Section 6: Solutions

So far this quarter, every week we've told you what technique to use. That's not how things work in the real world! In this section, we're going to practice "when you first read through a problem, how do you decide which technique to use?" Often the answer is "try a bunch of different options and see what happens", but in a time-constrained setting (say your final exam, or an interview) it's nice if your first or second choice turns out to work.

1. Technique Toolbox

1.1. Step 1: Read the problem carefully

And answer the usual quick-check questions

- Are there any technical terms in the problem? Any words that look like normal words but really are technical terms?
- What is the input type?
- What is the output type?

If you can't answer these, there's no way to figure out what technique to use—you don't even know what problem you're solving!

1.2. Step 2: Make some example input/outputs

You might stumble upon which technique to use here; for example, if you try to visualize an example input, and you start drawing a graph.

1.3. Some questions to ask

At this point, you should ask "is this really a graph modeling problem?" Some signs to look for

- The problem mentions a graph or something graph-sounding (like "routes" or "maps").
- There are "direct connecitons" between elements that could be edges.
- When you try to visualize an input example you end up drawing a graph.

If it **doesn't** feel like graph modeling, the next step to ask is probably "could I solve this problem recursively?" Try asking all of these

- Is there a natural way to split things "in half" (or thirds, or...)?
- Could I make the problem a little bit smaller?
- What's "one step" toward the solution?

These might start leading you to either a divide and conquer (with the first bullet) or dynamic programming solution. In all cases, be sure you can see "how the recursion is helping".

Finally

- Did an idea immediately jump to mind?
- Did you start a sentence with something like "Well, couldn't I *just*..."?

Then **maybe** it's time for a greedy algorithm. But really it's time for you to generate like 3 more examples and try them against your proposed algorithm. It's not fun to write a bunch of code only to realize it doesn't work! And greedy algorithms very often fail. Do some more checks before you jump into code writing.

1.4. None of the ideas worked

Take a deep breath, it's going to be ok.

1.4.1. Get a baseline algorithm

Figure out what "brute force" or any other baseline would be, and jot it down quickly on paper. And when you're scared, look to your baseline like that hang-in-there-cat motivational poster. Worst-case, you're going to use that one. And if you figured out that one, you can probably find another one. Hang in there.

1.4.2. Write a few more examples

And solve them. How are **you**, as a person, solving them? You're doing *some* process. See if you can reflect on it and realize what it looks like. That might inspire you toward an algorithm.

1.4.3. Ask yourself more questions

- Does this remind you of any of the problems you've seen before? If so, a similar approach **might** work.
- Can you solve a simpler version of the problem? If there are two variables, make one of them a constant (a **small** constant, like 1) and ask "now what would I do?" Maybe you can generalize from there.
- Can you sort the input? Assume a graph is connected or topologically sorted? What if it's a tree? What if the array contains only positive elements? Any of these might give you inspiration for the general case.

2. Try it yourself!

For each of these problems, get far enough that you're able to guess what technique you might want to use, and write down a sentence or two about what in the problem lead you toward that technique.

(a) There are a total of n courses you have to take, labeled from 1 to n. You are given a list `prerequisites` where `prerequisites[i]` $= (a_i, b_i)$ indicates that you must take course b_i first if you want to take a_i.

Return `true` if you can finish all courses. Otherwise return `false`. **Solution:**

Graph. The list `prerequisites` contains pairs of courses that can be represented as edges and the course numbers can represent vertices in a graph.

(b) You are given a list of integers `coins` representing coins of different denominations and an integer `amount` representing a total amount of money. Return the fewest number of coins you need to make up that amount. If that amount of money cannot be made up by any combination of coins, return -1. **Solution:**

Dynamic Programming. Different combination of coins must be tested to find the minimum number to use. By using a coin, the total amount of money is reduced which becomes the next sub-problem.

(c) You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i`th day. On each day, you may decide to buy and/or sell the stock. You can hold at most one share of the stock at any time. However, you can buy it and then immediately sell it on the same day. Find and return the maximum profit you can achieve. **Solution:**

Greedy/Dynamic Programming. The goal is to maximize the output, so dynamic programming can be used to memoize the smaller sub-problems for buying/selling on certain days. An indicator to try a greedy algorithm is that you can only hold one share at any time, so there might be a way to decide which share to hold and when to sell it.

(d) You are given an array of k linked-lists, each linked-list is sorted in ascending order. Merge all the linked-lists into one sorted linked-list and return it. **Solution:**

Divide and Conquer. We can split this into smaller sub-problems and we know that each linked list is sorted which is an indicator for divide and conquer algorithms.

(e) Given an array of distinct integers `nums` and a target integer `target`, return the number of possible combinations that add up to `target`. **Solution:**

Dynamic Programming. One key phrase is "number of possible combinations", so there must be a way to find the number of combinations for a sub-problem (a smaller target integer) which can be memoized using dynamic programming.

