Dynamic Programming for Computing Contests

Arpan Banerjee

2021

"Those who cannot remember the past are condemned to repeat it."
— George Santayana

Copyright ©2021 by Arpan Banerjee
All rights reserved. No part of this book may be reproduced or used in any manner without the prior written permission from the copyright owner.

Preface

Dynamic programming is arguably the most prevalent topic in competitive programming. It appears in almost every USACO Gold and Platinum contest as well as Codeforces, AtCoder, CodeChef, and most other popular contests. Many would also agree that it's one of the most interesting topics in competitive programming.

My primary motivation for writing this book was the abundance of people—from Codeforces blogs to Discord servers—saying something along the lines of "How do I get better at DP" in addition to my own struggle with DP. This book is primarily intended for people who have little to no experience with DP and want to be well-versed with standard topics and ideas.

I would like to thank Samarth Gupta and Elephant for valuable feedback and Evan Chen for help and inspiration with regard to LaTeX.

Prerequisites

It is recommended that the reader knows most concepts that appear on the Bronze and Silver divisions of USACO. Namely, including but not limited to:

- Familiarity with competitive programming (i.e. what it is/how it works)
- Computational complexity theory
- Prefix sums
- Graph traversals (DFS/BFS/floodfill) + basic graph theory
- Recursion
- Coordinate compression
- Custom comparators
- Binary search
- Two pointers
- Greedy algorithms

A Few Notes

There are problems at the end of each chapter, most of which are from USACO. They are arranged in roughly ascending order of difficulty. It is recommended that you do a few problems that are hard for you, but not to the extent that you make no progress after significant effort (i.e., problems that are reasonably above your comfort zone). All code provided is compatible with C++11. More advanced techniques, optimizations, and additions might be added here later (early 2022). For now, this can be considered an introduction.

Contact

I am open to any corrections, suggestions, or questions. If you wish to contact me, you can email `arnbnrch@gmail.com`.

Author's Profile

Arpan Banerjee is currently a USACO Platinum competitor and a high school senior in NC, USA.

Contents

I Dynamic Programming **7**

1 Introduction **8**
1.1 Overlapping Subproblems . . . 8
1.1.1 Fibonacci . . . 8
1.2 Optimal Substructure . . . 9
1.3 Maximum Sum Subarray . . . 9
1.4 DAG Formulation . . . 10
1.5 General Strategy . . . 11

2 Counting **13**
2.1 Modular Arithmetic . . . 13
2.1.1 Rules . . . 13
2.1.2 Modular Multiplicative Inverse . . . 14
2.1.3 With Fermat's Little Theorem . . . 14
2.1.4 Binary Exponentiation . . . 15
2.1.5 With the Extended Euclidean Algorithm . . . 16
2.2 Binomial Coefficients . . . 17
2.2.1 With Pascal's Identity . . . 17
2.2.2 With Modular Inverse . . . 18
2.3 Problems . . . 18

3 Knapsack **19**
3.1 Fractional Knapsack . . . 19
3.2 Coin Change . . . 19
3.3 When Greedy Fails . . . 20
3.4 0-1 Knapsack . . . 20
3.5 Problems . . . 22

4 Trees **23**
4.1 Introduction . . . 23
4.2 Motivating Problems . . . 24
4.3 Rerooting . . . 27
4.4 Problems . . . 29

5 Digits **30**
5.1 Introduction . . . 30
5.2 Problems . . . 36

6 Bitmasking **37**
6.1 Assignment Problem . . . 37
6.2 Representing Subsets With Bitmasks . . . 37
6.3 Full Solution . . . 38
6.4 Problems . . . 39

7 Ranges **40**

II Parting Shots 42

8 Problems 43

9 Resources 45

I

Dynamic Programming

1 Introduction

The essence of dynamic programming is breaking down a problem into **subproblems** and attaining and combining the answers for them in such a way that the answer for the original problem can be attained efficiently. The solutions to the subproblems are used to attain the solutions to other subproblems through **transitions**. The reason for the improved efficiency of dynamic programming (DP hereafter) is that redundant computations are avoided by keeping **states** which contain only the necessary data, usually such that they can be visited exactly once. In the scope of competitive programming, DP problems usually involve optimizing some quantity (i.e. minimizing or maximizing something) or counting some quantity (i.e. the number of ways to do something). Usually DP reduces the naive exponential solution to polynomial time. There are two properties a problem must exhibit to be able to be solved with DP:

§1.1 Overlapping Subproblems

Usually if you are doing **top-down dynamic programming** (starting at the largest subproblem and cascading to smaller subproblems from there), some subproblems are computed multiple times. These redundant computations can be avoided with **memoization**—storing the answer for each subproblem. Consider the following top-down approach.

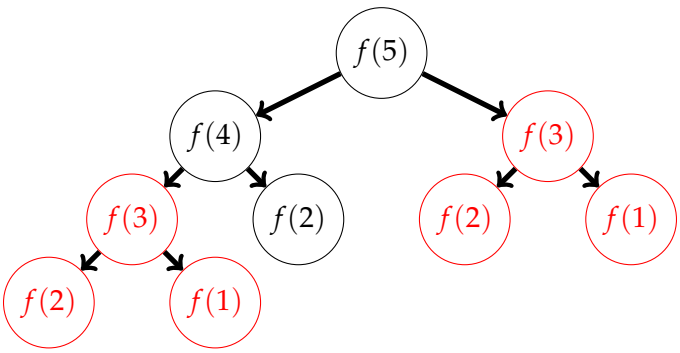
§1.1.1 Fibonacci

Let's consider the Fibonacci series: $1, 1, 2, 3, 5, 8, \ldots$. We can define it recursively as:

$$f(x) = \begin{cases} f(x-1) + f(x-2) & x \geq 3 \\ 1 & x \leq 2 \end{cases}$$

```
int f(int x){
    if(x<=2) return 1;
    else return f(x-1)+f(x-2);
}
```

Here is the recursion tree for $f(5)$:



It can be easily noticed that $f(3)$ is computed twice. It turns out that a lot of redundant recomputations happen if $f(x)$ is computed for some large x. Each $f(x)$ calls two children.

Each of those children call two children, and so forth. The children are only called for $x-1$ and $x-2$, so the values in the nodes decrease by a small constant (1 or 2). This means this approach takes roughly $O(2^x)$ time to find $f(x)$. A tighter bound is $O(\Phi^x)$, but the main takeaway here is that it's exponential due to the same subproblems being computed multiple times.

However, if we use memoization, it will take $O(x)$ time at the expense of $O(x)$ memory. This is because each state (subproblem) is visited at most once:

```
const int X=50;
int memo[X+1];

int f(int x){
    if(memo[x]!=-1) return memo[x];
    if(x<=2) return 1;
    else return memo[x]=f(x-1)+f(x-2);
}
int main(){
    for(int i=0; i<=X; i++) memo[i]=-1;
    int ans=f(X);
}
```

§1.2 Optimal Substructure

Note that the following language mainly applies to the concept of using dynamic programming to solve optimization problems. For counting problems, you can consider "optimal" to be replaced with "accurate".

Also known as Bellman's principle of optimality, the idea of optimal substructure is that a reaching a globally optimal solution is impossible if locally suboptimal decisions are made. In short, the subproblems must have optimal answers in order for the problem to be solvable with dynamic programming.

> **Remark 1.2.1.** Note that greedy solutions also demonstrate optimal substructure, but what differentiates a greedy approach from a DP one is that greedy does not use optimal solutions to subproblems to make a choice, but rather it first makes a greedy choice which results in optimal substructure.

§1.3 Maximum Sum Subarray

Let's say there is an array a of length n. The goal is to find the sum of the maximum sum subarray of a. For convenience and clarity, we are not considering the empty subarray. With brute force, we can iterate over all subarrays and keep track of the best sum. The sum of a subarray can be found in $O(1)$ with prefix sums, and there are $O(n^2)$ subarrays, so the brute force approach takes $O(n^2)$ time.

But how do we do this in subquadratic time? Of course, we can use DP. Since we are trying to find the maximum sum subarray of the entire array, we can try to find the max sum subarray ending at position i for all i, then pick the best one. The max sum subarray of a prefix i, denoted by $DP[i]$, is simply the max of the max sum subarray ending at $i-1$ with and without the addition of $a[i]$. More formally,

$$DP[i] = \max(DP[i-1] + a[i], a[i])$$

Algorithm

Precondition : a and DP are 1-indexed
array $DP \leftarrow 0$
for $i \leftarrow 1$ *to* n **do**
  $DP[i] \leftarrow \max(DP[i-1] + a[i], a[i])$
end
return $max(DP)$

Interestingly, this algorithm has a name: **Kadane's Algorithm**. Since we are pulling from $i-1$ when we are at i, this type of DP is sometimes called **pull DP**. A **push DP** approach, pushing from index i to $i+1$, is as follows:

Algorithm

Precondition : a and DP are 1-indexed
array $DP \leftarrow 0$
for $i \leftarrow 0$ *to* $n-1$ **do**
  $DP[i+1] \leftarrow \max(DP[i] + a[i+1], a[i+1])$
end
return $max(DP)$

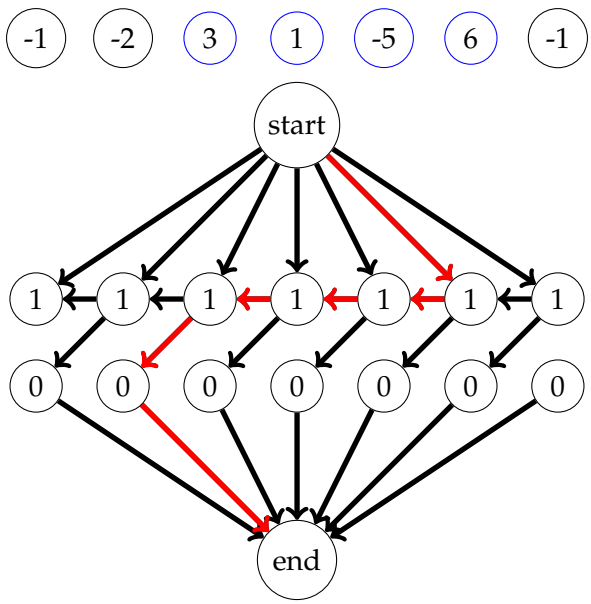
Notice that since the DP only requires data from an adjacent index while the indices are iterated upon, the space complexity can be reduced from $O(n)$ to $O(1)$ by using a few variables instead of an array. Both of these approaches can be classified as **bottom-up dynamic programming** since we are starting at the smallest subproblems and building up to the answer for the entire array.

