


When submitting a solution in C++, please select either C++14 (GCC 6-32) or C++17 (GCC 7-32) as your compiler. ✕

[ICECUBER](#) [BLOG](#) [TEAMS](#) [SUBMISSIONS](#) [GROUPS](#) [CONTESTS](#)

icecuber's blog

CSSES DP section editorial

By [icecuber](#), 4 years ago, 

Edit: More dp tasks have been added to CSSES since this blog was created. For solutions to those problems, check out [UnexpectedValue](#)'s blog [here](#).

I'm using bottom-up implementations and pull dp when possible. Pull dp is when we calculate each dp entry as a function of previously calculated dp entries. This is the way used in recursion / memoization. The other alternative would be push dp, where we update future dp entries using the current dp entry.

I think [CSSES](#) is a nice collection of important CP problems, and would like it to have editorials. Without editorials users will get stuck on problems, and give up without learning the solution. I think this slows down learning significantly compared to solving problems with editorials. Therefore, I encourage others who want to contribute, to write editorials for other sections of CSSES.

Feel free to point out mistakes.

Dice Combinations (1633)

$dp[x]$ = number of ways to make sum x using numbers from 1 to 6.

Sum over the last number used to create x , it was some number between 1 and 6. For example, the number of ways to make sum x ending with a 3 is $dp[x-3]$. Summing over the possibilities gives $dp[x] = dp[x-1] + dp[x-2] + dp[x-3] + dp[x-4] + dp[x-5] + dp[x-6]$.

We initialize by $dp[0] = 1$, saying there is one way with sum zero (the empty set).

The complexity is $O(n)$.

Code

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

int main() {
    int mod = 1e9+7;
    int n;
    cin >> n;
    vector<int> dp(n+1,0);
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= 6 && i-j >= 0; j++) {
            (dp[i] += dp[i-j]) %= mod;
        }
    }
    cout << dp[n] << endl;
}
```

Minimizing Coins (1634)

→ [Pay attention](#)

Before contest
[Codeforces Round 936 \(Div. 2\)](#)
 45:13:04

→ [Streams](#)

[CodeChef Starters 126 Solution Discussion](#)
 By [aryanc403](#)
 Before stream 23:23:04

[View all →](#)

→ [_Nongdamba_](#)

Rating: **1527**
 Contribution: 0

- [Settings](#)
- [Blog](#)
- [Teams](#)
- [Submissions](#)
- [Talks](#)
- [Contests](#)



[_Nongdamba_](#)

→ [Top rated](#)

#	User	Rating
1	jiangly	3640
2	Benq	3593
3	tourist	3572
4	orzdevinwang	3561
5	cnnfls_csy	3539
6	ecnerwala	3534
7	Radewoosh	3532
8	gyh20	3447
9	Rebelz	3409
10	Geothermal	3408

[Countries](#) | [Cities](#) | [Organizations](#)

[View all →](#)

→ [Top contributors](#)

#	User	Contrib.
1	maomao90	173
2	adamant	164
3	awoo	162
4	TheScrasse	160
5	nor	159
6	maroonrk	156
7	SecondThread	154
8	pajenegod	147
9	BledDest	145

This is a classical problem called the **unbounded knapsack problem**.

$dp[x]$ = minimum number of coins with sum x .

We look at the last coin added to get sum x , say it has value v . We need $dp[x-v]$ coins to get value $x-v$, and 1 coin for value v . Therefore we need $dp[x-v]+1$ coins if we are to use a coin with value v . Checking all possibilities for v must include the optimal choice of last coin.

As an implementation detail, we use $dp[x] = 1e9 = 10^9 \approx \infty$ to signify that it is not possible to make value x with the given coins.

The complexity is $O(n \cdot target)$.

Code

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

int main() {
    int n, target;
    cin >> n >> target;
    vector<int> c(n);
    for (int&v : c) cin >> v;

    vector<int> dp(target+1, 1e9);
    dp[0] = 0;
    for (int i = 1; i <= target; i++) {
        for (int j = 0; j < n; j++) {
            if (i-c[j] >= 0) {
                dp[i] = min(dp[i], dp[i-c[j]]+1);
            }
        }
    }
    cout << (dp[target] == 1e9 ? -1 : dp[target]) << endl;
}
```

Coin Combinations I (1635)

This problem has a very similar implementation to the previous problem.

