1.5em

CSCI-570 Fall 2023

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!
- If you require an additional page for a question, you can use the extra page provided at the end of this booklet. However please indicate clearly that you are continuing the solution on the additional page.

1. **True/False Questions (20 points)**

Mark the following statements as **T** or **F**. No need to provide any justification.

a) **(T/F)**. Prim's algorithm can be applied to directed weighted graphs.

False.

b) **(T/F)**. Let $T(n) = 3T(\frac{n}{3}) + O(\log n)$ be a recurrence equation. Then we can conclude that $T(n) = \Theta(n \log n)$ by the Master theorem.

False. Case 1 of the master theorem: $a = 3, b = 3$

c) **(T/F)** In a dynamic programming formulation, the sub-problems must be mutually independent.

False

d) **(T/F)** The main difference between divide and conquer and dynamic programming is that divide and conquer solves problems in a top-down manner whereas dynamic-programming does this bottom-up.

False

e) **(T/F)** There are no known polynomial-time algorithms to solve the 0-1 knapsack problem.

True.

f) **(T/F)** If the edge is not part of any MST of G, then it must be the maximum weight edge on some cycle in G.

True.

g) **(T/F)** A greedy algorithm can be used to solve any problems that are solved by the dynamic programming.

False.

h) **(T/F)** If $f = O(n)$ and $g = O(n)$, then $f(g(n)) = \Theta(n^2)$.

False. Counterexample: f= g = const.

i) **(T/F)** There is a path from any vertex to any other vertex in a connected directed graph.

False. That is true for undirected graphs.

j) **(T/F)** Insertion into a binomial heap of size n has an amortized cost of $\Theta(\log n)$

False. That is $\Theta(1)$.

2. Multiple Choice Questions (10 points)

Please select the most appropriate choice. Each multiple choice question has a single correct answer.

a) In the lecture we discussed the runtime complexity of Dijkstra's algorithm when it's implemented using a binary heap. What would be the algorithm runtime complexity if we replace a binary heap by a Fibonacci heap? Select the tightest upper bound.

a) $O(E + V \log V)$
b) $O((E + V) \log V))$
c) $O(E + V)$
d) $O(V + E \log V)$

a.

b) The solution to the recurrence relation $T(n) = n\,T(n/2) + O(\frac{n}{\log n})$ by the Master theorem is:

a) $\Theta(n)$
b) $\Theta(n \log n)$
c) $\Theta(\frac{n}{\log n})$
d) N/A

d.

c) Which of the following statement about dynamic programming is correct:

a) Any dynamic programming algorithm with n subproblems will run in $O(n)$ time.
b) Complexity of a dynamic programming algorithm is equal to the number of unique sub-problems in the solution space.
c) When finding the value of the optimal solution in a dynamic programming algorithm, one must find values of optimal solutions for all of its sub-problems.
d) The time complexity of a dynamic programming solution is always lower than that of an exhaustive search for the same problem.

c

d) Analyze the runtime complexity of printing in the following code snippet.

```
while(n > 0){
   for(int i = 0; i<n; i++){
       System.out.println("#");
   }
   n = n/2;
}
```

a) $\Theta(\log n)$
b) $\Theta(n)$
c) $\Theta(n^2)$
d) $\Theta(n \log n)$
b.
Starting from n, the inner for loop runs n times. Second time, it runs $n/2$ times, and then $n/4$ times and so on. $n + n/2 + n/4 + ... + 2 + 1 = 2n - 1 = O(n)$

e) Which of the following is not a characteristic of a binomial heap?

a) Each tree in the heap has a unique rank.
b) The root of each tree in the heap has the same value.
c) The number of elements in each tree is a power of 2.
d) The amortized cost of insertion is constant.
b

