Accessible PDF Content: Week_2_2.pdf

Accessibility Notes:

Processed for: ESL / Simple English Translation

Here is the simplified text:

Week 2: Greedy Algorithms for Optimization Problems (LeetCode Problem 322)

You are given a list of whole numbers called coins. These numbers show the value of different types of coins.

You are also given a whole number called amount. This number is the total money you need to make.

Your task is to find the smallest number of coins needed to reach the amount.

If you cannot reach the amount using any combination of the coins, you should show the number -1.

You can imagine that you have an unlimited supply of each coin type.

Here are examples from LeetCode:

Example 1:

Input: coins = [1, 2, 5], amount = 11

Output: 3

Explanation: 11 = 5 + 5 + 1

Example 2:

Input: coins = [2], amount = 3

Output: -1

Example 3:

Input: coins = [1], amount = 0

Output: 0

Important: The problem asks for *how many* coins you use. It does not ask for *which* coins you use. For example, in Example 1, the answer is 3. It is not the list of coins [5, 5, 1].

But you can easily change the computer code to also show which coins were used. This is common for problems that ask for the *number* of items in the best answer.

Example 4

Did you watch the YouTube video from Week 1, slide 18?

Total money needed: amount = 560

Coin values: coins = [500, 100, 50, 20, 10, 5, 1]

A Simple Plan (Called a "Greedy" Plan)

Here is a clear plan to try. It is different from trying every possible way. This plan might work, or it might not:

Use the largest coin value that is less than or equal to the total amount. Use it as many times as you can. Then, go to the next largest coin value.

Here is how this plan would work in a computer program (this is like basic instructions):

You give it the total amount (a) and the list of coin values (c).

First, the program puts the coin values in order, from the largest to the smallest.

It starts a counter at zero. This counter will count how many coins are used.

It looks at each coin value, starting with the largest.

For the current coin value, it checks: Can I use this coin for the remaining amount of money?

If yes, it uses the coin. It subtracts the coin's value from the amount. It adds 1 to the counter. It keeps doing this until it can no longer use that coin (because the coin value is now larger than the remaining amount).

Then, it moves to the next largest coin value and repeats step 5.

After checking all coins, the program tells you the final count of coins used. (Note: The original instruction to "return i" in the pseudocode seems like a mistake; it should likely return the "counter").

This is an example of what we call a "greedy algorithm."

* From Wikipedia: A greedy algorithm is a computer program that solves a problem by always choosing the best option *at that exact moment*. It does not look ahead to see if this choice will be best in the future.

* A "heuristic" is a simple rule or guide that the algorithm follows to make the best choice *right now*.

* Simply put, this rule is very direct. It does not plan for the future. We hope this simple method will work well.

Back to LeetCode Problem 322

The computer code we just described (the fewest_number_of_coins function) does not solve LeetCode problem 322 correctly.

If you use it with coins = [2] and amount = 3, it will say 1 coin. But the correct answer is -1 (because you cannot make 3 with only 2-value coins).

If you use it with coins = [1, 15, 25] and amount = 30, it will say 6 coins (30 = 25+1+1+1+1+1). But the correct answer is 2 coins (30 = 15+15).

So, what kind of plan should we use? We need a method called "Dynamic Programming." We will learn about this soon.

Coin Systems Where Our Greedy Plan Works

Our simple greedy plan does not work for *all* sets of coin values. However, it *does* work for many sets. These are called "canonical coin systems." (There are links to research papers about this.)

Here are examples of these "canonical coin systems":

1. **Coins that double:** Like [1, 2, 4, 8]. Or, more generally, coin values where each coin is at least twice the value of the previous coin.

2. **Coins with a "domination gap":** This means the next coin value is much larger than the previous one, in a specific mathematical way.

3. **"Superincreasing" coins:** Here, any coin value is larger than the sum of all the smaller coin values before it.

4. **Fibonacci coins:** Like [1, 2, 3, 5, 8, ...], where each new number is the sum of the two numbers before it.

(These are links if you want to learn more.) Are there other greedy plans that work for *all* these systems?

Greedoids (For Your Information)

(If you are interested, you can read about Greedoids here: [link1], [link2].) This is a very general idea about when a greedy plan works for a system.

Note

When you are trying to prove that your computer program is correct, do not simply assume it finds the best answer. You must *show* that it finds the best answer. That is the whole point of the proof.

When Does a Greedy Plan Work?

Usually, a greedy plan works if it meets two conditions:

1. **The "Greedy Choice" Rule:** This means that if you make the best choice at each small step, you will end up with the best choice for the whole problem.
2. **The "Optimal Substructure" Rule:** This means that the best solution for a big problem includes the best solutions for smaller parts of that problem.

How to Build a Greedy Algorithm (General Steps)

Understand the problem: Make sure it's a problem where you need to find the *best* answer from many possible answers.

Find all possible answers: Know what kinds of solutions are allowed.