§1.4 DAG Formulation

Every DP can be formulated as a DAG—a directed acyclic graph. The nodes of the DAG are the subproblems and the edges are the transitions between the subproblems. Consequently, the best path in the DAG is the path the DP takes. The reason its a directed **acyclic** graph is that if there are cycles, some states would lead back to themselves after a series of transitions (edges) with nowhere to stop at. Consider the following array:

-1	-2	3	1	-5	6	-1

The answer is highlighted in blue. Below is the DAG of the pull DP version with the optimal path highlighted in red.



Nodes with 1 represent $DP[i]$ and nodes with 0 represent not including that element in the array. Since $DP[i]$ is the max of $DP[i-1]+a[i]$ and $a[i]$, the edges (transitions) are converted into edges accordingly. From the 1 node, going to the previous 1 node corresponds to $DP[i-1]+a[i]$, and going to the previous 0 node corresponds to $a[i]$.

§1.5 General Strategy

1. **Identify the states**: Consider what information is necessary to find the desired result, and what can be disregarded. Remember that each state should exhibit optimal substructure.

2. **Determine the base cases**: These are the initial state(s) that all other states depend on.

3. **Make the transitions**: The transitions enable you to transition from one state to another. After all transitions are complete, the answer is usually attained trivially.

Here is an interesting problem that you can try:

Problem 1.5.1 (Patrick Zhang, TJ SCT)

There are N flowers. Each flower can either be red flower or a white flower. How many ways are there to arrange them in a line of length N such that there are never more than M flowers of the same color in a row?

$$1 \leq N, M \leq 5000$$

Solution. To take care of the condition of having no more than M flowers of the same type consecutively, the type of flower can be incorporated into the DP state: $DP[i][(0,1)]$ =number of configurations if

$$i: \text{ first } i \text{ flowers placed}$$

$$(0,1): \text{ type of the } i\text{th flower}$$

So, we can have a DP array of size $N \times 2$ with the initial states being

$$DP[0][0] = DP[0][1] = 1$$

Then, it can be filled from $DP[i][0]$ as follows:

$$\begin{aligned} DP[i+1][1] &\text{+=} DP[i][0] \\ DP[i+2][1] &\text{+=} DP[i][0] \\ &\vdots \\ DP[i+M][1] &\text{+=} DP[i][0] \end{aligned}$$

The reasoning for this is that if flower i is placed at position p, you can add $1 \ldots M$ flowers of the opposite type in positions $p+1 \ldots p+M$ while maintaining the constraint of not having more than M contiguous flowers of the same type. By storing the number of configurations of the prefix $1 \ldots p$ with flower i placed at position p, we can easily update the next M positions of the opposite flower. This transition can be repeated for all i between 0 and N and done similarly from $DP[i][1]$. This is colloquially called **push DP** since each state pushes into the next states. Then the answer is $DP[N][0] + DP[N][1]$. Since we loop at most M times for at most $2N$ states, this approach takes $O(NM)$ time.

$\square$

2 Counting

Many problems ask for the number of ways some conditions can be fulfilled under some constraints. Oftentimes, the problem asks for the answer to be outputted modulo a large number (usually 1000000007 or 998244353), since the answer could be too large to fit into a 64 bit integer. It is no coincidence that these two numbers are prime.

§2.1 Modular Arithmetic

% in many programming languages, or **mod** is the integer modulo operator:

Theorem 2.1.1 (Division Theorem)
For integers a and d, there exist unique integers q and r such that $a = dq + r$ and $0 \leq r < n$. From this, the definition of a modular congruence can be attained:

$$a \equiv r \mod d$$

spoken as "a is congruent to r modulo d." r is sometimes called the modulo d residue of a and d is called the modulus.

§2.1.1 Rules

Below are some provable identities:

$$\begin{aligned}
a^b \mod m &= (a \mod m)^b \mod m \\
(a \times b) \mod m &= ((a \mod m) \times (b \mod m)) \mod m \\
(a + b) \mod m &= ((a \mod m) + (b \mod m)) \mod m \\
(a - b) \mod m &= ((a \mod m) - (b \mod m)) \mod m
\end{aligned}$$

The main takeaway is that you can do basic algebra under a modulus by modding individual parts. Here is an easy problem to demonstrate this in practice:

Problem 2.1.1 (CSES Counting Towers)
Your task is to build a tower whose width is 2 and height is n. You have an unlimited supply of blocks whose width and height are integers. Given n, how many different towers can you build? Mirrored and rotated towers are counted separately if they look different.

The first input line contains an integer t: the number of tests. After this, there are t lines, and each line contains an integer n: the height of the tower. For each test, print the number of towers modulo $10^9 + 7$.

Solution. Consider the possible states of each row. For each row r, we can keep track of the number of ways to populate the $2 \times r$ grid, and increase r until we reach $2 \times N$, giving the required answer. Notice that at each row, we can either choose to join the left and right cells, or leave them disconnected. This motivates us to keep two counts, one for the number of ways to populate the $2 \times r$ grid if we join the left and right cells of row r, and another one for if we keep them separate. Then

$$\text{join}(r) = \underbrace{\text{join}(r-1)}_{\text{Continue all configurations joined at r-1}} + \underbrace{\text{sep}(r-1) + \text{join}(r-1)}_{\text{Make new joined block}}$$

If both rows $r-1$ and r are separate, there are four cases for the two cells in row r:

1. The left cell is extended from below
2. The right cell is extended from below
3. None are extended from below
4. Both are extended from below

From this, we get

$$\text{sep}(r) = 4 \cdot \text{sep}(r-1) + \underbrace{\text{join}(r-1)}_{\text{Creating two new cells in row } r}$$

Then the answer is $\text{sep}(N-1) + \text{join}(N-1)$ if we 0-index. The base cases are $join[0] = 1$ and $sep[0] = 1$. *join* and *sep* are initialized to 0 since nothing can be built initially. Below is an implementation of this with the answer being taken modulo $mod = 10^9 + 7$.

```
for(int i=1; i<n; i++){
        sep[i]=((sep[i-1]*4)+join[i-1]);
        sep[i]%=mod;
        join[i]=(join[i-1]*2+sep[i-1]);
        join[i]%=mod;
}
cout<<((sep[n-1]+join[n-1])%mod)<<'\n';
```

□

§2.1.2 Modular Multiplicative Inverse

You may have noticed there is no division rule to accompany the multiplication rule. This is because dividing under modulo is significantly more difficult, as the modulo operation only operates on integers. However, it is possible using a **modular multiplicative inverse** b^{-1} of the divisor b satisfying $b \cdot b^{-1} \equiv 1 \mod m$. So, $(a/b) \mod m$ can be written as $(a \times b^{-1}) \mod m$. Consider

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

This obviously does not hold since the right quotient is not necessarily an integer, but

$$(a/b) \mod m = (a \times b^{-1}) \mod m = ((a \mod m) \times (b^{-1} \mod m)) \mod m$$

does, according to the multiplication rule. There are two main ways to find the modular multiplicative inverse: Fermat's Little Theorem and the Extended Euclidean Algorithm. Moreover, there is an infinite number of modular multiplicative inverses of a under modulo p; they just have to be part of the same equivalence class, but usually the smallest one is found.

§2.1.3 With Fermat's Little Theorem

Theorem 2.1.2 (Fermat's Little Theorem)

For a prime p and any integer a, $a^p \equiv a \mod p$.

Using the multiplication rule in reverse, $a^{p-1} \equiv 1 \mod p \implies a \cdot a^{p-2} \equiv 1 \mod p \implies \boxed{a^{-1} = a^{p-2}}$.

> **Remark 2.1.3.** This result only holds for $a \mod p \neq 0$ since it's impossible for $0 \cdot a^{-1}$ to be congruent to 1 mod p.

This can be generalized:

> **Problem 2.1.2**
>
> Prove that if $a \cdot a^{-1} \equiv 1 \mod m$, $\gcd(a, m)$ must be 1.

