icecuber's blog

CSES DP section editorial

By **icecuber**, 10 months ago,

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 CSES 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 CSES.

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)

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++) {
    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 — ‹*value of i'th coin*›] possibilities (not dp[i-1][x — ‹*value of i'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 $O(n \cdot target)$.

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] = $\min_{d \in digits(x)}$ 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 x 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,..,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 — <*value of i'th coin*> using the first i-1 coins.

Note that we only need to consider sums up to 1000 · n, since we can't make more than that using n coins of value $\leq$ 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] = 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[i][i] = x[i].

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[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;
}
```

cses, editorial, dp, #dp

+192 icecuber 10 months ago 57

Comments (57)

Write comment?

fanof300iq

10 months ago, # | 0

Hello. Thanks for editorial. Do you have links to above problems or where I can submit them. I want to try all of them as they look good from IPOV.

→ Reply

arknave

10 months ago, # ^ | 0

https://cses.fi/problemset/

→ Reply

fanof300iq

10 months ago, # ^ | 0

Thanks a lot.

→ Reply


Errichto

10 months ago, # | +46

Cool, thanks. I already recommend CSES problem set to some people and I will now link to this blog for dp. Just next time consider using names like `ways` or `best_score` in the future instead of `dp` in every single problem — if you want to then publish your codes to help others.

→ Reply


meooow

10 months ago, # | +21

In my opinion, open editorials to CSES problems (with code!) goes against the spirit of the platform. Not providing solutions or access to accepted submissions must have been a conscious decision by the maintainers.

→ Reply

icecuber

10 months ago, # ^ | ← Rev. 2 +37

I thought about that, and therefore asked **pllk** for permission before writing the editorial. He encourages editorials for the problems. However, he mentioned that it might cause some people to read the editorial without thinking about the problem.

→ Reply

fanof300iq

10 months ago, # ^ | 0

meooow I think you are wrong here. I agree one should think before rushing to editorials but having no editorials forces one to leave the question without any further learning and development.

Thanks to **icecuber** who invested his precious time in writing some editorials and **pllk** for allowing it.

→ Reply


meooow

10 months ago, # ^ | +20

Having no editorial is not enough reason to leave a problem. There are many avenues to ask for help once you have truly exhausted all approaches you can think of.

However, since pllk approves of this, my point of complaint isn't valid.

→ Reply

10 months ago, # ^ | +1

Can you kindly suggest some of the many avenues you mention of? And are they

more effective and efficient than having the editorial at your disposal?

shoya

→ Reply

10 months ago, # ^ | +2

Asking for help FAQ

meooow

Is the answer you get by asking for help going to be better than an editorial? Maybe not.
Is the absence of an editorial going to make you work harder to solve the problem? Absolutely yes.

Please do not think that I am preaching for a world with no editorials. It is just that the design of the CSES problemset implies that it is meant to solved by putting in maximum individual effort.

→ Reply

10 months ago, # ^ +1

shoya

