**Advanced Dynamic Programming**

**Table of Contents**

[Longest Palindromic Subsequence 3](#_Toc63010701)

[Subproblem Definition 3](#_Toc63010702)

[Relating Subproblems 4](#_Toc63010703)

[Base Case 4](#_Toc63010704)

[Solution 4](#_Toc63010705)

[Time Complexity 5](#_Toc63010706)

[Example 5](#_Toc63010707)

[Optimal Binary Search Tree 7](#_Toc63010708)

[Greedy Approach 9](#_Toc63010709)

[Dynamic Programming Approach 11](#_Toc63010710)

[The Floyd-Warshall Algorithm 14](#_Toc63010711)

[Subproblem Definition 15](#_Toc63010712)

[Relating Subproblems 15](#_Toc63010713)

[Base Cases 17](#_Toc63010714)

[Complete Solution 17](#_Toc63010715)

[Time Complexity 17](#_Toc63010716)

[Iterative Solution 18](#_Toc63010717)

[Negative Weight Cycles 18](#_Toc63010718)

[The Alternating Coin Game 19](#_Toc63010719)

[Subproblem Definition 20](#_Toc63010720)

[Relating Subproblems 20](#_Toc63010721)

[Base Case 21](#_Toc63010722)

[Solution 21](#_Toc63010723)

[Time Complexity 21](#_Toc63010724)

## Longest Palindromic Subsequence

To start off, we will go over a problem that we can solve using the basic dynamic programming methods we have previously learnt.

A palindrome is a string of characters that is unchanged when reversed. ‘racecar’, ‘a’, ‘kayak’, ‘wow’, etc. are all examples of palindromes.

Our challenge is to find the longest palindrome that is a subsequence of a given string , of length .

For example, the word ‘deified’ is a palindrome and is a subsequence of the word ‘underqualified’. Similarly, the word ‘rotator’ is a palindrome and is a subsequence of the word ‘turboventilator’. Notice that we must consider the longest palindrome. Every letter in the word is a palindrome. Additionally, ‘rotor’ is a palindrome that is a subsequence of ‘turboventilator’. However, these are not the longest palindromes and thus not what we are looking for.

### Subproblem Definition

We want to divide the main problem into smaller parts, and for each part, we want to find the longest common palindromic subsequence.

where and are indices of the portion of that is being considered, and .

### Relating Subproblems

It is possible that . In this case, they might be part of the longest palindrome. Thus, we can increase the length of the palindrome by , increment and decrement .

Otherwise, we can either decrement or increment , thus ignoring one of the values.

### Base Case

We are essentially just incrementing and decrementing . The base case is when these two are the same. Thus,

We are returning and not because a single character is still a palindrome.

Alternatively, we could end up crossing (say if and are both shifted at the same time). This would lead to invalid positions. Thus,

If , .

### Solution

The solution is to find the longest palindromic subsequence for the entire string.

### Time Complexity

ranges from and also ranges from . We are taking all possible combinations, so there are subproblems.

In each subproblem, the time taken is constant.

Thus, the total time complexity is .

### Example

Consider the string ‘’.

|  |  |  |  |
| --- | --- | --- | --- |
| and do not match. | | | |
|  | and match. | | |
|  |  | is a single character. | |
|  |  | Return . | |
|  | Return | | |
|  | and do not match. | | |
|  |  | and do not match. | |
|  |  |  | is a single character. |
|  |  |  | Return . |
|  |  |  | is a single character. |
|  |  |  | Return . |
|  |  | Return . | |
|  |  | and do not match. | |
|  |  |  | is a single character. |
|  |  |  | Return . |
|  |  |  | is a single character. |
|  |  |  | Return . |
|  |  | Return . | |
|  | Return . | | |
| Return . | | | |