Proof. For some nonnegative integer k,

$$a \cdot a^{-1} = km + 1 \implies a \cdot a^{-1} - km = 1$$

$\gcd(a, m)$ can be factored out:

$$\gcd(a, m)\left(\frac{a^{-1} - km}{\gcd(a, m)}\right) = 1$$

This only holds if $\gcd(a, m) = 1$. □

§2.1.4 Binary Exponentiation

But how do we find a^{p-2}, or more generally $a^b \mod m$, quickly? We can use a technique called **binary exponentiation**, or exponentiation by squaring. It relies on the fact that b can be written as the sum of $\leq \lceil \log_2 b \rceil$ powers of two from its binary representation. Let's say $b = 2^{p_1} + 2^{p_2} + \ldots 2^{p_n}$ for $p_1 < p_2 \ldots < p_n \leq \lceil \log_2 b \rceil$ and $n \leq \lceil \log_2 b \rceil$. Then we are looking for $a^{(2^{p_1} + 2^{p_2} + \ldots 2^{p_n})} = a^{2^{p_1}} \cdot a^{2^{p_2}} \cdots a^{2^{p_n}}$. We can repeatedly square a at most $\lceil \log_2 b \rceil$ times, enumerating from $a^{2^{p_1}}$ to $a^{2^{p_n}}$, resulting in a time complexity of $O(\log b)$.

> **Remark 2.1.4.** This concept can also be extended to matrices. When applied to matrices, it is called Matrix Exponentiation.

Here is an implementation in C++. Of course, this can be used to calculate a modular multiplicative inverse in $O(\log m)$ of some a under prime modulo m as mentioned previously.

```
const int m=1e9+7;

int expo(int a, int b){
        if(a==0) return b==0;
        int ans=1;
        while(b){
                if(b&1) ans=(ans*a)%m;
                b>>=1LL;
                a=(a*a)%m;
        }
        return ans;
}
```

Here is an easy problem if you want to test your implementation.

§2.1.5 With the Extended Euclidean Algorithm

What if the modulus is not prime? We can still find inverses granted that the modulus and the integer we're trying to find the inverse for are coprime. For a nonnegative integer k,

$$a \cdot a^{-1} \equiv 1 \mod m \implies a \cdot a^{-1} - mk = 1$$

Since k is an irrelevant constant, if we let $k := -k$,

$$a \cdot a^{-1} + mk = 1$$

For simplicity, let a be a, a^{-1} be x, k be y, and m be b:

$$ax + by = 1$$

This is called a **linear Diophantine equation**, and one solution can be found with the Extended Euclidean Algorithm. Note that we are trying to solve for (x, y). The existence of solutions is guaranteed by **Bézout's Lemma** as well as the fact that this is a reduced version of our original problem of finding a modular inverse which must exist. With the Extended Euclidean Algorithm, we find x, y such that $ax + by = \gcd(a, b)$. In our case, $\gcd(a, b) = 1$. The answer, a^{-1}, is the value of x.

Lemma 2.1.5 (Euclidean Algorithm)
For positive integers $a, b, a > b$,

$$\gcd(a, b) = \gcd(b, a - b \cdot \lfloor a/b \rfloor)$$

In other words, when dividing a by b, if $a = bq + r$,

$$\gcd(a, b) = \gcd(b, r) = \gcd(b, a \mod b)$$

The algorithm is just repeatedly using this identity until a trivial state is reached:

```
int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}
```

The time complexity is roughly logarithmic with respect to a and b.

The **Extended Euclidean algorithm** is a small extension to the Euclidean algorithm enabling us to find a solution to $ax + by = \gcd(a, b) = 1$. Here is an example demonstrating the extension:

Problem 2.1.3
Find the modular multiplicative inverse of $27 \mod 100$.

Solution. We are looking for 27^{-1} where $27 \cdot 27^{-1} \equiv 1 \mod 100$.

$$27x + 100y = 1$$

Let's first use the Euclidean algorithm to find $\gcd(27, 100)$:

$$
\begin{aligned}
100 &= 27 \cdot 3 + 19 & \gcd(100, 27) &= \gcd(27, 19) \\
27 &= 19 \cdot 1 + 8 & \gcd(27, 19) &= \gcd(19, 8) \\
19 &= 8 \cdot 2 + 3 & \gcd(19, 8) &= \gcd(8, 3) \\
8 &= 3 \cdot 2 + 2 & \gcd(8, 3) &= \gcd(3, 2) \\
3 &= 2 \cdot 1 + 1 & \gcd(3, 2) &= \gcd(2, 1)
\end{aligned}
$$

Notice that it's guaranteed that the last residue is 1 because of the precondition $\gcd(a, b) = 1$, or in this case, $\gcd(27, 100) = 1$. Rearranging:

$$
\begin{aligned}
\mathbf{100} - \mathbf{27} \cdot 3 &= \mathbf{19} \\
\mathbf{27} - \mathbf{19} \cdot 1 &= \mathbf{8} \\
\mathbf{19} - \mathbf{8} \cdot 2 &= \mathbf{3} \\
\mathbf{8} - \mathbf{3} \cdot 2 &= \mathbf{2} \\
\mathbf{3} - \mathbf{2} \cdot 1 &= \mathbf{1}
\end{aligned}
$$

We can treat the bold numbers as variables. The first equation resembles our original Diophantine equation with the exception of the right side, so let's change that with some substitutions to get **19** in terms of **1**. Starting at the bottom, we can see that if we treat the bold numbers as variables and substitute the bold numbers from the right side of the above equation into the left side of the current equation and collect like terms, the intermediary bold numbers disappear, leaving only 1 and the highest bold numbers, as desired. Doing this results in

$$27(-37) + 100(10) = 1$$

so $27^{-1} \equiv -37 \equiv 63 \mod 100$. □

§2.2 Binomial Coefficients

The number of ways to choose an unordered set of k items from a pool of $n \geq k$ distinguishable items is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. $\binom{n}{k}$ is equal to the coefficient of $x^k y^{n-k}$ in the expansion of $(x+y)^n$ from the Binomial Theorem, hence $\binom{n}{k}$ is called a binomial coefficient.

> **Remark 2.2.1.** For $n < k$, $\binom{n}{k}$ is considered to be 0.

Most often, you will need to compute a binomial coefficient under a modulo m.

§2.2.1 With Pascal's Identity

We can find $\binom{n}{k}$ with the help of DP and Pascal's Identity, which says

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

This is directly derived from Pascal's Triangle:

$$
\begin{array}{lccccccccc}
n=0: & & & & & 1 & & & & \\
n=1: & & & & 1 & & 1 & & & \\
n=2: & & & 1 & & 2 & & 1 & & \\
n=3: & & 1 & & 3 & & 3 & & 1 & \\
n=4: & 1 & & 4 & & 6 & & 4 & & 1 \\
\vdots & & & & & & & & &
\end{array}
$$

However, computing $\binom{n}{k}$ takes $O(n^2)$ time, which is infeasible if n is a large number.

§2.2.2 With Modular Inverse

Since we are finding the quotient $\dfrac{n!}{k!(n-k)!}$ under a modulo m, we can use modular inverses to directly find the binomial coefficient.

$$\frac{n!}{k!(n-k)!} \equiv n! \cdot k!^{-1} \cdot (n-k)!^{-1} \mod m$$

$n!$ and $k!$ can be found easily by precomputing factorials $\mod m$.

§2.3 Problems

1. CF 1228E
2. USACO Platinum Counting Graphs

3 Knapsack

Knapsack problems usually entail filling a container—usually of fixed capacity—with a subset of items while optimizing or counting some quantity. There are two main categories of knapsack problems:

1. Each item consists of (weight, value)
2. Each item consists of (weight)

The standard knapsack problems rarely (if ever) directly appear in contests. Instead, they appear with variations and twists or in the guise of a different idea. Below are two of the most traditional knapsack problems:

§3.1 Fractional Knapsack

Problem 3.1.1 (Fractional Knapsack)

There are n items, each with weight w_i and value v_i. You need to fill a knapsack with weight capacity W with the items. Items can be broken into *fractional amounts* with the w/v ratio of the pieces being the same as that of the original item. The items are finite, that is, you can put at most 1 of each item into the knapsack. The objective is to find the maximum value the knapsack can carry.

```
Sample Input
W=3
(w, v):
(1, 1)
(3, 2)

Sample Output
1.666667 (take the first item and $1/3$ of the first item)
```

Solution. We can greedily choose the items with largest v/w ratio until the entire item has been taken. In that case, we choose the next highest v/w ratio and do the same procedure. This takes $O(n \log n)$ time because of sorting the items. □

§3.2 Coin Change

Problem 3.2.1 (Coin Change)

Given a set of n coin values $C = \{c_1, c_2, \ldots c_n\}$, what is the minimum number of coins needed to achieve a sum of k, assuming there is an unlimited supply of coins of each type.

Solution. Consider the following constraint: $1 \in C$, and if the elements C are in non-decreasing order $c_1 \leq c_2 \cdots \leq c_n$, then for all $i, c_i | c_{i+1}$. In this case, we can do a similar greedy approach as we did with fractional knapsack: deduct k by the maximum value in C until k is less than it, in which case repeat the process until $k = 0$. A proof that this works is that for each larger

denomination, all smaller denominations divide it. Therefore, a mapping can always be made from a larger denomination to a set of smaller ones, ensuring there is no better way to deduct from k than by greedily choosing the largest elements in C. It turns out that this greedy approach only works for canonical coin systems, of which the aforementioned constraint is a subset of.