$dp[x]$ = number of ways to make value x .

We initialize $dp[0] = 1$, saying the empty set is the only way to make 0.

Like in "Minimizing Coins", we loop over the possibilities for last coin added. There are $dp[x-v]$ ways to make x , when adding a coin with value v last. This is since we can choose any combination for the first coins to sum to $x-v$, but need to choose v as the last coin. Summing over all the possibilities for v gives $dp[x]$.

The complexity is $O(n \cdot target)$.

Code

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

int main() {
    int mod = 1e9+7;
    int n, target;
    cin >> n >> target;
    vector<int> c(n);
    for (int&v : c) cin >> v;

    vector<int> dp(target+1, 0);
    dp[0] = 1;
    for (int i = 1; i <= target; i++) {
```

9	Um_nik	145
View all →		

→ Find user

Handle:

Find

→ Recent actions

maroonrk → [AtCoder Regular Contest 174 Announcement](#)

Kirill_Maglysh → [Codeforces Round #935 \(Div. 3\)](#)

mohamedhesham → [Contest Time](#)

yunitive29 → [Codeforces Round #936 \(Div. 2\)](#)

huikang → [Cognition, competitive programmers, and Devin, the first AI software engineer](#)

chenjb → [The 2nd Universal Cup Semifinal & 2024 Summer Summit Announcement](#)

Dominator069 → [Codeforces Round #934 \(Div1, Div2\)](#)

Norp → [Pupil in 80 days - Day 6](#)

Dominator069 → [Codeforces Round #934 \(Div1, Div2\) Editorial](#)

transfermarket → [Unexpected AC](#)

ReplyChallenges → [Reply Code Challenge 2024 — Teen & Standard Edition](#)

coderdhanraj → [Invitation to Insomnia Qualifier 2024](#)

Zhtluo → [All You Need is Randomly Guessing — How to Improve at Codeforces](#)

Norp → [What should I do?](#)

vedant_vaidya_77 → [CP TITANS 5.0 : The Code Heist](#)

I_love_Leyli_Meredova → [How fast do you type?](#)

MuhammadSawalhy → [String hashing collision, multiple bases don't work, weird behavior with 1944D](#)

satyam343 → [think-cell Round 1](#)

Ormlis → [Codeforces Round #879 Editorial](#)

MikeMirzayanov → [Rule about third-party code is changing.](#)

E869120 → [JOI Spring Training 2024 Online Contest](#)

Friren → [Hi guys! Please help me to understand this code!](#)

Arshavir → [I have nothing to say :/](#)

limbo16 → [The Curse of Segment Tree](#)

rhymehatch → [Problem F — Random Walk — Codeforces Round #868](#)

[Detailed →](#)

```

for (int j = 0; j < n; j++) {
    if (i-c[j] >= 0) {
        (dp[i] += dp[i-c[j]]) %= mod;
    }
}
}
cout << dp[target] << endl;
}

```

Coin Combinations II (1636)

$dp[i][x]$ = number of ways to pick coins with sum x , using the first i coins.

Initially, we say we have $dp[0][0] = 1$, i.e we have the empty set with sum zero.

When calculating $dp[i][x]$, we consider the i 'th coin. Either we didn't pick the coin, then there are $dp[i-1][x]$ possibilities. Otherwise, we picked the coin. Since we are allowed to pick it again, there are $dp[i][x - \text{value of } i\text{'th coin}]$ possibilities (not $dp[i-1][x - \text{value of } i\text{'th coin}]$ possibilities).

Because we consider the coins in order, we will only count one order of coins. This is unlike the previous task, where we considered every coin at all times.

The complexity is $O(n \cdot \text{target})$.

Code