(f) There are n cities. Some of them are connected, white some are not. If city a is connected directly with city b and city b is connected directly with city c, then city a is connected indirectly with city c. A province is a group of directly or indirectly connected cities and no other cities outside of the group. You are given an $n \times n$ matrix `isConnected` where `isConnected[`i`][`j`]` $= 1$ if the i^{th} city and the j^{th} city are directly connected, and `isConnected[`i`][`j`]` $= 0$ otherwise. Return the total number of provinces. **Solution:**

Graph. Some key words here are "n cities" and "connected directly/indirectly" which represents a graph where cities are the vertices and whether they are connected or not as the edges.

(g) You are given an integer array `nums`. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position. Return `true` if you can reach the last index or `false` otherwise. **Solution:**

Greedy/Dynamic Programming. A greedy algorithm can be used to always keep track of the furthest location you can jump to given which locations you've already been and similarly, with dynamic programming, it can be done by memoizing the furthest location for each location.

(h) Write an efficient algorithm that searches for a value `target` in an $m \times n$ integer matrix. This matrix has the following properties

(i) Integers in each row are sorted in ascending order from left to right.

(ii) Integers in each column are sorted in ascending order from top to bottom.

Solution:

Divide and Conquer. Some keywords here are "search" and "ascending order", so we know each column and row are sorted which can be split into smaller matrices that are sorted as well to find a value.

(i) You are given an integer array `height` of length n. There are n vertical lines drawn such that the two endpoints of the i^{th} line are $(i, 0)$ and $i,$ `height`$[i]$. Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. **Solution:**

Greedy. The goal is to maximize the height and width, so walls that are further away and taller should be prioritized.

(j) There is a group of n people labeled from 0 to $n - 1$ where each person has a different amount of money and a different level of quietness. You are given an array `richer` where `richer[`i`]` $= [a_i, b_i]$ indicates that a_i has

more money than b_i and an integer array `quiet` where `quiet`$[i]$ is the quietness of the i^{th} person. All the given data in `richer` are logically correct (i.e., the data will not lead you to a situation where x is richer than y and y is richer than x at the same time). Return an integer array `answer` where `answer`$[x] = y$ if y is the least quiet person (that is, the person y with the smallest value of `quiet`$[y]$) among all people who definitely have equal to or more money than the person x. **Solution:**

> Graph. There is a relationship that is described here that shows a dependency such as showing that person a is richer than person b, so a graph would be used to map out the correct dependencies.

(k) Given an integer array `nums`, return an integer array `counts` where `counts`[i] is the number of smaller elements to the right of `nums`[i]. **Solution:**

> Divide and Conquer. Since we want to find the number of smaller elements to the right of an element in an unsorted array, there must be some sorting that occurs, so a modified divide and conquer sorting algorithm can be used.

(l) You are given several `boxes` with different colors represented by different positive numbers. You may experience several rounds to remove boxes until there is not box left. Each time you can choose some continuous boxes with the same color (i.e., composed of k boxes, $k \geq 1$), remove them and get $k * k$ points. Return the maximum points you can get. **Solution:**

> Dynamic Programming. The indicator here is that we want to try out different permutations of removing boxes to maximize the output. After removing boxes, the problem becomes smaller, so dynamic programming should be used to memoize the solutions to smaller sub-problems.

(m) There are n rooms labeled from 0 to $n - 1$ and all the rooms are locked except for room 0. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key. When you visit a room, you may find a set of distinct keys in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms. Given an array `rooms` where `rooms`$[i]$ is the set of keys that you can obtain if you visit room i, return `true` if you can visit all the rooms, or `false` otherwise. **Solution:**

> Graph. There is a dependency on being able to visit a room since you must have found a key in the previous room to unlock it and the goal is to visit all rooms. Because of this, the problem can be represented as a graph with the goal as traversing through all nodes of the graph.

(n) Given an integer n, return the least number of perfect square numbers that sum to n. A perfect square is an integer that is the square of an integer, in other words, it is the product of some integer with itself. For example, 1, 4, and 9 are perfect squares while 3 and 11 are not. **Solution:**

> Dynamic programming. If a perfect square s is chosen to be part of the solution, then finding the least number of perfect square numbers that create $n - s$ is another sub-problem that should be memoized.

(o) You are given a network of n locations labeled from 1 to n. You are also given `times`, a list of travel times such that `times`$[i]$ = (u_i, v_i, w_i), where u_i is the source, v_i is the destination and w_i is the time it takes for a signal to travel from the source to the destination. We will send a signal from a given location k. Return the minimum time it takes for all n locations to receive the signal. If it is impossible for all n locations to receive the signal, return -1. **Solution:**

Graph. The key takeaway here is that the list `times` is a list of weighted edges since for every element, it provides the source, destination and the weight. Another element in this question is that it wants to find the minimum time it takes for all n locations to receive a signal which alludes to some sort of graph traversal.

