The Greedy Method & Dynamic Programming

# The Greedy Method

The greedy method is a method for algorithm design. More specifically its used to design algorithms which take in a set of inputs and output a subset of these inputs. This could be a set of nodes and connections. The algorithm arrives at a solution by stepping through the input piece by piece making decisions based on the current state.

## Optimal vs Feasible solution

When solving a problem there can be two types of solutions, optimal or feasible. If the solution is optimal it is the “best” solution to the problem, a feasible solution is any valid solution. Such as a path finding algorithms, Dijkstra’s algorithm will find the optimal solution, but A\* path finding could only find a feasible solution.

This shows an important trade off, sometimes calculating the optimal solution takes a lot of processing time, and a much faster algorithm could find a feasible solution that is near optimal.

## Example

The travelling salesman problem is a famous problem. The problem is given a set of points find the optimal route between every point that minimizes distance. This is very important in the delivery business as algorithms are required to calculate the route of delivery drivers. Using a basic method of trying every route and recording the fastest would find the optimal route but has a complexity of O(n!). So, in the real world this algorithm isn’t practical as it would take far too long to calculate a route. Instead companies use an algorithm to approximately calculate the shortest route.

## Challenge Time

Okay so you are breaking into someone’s house to steal their stuff. You have a sack which can carry objects, but as you have weak comp sci arms you can only carry a weight of M. The person has n objects, which will have a weight w, and value v, in this case:

This is in the layout of, object 1 will have a weight of 18, and a value of 25, object 2 would be 15, 24 etc.

And one last rule is that you can take a fractional amount of something, so object 1 has a weight of 18 and value of 25, but if you wanted to you can take half of it, so you would only carry a weight of 9 with a value of 12.5.

A feasible solution here would to just find any combination of object such that the sum of the weights is less than M. But you are a student and need money to spend on Robux so the optimal solution is the one that gives you the most money.

Come up with an algorithm to find the optimal solution. (don’t code this just come up with an algorithm)

## The Solution

You want the most value out of these objects but are limit by the amount you can carry. So if the sum of all the weights is less than you can carry then its simple, take all the objects.

Otherwise you will want to find out the what object will give you the most value per weight. You can find this by sorting the list of objects by the value divided by the weight (biggest to smallest). This then sorts the objects into the objects you want to take the most to least. Then just keep taking objects until you cannot carry anymore.

For the example above this is the sorted list (the object will be represented as a tuple):

(24, 15), (15, 10), (25, 18)

So then you start by taking the first object (24, 15). M – 15 = 5 so you can take more object(s) still.

The next object is (15, 10) which has a weight greater than you can carry, so you can only take half of it as 5/10 = 0.5.

So in the end you take (24, 15) + 0.5 \* (15, 10) which gives a final value of 31.5. This is the optimal solution.