3. Amortized Cost Analysis (10 points)

Recall the example of an unbounded (dynamic) array from lecture with a "doubling-up resizing policy," where we showed that the amortized cost per insert operation is a constant. Now we also want to implement a resizing policy for the delete operation. Consider the following options:

a) When the array is a half-full after delete, we create a new array of half the size and copy all the elements. Identify a sequence of operations where this approach would not result in a constant amortized cost per insert or delete operation. (5 pts)

b) When the array is a quarter-full after delete, we create a new array of half the size and copy all the elements. Show using the aggregate method that this approach would give a constant amortized cost for insert and delete operations. (5 pts)

a) Perform a series of operations on an array with n elements (the array is already full to begin with): Append, Delete, Append, Delete, Append, and so on upto n times. Let's say we perform n operations, and at every operation, be it append or delete, we'll have to copy n or n+1 elements to a new array. So,

$$T(n) = \left(n \cdot \frac{n}{2}\right) + \left((n+1) \cdot \frac{n}{2}\right)$$

$$\frac{T(n)}{n} = \mathcal{O}(n)$$

b) For the scenario mentioned in 3a, in this case we'll have to copy only n elements when we perform first append, and rest all the other n-1 operations are constant time operations., So,

$$T(n) = 1 \cdot n + (n-1) \cdot 1$$

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

Rubrics 3a (5 points):

- (upto 3 pt) Correct Description of the working scenario
- (upto 2 pts) The description of the scenario is self-explanatory to establish that the cost would not be constant OR Correct calculation of T(n) and showing that T(n)/n = O(n) OR Correct explanation "in words" that the cost is not constant OR Correct mathematical argument proving that cost would not be constant.
- (-2 pts) If the description is very vague, but close to correct solution.

Rubrics 3b (5 points):

- (upto 3 pts) Correct calculation of T(n). Simply stating T(n) = O(n) without any concrete reasoning or mathematical expression gets 0 points.
- (upto 2 pts) Showing that T(n)/n = O(1). Simply stating that T(n)/n = O(n)/n = O(1) gets 0 points. The reasoning should be explained with sequence of operations, and correct T(n) expression.
- (-2 pts) If using scenario (sequence of operation) other than 3a and description is vague. Note: If description is clear than no points deduction.

4. **Short Problem (10 points)**

Given a $n \times n$ matrix where each of the rows and columns are sorted in ascending order. Design an algorithm to find the k-th smallest element in the matrix using a binary heap. Discuss its runtime complexity. You may assume that $k < n$.

Build a min heap containing the first element of each row using $O(n)$ time and then run deleteMin. If the element from the i-th row was deleted, then insert another remaining element in the same row to the min heap. Repeat the procedure k times. The total running time is $O(n + k \log n)$.

As we have $k < n$, actually we only need the first $k \times k$ part of the matrix. Thus the time complexity can be further improved to $O(k + k \log k) = O(k \log k)$. Both answers are considered as correct.

(4 pts) Build the min-heap (not a max-heap). Run deleteMin or remove the min element each time.

(2 pts) Min-heap include the first element of each row.

(1 pts) Insert the remaining element from the same row and repeat for k times.

(3 pts) $O(n + k \log n)$ or $O(k \log k)$ is the final time complexity.

5. **Greedy Algorithm (15 points)**

Given a binary array A of n elements that contain at least one 1 and one 0, we are interested in the maximum distance between a single one and a single zero or vise versa. Specifically, we try to find

$$\max_{0 \leq i,j < n, A[i]=0, A[j]=1} |i - j|$$

For example, $[0, 1]$ has the maximum distance 1 and $[0, 1, 1, 0, 0]$ has the maximum distance 3.

a) (5 points) Let i, j be the pair of indices that attain the maximum distance ($A[i] = 0, A[j] = 1$). Prove by contradiction that at least one of the following holds:

- $i = 0$
- $i = n - 1$
- $j = 0$
- $j = n - 1$

b) (10 points) Design a greedy algorithm that finds the maximum distance in $O(n)$ time complexity.

a) Suppose none of the equalities holds. We then have $0 < i, j < n-1$. We consider three cases:

- $A[0] = A[n-1] = 0$
  - $i < j$: $j - 0 > j - i$, contradiction
  - $j < i$: $n - 1 - j > i - j$, contradiction
- $A[0] = A[n-1] = 1$
  - $i < j$: $n - 1 - i > j - i$, contradiction
  - $j < i$: $i - 0 > i - j$, contradiction
- $A[0] \neq A[n-1]$: $n - 1 - 0 > |i - j|$, contradiction

It contradicts in all the cases, so at least one of the equalities holds.

b) **Solution 1:** Find max j so that $A[j] \neq A[0]$ and min i so that $A[i] \neq A[n-1]$. Then the maximum distance is $\max(j, n-1-i)$. Clearly the algorithm runs in $O(n)$. **Solution 2:** Keep track of the positions of the first and last 0's and 1's, compute the maximum distance between 0's and 1's according to these positions.