```

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

int main() {
    int mod = 1e9+7;
    int n, target;
    cin >> n >> target;
    vector<int> x(n);
    for (int&v : x) cin >> v;

    vector<vector<int>> dp(n+1, vector<int>(target+1, 0));
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= target; j++) {
            dp[i][j] = dp[i-1][j];
            int left = j-x[i-1];
            if (left >= 0) {
                (dp[i][j] += dp[i][left]) %= mod;
            }
        }
    }
    cout << dp[n][target] << endl;
}

```

Removing Digits (1637)

$dp[x]$ = minimum number of operations to go from x to zero.

When considering a number x , for each digit in the decimal representation of x , we can try to remove it. The transition is therefore: $dp[x] = \min_{d \in \text{digits}(x)} dp[x-d]$.

We initialize $dp[0] = 0$.

The complexity is $O(n)$.

Note that the greedy solution of always subtracting the maximum digit is also correct, but we are practicing DP :)

Code



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

int main() {
    int n;
    cin >> n;
    vector<int> dp(n+1, 1e9);
    dp[0] = 0;
    for (int i = 0; i <= n; i++) {
        for (char c : to_string(i)) {
            dp[i] = min(dp[i], dp[i-(c-'0')]+1);
        }
    }
    cout << dp[n] << endl;
}
```

Grid Paths (1638)

$dp[r][c]$ = number of ways to reach row r , column c .

We say there is one way to reach (0,0), $dp[0][0] = 1$.

When we are at some position with a , we came either from the left or top. So the number of ways to get to there is the number of ways to get to the position above, plus the number of ways to get to the position to the left. We also need to make sure that the number of ways to get to any position with a  is 0.

The complexity is $O(n^2)$, so linear in the number of cells of input.

Code

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

int main() {
    int mod = 1e9+7;
    int n;
    cin >> n;
    vector<vector<int>> dp(n, vector<int>(n, 0));
    dp[0][0] = 1;
    for (int i = 0; i < n; i++) {
        string row;
        cin >> row;
        for (int j = 0; j < n; j++) {
            if (row[j] == '.') {
                if (i > 0) {
                    (dp[i][j] += dp[i-1][j]) %= mod;
                }
                if (j > 0) {
                    (dp[i][j] += dp[i][j-1]) %= mod;
                }
            } else {
                dp[i][j] = 0;
            }
        }
    }
    cout << dp[n-1][n-1] << endl;
}
```

Book Shop (1158)

This is a case of the classical problem called **0-1 knapsack**.

$dp[i][x]$ = maximum number of pages we can get for price at most x , only buying among the first i books.

Initially $dp[0][x] = 0$ for all x , as we can't get any pages without any books.

When calculating $dp[i][x]$, we look at the last considered book, the i 'th book. We either didn't buy it, leaving x money for the first $i-1$ books, giving $dp[i-1][x]$ pages. Or we bought it, leaving $x - price[i-1]$ money for the other $i-1$ books, and giving $pages[i-1]$ extra pages from the bought book. Thus, buying the i 'th book gives $dp[i-1][x - price[i-1]] + pages[i-1]$ pages.

The complexity is $O(n \cdot x)$.

Code

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

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> price(n), pages(n);
    for (int&v : price) cin >> v;
    for (int&v : pages) cin >> v;
    vector<vector<int>> dp(n+1, vector<int>(x+1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= x; j++) {
            dp[i][j] = dp[i-1][j];
            int left = j - price[i-1];
            if (left >= 0) {
                dp[i][j] = max(dp[i][j], dp[i-1][left] + pages[i-1]);
            }
        }
    }
    cout << dp[n][x] << endl;
}
```

Array Description (1746)

$dp[i][v]$ = number of ways to fill the array up to index i , if $x[i] = v$.

We treat $i = 0$ separately. Either $x[0] = 0$, so we can replace it by anything (i.e $dp[0][v] = 1$ for all v). Otherwise $x[0] = v \neq 0$, so that $dp[0][v] = 1$ is the only allowed value.

Now to the other indices $i > 0$. If $x[i] = 0$, we can replace it by any value. However, if we replace it by v , the previous value must be either $v-1$, v or $v+1$. Thus the number of ways to fill the array up to i , is the sum of the previous value being $v-1$, v and $v+1$. If $x[i] = v$ from the input, only $dp[i][v]$ is allowed (i.e $dp[i][j] = 0$ if $j \neq v$). Still $dp[i][v] = dp[i-1][v-1] + dp[i-1][v] + dp[i-1][v+1]$.

The complexity is $O(n \cdot m)$ with worst-case when x is all zeros.

