Complexity and Knapsack DP

Patrick Zhang

October 2019

1 Complexity Introduction

Complexity, which is often expressed in Big-O notation, is useful for estimating how long a solution for take for large test cases. You should calculate the complexity of your solution before implementing to ensure that it will fit within the time limit. Also, knowing what complexity is required to solve the problem based on the sizes of the test inputs can give you a massive hint on how to solve the problem.

2 Calculating Complexity

Calculating the complexity of a solution is very simple. Consider all loops that vary based on a variable given from the test input, such as:
for(int $k = 0$; $k \leq N$; $k++$), where $1 \leq N \leq 100000$.
Those loops are of size O(N). You may have a loop, such as a binary search, that you know will only loop $\log N$ times, so those loops are of size O($\log N$).
Nested loops multiply. A O($\log N$) loop nested inside of a O(N) loop will result in O($N \log N$).
Different sets of loops add. Usually, only the largest term is represented, but you can write every term for clarity. For example, if you have an O(N^2) loop followed by O(N) loop, the Big-O will be O(N^2), although O($N^2 + N$) would be more precise.

3 Predicting Complexity Required based on the Problem

The following table details the complexity required based on N.

N	Complexity
10^9	O(1)
10^6	O(N)
10^5	O(N) or O($N \log N$)
10^4	O(N^2)
10^3	O(N^2)
10^2	O(N^3)

You can intuitively determine if a complexity is fast enough without memorizing that chart by substituting the values of N into the Big-O expression and seeing if the result is $<< 10^9$.
Before implementing a solution, you should first calculate the complexity of your solution and see if it will fit within the time limit. If it doesn't, don't waste time implementing a solution that will time out (unless you are desperate).

4 Dynamic Programming Introduction

Dynamic Programming (DP) is one of the most tested concepts at he USACO Gold and Platinum levels. About 1/3 of USACO Gold problems are DP problems.
DP is solving a problem using previously solved subproblems. It is similar, but objectively different than Greedy.

5 Example 1: The Knapsack Problem

5.1 Problem

The Knapsack Problem is one of the most fundamental DP problems. It goes as follows:
You have a bag (a "knapsack"), and three different types of coins with the values of 1, 3, and 5. How many ways are there to put the coins in the bag such that the coins sum to N $(0 \leq N \leq 10^5)$? For this problem, order matters, meaning putting a 1 coin then a 3 coin is different than putting a 3 coin then a 1 coin.

5.2 Solution

Let $f(N)$ be the number of ways to add coins such that the sum is N. Note that $f(N) = f(N-1) + f(N-3) + f(N-5)$. This is because you can reach a sum of N by adding a 1 coin to $N-1$, a 3 coin to $N-3$, or a 5 coin to $N-5$. Obviously, $f(N) = 0$ if $N < 0$, so we know that we can calculate $f(N)$ if we know $f(M)$ for all $0 \leq M < N$.

5.3 Walkthrough

Let N be 9.
We need to set an initial state. For this problem, the initial state is $f(0) = 1$. This means that there is 1 way to add coins that sum to 0 (which is to not add any coins). $f(1) = f(0) + f(-2) + f(-3)$, which equals 1.
We can continue this way until we reach 9.

N	$f(N)$
0	1
1	1
2	1
3	2
4	3
5	5
6	8
7	12
8	19
9	30

We can test our solution on smaller values to ensure that our solution is correct. For instance, $f(5) = 5$, which we know is true because there are 5 ways to add coins such that they sum to $5 : (1,1,1,1,1), (1,1,3), (1,3,1), (3,1,1), (5)$
Our answer is thus $f(9) = \boxed{30}$.

5.4 Walkthrough 2: Forward DP

That solution is called "Backward DP". When we reached a state of $f(N)$, every $f(M)$ for $0 \leq M < N$ was solved, and we used those states to solve $f(N)$. In Forward DP, when we reach a state of $f(N)$, $f(N)$ will already be completely solved. We will use $f(N)$ to help solve the states that succeed it.
Every time we reach a state of $f(N)$, we will run the following operations:

$$f(N+1) += f(N)$$
$$f(N+3) += f(N)$$
$$f(N+5) += f(N)$$

(Don't forget to check for ArrayOutOfBoundsException!)