Well, I believe that every problem should have an editorial. Let's say I put in 2 days worth of effort solving a problem without making any real progress then an editorial guarantees me closure for my efforts. On the other hand, asking help from others doesn't provide me that certainty. Especially for the guys like me who don't have coding circles where we can ask for help from friends. Cyans(or lower rating) people asking for help from a random person usually gets ignored. That's just what I have observed and my opinion.

→ Reply

9 months ago, # ^ |

mokoto

shoya right. exactly right.

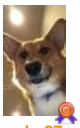
→ Reply

8 months ago, # ^ |

tfg

Idk, I usually answer people that ask nicely. Show the problem source and more people might help.

→ Reply



nchn27

10 months ago, # | +12

Thanks for this. I love the work here. CSES needs more editorials like this; sometimes even just stalking people's code isn't enough to understand what's going on. I hope you keep going with this too.

→ Reply

Hd7

10 months ago, # | 0

Thanks to this blog, I go back to my account on CSES problem set. I found out that I didn't solve 2 problems in DP section (Got `TLE` several testcases). I solved it and AC in first submit, but the logic is the same as I have solved it before. Lmao =)))

→ Reply

icecuber

9 months ago, # ^ | +7

I've gotten two more questions about TLE in Coin Combinations II, so I guess it deserves a comment. Since we are doing 10^8 operations in one second, we need those operations to be efficient. This means we can't have many cache misses.

You get cache misses by accessing array entries that are far away from each other. My implementation loops through `i`, then `j`. It accesses `dp[i][j]`, `dp[i-1][j]` and `dp[i][j-x[i-1]]`.

If you order your loops differently (`j`, then `i`), or use `dp[j][i]` instead of `dp[i][j]` (so you swapped the meaning of the dimensions), you will likely get TLE.

In terms of rules of thumb, we see that the dimension containing `j-x[i-1]` varies most, and therefore put it as the inner dimension of the dp array. And we loop through the dimensions the same order as the dimensions of the array. Below are some more detailed explanations.

If we loop through `i`, `j-x[i-1]` varies a lot, this means cache misses. Therefore we need to loop through `i` in the outer loop. Looping through `j` gives contiguous memory accesses, so we get few cache misses by having `j` as the inner loop.

If you define your array as `dp[j][i]` instead of `dp[i][j]`, then `dp[j-x[i-1]][i]` goes far from `dp[j][i]`, compared to `dp[i][j]` and `dp[i][j-x[i-1]]`. This is because the outer dimension gives smaller distance in memory than the inner dimension (you can think of `dp[i][j]` as accessing index $10^6 i + j$ in a linear array, if the second dimension has size 10^6).

→ Reply

nilesh8757

7 months ago, # ^ | ← Rev. 2 0

Is it the same reason (that O(n * sum) solution requires 10^8 ops, in worst case) and hence top-down is not feasible due to extra overhead of recursion ?

→ Reply

Ihtemadul

3 weeks ago, # ^ | ← Rev. 2 0

Thanks,i didn't think caching might be a problem in a program although it makes quite a sense now. however, we can do the program in *O(x)* space complexity,right? That could avoid the problem of caching.

→ Reply

Drom

10 months ago, # | 0

What's essentialy the difference between coins combinations **I** and **II**, why the added dimension and why we reversed the order of loops?

→ Reply

knakul_853

9 months ago, # | 0

Next Graph theory, please...

→ Reply

mokoto

9 months ago, # | 0

Thanks, Due to no editorial novice programmers couldn't try those problems or either leave it after sometime. Please make next one on graph and tree. **icecuber**

→ Reply

steelarm

8 months ago, # | 0

In **Book Shop** , if I buy the `i'th` book then shouldn't `x-price[i]` money be left for the remaining `i-1` books , so I can't see why *x-price[i-1]* money is left for the remaining `i-1` books as that would imply that the money left is after buying the `(i-1)'th` book. So if **icecuber** or anyone else could please explain this it would be a great help. Thanks.

→ Reply

icecuber

8 months ago, # ^ | 0

The array "price" is 0-indexed, so the price of the i'th book is price[i-1].

→ Reply

steelarm