Code

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

int main() {
    int mod = 1e9+7;
    int n, m;
    cin >> n >> m;
    vector<vector<int>> dp(n, vector<int>(m+1, 0));
    int x0;
    cin >> x0;
    if (x0 == 0) {
        fill(dp[0].begin(), dp[0].end(), 1);
    } else {

```

```

    dp[0][x0] = 1;
}
for (int i = 1; i < n; i++) {
    int x;
    cin >> x;
    if (x == 0) {
        for (int j = 1; j <= m; j++) {
            for (int k : {j-1, j, j+1}) {
                if (k >= 1 && k <= m) {
                    (dp[i][j] += dp[i-1][k]) %= mod;
                }
            }
        }
    } else {
        for (int k : {x-1, x, x+1}) {
            if (k >= 1 && k <= m) {
                (dp[i][x] += dp[i-1][k]) %= mod;
            }
        }
    }
}
int ans = 0;
for (int j = 1; j <= m; j++) {
    (ans += dp[n-1][j]) %= mod;
}
cout << ans << endl;
}

```

Edit Distance (1639)

This is a classic problem called **edit distance**.

We call the input strings a and b , and refer to the first i characters of a by $a[:i]$.

$dp[i][k]$ = minimum number of moves to change $a[:i]$ to $b[:k]$.

When we calculate $dp[i][k]$, there are four possibilities to consider for the rightmost operation. We check all of them and take the cheapest one.

1. We deleted character $a[i-1]$. This took one operation, and we still need to change $a[:i-1]$ to $b[:k]$. So this costs $1 + dp[i-1][k]$ operations.
2. We added character $b[k-1]$ to the end of $a[:i]$. This took one operation, and we still need to change $a[:i]$ to $b[:k-1]$. So this costs $1 + dp[i][k-1]$ operations.
3. We replaced $a[i-1]$ with $b[k-1]$. This took one operation, and we still need to change $a[:i-1]$ to $b[:k-1]$. So this costs $1 + dp[i-1][k-1]$ operations.
4. $a[i-1]$ was already equal to $b[k-1]$, so we just need to change $a[:i-1]$ to $b[:k-1]$. That takes $dp[i-1][k-1]$ operations. This possibility can be viewed as a replace operation where we don't actually need to replace $a[i-1]$.

The complexity is $O(|a| \cdot |b|)$.

Code

```

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

int main() {
    string a, b;
    cin >> a >> b;
    int na = a.size(), nb = b.size();
    vector<vector<int>> dp(na+1, vector<int>(nb+1, 1e9));
    dp[0][0] = 0;
}

```

```

for (int i = 0; i <= na; i++) {
    for (int j = 0; j <= nb; j++) {
        if (i) {
            dp[i][j] = min(dp[i][j], dp[i-1][j]+1);
        }
        if (j) {
            dp[i][j] = min(dp[i][j], dp[i][j-1]+1);
        }
        if (i && j) {
            dp[i][j] = min(dp[i][j], dp[i-1][j-1]+(a[i-1] != b[j-1]));
        }
    }
}
cout << dp[na][nb] << endl;
}

```

Rectangle Cutting (1744)

$dp[w][h]$ = minimum number of cuts needed to cut a $w \times h$ piece into squares.

Consider a $w \times h$ piece. If it is already square ($w = h$), we need 0 cuts. Otherwise, we need to make the first cut either horizontally or vertically. Say we make it horizontally, then we can cut at any position $1, 2, \dots, h-1$. If we cut at position k , then we are left with two pieces of sizes $w \times k$ and $w \times h - k$. We can look up the number of moves to reduce these to squares in the dp array. We loop over all possibilities k and take the best one. Similarly for vertical cuts.

The complexity is $O(a^2 \cdot b + a \cdot b^2)$.

Code