However, if this was not the case, the greedy approach would not necessarily work. Consider $C = \{1, 4, 5, 6\}$ and $k = 9$, for instance.

§3.3 When Greedy Fails

Many times when greedy solutions fail, dynamic programming can be done. This is also a reason why extreme caution is necessary when certifying the correctness of a greedy solution. Since our goal is to minimize the number of coins, we naturally consider doing DP with the state being (sum $\rightarrow$ minimum number of coins), yielding a time complexity of $O(nk)$:

Algorithm Coin Change

```
array DP ← 0
for i ∈ [0, k) do
    for Coin c ∈ C do
        if i + c ≤ k then
            DP[i + c] = min(DP[i + c], DP[i] + 1)
        end
    end
end
return DP[k]
```

It turns out that this is the widely accepted DP solution to this problem. Some variations include:

1. Number of ways to produce a sum with a set of coins (unordered)(ordered)
2. All possible sums producible with a set of coins (modification of the subset sum problem)

□

§3.4 0-1 Knapsack

Problem 3.4.1 (0-1 Knapsack)

There are n items that need to be put into a knapsack with weight capacity W. Each item has a weight w_i and a value v_i. Items cannot be broken; they are either taken or not taken. Find the maximum value the knapsack can carry.

```
Sample Input
W=3
(w, v):
(1, 1)
(3, 2)

Sample Output
2 (take item 2)
```

Solution. If we use the same greedy strategy as we did for fractional knapsack, we will take the first item, but since there isn't enough space for the second item, our answer is 1. This is clearly not optimal, as we can take the second item instead. But how do we find the optimal subset of items?

Remember, the goal is to find the *maximum value* of *at most n items* that we put into a knapsack of *weight capacity* W. This motivates us to keep a DP to keep track of the maximum value for each state, with a state being defined by (x: first x items, y: knapsack weight capacity $y \leq W$). Then if $w = weights[i]$ and $v = values[i]$, our transitions are

$$DP[i][j] = \begin{cases} \max(v, DP[i-1][j-w] + v, DP[i-1][j]) & \text{if } j \geq w, \\ DP[i-1][j] & \text{otherwise.} \end{cases}$$

and the full algorithm is:

Algorithm 0-1 Knapsack

array $DP \leftarrow 0$
for $i \in [1, n)$ **do**
  for $j \in [0, W)$ **do**
    $DP[i][j] \leftarrow DP[i-1][j]$
    if $j \geq w$ **then**
      $DP[i][j] \leftarrow \max(v, DP[i-1][j-w] + v)$
    end
  end
end
return $DP[n][W]$

This can be optimized further to $O(n)$ space complexity rather than $O(nW)$ by realizing that the DP only pulls from one row before; hence, rows 1 through $i-2$ can simply be trashed when going through row i:

```
// dp size is [2][W+1]
for (int i = 1; i <= n; i++){
        int v = values[i], w = weights[i];
        for (int j = 0; j < w; j++) dp[1][j] = dp[0][j]; // copy row 0 to row 1
        for (int j = w; j <= W; j++){
                dp[1][j] = max({ v, dp[0][j], dp[0][j - w] + v });
        }
        for (int j = 0; j <= W; j++){
                dp[0][j] = dp[1][j];
                dp[1][j] = 0;
        }
}
cout<<dp[0][c]<<endl;
```

Another less standard way of doing it is using a one dimensional DP table with $DP[i]$ being the maximum value sum at a weight of i. In the implementation below, $values, weights$, and DP are 0-indexed:

```
for(int i=0; i<n; i++){
      int v=values[i], w=weights[i];
      for(int j=W; j>=0; j--){
```

```
                if(j+w>W) continue;
                if(dp[j]!=0 || j==0){
                        dp[j+w]=max(dp[j+w], dp[j]+v);
                }
        }
}
```

□

Both of these approaches take $O(nW)$ time.

§3.5 Problems

1. USACO Gold Fruit Feast
2. AtCoder DP 0/1 Knapsack
3. USACO Gold Cow Poetry
4. USACO Gold Talent Show
5. CF 1132E
6. USACO Platinum Mooriokart
7. USACO Platinum Exercise

> **Remark.** Consider why doing a knapsack for the minimum weight required for each talent sum does not work for Talent Show. This solution passes all the test cases on USACO, but stresstesting should make the mistake clear.

4 Trees

§4.1 Introduction

Conventionally, let n be the number of nodes in the tree. DP on trees is usually done to optimize the naive solution—usually $O(n^2)$—to a smaller order, such as $O(n)$. Most often:

- The tree is rooted (either by nature of the problem or done intentionally by the solver)
- There are state(s) associated with each node
- The base cases are the leaves
- The subproblems are built from the leaves to the root via a BFS/DFS traversal

Below are two of the most canonical tree DP problems.

> **Problem 4.1.1**
> Find the size of the subtrees for all n nodes in a tree in $O(n)$ time.

Solution. Keep an array `size` to keep track of the sizes of each subtree. `size[i]` is the size of the subtree rooted at `i`. Initialize `size` to 1 since that is the minimum possible size of a subtree. Then after the DFS covers all subtrees of a node i, add the `size` values of i's children to `size[i]`.

Algorithm

```
array size[number of nodes] ← 1
Function DFS(i, parent):
    foreach j ∈ adjacent[node] do
        if j ≠ parent then
            DFS(j, node)
            size[i] ← size[i] + size[j]
        end
    end
End Function
```

There is also another way to do this. Keep a global variable `time`. Increment it whenever an instance of the DFS function is called. Then for each node, its subtree size is `time` − `entry_time`.

Algorithm With timestamps

```
time ← 0
array size[number of nodes]
Function DFS(i, parent):
    entry_time ← time
    time ← time + 1
    foreach j ∈ adjacent[node] do
        if j ≠ parent then
            DFS(j, node)
        end
    end
    size[i] ← time − entry_time
End Function
```

□

Problem 4.1.2

Find the diameter (longest path) in a tree in $O(n)$ time.

Solution. This is very similar to the previous problem. Instead of keeping track of the subtree size for all children of a node, keep track of the farthest distance from a leaf. The answer for a node is simply the sum of the two maximum farthest distances of it's children. □

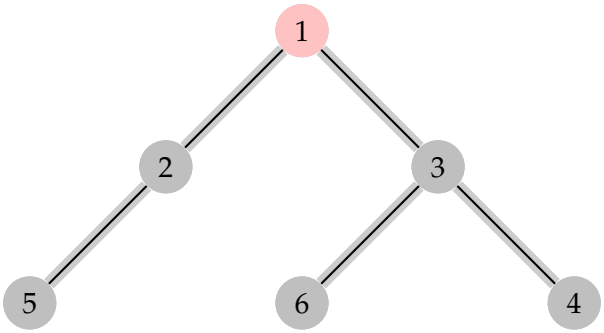
§4.2 Motivating Problems

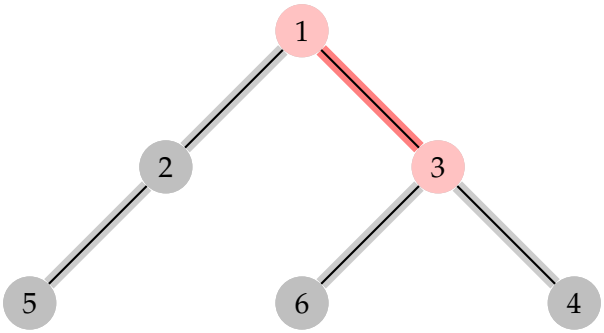
Problem 4.2.1

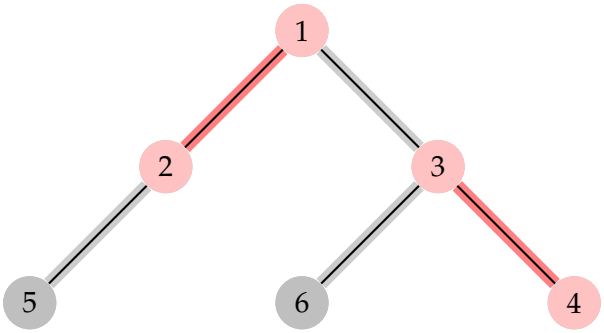
n bees numbered $1 \ldots n$ are spreading word of a flower. Bee 1 is the one that first located the flower and initially the only bee with the information. $n - 1$ pairs of bees are friends, and bees only talk to friends. It is possible for all bees to hear the news. Each minute, a bee with the news can transmit it to one friend. Compute the minimum time for all bees to receive the news.

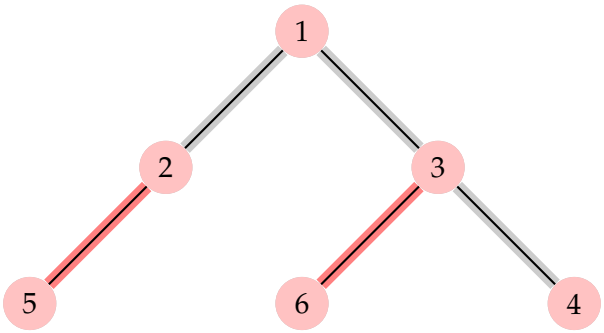
```
Sample Input
6
1 3
3 4
1 2
2 5
3 6

Sample Output
3
```

Solution. Without loss of generality, let's assume the tree is rooted at 1. Below is an optimal way the bees receive the news, from time 0 to time 3 inclusive.