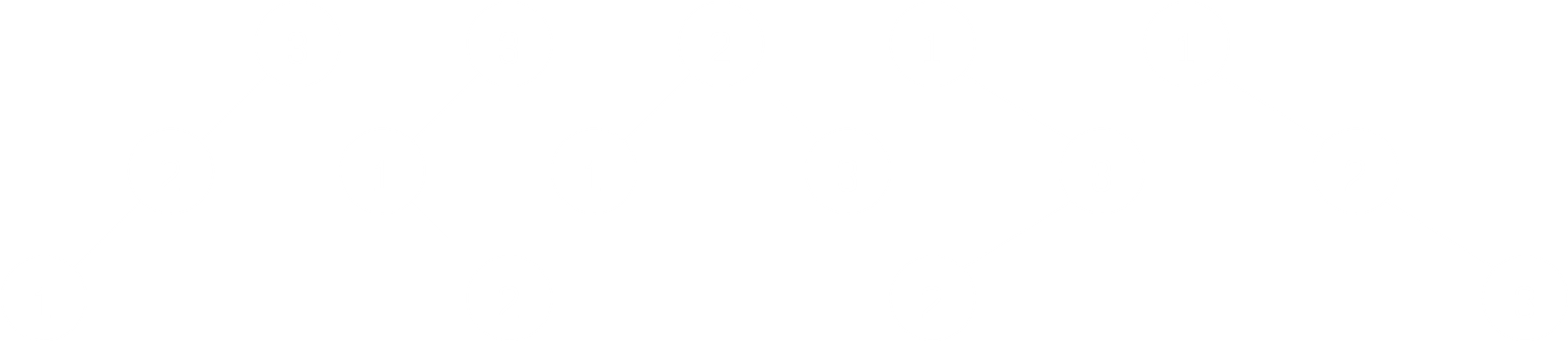
## Optimal Binary Search Tree

Now we will begin diving into more advanced dynamic programming problems, starting with this one.

A binary search tree is a data structure where keys are set up such that for each node, all values smaller than the node value are on the left and all values larger than the node value are on the right.

In this problem, we will be given items, from to , and we have to create a binary search tree (BST) from it. For simplicity, we will consider that , even though this might not be true in actual problems.

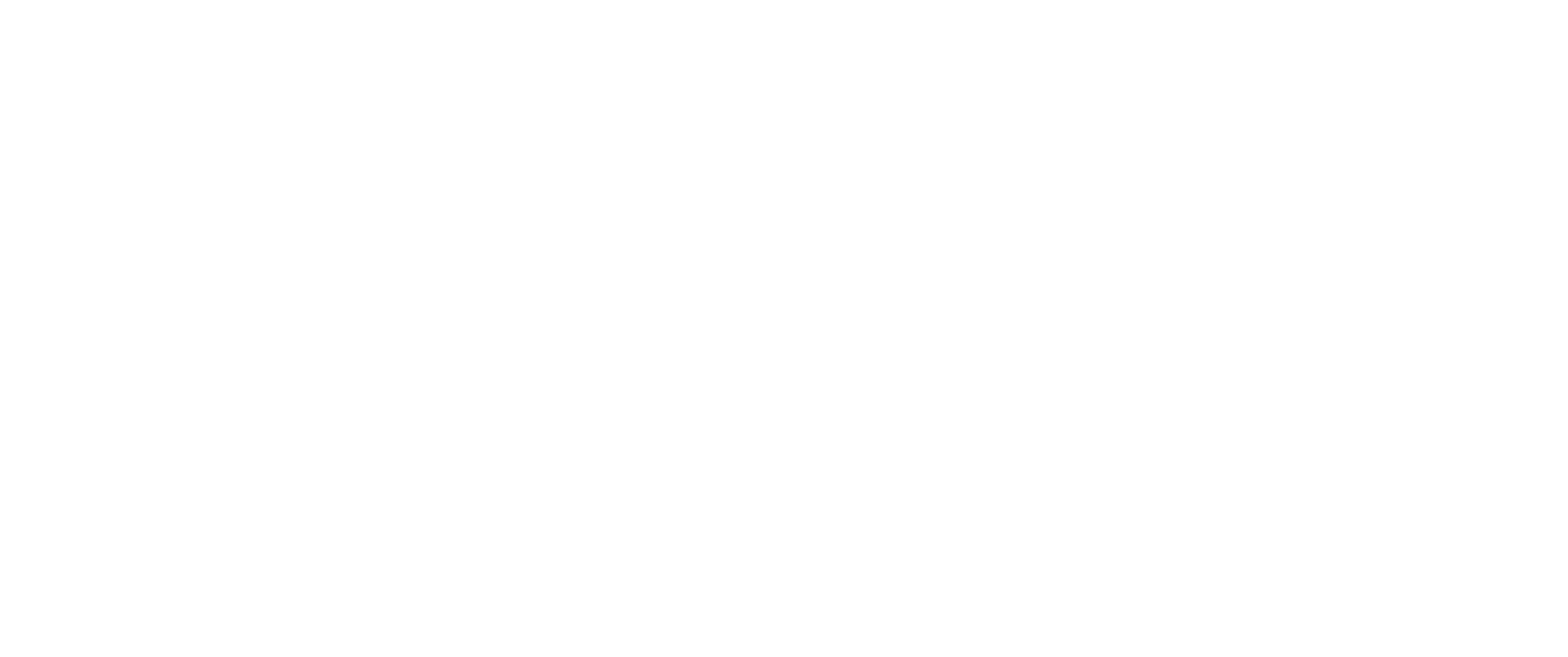
Understand that there are multiple different BSTs possible for any given set of items, depending on the order in which we insert the items. For example, for the values , and , all of the following BSTs are valid.



