**Dynamic Programming**

**Table of Contents**

[Top-Down Approach 4](#_Toc63000524)

[Bottom-Up Approach 6](#_Toc63000525)

[Single Source Shortest Path 7](#_Toc63000526)

[Time Complexity 8](#_Toc63000527)

[Cyclic Graphs 10](#_Toc63000528)

[Time Complexity 12](#_Toc63000529)

[Designing General Solutions 13](#_Toc63000530)

[Defining Subproblems 13](#_Toc63000531)

[Relating Subproblems 14](#_Toc63000532)

[Identifying Bases Cases 14](#_Toc63000533)

[Computing Solutions 14](#_Toc63000534)

[Analysing Time Complexity 15](#_Toc63000535)

[Example 15](#_Toc63000536)

[String or Sequence Problems 18](#_Toc63000537)

[Longest Common Subsequence 18](#_Toc63000538)

[Edit Distance Problems 22](#_Toc63000539)

[Knapsack Problem 25](#_Toc63000540)

[Subset Sums 29](#_Toc63000541)

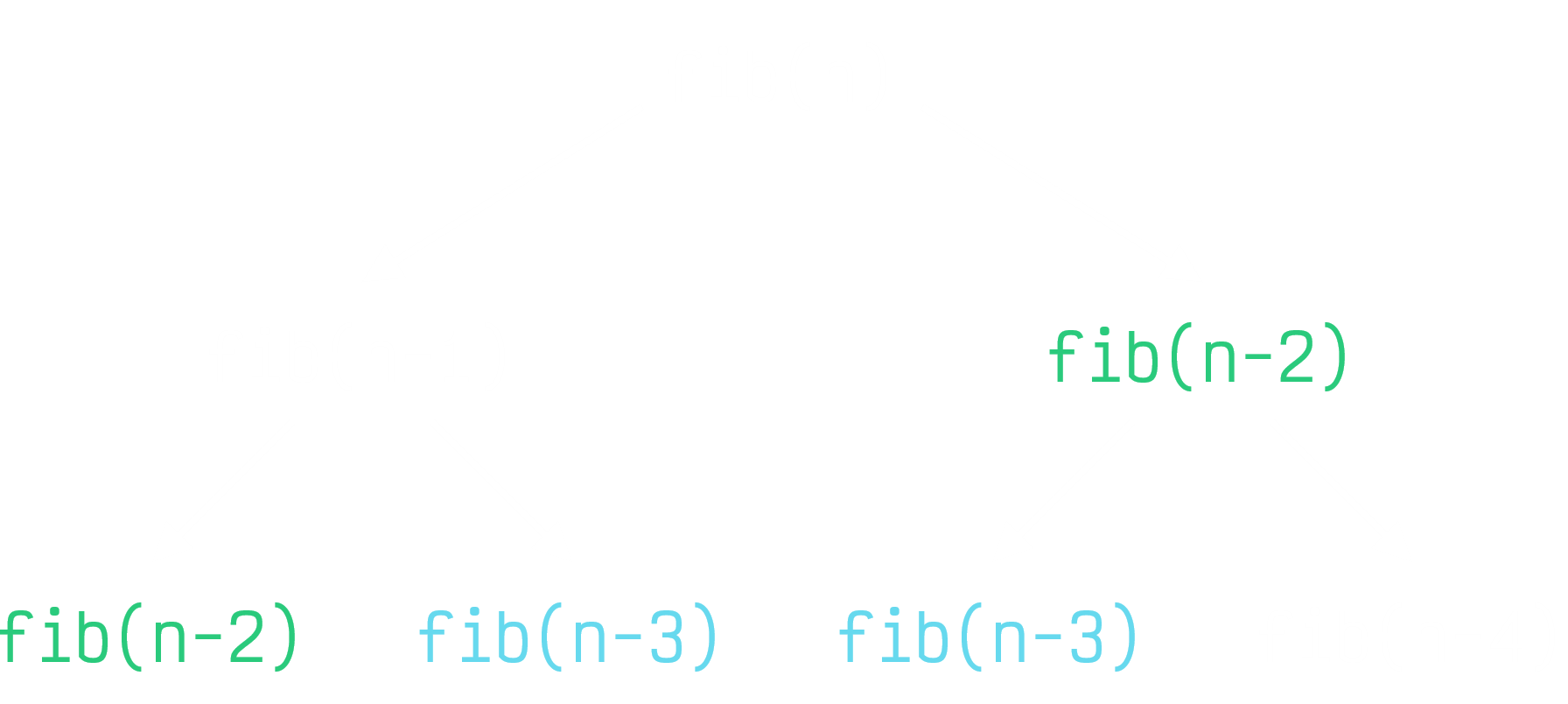
[Sub-Multiset Sum Problem 32](#_Toc63000542)

[Parenthesization 34](#_Toc63000543)

[General Tips for Dynamic Programming 37](#_Toc63000544)

Dynamic programming is not an algorithm in itself, but rather a general algorithm design technique. It is an optimization problem, meaning we have to figure out how to minimize or maximize something. For example, minimizing the distance between two points, as we saw with shortest paths, or maximizing the profits of a process. It is also considered an exhaustive search, sometimes compared to brute force, since we check for all possible answers. Normally, an exhaustive search would require exponential time, but dynamic programming handles this in a clever manner so that it can be accomplished in polynomial time.