# Dynamic Programming

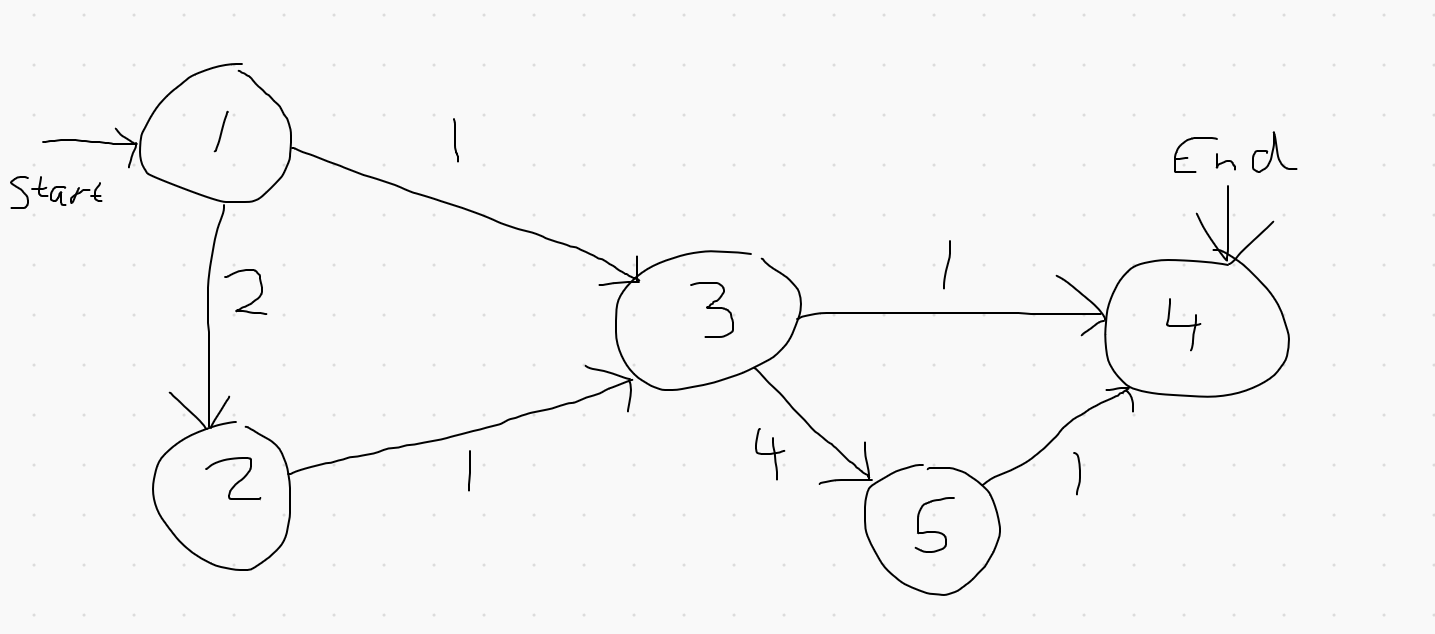
This is an algorithm design method that can be used when a problem can be viewed as a sequence of decisions. Sounds kind of similar to the Greedy Method. Well there is one key difference the Greedy Method only considers the local information available to it, such as a node on a graph would only considers its neighbours. The Greedy Method will step through a single sequence of steps making decisions as it comes to them. This can be viewed as someone driving down the motor way and making decisions based off the signs they see and make decisions as they go.

This is where the difference is, Dynamic Programming will consider many different sequences of steps. This would be analogous to putting someone in a maze and asking them to find the way out, you can only find the way out by exploring different routes.

This can be shown for the shortest path algorithm:

Lets say you are at the start node in a graph, and want to get to the end node, only looking at the nodes connected to the current node, can you decide which path would take you to the end. Unless the end node is directly connected, I’m going to say no. This is why the greedy method cannot feasibly be used for designing an algorithm to find the shortest path, it cannot step through the input and know what decisions to make.

This problem can be solved using Dynamic Programming. You can consider multiple different paths at once. Such as a bad algorithm could consider all possible routes. Lets see it in action below:



By trying every possibility (followed by their distance):

1, 3, 4 = 2

1, 3, 5, 4 = 6

1, 2, 3, 4 = 4

1, 2, 3, 5, 4 = 8

We can do a simple optimisation by stop following a path if its current distance is worse than the current best distance, so:

1, 2, 4 = 2, best distance is now 2

1, 3, 5, this now adds to 5 so stop following

1, 2 this adds to 2 so stop following

By considering multiple different possibilities at once and using results of other paths to narrow down the number of possibilities we have arrived at a way of finding the optimal solution to this problem.

Then if we also add an optimisation to so we don’t retrace our steps we then essentially have Dijkstra’s algorithm (kind of).