To resolve the issue of which specific BST we want to create, we will be given a weight, , for each value of . These weights could be used to signify anything. For example, the weight could denote the probability of that item being searched. As such, we want to place items with higher weights higher up in the tree, so as to reduce the cost associated with searching for that item. This restriction will help us decide which BST to use.

Mathematically, we want to find a BST such that the following value is minimized:

For example, say we have two items. A BST for these two items can be setup in two ways.



In the first case, the total weight would be

In the second case,

Depending on the values of and , we will prefer whichever equation gives us the lowest value.

Sadly, the problem becomes exponentially harder the more items we have. For just item, there are possible BSTs. We could try a brute force approach, creating all the trees and taking the one which gives the lowest value, but that would be extremely inefficient.

### Greedy Approach

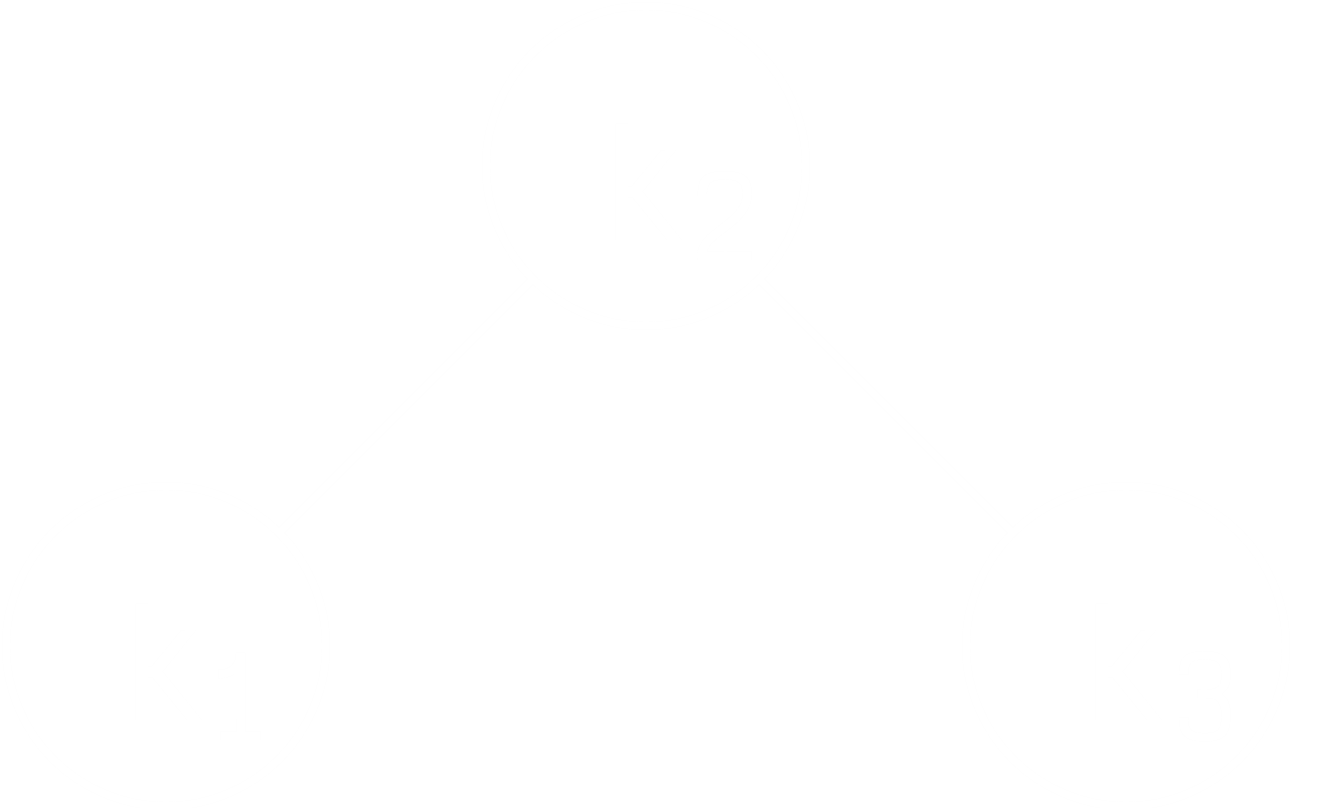
For a set of items from to , we want to find the optimal weight . In order to do this, we must pick an item, , and set it as the root node.