Rubrics

a) Contradiction assumption (1 pt); 1 pt for each case; 1pt for the conclusion.

b) **Solution 1:** 10 pts for the algorithms (3 pts for $\max j$; 3 pts for $\min i$; 4 pts for the final output); give at most 4 pts for the algorithm not running in $O(n)$. 3 pts for calculating one direction (for example, $A[0]$ and $A[j]$).**Solution 2:** 10 pts for the algorithms. 3 pts for less than four positions (for example, only consider first 1 and last 0).

6. **Divide and Conquer (15 points)**

Suppose that you are given a square matrix M that has n rows and n columns. Each row and each column contains no duplicates and is already sorted in increasing order. More formally: for each row r of the matrix, $M[r][i] < M[r][j]$ whenever $i < j$, and, for each column c of the matrix, $M[i][c] < M[j][c]$ whenever $i < j$. Design a divide-and-conquer algorithm that finds a given value k in the matrix M. You may assume that n is a power of 2.

a) Describe your algorithm in plain English or using pseudocode. (7 points)

Plain English:

We assume that n is a power of 2, and that the rows and columns of M are indexed $0, ..., n-1$. All division operations are integer division, i.e. $x/2 = x/2$.

The general idea is to partition M into four pieces:

- M_{11} is the submatrix consisting of rows $0, ..., \frac{n}{2}-1$ and columns $0, ..., \frac{n}{2}-1$.
- M_{12} is the submatrix consisting of rows $0, ..., \frac{n}{2}-1$ and columns $\frac{n}{2}, ..., n-1$.
- M_{21} is the submatrix consisting of rows $\frac{n}{2}, ..., n-1$ and columns $0, ..., \frac{n}{2}-1$.
- M_{22} is the submatrix consisting of rows $\frac{n}{2}, ..., n-1$ and columns $\frac{n}{2}, ..., n-1$.

We notice that if k is less than $M[n/2][n/2]$, then k cannot appear in M_{22}, since all entries in M_{22} are at least as large as $M[n/2][n/2]$ (based on the given properties of M). So in this case, we search in M_{11}, M_{12}, M_{21} recursively. Similarly, if k is larger than $M[n/2][n/2]$, then k cannot appear in M_{11}, since all entries in M_{11} are at most as large as $M[n/2][n/2]$ (based on the given properties of M). So in this case, we search in M_{12}, M_{21}, M_{22} recursively.

Our recursive algorithm will be parameterized like binary search. But instead of having a `low` and a `high` to indicate the boundaries of where we're looking in an array, we now have two dimensions. So we'll use a `lowRow` and `highRow` to describe the range of rows we're looking at, and we'll use a `lowCol` and `highCol` to describe the range of columns we're looking at.
Specifically, we compare the search value k with the middle (i.e., indexed by the 4 parameters).

- If it's equal, return the index of the middle.
- If it's less, search in M_{11}, M_{12}, M_{21}.
- If it's greater, search in M_{12}, M_{21}, M_{22}.
- If the entire matrix has been searched (i.e., indicated by the 4 parameters), terminate.

Note:

- You need to include more parameters (e.g., (`lowRow, highRow, lowCol, highCol`) or (`lowRow, lowCol, lengthOfSubmatrix`), *etc.*) to keep track of the indices. Otherwise, you'll return the row and column numbers of the current submatrix, not those of the original matrix.
- Another benefit of using parameters is that you don't need to "copy" the matrix into 4 submatrices recursively, which would incur additional running time of $O(n)$.