In dynamic programming, we divide a large problem in subproblems and then reuse the results of those subproblems. Consider the Fibonacci series. We generally use a recursive approach to solving this problem. To find the sum of the first Fibonacci numbers, we recursively call fib(n-1) and fib(n-2). However, this results in several repeated recursive calls.



The result is an algorithm that has a time complexity of , which is an exponential time complexity.

## Top-Down Approach

If we divide this problem into subproblems, which the above diagram technically already does, we can simply store the value of each subproblem, perhaps in a dictionary using hashing, and use that value for future calls instead of going through the entire process again. For example, we can store the result of fib(1) and fib(2) by computing them the first time they are called. Thus, when fib(3) is called, we can directly retrieve the results of fib(1) and fib(2) without having to recompute them. This is called a memoized approach.

memo = {} // this is a dictionary by the way  
fib(n)  
 if n in memo return memo[n]  
 if n <= 2 f = 1  
 else f = fib(n-1) + fib(n-2)  
 memo[n] = f  
 return f

PSEUDOCODE

In this process, we are having to calculate the value of each of the elements just once, which gives a time complexity of . Any future calls to an element need constant time, since a hash-table is used.

The general steps to a dynamic programming solution are:

1. Start from the top
2. Divide the problem into subproblems
3. Solve the subproblems
4. Memoize the results
5. Reuse the solutions

Thus, dynamic programming is just a combination of recursion and memoization.

The running time of a dynamic programming solution is general given by , where is the total number of subproblems and is the time needed per subproblem. In the above solution, elements gave us subproblems, each costing constant time in and of themselves.

## Bottom-Up Approach

Another approach to solving this problem with dynamic programming is the bottom up approach. Here, the values are calculated using a for loop, starting with the first value. Thus, for each value, we can simply retrieve the values it depends on from the hash-table. It should be easier to see that this approach has a time complexity of .

fib = {}  
for k in range (1, n+1)  
 if k <= 2 f = 1  
 else f = fib[k-1] + fib[k-2]  
 fib[k] = f  
return fib[k]

PSEUDOCODE

A more formal way of thinking about this could be that we found all of the subproblems and created a dependency graph from it. Thus, by performing a topological sort, we can find the order in which we need to work with the subproblems.

This approach can also save us a lot of space in some situations. For example, if we are printing values immediately, we would only need to store the values of the two operations that come immediately before the current one instead of the storing everything. However, in dynamic programming in general, the bottom up approach can be difficult to implement. The recursive approach is generally easier to figure out.

## Single Source Shortest Path

We shall now look at how we can use dynamic programming to solve the shortest path problems we have seen in the last module. Remember that our objective is to calculate .

To solve this, we can just find a naive recursive approach first and then memoize it to make it faster. We will also make use of a tool, guessing. If we do not know the answer, we will guess. In fact, we will try all of the possible guesses and take the best one. This would be computationally costly, but we can do this a little differently to save us the trouble.

Say we are calculating . We need to know which incoming edge is included in this path, but we do not, so we guess that for some node , is included. We do this for all the incoming edges to . If a guess is correct, then . If the guess is incorrect, we simply try a different node. But how do we find ? We do this by performing the same process on the node . Thus, we are falling into a recursive pattern. Since the process is recursive, the time complexity will be exponential.

Coming to memoizaiton, we can simply store the shortest path to a node once we have found it to use the concept of memoization.

Notice how there is a dependency graph here. To find the shortest path to any node, we first had to find the shortest path to the nodes it depended on. Thus, the concept of DAGs will be useful. Make note of the fact that this is a directed **acyclic** graph. We shall get into what happens if there are cycles in a bit.

### Time Complexity

We know that, to calculate time complexity, we have to find the number of subproblems and the time taken per subproblem.

In this case, the subproblems are the shortest paths to each of the nodes in the shortest path to the target vertex. Thus, there are subproblems for the nodes.

Now to consider the time for each subproblem. At each node, we had to consider the shortest paths for all the incoming edges. This is also called the indegree of a vertex. Thus, there is a time complexity of here. Finding the minimum value from all the incoming edges takes constant time though, since we can use a variable to keep track of the minimum value. Thus, the total time for a subproblem is given by .

However, this is not equal for all of the subproblems, which means we cannot simply find the product of time per subproblem and the number of subproblems to calculate the total time complexity. Instead, we have to find the sum of the time taken for all the subproblems. Thus, the total time complexity is given by

Of course, the total number of indegrees for all the vertices is just the total number of edges, thus

Thus, the time complexity is .

Notice that this is the same as the time complexity we got when we performed topological sort on the graph and then relaxed all the edges, aka Dijkstra’s algorithm. We are essentially doing the same thing here, but in the opposite direction.

