Competitive Programming and Mathematics Society

Programming Workshop 3

Intro to DP

Programming Team

What is Dynamic Programming?

- A way to solve a complex problem by breaking it down into smaller "subproblems".
- Let’s take a look at an example to visualise the key concepts of DP!

Fibonacci Numbers

The **Fibonacci Sequence** is the series of numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- The first two numbers are defined as 0 & 1
- From then on, each number is the sum of the previous two numbers

Let's say we want to come up with a function fib(n) that computes the nth Fibonacci number.
We can break the problem up into simpler subproblems!
First we know some base cases:

- fib(0) = 0
- fib(1) = 1

For $n > 1$, we can break down the problem into two simpler subproblems using the formula:

- fib(n) = fib(n-1) + fib(n-2)

Fibonacci Numbers

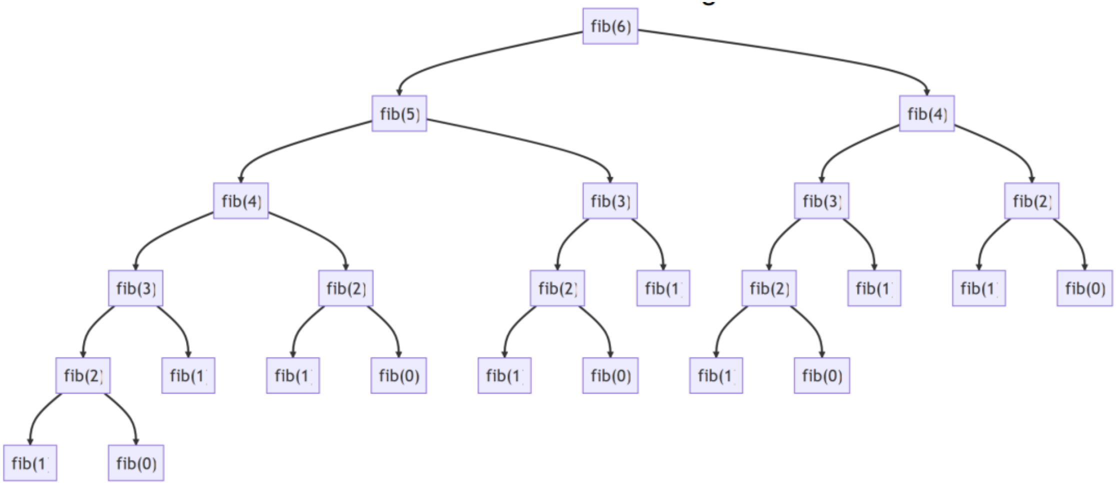
```
    int fib(int n) {
        // Base cases
        if (n == 0) {
            return 0;
        }
        if (n == 1) {
            return 1;
        }

        // Recursion
        return fib(n - 1) + fib(n - 2);
    }
```

This gets us the right answer.
However, there is one big issue with this implementation! (Hint: think about how many times this function is being called for a given n)

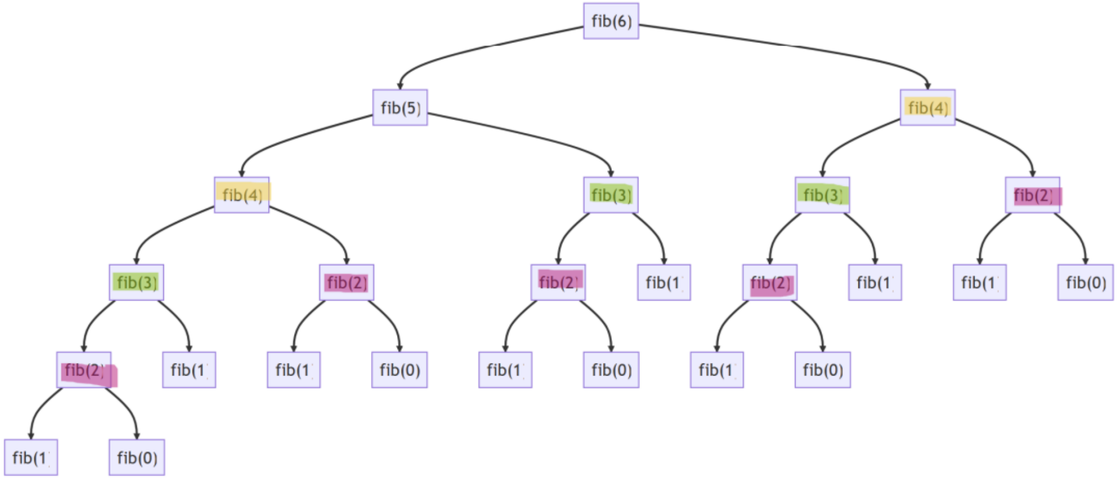
Fibonacci Numbers

Let’s see what happens when we call fib(6). Can we see a way to speed up this function?



