Section 07: Solutions

1. Tree method

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

(a) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/6) + n & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 3^i
- Total work done per level: $3^i \cdot \frac{n}{6^i}$
- Total number of *recursive* levels: $\log_6(n)$
- Total work done in base case: $3^{\log_6(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_6(n)-1} 3^i \cdot \frac{n}{6^i}\right) + 3^{\log_6(n)}$$

After applying the finite geometric series identity (see quickcheck solutions for details), we get:

$$n \cdot \frac{\left(\frac{3}{6}\right)^{\log_6(n)} - 1}{\frac{3}{6} - 1} + 3^{\log_6(n)}$$

If we were to simplify (again, see quickcheck solutions for details), we get:

$$\begin{aligned} T(n) &= n \cdot \frac{\left(\frac{3}{6}\right)^{\log_6(n)} - 1}{\frac{3}{6} - 1} + 3^{\log_6(n)} \\ &= -2n \cdot \left(\left(\frac{3}{6}\right)^{\log_6(n)} - 1\right) + 3^{\log_6(n)} \\ &= -2n \cdot \left(n^{\log_6(3/6)} - 1\right) + n^{\log_6(3)} \\ &= -2n^{\log_6(3)} + 2n + n^{\log_6(3)} \\ &= 2n - n^{\log_6(3)} \end{aligned}$$

We can apply the master thereom here. Note that $log_b(a) = \log_6(3) < 1 = c$, which means that $T(n) \in \Theta\left(()\, n^c\right)$. So, we get $T(n) \in \Theta\left(n\right)$. This agrees with our simplified form.

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

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 8^i
- Total work done per level: $8^i \cdot \frac{q^3}{2^{3i}} = 8^i \cdot \frac{q^3}{8^i}$
- Total number of *recursive* levels: $\log_2(q)$
- Total work done in base case: $8^{\log_2(q)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(q)-1} 8^i \cdot \frac{q^3}{8^i} \right) + 8^{\log_2(q)}$$

While we could apply the finite geometric series identity here, there's actually a simpler approach. Notice that the 8^i term cancels itself out. So, we're left with:

$$\left(\sum_{i=0}^{\log_2(q)-1} q^3 \right) + 8^{\log_2(q)}$$

We can then apply the "summation of a constant" identity to get:

$$q^3 \log_2(q) + 8^{\log_2(q)}$$

We can also apply the master theorem here. Note that $\log_b(a) = \log_2(8) = 3 = c$, so we know $Y(q) \in \Theta\left(q^3 \log(q)\right)$.

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

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 5^i
- Total work done per level: $5^i \cdot \frac{k^3}{5^{3i}} = 5^i \cdot \frac{k^3}{125^i}$
- Total number of *recursive* levels: $\log_5(k)$
- Total work done in base case: $5^{\log_5(k)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_5(k)-1} 5^i \cdot \frac{k^3}{125^i} \right) + 5^{\log_5(k)}$$

We apply the finite geometric series to get:

$$k^3 \frac{\left(\frac{5}{125}\right)^{\log_5(k)} - 1}{\frac{5}{125} - 1} + 5^{\log_5(k)}$$

If we wanted to simplify, we'd get:

$$\frac{25k^3}{24} - \frac{k}{24}$$

We can also apply the master theorem here. Note that $\log_b(a) = \log_5(5) = 1 < 3 = c$, so we know $J(k) \in \Theta\left(k^3\right)$.

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

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 2^i
- Total work done per level: $2^i \cdot 1$
- Total number of *recursive* levels: $q - 1$
- Total work done in base case: 2^{q-1}

Note that these expressions look a little different from the ones we've seen up above. This is because we aren't *dividing* our terms by some constant factor – instead, we're *subtracting* them.

So we get the expression:

$$\left(\sum_{i=0}^{q-1-1} 2^i\right) + 2^{q-1}$$

We apply the finite geometric series to get:

$$\frac{2^{q-1} - 1}{2 - 1} + 2^{q-1}$$

If we wanted to simplify, we'd get:

$$2^q - 1$$

Note that we may NOT apply the master theorem here – our original recurrence doesn't match the form given in the theorem.