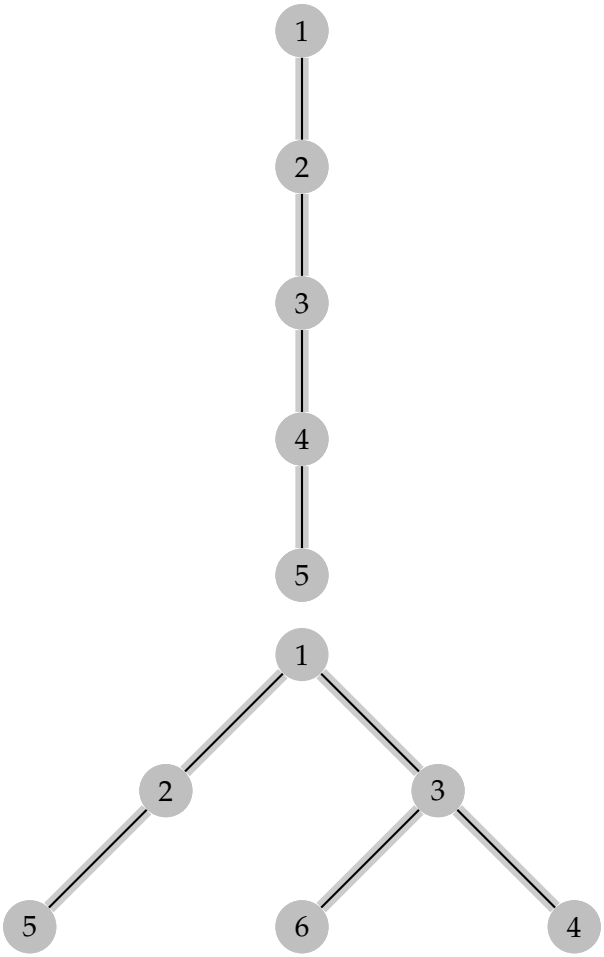








Intuitively, it makes sense to send the message to the larger subtrees earlier than the smaller ones. This is how the message spreads in the above diagram as well: it goes to 3 before 1. So, we may think that larger subtrees always require more time to spread the message than smaller subtrees. However, this is not always true. Consider the following two trees:



The second one is larger but takes less time for the message to fully spread. So, instead of looking at subtree sizes, we can look at the answer for each subtree. We can prioritize subtrees that take longer for the message to spread over those that take a shorter amount of time by keeping track of the amount of time required to spread the message across each subtree using DP. The algorithm below should make this clear.

Algorithm

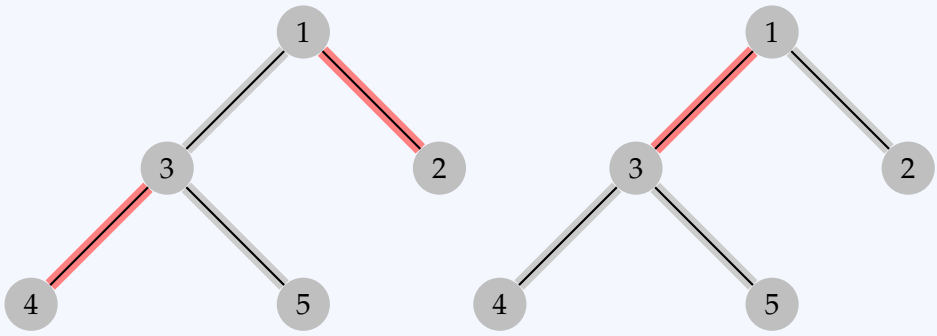
```
ans ← 0
array times[number of nodes] ← 0
Function DFS(node, parent):
    list children
    foreach i ∈ adjacent[node] do
        if i ≠ parent then
            DFS(i, node)
            children.push_back(times[i])
        end
    end
    sort(children, greatest to least)
    for i = 0 to size(children)-1 do
        times[node] = max (times[node], children[i] + i + 1)
    end
End Function
DFS(1, 1)
return times[1]
```

□

Problem 4.2.2 (CSES Tree Matching)

You are given a tree consisting of n nodes. A matching is a set of edges where each node is an endpoint of at most one edge. What is the maximum number of edges in a matching?

For example, the left tree is an optimal matching, but the right one is not:



Solution. The most obvious thing to do here is have two states for each node i:

- It is included in a matching in its subtree
- It is not included in a matching in its subtree

For each state, the maximum number of pairings in the subtree with root i can be stored. It can also be noticed that in the frame of some subtree, a matching with the root of that subtree in a pair always results in the optimal answer. The proof of this and the rest of the solution are left as an exercise for the reader. □

§4.3 Rerooting

Rerooting is a technique where information from the root is pulled (usually in sublinear time) to an adjacent node in order to attain the desired information for the adjacent node. The adjacent node is then treated as the new root, and the process continues until all the nodes in the tree have acquired their desired quantity. From there, completing the problem becomes trivial. Below is an example that nicely illustrates this concept.

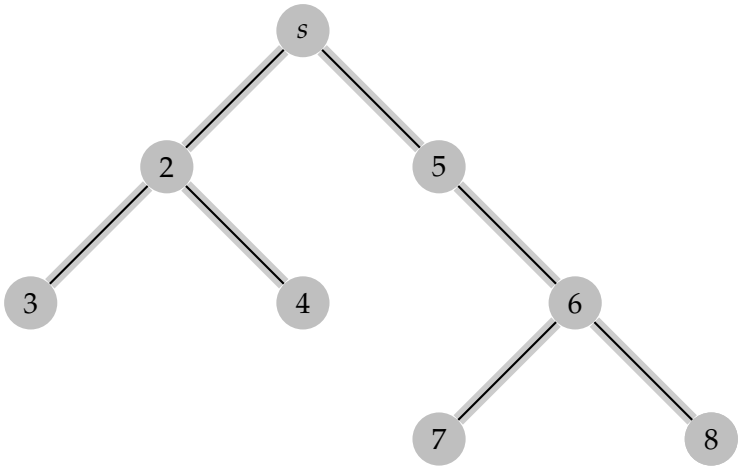
Problem 4.3.1

There is a weighted tree with n nodes $1 \ldots n$ and $n-1$ edges. What is the expected value of the sum of a randomly chosen path.

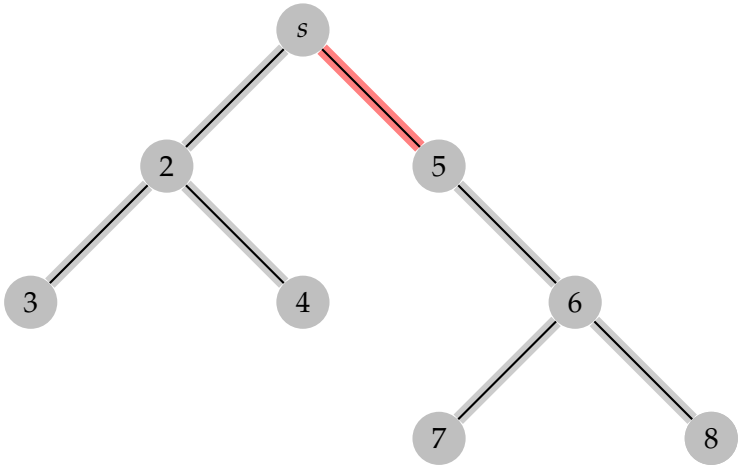
Solution. For each node i, let's consider all the paths with one endpoint at i. Call the sum of the weights of these paths sum_i. Then the answer is

$$\mathbb{E}[\text{path sum}] = \frac{\sum_{i=1}^{n} sum_i}{2}$$

Furthermore, let's start at a node that we call s. Then sum_s can be computed in $O(n)$ time with a BFS/DFS. Now, we need to extract $sum_i \forall i \neq s$ in linear time. This can also be done with a BFS/DFS. Consider the following tree:



Without loss of generality, assume the traversal initially goes from node s to 5. Then we need to find sum_5 from sum_s.



Notice that compared to sum_s, the highlighted edge is added in all of the paths from 5 that end in a node that is not in the subtree of 5. That edge is also removed in all of the paths from 5 to another node in the subtree of 5. Let the subtree size of i be denoted by $size_i$. Then the transition between adjacent nodes $i \to j$ can be written as follows:

$$sum_j = sum_i + w_{i\to j} \cdot (n - size_j) - w_{i\to j} \cdot (size_j)$$

or more concisely

$$sum_j = sum_i + w_{i\to j}(n - size_j - size_j) = \boxed{sum_i + w_{i\to j}(n - 2 \cdot size_j)}$$

□

Solution. There is also another way of doing this with linearity of expectation. Let's call the number of different paths where an edge i appears f_i. The contribution of each edge to the expected value is proportional to its f_i. Then the answer is

$$\mathbb{E}[\text{path sum}] = \frac{\sum_{i=1}^{n} f_i}{n-1}$$

f_i can easily be found by (pre)computing the subtree sizes for each node and doing a traversal of the tree. □

Remark 4.3.1. The ideas mentioned here can be extended to find the

- average length of a path
- sum of all path lengths
- number of paths that go through an edge, for all edges
- the sum of the distances from a node to all other nodes, for each node

for a weighted or unweighted tree in linear time.

