CSE 101
MT Review Session
Week 4

January 29

How to prepare for the midterm. Part 1.

- Look through the previous midterms on the website. Try solving at least one of them entirely by yourself. Look at their solutions, try to understand them, and ask questions if you have any.
- Look at the solutions of HW1 and HW0 (even if you solved everything and got full credit). Again, try to understand them and also remember the style.
- Look at the model solutions at the end of the website.
- Look through all the lecture and the discussion section slides. Look at the examples and make sure you understand them and are able to "reproduce them".

How to prepare for the midterm. Part 2.

- Have a good understanding of all the algorithms covered in the lectures. Run them on several examples (for example, run BFS on some graphs).
- Know the differences and similarities between the algorithms (BFS vs DFS, BFS vs Dijkstra and so on)
- Know when each algorithm can be applied. Know why we can't apply them for specific types of inputs (Dijkstra vs negative weights)
- You should know the time complexities of the algorithms. Some algorithms may have several time complexities depending on which data structures they use (Dijkstra, for example).
- You should be able to answer questions like "What is the time complexity of BFS when the graph is nearly full?" or "What is the time complexity of DFS when the graph is a tree?".

How to prepare for the midterm. Part 3.

- Learn different applications of each algorithm (lecture, discussion section, and homework review slides)
- Again, be familiar with all the examples from the slides. Look for HW2 review slides as well.
- Look through the quizzes on the website.
- Look at the **Things to Know** section on the website. There are many useful questions. You should know answers to all of them (the relevant ones, no need to study flows, for example)

Example 1

Statement. You are given an undirected graph $G = (V, E)$ and a sequence of m pairs of nodes (queries) (u_1, v_1), (u_2, v_2), ... , (u_m, v_m). Determine whether u_i is reachable from v_i for each pair (u_i, v_i).

Example 1

Statement. You are given an undirected graph $G = (V, E)$ and a sequence of m pair of nodes (queries) (u_1, v_1), (u_2, v_2), ... , (u_m, v_m).

Solution. Find all connected components in this graph. Two nodes belong to the same connected component if and only if they are reachable from each other.
Use DFS to color all the nodes with colors from 1 to k, where k is the number of connected components in the graph. For each pair check, whether both nodes have the same color.

Time complexity. $\Theta(|V| + |E| + q)$

Example 2

Statement. You are given an array of n elements - $A[1...n]$. Find the sum of all elements in the array. Implement a Divide-and-Conquer algorithm to solve this problem.

Example 2

Statement. You are given an array of n elements - $A[1...n]$. Find the sum of all elements in the array. Implement a Divide-and-Conquer algorithm to solve this problem.

Solution 1. One of the possible ways is to divide the array into two equal (or almost equal parts) $A[1...m]$ and $A[m+1...n]$, where $m = n/2$. Find the sum of all elements in these two sub-arrays and then combine their results (add them). If there is only one element in the array, then just return the element.

Time complexity. $T(n) = 2T(\frac{n}{2}) + \Theta(1)$. Let's apply Master theorem: $a = 2, b = 2, d = 0$, $a > b^d$, thus,
$T(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$

Example 3

Statement. You are given an array of n elements - $A[1...n]$. Find the sum of all elements in the array. Implement a Divide-and-Conquer algorithm to solve this problem.

Example 3

Statement. You are given an array of n elements - $A[1...n]$. Find the sum of all elements in the array. Implement a Divide-and-Conquer algorithm to solve this problem.

Solution 2. Another way (a bad way, don't do it) is to break the array into sub-arrays: $A[1...n-1]$, $A[2...n]$. Find sums in these sub-arrays, let them be s_1 and s_2 respectively. Then, combine them: $(s_1 + s_2 + A_1 + A_n)/2$ - the sum of all elements in the array A.

Time complexity. $T(n) = 2T(n-1) + \Theta(1)$. Using unraveling we can show that $T(n) = \Theta(2^n)$.

Example 4

Statement. You are given an array A of n elements: $a_1, a_2, ..., a_n$. Find the sum of continuous sub-array of the array A, that has the largest possible sum (the sum of all elements in it is maximized)

Example. $A = (1, -2, 3, 4, -5, 20)$
The answer is: 22, the sub-array is $(3, 4, -5, 20)$

Example 4

Solution. Part 1.

- Let's solve this problem with Divide-and-Conquer. Let's say we have an array $A[i...j]$. Let $s(i,j)$ be the sum of all elements in this array, $f(i,j)$ be the sum of the sub-array with the largest sum, $L(i,j)$ be the sum of the sub-array with the largest sum that start at index i, and $R(i,j)$ be the sum of the sub-array with the largest sum that ends at index j. The answer to the problem is $f(1,n)$.
- Let's divide our array A into two parts: $A[0...m]$ and $A[m+1...n]$, where $m = n/2$. Find values f, g, h for each of them recursively and then combine them to find values for the entire array $A[1...n]$:
  - $L(1,n) = max(L(1,m), s(1,m) + L(m+1,n)$
  - $R(1,n) = max(R(m+1,n), s(m+1,n) + R(1,m))$
  - $f(1,n) = max(f(1,m), f(m+1,n), R(1,m) + L(m+1,n)$

Example 4

Solution. Part 2. Let's find an efficient way to find $s(i, j)$ - the sum of all elements in the sub-array $A[i...j]$. We can compute array $Sum[k]$, which equals the sum of all elements in the sub-array $A[1...k]$. Then, $s(i, j) = Sum[i] - Sum[j - 1]$, where $Sum[0] = 0$. So $s(i, j)$ can be computed in $\Theta(1)$ time.

Time complexity. $T(n) = \Theta(n) + R(n)$, $R(n) = 2R(n/2) + \Theta(1)$, $a = 2, b = 2, d = 0$, $a > b^d$, thus, $R(n) = \Theta(n)$. So, $T(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$.

Example 5

Statement. You are given an undirected unweighted graph $G = (V, E)$ and two nodes v and u in this graph. Output all nodes in V that belong to some shortest path from u to v.

Observations.

- The graph is unweighted, thus, the weights of all edges are equal to one.
- There may exist several shortest paths from u to v.
- We can find the shortest distances from u and v to all other nodes with BFS.

Example 5

Let $d(x, y)$ be the shortest distance from node x to node y. It equals ∞, if there is no path between them.

Key observation. The node x belongs to some shortest path from u to v if and only if $dist(v, x) + dist(x, u) = dist(v, u)$. (Why?)

Solution. Find the shortest distance from u and v to all other nodes with two lunches of BFS. Then, for every node x in V, output x if $dist(v, x) + dist(x, u) = dist(v, u)$.

Time complexity. $\mathcal{O}(|V| + |E|)$

Example 6

Statement. You are given a directed weighted graph $G = (V, E)$ with positive weights and a node t in this graph. Find node s in V, such that the shortest distance from s to t is as large as possible. If there is no such node, output -1.

Observations.

- The graph has positive weights, thus, we can use Dijkstra's algorithm to find the shortest paths from a single source.
- A naive solution would be to launch Dijkstra's algorithm from each node s in V and choose the node s, such that $d(s, t)$ is the largest.
- The naive solution would take $\mathcal{O}(|V|^3)$ time in the worst case. Can we do better?

Example 6

Key observation. Construct a reversed graph G^R. The shortest distance from t to s in G equals the shortest distance from s to t in G^R. (Why?)

Solution. Find the shortest distance from t to all other nodes in G^R. Choose the node s, such that $d(t, s)$ is the largest. If there is no such node, output -1.

Time complexity. $\mathcal{O}(|V|^2)$