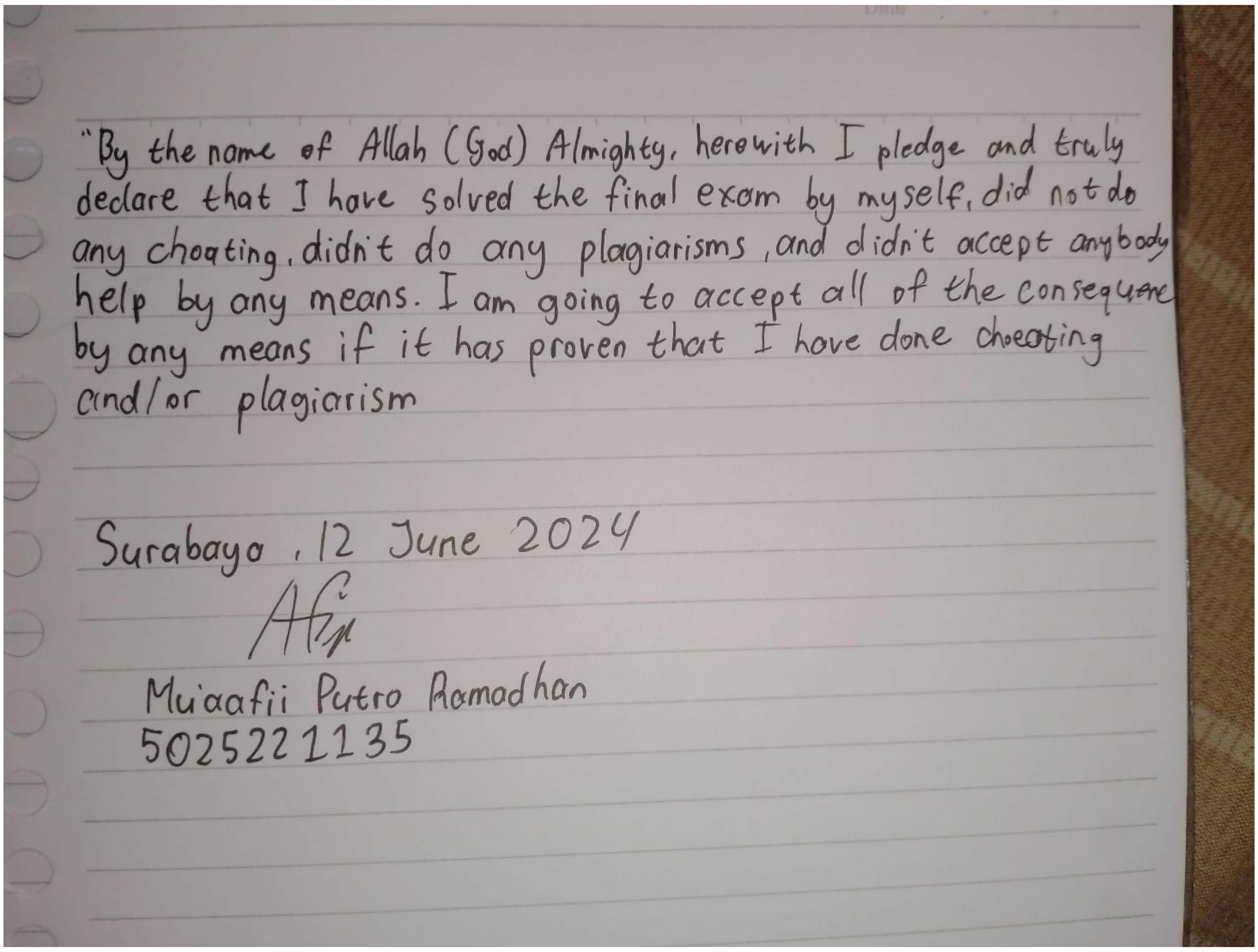
Name : Mu'aafii Putra Ramadhan
NRP : 5025221135
Class : Design & Analysis of Algorythm B

Final Test

Declaration :



"By the name of Allah (God) Almighty, herewith I pledge and truly declare that I have solved the final exam by myself, did not do any choating, didn't do any plagiarisms, and didn't accept anybody help by any means. I am going to accept all of the consequne by any means if it has proven that I have done choeating and/or plagiarism

