CSES Notes

Coordinate Compression

```
vl temp = v;
sort(temp.begin(), temp.end());
temp.resize(unique(temp.begin(), temp.end()) - temp.begin());
for (ll i = 0; i < n; i++)
    v[i] = lower_bound(temp.begin(), temp.end(), v[i]) - temp.begin();
// Alternate way of coordinate compression (slower)
// map<ll, ll> m;
// for (ll i = 0; i < n; i++)
//     m[temp[i]] = i;
// for (ll i = 0; i < n; i++)
//     v[i] = m[v[i]];
```

Maths - Fibonacci Numbers :

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

See next q for matrix exp.

Using this we can derive :

- If n is even then k = n/2: Nth Fibonacci Number = F(n) = [2*F(k-1) + F(k)] * F(k)
- If n is odd then k = (n + 1)/2: Nth Fibonacci Number = F(n) = F(k)* F(k) + F(k-1)* F(k-1)

Code snippet :

```
const int MOD = 1e9 + 7;
map<ll, ll> fib; // you can't make an array of size 1e18
ll f(ll n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    if (fib.find(n) != fib.end())
        return fib[n];
    if (n % 2)
```

```
    {
        ll k = (n + 1) / 2;
        ll a = f(k) % MOD;
        ll b = f(k - 1) % MOD;
        return fib[n] = ((a * a) % MOD + (b * b) % MOD) % MOD;
    }
    else
    {
        ll k = n / 2;
        ll a = f(k) % MOD;
        ll b = f(k - 1) % MOD;
        return fib[n] = (a * ((a + (2 * b % MOD)) % MOD)) % MOD;
    }
    return -1;
}
```

Maths - Throwing Dice :

Matrix Exponentiation :

Any linear recurrence can be expressed as a matrix * vector leading to a new vector.

That also holds true for any dp you can solve in O(1) space (generally).

So if next term comes by Ax = b, then you can get nth term by A^n*x.

You can get A^n quickly by matrix exponentiation, ie finding power just like you do for bin_exp. This speeds up from O(n) to O(lg n).

So reform your dp, write it as transformation matrix, and use matrix exp to speed it up.

One nice way to do that is formulate such that : new_dp[i] += dp[j] * a[i][j]

If your new state depends on k values, matrix will be of k * k and vector of k * 1. So to be more precise : The matrix multiplication algorithm will have a complexity of O(k^3). Hence, the overall complexity turns out be O(k^3 * log(n)).

To efficiently calculate Fibonacci numbers, we represent the Fibonacci formula as a square matrix X of size 2×2, for which the following holds:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Thus, values $f(i)$ and $f(i+1)$ are given as "input" for X, and X calculates values $f(i+1)$ and $f(i+2)$ from them. It turns out that such a matrix is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Thus, we can calculate $f(n)$ using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The value of X^n can be calculated in $O(\log n)$ time, so the value of $f(n)$ can also be calculated in $O(\log n)$ time.

General case

Let us now consider the general case where $f(n)$ is any linear recurrence. Again, our goal is to construct a matrix X for which

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Such a matrix is

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

In the first $k-1$ rows, each element is 0 except that one element is 1. These rows replace $f(i)$ with $f(i+1)$, $f(i+1)$ with $f(i+2)$, and so on. The last row contains the coefficients of the recurrence to calculate the new value $f(i+k)$.

Now, $f(n)$ can be calculated in $O(k^3 \log n)$ time using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

Code Snippet :

```cpp
// Matrix Exp template
const int MOD = 1e9 + 7;
using Matrix = array<array<ll, 2>, 2>; // change 2 to order of matrix
Matrix mul(Matrix a, Matrix b)
{
    Matrix res = {{{0, 0}, {0, 0}}};
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++)
            {
                res[i][j] += a[i][k] * b[k][j];
                res[i][j] %= MOD;
            }
    return res;
}

Matrix mat_exp(Matrix a, ll b)
{
    Matrix res = {{{1, 0}, {0, 1}}};
    while (b > 0)
    {
        if (b & 1)
            res = mul(res, a);
        a = mul(a, a);
        b >>= 1;
    }
    return res;
}

// Evergreen Fibonacci
int main()
{
    ll n;
    cin >> n;
    ll dp[2] = {0, 1};
    Matrix a = {{{0, 1}, {1, 1}}};
    a = mat_exp(a, n);
    ll new_dp[2] = {0, 0};
    for (ll i = 0; i < 2; i++)
        for (ll j = 0; j < 2; j++)
            new_dp[i] += dp[j] * a[i][j];
    cout << new_dp[0];
    return 0;
}
```

Linear Recurrence for Throwing Dice:

f(n) = sum(f(n-i)) : i - 1 to 6 for n>=6

`math/throwing_dice.cpp`

Note: No need to enforce n>=6 condition since before that the multiplication only shifts vector elements and hence the first element still carries the right answer.

Maths - Graph Paths I :

A directed, unweighted graph with n vertices and m edges and we are given an integer k. For each pair of vertices (i, j) we have to find the number of paths of length k between these vertices. Paths don't have to be simple, i.e. vertices and edges can be visited any number of times in a single path.