Look for "Optimal Substructure": Figure out how the best answer for the whole problem can be built from the best answers to smaller parts of the problem.

Create a greedy plan: Make a step-by-step plan. At each step, choose the best option *right now*.

Show it works: Prove that these choices at each step actually lead to the best answer for the entire problem.

"Optimal" vs. "Locally Optimal"

It is hard to explain these words perfectly with just a short definition. The best way to understand them is to see examples.

Activity Scheduling (LeetCode Problem 435 / Wikipedia)

The Scheduling Problem

You have a list of activities. Each activity has a start time and an end time. Your goal is to choose the most activities possible, but none of them can happen at the same time (they cannot overlap).

This problem was also in the video.

* One greedy plan did not work: Choosing activities based on their start time.

* Another greedy plan seemed to work: Choosing activities based on their end time.

Can you think of other simple greedy plans?

Choosing by End Time Works

The greedy plan of choosing activities based on their end time worked in the video example.

Does this plan work for this type of problem always? Yes!

To prove this, we need to describe the problem very carefully. We also need a mathematical proof, using a method called "induction."

Let's Describe the General Problem

Imagine we have a list of activities, S. Each activity has a start time (si) and an end time (fi). The start time is always earlier than the end time (activities are not empty).

We say two activities "are compatible" if they do not happen at the same time. This means one activity must finish before the other one starts.

The problem is to choose the largest possible group of activities from list S, where all chosen activities are compatible (none of them overlap).

Something We Notice:

There will always be at least one best solution for this problem. Why?

1. You can always pick at least one activity by itself. So, there is always at least one group of non-overlapping activities.

2. There are a limited number of ways to group the activities. Because of this, there must be one group that is the largest.

So, if you tried every possible group (a "brute force" method), you would definitely find the answer.

How to Find the Best Solution (Using the Greedy Plan)

This greedy plan seemed to find the best answer:

Input: A list of activities, S.

Steps:

1. As long as there are activities left in S:

* Find the activity that has the *earliest* end time. Let's call this end time 'f'.

* Choose one activity that finishes at time 'f'. Let's call this activity 'A'.

* Remove activity 'A' from the list S.

* Also, remove *all other activities* from S that overlap with activity 'A'.

Output: A list of chosen activities that do not overlap.

How to Write the Computer Code (Pseudocode Explanation)

Here are the steps for a computer program using this plan:

1. First, arrange all the activities by their end times, from earliest to latest.

2. Create an empty list to store the activities you choose. Let's also set a variable, last_finish_time, to a very early time (like negative infinity).

3. Go through each activity in your sorted list:

* If the current activity's start time is *after or at the same time as* last_finish_time:

* Add this activity to your chosen list.

* Update last_finish_time to be the end time of this activity.

4. After checking all activities, give back the list of activities you chose.

Our Main Statement (Claim):

We state that for any list of activities (that is not empty), our computer program will find the *best possible group* of non-overlapping activities.

* If we can prove this statement, it means our program is correct.

* We can prove this statement using a method called "induction," by looking at "partial solutions" (parts of the answer).

Partial Solutions

Each time our program takes a step, it creates a "partial solution." This is a group of activities chosen so far.

For example, after the 'j-th' step, we have a partial solution Sj.

We start with S0 (an empty group of activities).

The groups grow bigger as the program runs: S0 is part of S1, S1 is part of S2, and so on, until Sn is the final solution.

What "Promising" Means

We say a partial solution Sj (after step 'j') is "promising" if there is a *true best solution* (let's call it OPT) that matches Sj for all the activities considered up to step 'j'.

This means: if an activity Ai (from the first 'j' activities) is in Sj, then it must also be in OPT. And if Ai is in OPT, it must also be in Sj.

Proof (Starts Here)

(We will either cover the rest of the proof today or next week.)

First Step (Base Case): Let's look at step j=0.

We know S0 is an empty group of activities. This is "promising" in a simple, logical way.

To explain: Is there a true best solution (OPT) that matches S0 for all activities from 1 to 0?

The set {1, ..., 0} is actually empty. So, the question becomes: "For every activity i in an empty list, is Ai in S0 *if and only if* Ai is in OPT?"

This statement is automatically true because there are no activities in the empty list to check.

General Note:

In math, saying "For every item 'x' in group 'X', property 'P(x)' is true" is a short way of saying "For every item 'x', *if* 'x' is in group 'X', *then* property 'P(x)' is true."

This is why the statement "For every i in the empty group, Ai is in S0 if and only if Ai is in OPT" is automatically true. It is true because the condition "i is in the empty group" is never true.

Next Step (Induction Assumption):

Now, let's assume that for some step 'j' (from 0 up to n-1), our partial solution Sj is "promising."

This means we assume there is a true best solution (let's call it OPT') that matches Sj for all activities considered up to step 'j'.

(The proof will continue next week.)