```

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

int main() {
    int w, h;
    cin >> w >> h;
    vector<vector<int>> dp(w+1, vector<int>(h+1));
    for (int i = 0; i <= w; i++) {
        for (int j = 0; j <= h; j++) {
            if (i == j) {
                dp[i][j] = 0;
            } else {
                dp[i][j] = 1e9;
                for (int k = 1; k < i; k++) {
                    dp[i][j] = min(dp[i][j], dp[k][j]+dp[i-k][j]+1);
                }
                for (int k = 1; k < j; k++) {
                    dp[i][j] = min(dp[i][j], dp[i][k]+dp[i][j-k]+1);
                }
            }
        }
    }
    cout << dp[w][h] << endl;
}

```

Money Sums (1745)

This is a case of the classical problem called **0-1 knapsack**.

$dp[i][x]$ = true if it is possible to make x using the first i coins, false otherwise.

It is possible to make x with the first i coins, if either it was possible with the first $i-1$ coins, or we chose the i 'th coin, and it was possible to make $x - \text{value of } i\text{'th coin}$ using the first $i-1$ coins.

Note that we only need to consider sums up to $1000 \cdot n$, since we can't make more than that using n coins of value ≤ 1000 .

The complexity is $O(n^2 \cdot \max x_i)$.

Code

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

int main() {
    int n;
    cin >> n;
    int max_sum = n*1000;
    vector<int> x(n);
    for (int&v : x) cin >> v;
    vector<vector<bool>> dp(n+1, vector<bool>(max_sum+1, false));
    dp[0][0] = true;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= max_sum; j++) {
            dp[i][j] = dp[i-1][j];
            int left = j-x[i-1];
            if (left >= 0 && dp[i-1][left]) {
                dp[i][j] = true;
            }
        }
    }

    vector<int> possible;
    for (int j = 1; j <= max_sum; j++) {
        if (dp[n][j]) {
            possible.push_back(j);
        }
    }
    cout << possible.size() << endl;
    for (int v : possible) {
        cout << v << ' ';
    }
    cout << endl;
}
```

Removal Game (1097)

The trick here is to see that since the sum of the two players' scores is the sum of the input list, player 1 tries to maximize $score_1 - score_2$, while player 2 tries to minimize it.

$dp[l][r] = \text{difference } score_1 - score_2$ if considering the game played only on interval $[l, r]$.

If the interval contains only one element ($l = r$), then the first player must take that element. So $dp[l][l] = x[l]$.

Otherwise, player 1 can choose to take the first element or the last element. If he takes the first element, he gets $x[l]$ points, and we are left with the interval $[l+1, r]$, but with player 2 starting. $score_1 - score_2$ on interval $[l+1, r]$ is just $dp[l+1][r]$ if player 1 starts. Since player 2 starts, it is $-dp[l+1][r]$. Thus, the difference of scores will be $x[l] - dp[l+1][r]$ if player 1 chooses the first element. Similarly, it will be $x[r] - dp[l][r-1]$ if he chooses the last element. He always chooses the maximum of those, so $dp[l][r] = \max(x[l] - dp[l+1][r], x[r] - dp[l][r-1])$.

In this problem $dp[l][r]$ depends on $dp[l+1][r]$, and therefore we need to compute larger l before smaller l . We do it by looping through l from high to low. r still needs to go from low to high,

since we depend only on smaller r ($dp[l][r]$ depends on $dp[l][r-1]$). Note that in all the other problems in this editorial, dp only depends on smaller indices (like $dp[x]$ depending on $dp[x-v]$, or $dp[i][x]$ depending on $dp[i-1][x]$), which means looping through indices in increasing order is correct.

We can reconstruct the score of player 1 as the mean of, the sum of all input values, and $score_1 - score_2$.

The complexity is $O(n^2)$.

Code

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

int main() {
    int n;
    cin >> n;
    vector<int> x(n);
    long long sum = 0;
    for (int&v : x) {
        cin >> v;
        sum += v;
    }

    vector<vector<long long>> dp(n, vector<long long>(n));
    for (int l = n-1; l >= 0; l--) {
        for (int r = l; r < n; r++) {
            if (l == r) {
                dp[l][r] = x[l];
            } else {
                dp[l][r] = max(x[l] - dp[l+1][r],
                               x[r] - dp[l][r-1]);
            }
        }
    }
    cout << (sum + dp[0][n-1]) / 2 << endl;
}
```

Two Sets II (1093)

This is a 0-1 knapsack in disguise. If we are to have two subsets of equal sum, they must sum to half the total sum each. This means if the total sum $\frac{n(n+1)}{2}$ is odd, the answer is zero (no possibilities). Otherwise we run 0-1 knapsack to get the number of ways to reach $\frac{n(n+1)}{4}$ using subsets of the numbers $1..n-1$. Why $n-1$? Because by only considering numbers up to $n-1$, we always put n in the second set, and therefore only count each pair of sets once (otherwise we count every pair of sets twice).

$dp[i][x]$ = number of ways to make sum x using subsets of the numbers $1..i$

We say there is one way (the empty set) to make sum 0, so $dp[0][0] = 1$;

For counting number of ways to make sum x using values up to i , we consider the number i . Either we didn't include it, then there are $dp[i-1][x]$ possibilities, or we included it, and there are $dp[i-1][x-i]$ possibilities. So $dp[i][x] = dp[i-1][x] + dp[i-1][x-i]$.

The complexity is $O(n^3)$.

Code

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

int main() {
```

```

int mod = 1e9+7;
int n;
cin >> n;
int target = n*(n+1)/2;
if (target%2) {
    cout << 0 << endl;
    return 0;
}
target /= 2;

vector<vector<int>> dp(n,vector<int>(target+1,0));
dp[0][0] = 1;
for (int i = 1; i < n; i++) {
    for (int j = 0; j <= target; j++) {
        dp[i][j] = dp[i-1][j];
        int left = j-i;
        if (left >= 0) {
            (dp[i][j] += dp[i-1][left]) %= mod;
        }
    }
}
cout << dp[n-1][target] << endl;
}