No of paths of length k from i to j = A^k[i][j], where A is adjacency matrix of the graph.

Total no of paths of length k -> Sum over all values of A^k[i][j].

So just do mat_exp of adjacency matrix A to power k.

See AC CSES submission if any doubt.

DP - Counting Numbers

Digit DP

- Allows us to solve mainly two types of problems:
  1. Count of no of integers X in [0, R] such that f(X) = true (where f(X) is a function which only returns true or false)

  For small R, you can iterate over all integers from 0 to R but what about R upto 1e18 or larger.

  In that case you will make use of properties of digits using digit DP.

  2. Count of no of integers X in [L, R] such that f(X) = true : Very easy if you did the first and got ans as S(R), then ans for second is S(R) - S(L-1).
- Hence, 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.

- L and R can be made into strings so that large numbers that would normally overflow 64 bit integers can be used. Just that you will need to find L-1.
- IMP points of almost every digit DP question :
  - You will build number digit by digit from left to right
  - Digits that can be placed vary from 0 to (actual digit) or 9 –> decided by *tight* constraint
  - Some valid numbers may be skipped due to leading zeroes (like in our question), to avoid that we carry a boolean - *started* to see if the number has started or are we still in the leading zeroes phase.
  - Propagating constraints : (tight && (dig == ub)), (started || (dig != 0)), ie, if already tight and dig is upper bound then tight remains true else becomes false. Once started becomes true it remains true, or if it was false but we got our first non zero digit then started is true.
  - Memoization tends to be easier and a better choice.
- For more insight into general digit DP and approach for this question in particular : Read code, comments and this link : https://codeforces.com/blog/entry/111675

DP - Projects

Let dp[i] denote optimal reward for projects (p1, p2, ... pi)

Then our required ans = dp[n]. Base case : dp[0] = 0

Recurrence :

```
dp[i] = max(dp[i-1], reward[i] + dp[j])
        where pj denotes first project before pi that you can do
        ie : j such that e[j]<s[i] and e[j] is largest
```

Clearly, you can easily get j as lower bound of s[i] over sorted ending times.

T.C : O(n*lg n)

Maths - Games :

Nim and Grundy

Read CPH chapter on Game Theory and this link :

https://letuskode.blogspot.com/2014/08/grundy-numbers.html

Maths - Stick Game

Reducing it to a single pile using Grundy's numbers, we get this code :

```
ll mex(set<ll> a)
{
```

```
    ll result = 0;
    while (a.count(result))
        ++result;
    return result;
}
```

```
    vl dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;
    for (ll i = 2; i <= n; i++)
    {
        set<ll> a;
        for (auto &x : v)
        {
            if (i >= x)
                a.insert(dp[i - x]);
        }
        dp[i] = mex(a);
    }
    for (ll i = 1; i <= n; i++)
    {
        if (dp[i] > 0)
            cout << "W";
        else
            cout << "L";
    }
```

But realise that you are only checking if the Grundy no is zero or not. For that you don't actually need to find mex.

MEX will be zero (L) only if you never encounter a zero or (L) state. Otherwise if you encounter even a single zero (L) state, your MEX must be >0 and hence a W.

So you can do that in this:

```
    vector<bool> dp(n + 1, 0);
    dp[0] = 0;
    dp[1] = 1;
    for (ll i = 2; i <= n; i++)
    {
        for (auto &x : v)
        {
            if (i >= x)
            {
                if (!dp[i - x])
                {
                    dp[i] = 1;
```

```
                break;
            }
        }
    }
}
for (ll i = 1; i <= n; i++)
{
    if (dp[i])
        cout << "W";
    else
        cout << "L";
}
```

Maths - Nim Game I

Classic Nim Game : Nim sum == 0, second wins else first.

Maths - Nim Game II

Like described in the blog, we take nim sum of grundy numbers which are v[i]%4. If it's zero then second else first.

Maths - Stair Game

We consider elements at even position and proceed as Nim.

Some intuition - to empty a pile at position k, we will have to empty it all the way to position 1, which means there are (k-1) copies of that pile which we have to empty. So if k is odd, xor of k even number of times is 0, so the odd positions don't create an effect, and hence we consider the even positions only.

Maths - Grundy's Game

Classic Grundy's game. TC : O(n^2 + t)

To pass it under 1 sec, we use the fact that : for larger values of n (n>=2000), its Sprague-Grundy value is never 0.

So we have to compute the nos till 2000 and beyond them we can be assured that first wins. So TC : O(t + 4e6)

Code:

```
ll mex(set<ll> a)
{
    ll result = 0;
    while (a.count(result))
        ++result;
    return result;
```

```
}

void solve()
{
    vl dp(2001);
    dp[1] = 0;
    dp[2] = 0;
    for (ll i = 3; i <= 2000; i++)
    {
        set<ll> a;
        for (ll j = 1; j <= i / 2; j++)
        {
            if (j != i - j)
                a.insert(dp[j] ^ dp[i - j]);
        }
        dp[i] = mex(a);
    }
    ll n, t;
    cin >> t;
    while (t--)
    {
        cin >> n;
        if (n > 2000 || dp[n] > 0)
            cout << "first\n";
        else
            cout << "second\n";
    }
}
```