Surabaya, 12 June 2024

[illegible]

Mu'aafii Putro Ramadhan
5025221135

Questions

https://drive.google.com/file/d/12UQYZtWR2YOpC63nLduiYq5jdqLQlbIP/view

Answer

1.True or False question

1. Quick-sort has worse asymptotic complexity than merge-sort (T/F).
   →True.
   Quick-sort has an average-case time complexity of O(n log n), which is the same as merge-sort. However, in the worst-case scenario, quick-sort can degrade to O(n^2), while merge-sort maintains its O(n log n) performance.
2. Binary search is O(log n) (T/F).
   →True.
   Binary search has a logarithmic time complexity of O(log n) when searching in a sorted array.
3. Linear search in an unsorted array is O(n), but linear search in a sorted array is O(log n).
   →False.
   Linear search in both sorted and unsorted arrays has a linear time complexity of O(n).
4. Linear search in a sorted linked list is O(n).
   True.
   →Linear search in a sorted linked list still requires traversing each element, resulting in a linear time complexity.
5. If you are only going to look up one value in an array, asymptotic complexity favors doing a linear search on the unsorted array over sorting the array and then doing a binary search (T/F).
   →True.
   For a single lookup, the overhead of sorting the array outweighs the benefits of using binary search. Linear search on the unsorted array is more efficient in this case.
6. If all arc weights are unique, the minimum spanning tree of a graph is unique (T/F).
   →True.
   If all arc weights are unique, the minimum spanning tree (MST) is unique. However, if there are duplicate weights, multiple MSTs may exist.
7. Binary search in an array requires that the array is sorted (T/F).
   →True.
   Binary search relies on the array being sorted to work correctly.
8. Insertion into an ordered list can be done in O(log n) time (T/F).
   →False.
   Insertion into an ordered list (such as an array or linked list) requires shifting elements, resulting in a linear time complexity of O(n).
9. A good hash function tends to distribute elements uniformly throughout the hash table (T/F).
   →True.
   A good hash function minimizes collisions and ensures even distribution of elements in the hash table.
10. In practice, with a good hash function and non-pathological data, objects can be found in O(1) time if the hash table is large enough (T/F).
    →True.

In ideal conditions, hash table lookups can be O(1) with a well-designed hash function and sufficient table size.

11. If a piece of code has asymptotic complexity O(g(n)), then at least g(n) operations will be executed whenever the code is run with parameter n (T/F).
    →False.
    Asymptotic complexity describes the upper bound, not the exact number of operations. The actual execution may be fewer than g(n) operations.
12. Given good implementations for different algorithms for some processes such as sorting, searching, or finding a minimum spanning tree, you should always choose the algorithm with the better asymptotic complexity (T/F).
    →False.
    Asymptotic complexity is essential, but practical considerations (such as constant factors, memory usage, and ease of implementation) also play a role in choosing an algorithm.
13. It is not possible for the depth-first and breadth-first traversal of a graph to visit nodes in the same sequence if the graph contains more than two nodes (T/F).
    →False.
    Both depth-first and breadth-first traversals can visit nodes in different sequences, even in graphs with more than two nodes.
14. The maximum number of nodes in a binary tree of height H (H = 0 for leaf nodes) is 2H+1 – 1 (T/F).
    →True.
    The maximum number of nodes in a binary tree of height H follows this formula.
15. In a complete binary tree, only leaf nodes have no children (T/F).
    →True.
    In a complete binary tree, all levels are filled except possibly the last level, where leaf nodes reside. Leaf nodes have no children.