The rule we will follow is the pick the item with the highest as . Thus, all the items with a smaller value than , i.e. the items from to , will go on the left subtree, and all the items with a larger value then , i.e. the items from to , will go on the right subtree.

We now have two smaller subproblems, and . We need to repeat the same process on each of these recursively. Thus, the overall problem is to find .

Example

Say , and have weights , and respectively. Since has the highest weight, it will be set as the root. goes to the left of and goes to the right of .

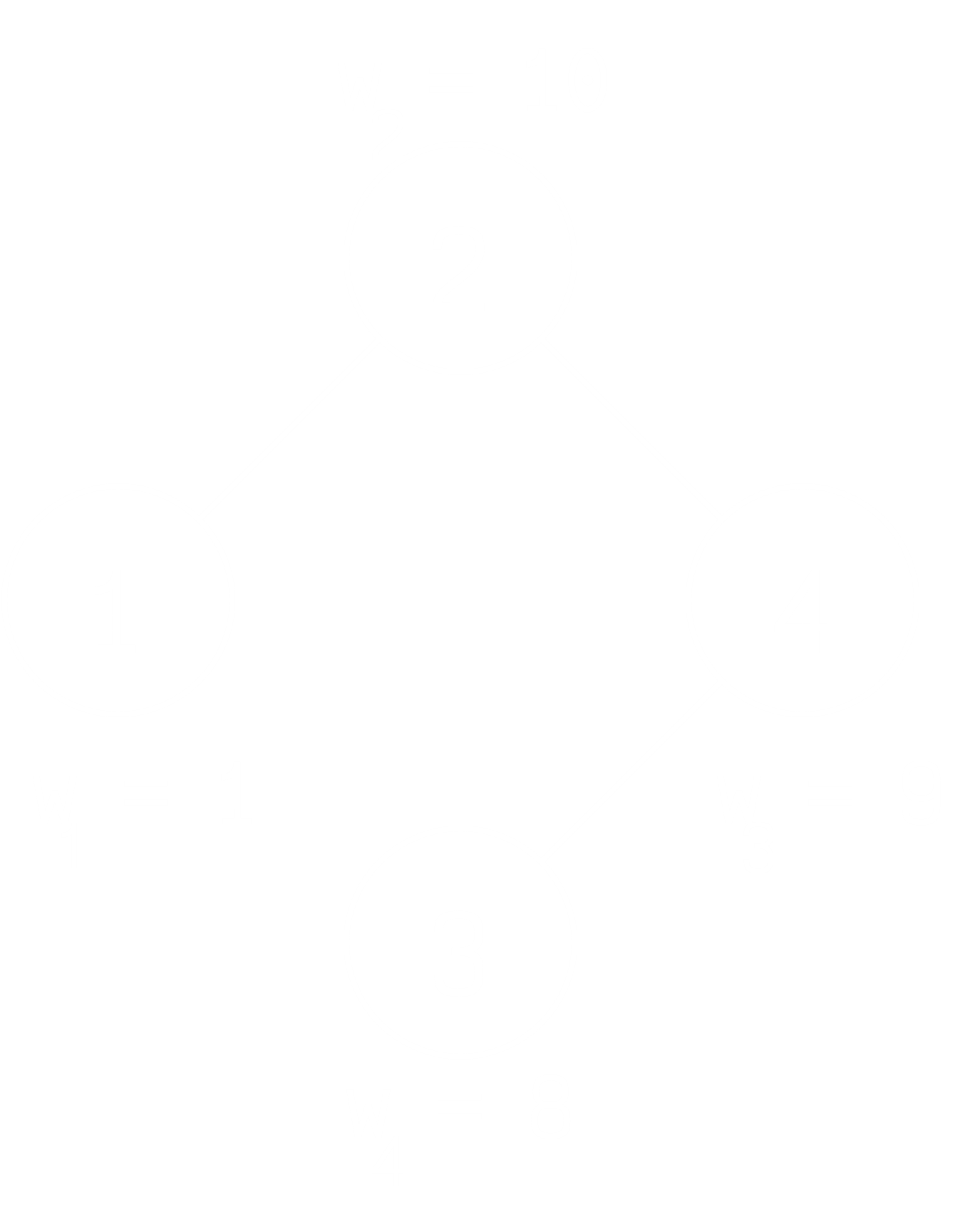


This is the optimal solution for this particular problem.

Counter Example

Say we have four items, , , and with weights , , and respectively.

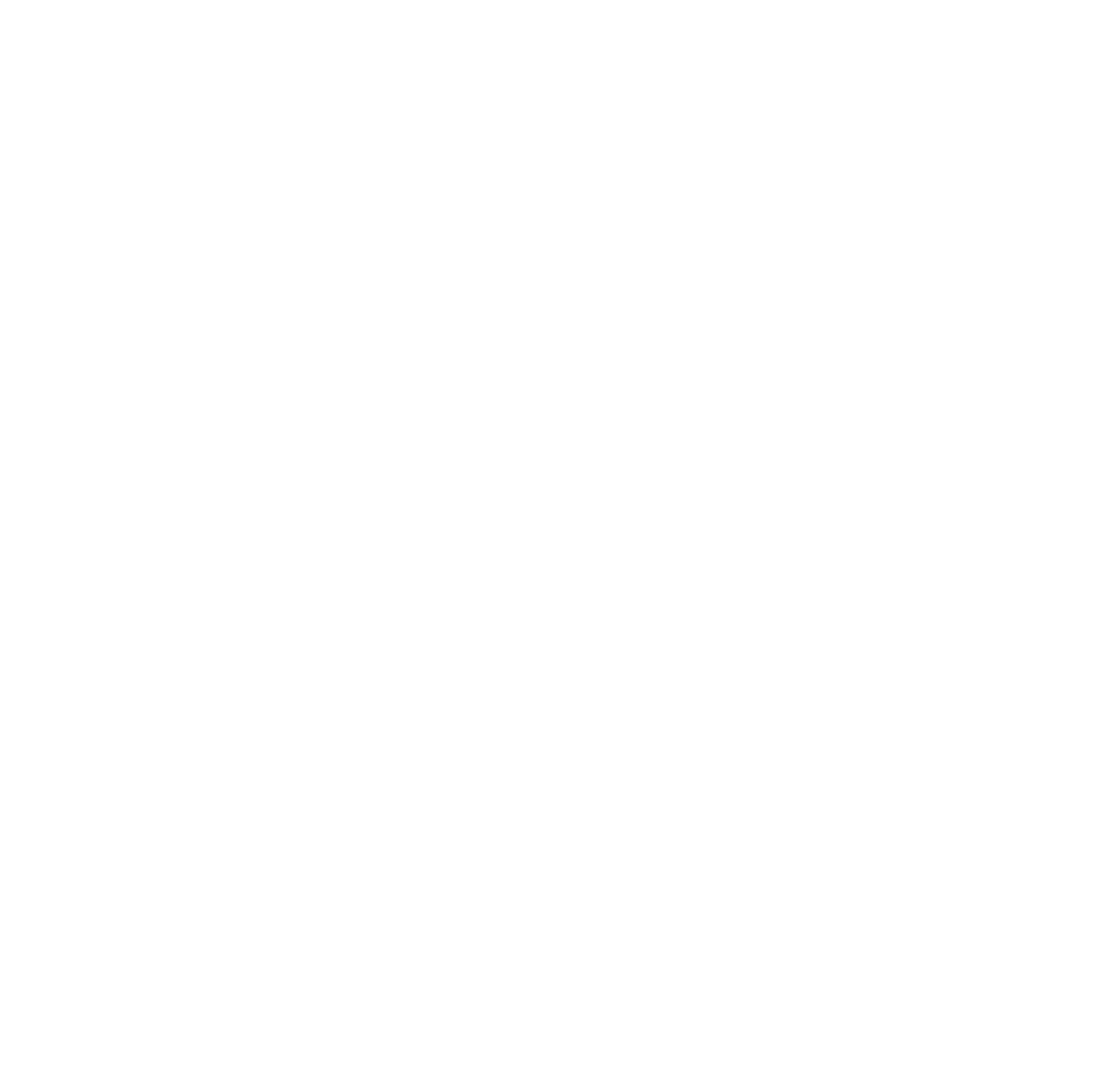
Since has the highest weight, it will be set as the root. will go to the left subtree and and will go to the right subtree. In the right subtree, has the highest weight and will thus be set as the root. will go to the left subtree of . The resulting BST looks like this:



This is very obviously not the optimal solution, seeing as though , which has a higher weight than , is further down in the tree than . Thus, the Greedy approach did not work.

The total cost for the given tree is .

The actual optimal solution is this one:



The total cost here is .

One thing we can take away from this is that the higher weights are towards the top of the tree. However, this does not necessarily mean that the maximum weight is placed at the root.

### Dynamic Programming Approach

#### Subproblem Definition

As stated before, the subproblem is simple , where we try to calculate the optimal cost for a portion of the items, from to .

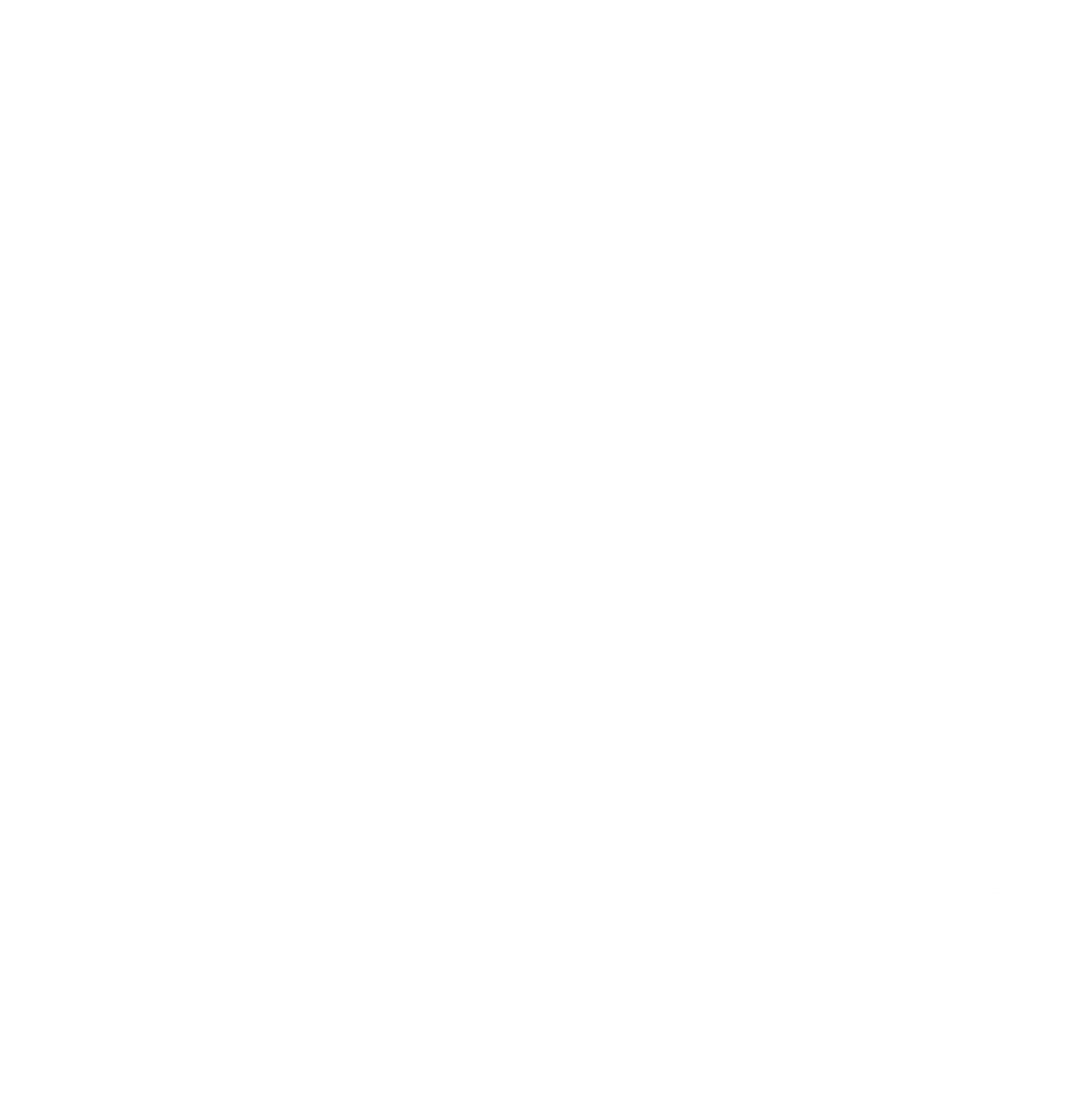
#### Relating Subproblems

Since we do not know which item to pick as the root, we will simply try all of them and try to minimize the cost. Thus,

Essentially, we are taking the sum of the two subproblems created by picking and also adding the cost for . By choosing different items for , we are finding the minimum value.

There is a problem here. Since we need to multiply the weight of whatever item we pick with the depth of current node, we also need to keep track of the depth. This means, we essentially have three parameters, making the problem . Adding another parameter means increasing the number of subproblems and thus making the problem more complex.

To avoid doing this, we will instead take , which is the total weight for all the items from to . The reason we do this is so that all the weights get added multiple times. Consider that this is the optimal BST for some problem.



If we wanted to calculate the total cost here, it would be . The further down the tree a particular item is, the more times it will appear in different subproblems and thus the more times its weight will get added. If we actually try to apply the suggested method here, we will see that the total weight will be like this:

For

For

For

For

Total .

Thus, we can use , which means

#### Base Case

The base case is when . The optimal cost here is the weight of the single item itself.

#### Solution

The solution is simple the result of trying the function call for the whole set, i.e. .

#### Time Complexity

We have two parameters, and , both of which can go from to . Thus, there are subproblems.

