CSCI-570 Fall 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) 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.

c) (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.

d) (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.

e) (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.

f) (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, $f(n) = n^2, g(n) = n$.

g) (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.

h) (T/F) The Standard Merge sort will take $\theta(n^2)$ time in the worst case.

False. The Standard Merge Sort Algorithm will always run in $O(n \log n)$ and this time complexity is independent of the order and the input provided.

i) (T/F) The recurrence equation T(n)=$2 * T(\frac{n}{2}) + 3n$ has the solution $T(n) = \theta(\log(n^2) * n)$

True

T(n) $= aT(\frac{n}{b}) + f(n)$ and f(n) $= \theta(n)$

$log_b a = log_2 2 = 1$

Case 2 of Master theorem

Thus, T(n) $= \theta(nlgn)$

Also,

$\theta(lg(n^2) * n) = \theta(2 * lg(n) * n) = \theta(lg(n) * n)$

j) (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 $T(n) = 2T(\frac{n}{2}) + \sqrt{n}$. Case 1 of Master Theorem comes into play. Using this theorem, we get $T(n) = O(n)$

k) (T/F) The memory space required for any dynamic programming algorithm with n^2 unique subproblems is $\Omega(n^2)$

False. We have seen examples where only very few subproblems (and not all) need to be stored at any point during the DP computation of all the sub-problems. In such cases, the memory needed can be much less. For instance, longest palindrome subsequence problem can be solved with $O(n)$ space.

l) (T/F) In a 0/1 knapsack problem with n items, suppose the value of the optimal solution for all unique subproblems has been found. If one adds a new item to the list now, one must re-compute all values of the optimal solutions for all unique subproblems in order to find the value of the optimal solution for the n+1 items.

False. We can simply compute $Opt(W, n+1) = \max\{Opt(W, n), \text{value}[n+1] + Opt(W - W_{n+1}, n)\}$ with subproblems already computed.

m) (T/F) In dynamic programming you must calculate the optimal value of a sub-problem twice, once during the bottom up pass and once during the top down pass.

False.

2. Multiple Choice Questions

a) Consider the recurrence relation for two algorithms given as:

1) $T_1(n) = 3 * T_1(\frac{n}{2}) + n^2$

2) $T_2(n) = 2 * T_2(\frac{n}{2}) + n * logn$

Select the option which represents the correct Asymptotic time complexity for $T_1(n)$ and $T_2(n)$ respectively.

a) $\theta(n^2)$, $\theta(n * log^2 n)$

b) $\theta(n)$, $\theta(n * log^2 n)$

c) $\theta(n^2)$, $\theta(n^2 * logn)$

d) None of the above

Option-a:

For $T_1(n)$:

It falls under the case 3 of master theorem where f(n) $= n^2$ and $log_b a = log_2 3$. Thus, we get the time complexity of $T_1(n) = \theta(n^2)$

For $T_2(n)$:

It falls under the case 2 of master theorem where f(n) $= n^2$ and $log_b a = log_2 2$.

Thus, we get the time complexity of $T_1(n) = \theta(n * log^2 n)$

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.

f) Which of the following statement about dynamic programming is correct?

a) Any problem that can be solved using dynamic programming has a polynomial time worst case time complexity with respect to its input size.

b) Dynamic Programming approach only works on problems with non-overlapping subproblems.
c) 0/1 knapsack problem can be solved using dynamic programming in polynomial time.
d) In the sequence alignment problem, the optimal solution can be found in linear time by incorporating the divide-and-conquer technique with dynamic programming.

None of above is correct.

g) Which description about dynamic programming is correct?
a) Dynamic programming is exclusively used for sorting algorithms.
b) Dynamic programming can only be applied to problems with linear complexity.
c) Dynamic programming relies on solving problems by dividing them into smaller, overlapping subproblems.
d) Dynamic programming always guarantees the fastest possible runtime for any algorithm.

c

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 $i = 1 \to n$ 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

You are given a sorted array of n positive integers such that $A[1] < A[2] < \ldots < A[n]$. Your job is to determine if there exists an array index k, such that $k = A[k]$.