Pseudocode:

Algorithm 1 `searchAlgorithm`(M, k)

```
n ← M.length
return searchRecurse(M, k, 0, 0, n − 1, n − 1)
```

Algorithm 2 `searchRecurse`$(M, k,$`lowRow,highRow,lowCol,highCol`$)$

```
halfSize ← (highRow - lowRow + 1)/2
if k == M[lowRow+halfSize][lowCol+halfSize] then
    return "(lowRow+halfSize, lowCol+halfSize)"
else if halfSize == 0 then
    return "Not Found"
else if k < M[lowRow+halfSize][lowCol+halfSize] then      ▷ search M_11
    answer = searchRecurse(M, k, lowRow,lowRow+halfSize-1,
              lowCol,lowCol+halfSize-1)
else                                                       ▷ search M_22
    answer = searchRecurse(M, k, lowRow+halfSize,highRow,
              lowCol+halfSize,highCol)
end if
if answer = "Not Found" then               ▷ search M_12 if k not found yet
    answer = searchRecurse(M, k, lowRow,lowRow+halfSize-1,
              lowCol+halfSize,highCol)
end if
if answer = "Not Found" then               ▷ search M_21 if k not found yet
    answer = searchRecurse(M, k, lowRow+halfSize,highRow,
              lowCol,lowCol+halfSize-1)
end if
return answer
```

Rubrics (7 points):
Plain English:

- (2 pts) Specify how the 4 (square) submatrices are defined.
- (2 pts) Clearly state / explain the 3 submatrices to search in 2 cases.
- (2 pts) Clearly state how to return the indices (e.g., by the 4 parameters like binary search).
- (1 pt) Mention the 2 base cases (i.e., equal value or the entire matrix has been searched).
- Deduct 1 mark for partially correct.
- Searching 4 submatrices is incorrect.

Pseudocode:
Similarly,

- (2 pts) Specify the 4 submatices.
- (2 pts) State which submatrices to search in each scenario.
- (2 pts) Include more parameters / arguments to index.
- (1 pt) Specify the 2 base cases.
- You can only write the (private) second algorithm with additional parameters.
- Searching 4 submatrices is incorrect.

b) Define a recurrence relation for the runtime complexity. (5 points)

Recurrence Relation

We give a recurrence relation $T(n)$ that describes the worst-case running time of `searchRecurse` in terms of n, which represents the number of rows (or columns) in the submatrix being searched, i.e., `highRow` $-$`lowRow`$+1$.

$$T(n) = \begin{cases} C_1 & \text{if } n = 1 \\ 3T(\frac{n}{2}) + C_2 & \text{if } n > 1 \end{cases}$$

where C_1 and C_2 are two constats.

A quick explanation of the recurrence relation:

In the base case, the worst-case number of steps consists of executing the lines $1, 2, 4, 5$ (i.e., $C_1 = 4$ by the pseudocode).

In the recursive case, the worst-case number of steps consists of the executing the lines $1, 2, 4, 6, 9, 11, 12, 14, 15, 17$ (i.e., $C_2 = 10$ by the pseudocode, which is constant), plus the 3 recursive calls to the function on a matrix with dimensions $(n/2)$-by-$(n/2)$.

Rubrics (5 points):

- (3 pts) Define the recurrence relation correctly, i.e., $a = 3, b = 2$.
- (2 pts) Define $f(n)$ such that $f(n) \in O(1)$ corrrectly. A reasonable answer based on the student's own algorithm is also acceptable.
- Answers with $a = 4$ will be considered incorrect.

c) Solve the above equation using the Master Theorem. (3 points)

Master Theorem

We determine the asymptotic behaviour of $T(n)$ using the Master Theorem. The relevant values are $a = 3, b = 2, f(n) = 10$.

The number of leaves of the recurrence tree is calculated as: $n^{\log_b a} = n^{\log_2 3}$.

Note that $f(n) = C_2 \in O(1)$, so $f(n) \in O(n^0)$.

Since $\log_2 3 \approx 1.585... > 0$, we can write $n^0 = n^{\log_2 3 - \epsilon}$ with an $\epsilon > 0$. This proves that $f(n) \in O(n^{\log_2 3 - \epsilon})$.

