Final Exam

Date: 2023-08-03 19:16
type: #Homework
Category: 📌cs 311 MOC

Cumulative Assessment

Question 1 - Short Answer

> Short Answer Questions. You are not required to justify your answers.
> What is the tightest upper bound (in terms of big-O) on the runtime of $T(n) = 2T\left(\frac{n}{2}\right) + n$?

$O(n \log n)$

> What is the runtime of BFS (Breadth First search)?

$O(V + E)$, where V is the number of vertices and E is the number of edges.

> Consider the directed graph $G = (V, E)$ where $V = (A, B, C)$ and $E = ((A, B), (B, C), (C, B))$, How many strongly-connected components does G have?

2

> True/False: $n + 1 \in O(n^2)$

True

> True/False: $n^3 + n \in O(n^2)$

False

Question 2 - Divide and Conquer

> Let $A = (a_1, a_2, \ldots, a_n)$ be an array of integers. Write an algorithm that will output i and j such that $i < j$ and $a_j - a_i$ is maximized. Use a divide and conquer to arrive at your algorithm. Your algorithm must be recursive. No credit for non-recursive algorithms. Describe the algorithm, state the recurrence to capture the runtime of your algorith and write the solution to the recurrence. Part of the grade depends on efficiency.

Keep dividing the array into equal halves and find the maximum difference in each half, on top of considering the difference between the max element of the right and min element of the left.

1. Divide the array into two halves, `left` and `right`.
2. Recursively find the (i_l, j_l) and (i_r, j_r) for left and right that maximize the differences.
3. Find the min element in the left half and the max element in the right half.

4. Compare the differences from each half to find the overall max difference.
5. Return the (i, j) that corresponds to the overall maximum difference.

```
findMaxDifference(A, low, high){
   if high - low <= 1:
       return (low, high)

   mid = (low + high) // 2

   (i_l, j_l) = findMaxDifference(A, low, mid) # Left half
   (i_r, j_r) = findMaxDifference(A, mid, high) # Right half

   min_left = min(A[low:mid])
   max_right = max(A[mid:high])

   left_diff = A[j_l] - A[i_l]
   right_diff = A[j_r] - A[i_r]
   cross_diff = max_right - min_left

   # Return the pair of indices that gives the maximum difference
   if left_diff >= right_diff and left_diff >= cross_diff:
       return (i_l, j_l)
   elif right_diff >= left_diff and right_diff >= cross_diff:
       return (i_r, j_r)
   else:
       return (A.index(min_left), A.index(max_right))
}
```

Recurrence Relation:

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

The constant dividing of each half gives the $2 \cdot T\left(\frac{n}{2}\right)$, and finding the max and min in each half contributes to the $O(n)$ term.

By using the master theorem, we can deduce that this recurrence into a case 2, thus the solution is:

$$T(n) = \Theta(n \log n)$$

Which is the final time complexity.

Question 3 - Greedy Algorithms

> Suppose you are traveling from Metroplois to Gotham (a distance of D miles) along highway I-W. The car in which you are traveling can hold enough gas to go $m < D$ miles. There are n gas station on the way at distances $d_1, d_2, \ldots, d_n$ from Metropolis, such that $d_1 < d_2 < \cdots < d_n$ and no two consecutive gas stations are more than m miles apart. More precisely:
>
> - $\forall i, j \in (1, 2, \ldots, n) : i < j \implies d_i < d_j$,
> - $d_1 < m$,

- $\forall i \in (1, 2, \ldots, n-1) : d_{i+1} - d_i \leq m$,
- $D - d_n \leq m$

Your objective is to make as few gas stops as possible along the way. Design an algorithm that will identify a strategy to achieve this objective. I.e., your algorithm gets $d_1, d_2, \ldots, d_n$, m and D as inputs and outputs the smallest possible list of gas stations to stop at. State the run-time of the algorithm. Part of your grade depends on the efficiency of your algorithm.

1. Start at the beginning of the highway with current location at `0`.
2. Iterate through the gas stations and check if the next station is farther away than the current location plus the car's maximum range, if so then add the current station to the list of stops and update current location
3. Continue until at the end of highway

```
minGasStops(stations, m, D){
       stops = []               # List to store the stops
   current_location = 0     # Current location of the car

   for i in range of [len(stations) - 1]:
       if stations[i + 1] > current_location + m:
           stops.append(stations[i])
           current_location = stations[i]

   # Check if need one more stop before reaching the destination
   if D > current_location + m:
       stops.append(stations[-1])

   return stops
}
```

The algorithm iterates through each gas station once, performing constant time operations for each station, therefore making the overall runtime $O(n)$, where n is the number of gas stations.

Question 4 - Dynamic Programming

Let $G = (V, E)$ be aline graph such that $V = v_1, v_2, \ldots, v_n$ and for every i, $1 \leq i \leq n-1$, there is an edge $\in E$ from v_i to v_{i+1}. Each vertex v has a non-negative weight $w(v)$. A set $S \subseteq V$ is an *Independent Set* if for every $x, y \in S$, there is no edge from x to y in G. The weight of an independent set S is $\sum_{v \in S} w(v)$. Give a dynamic programming based algorithm to compute a maximum weight independent set. Your solution must give the recurrence relation that captures the weight of maximum weight independent set and describe an interative algorithm based on the recurrence relation. No credit is given if your solution uses a recusrive algorith. Part of the grade depends on efficiency.