Give a detailed divide and conquer algorithm for the Boolean function that returns FALSE if no such k can be found, and returns TRUE if such an index exists. The complexity of this function must be $O(\log n)$. Prove that your algorithm will also run in $O(\log n)$.

```
indexExists(A, size)
    low = 1
    high = size
    while low ≤ high:
        mid = low + ( high-low )/2
        if (A[mid] == mid ):
            return TRUE
        if (A[mid] < mid)
            low = mid+1
        if (A[mid] > mid)
            high = mid-1
    return FALSE
```

Proof: The algorithm is based on Divide and Conquer approach where we are dividing the array into two halves on every iteration

If the middle element "mid" satisfy A[mid] < mid, we can see that for ∀ k < mid, A[k] < k. This is because A is a sorted array of DISTINCT integers. To see this we note that A[j + 1] A[j] ≥ 1 ∀ j.

Thus, in the next round of search we need to focus on A[mid + 1 : high].

Similarly, if A[mid] > mid, we only need to search A[low : mid-1] in the next round.

The recurrence relation for our algorithm can be written as:

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

This follows, case 2 of the Master Theorem.

T(n) = O(log n)

5. Heaps

Design a data structure that has the following properties:

- Find median takes $O(1)$ time.
- Extract-Median takes $O(\log n)$ time.
- Insert takes $O(\log n)$ time.
- Delete takes $O(\log n)$ time.

where n is the number of elements in your data structure. Describe how your data structure will work and provide algorithms for all aforementioned operations. You are not required to prove the correctness of your algorithm.

We use the $\lceil n/2 \rceil$ smallest elements to build a max-heap and use the remaining $\lfloor n/2 \rfloor$ elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time $O(1)$ (we assume the case of odd n, where the median is the $(n/2)$-th element when elements are sorted in increasing order).

Insert() Algorithm: For a new element x,

a) Compare x to the current median (root of the max-heap).

b) If x ¡ median, we insert x into the max-heap. Otherwise, we insert x into the min-heap. This takes $O(\log n)$ time in the worst case.

c) If size(maxHeap) $>$ size(minHeap) + 1, then we call Extract-Max() on the max-heap and insert the extracted value into the min-heap. This takes $O(\log n)$ time in the worst case.

d) Also, if size(minHeap) $>$ size(maxHeap), we call Extract-Min() on the min-heap and insert the extracted value into the max-heap. This takes $O(\log n)$ time in the worst case.

Extract-Median() Algorithm: Run ExtractMax() on the max-heap. If after extraction, size(maxHeap) $<$ size(minHeap), then execute Extract-Min() on the min-heap and insert the extracted value into the max-heap. Again, the worst-case time is $O(\log n)$.

Delete(): We will store the memory locations of all the elements while constructing the heaps and inserting new elements in a hash table/map. Since we will have direct access to the location of the elements, we can go to the element to be deleted and delete it in $O(\log n)$ time. We will have similar conditions as mentioned in the insert operation to maintain the size of each heap.

6. Greedy Algorithm

You have a finite set of points $P = \{p_1, p_2, p_3, \ldots, p_n\}$ along a real line. Your task is to find the smallest number of intervals, each of length 2, which would contain all the given points. For example, when $P = \{5.7,\ 8.8,\ 9.1,\ 1.5,\ 2.0,\ 2.1,\ 10.2\}$, an optimal solution has 3 intervals $[1.5, 3.5]$, $[4, 6]$, and $[8.5, 10.5]$ are length-2 intervals such that every element is contained in one of the intervals. Device a greedy algorithm for the above problem and analyze its time complexity. You don't need to prove the correctness of your algorithm.

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.

The overall runtime is $O(n \log n)$ because sorting takes $O(n \log n)$ and the while loop takes $O(n)$.

Proof of correctness (not required):
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

In a data structure, there is a doubly linked list with 'head' and 'tail' pointers for storing n elements. When searching for a specific element, the search starts from the head and proceeds until the target element is found, at an index i where $i \leq n$. The cost for each 'search' operation is i, and when the target element is found, it's moved towards the head of the list by 1 index at a cost of c (a positive constant). We want to calculate the amortized time complexity for searching and moving an element in the worst-case scenario. Assume that the number of times the target element is searched is less than or equal to the total number of elements in the list, and consider a scenario where we search for this specific element repeatedly.

Taking the worst-case scenario, we will consider that the element is placed at the end of the doubly linked list (i.e., at index n). Let's say that the element is searched n times. When the element is searched for the first time, it's search cost will be n. Next time it will be (n-1) and so on.
Additionally, cost to move the target element would be c every time (except when the target element is at position 1). So, for n-1 moves, it will be (n-1)*c
Approximately after n searches, the element would be placed at index 1.
T(n) = n + (n-1) + (n-2) + . . . 1 + (n-1)*c = (n*(n+1))/2 + (n-1)*c

Therefore, T(n)/n = O(n)