In each subproblem, we have a loop going from to . This is in the worst case so each subproblem might have a time complexity of .

This makes the overall time complexity .

## The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm can help us solve the All-Pair Shortest Path problem. This problem states that there is a directed graph, , where each edge has a weight, , a real number. Our goal is to find the shortest path, , for all . We want to find the shortest paths between all possible pairs of vertices.

If we use the algorithms we have learnt previously to try and do this, the possibilities are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Situation** | **Algorithm** | **Time** |  |
| Unweighted | BFS |  |  |
| Non-Negative Weights | Dijkstra |  |  |
| General | Bellman-Ford |  |  |

In every case, the existing time complexity was increased times. For example, the time complexity of BFS was , so running it times makes the complexity . In the worst case, , so the time complexity would be .

Clearly, these are terrible time complexities, especially the one for the general solution where we can have negative weighted edges. We will now look at a dynamic programming solution that can be achieved in time, even for graphs with negative weighted edges.

### Subproblem Definition

We could follow our previous approaches and start with each of the target vertices, but that will lead us to an algorithm. Instead, we will define the subproblems as:

If we number all our vertices from to , then this is the shortest path form to given that a set of vertices from the total vertices is available to us. Thus, we might have a path directly from to , consider of the intermediate vertices, or the path from to consider that there is intermediate vertex that we can use if we choose to and so on.

### Relating Subproblems

For any given set of intermediate vertices, say the last vertex is the -th vertex. We do not know whether or not we need this -th vertex to get the shortest path from to . Thus, we will consider both possibilities.

If we decide that the -th vertex is not in the shortest path from to , then .

If we decide that the -th vertex is in the shortest path from to , then we need to know the shortest path from to and from to .

