynomial run-time in n.

a) Define (in plain English) sub-problems to be solved.

b) Write a recurrence relation for the sub-problems

c) Using the recurrence formula in part b, write an iterative pseudo-code to find the solution.

d) Make sure you specify

- base cases and their values
- where the final answer can be found

e) What is the complexity of your solution?

If tonight is the last night, and there's free food, then $OPT(n) = OPT(n-1)$, because we can survive optimally until last night and then get free food tonight. Otherwise, we need to either go to the EVK Dining Hall or use the last of our groceries, as there's no sense leaving groceries here afterward. Accordingly, $OPT(n)$ for nights without free food is the smaller of $OPT(n-1) + \$7$ or $OPT(n-7) + \$21$.
The base case is $OPT(0) = 0$, as it is for any $OPT(n < 0)$ in the recursive formulation. When we fill this in iteratively, we will instead let the parameter to the second case be the larger of 0 or $n-7$ to avoid out-of-bounds exceptions, especially if our language doesn't permit negative array indices (many don't).

Because each case requires only smaller parameter values as prerequisites to solving, we can fill it in increasing order:

$OPT[0] = 0, OPT[< 0] = 0$
for $i = 1 \rightarrow n$ do
if free food on night i then
$OPT[i] = OPT[i-1]$
else (visit dining hall / buy groceries)
$OPT[i] = min(OPT[i-1] + \$7\ or$
$OPT[max(0, i-7)] + \$21)$

The minimum amount to spend will be stored at $OPT[n]$.
The run-time of the algorithm is linear in n, i.e., polynomial with degree 1.

4. Divide and Conquer Algorithm

Emily has received a set of marbles as her birthday gift. She is trying to create a staircase shape with her marbles. A staircase shape contains k marbles in the k^{th} row. Given n as the number of marbles help her to figure out the number of rows of the largest staircase she can make.

For example a stair case of size 4 looks like:

*			
*	*		
*	*	*	
*	*	*	*

Table 0.1: Staircase of size 4

Let $g(n)$ be the function indicating the number of marbles in a staircase of size n. Then:

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

The goal is to find a largest number $1 \leq k \leq n$ such that:

$$g(k) \leq n < g(k+1)$$

We find such k by using the binary search algorithm. In the first step, we set $u_bound = n$ and $l_bound = 1$. At every step there are 3 cases:

- Set $k = l\_bound + \frac{u\_bound - l\_bound}{2}$
- if $g(k) = n$ then k is the solution
- if $g(k) > n$ then we set $u\_bound = k$ and continue the search
- if $g(k) < n$ then we set $l\_bound = k$ and continue the search

The time complexity of the algorithm is: $T(n) = T(\frac{n}{2}) + O(1)$ $\rightarrow$T(n) = O($\log n$)

5. Heaps

The United States Commission of Southern California Universities (USCSCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students by GPA and the commission needs an algorithm to combine all the lists. There are a few hundred colleges of interest, but each college can have thousands of students, and the USCSCU is terribly underfunded so the only computer they have on hand is an old vacuum tube computer that can do about a thousand operations per second. They are also on a deadline to produce a report so every second counts. Find the fastest algorithm for yielding the combined list and give its runtime in terms of the total number of students (m) and the number of colleges (n).

Use a minheap H of size n.
Insert the first elements of each sorted array into the heap. The objects entered into the heap will consist of the pair (GPA, college ID) with GPA as the key.
Set pointers into all n arrays to the second element of the array CP(j) = 2 for j=1 to n.
Loop over all students (i= 1 to m)
    S = ExtractMin(H)
    CombinedSort (i) = S.GPA
    j = S. collegeID
    Insert element at CP(j) from college j into the heap
    Increment CP(j)
endloop
Runtime complexity - O(mlog n)

6. Greedy Algorithm

You have a finite set of points $P = (p_1, p_2, p_3, ..., p_n)$ along a real line. Your task is to find the smallest set of intervals, each of length 2, which would contain all the given points.

Example (may or may not be an optimal solution/smallest set): Let $X = 1.5, 2.0, 2.1, 5.7, 8.8, 9.1, 10.2$. Then the three intervals [1.5, 3.5], [4, 6], and [8.7, 10.7] are length-2 intervals such that every element is contained in one of the intervals.

Device a greedy algorithm for the above problem and argue that your algorithm correctly finds the smallest set of intervals.

Algorithm: Sort P and call it S. Initialize T as an empty set. While S is not empty, select the smallest p from S, add $[p, p+2]$ into T, and remove all elements within $[p, p+2]$ from S. At last, return T as the answer.

Proof:
1) Statement: Intervals in our solution are never to the left of the corresponding intervals in the optimal solution. We can prove this by induction. Let the nth interval in our solution be t_n, the nth interval in the optimal solution be t'_n, and $p^{(n)}$ be the starting point of t_n.

Base Case: When $n = 1$, we pick $p^{(1)}$, the smallest point in S. To cover $p^{(1)}$, the rightmost starting point of the first interval is $p^{(1)}$. Thus, our first interval t_1 could not be to the left of t'_1 in the optimal solution. The base case holds.

Induction Hypothesis: Assume the statement holds for the first $n = k$ intervals.

Show $n = k + 1$ holds: Our t_{k+1} starts at $p^{(k+1)}$ and $p^{(k+1)} \notin t_k$. by induction hypothesis, we know $p^{(k+1)} \notin t'_k$ as well. To cover $p^{(k+1)}$, the rightmost starting point is $p^{(k+1)}$, so t_{k+1} can not be to the left of t'_{k+1}. Hence, the statement holds for $n = k + 1$.

2) Statement: Our solution has m intervals and the optimal solution O has fewer intervals. We will prove that this statement is not possible by contradiction. We look at the first point x which is not covered by t_{m-1}. Using the statement in (1), x will also not be covered in the t'_{m-1}. O which has at most $m-1$ intervals doesn't cover x. This contradicts to O is the optimal solution. Thus, the optimal solution can not contain fewer intervals than our solution.

With (1) and (2), we show our algorithm returns the optimal solution.

7. Amortized Analysis

An ordered stack is a singly linked list data structure that stores a sequence of items in increasing order. The head of the list always contains the smallest item in the list. The ordered stack supports the following two operations. POP() deletes and returns the head (NULL if there are no elements in the list). PUSH(x) removes all items smaller than x from the beginning of the list, adds x and then adds back all previously removed items. PUSH and POP can only access items in the list starting with the head. What would be the amortized cost of PUSH operation, if we start with an empty list? Use the aggregate method.

The worst sequence of operations is pushing in order. Assume we push 1,2,3, ..., n, starting with an empty list. The first push costs 1. The second push costs 3 (pop, push(2), push(1)). The last push costs 2n-1 (n-1 pops followed by push(n), followed by n-1 pushes). The total cost is given by:
T(n) = 1 + 3 + 5 + ... +(2n-1) = $O(n^2)$. The amortized cost is T(n)/n = O(n).