(e) $Z(x) = \begin{cases} \log(x) & \text{if } x = 7 \\ 3Z(x/3) + 1 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 3^i
- Total work done per level: 3^i
- Total number of *recursive* levels: $\log_3(x) - 6$
- Total work done in base case: $\log_2(7) \cdot 3^{\log_3(x) - 6}$

Note that the height here is different, since the recursive function hits the base case when $i = 7$.

So we get the expression:

$$\left(\sum_{i=0}^{\log_3(x)-6-1} 3^i\right) + 3^{\log_3(x)-6}$$

We apply the finite geometric series to get:

$$\frac{3^{\log_3(x)-6} - 1}{3 - 1} + 3^{\log_3(x)-6}$$

If we wanted to simplify, we'd get:

$$\frac{x}{2 \cdot 3^5} - \frac{1}{2}$$

However, our closed form only holds when $x > 7$ – our recurrence doesn't define what happens if x is less then that.

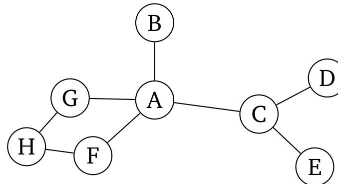
Initially, it doesn't seem like we can apply the master theorem because the base case doesn't match the exact form of the theorem.

However, since $x = 7$ in the base case, $\log(x)$ always ends up being a constant, so it actually works out.

Note that $\log_b(a) = \log_3(3) = 1 > 0 = c$, so we know $Z(x) \in \Theta\left(n^{\log_b(a)}\right)$. This ends up being $Z(x) \in \Theta\left(n^{\log_3(3)}\right)$ which simplifies to just $Z(x) \in \Theta(n)$.

2. 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*?

Solution:

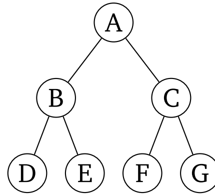
Here are two possible orderings for BFS:

- A, G, F, B, C, H, D, E
- A, C, B, F, G, D, H, E

Here are two possible orderings for DFS:

- A, G, H, F, C, D, E, B
- A, B, C, E, D, F, H, G

(b) Same question, but on this graph:



Solution:

Here are two possible orderings for BFS:

- A, B, C, D, E, F, G
- A, C, B, F, G, D, E

Here are two possible orderings for DFS:

- A, B, D, E, C, F, G
- A, C, G, F, B, E, D

3. Graph representations

(a) In class, we studied two different graph representations: adjacency lists, and adjacency matrices. Which representation would be more suited for each case?

(a) Your graph represents a map of downtown Seattle, and you want to count how many blocks are between two intersections.

(b) Your graph represents a map of downtown Seattle, and you want to check if two blocks are right next to each other.

(c) Your graph represents available flights between cities in the U.S., and you want to know if there is a direct flight between two particular cities.

(d) Your graph represents course prerequisites at UW, and you want to know what courses you need to have taken in order to take CSE 373.

(e) Your graph represents a social network like Facebook. You have several friends who claim to be friends with Mark Zuckerburg, and you wish to verify whether they are correct.

Solution:

For all of these, we can probably get away with using an adjacency list since in all five scenarios, we'd end up with a *sparse* graph.

The only exception might be scenario 1 – in that case, what we might want to do is to deliberately create a dense graph where every intersection is connected to every other intersection by an edge. We then make the edge weight the distance between the two intersections.

We'd have to do a lot of work up-front to compute all these values, but once we do, we have a quick and easy $\mathcal{O}(1)$ way of getting the answer for any inputs the user may plug in. If we expect to need to check the distance between intersections very frequently, spending the time up-front to generate the full (dense) graph mauy be worth it.

(b) Suppose you are trying to implement a graph. You decide to use a similar strategy to how we implemented trees, and decide to do the following:

   (a) You create a "Graph" class with a "Vertex" private inner class. Each "Vertex" object contains any data about that vertex, as well as a list of pointers to any directly adjacent vertex.

   (b) The "Graph" class then has a single field pointing to some arbitrary vertex in the graph.

   Evaluate the strengths and weaknesses of this design.

   Solution:

   The nice thing about this approach is that it lets us write graph algorithms in a way that's similar to how we've been writing tree problems: we have an inner node class with a map or list of adjacent nodes (similar to how tree nodes have pointers to their children).

   The problem, however, is that given a vertex we have no clean way of looking up the corresponding node. The "Graph" class contains a pointer to only a single node so to look up any other one, we'd need to traverse the graph in some way.