2. Please write down a polynomial-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. Please explain in detail your assumption, the pseudo-code of this algorithm, and explain the time complexity of this algorithm. (Assume that each integer appears once in the input sequence of n numbers) [20 points]

→ **Assumptions**

- The sequence contains n integers where each integer appears only once.
- The integers are not necessarily contiguous.
- The output is the length of the longest increasing subsequence, not the subsequence itself

Pseudo-code

```
1) 1. Initialize an array dp of size n. Set all elements to 1.
      dp[i] = 1 for all 0 <= i < n
2) 2. For each element i from 0 to n-1:
      a. For each element j from 0 to i-1:
        i. If arr[j] < arr[i] and dp[j] + 1 > dp[i]:
           dp[i] = dp[j] + 1
3)  3. Initialize a variable max_length to keep track of the maximum value in dp.
4)  4. Iterate over dp to find the maximum value:
       max_length = max(max_length, dp[i])
5)  5. Return max_length
```

Python code

```python
def find_lis(arr):
    n = len(arr)
    dp = [1] * n  #LIS Length Initialization

    for i in range(1, n):
        for j in range(i):
            if arr[j] < arr[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    #LIS Length maxumum searching
    max_length = max(dp)
    return max_length
```

Time Complexity

- ➔ The outer loop runs n times (for each index).
- ➔ The inner loop runs at most n times (for each previous index).
- ➔ Overall, the time complexity is O(n^2), which is polynomial time.

3. After all, binary trees have been discussed a lot in our lecture. It's often used in search applications, where the tree is searched by starting at the root and then proceeding either to the left or the right child, depending on some condition. In some instances, the search stops at a node in the tree. However, in some cases, it attempts to cross a null link to a non-existent child. The search is said to "fall out" of the tree. The problem with falling out of a tree is that you have no information about where this happened, and often this knowledge is useful information. Given any binary tree, an extended binary tree is formed by replacing each missing child (a null pointer) with a special leaf node, namely an external node. The remaining nodes are called internal nodes. An example is shown in the figure below, where the external nodes are shown as squares and the internal nodes are shown as circles. Note that if the original tree is empty, the extended tree consists of a single external node.

Also, observe that each internal node has exactly two children and each external node has no children. Let n denote the number of internal nodes in an extended binary tree. An extended binary tree with n internal nodes has n + 1 external node. Prove this statement by induction. [20 points]

➔ Statement : An extended binary tree with n internal nodes has n + 1 external nodes.

Base Case (n = 0) :

When there's no internal node (empty tree), there's only the one external node (the root)

For this case, the statement is true (n = 0, but there's 1 node created)

Inductive step :

➔ Inductive Hypothesis: Assume that for a binary tree with k internal nodes, the extended binary tree has k+1 external nodes.

Calculation :

- Original number of external nodes: k+1
- After replacement: k+1−1=k (replace one external node)
- Add two new external nodes: k+2

By induction, we have shown that for any non-negative integer n, an extended binary tree with n internal nodes will have n+1 external nodes. This proves the statement as true

4. We assumed that G = (V, E) be a complete graph with the set of vertices V = {1, 2, ..., n} and weights w(i, j) = dij where dij isthe distance from the input of Traveling Salesman Problem (TSP). Then we can do the following steps.

(1) Select a vertex r □ V to serve as a root for the future tree. (2) Build a minimum spanning tree T for G with the root r using a polynomial-time minimum spanning tree algorithm as a subroutine. (3) Construct the list L of vertices being a Pre-order Walk of T. (4) Return L as an approximate tour of G.

Please prove the following statements.
(1) The approximation algorithm for Travelling Salesman with triangle inequality is correct (i.e., produces a tour).[5 points]
(2) The running time of the algorithm is polynomial. [5 points]
(3) The algorithm has a ratio bound □(n) = 2. [10 points]

➔(1) Correctness of the Approximation Algorithm

Statement: The approximation algorithm for the Traveling Salesman with the triangle inequality is correct, i.e., it produces a tour.

Proof:

Selection of Root and MST Construction:

Selecting a vertex r as the root and constructing a minimum spanning tree (MST) T from

r ensures that all vertices V are connected without cycles, minimizing the total weight of the edges.

- Pre-order Walk:

The pre-order traversal of T creates a list L of vertices. In a pre-order walk, each vertex is visited before its descendants. This traversal touches every vertex once as T is a tree that spans all vertices.

- Formation of a Tour:

The pre-order walk returns to the starting vertex r after completing the traversal. This return step is crucial as it closes the loop, forming a Hamiltonian cycle (tour). Since

G is a complete graph, there is a direct edge from the last vertex visited in the pre-order walk back to r.