Fibonacci Numbers

We can see we have a lot of overlapping subproblems.



Fibonacci Numbers

We see that:

- fib(4) is called 2 times
- fib(3) is called 3 times
- fib(2) is called 5 times
- ...

In general, in dynamic programming, we only want to solve each subproblem once, otherwise we are doing lots of repeated work.

What that means is we want to remember the answer to a subproblem when we solve it for the first time. From then on, if we need the answer to the subproblem again, we can just look up the answer that we got previously. This is called "memoisation" or "caching".

Fibonacci Numbers

We can store our answers in an array, say cache[]. We can initialise the array with a value that can never occur, say -1.

From then on, for fib(n), we first check if cache[n] is -1.

- If it is not, it means we already know the answer to fib(n) and return it.
- If it is, we will have to solve the subproblem and then store it in the array.

Fibonacci Numbers

```
int fib(int n) {
    // Base Cases
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }

    // Check if we have solved the subproblem before
    if (cache[n] != -1) {
        return cache[n];
    }

    // Solve subproblem and store in cache
    cache[n] = fib(n - 1) + fib(n - 2);
    return cache[n];
}
```

Fibonacci Numbers

Let's compare the **time complexities** of these two algorithms.

- In our original algorithm, for every function call, the function was called two more times (e.g. fib(5) calls fib(4) and fib(3)). Hence the complexity is $\mathcal{O}(2^n)$ which is *very slow*.
- In our new algorithm, we only solve each subproblem once (e.g. fib(0), fib(1), fib(2), ...). This means that our complexity is just $\mathcal{O}(n)$ as we have n subproblems. Much quicker!

DP Approach

With that example, let’s look at the main steps to solve a DP problem.

1. Define subproblems.
2. Formulate a recurrence that relates the subproblems.
3. Recognise and solve the original problem.
4. Define the base cases.

The Frog Problem

There are N stones, numbered from 1 to N. For each index i $(1 \leq i \leq N)$, the height of the stone i is h_i. Initially, our frog friend is sitting on the first stone and they will continuously perform a series of actions as follows:

- If the frog is sitting on stone i, it can jump to stone $i + 1$ or $i + 2$. The cost of jumping will be $|h_i - h_j|$ where j is the stone the frog jumps to.

Help our friend find the minimum cost to jump from the first stone to the N^{th} stone.

Input and Output

Input

- The first line of input contains a positive integer N $(2 \leq N \leq 10^5)$, the number of stones.
- The second line consists of N integers h_i $(1 \leq i \leq N, 1 \leq h_i \leq 10^4)$, the height of stone i.

Output

- Output a single integer, the minimum cost to jump from the first stone to the N^{th} stone.

Sample Input:

- 6
- 30 10 60 10 60 50

Sample Output:

- 40

Explanation: if we follow the path $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$, we incur a minimum total cost of $|30 - 60| + |60 - 60| + |60 - 50| = 40$.

Our thought process

1. What are our base cases? (ie. the trivial subproblems)

Our thought process

1. What are our base cases? (ie. the trivial subproblems)
   - Clearly our lowest cost to reach stone 1 is 0, and our lowest cost to reach stone 2 is $|h_2 - h_1|$.

Our thought process

1. What are our base cases? (ie. the trivial subproblems)
    - Clearly our lowest cost to reach stone 1 is 0, and our lowest cost to reach stone 2 is $|h_2 - h_1|$.
2. Now what about stone 3?

Our thought process

1. What are our base cases? (ie. the trivial subproblems)
   - Clearly our lowest cost to reach stone 1 is 0, and our lowest cost to reach stone 2 is $|h_2 - h_1|$.
2. Now what about stone 3?
   - We can either come from stone 1 for a cost of $|h_3 - h_1| + min\_cost(1)$ or from stone 2 for a cost of $|h_3 - h_2| + min\_cost(2)$. We obviously want the lower of these costs

Our thought process

1. What are our base cases? (ie. the trivial subproblems)
   - Clearly our lowest cost to reach stone 1 is 0, and our lowest cost to reach stone 2 is $|h_2 - h_1|$.
2. Now what about stone 3?
   - We can either come from stone 1 for a cost of $|h_3 - h_1| + min\_cost(1)$ or from stone 2 for a cost of $|h_3 - h_2| + min\_cost(2)$. We obviously want the lower of these costs
   - Generalising our idea for any stone $i \geq 2$, we have

$$min\_cost(i) = min(min\_cost(i-1) + |h_i - h_{i-1}|, min\_cost(i-2) + |h_i - h_{i-2}|)$$

Our thought process

1. What are our base cases? (ie. the trivial subproblems)
   - Clearly our lowest cost to reach stone 1 is 0, and our lowest cost to reach stone 2 is $|h_2 - h_1|$.
2. Now what about stone 3?
   - We can either come from stone 1 for a cost of $|h_3 - h_1| + min\_cost(1)$ or from stone 2 for a cost of $|h_3 - h_2| + min\_cost(2)$. We obviously want the lower of these costs
   - Generalising our idea for any stone $i \geq 2$, we have

$$min\_cost(i) = min(min\_cost(i-1) + |h_i - h_{i-1}|, min\_cost(i-2) + |h_i - h_{i-2}|)$$

3. We keep building up our minimum costs for each stone until we reach our N^{th} stone.
   - Notice how we're solving our problem by building on overlapping subproblems which is what DP is all about!