   We also have no good way of representing graphs that contain multiple unconnected components (or for that matter, nodes that are unreachable from whatever our starting point is).

   We can try and repair these issues by keeping track of a pointer to each node, but at that point, we're basically left with a solution that's nearly identical to adjacency lists.

4. Designs with graphs

Suppose you have social network data for some people (including yourself and a famous person). Write pseudocode to find answers to the following questions:

(a) How would you represent this social network with a graph?

   Solution:

   One way is to represent each person as a vertex, and an edge as a friendship.

   Friendships, at least on most social media websites, are bidirectional so our edges would be undirected (and also likely unweighted). We (presumably) can't be friends with ourselves, and can't have "multiple" friendships with people, so our graph wouldn't contain self-loops or parallel edges. Our graph is likely to be sparse (most people are friends with only a small percentage of the total userbase), so an adjacency list is probably the best representation.

   The only exception might be celebrities, who might legitimately be friends with thousands to millions of people. This isn't necessarily a problem – if we make our adjacency list internally use sets instead of lists to store edges, the asymptotics end up being the same.

   However, if the number of friends a celebrity ha ends up extremely large, we may want to special-case them somehow. Perhaps in our graph, we label vertices with many connections with some sort of special marker that tells any graph traversal algorithm to look up details about that person in a special way?

   In practice, there are many other considerations you need to take into account, especially for websites like Facebook. If you can't fit information about all your users into a single computer, what do you do? What happens if the computer crashes – do you have backups? Can your algorithm correctly detect when a computer crashes/when to switch to a backup? How do you keep your backups in sync? If a user changes their info from a computer and their phone nearly simultaneously, how do you decide which update "wins"?

   That said, while these are all major engineering concerns, they're also out of the scope of this class. We'll stick with the "we can represent this as an simple undirected graph in adjacency list form" answer.

(b) Given two people, how would you determine if they were friends?

Solution:

Look up one of the friends and check and see if there's an edge between them.

(c) How would you find the person with the most friends in the data?

Solution:

Iterate through each vertex, and see which one has the largest degree.

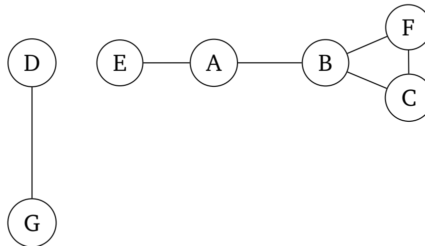
(d) How would you find the length of the shortest path from yourself to the famous person?

Solution:

Run BFS starting from the vertex representing yourself, and stop when you hit the desired celebrity. BFS radiates outwards in "rings" from the start – return the distance from the start once you hit the celebrity.

5. Graph properties

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



Answer the following questions about this graph:

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

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

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

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

(e) 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?

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

Solution:

(a) $V = \{A, B, C, D, E, F, G\}$ and $E = \{(D, G), (E, A), (A, B), (B, F), (F, C), (C, B)\}$. This means that $|V| = 7$ and $|E| = 6$.

(b) The vertex with the max degree is B, which has a degree of 3.

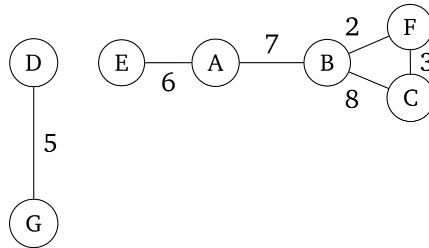
(c) There is indeed a cycle, between B, C, and F.

(d) The maximum length simple path is $(E, A), (A, B), (B, F)$.

(e) We could add the edge (D, E).

(f) One connected component is $\{D, G\}$. Another one is $\{E, A, B, C, F\}$.

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



Answer the following questions about this graph:

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

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

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

Solution:

(a) The path with the least number of nodes is $(E, A), (A, B), (B, C)$. The cost is 21.

(b) The minimum cost path is actually $(E, A), (A, B), (B, F), (F, C)$. The cost is 18.

(c) The path with the shortest length is $(E, A), (A, B), (B, C)$. The length is 3.

6. 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?

Solution:

See the pseudocode given in lecture.

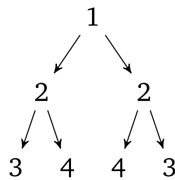
(b) Come up with pseudocode to implement *depth-first search* on a graph, given a starting node and an adjacency list representation of the graph. Is your method recursive or not? What data structures do you use?