Here are a few problems that can be solved with rerooting:

1. USACO Gold Directory Traversal
2. CF 1187E
3. CF 1092F

§4.4 Problems

All of these except the last two should be done in subquadratic time:

1. Find the maximum value path in a weighted tree
2. USACO Gold Barn Painting
3. Find the minimum and maximum distance from each node to a leaf
4. ICPC Brazil 2019-2020 Denouncing Mafia (not exactly DP, but very nice nonetheless)
5. USACO Gold/Platinum Cow At Large
6. Find the sum of XORs of all paths in a tree (hint: consider the contribution of each bit)
7. CF 1528E
8. AtCoder ABC207F
9. USACO Platinum Tree Depth

Remark 4.4.1. A non-DP way of computing the minimum distance from each node to a leaf is doing a multisource BFS from the leaves. This gives the minimum distance to each node from the leaves, which is equivalent to the original problem. The maximum distance to a leaf from each node can also be found without DP by doing a BFS from each endpoint of a diameter. The answer for each node is the maximum distance of the two BFSs for the node.

5 Digits

This topic is not extremely common in competitive programming; however, it is still good to know. Digit DP can be used to solve problems that ask for how many integers in a range have some property. The crux of it lies in the fact that the necessary information for each state does not necessarily have to be an entire integer, but rather some information pertaining to digits. Then since the full information of an integer is not in our state, it turns out that multiple integers can be represented by a single state, which obviously reduces the time complexity. This concept is best explained with an example:

§5.1 Introduction

Problem 5.1.1

There are two integers n, m where $0 \leq n \leq m \leq 10^{18}$. Find the number of integers between n and m inclusive containing exactly k d digits. n, m, k, d are given in input.

```
Sample Input
(n, m, k, d) = (7, 111, 2, 1)
```

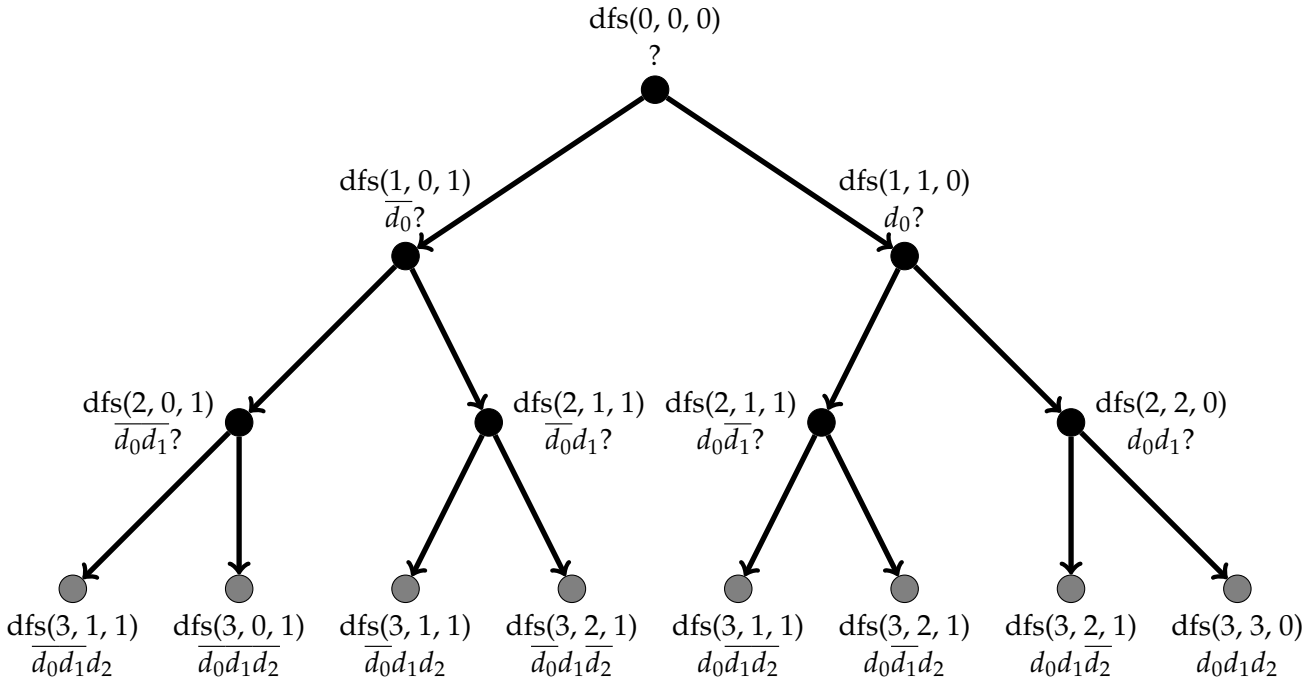
This translates to finding the number of integers in the range $[7, 111]$ with exactly two 1 digits. There are three such integers: $11, 101, 110$.

Solution. Notice that the answer $ans[n, m] = ans[0, m] - ans[0, n-1]$. Now the problem is reduced to solving for the range 0 to c for some c. The naive solution is to iterate through all integers between 0 and c which is at worst $O(10^{18})$. Here, the state is defined by an integer $[0, 10^{18}]$. Call an integer $\leq c$ with exactly k d digits **valid**. Naively, if we consider iterating through the digits instead of the integers, the state is the current digit and all digits already placed. This is also too expensive, so we are motivated to look for a state without the number we are building, but rather with a small amount of data from it.

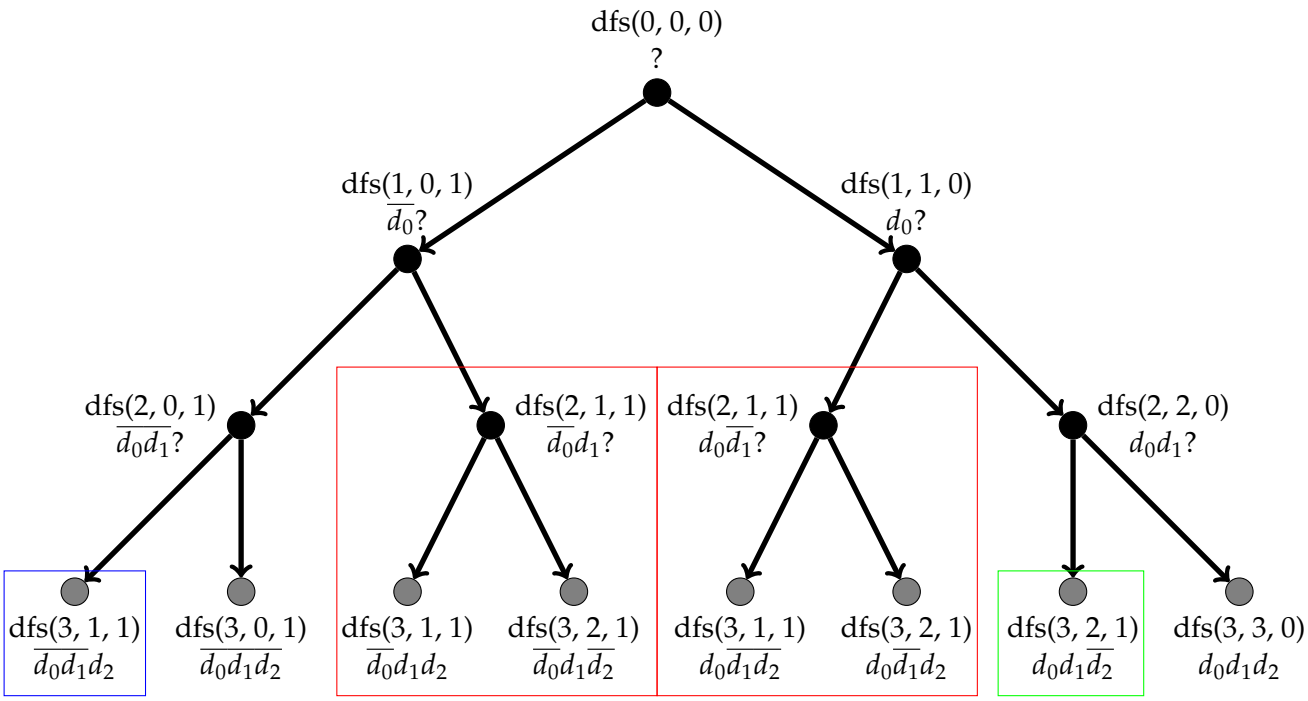
We can instead build the digits of the number left to right, with the state being defined by the position from the left, the number of times k has been placed, and a boolean value indicating whether the number built cannot possibly exceed k. For example, if the number built is $122d_4d_5d_6$ and k is 123222, we know that regardless of what d_4, d_5, d_6 are, the number will always be less than k ... the boolean value is true in this case, otherwise it is false. This technique is very common in digit DP solutions.

Then we can let each state `dp[pos][# times k has been placed before pos][bool]` be the number of valid integers $\leq c$ **from** that state, or in other words, the number of valid integers that satisfy the conditions of that state, ignoring all previous states. So, the answer should be `dp[0][0][0]`.

To properly execute the transitions between the states, a DFS can be done. Let $d_0d_1d_2$ be the digits of 111 (c) and $\overline{d_i}$ be any digit not equal to d_i of c; if the boolean value is false, assume that $\overline{d_i} < c_i$. Furthermore, let ? denote the digit we are currently choosing. Let's see the recursion tree for the aforementioned sample. The base cases are the nodes in the bottommost layer.