(p) You are given two integers n and k and two integer arrays `speed` and `efficiency` both of length n. There are n engineers numbered from 1 to n. `speed`$[i]$ and `efficiency`$[i]$ represent the speed and efficiency of the ith engineer respectively. Choose at most k different engineers out of the n engineers to form a team with the maximum performance. The performance of a team is the sum of their engineers' speeds multiplied by the minimum efficiency among their engineers. Return the maximum performance of this team. **Solution:**

Greedy: Less trivial greedy idea: for each candidate, we treat him/her as the one who has the minimum efficiency in a team. Then, we select the rest of the team members based on this condition.

(q) There are n piles of stones arranged in a row. The ith pile has `stones`$[i]$ stones. A move consists of merging exactly k consecutive piles into one pile, and the cost of this move is equal to the total number of stones in these k piles. Return the minimum cost to merge all piles of stones into one pile. **Solution:**

DP: since different merging orders could lead to the same subproblem, using a memo structure could reduce the amount of calculation needed.

(r) A series of highways connect n cities numbered from 0 to $n-1$. You are given a 2D integer array highways where `highways`$[i] = [$`city1`$_i$, `city2`$_i$, `toll`$_i]$ indicates that there is a highway that connects `city1`$_i$ and `city2`$_i$, allowing a car to go from `city1`$_i$ to `city2`$_i$ and vice versa for a cost of `toll`$_i$.You are also given an integer discounts which represents the number of discounts you have. You can use a discount to travel across the ith highway for a cost of `toll`$_i/2$ (integer division). Each discount may only be used once, and you can only use at most one discount per highway. Return the minimum total cost to go from city 0 to city $n-1$, or -1 if it is not possible to go from city 0 to city $n-1$. **Solution:**

Graph: Cities are the nodes and highways are undirected edges.

(s) A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not). Given two strings source and target, return the minimum number of subsequences of source such that their concatenation equals target. If the task is impossible, return -1. **Solution:**

Greedy: we greedily concatenate source and check if we have target as a subsequence of the concatenated string or not.

3. DP on Trees

You are given a tree $T = (V, E)$ with nonnegative edge weights. You want to determine the diameter of the tree, which is the longest distance between any two nodes. The goal is to design a DP which runs in $\mathcal{O}(n)$ where $n = |V|$.

3.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation? It will be useful to fix a root node in the tree first, then every node in the tree has a list of children nodes you can access. **Solution:**

Let START(v) be the distance of the longest path starting and v and ending in node in the subtree rooted at v. Let THRU(v) be the longest path which passes through v but stays inside the subtree rooted at v.

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time). **Solution:**

$$\text{START}(v) = \begin{cases} 0 & \text{if } v \text{ has no children} \\ \max\limits_{c \text{ is a child of } v} \{w_{(c,v)} + \text{START}(c)\} & \text{otherwise} \end{cases}$$

$$\text{THRU}(v) = \begin{cases} 0 & \text{if } v \text{ has less than two children} \\ \max\limits_{\substack{c_1, c_2 \text{ distinct} \\ \text{children of } v}} \{w_{(c_1,v)} + \text{START}(c_1) + w_{(c_2,v)} + \text{START}(c_2)\} & \text{otherwise} \end{cases}$$

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)? **Solution:**

$\max_v\{\text{START}(v), \text{THRU}(v)\}$.

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one. **Solution:**

The longest path that starts at a node v and ends in a node in a subtree rooted at v has two cases. Either this longest path is empty since v has no children, or the longest path goes through some child of v. That path must consist of the edge from v to its child plus the longest path starting at the child and staying within the subtree rooted at the child.

Similarly, for the longest path that passes through a node v, if it has less than two children then there is no such path. Otherwise we know that the path must go through two of v's children. Hence the longest path is the sum of the two edges that go to v's children plus the longest paths that start at each children respectively and stays within the subtrees rooted at each child.

Both recurrences encode only distance of paths. Also the longest path in the tree will have a unique node which is closest to the root and that path can either start at that node or pass through it, thus maxing over all vertices in both START and THRU is sufficient for finding this distance.

3.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm. **Solution:**

We need two lists both of size n.

(b) Describe a filling order for your memoization structure. **Solution:**

We fill from leaves up to the root, making sure that we have evaluated on all children node before evaluating a parent node.

(c) State and justify the running time of an iterative solution. **Solution:**

For each node, we need to only access the weight of the edges between the node and its children. For START this means we in total only need $|E|$ many calls total. For THRU we can implement the max over two children by just finding the largest and second largest child, so again $|E|$ many calls in total. Thus our runtime is $\mathcal{O}(|E|) = \mathcal{O}(n)$.