Thus,

This might be a little difficult to visualize, so consider this.

Say we are considering a set of intermediate vertices that has only one vertex, . Thus, we want to check if , which is a direct path from to , is smaller than , the combination of the direct path from to and from to . Easy enough.

Next, say we have a set of two vertices as intermediate vertices, and . Now, we want to find if , the subproblem we just solved above, gives us a shorter path than . In the latter case, we are going to end up with a subproblem that is similar to the first subproblem we solved.

The recursions are little complex here, so drawing it out might help.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
|  |  | | |
|  |  |  | Direct path weight returned |
|  |  |  | Direct path weights summed and returned |
|  |  | | |
|  |  |  | Direct path weight returned |
|  |  |  | Direct path weights summed and returned |
|  |  |  | |
|  |  |  | Direct path weight returned |
|  |  |  | Direct path weights summed and returned |

In every case, we are only considering smaller subproblems. Thus, the dependency graph is guaranteed to be acyclic.

### Base Cases

The base case will be when , which means there are no intermediate vertices. In this case, the direct edge weight between and will be returned.

### Complete Solution

The complete solution is to find the shortest path for all possible vertices, so this formula

must be run for all possible pairs of and .

### Time Complexity

Since we have vertices, a single combination of and will have subproblems. There are possible combinations of and . Thus, there are subproblems.

Each subproblem only makes a decision, so the time complexity is constant.

Thus, the overall time complexity if .

### Iterative Solution

The iterative solution to this problem uses an adjacency matrix, . It looks like this:

for k=1 to n:  
 for u in V:  
 for v in V:  
 if adj[u][v] > adj[u][k] + adj[k][v]:

adj[u][v] = adj[u][k] + adj[k][v]

PSEUDOCODE

### Negative Weight Cycles

The concept of a negative weight cycle tells us that if we start from some vertex , we will come full circle back to only to find that the path from to has a negative weight. This is, in essence, what causes the infinite loop to start.

Thus, in our adjacency matrix, if there is a negative weight cycle.

## The Alternating Coin Game

In the Alternating Coin Game, we are given a row of coins with values , where is an even number. Two players take alternating turns selecting coins. In each turn, the player can take either the first or the last coin in the row. The coin they take is removed permanently from the row and the player receives points equal to the value of the coin. The player with the most points in the end wins.

The solution to this problem can be found with some trickery. The first player actually does always win if they use an optimal strategy. That strategy is to compare the sum of all the coins in odd positions, , to the sum of all the coins in even positions, . During the game, the player should pick coins only from the given subset.

Say the coins we have are:

The sum of the values of the coins in odd positions is and the sum of the coins in even positions is .

Player goes first. Since they can only pick coins in odd positions, they pick . Player goes next. Say they pick since that has a higher value than . Player again can only pick . Following their old strategy, Player picks . Finally, Player goes again and must pick , leaving for player . Thus, Player got all the odd coins and Player got all the even coins. No matter what Player does, as long as Player sticks to their strategy, Player will be forced to pick the losing set of coins.

Our goal is not actually to find out if Player wins, since we know that, as long as they follow this strategy, they do win. Our goal is to find the maximum amount Player can achieve.

### Subproblem Definition

The subproblems can be divided as , the maximum value that the player can get if it is their turn with the set of coins .

### Relating Subproblems

For the given set, we can take either or . In either case, the range of coins decreases from that direction. We need to consider the maximum value obtained from either case.

On the opponent’s move, we need to assume that they will take away the coin that has the higher value, leaving us with a subset with a lower value.

For , the opponent could pick or . We are assuming the worst, so we will be left with the subset with the minimum value. Thus,

Similarly, for ,

Thus,

Every subproblem only depends on smaller subsets, so there is no possibility of any cycles forming.

### Base Case

Since we are Player , we will not be picking the last coin. Thus, if there is one coin remaining, we do not get its value.

If there are two coins remining, due to how we set up the relations between subproblems, we cannot just use the general equation, since we would go out of bounds. Instead, we need to explicitly say that we will take the coin with the maximum value in this case.

### Solution

The overall solution is to find .

### Time Complexity

Since there are two variables involved, we will have subproblems.

In each subproblem, we are just making a comparison, so the time complexity is constant.

Thus, the overall time complexity is .