```

Increasing Subsequence (1145)

This is a classical problem called **Longest Increasing Subsequence** or LIS for short.

$dp[x]$ = minimum ending value of an increasing subsequence of length $x+1$, using the elements considered so far.

We add elements one by one from left to right. Say we want to add a new value v . For this to be part of an increasing subsequence, the previous value in the subsequence must be lower than v . We might as well take the maximum length subsequence leading up to v , as the values don't matter for the continuation to the right of v . Therefore we need to extend the current longest increasing subsequence ending in a value less than v . This means we want to find the rightmost element in the dp array (as the position corresponds to the length of the subsequence), with value less than v . Say it is at position x . We can put v as a new candidate for ending value at position $x+1$ (since we have an increasing subsequence of length $x+1 + 1$, which ends on v). Note that since x was the rightmost position with value less than v , changing $dp[x+1]$ to v can only make the value smaller (better), so we can always set $dp[x+1] = v$ without checking if it is an improvement first.

Naively locating the position x with a for loop gives complexity $O(n^2)$. However, dp is always an increasing array. So we can locate x position by binary search (`std::lower_bound` in C++ directly gives position $x+1$).

The final answer is the length of the dp array after considering all elements.

The complexity is $O(n \cdot \log n)$.

In this task we were asked to find the longest strictly increasing subsequence. To find the longest increasing subsequence where we allow consecutive equal values (for example 1,2,2,3), change `lower_bound` to `upper_bound`.

Code

```

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

int main() {
    int n;
    cin >> n;
    vector<int> dp;
    for (int i = 0; i < n; i++) {
        int x;

```

```

cin >> x;
auto it = lower_bound(dp.begin(), dp.end(), x);
if (it == dp.end()) {
    dp.push_back(x);
} else {
    *it = x;
}
}
cout << dp.size() << endl;
}

```

Projects (1140)

Even though days can go up to 10^9 , we only care about days where we either start or just finished a project. So before doing anything else, we compress all days to their index among all interesting days (i.e days corresponding to a_i or $b_i + 1$ for some i). This way, days range from 0 to less than $2n \leq 4 \cdot 10^5$.

dp[i] = maximum amount of money we can earn before day i.

On day i , maybe we just did nothing, so we earn what we earned on day $i-1$, i.e $dp[i-1]$. Otherwise, we just finished some project. We earned some money doing the project, and use $dp[\text{start of project}]$ to know how much money we could have earned before starting the project. Loop through all projects finishing just before day i , and take the best one.

The complexity is $O(n \cdot \log n)$, \log comes from day compression.

Code

```

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

int main() {
    int n;
    cin >> n;
    map<int,int> compress;
    vector<int> a(n),b(n),p(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i] >> b[i] >> p[i];
        b[i]++;
        compress[a[i]], compress[b[i]];
    }

    int coords = 0;
    for (auto&v : compress) {
        v.second = coords++;
    }

    vector<vector<pair<int,int>>> project(coords);
    for (int i = 0; i < n; i++) {
        project[ compress[b[i]] ].emplace_back( compress[a[i]], p[i] );
    }

    vector<long long> dp(coords, 0);
    for (int i = 0; i < coords; i++) {
        if (i > 0) {
            dp[i] = dp[i-1];
        }
        for (auto p : project[i]) {
            dp[i] = max(dp[i], dp[p.first]+p.second);
        }
    }
    cout << dp[coords-1] << endl;
}

```