Dynamic Programming

Jesse Choe, Sonny Chen, and Akshat Alok

May 20 2022

1 Introduction to DP

Dynamic Programming (DP) is one of the most frequent topics in competitive programming and appears frequently on Gold and Platinum contests. DP is important to know as this technique can be used to solve a wide variety of problems with various difficulties (recently the Gold DP problems have been much harder as compared to pre-2020 contests).

DP is a technique where you divide a problem into multiple subproblems that is solved independently. The two common applications of dynamic programming include:

- Finding the maximum or minimum solution (optimization). DP can be useful when greedy fails.
- Counting the number of possible solutions, typically modulo a large prime number such as $10^9 + 7$.

Now, let's go over two introductory DP problems to demonstrate when and how DP is used! The first problem demonstrates when greedy fails and the second problem is a counting problem.

1.1 Comfortable Trip

Problem:
Bessie wants to travel to pasture N ($1 \leq N \leq 10^5$). Each pasture is described by how uncomfortable it is (or the number of cows in that pasture). In other words, there are a_i ($0 \leq a_i \leq 10^9$) cows in the i-th pasture. Bessie wants to be as comfortable as possible in her trip to pasture N, so she decides to jump across at most K ($1 \leq K \leq 100$) pastures. Since Bessie wants to be as comfortable as possible, help her find the minimum number of cows Bessie encounters in her journey.

Sample Input:
$N = 4, K = 2, a = [2, 7, 3, 1]$

Sample Output:
6
Sample Explanation:
Bessie can visit pastures 1, 3, and 4.

Solution:
As shown by the sample input, the greedy solution (visiting as few pastures as possible) does not work here as the greedy solution would output 8 (farms 2 and 4).

Given that greedy fails, let's check all possible trips and find the trip with the minimum number of cows using the following recursive code in C++:

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

using namespace std;
using ll = long long;
using vl = vector<ll>;

#define forn(i, n) for(ll i = 0; i < n; i++)

const ll MAXN = 1e5, INF = 1e14;

ll n, k, a[MAXN];

// check(i) represents the answer if the last pasture visited in
    the trip is pasture i

ll check(ll i){
    if(i <= 0){
        return 0; // base case
    } else if(i <= k){
        return a[i - 1]; // base case
    } else {
        ll ans = INF;
        for(ll j = 1; j <= k; j++){
            // Checks all possible paths going to pasture i
            ans = min(ans, check(i - j) + a[i - 1]);
        }
        return ans;
    }
}

int main(){
    cin >> n >> k; // read in the number of pastures and the
    maximum jump size
    forn(i, n){
        cin >> a[i]; // read in the number of cows in the i-th farm
    }
    cout << check(n) << endl;
}
```

Notice that there are multiple recomputations as shown in the recursion below. Specifically, `check(1)` is called three times and `check(2)` is called two times.

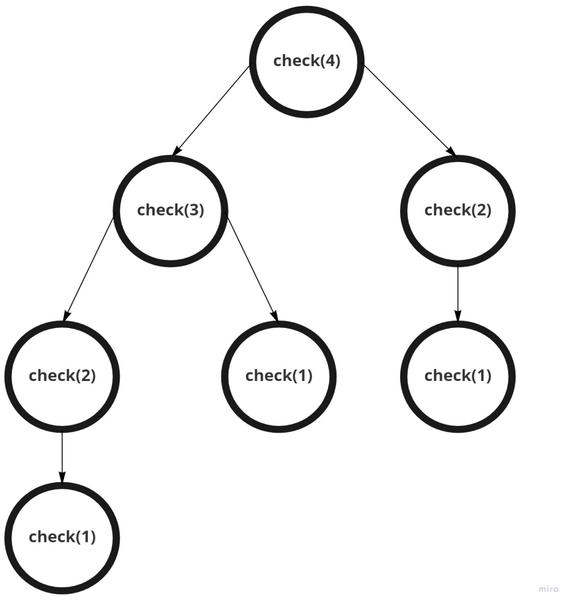

Figure 1: Recursion tree for the sample input

These extra recomputations are unnecessary, hence the exponential runtime of $O(k^n)$. Fortunately, we can improve the runtime of our algorithm by storing each recursive call. This technique is called **memoization**, which stores the answer of each recursive call in a data structure such as an array. In this particular case, we can store an array called `dp[i]`, where `dp[i]` represents the answer for the first i pastures. In competitive programming, we often refer the indices (in this case, we have only one state - the most recently visited pasture) of our `dp` as the **states**, as they identify each node in our recursion tree. We refer to the directed edges in our tree as the **transitions**, since they describe how one state transitions to the next. In the code above, the transitions were from pastures $i-k, i-k+1, \ldots, i+2, i+1$ to pasture i. The following memoized C++ optimizes the recursive code above from $O(k^n)$ to $O(nk)$:

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

using namespace std;
using ll = long long;
using vl = vector<ll>;

#define forn(i, n) for(ll i = 0; i < n; i++)

const ll MAXN = 1e5, INF = 1e14;

ll dp[MAXN];

int main(){
    ll n, k; cin >> n >> k; // reads in the number of pastures and
    jumps
    vl a(n); forn(i, n) cin >> a[i]; // reads in the number of cows
    // initializes the dp array
    forn(i, n) dp[i] = INF;
    forn(i, k){
        dp[i] = a[i];
        /* for the first k pastures, it's optimal to jump
           from FJ's pasture to the i-th pasture.
        */
    }
    for(ll i = k; i < n; i++){
        for(ll j = 1; j <= k; j++){
            // Transition/jump from i - j to i
            dp[i] = min(dp[i], dp[i - j] + a[i]);
        }
    }
    cout << dp[n - 1] << endl;
}
```