Our thought process

1. What are our base cases? (ie. the trivial subproblems)
   - Clearly our lowest cost to reach stone 1 is 0, and our lowest cost to reach stone 2 is $|h_2 - h_1|$.
2. Now what about stone 3?
   - We can either come from stone 1 for a cost of $|h_3 - h_1| + min\_cost(1)$ or from stone 2 for a cost of $|h_3 - h_2| + min\_cost(2)$. We obviously want the lower of these costs
   - Generalising our idea for any stone $i \geq 2$, we have

$$min\_cost(i) = min(min\_cost(i-1) + |h_i - h_{i-1}|, min\_cost(i-2) + |h_i - h_{i-2}|)$$

3. We keep building up our minimum costs for each stone until we reach our N^{th} stone.
   - Notice how we're solving our problem by building on overlapping subproblems which is what DP is all about!

Now that we have a plan, it's your turn to implement! When you're ready, test your solutions below for *Frog 1*, and let us know if you have any questions!
https://vjudge.net/contest/618826#overview

Our Sample Solution

```cpp
    int main(void) {
        int stone_heights[MAX_SIZE];
        long long min_costs[MAX_SIZE];
        int N;

        // scan in input:
        cin >> N;
        for (int i = 0; i < N; i++) {
            cin >> stone_heights[i];
        }

        // handle base cases (first and second stones):
        min_costs[0] = 0;
        min_costs[1] = abs(stone_heights[1] - stone_heights[0]);
```

Our Solution

```
    // find shortest cost to current stone by building on
    // two previous stones (subproblems)
    for (int i = 2; i < N; i++) {
        long long two_cost = min_costs[i - 2] +
        abs(stone_heights[i] - stone_heights[i - 2]);
        long long one_cost = min_costs[i - 1] +
        abs(stone_heights[i] - stone_heights[i - 1]);

        min_costs[i] = min(two_cost, one_cost);
    }

    cout << min_costs[N - 1] << endl;

    return 0;
}
```

The 0/1 Knapsack Problem

You are a thief carrying a single knapsack with limited capacity. The museum you broke into had artifact that you could steal. Unfortunately you might not be able to steal all the artifact because of your limited knapsack capacity. Each artifact has a weight and a value. You want to not have the weight exceed the maximum capacity you can put in your bag, but still take home the maximum value of items.

Input range:

- N, number of items <= 100
- W, maximum weight capacity of your bag <= 1000

Fractional knapsack?

Let's take a quick look first at fractional knapsack, albeit just minor changes to the problem statement, creates a completely different version of the knapsack problem. Say you had a bag with max capacity = 10. And you broke into a bank with

- Gold fractions that has total weight = 7, value = 14
- Silver fractions that has total that is weight = 5, value = 9
- Bronze fractions that has total weight = 4, value = 6

The only difference between fractional and 0/1 knapsack is the item now becomes divisible, which means now its weight and value are uniform throughout the item and you can take units of it.

Any ideas for how we might solve this (simpler) version of the problem?

Fractional knapsack - Solution

The problem now becomes very simple! We can keep taking units of items that have the highest value per unit weight.

So in this case, we'll take all 7 gold pieces with a value of 2 per unit weight, and then 3 units of silver fractions with a value of 1.8 per unit weight. Now we've clearly filled up our knapsack full of the largest value items.

However, does this approach work for the 0/1 knapsack? If we tried to use the same strategy of taking the highest value/weight ratio first i.e gold, then we'd take all 7 pieces. But, then we don't have any space left since we need to take everything now in its entirety. Thus we'd get a max value of 14 when in fact we could've taken the silver and bronze with weight 9 and value 15.

We need a different approach entirely!

Naive solution to 0/1 knapsack

If we were able to find all the possible subsets of items that we can take and just keep a maximum of the value of the subsets which total weight is <= W. This would work!

But unfortunately for N items, there are 2^N subsets. Which means that for our problem when N could be as large as 100, we'd be facing over 10^{30} operations! This is a bit large...

Let's see if we can try to use DP for this optimization problem?

What values are we storing?

- What is the value that we are looking for, so what should we store in our state?
- How many dimensions do we need?