A quick look at the tree reveals that some states are repeated. Even if the numbers formed so far in the repeated nodes may be different, they have the same state, and are hence equivalent to each other:



To avoid recomputing them, we can use **memoization**. This can be thought of as the `visited` array in a normal DFS. Memoizing the values of the DFS in a DP table would ensure that no state is visited twice. This means the time (and space) complexity is simply the number of states in the DP table: $O(max_num_digits^2)$. The code below should make this clear:

```cpp
#include<bits/stdc++.h>
using namespace std;
#define int long long

const int max_num_digits = 100;
int dp[max_num_digits][max_num_digits][2];
int n, m, k, d;

void reset(){
    for (int i = 0; i < max_num_digits; i++){
        for (int j = 0; j < max_num_digits; j++){
            dp[i][j][0] = -1;
            dp[i][j][1] = -1;
        }
    }
}

int dfs(int c, int x = 0, int y = 0, bool z = 0){ // range [0, c] with
    // state [x][y][z]
    if (dp[x][y][z] != -1){ // memoization
        return dp[x][y][z];
    }
    dp[x][y][z] = (y == k);
    if (x == to_string(c).length()){
        return dp[x][y][z];
    }
    int limit = 9;
    if (!z){ // if the number being formed CAN exceed c
        limit = to_string(c)[x] - '0'; // the xth digit of c
    }
    // now setting the xth digit of the number we're building
    dp[x][y][z] = 0; // going to permanently set dp[x][y][z] now
    for (int xth_digit = 0; xth_digit <= limit; xth_digit++){
        if (z){
            dp[x][y][z] += dfs(c, x + 1, y + (xth_digit == d), 1);
        }
        else{
            dp[x][y][z] += dfs(c, x + 1, y + (xth_digit == d), xth_digit < limit);
        }
    }
    return dp[x][y][z];
}

signed main(){
    reset();
    cin >> n >> m >> k >> d;
    int total = dfs(m);
    reset();
    cout << total - dfs(n - 1) << endl;
```

```
}
```

Remark 5.1.1. n and m can be made into strings so that large numbers that would normally overflow 64 bit integers can be used. The only drawback to this is that $n-1$ would need to be calculated to call $f(n-1)$. Alternatively, you can use a custom `bignum` object.

Remark 5.1.2. Note that for something like $(n, m, k, d) = (0, 10, 1, 0)$, this code will output 9 since it considers $01, 02, 03 \ldots 09$ to be valid. If valid numbers are not allowed to have leading zeros in their $digit = 0$ count, some casework can be done to eliminate those numbers. One way is replacing `int xth_digit=0` with `int xth_digit=(x==0 && d==0)`.

□

Here is a slightly easier problem that you can try for yourself:

Problem 5.1.3. For some $a, b \leq 10^{15}$, find the sum of the digits of all integers in the range $[a, b]$ (submit)

Problem 5.1.2 (USACO Gold Count the Cows)

There is an infinite two dimensional grid with each cell defined by (x, y) 0-indexed. Each cell in the grid is either 1 or 0. A cell is 1 iff for all integers $k \geq 0$, the remainders when $\left\lfloor \frac{x}{3^k} \right\rfloor$ and $\left\lfloor \frac{y}{3^k} \right\rfloor$ are divided by 3 have the same parity. So, either both are even or both are odd. There are Q queries, each with integers d, x, y, asking how many 1s lie on the diagonal range from (x, y) to $(x+d, y+d)$ inclusive.

$$1 \leq Q \leq 10^4$$

For each query,

$$0 \leq x, y, d \leq 10^{18}$$

Solution

Remark 5.1.4. There are other solutions to this problem (see the official solution), but here, we will specifically focus on the digit DP approach.

$\left\lfloor \frac{x}{3^k} \right\rfloor$ and $\left\lfloor \frac{y}{3^k} \right\rfloor$ in ternary are equivalent to shifting right by k. This means when converted to ternary, the parity of all trits of x and y must be the same if there is a cow at (x, y); call this **valid**. Also note that moving across diagonally means adding the same amount to x and y. This means we can focus specifically on x. From now on, assume x and y refer to their ternary representations.

We want to count the number of $x + \Delta$ and $y + \Delta$ such that $\Delta \leq d$ and the parity of all trits of $x + \Delta$ and $y + \Delta$ are the same. Since we are working in ternary, we are motivated to go from the least to most significant trits. Of course, our state must include the position pos as one of its attributes. Similar to the previous problem, a state stores the number of valid numbers satisfying the constraints of only that state in particular, ignoring the previous states and only focusing on the future. If we are at some state, we can ensure that the previous trits are all valid, that is, the parity of all less-significant trits are the same.

We can also incorporate a flag into our definition of a state. The flag, call it `leq`, denotes whether the suffix of the number $x + \Delta$ we are building, from position pos to the end of $x + \Delta$, is

less than or equal to the corresponding suffix of $x + d$. This necessitates some simple casework. Call the trit candidate we are placing at pos "q":

1. $leq_{new} = true$ when $q < (x + d)_{pos}$
2. $leq_{new} = true$ when $q = (x + d)_{pos}$ and $leq_{old} = true$
3. Otherwise, $leq_{new} = false$

> **Remark 5.1.5.** Consider why keeping leq for x is equivalent to keeping it for y or for both x and y.

There is one more factor we need to consider. Currently, our state is defined by leq and pos. But how do we go from one state to the next? We add some Δ to the trit at position pos of both x and y that we are building. This might result in trits ≥ 3. So we need to keep track of carries on both of the numbers we are building as well.

> **Remark 5.1.6.** Consider why it's never beneficial to increase the trit at some pos by 3 or greater.

If $x < y$, swap x, y. This will give the same answer because of symmetry. The reason this is necessary is to know what the highest set trit is, i.e., where to stop. Our base case is defined by the most significant trit in x. If leq is true and there are no carries, the value of the base is 1. Otherwise, it is 0. The implementation should make everything more clear. Here are the utility functions:

```
namespace util{
    // returns dec->ternary conversion with 0's padded in front to make
    // the length 40
    vector<int> dec_to_ternary(int d){ // approx O(1000)
        string ans = string(40, '0');
        while (d != 0){
            int i = 1, pow = 0;
            while (i * 3 <= d) i *= 3, pow++;
            if (i * 2 <= d) i *= 2, ans[40 - pow - 1] = '2';
            else ans[40 - pow - 1] = '1';
            d -= i;
        }
        vector<int> v;
        for (char c : ans) v.push_back(c - '0');
        return v;
    }
    int carry(int val, int& add){
        if (val + add < 3){
            int addcpy = add; add = 0;
            return val + addcpy;
        }
        val = (val + add) % 3; add = 1;
        return val;
    }
    void reset(){
        for (int i = 0; i < 40; i++)
            for (int j = 0; j < 3; j++)
                for (int k = 0; k < 3; k++)
```

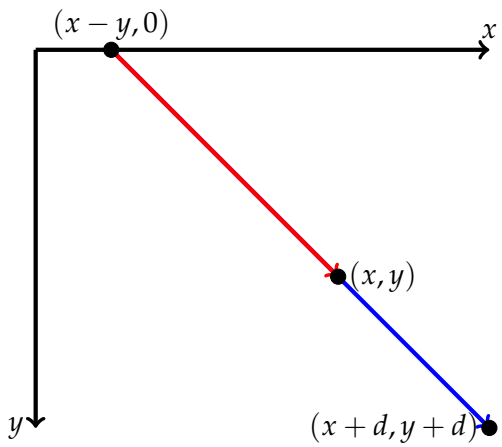
```
                    for (int l = 0; l < 2; l++)
                        dp[i][j][k][l] = -1;
    }
}
using namespace util;
```

and here are the important parts:

```
vector<int> xlim, ylim, xvec, yvec;
int highest_trit = 39; // most significant set trit in x
int dp[40][3][3][2]; // [pos][ xcarry to pos ][ ycarry to pos ]
// [ suffix of x being formed is leq suffix of xlim ]

int dfs(int pos, int carryX, int carryY, bool leq){ // approx O(1000)
    int cx = carryX, cy = carryY;
    int& state = dp[pos][carryX][carryY][leq];
    if (state != -1) return state; // memoize to ensure each state is done
    // at most once
    state = 0; // going to set it permanently now
    if (pos == highest_trit - 1){ // base case
        if(carryX==0) return(state = leq);
        return state;
    }
    int xpos = carry(xvec[pos], carryX);
    int ypos = carry(yvec[pos], carryY);
    array<int, 2> carries = { carryX, carryY };
    for (int add : {0, 1, 2}){  // setting trits at pos of the (x, y) we are
    // building
        carryX = carries[0]; carryY = carries[1];
        int addX = add;
        int newxpos = carry(xpos, addX); // new val of x[pos]
        int addY = add;
        int newypos = carry(ypos, addY); // new val of y[pos]
        if (newxpos % 2 != newypos % 2) continue; // if not same parity,
        // ignore and move on
        carryX += addX, carryY += addY;
        state += dfs(pos - 1, carryX, carryY,
        (newxpos < xlim[pos] || (newxpos == xlim[pos] && leq)));
    }
    return state;
}

signed main(){
    int t; cin >> t;
    while (t--){
        reset(); // fill dp with -1
        int x, y, d; cin >> d >> x >> y;
        if (x < y) swap(x, y); // symmetric, so same result. needed for
        // highest_trit
        xlim = dec_to_ternary(x + d); ylim = dec_to_ternary(y + d);
```

```
        xvec = dec_to_ternary(x); yvec = dec_to_ternary(y);
        for (int i = 0; i < xlim.size(); i++){
            if (xlim[i]){
                highest_trit = i; break;
            }
        }
        cout << dfs(39, 0, 0, 1) << endl;
    }
}
```