1.2 K-Sided Dice

Problem:
Bessie and Elsie are playing a dice game on a K-sided dice ($1 \leq K \leq 100$). However, they decided to make the game more interesting by making it a betting game. Elsie must pay Bessie if the total sum of all dice rolls is between A and B (inclusive) ($1 \leq A, B \leq 2 \times 10^5, A \leq B$) (note that Bessie can roll the dice as many times as possible). How many different ways can Bessie win? Output the answer modulo $10^9 + 7$.

Sample Input:
$K = 6, A = 3, B = 3$
Sample Output:
4
Sample Explanation:
There are four different ways to roll the dice such that the sum of the rolls is three:

- $1 + 1 + 1$
- $1 + 2$
- $2 + 1$
- 3

Solution:
Let `dp[i]` represent the number of different ways to make a sum of i, modulo $10^9 + 7$. Each roll, we can either roll $1, 2, \ldots,$ or k, so we get the following transitions:

$$\texttt{dp[i]} = \sum_{j=i-k}^{i} \texttt{dp[j]}$$

The base case is when i is 1, with the answer being 1. The answer to this problem is:

$$\sum_{i=A}^{B} \texttt{dp[i]}$$

To get our answer modulo $10^9 + 7$, we can use the modular identity:

$$(a + b) \mod m = ((a \mod m) + (b \mod m)) \mod m$$

Here is the C++ code that solves this problem:

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

using namespace std;
using ll = long long;
using vl = vector<ll>;

#define forn(i, n) for(ll i = 0; i < n; i++)

const ll MOD = 1e9 + 7;