This satisfies the condition of Case 1 of the Master Theorem, which tells us that $T(n) \in \Theta(n^{\log_2 3})$.

Rubrics (3 points):

- (1 pt) Correctly define $a, b, c, f(n)$.
- (2 pts) Specify that it falls under Case 1 and obtain the correct answer.
- Answers indicating $O(n^2)$ will be considered incorrect.

7. **Dynamic Programming (20 points)**

Consider a row of n lamp posts standing sequentially. The heights of these lamp posts are $H = [h_0, h_1, \ldots, h_{n-1}]$. Your task is to find a subsequence of lamp posts that contains the maximum number of lamp posts with strictly increasing heights.

Example.
Suppose we have lamp posts with heights $H = [10, 9, 2, 5, 3, 7, 101, 18]$. The maximum subsequence length is 4, and a subsequence is $[2, 3, 7, 101]$.

a) Define (in plain English) sub-problems to be solved. (5 pts)

There are two perspectives to answer this question with full credits, either should be good:

length perspective: $dp[i]$, the **length** of subsequence with maximum amounts of lamp posts in increasing height, and the end element is the i-th lamp post.

or

subsequence perspective: $s[i]$, the **subsequence** with maximum amounts of lamp posts in increasing height, and the end element is the i-th lamp post.

Rubrics (5 points):

- Incorrect or irrelevant definition: -5 points
- Improper conversion from LIS into LCS

b) Write a recurrence relation for the sub-problems. (8 pts)

$dp[i] = \max(dp[j] + 1, dp[i])$ for $0 <= j < i$ and $h_i > h_j$

or

$s[i] = \max_{\text{max length}}(s[j] + [i], s[i])$ for $0 <= j < i$ and $h_i > h_j$

Rubrics (8 points):

- incorrect/irrelevant recurrence: -8 points
- recurrence relation for i takes the value of i-1 as start value before finding the correct value as default case: -4 points
- if the recurrence relation is correct. follow the below condition
- If correct recurrence but missing $0 <= j < i$: -2 points
- If correct recurrence but missing $h_i > h_j$: -2 points
- If correct recurrence but missing max operation between lengths: -2 points
- Minor error: -1 point
- LCS recurrence relation is eligible for full credits.

c) Using the recurrence formula in part b, write an iterative pseudo-code to find the solution. (5 pts)

Make sure you specify

- base cases and their values (1 pt)
  $dp[i] = 1$ for all $0 \leq i \leq n-1$
  or
  $s[i] = [i]$ for all $0 \leq i \leq n-1$
- where the final answer can be found (2 pts)
  $\max(dp[i])$ for $0 \leq i \leq n-1$ and the corresponding $s[i]$ as subsequence
  or
  $\max_{\text{max length}}(s[i])$ for $0 \leq i \leq n-1$ and the corresponding $\text{len}(s[i])$ as max length.

pseudocodes (2 pts):

for $0 \leq i \leq n-1$:
    $s[i] = [i]$ //store the element index in subsequence
    for $0 \leq j \leq i-1$:
        if $h_i > h_j$:
            if $dp[j] + 1 > dp[i]$:
                $dp[i] = dp[j] + 1$
                $s[i] = s[j] + [i]$ //concatenate two list

or

for $0 \leq i \leq n-1$:
    for $0 \leq j \leq i-1$:
        if $h_i > h_j$:
            if $\text{len}(s[j]) + 1 > \text{len}(s[i])$:
                $s[i] = s[j] + [i]$ //concatenate two list

Rubrics (5 points: 1 for base, 2 for final, 2 for pseudo):

- if recurrence relation uses opt[n-1] by default : pseudocode points: -2
- Wrong base case: -1 point, wrong final answer: -2 points
- Missing subsequence or max length as final answer: -1 point
- Incorrect pseudocode: -2 points

d) What is the complexity of your solution? (2 pts)

$O(n^2)$

Rubrics (2 points):

- Wrong/missing time complexity: -2 points
- NOTE: only time complexity is expected, space complexity is NOT necessary (no points deduction if their space complexity is computed but wrong)
- Inefficient algorithm (exponential time complexity): -1 point

Additional space

Additional space

Additional space