Dynamic Programming

UF Programming Team

What is dynamic programming?

Let's look at an example

What is $2^5 + 5! + 100$?

$\rightarrow 252$

Now imagine I write a '$+\ 1$' to the end of the previous expression

What is $2^5 + 5! + 100 + 1$?

$\rightarrow 253$

How were you able to calculate that expression so quickly?

What is dynamic programming? (cont.)

You didn’t need to recalculate the entire expression because you ***memorized*** the value of the partial expression to use for a later time

That’s exactly what ***dynamic programming*** is!

To put it simply, dynamic programming is <u>remembering stuff to save some time later</u>

What is dynamic programming? (cont.)

A more "mathematical" definition

Dynamic programming is a method for solving complex problems by breaking it down into a collection of simpler subproblems

Dynamic programming problems exhibit two properties:

1) ***Overlapping subproblems***
2) ***Optimal substructure***

What does all of this mean? Let's look at some examples

Fibonacci Sequence

As you might have seen before, the Fibonacci sequence is one in which a term in the sequence is determined by the sum of the two terms before it

→ 1, 1, 2, 3, 5, 8, 13, 21, 34, …

We can write it as $f_n = f_{n\text{-}1} + f_{n\text{-}2}$ where $f_0 = 1$ and $f_1 = 1$

Let's say we wanted to write a recursive function for this sequence to determine the ***n***th Fibonacci number

Fibonacci Sequence (cont.)

```
public int fib( int n ) {
    if ( n == 0 || n == 1 )
        return 1;
    return fib( n - 1 ) + fib( n - 2 );
}
```

Fibonacci Sequence (cont.)

What will be the function calls if we wanted to determine `fib( 5 )`?

$\rightarrow$ `fib( 5 )`

$\rightarrow$ `fib( 4 ) + fib( 3 )`

$\rightarrow$ `fib( 3 ) + fib( 2 ) + fib( 3 )`

$\rightarrow$ `fib( 2 ) + fib( 1 ) + fib( 2 ) + fib( 3 )`

$\rightarrow$ `fib( 1 ) + fib( 0 ) + 1 + fib( 2 ) + fib( 3 )`

$\rightarrow$ `1 + 1 + 1 + fib( 2 ) + fib( 3 )`

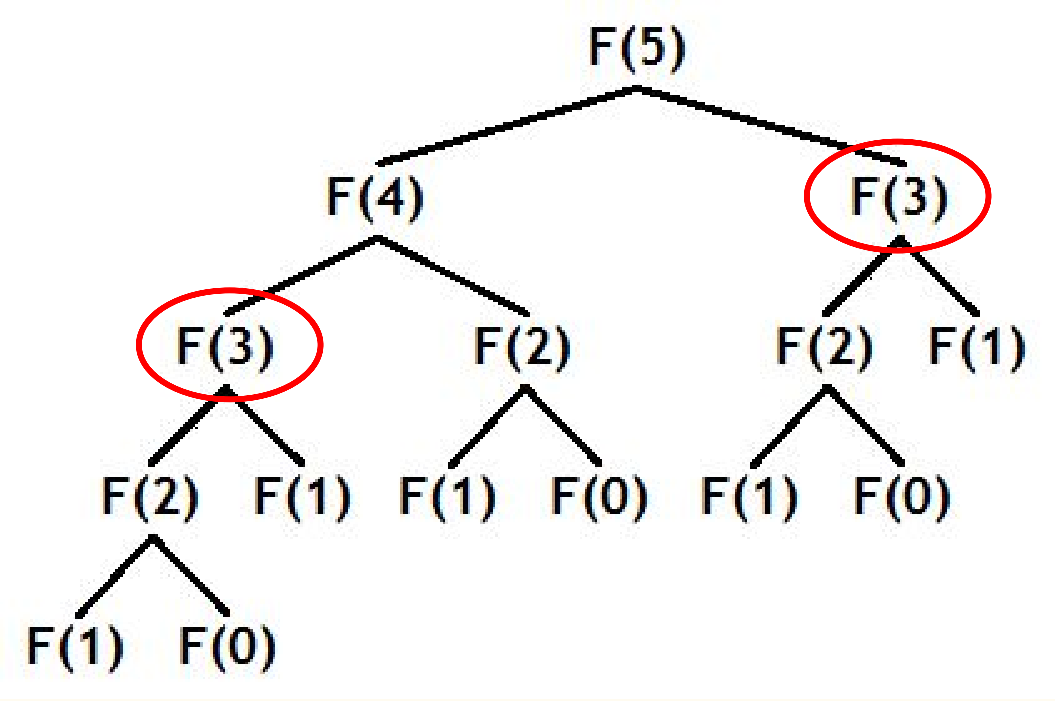
Is it necessary to compute `fib( 2 )` and `fib( 3 )`?