The implementation can be simplified slightly by using a prefix sum-styled DP as was done with the previous problem. Assuming $x \geq y$, this would ensure there will be no carries on the y being built. If we denote $f(x, d)$ to be the answer for the original query of $(x, y, d) = (x, 0, d)$, then the answer for a query (x, y, d) is $f(x - y, y + d) - f(x - y, y - 1)$:



§5.2 Problems

1. CF 1036C
2. CodeChef DIGIMU
3. CF 1327E
4. Count the number of integers in the range $[1, n]$ such that each digit $0 \ldots 9$ appears at least once
5. CF 1073E
6. Baltic OI Palindrome-Free Numbers

6 Bitmasking

For problems in which the optimal answer can be naively derived from some permutation of items, bitmask DP can usually be done. The crux of bitmask DP is that the naive $\Omega(n!)$ brute force solution can be reduced to $\Omega(2^n)$ by iterating over subsets instead of permutations. This is useful because $2^n < n \cdot 2^n < n^4 \cdot 2^n < n!$ for all reasonably large n pertaining to competitive programming.

§6.1 Assignment Problem

Problem 6.1.1 (Assignment Problem)

There are n people numbered $1 \ldots n$ who are looking for rides and need to be assigned to n taxis also numbered $1 \ldots n$. There is also a `cost` matrix, where `cost[i][j]` denotes the cost of letting person i ride taxi j. Each person needs to be assigned a taxi, and each taxi must be assigned to exactly one person. What is the minimum total cost.

The naive solution is to try all $n!$ assignments and pick the best one. This obviously takes $O(n!)$ time, which is not good. How can we do better? We can notice that for some subset of size k of taxis that are filled with the first k people, we don't care about how it's filled, but rather the minimum cost of it being filled. This makes sense when considering what we're looking for: the minimum cost of filling the size n subset. More specifically, we don't care about how the people are permuted in a taxi subset consisting of the first k people. This already hints at reducing the time complexity from factorial to exponential.

§6.2 Representing Subsets With Bitmasks

First, you should be familiar with how to represent subsets with bitmasks. This is a common technique when generating subsets. Notice that the only set bit in 2^k is in the kth position from the right. All subsets of the bits from position 0 to $k-1$ are generated when enumerating from 0 to $2^k - 1$. Consider the sets of positions (from the right) of set bits for $k = 3$:

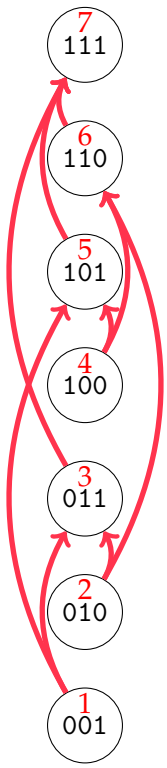
$$
\begin{aligned}
000 &= \{\} \\
001 &= \{0\} \\
010 &= \{1\} \\
011 &= \{0, 1\} \\
100 &= \{2\} \\
101 &= \{0, 2\} \\
110 &= \{1, 2\} \\
111 &= \{0, 1, 2\}
\end{aligned}
$$

§6.3 Full Solution

We can keep an n bit number where each bit denotes whether the corresponding taxi has been taken or not. Our state is defined by the n bit number. If k of the n bits are 1, that means the first k people have been assigned taxis corresponding to the k set bits.

Consider `State k=0: 000` → `State k=1: 100` → `State k=2: 101`. This represents person 1 assigned taxi 1 and person 2 assigned taxi 3. However, we can also attain `State k=2: 101` through `State k=0: 000` → `State k=1: 001` → `State k=2: 101`. This directly shows how permutations are reduced to subsets.

Notice that you only need to iterate over the numbers 1 to n and everything happens automatically:



This is because the arrows are all pointed in the same direction, enabling optimal substructure. Proving this is always the case is left as an exercise for the reader.

Algorithm Assignment Problem

```
array DP ← ∞
for mask ∈ [0, 2^n) do
    nsb ← mask&(2^n − 1)
    for i ∈ [0, n) do
        if mask&(1 << i) == 0 then
            DP[mask|(1 << i)] = min(DP[mask|(1 << i)], cost[nsb][i])
        end
    end
end
return DP[2^n − 1]
```

This takes $O(n \cdot 2^n)$ as opposed to $O(n!)$.

Remark 6.3.1. There are actually faster ways of solving the Assignment Problem, but the bitmask DP solution was described here for demonstration purposes.

§6.4 Problems

1. CSES Hamiltonian Flights
2. CF 16E
3. USACO Gold Guard Mark
4. CF 11D
5. CF 895C
6. COCI Burza

7 Ranges

We can use DP on ranges to solve the following problem.

Problem 7.0.1 (USACO Gold Modern Art 3)

Having become bored with standard 2-dimensional artwork (and also frustrated at others copying her work), the great bovine artist Picowso has decided to switch to a more minimalist, 1-dimensional style. Her latest painting can be described by a 1-dimensional array of colors of length $N(1 \leq N \leq 300)$, where each color is specified by an integer in the range $1 \ldots N$.

To Picowso's great dismay, her competitor Moonet seems to have figured out how to copy even these 1-dimensional paintings! Moonet will paint a single interval with a single color, wait for it to dry, then paint another interval, and so on. Moonet can use each of the N colors as many times as she likes (possibly none).

Please compute the number of such brush strokes needed for Moonet to copy Picowso's latest 1-dimensional painting.

Solution. Let's create a two dimensional DP with the state being defined by the minimum number of strokes to copy a specific range. Then we can iterate over ranges by their size and get the answer for a state by looking at states corresponding to subranges in the current range. Notice that each range can be broken into two ranges, and it is never better to break a range into more than two ranges. The implementation should make this clear. □

```
#include <bits/stdc++.h>
using namespace std;

const int sz=305;
int n;
int dp[sz][sz]; // dp[a][b] is min strokes to copy [a, b]
int p[sz]; // original painting
// one index everything

signed main(){
    cin>>n;
    for(int i=1; i<=n; i++) cin>>p[i];
    for(int i=0; i<=n; i++){
        for(int j=0; j<=n; j++){
            if(i==j) dp[i][j]=1;
            else dp[i][j]=sz;
        }
        if(i!=n) dp[i][i+1]=1+(p[i]!=p[i+1]);
    }

    for(int gap=2; gap<n; gap++){
        for(int pos=1; pos+gap<=n; pos++){
            dp[pos][pos+gap]=dp[pos+1][pos+gap-1]+(p[pos]==p[pos+gap] ? 1 : 2);
```

```
            for(int point=pos+1; point<n; point++){ // inside the gap
                dp[pos][pos+gap]=min(dp[pos][pos+gap],
                dp[pos][point]+dp[point][pos+gap]-1);
            }
        }
    }
    cout<<dp[1][n]<<endl;
}
```

Remark 7.0.1. Consider why `dp[pos][point]+dp[point+1][pos+gap]` does not work, but `dp[pos][point]+dp[point][pos+gap]-1` does.

Solution. Consider `1 1 2 1` when `dp[1][4]` is being calculated. Initially, `dp[1][4]=dp[2][3]+1=3`. The former does not take into account the `1` being already present in `[2, 3]`, but the latter does. The latter would do `dp[1][4]=min(dp[1][4], dp[1][2]+dp[2][4]-1)=2`. Although somewhat unintuitive, it turns out that the latter always works. □

II

Parting Shots

8 Problems

It is equally important to know DP topics as it is to recognize when and how to apply them. The best way to get better at DP problems is by solving them. Below, I leave you with a variety of carefully chosen, high quality problems arranged in three sets of ten. The problems in each of the sets are in approximate ascending order of difficulty. Most of these problems are from Codeforces, but there are also a few nice problems from other sources.

Set 1

1. USACO Gold Hoof, Paper, Scissors
2. CF 118D
3. CF 101B
4. CF 1133E
5. USACO Platinum Team Building
6. USACO Platinum Circular Barn
7. CF 1551F
8. CF 1327F
9. SWERC 2020 Mentors
10. CF 449D

Set 2

1. CF 1535C
2. CF 1538C
3. USACO Gold Radio Contact
4. CF 69D
5. CF 1472G
6. CF 1067A
7. CF 1426F
8. IOI 2020 Jelly
9. USACO Platinum Balanced Subsets
10. CF 1188C

Set 3

1. CF 1534C
2. CF 1513C
3. CF 1525D
4. CF 118D
5. USACO Gold Snakes
6. CF 1552F
7. CF 123C
8. UTS Open '21 P3
9. CF 1541E1
10. CF 1540C2

9 Resources

Here, I leave some useful resources for you:

1. CP Algorithms
2. Codeforces DP Problems
3. USACO Guide
4. Codeforces DP Blogs
5. Brian Dean's DP list