## Cyclic Graphs

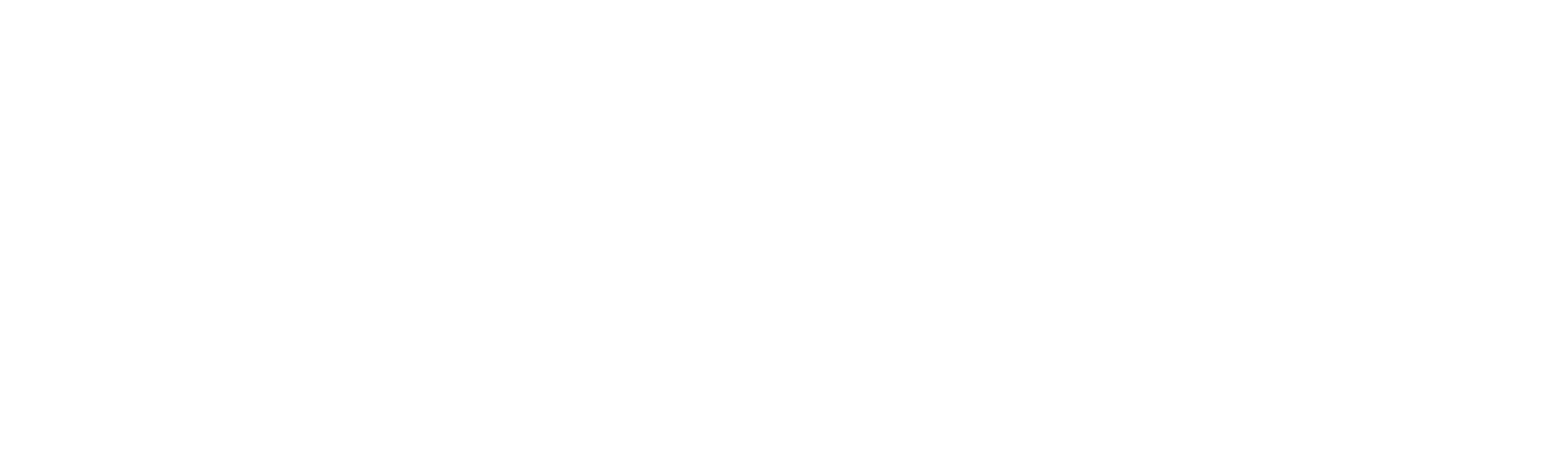
Now to deal with those bloody cycles. Firstly, if there is a negative cycle, the shortest path is not even defined, so let’s avoid those things for now. Even positive cycles are a problem though.



Say we want to find . To do that, we have to find . To do that we need to find , for which we need two things, and . We have come full circle. That is bad because our algorithm is just going to try to find the value of again and fall into an infinite loop.

In order to solve this problem, we need to make our graph acyclic.

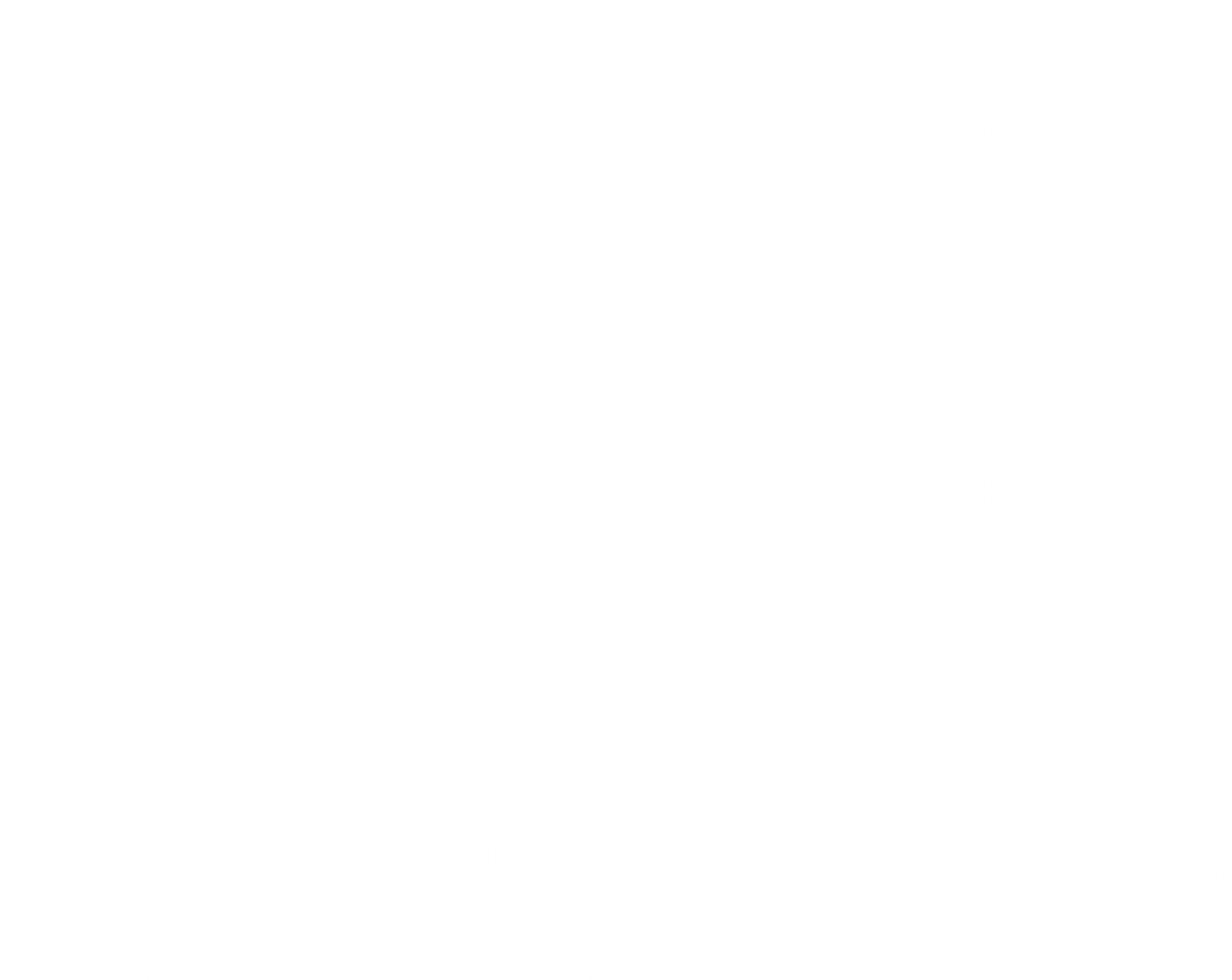
Say we have this graph.