- Triangle Inequality:

Given G adheres to the triangle inequality (where the direct path between any two vertices does not exceed the path through an intermediary vertex), this method ensures the path taken between consecutive vertices in L is the shortest direct route.

Thus, the algorithm indeed produces a valid TSP tour that visits each city exactly once and returns to the starting city.

(2) Polynomial Running Time

Statement: The running time of the algorithm is polynomial.

Proof:

MST Construction:

Using a polynomial-time algorithm such as Kruskal's or Prim's, constructing the MST of a complete graph G with n vertices takes O(n^2logn) time because the graph has $\binom{n}{2}$

$\frac{n(n-1)}{2}$ edges in total.

Pre-order Traversal:

The traversal of the MST, a tree with n−1 edges, is O(n).

Overall Complexity:

The most computationally intensive step is the MST construction. Given both the MST construction and the traversal are polynomial in complexity, the overall complexity of the algorithm is polynomial, dominated by the MST construction phase.

(3) Approximation Ratio Bound ρ(n)=2

Statement: The algorithm has a ratio bound ρ(n)=2.

Proof:

- Cost of MST and TSP Tour:

Let cost(T) be the weight of the MST. It's known that cost(T)≤cost(OPT), where cost(OPT) is the weight of the optimal TSP tour.

- Tour Construction by Pre-order Walk:

The tour constructed by the pre-order walk of T follows the tree edges and potentially adds one additional edge (from the last vertex back to r). The weight of this tour is at most twice the weight of T because each edge of the MST might be traversed at most twice (once during the actual traversal and once potentially on the return to r).

- Bounding the Approximation:

Hence, cost(L)≤2×cost(T)≤2×cost(OPT). This establishes that the approximation ratio ρ(n)=2.

5. Bellman-Ford and Dijkstra are two algorithms to find the shortest path from one vertex to all other vertices of a weighted graph.

Please do the following:
(1) Describe their differences and their respective real-life applications. [10 points]
(2) Write pseudocode for each algorithm and try to show where they differ. [10 points]
(3) Create a short illustration of how each algorithm works. [5 points]

➔ **(1) Description :**
Djikstra → Find the shortest path from a single source vertex to all other vertices in a non-negative weighted graph.
Bellman-Ford →Find the shortest path from a single source vertex to all other vertices in a weighted graph.

Differences :
Bellman-Ford Algorithm:
- Works with graphs where edges can have negative weights.
- Detects negative weight cycles.
- Overestimates path lengths initially and iteratively relaxes them.
- Applications: Used in scenarios involving cashflow, chemical reactions (heat absorption/heat dissipation), and optimization problems with negative weights.

Dijkstra's Algorithm:
- Works on graphs with non-negative edge weights.
- Greedy approach (selects the nearest unvisited vertex).
- Cannot handle negative weight edges or detect negative weight cycles.
- Applications: Network routing, GPS navigation, and shortest path problems in transportation networks.

(2) Psudeucode :

- **Bellman-Ford**

```
function bellmanFord(G, S):
  for each vertex V in G:
      distance[V] = infinity
      previous[V] = NULL
  distance[S] = 0

  for each vertex V in G:
      for each edge (U, V) in G:
          tempDistance = distance[U] + edge_weight(U, V)
          if tempDistance < distance[V]:
              distance[V] = tempDistance
              previous[V] = U

  for each edge (U, V) in G:
      if distance[U] + edge_weight(U, V) < distance[V]:
          Error: Negative Cycle Exists

  return distance[], previous[]
```

- **Djisktra**

```
function dijkstra(G, S):
    for each vertex V in G:
        distance[V] = infinity
        previous[V] = NULL
    distance[S] = 0

    while Q is not empty:
        U = Extract MIN from Q
        for each unvisited neighbor V of U:
            tempDistance = distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]:
                distance[V] = tempDistance
            previous[V] = U


   return distance[], previous[]
```

)(3) Illustration :

Illustration of Bellman-Ford Algorithm:

1) Initialize distances and predecessors.
2) Relax edges iteratively, updating distances.
3) Detect negative weight cycles (if any).

Illustration of Dijkstra’s Algorithm:

1) Initialize distances and priority queue.
2) Extract the minimum distance vertex.
3) Relax neighbors and update distances.