Fibonacci Sequence (cont.)

This tree represents the function calls for the function `fib( 5 )`

Notice that when we calculate `fib( 4 )`, we also calculate the value of `fib( 3 )`

When we call `fib( 5 )`, it calls `fib( 3 )`, but if we already have the value of that function call from before, there's no need to recompute it!



Fibonacci Sequence (cont.)

Since `fib( 5 )` asks us to call the function `fib( 3 )` *multiple* times, we have found an ***overlapping subproblem*** - a subproblem that is reused numerous times, but gives the same result each time it's solved

Rather than solving these overlapping subproblems every time, we can do the following:

→ solve the subproblem once and save it's value

→ when you need to solve the subproblem *again*, rather than going through all the computation to do the solving, return the ***memorized*** value

Fibonacci Sequence (cont.)

```
int[] dp = new int[ SIZE ]; // all values initialized to 0

public int fib( int n ) {

    if ( n == 1 || n == 0)

        return 1;

    if ( dp[ n ] != 0 )

        return dp[ n ];

    return dp[ n ] = fib( n - 1 ) + fib( n - 2 );

}
```

Knapsack Problem

Condensed Problem Statement

You have a knapsack that can hold up to a certain weight $\boldsymbol{W}$, and you have $\boldsymbol{N}$ items that you can pack for a trip; however, all of these items might not fit in your knapsack

Each item has a weight $\boldsymbol{w_i}$ and a value $\boldsymbol{v_i}$

Your goal is to pack some items into your bag such that the total weight of all the items doesn't exceed your knapsack's weight limit $\boldsymbol{W}$, and the sum of all the values of the items in your knapsack is ***maximized***

Knapsack Problem (cont.)

What are some approaches we can take to solve this problem?

→ Calculate all subsets of items and see which subset has the highest value, yet stays under the knapsack's weight limit

→ Create some ratio for each item v_i / w_i and take the items with the highest ratio until you run out of items

→ Dynamic programming!

Before we go over how to solve this problem using dynamic programming, let's see why the other solutions don't work

Solving Knapsack: Take I

Subsets Solution

Let's calculate all subsets of the $\boldsymbol{N}$ items that we have

How many subsets are there?

$\rightarrow 2^{\boldsymbol{N}}$

So we need to go through all $2^{\boldsymbol{N}}$ subsets….but if $\boldsymbol{N}$ is *big**, this is going to take a while

**big* in this case can be as small as 20, since $2^{20} \sim 1{,}000{,}000$

Solving Knapsack: Take II

Ratio Solution

Imagine we have a knapsack with $\boldsymbol{W} = 6$, and are given the following set of items

$\boldsymbol{w} = \{2, 2, 2, 5\}$

$\boldsymbol{v} = \{7, 7, 7, 20\}$

Let's find the $\boldsymbol{v}$ / $\boldsymbol{w}$ ratio of each item

$\rightarrow \boldsymbol{v} / \boldsymbol{w} = \{3.5, 3.5, 3.5, 4\}$

Now that we have the ratios, let's take items with the highest ratio until we no longer have any space left

Solving Knapsack: Take II

We will take the item with $\boldsymbol{w} = 5$, $\boldsymbol{v} = 20$ first and put it into our knapsack

We now have $\boldsymbol{6 - 5 = 1}$ weight left in our knapsack

Since we only have $\boldsymbol{1}$ weight left, we cannot fit any of the other items in our knapsack (since each of their weights are $\boldsymbol{2}$), so we end up with a value of 20!

But….if we would've taken the other three items and left the last item behind, we would have a value of 21 instead $(7 + 7 + 7 = 21)$

$\boldsymbol{w} = \{2, 2, 2, 5\}$

$\boldsymbol{v} = \{7, 7, 7, 20\}$

$\boldsymbol{v} \,/\, \boldsymbol{w} = \{3.5, 3.5, 3.5, 4\}$

Solving Knapsack: Take III

We determined that the greedy approach for knapsack does not work, so now we will try to use dynamic programming to solve this problem; let's try and create a ***recursive function*** that can get us the answer!