Recurrence Relation:

Let's define `dp[i]` as the maximum weight of the independent set from vertices v_1 to v_i. We can define the recurrence relation as follows:

- Base cases:

- $dp[0] = 0$ (no vertices)
- $dp[1] = w(v_1)$ (only one vertex)

- For $i > 1$:
  - $dp[i] = max(dp[i-1], dp[i-2] + w(v_i))$

Here, $dp[i-1]$ represents the case where we do not include the current vertex in the independent set, and $dp[i-2] + w(vi_i)$ represents the case where we do include the current vertex, and thus must skip the previous one.

Iterative Algorithm:

1. Initialize a `dp` array of length `n+1` and set `dp[0]=0` and `dp[1]=w(v_1)`.
2. Iterate from $i = 2$ to n, computing `dp[i]` based on the recurrence relation.
3. Return `dp[n]` as the maximum weight independent set.

```
maxWeightIndependentSet(weights){
    n = len(weights)
    dp = [0] * (n + 1)
    dp[1] = weights[0]

    for i in range(2, n + 1):
        dp[i] = max(dp[i - 1], dp[i - 2] + weights[i - 1])

    return dp[n]
}
```

Time Complexity:

We iterate through the vertices once and perform constant work each vertice, therefore the time complexity is $O(n)$, where n is the number of vertices in the graph.

Question 5 - Graphs

> Given a directed graph, we say that a vertex is non-dominating if it is not reachable from any other vertex. Write an algorithm to detect the existence of such a non-dominating vertex. Your grade will depend on the efficiency of the algorithm.

1. Initialize an array to be size of n with all zeroes to represent the in-degree of each vertex, where n is the number of vertices.
2. Iterate through all the edges of the graph, for each edge (u, v), increment `arr[v]` by 1.
3. Check if any vertex has an in-degree of zero, if such a vertex exists, return true, otherwise false.

```
hasNonDominatingVertex(graph){
    n = len(graph)
    in_degree = [0] * n

    for u in range(n):
        for v in graph[u]:
            in_degree[v] += 1
```

```
    for degree in in_degree:
        if degree == 0:
            return True

    return False
}
```

The algorithm iterates through each vertice and its neighbor once, so the time complexity is $O(V+E)$ for V number of vertices and E number of edges.

Question 6 - Weighted Graphs

> Let G be a graph and let T be an MST for G. Let G' be the graph obstained by selecting one edge $e \notin T$ and decreasing its weight by a positive quantity δ. Given an algorithm that gets G, T, e and δ as input and returns a MST of the new graph G'. State the run-time. You do not have to prove the correctness. Your grade will depend on the efficiency of the algorithm.

Since T is already an MST of G, we can use it to find G':

1. Insert the edge e into T, forming a cycle.
2. Find the cycle in the updated graph
3. Identify the edge with the maximum weight in the cycle.
4. Remove the identified edge
5. return updated graph

```
updateMST(G, T, e, delta){
    # Decrease the weight of edge e by delta
    G[e] -= delta

    # Add edge e to the MST
    T.add_edge(e)

    # Find the cycle created by adding edge e
    cycle = find_cycle(T)

        # Find the maximum-weight edge in the cycle (not equal to e)
    max_weight = -1
    max_weight_edge = None
    for edge in cycle:
        if G[edge] > max_weight and edge != e:
            max_weight = G[edge]
            max_weight_edge = edge

    # If max_weight_edge is still e, find another max weight edge
    if max_weight_edge == e:
        max_weight = -1
        for edge in cycle:
            if G[edge] > max_weight and edge != e:
                max_weight = G[edge]
```

```
                max_weight_edge = edge

    # Remove the maximum-weight edge from the MST
    T.remove_edge(max_weight_edge)

    return T
}
```

the `find_cycle` algorithm can be implemented using DFS, giving a runtime analysis of `updateMST`:

- Decreasing edge weight: $O(1)$
- Adding edge e to T: $O(1)$
- Finding the cycle: $O(V+E)$ for number of vertices and edges
- Identifying and removing the max weight edge: $O(E)$

So the total run-time of `updateMST` is $O(V+E)$ for V vertices and E edges.

Question 7 - NP-Complete (EC)

> Show that the following decision problem is in NP:
> An undirected graph G is *k-colorable* if we can assign a color to each vertex of the graph such that if u and v are adjacent, then color of u differs from color of v, and the total number of colors used is at most k.
> Problem: Graph coloring
> Input: Undirected graph G
> Decision/Output: Is the graph $\lceil \log n \rceil$-colorable? Here n denotes number of vertices of G.

A graph is $\lceil \log n \rceil$-colorable if there exists a valid coloring of its vertices using at most $\lceil \log n \rceil$ distinct colors, such that no two adjacent vertices have the same color.

Verifying the Solution:

1. Guess a solution: Suppose there exists a coloring scheme that assings a color from the set $(1, 2, \ldots, \log n)$ to each vertex of the graph. Since we don't know this coloring scheme, it is a non-deterministic guess.
2. Verify the solution: Given the guessed coloring scheme:
    1. *No two adjacent vertices have the same color:* We can simply iterate through the edges of the rgaph and check that the colors of the vertices of each edge are different. This takes $O(E)$ time for E edges.
    2. *At most $\log n$ colors are used:* We check all assinged colors in the set $(1, 2, \ldots, \log n)$ which takes $O(n)$ time for n vertices.

If both conditions are met, then graph G is $\lceil \log n \rceil$-colorable, otherwise it is not.

The vertification process takes polynomial time proportional to the size of the graph's vertices and edges, this making the decision problem in NP.