int main(){
    ll k, a, b; cin >> k >> a >> b;
    vl dp(b + 1); dp[0] = 1; // base case
    for(ll i = 1; i <= b; i++){
        for(ll j = 1; j <= k; j++){
            if(i - j >= 0){ // make sure that sum i is achievable
                // transition
                dp[i] += dp[i - j];
                // (a + b) mod m = ((a mod m) + (b mod m)) mod m
                dp[i] %= MOD;
            }
        }
    }
    ll res = 0;
    // sum of dp[i] between a...b
    for(ll i = a; i <= b; i++){
        res += dp[i];
        res %= MOD;
    }
    cout << res << endl;
}
```

1.3 Practice Problems

1. Time is Mooney - This is a good introductory problem, which involves the nodes of the directed graph as the states.
2. Snakes - This is an interesting DP problem that uses prefix sums (a silver topic).
3. Taming the Herd - This is a more challenging problem (try solving this problem in $O(n^2)$ time!).

4. Other Problems - USACO Guide is a good resource for topic based practice. Other problems not mentioned above can be found here.

1.4 General Strategies

As you solve the problems above, we recommend you use the following strategy to tackle DP problems (as they can be quite challenging at times).

1. **Identify the states**: consider what information is important for a given subproblem (the sum of the dice rolls, for example).
2. **Determine the base cases**: consider what the most simple subproblem is (n = 0, for example).
3. **Make the transitions**: These transitions allow you to transition between states (usually these transitions will be more complicated than the ones from Comfortable Trip and K-Sided Dice).

2 Knapsack DP

Knapsack problems are problems where a subset of a set of objects is used to satisfy a property that can be found with those sets of objects. Knapsack problems, like normal DP problems, usually involves either counting or optimizing a certain value, using a subset of the objects. Knapsack DP solutions usually have the capacity of the knapsack as the state and the transition adding an object to that knapsack. Knapsack problems in competitive programming will be more difficult than many of the classical knapsack problems.

Now, let's go over a classical knapsack problem, which is a modification of the K-Sided Dice problem above!

2.1 K-Sided Dice 2

Problem:
Bessie kept on losing, so she decided to change the rules. She asked Elsie to bring N ($1 \leq N \leq 100$) dice, with the i-th dice being a k_i-sided dice ($1 \leq k_i \leq 10^5$), each with a 100% chance of landing on k_i. Bessie wins the game if the sum of all dice rolls is between A and B ($1 \leq A, B \leq 10^5, A < B$). Note that Bessie can roll each die as many times as she wants (she can even roll each die zero times). How many distinct, **ordered** ways are there for Bessie to win. Output the answer modulo $10^9 + 7$. It is guaranteed that k_i is distinct across all N dice.

Sample Input:
$N = 3, A = 3, B = 3, k = [1, 2, 3]$
Sample Output:
3

Sample Explanation: There are three different ordered ways to roll the dice, such that the sum of the rolls is three:

- $1 + 1 + 1$
- $1 + 2$
- 3

Solution: Let `dp[i][j]` represent the number of different ordered ways for Bessie to win with the sum of the die as j, using the first i dice. The base case is when $i = 0, j = 0$ in which the answer is 1, since there is one way to make a sum of 0 (when no die is rolled). The transition of the DP is: `dp[i][j]` = `dp[i - 1][j]` + `dp[i][j - k[i]]`, since Bessie can either roll the i-th die or not roll it. The answer to this problem is:

$$\sum_{j=A}^{B} \texttt{dp[n][j]}$$

Note that we can optimize the memory using the sum of the die as our state so instead of `dp[i][j]` we would store the DP as `dp[i]`, which represents number of different ordered ways for Bessie to win with the sum of the die as i. However, this optimization is not required under the time and memory constraints. Here is the C++ code for the original solution:

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

using namespace std;
using ll = long long;
using vl = vector<ll>;

#define forn(i, n) for(ll i = 0; i < n; i++)

const ll MOD = 1e9 + 7, MAXN = 100, MAXS = 1e5;

ll dp[MAXN + 1][MAXS + 1];

int main(){
    ll n, a, b; vl k(n);
    cin >> n >> a >> b;
    forn(i, n){
        cin >> k[i];
    }
    dp[0][0] = 1; // base case
    for(ll i = 1; i <= n; i++){ // Used to make the knapsack
    ordered
        forn(j, b + 1){
            dp[i][j] = dp[i - 1][j]; // Account for all the
    previous ways
            if(j - k[i - 1] >= 0){
                dp[i][j] += dp[i][j - k[i - 1]]; // Roll the
    current dice
                dp[i][j] %= MOD;
            }
```

```
                dp[i][j] %= MOD;
            }
        }
    ll res = 0;
    for(ll i = a; i <= b; i++){
        res += dp[n][i];
        res %= MOD;
    }
    cout << res << endl;
}
```

2.2 Problems

1. Fruit Feast - A good introductory knapsack problem
2. Cow Poetry - A knapsack problem with a number theory twist
3. Talent Show - A knapsack problem that involves binary searching on the answer for the solution. This problem is quite interesting.
4. Other Problems - USACO Guide is a good resource for topic based practice. Other problems not mentioned above can be found here.

3 Parting Shots

Getting better at DP requires lots and lots of practice! DP problems are usually one of the harder topics, so active practice is necessary in order to get better at solving these problems. Here is a list of resources and problems to help improve your DP skills:

1. CSES DP Section - K-Sided Dice is a modified problem of Dice Combinations and Coin Combinations II. Other good problems can be found here.
2. AtCoder DP Contest - Comfortable Trip is a modified problem of Frog 2. Other good problems can be found here.
3. USACO Guide DP Problems - USACO Guide has various DP problems including normal DP problems, knapsack DP problems, grid paths, longest increasing subsequence, bitmask DP, range DP, and tree DP.
4. DP Book - This book is an excellent resource to learn the different types of DP. At the end of the book, there are extra problems for you to practice with.
5. Codeforces DP - CF has many high quality DP problems. There are many problems for you to practice with. It is highly encouraged to do these problems.