What values are we storing?

- What is the value that we are looking for, so what should we store in our state?
- How many and dimensions do we need?

Let Knapsack(N, W) be the maximum value we can fit into our sack if we are limited to W weight and only consider the first n items.

Why might this be a DP problem?

- **Sub-problems:** Let's denote the problem of finding the maximum value we can receive given N items and W bag capacity as Knapsack(N,W). Now if we took a subset of the N items which has size N' and reduced the capacity of the bag from W to W', we can try to solve the same type of knapsack problem Knapsack(N', W') on what is now a smaller version of the original problem. Thus, this problem satisfies the property of having sub-problems.

Finding sub-problem relationships

If we had our original problem Knapsack(N,W) and we now wanted to reduce the problem down to some sub-problem with a smaller N and W value, what can we do? Let's say that we were wondering whether we should put item N into our knapsack or not.

If we chose not to pick up the N^{th} item, then we'd now have only N-1 items to consider, and our bag still has weight capacity W, so one sub-problem that we can get to is Knapsack(N-1,W)

If we did choose to pick up the N^{th} item, then we'd still have N-1 items left to consider, but our bag will now have weight capacity of only W - weight[N] available. So we transition to the sub-problem Knapsack(N-1,W-weight[N]). The key thing to not forget is by picking up item N, we received value[N].

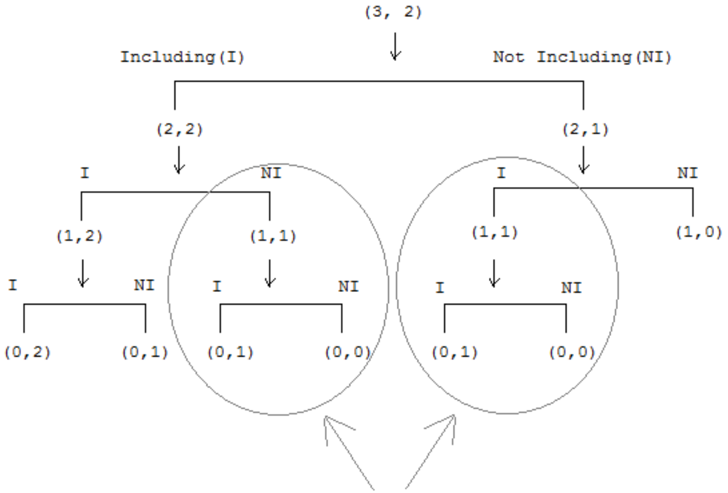
Transition formula

$$\text{KS}(N, W) = \max\left(\text{KS}(N-1, W),\ \text{KS}(N-1, W - \text{weight}[N]) + \text{value}[N]\right)$$

In this formula: KS(N, W) is the same as Knapsack(N, W) which refers to the maximum value we receive from optimally picking from the first N items given we have a bag with max capacity W
The formula calculates the maximum value that can be obtained by choosing either:

1. Not including the Nth item, which would result in the value obtained from the knapsack with $N-1$ items and the same capacity W.
2. Including the Nth item, which would result in the value obtained from the knapsack with $N-1$ items and the remaining capacity $W-\text{weight}[i]$, plus the value of the Nth item value$[i]$.

Overlapping sub-problems



Complete Recursive Calculation is repeated twice for same input (1,1).

Putting together a solution

What are the DP states?

in KS(N, W) we store the maximum value we can fit into our sack if we are limited to W total weight and only consider the first n items.

How do we construct the value for one state using previous ones?

$$\text{KS}(N, W) = \max\left(\text{KS}(N-1, W),\ \text{KS}(N-1, W - \text{weight}[N]) + \text{value}[N]\right)$$

Either taking or not taking the Nth item, and taking whichever one gives a higher value.

Whenever we compute a value, store so it can be re value.

Base case?

When we we're on 0 items, or when there is no more space, we cannot add anything into our sack (return 0).

Code

```
int knapsack(int n,int m){
    if(n <= 0) return 0; //no more items
    if(m == 0) return 0; //no more space

    if(memo[n][m] != -1){ //already calculated
        return memo[n][m];
    }

    //max result achieved from not taking this item
    int yes_take = 0, no_take = max(0, knapsack(n-1,m));

    if(weight[n] <= m){ //enough space to carry this item
        yes_take = value[n] + knapsack(n-1, m-weight[n]);
    }
    memo[n][m] = max(yes_take, no_take);
    return memo[n][m];
}
```

Attendance form :D

Further events

Please join us for:

- Further Dynamic Programming (same time next Wednesday!)
- *UNSFW* Competition (weeklong starting Monday Week 8)
- Mathematics: Invariants and Methods of Counting (Tuesday Week 8)
- Citadel Programming Collaboration (Thursday Week 9)

If you have any feedback for today's workshop!