8 months ago, # ^ | +3

I feel so dumb for asking such a question as now it is obvious what you were trying to say. And I am sorry for asking something like this. I am thankful to you for answering and also for the editorial. Thanks **icecuber**.

→ Reply

nilesh8757

7 months ago, # | ← Rev. 2 0

For Coin Combinations II (1636), my solution getting TLE in 3 TCs. Unlike editorial, I'm trying top-down approach. Can someone help please?

→ Reply

muhammadalamsyah47

3 weeks ago, # ^ | 0

TLE on 100 1000000? I don't know but I think top-down will be slower when the table is filled-up

→ Reply

tautology

7 months ago, # | 0

In Two Sets II problem,

`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).`

Why putting n always in second set gurantees unique sets? How to prove above statement.

→ Reply

victor142

7 months ago, # ^ | ← Rev. 2 0

Since you aren't having 'n' in your first set, you'll never have a first set that consists of a combination involving 'n' i.e. all the times a combination requires 'n' you are simply not counting it... thus you end up counting things only once.

→ Reply

overwrite

4 months ago, # | +7

Please do the same for "Graph Algorithms" section. Thanks.

→ Reply

Saifurrahman

3 months ago, # | 0

please provide editorial for graphs problem. Thanks:)

→ Reply

SuperJ6

3 months ago, # ^ | +3

It takes a long time to write up an editorial like this, and the graph section has like a billion problems. If icecuber provides an editorial that's great, but there exists code on github for all the problems so if you're stuck you shouldn't rely on him making an editorial, as he may not want to. Notice how people asked 7 months ago (comment) and he still has not, so I doubt it'll happen smh. No need for people to keep asking.

→ Reply

kaiyu2000

2 months ago, # | ← Rev. 3 +13

Check out the AC codes for all CSES dp problems here:

https://github.com/pratikkundnani/cses-dp-solutions/tree/master

I will keep updating the repository! I will be solving graph problems too.

Star it if you liked it.

Thank you!

→ Reply

deadpool221b

2 months ago, # | 0

Can you write an editorial for the graph section too...it will be a great help..

→ Reply

kaiyu2000

2 months ago, # ^ | ← Rev. 2 0

I am not sure about the editorials but I will be solving graph section and I will upload all AC codes of graph section on my Github. I will update you'll.

→ Reply

shah_entrance

2 months ago, # | ← Rev. 2 0

Why my code for Minimizing Coins gives TLE for some test cases, even it's complexity is also O(n*target)?

Code

→ Reply

lohit_97

2 months ago, # ^ | ← Rev. 2 +1

The `i%arr[j]` is causing the TLE I believe. I removed it, since it isn't required as `dp[i-arr[j]]` will handle it.

This is because the constraints are already too tight and modulo operation is time costly operation. Also, sorting `arr` also improved the time a bit, but major contribution was made by remove the module operation.

Hope this helps.

Code

Sorry for the poor formatting, I am not able to figure out a better way.

→ Reply

shah_entrance

2 months ago, # ^ | 0

Yeah, Got it!!

→ Reply

4 weeks ago, # ^ | ← Rev. 2 0

You can also use something like this:

if (value >= MOD) value -= MOD;

This is much much faster than the costly operation of modulus operator!

→ Reply

MSH

Phantom2611

2 months ago, # | 0

can you also provide editorials for the graph section as well

→ Reply

Shwetank

2 months ago, # | 0

You can also refer this competitive programming book by cses

→ Reply

code_episky

2 months ago, # | 0

Hello coders!! I'm listing out my AC solutions of CSES DP questions. It's not fully updated. I'm working on it. you can have a visit to look into the AC codes. Do star it if you like it. **Thank you. Suggestions are welcome**.

https://github.com/noob-hu-yaar/CSES-Problem-set-solutions-DP-?files=1

→ Reply

sigma_g

6 weeks ago, # | 0

Can anyone explain the matrix expo solution to Dice Combinations? It has much lower complexity ($6^3 \log n$ afaict). This is the code, for example, which used the matrix

