Section 07: Sorting, divide-and-conquer graph traveral

Section Problems

1. In-Depth Recurrence

Consider the following recurrence: $A(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 3A(n/6) + n & \text{otherwise} \end{cases}$

We want to find an *exact* closed form of this equation by using the tree method. Suppose we draw out the total work done by this method as a tree, as discussed in lecture. Let n be the initial input to A.

(a) What is the size of the input at level i (as in class, call the root level 0)?

(b) What is the number of nodes at level i?

Note: let $i = 0$ indicate the level corresponding to the root node. So, when $i = 0$, your expression should be equal to 1.

(c) What is the total work at the i^{th} **recursive** level?

(d) What is the last level of the tree?

(e) What is the work done in the base case?

(f) Combine your answers from previous parts to get an expression for the total work.

(g) Simplify to a closed form.

Note: you do not need to simplify your answer, once you found the closed form. Hint: You should use the finite geometric series identity somewhere while finding a closed form.

(h) Use the master theorem to find a big-Θ bound of $A(n)$.

2. Recurrences

For each of the following recurrences, find their closed form using the tree method. Then, check your answer using the master method (if applicable). It may be a useful guide to use the steps from part 2 of this handout to help you with all the parts of solving a recurrence problem fully.

(a) $J(k) = \begin{cases} 1 & \text{if } k = 1 \\ 5J(k/5) + k^3 & \text{otherwise} \end{cases}$

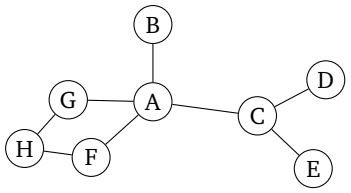
(b) $S(q) = \begin{cases} 1 & \text{if } q = 1 \\ 2S(q-1) + 1 & \text{otherwise} \end{cases}$

(c) $Z(x) = \begin{cases} \log(x) & \text{if } x \leq 9 \\ 3Z(x/3) + 1 & \text{otherwise} \end{cases}$

(d) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 3 & \text{otherwise} \end{cases}$

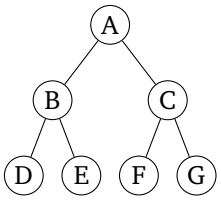
3. Graph traversal

(a) Consider the following graph. Suppose we want to traverse it, starting at node A.



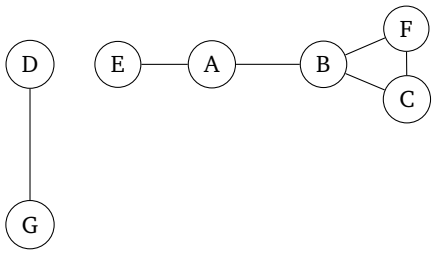
If we traverse this using *breadth-first search*, what are *two* possible orderings of the nodes we visit? What if we use *depth-first search*?

(b) Same question, but on this graph:



4. Graph properties

(a) Consider the *undirected, unweighted* graph below.



Answer the following questions about this graph:

(i) Find V, E, $|V|$, and $|E|$.

(ii) What is the maximum *degree* of the graph?

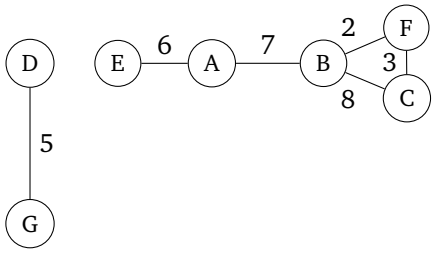
(iii) Are there any cycles? If so, where?

(iv) What is the maximum length simple path in this graph?

(v) What is one edge you could add to the graph that would increase the length of the maximum length simple path of the new graph to 6?

(vi) What are the *connected components* of the graph?

(b) Consider the *undirected, weighted* graph below.



Answer the following questions about this graph:

(i) What is the path involving the least number of nodes from E to C? What is its cost?

(ii) What is the minimum cost path from E to C? What is its cost?

(iii) What is the minimum length path from E to C? What is its length?

5. Implementing graph searches

(a) Come up with pseudocode to implement *breadth-first search* on a graph, given a starting node and an adjacency list representation of the graph. Is your method recursive or not? What data structures do you use?

(b) Come up with pseudocode to implement *depth-first search* on a graph, given a starting node and an adjacency list representation of the graph. Is your method recursive or not? What data structures do you use?

Challenge Problems

6. Recurrences

For each of the following recurrences, find their closed form using the tree method. Then, check your answer using the Master Theorem (if applicable).

(a) $Y(q) = \begin{cases} 1 & \text{if } q = 1 \\ 8T(q/2) + q^3 & \text{otherwise} \end{cases}$

(b) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + 4n^2 & \text{otherwise} \end{cases}$

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/3) + 18n^2 & \text{otherwise} \end{cases}$

7. Divide and conquer

Given an array containing elements of type `E` design an algorithm that finds the **majority element** – that is, an element that appears more then $n/2$ times. If no majority element exists, return `null`.

Your algorithm should run in $\mathcal{O}(n \log(n))$ time (and use only $\mathcal{O}(1)$ extra memory).

Note: the items in the array do **NOT** implement `compareTo`. This means you cannot sort the array!

Challenge: can you find the majority in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ extra memory?