While studying the Bellman-Ford algorithm, we discovered that no matter what our graph is, the shortest simple path will always have edges. To solve this problem, the first thing we do is make copies of the graph. The graph is divided into layers.



Next, for every edge in the original graph, we create that edge in our layered graph, but each node makes the edge with the required node in the next layer.



This graph tells us how many edges we need using the number of layers. If we want to go from to , we need edges, . Similarly, if we want to travel from to , then we need edge, . Thus, we can simply go to and run any of the shortest path algorithms to find the shortest path to any node.

More specifically, we are finding , where is the maximum number of edges between and . For example, means we want to get from to . There is an edge from to , so this is possible. If instead we wanted , this would mean going from to , which is not possible.

Using this approach, going back to dynamic programming, we want to find .

In the last layer, .

### Time Complexity

In this scenario, we have increased the number of subproblems. Now we have subproblems to deal with, since ranges from to , and each layer has subproblems. The rest is the same as for the original acyclic version. Thus, for each layer, we have a time complexity , and for the complete problem, the time complexity is . You may be wondering where the term went. That is being ignored since is much smaller than , since in the worst case, .

Again, notice that this is exactly the same time complexity as in Bellman-Ford. We actually did the same thing in Bellman-Ford, just in the opposite direction. This is the idea from where Bellman-Ford was developed. Bellman-Ford was the bottom-up approach version of this same algorithm. In fact, our algorithm can even detect negative weight cycles if we modify it a bit, take layers up to in the acyclic version and checking for changes in the last layer.

## Designing General Solutions

Now we shall begin to study how to design our approach for a dynamic programming problem. The five steps to this design are:

1. Define the subproblems
2. Find the relationship between the subproblems
3. Identify the base cases, which have trivial independent solutions
4. Compute the final solution using the subproblems
5. Analyse the time complexity of the designed algorithm

### Defining Subproblems

The process of defining the subproblems includes describing what the subproblem actually is, including its parameters. A subproblem will most likely involve a subset of the original input, such as a prefix or a suffix, or it might record a partial state, the results of which we have to add up. Look back at the problems we have already solved and notice that the subproblems there were essentially just this, smaller portions of the complete problem which we can combine to get the final result.

One important thing we need to keep in mind is that the subproblems must overlap. This means that we must be using the same subproblem at different points in our algorithm. Otherwise, if we are only encountering each subproblem just once, the memoization process becomes useless.

Thus, the two must have features of subproblems are:

* The subproblems must combine to form the final solution
* The subproblems must overlap

### Relating Subproblems

In the process of relating the subproblems, the first thing we need to do is to figure out which subproblems relate to which ones. Essentially, we must find the topological order of the subproblems. If this is not entirely possible, we need to guess the order, somewhat like we did with our shortest path solutions.

The second step is to ensure that the dependencies we found are acyclic. If a dynamic programming problem involves a cyclic graph, it cannot be solved. If the dependencies are not acyclic, we need to change the graph and make it acyclic.

The third step is to form a DAG.

### Identifying Bases Cases

Bases cases are simply solutions for subproblems that are independent. For the Fibonacci series, and were base cases. For the shortest path problems, and for all where and has no incoming edges were base cases.

### Computing Solutions

In the computation process, we use either the top-down or the bottom-up approach. In order to store solutions to subproblems, we can use parent pointers, which we will study later on. For example, in the shortest path problems, we know that the parent pointers point to the parent vertex of a node in the shortest path for that node. Thus, we can use this pointer to get the shortest path of the parent.