```
1 1 1 1 1 1
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
```

raises it to the power n, and then prints `dp[0][0]` as the answer. Can someone explain how do we come up with this matrix in the first place?

→ Reply

6 weeks ago, # ^ | 0

Check CP handbook's matrix chapter's linear recurrences section.

→ Reply

overwrite

6 weeks ago, # | 0

Why does this code give a runtime error? It works fine on my computer for the same inputs.

Code

→ Reply

escrow

6 weeks ago, # ^ | ← Rev. 2 0

I figured it out myself, the problem with the code was that I was using long long for dp array which was exceeding the memory limit, changing it to int passed all the test cases. I just wrote it here in case someone faces

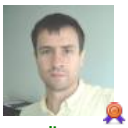
escrow

the same problem.
→ Reply

5 weeks ago, # | 0

Thanks a lot **icecuber**! I found your editorial super useful.

It was hard for me to understand the Removal Game (1097) solution at first. I even found another version here, and struggled comprehending it as well :-) Like, I saw how the code could possibly work, of course, but the trail of thought leading to that code was eluding me. So I found my own trail then, which may be somewhat more beginner-friendly. So, here it is:


ikr

Given the input numbers `xs`, I started with two arrays: `A[l][r]`, `B[l][r]` -- the scores for each player, when they have the first move on the range of `[l, r]`. Now, say, I'm the player A, and I pull from the front of the range -- the `xs[l]`. Let `t` be the `range_sum(l, r)`. Then, `t = xs[l] + B[l + 1][r] + alpha`, which leads to `alpha = t - xs[l] - B[l + 1][r]`. Thus, `A[l][r] = xs[l] + alpha = t - B[l + 1][r]`. Similarly, if I pull from the back of the range, `A[l][r] = t - B[l][r - 1]`. The optimal strategy is to take the largest of the two numbers.

Next, we make an observation, that the `A` will be identical to `B`, and we only need one DP matrix.

The code
→ Reply

5 weeks ago, # | 0


arvindr9

In the edit distance problem, it says -- "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."

Why do we only need to consider the rightmost operation? I feel this is worth discussing (either in the post or as a reply to my comment).
→ Reply

5 weeks ago, # | ← Rev. 4 +3


rupav

In Coin Combinations 2, you don't need extra dimension/state. dp[x + 1] = {0} would work fine. dp[0] = 1. dp[i] represents no. of ways to reach value 'i' with coins considered so far. So loop through coins in any order, and update values of future states of dp by looping through previous states of dp from left to right i.e. from 0 to x. That is dp[current_coin + state] += dp[state];
Spoiler - Code

Space: O(target)

Time: O(target * n)
→ Reply

4 weeks ago, # | 0


MesutOzil

Can anyone explain me the array description question? Or just explain me the output of this test case: 4 5 2 0 3 0 I thought it would be 6 but it turns out to be 2. Any help would be appreciated.
→ Reply

4 weeks ago, # | ← Rev. 3 0

rum3r

Thanks, to **icecuber** for writing an amazing editorial. Without this I wouldn't have thought of even touching single dp problem and someone can explain the array description problem.
→ Reply

3 weeks ago, # | 0


sarkarasm

In Coin Combinations II(1636) shouldn't the vector x be sorted?
→ Reply

sumantopal07

3 weeks ago, # | 0

Two Sets II (1093) problem we can also do it in normal way (that is including n too) then finally divide by 2 to avoid repetition. NOTE:- use modulo inverse while performing division.

CODE

→ Reply

bhuvnesh97

2 weeks ago, # | 0

In the problem two set (1093) u said that if we count subset till the last element than we end up repeating the same set twice.so we can get our desired answer by just dividing it by 2. but here it fails .

→ Reply

TheLogiqueViper

11 days ago, # | 0

please consider making one editorial on graph section also **icecuber**

→ Reply

YGBoss007

6 days ago, # | 0

In problem Projects(1140), I'm not able to get the compressing the days part. Can anyone explain with an example?

→ Reply

kartik8800

6 days ago, # | ← Rev. 2 0

I am making detailed video explanations for some of the problems of this section, I will keep updating this comment whenever I add a new video. Along with the explanation I live code the solution at the end of the video and submit the code on the platform.

Interested may check the following videos.
Dice Combinations: https://youtu.be/5gd5jptXWAM
Coin Combinations: https://youtu.be/-pXjopzMVrE
Grid Paths: https://youtu.be/V64F4wlodUM
Book Shop: https://youtu.be/qpNy2nuSuaI
Also added on my channel videos for coin combinations 2, array Description, edit distance and Rectangle cutting.

Hopefully people will find this helfpul.

→ Reply

aop

4 days ago, # | 0

Editorials like this should be appreciated!

→ Reply