Solution:

See the pseudocode given in lecture.

7. Applying graph algorithms

(a) Given a binary tree, check whether it is a mirror of itself (i.e. symmetric around its center). For example, this binary tree is symmetric:



Solution:

One method is to run BFS, keeping track of the nodes per each level. This lets us effectively traverse the graph level-by-level. Once we get a list of nodes on level i, we can check and see if that list is equal to the reverse of itself.

(b) You are given a data structure of employee information, which includes the employee's **unique id**, her **importance value** and her **direct** subordinates' id. Find the total importance value of a particular employee *and* all of their subordinates.

Note that this includes subordinates whose relationship with the employee is **not direct** – for example, if employee 1 is the leader of employee 2, and employee 2 is the leader of employee 3, the total importance value of employee 1 is the sum of all three employees' importances.

Solution:

Note that this list of employees, their IDs, and their subordinate IDs effectively form a graph in adjacency list form.

We can then run either BFS or DFS starting on the employee, visit all of her subordinates recursively, and sum up each importance value as we visit each node. We return that number.

(c) You have a graph with vertices representing pastures cows can graze on, and edges representing dirt roads between those pastures.

You also have several spotted and unspotted cows. The spotted cows are scared of other spotted cows, and the unspotted cows are scared of other unspotted cows. A cow on a pasture will be scared if there is a road connecting it to another pasture containing a cow of the same spottiness.

Determine if it is possible to fill all the pastures with a single cow such that none of the cows are scared.

Solution:

Model this solution as a graph, where a pasture is a vertex and a road is an edge.

We start with an arbitrary vertex and run either BFS or DFS. We also keep track of an additional boolean variable with each vertex. If a vertex's boolean is 'true', we place a spotted cow there. If a vertex's boolean is 'false', we place an unspotted cow there.

When we visit adjacent nodes, we take whatever our boolean was and flip it. For example, suppose pasture A is connected to pastures B, C, and D. If the boolean was 'true' for pasture A, we'd flip it and have it be false for pastures B, C, and D.

On a more general level, what we're trying to do is to produce a 2-coloring of the graph: that is, assign each node one of two colors such that no two adjacent nodes share the same color.

(d) You are trying to plan a dinner party for some event. You have exactly two tables you can use to seat everybody. Your goal is to place everybody at one of the two tables such that everybody is friends with at least one other person sitting in that table.

Assuming you know who's friends with who, design an algorithm that determines where everybody will sit.

Solution:

One cheesy solution is to just make everybody sit at the same table. This is technically a valid solution, since the problem doesn't prohibit this approach.

Suppose that instead we wanted to seat roughly the same number of people in each table – at least, as much as possible.

There are many different ways of doing this, but one quick and easy way would be to start iterating through the edges. (Each person is a vertex, each edge is a friendship.)

For every given edge, check and see whether one of the people in that friendship are already seated. If so, set that pair aside. If both people are unseated, seat them in whatever table has fewer people.

Once we're done, we're left with a list of people who are still unseated.

First, find all the people in that list who happen to have friends sitting in just one table. In that case, that person needs to be seated at precisely that table.

Finally, find all the people in the list who happen to have friends at either table. Seat them at whichever table has fewer people.

If there are people in the list who aren't friends with anybody, well, then the there's not much we can do, according to the premise of the question – we'd just have to sit them somewhere and hope they bond with somebody.

A more extreme solution might be to pick them out of the party, or lock them in a room with a few other people beforehand and force them to become friends before you let them out.

One possible follow-up question: is it possible to pick the edges in a way that minimizes the number of stragglers we need to take care of in the end? Yes – try looking up the "maximum bipartite matching" problem.

(e) Later, you change your mind: you decide that to force people to mingle, you want to make sure that each table contains mutual strangers: if somebody is sitting in table A, they are not friends with any other person who's also sitting in table A.

Again assuming you know who's friends with who, design an algorithm that will (a) determine if it's even possible to seat people this way and (b) if so, decide where everybody will sit.

Solution:

This problem is basically 2-COLOR in disguise. Produce a 2-coloring, and seat all people with one color at one table, and all people with another color at the other.

If the algorithm determines there is no possible 2-coloring that works, report an error.