Let's start by iterating over each item we have, starting from item ***0*** and going to item ***N-1***

We will also be keeping track of how much weight we have left in the knapsack as we go through the items

Note: when we're on an item, we only have two options: ***take*** or ***ignore***

→ 0/1 Knapsack name derivation

Solving Knapsack: Take III

```
public int solve( int index, int remainingWeight ) {
    // Determines the max value you can get when starting at
    // the item at index with remainingWeight left to use
}
```

Imagine that the function has an implementation that gives us the answer we want; what would be the values of `index` and `remainingWeight` on our initial call so that we get the answer for the ***entire*** problem?

→ **`solve( 0, W )`**

Solving Knapsack: Take III

Let's try and figure out some ***base cases*** for this function (***e.g.***, in the function `fib()`, the base cases were `n == 0` and `n == 1`)

What happens if `remainingWeight == 0`?

→ the max value we get by starting at any index with ***no weight left*** will be 0!

What happens if `index == N`?

→ the max value we get by starting at an index ***outside of our item range*** will be 0!

Great, we have some base cases to work with now!

Solving Knapsack: Take III

Let's look at every other case where `index != N` and `remainingWeight > 0`

What happens if `weight[ index ] > remainingWeight?`

Well, if the weight of the item we are looking at is ***greater*** than the amount of weight we have left, there's no way we can take this item, so we ***ignore*** it!

But what do we return? `0`? `Integer.MAX_VALUE`? A different function call?

→ **`solve( index + 1, remainingWeight )`**

→ the max value we get by starting at an index ***where we can't take the item*** will be determined by the max value we get by starting at the ***next*** item with the ***same*** remaining weight to use

Solving Knapsack: Take III

Last case: what happens if `weight[ index ] <= remainingWeight`?

Remember, we only have two options: ***take*** **or** ***ignore***

We know what ignoring an item looks like:

→ **`solve( index + 1, remainingWeight )`**

But what does ***taking*** an item look like?

Well, we know that `remainingWeight` will be different

→ we took the item, so we need to subtract out its weight from the remaining weight

Solving Knapsack: Take III

We also know that since we ***took*** the item, we can add `value[ index ]` to our result

Our resulting function call for ***taking*** an item would be

→ **`value[ index ] + solve( index + 1, remainingWeight - weight[ index ] )`**

→ the max value we get by starting at an index ***where we take the item*** can be determined by the sum of

→ the value of the item, and

→ the max value we get by starting at the ***next*** item with the new remaining weight

Solving Knapsack: Take III

Wait, we just saw two separate function calls for the case where we have enough weight to take an item

→ `solve( index + 1, remainingWeight )`

→ `value[ index ] + solve( index + 1, remainingWeight - weight[ index ] )`

So which one do we use? Which one are we returning? Which one will get us the max value?

```
→ max( solve( index + 1, remainingWeight ) ,
       value[ index ] + solve( index + 1, remainingWeight - weight[ index ] )
```

Solving Knapsack: Take III

We now know our solution handles every case for Knapsack, giving us the optimal solution when we call `solve( 0, W )`, and it shows the problem has an ***optimal substructure*** - that is, an optimal solution can be constructed from the optimal solutions of its subproblems

If we solve any subproblem, `solve( i, j )`, we are guaranteed to get the optimal answer, and we can use these optimal subproblems to solve ***other*** subproblems

We know that the problem has optimal substructure, but does it have any ***overlapping subproblems***?

Solving Knapsack: Take III

Let's look at an example

$\boldsymbol{v} = \{100, 70, 50, 10\}$

$\boldsymbol{w} = \{10, 4, 6, 12\}$

$\boldsymbol{W} = 12$

Imagine that we take the first item and then ignore the next two

$\rightarrow$ we will be at `solve( index = 3, remainingWeight = 2 )`

Imagine that we ignore the first item and then take the next two

$\rightarrow$ we will be at `solve( index = 3, remainingWeight = 2 )`

Solving Knapsack: Take III

We have now determined we have ***overlapping subproblems***, so we can save the value of a specific `index/remainingWeight` pair so we don't have to recompute the solution to that subproblem each time!

```
int[][] dp = new int[ N ][ W ]

                    …

if ( dp[ index ][ remainingWeight ] != 0 )

    return dp[ index ][ remainingWeight ]
```

Solving Knapsack: Take III

We can now code a solution to the Knapsack Problem!

Break up into groups of 2-3 and work on the following problem

spoj.com/problems/KNAPSACK

Other problems to work on

→ spoj.com/problems/COINS

→ spoj.com/problems/PARTY

→ spoj.com/problems/ROCK