6 Example 2: The Knapsack Problem 2

6.1 Problem

This problem is the same as the previous one, only order doesn't matter, so adding a 1 coin then a 3 coin is the same as adding a 3 coin then a 1 coin. One way to ensure you avoid duplicates is to only add coins greater than or equal to in value to the coin in the bag with the maximum value. For example, if you've already added a 3 coin, you can only add a 3 coin or a 5 coin. You may have realized that we will need to somehow store the maximum coin that we have already added. In order to do this, we will need more complicated states.

6.2 Solution

For the first problem, we have an array dp of size $[N]$, where dp$[K]$ represented the state where the sum of the coins is K. In this problem, we will have a two dimensional matrix dp of size $[N][3]$, where the state of dp$[K][J]$ represents the state where the sum of the coins is K and the maximum coin used is the Jth highest coin.
In the previous problem, we added all of the coins to each state. In this problem, we will only add the coins bigger or equal to the Jth coin.
The answer to the problem is dp$[N][0]$ + dp$[N][1]$ + dp$[N][2]$.

6.3 Walkthrough

Let N be 9.
We will be using Forward DP for this walkthrough. For simplicity, we will set dp[1][0], dp[3][1], and dp[5][2] to 1 as our initial states. They represent adding a 1 coin, 3 coin, and 5 coin, respectively.
After two interations, we end up with the following matrix:

	1	2	3	4	5	6	7	8	9
0	1	1							
1			1	1					
2					1	1			

After we finish iterating, we end up with the following matrix:

	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1
1			1	1	1	2	2	2	3
2					1	1	1	2	2

The answer is $1 + 3 + 2 = \boxed{6}$. We know the answer is correct because there are 6 ways to reach 9 : $(1,1,1,1,1,1,1,1,1), (1,1,1,1,1,1,3), (1,1,1,3,3), (3,3,3), (1,1,1,1,5), (1,3,5)$. Note that there is 1 way to reach 9 with only 1 coins, 3 ways using 1 and 3 coins, and 2 ways using 1, 3, and 5 coins.

6.4 Code

```
#include <bits/stdc++.h>

using namespace std;

//N will be <10^5 so just make a big matrix.
int dp[100005][3];

int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);

   /*
```

```
    Use this for file input/output (like for USACO),
    and use fin and fout instead of cin and cout
    ifstream fin (".in");
    ofstream fout (".out");
    */


    int N;
    cin >> N;

    //initialize putting a 1 coin, 3 coin, and 5 coin.
    dp[1][0] = 1;
    dp[3][1] = 1;
    dp[5][2] = 1;

    for(int k = 1; k < N; k++){
        for(int j = 0; j < 3; j++){
            if(dp[k][j] == 0) continue;
            if(j <= 0 && k+1 <= N) dp[k+1][0] += dp[k][j];
            if(j <= 1 && k+3 <= N) dp[k+3][1] += dp[k][j];
            if(j <= 2 && k+5 <= N) dp[k+5][2] += dp[k][j];
        }
    }

    int answer = dp[N][0] + dp[N][1] + dp[N][2];
    cout << answer << endl;


    return 0;
}
```

7 Extra Problems

Here are some quick extensions to the previous problems. Try to think about how to solve them.

- Find the Nth ($0 <= N <= 100000$) fibonacci number. What is its complexity? Observe how much faster it is than the recusive solution.
- For Knapsack Problem 2, instead of being given 3 numbers, you are given M numbers. What is the complexity of this solution? What are the possible bounds for N and M?

8 The Best Way to Practice Dynamic Programming

Go to `https://codeforces.com/problemset/page/1?tags=dp&order=BY_SOLVED_DESC`. Codeforces is a competitive programming platform, like USACO, that has a database of thousands of problems. That link filters out all DP problems and sorts them by difficulty. Start at the top and solve 1 out of every 2 or 3 problems.