### Analysing Time Complexity

We have already seen how to find the total time complexity. If the time taken for each subproblem is the same, the total time is the product of the number of subproblems and the time taken per subproblem. If it is not the same, the total time is the sum of the time taken in each subproblem.

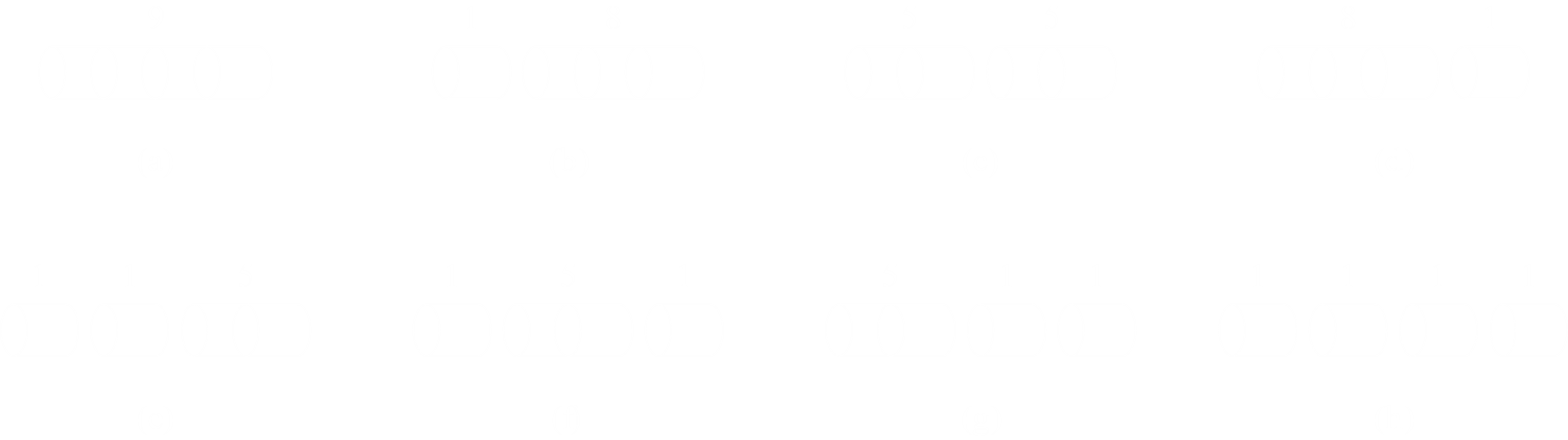
### Example

To illustrate the dynamic programming approach, let us consider the rod cutting problem. Here, we are given a rod of length , which we need to cut into pieces. Each piece will have a price associated with it, where is the length of the piece. Thus, . We need to cut the rod to maximize our profit.

Say we have a rod of length , with the following prices for pieces of different lengths.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

Thus, the profit for the different cuts are:



Obviously, the maximum profit in this case is earnt by dividing the rod into two parts of length each, earning a profit of .

For a rod of length , we can make cuts. For each of the cuts, we can make the cut or not make it. Thus, we have positions and for each position we can either make the cut or not make it, meaning we have choices. Thus, we have choices.

Let us divide the problem into subproblems like this. Let be the maximum profit that can be earnt by cutting a rod of length .

Next, the relationships between the subproblems. Here, the left-most cut, the first one we make in the original rod, has some length. The length of this cut is . Essentially, is the length of the left-most cut, and we are finding the price of that length, and then finding the maximum price for whatever part is remaining of the total length. We are doing this for different values of and taking the maximum profit we can find.

In the next step, we need to ensure that the relationships we have set up do not form a cycle. We know that only depends on smaller , thus it cannot form a cycle at any point.

Next, the base case. This is just , since a rod of length will get profit.

And finally, the solution. Using the process we have created, we can find the profit for recursively. If we reach , we can just return . For each value of , once we find the maximum possible profit for a rod of length , we can store the value. We can also store parent pointers in a fashion. Since, for a rod of length , the optimal first-cut is of length , we can store a pointer to the maximum profit for a rod of length .

Here, we have subproblems. For each subproblem, we need to check the values of different lengths form to . Thus, the time for each subproblem is different and we need to sum them up. For a subproblem with length , we need to check sub-length. For a subproblem with length , we need to check sub-lengths. Omitting repetitions, the main problem requires sub-lengths, giving an approximate time complexity of .

These rest of the notes can get a little confusing. Concentrate on understanding what happens in the example given. That should help to understand what is going on.

## String or Sequence Problems

Dynamic programming problems often involve strings or sequences along with some problem that we need to solve. Such problems could involve:

* Prefixes, where each of the subproblems is a prefix of the original string or sequence, , where is the ending index of the prefix
* Suffixes, where each of the subproblems is a suffix of the original string or sequence, , where is the starting index of the suffix and is the last index of the string or sequence
* Substrings, where each of the subproblems is a part of the string somewhere in between, , where and are both valid indices for the string or sequence

### Longest Common Subsequence

A subsequence is a string created by taking specific characters from the original string in the order they are given, while skipping over the other characters. For example, in the string ‘abc’, each character could have two states, it is either taken, or not taken. Thus, we have a total of subsequences, ‘’, ‘a’, ‘b’, ‘c’, ‘ab’, ‘bc’, ‘ac’ and ‘abc’.

Our challenge is to find the longest common subsequence (LCS), given two strings. Say the strings are , ‘TAGTCACG’, and , ‘AGACTGTC’. Here, the longest common subsequence is ‘AGACG’, which takes the letters ‘TAGTCACG’ from and ‘AGACTGTC’ from . Since a string can have sub-sequences, simply generating the sub-sequences of both strings and searching for the longest common one will take exponential time.

#### Defining Subproblems

Let’s break this problem down. Since this is a string problem, the subproblems could involve either prefixes, suffixes or substrings. Let’s start by considering that the subproblems involve prefixes. denotes the prefixes and from and respectively. For some prefix length, our goal is to find the LCS of the two prefixes of and of that length. These will be our subproblems. Thus, we start from the first character, find the LCS, then take the first two characters, find the LCS, then take the first three characters, find the LCS and so on. Thus, the LCS of any prefix depends on us first finding the LCS of the smaller prefixes.

#### Defining Subproblem Relations

Say we have some prefix where we already know the longest common subsequence. Now we increase the prefix length by one character and try to find the LCS. If the last character, the -th character, of the new prefix from matches the last character, the -th character, of the new prefix from , then we can just add this character to the LCS and keep going.

However, if the -th and -th characters of the prefixes of and do not match, then we can have two scenarios. It is possible that the -th character from the prefix of matches one of the previous, unmatched characters in the prefix of , thus contributing to the subsequence. It is also possible that the -th character from the prefix of matches one of the previous, unmatched characters in the prefix of , thus contributing to the subsequence. We need to try both of these scenarios to find the maximum possible length.

Thus, for , if , then , i.e. the LCS length of the prefixes that are one character shorter, plus . Otherwise, , i.e. we find the LCS of the prefix from and the prefix from without the -th character, and find the LCS of the prefix from and the prefix from without the -th character and take whichever value is larger.

These cover all possible cases for subproblems.

The value of only depends on smaller values of and , thus it is guaranteed that this scenario is acyclic, meaning a topological order can be found.

#### Identifying the Base Cases

The base case is for prefixes of length , which give an LCS of ‘’.

#### Solution

We can use prefixes to find as shown above. If a particular character matches, both and are decreased, and we add that character to the LCS. We can also use this to store parent pointers.

#### Time Complexity

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | |
|  |  | | | |
|  |  |  | | |
|  |  |  | Since one of the indices is , this function call returns . | |
|  |  |  | | |
|  |  | | | |
|  |  | | | |
|  |  |  | | |
|  |  |  |  | |
|  |  |  |  | This value was memoized earlier.  is returned. |
|  |  |  |  | Since one of the indices is , this function call returns . |
|  |  |  |  | |
|  |  |  | Since one of the indices is , this function call returns . | |
|  |  |  | | |
|  |  | | | |
|  | | | | |

Here, the number of subproblems is . For each subproblem, a few conditions are checked, which means the time complexity of each subproblem is . Thus, the overall time complexity is .

#### Example

\_TAG \_AGA

### Edit Distance Problems

In edit distance problems, we are given two strings, and , and we have to convert to . We do this by modifying using single character edit operations.

Say we have two strings, ATCG and AAGC.

For each character, we can:

* Replace the character – We could for example, replace the T in the first string with an A. Thus, the first string becomes AACG.
* Remove a character – We could delete the in the first string to make it AAG.
* Insert a character between two characters – We could add a C between the G and the end of the string in the first string to make it AAGC.

Our challenge is to find the smallest number of such edits we need to make to convert the string. In the above example, we need at least edits. We do not necessarily need these particular edits in this particular order though.

Edit distances are used to suggest corrections for typos by finding the strings with the least edit distances from what was typed and suggesting them.

#### Defining Subproblems

Say our subproblems are to find the minimum edit distance for any prefix for the strings given. Say is the minimum number of edits required to convert to .

#### Relating Subproblems

Again, either the -th and -th characters of the prefixes of and respectively match, or they do not. If they match, we keep them and if they do not, we must edit them. However, we do not know which method we need to use to edit them, so we try all three.

The three options are:

* Remove and find since the -th character does not exist anymore
* Insert and find , since the characters at the indices and definitely match
* Replace with and consider since the characters at the indices and definitely match

Thus, for , if , then , i.e. the number of edits does not increase. Otherwise, , i.e. we find the edit distances caused as a result of each of the three methods being used, and take whichever value is smaller. is added in this case since there was an edit for as well.

Since each call to only depends on smaller values of and , there are no cycles and dynamic programming can be used.

#### Identifying the Base Cases

The base case is when or is called. Here, the value returned will be or , since either characters must be deleted or characters must be inserted.

#### Solution

The overall solution to find involves finding the edit distances for all the prefixes as shown before. In this case as well, we can choose to store parent pointers of the prefixes.

#### Time Complexity

There are subproblems, each taking constant time. Thus, the overall time complexity is .

### Knapsack Problem

Say we have a knapsack that can hold a total weight of , and items, each with weight and a value, . The value indicates the importance of the item. Our challenge is to choose a subset of items such that the total value is maximized, while not crossing the total weight limit.

For example, if we have a knapsack that can hold a weight of and items with weights , , and and values , and , the subset with the last items would give us the best results, with the total value and total weight .

#### Defining Subproblems

Let’s try this problem using suffixes. For each item, we can choose to either take the first items or not, and then consider the problem for the remaining items. An additional constraint we need to keep track of is the remaining space in the knapsack, .

Thus, we can define the subproblems as , in which we attempt to maximize the value of the first items, while is the remaining space in the knapsack.

#### Relating Subproblems

For each item, we either take the item or leave it. If it is possible to take the item without crossing the value of , then we can take it. This would increase the total value of items we have taken, but decrease the value of . Alternatively, we could simply leave the item and move on to the next item.

Thus, , given that .

Notice that strictly depends only on larger values of and smaller values of . As such, the dependency graph would be acyclic, meaning dynamic programming can be applied.

#### Identification of Base Case

Since we have items, we can only keep considering subproblems from the -th item to the -th item. Thus, the base case is .

#### Complete Solution

We start from , and solve subproblems as discussed above. We can also use parent pointers to keep track of which subproblem came from which.

#### Time Complexity

We have items and possible values for . Thus, the total number of possible subproblems is . Each subproblem takes constant time to make some comparisons. Thus, the total time complexity is .

If , the time complexity is polynomial. In the absolute worst case, the value of is the input size, the maximum amount the machine can hold, and the value of also depends on the machine.

Say we are using a 32-bit machine, making the maximum possible value of . Even for , the total number of subproblems becomes . If we double the value of , there is not much difference, but if we double the value of , we have a huge increase. In fact, the increase is exponential. Thus, the worst-case time complexity is not dependent on , but rather on , which is limited by the word size of the machine, unlike , which is limited by memory size. If the word size if , the maximum value of . As such, the solution has a time complexity of .

The time complexity for this solution is not polynomial, since the increase, as we saw, is exponential, but it is not exactly exponential either, since the increase is dependent on , not . As such, this solution is said to have a pseudo polynomial time complexity. Pseudo polynomial complexities are not dependant on an input value, but rather on some other number.

Note that the pseudo polynomial run time does not appear because we used a suffix approach. The same would happen with a prefix approach.

#### Example

Let’s use the example we saw at the beginning here.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
|  |  | | |
|  |  |  | |
|  |  |  |  |
|  |  |  |  |
|  |  |  | |
|  |  | | |
|  |  | | |
|  |  |  | |
|  |  |  |  |
|  |  |  |  |
|  |  |  | |
|  |  |  | |
|  |  |  |  |
|  |  |  |  |
|  |  |  | |
|  |  | | |
|  | | | |

Notice that none of the function calls were repeated. It may seem like memoization would be pointless here. However, if we have a larger number of items, then the repeated function calls would start to appear. A small example was chosen since a larger one would become very complicated to show.

### Subset Sums

Given a list, , of positive integers, our challenge is to find out if there is a subset , such that , where is a given target value.

For example, say and . Thus, .

This problem is in contrast to the Knapsack problem, in that we are having to meeting a target value exactly. It is not enough to simply stay below this value, as we did in the Knapsack problem. Also, there are no additional values we need to consider.

#### Defining Subproblems

Again, we will consider suffixes of items. For each element, we can either pick that element or leave it. If we pick it, we have to meet a smaller sum using the suffixes. If we do not pick, we have to meet the same sum using suffixes.

Thus, is a subproblem where we try to find out if there is a subset , where is the length of , such that .

#### Relating Subproblems

At the -th element, we can either take the element or leave it. If we pick it, the next function call is , given . If we do not pick it, then the next function call is .

The difference from previous problems is that we are not trying to maximize or minimize any value. Instead, we are trying to choose one of the values. Thus,

Keep in mind that we are simply trying to return a true or false value, nothing more.

Since depends only on larger values of or smaller values of , there is no concern with the dependency graph being acyclic.

#### Identifying the Base Case

The base case here is when . Thus, true. Alternative, false, since we have run out of items and the target has still not been met.

#### Complete Solution

The complete solution depends on finding recursively as discussed above.

#### Time Complexity

Again, we have two parameters, meaning the number of subproblems is , since we have to go from to and to respectively, the last check being the base cases. The work done per subproblem is constant, since we only perform a comparison in each subproblem. As such, the time complexity is . Again, we have a pseudo polynomial complexity.

#### Example

Let’s work with the example we saw at the beginning.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
|  |  | | | | |
|  |  |  | | | |
|  |  |  |  | | |
|  |  |  |  |  | |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  | |
|  |  |  |  | | |
|  |  |  |  | | |
|  |  |  |  |  | |
|  |  |  |  | | |
|  |  |  | | | |
|  |  | | | | |
|  | | | | | |

### Sub-Multiset Sum Problem

In sub-multiset sum problems, we are given an array of positive integers, and each of the integers can be reused an infinite number of times. Our challenge is to reach a given target sum, , using the minimum number of integers possible. We will also be keeping track of how many times the -th integer was used, .

Thus, we want to find , while minimizing .

For example, if and , a possible solution is , thus using elements in total.

#### Defining the Subproblems

In subset sum, we took a part of the sum and tried to get to that sum using the suffixes. Here, we can use the same approach, except that we can take each element multiple times. Since the goal is to minimize the number of items of each denomination taken,

#### Relating the Subproblems

For each item , with the value , we can take it times. The next step is to decrease the target by . However, we need to ensure that . The minimum number of coins we can take is, of course, . Thus,

Since only depends on larger values of and smaller values of , there is no possibility of any cycles forming.

#### Base Case

There are two possible bases cases. One is if the target has been reached. Thus, . Secondly, if all types of coins have been used and the target has not been reached. Thus, , to ensure this combination is not considered.

#### Complete Solution

Using the methods outlined above, our target is to find .

#### Time Complexity

For numbers, the value of ranges from to , meaning we have as the maximum depth of recursion. In the worst case, each of those numbers is , meaning the value of decreases by for each call of . Thus, the total number of subproblems .

Each subproblem has a for loop running, which runs times.

Thus, the total time complexity is .

### Parenthesization

In this problem, we are given an associative expression and we have to find the optimal evaluation order.

An associative expression is one in which the elements can be grouped in any manner that we want. For example, . Thus, we have to find out which grouping pattern to create.

However, we cannot reorder the items, similar to how matrix multiplication works. In fact, that is exactly where parenthesization would be useful. Knowing which matrices to multiply in which order would allow us to do the least amount of work.

For example, say we want to multiply three matrices, an matrix, followed by an matrix, followed by an matrix. Multiplying the first two matrices first forces us to perform multiplications and gives us an matrix. After that, we must multiply this matrix with a matrix, which is another multiplications. However, if we had multiplied the second and third matrices first, we would have had to perform multiplications, giving us a matrix, and then multiply the matrix with this matrix, which is another multiplications. Thus, we had to do much less work in the second case.

#### Defining the Subproblem

We can use the thinking of substrings here as well. We will group the multiplications off in a prefix part and a suffix part. First, we make an outermost division into two groups, with element from to in the prefix group and elements from to in the suffix group.

We shall repeatedly make this division for different values of form to .

Next, we shall make two divisions like this for the both the prefix and the suffix group and then two divisions for all four of those groups and so on. Thus, we shall get the minimum value for all possible partitions.

Since we are taking divisions of the original string and not just prefixes and suffixes, this problem is classified as a substring problem.

Here, .

#### Relating Subproblems

Firstly, we are going to look into every partition. For each of those partitions, we shall divide it into smaller partitions and look into each of those possible partitions.

At every partition, we need to keep in mind that the cost of the two partitions created must also be considered. This means that multiplying the two matrices we created from to and from to also has a cost that we must take into consideration.

Thus, .

Here, larger substrings are only dependant on smaller ones. Thus, there are no cycles.

#### Base Case

The base case is when we have a single matrix. There is no cost of multiplication here.

Thus, .

#### Complete Solution

is the complete solution which can be found from the above methods.

#### Time Complexity

The value of can range from to , as can the value of . Thus, we have subproblems. At each subproblem, we have a for loop running over elements. Thus, the total time complexity is .

## General Tips for Dynamic Programming

The most difficult part of solving a dynamic programming problem is deciding what subproblem to use and how to relate them. Some methods to making subproblem choices are:

* For inputs that are sequences, prefixes and suffixes can be used. Which one to use is generally about personal preference. If we are seeming to need both at the same time, we most likely have a substring problem on our hands. Substring problems have many more subproblems, which makes the algorithm slower, but they do work.
* For inputs that are multiple sequences, it is best to consider subproblems made of prefixes, suffixes or substrings of each.
* For inputs that are single integers, subproblems are usually smaller integers.

If none of these seem to work:

* Try to come up with the recursive pattern first, like we did with the Fibonacci series at the beginning of this module.
* Another possibility is that the subproblems we are defining do not actually capture everything we need to consider. For example, in the knapsack problem, we had to consider the current element being traversed and the remaining size of the knapsack. Leaving the size of the knapsack out of consideration will cause the subproblems to be incomplete and not work properly.
* Maybe the dependency graph has a cycle. Try the graph duplication trick.

About creating relations, we are essentially having to find a feature in the problem that allows us to create smaller subproblems that contributes to the overall solution. For example, using prefixes, if we are at the -th element, we can find the minimum value up till the -th element.

The next step is to try all possible versions of the feature we have chosen using a loop and take the best solution among them.

And finally, and most importantly, remember to memoize.