## Challenge time

Given a set of numbers, give a subset of these numbers such that, they’re still in the same order but some elements are excluded, and that it increases from smallest to largest. For example:

8, 1, 6, 2, 3, 9

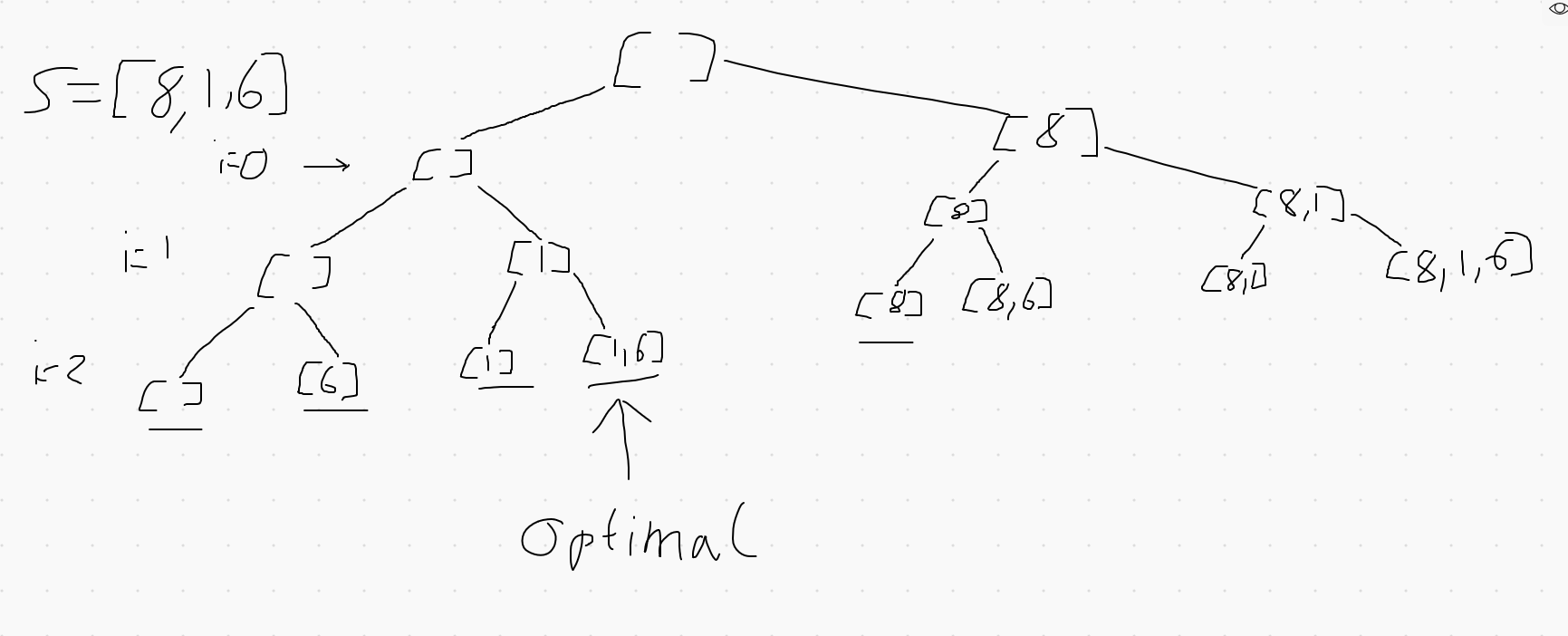
A solution could be 1, 6 or 1, 2, 9 or 1, 2, 3, 9 etc.

The optimal solution will be defined as the longest set. So, for the above problem it would be 1, 2, 3, 9.

## A solution

Using a recursion, we can exclude elements in the array to search all possible subset. This however will take a very long time as for a set of length n there are .

This algorithm works like this:



I only drew this for a set of length 3 because it would take forever for me to draw it, as you can see the leaves of the tree are all the possibilities. As you can see some of the sets generated are not valid so we ignore them, then the valid solution with the longest length is the optimal solution.

Here is some Python code than can be used to this:



Can you come up with a way to optimise this by narrowing down the routes we go down.

Hint: Think which routes can never lead to a valid solution.

\*give them time to think, like 5 mins\*

So they key optimisation I was looking for was the elimination of the routes than generate invalid sets (the ones not in increasing order). We can do this by only adding the next number in the sequence and branching again IF it is more than the number at the end of the “s” array.

The example above now looks like this:

