Midterm Exam

CS 6150

Exam date: Monday, Oct 23, 2023

NAME: ____________________ UID: ____________________

This exam has 8 questions for a total of 50 points plus 6 bonus points (the grading will be on a scale of 50).

Question	Points	Score
1	6	
2	6	
3	10	
4	10	
5	6	
6	6	
7	6	
8	6	
Total:	56	

Question 1: .... **[6]**

Consider the recurrence equation $T(n) = \frac{3}{2}T(n/2) + O(n)$. What is a closed form for $T(n)$? (No explanation/derivation is necessary)

(a) $\underline{T(n) = \Theta(n)}$

(b) $T(n) = \Theta(n^2)$

(c) $T(n) = \Theta(n^3)$

(d) $T(n) = \Theta(n^2 \log n)$

Selecting the correct option	6 points

We can use an inductive argument to show that $T(n) \in O(n)$. Since $T(1)$ is some constant then trivially there exists some C such that $T(1) \leq C \cdot 1$. Assume $T(n) \leq Cn$ for all $n < k$ for some k. Then, we can see that $T(k) = \frac{3}{2}T(k/2) + O(n) \leq \frac{3}{2}C\frac{k}{2} + C'k = \frac{3}{4}Ck + C'k$ where C' is the constant associated with the $O(n)$ term). By setting $C > 4C'$, we get $T(k) \leq Ck$ and therefore by induction we get that $T(n) \in O(n)$. Note that all the other options except (a), have complexity that is in $\omega(n)$. Therefore, (a) would be the correct choice.

NAME: ______________________ UID: ______________________

Question 2: .... **[6]**

Suppose we are given an unsorted array $A[0, 1, \ldots, n-1]$ whose elements are **all distinct**. Our goal is to pre-process the array, so as to answer rank queries quickly. Given some number x that belongs to the array, its rank is the position of x in the sorted order.

Show how to pre-process the array, so that (a) pre-processing takes time $O(n \log n)$, and (b) any rank query takes time $O(\log n)$. [You must clearly write down the steps involved, and why the desired time bounds hold – using any knowledge from class as a blackbox.]

Recognizing the pre-processing method	3 points
Correctly explaning how to solve a rank query in $O(\log n)$ time	3 points

Since we have an array of size n, we can see that the array A can be sorted in $O(n \log n)$ time using Merge Sort. This would place each element of A at the position of it's correct rank. So for part (a), we will simply sort the array A and get a new array B where rank of $B[i]$ is i. Now given any rank query, i.e. given any x, we just need to use the binary search. We use binary search and find the index at which x can be found in B. Let this index be r, we return r as the rank of x. Since the array is sorted and all values are distinct, r should be the correct rank. Since we use binary search, this only takes $O(\log n)$ time.

Question 3: .... **[10]**

Suppose you find that you have too many coins of each denomination in your wallet. Now, you are asked to make change for a certain amount, the goal is to use as many coins as possible.

Formally, suppose that there are k denominations of coins $d_1, d_2, \ldots, d_k$ cents. Now, suppose we wish to make change for N cents using the most number of coins. Suppose $d_1, \ldots, d_k, N$ are all positive integers.

Give an $O(Nk)$ time algorithm to find the largest number of coins that can be used to make change for N. Provide a solution where you: (1) write down the pseudocode, (2) give a 2-3 line explanation for its correctness, and (3) analyze the running time.

[*Hint: Use dynamic programming*]

Outline general solution (imprecise but correct idea)	2 points
Correct pseudocode	3 points
Correctness explanation	3 points
Time complexity with explanation	2 points

Let us solve using dynamic programming. Let us define a subproblem $dp[i]$, which gives the maximum number of coins that can be used to make the amount i cents. We use this subproblem to build the next ones. $dp[N]$ gives the answer using N cents. Following is the pseudocode.

Algorithm 1 Coin Change

```
1: function GetMaxCoins(d, N):
2:     $dp[0, .....N] = \{0\}$
3:     for i = 1 to N do                                        ▷ amount in cents
4:         for j = 1 to k do                                    ▷ denominations
5:             if $dp[j] < i$ then $dp[i] = max(dp[i], 1 + dp[i - d[j]])$
6:     return $dp[N]$
```

Correctness:
The solution uses the optimal substructure of the problem. For each amount i, we compute the maximum number of coins that can be used by considering each denomination d[j]. For any amount i and coin denomination $d[j]$, the maximum coins that can make the change is either we exclude the current coin (which keeps the count as $dp[i]$) or include it (which is $1 + dp[i - d[j]]$). We are breaking the problem down into smaller overlapping subproblems and building our solution in a bottom-up manner.

Time Complexity:

We run two nested loops. The outer loop runs for N iterations and the inner loop runs for k iterations. Therefore the time complexity of O(Nk).

NAME: ______________________ UID: ______________________

Question 4: .... **[10]**

Suppose $G = (V, E)$ is a d-regular (all vertioces have the same degree d) simple, undirected graph on n vertices, which is given in the form of an adjacency list. Specifically, for each vertex, we have a *sorted* list of neighbors.

Describe an algorithm that can find if the graph contains a *triangle* (a set of three vertices, every pair of which have an edge). Your algorithm should run in polynomial time (partial credit), ideally in $O(nd^2)$ time (full credit).

You must state the pseudocode and analyze the running time. Proof of correctness is NOT needed.

Any polynomial time algorithm (partial)	4 points
Algorithm in $O(nd^2)$	3 points
Runtime Analysis	3 points

Algorithm 2 To find if graph contains triangle

```
function ListCommonVertices(adj1, adj2):
    $i \leftarrow 0$
    $j \leftarrow 0$
    $commonList \leftarrow []$
    while $i <$ len(adj1) and $j <$ len(adj2) do
        if adj1[$i$] == adj2[$j$] then
            commonList.add(adj1[$i$])
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$
        else if adj1[$i$] < adj2[$j$] then
            $i \leftarrow i + 1$
        else
            $j \leftarrow j + 1$
    return $commonList$

function ContainsTriangle(G):
    for $v$ in $G$ do
        for each neighbour $u$ of $v$ do
            $commonNeighbours = ListCommonVertices(adj[u], adj[v]$
            for $w$ in $commonNeighbours$ do
                if $w! = u$ and $w! = v$ then
                    return $true$
    return $false$
```

Time Complexity:

For the ListCommonVertices function, we are comparing the sorted adjacency lists of two vertices using the merging approach (from merge-sort). The runtime of this function is linear to the length of the lists. Therefore, the runtime is $O(d)$.
For the ContainsTriangle function, the outer loop runs for all vertices, which is n times. The second loop iterates over each neighbor of a vertex, which can be at most d times. Inside this loop, we are calling the ListCommonVertices function, which takes $O(d)$. So, the total running time is $O(nd^2)$.

Question 5: . . . . **[6]**

A burglar enters a house with a sack of "capacity" U. He scans the items in the house. There are N of them, with the ith item having size s_i and value v_i. He starts filling his sack using the rule: take the item of the largest value v_i that still fits into the sack (sum of the sizes of all the items picked must be $\leq U$).

Does this always lead to an *optimal* choice of items? (I.e., how does the total value of the items chosen compare to the value of the best subset under the size constraint?)

Answer Yes/No. If you answer Yes, you need to provide an explanation. If you answer No, provide a counter-example

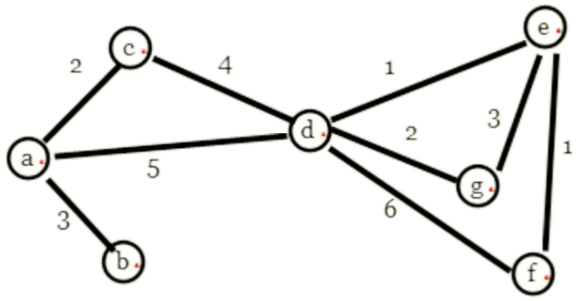
Correct answer	3 points
Counter example	3 points

No. Consider the following example, represented as an array of (size, value) pairs: [(8, 8), (5, 5), (5, 5)]

Let's consider the case where the sack has a capacity of 10. The burglar will take the first item since it fits in the sack and has the highest value. Then the burglar will leave with a total value of 8 since none of the remaining items will fit. However, if the burglar had taken the other two items with a total capacity of 10, they would leave with a total value of 10 instead of 8. This means this strategy does not work in all cases since it does not work here.

Question 6: .... **[6]**

Consider the graph shown below. Is it true that every minimum spanning tree of the graph contains edge *dg*? Provide (1) a Yes/No answer and (2) a brief explanation.



Correct answer	3 points
Explanation	3 points

Yes. For the sake of contradiction, let's assume that there is some MST of this graph that does not include the edge dg. By the definition of a spanning tree, g must be reachable from all other vertices by some path to be included in the spanning tree. As there is only one other edge connecting to g, the edge eg must be included in our MST. We can improve the cost of the MST by removing the edge eg and adding the edge dg, which has lower weight.

This does not upset our spanning tree requirement since d must have been reachable before the change, and that means g is still reachable through d now. e must also have been reachable before, and since g was only connected by a single edge, the path that connected e must not have used the edge eg, so removing it does not cut off e from the rest of the spanning tree.

Because we have improved the value of our MST by making this change, that spanning tree must not have been a minimum spanning tree, which contradicts out original assumption that an MST existed that did not include dg. Therefore, all MSTs must include the edge dg, or they could be improved by making the swap described here.

Questions 7,8

Suppose $G = (V, E)$ is a *directed* graph in which every vertex v has a value $f(v)$ associated with it. We are given G and the $f(v)$ values for all v. Suppose that the values are **all distinct**.

Consider the following problem: we wish to find the *length of the longest path* in G that has *monotonically increasing vertex values.*

Question 7: .... **[6]**

Can such a path re-visit a vertex? Provide (1) a Yes/No answer and (2) a brief explanation.

Yes/No answer	2 points
Correctly using monotonically increasing	2 points
Correctly using distinct values	2 points

(1) No;

(2) G is a simple graph, meaning no self-loop on the vertex. Suppose there exists some path P which can revisit a vertex, say v_i, along it. Due to no self-loop on the vertex, there must exist at least one another vertex called v_j immediately following v_i and followed by the same v_i when the path cycles back to v_i. With the monotonically increasing requirement along the path, the values of those vertices must satisfy:

$$f(v_i) \leq f(v_j) \leq f(v_i)$$

The only possible situation is

$$f(v_i) = f(v_j)$$

which conflicts with the prerequisite of distinct values among the vertices.

Question 8: .... **[6]**

Define "Lpath[u]" to be the lenth of the longest path with increasing vertex values *starting at u*. Consider the following recurrence for Lpath:

```
procedure LPATH($u$)
    define $S$ = set of all out-neighbors of $u$ with value $> f(u)$
    if $S$ is empty then
        return 1
    else
        return $1 + \max_{w \in S}$ LPATH($w$)
```

Does this give a **correct** way of computing Lpath values? Answer Yes/No. If you answer Yes, you need to provide an explanation. If you answer No, provide a counter-example.

Yes/No answer	2 points
Observe exhaustive search	2 points
Observe increasing vertex values along the path	1 points
Observe stop guarantee	1 points

(1) Yes;

(2) This algorithm is like BFS.

The set S contains all the out-neighbors of value > current node. This can guarantee the resulting path of increasing vertex values.

The usage $max_{w \in S} LPATH(w)$ will check all valid out-neighbors, recursively compute the longest path starting from them, and compare their results to get the maximum one. This way can search all possible paths starting from the current node.

According to Question 7, no path can re-visit the same node, meaning no unlimited nodes to explore. Thus, it guarantees that each path will be ended by some node called v of which all out-neighbors of value $< f(v)$, meaning S empty. Thus, the base case can stop the exploration and return the final result.