Competitive Coding @ Berkeley

Decal Lecture #8 - Dynamic Programming

What is dynamic programming?

- Dynamic programming (DP) is a programming technique that efficiently solves tasks by utilizing cached solutions to smaller sub-tasks

What is dynamic programming?

- Dynamic programming (DP) is a programming technique that efficiently solves tasks by utilizing cached solutions to smaller sub-tasks
- Commonly used to…
  - Find minimum or maximum solution
  - Counting number of ways
- You can think of it as “optimized brute force”

Fibonacci Numbers

- F(0) = 0
- F(1) = 1
- For n >= 2:
  - F(n) = F(n-1) + F(n-2)
- The first 10 numbers of the fibonacci series:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

Fibonacci Numbers

Slow solution: write a recursive function using this recurrence relation

```
def fib(n):
     if n <= 1:
          return 1
     return fib(n - 1) + fib(n - 2)
```

Fibonacci Numbers

What is the time complexity of this solution?

Fibonacci Numbers

Roughly O(2^N), since each recursive call results in two others, and the tree will grow exponentially large.

This is very slow!

Fibonacci Numbers

We can optimize this by storing each fibonacci number in an array once we compute it.

This technique is called **memoization**, and is the heart of dynamic programming.

The time complexity of this is now O(N), since we will run the recursive formula at most once for each number from 1 to N.

Fibonacci Numbers - Iterative Solution

- How to find the n^{th} fibonacci number iteratively
  - Make an array **fib** of size ***n***
  - Set base cases of **fib[0] = 0** and **fib[1] = 1**
  - Loop from **2** to **n** to solve the rest

Iterative Solution

```
n = int(input())
fib = [0] * (max(n+1, 2))
fib[0] = 0
fib[1] = 1
for i in range(2, n+1):
    fib[i] = fib[i-1] + fib[i-2]
print(fib[n])
```

Saving Memory

- We might not need to save all of the values we compute!
- In the case of fibonacci numbers
  - We only need to save the last two fib numbers
- Store the last two computed fibonacci numbers as a pair

Constant Memory Fibonacci Solution

```
n = int(input())
fib = [0, 1]
if n == 0:
    print(0)
    continue
for i in range(2, n+1):
    new_fib = [fib[1], fib[0] + fib[1]]
    fib = new_fib
print(fib[1])
```

Memory Saving Summary

- With this fibonacci problem, both solutions easily meet memory restrictions
- But some dp problems require several dimensions!
  - For problems with $n >= 5000$, using $O(n^2)$ memory would be too much!
  - This memory optimization is sometimes very necessary

Coin Change Problem

- https://cses.fi/problemset/task/1634

Time limit: 1.00 s **Memory limit:** 512 MB

Consider a money system consisting of n coins. Each coin has a positive integer value. Your task is to produce a sum of money x using the available coins in such a way that the number of coins is minimal.

For example, if the coins are $\{1, 5, 7\}$ and the desired sum is 11, an optimal solution is $5 + 5 + 1$ which requires 3 coins.

Input

The first input line has two integers n and x: the number of coins and the desired sum of money.

The second line has n distinct integers $c_1, c_2, \ldots, c_n$: the value of each coin.

Output

Print one integer: the minimum number of coins. If it is not possible to produce the desired sum, print -1.

Constraints

- $1 \leq n \leq 100$
- $1 \leq x \leq 10^6$
- $1 \leq c_i \leq 10^6$

Example

Input:
```
3 11
1 5 7
```

Output:
```
3
```

Solution

- Previously we tried a greedy solution
  - Always take as many of the biggest coins first!
- Fails for x = 14 and coins = {1, 7, 10}

Solution

- Let’s find our recurrence
  - Let dp[x] be the minimum number of coins needed to make change for ***x*** cents
  - If we have a coin value of ***k*** and **dp[x-k]** computed we can set **dp[x] = dp[x-k] + 1**

Solution

- Let's find our recurrence
  - Let dp[x] be the minimum number of coins needed to make change for ***x*** cents
  - If we have a coin value of ***k*** and **dp[x-k]** computed we can set **dp[x] = dp[x-k] + 1**
    - But this might not result in the **minimum** value for **dp[x]**
- So $\mathbf{dp[x] = \min_{0 \le i \le n} (dp[x\text{-}coin[i]] + 1)}$
  - Try to transition through all coins you have, take the minimum answer

Solution

- Base case:
  - It takes 0 coins to make change for 0 cents
    - **dp[0] = 0**
- We want to compute the minimum value for all other amounts of change
  - For all ***i*** ≠0, **dp[i] = ∞**

Code

```
n, x = [int(x) for x in input().split(" ")]
coins = [int(x) for x in input().split(" ")]

INF = 1e18
dp = [INF] * (x+1)
dp[0] = 0
for i in range(1, x+1):
    for coin in coins:
        if i >= coin:
            dp[i] = min(dp[i], dp[i-coin] + 1)

if dp[x] == INF:
    print(-1)
else:
    print(dp[x])
```

Longest Increasing Subsequence (LIS)

- https://cses.fi/problemset/task/1145
- The constraints are very large!
  - Assume that ***n <= 1000***
  - In a few weeks we'll learn how to solve this problem with the original constraints using segment tree

Time limit: 1.00 s **Memory limit:** 512 MB

You are given an array containing n integers. Your task is to determine the longest increasing subsequence in the array, i.e., the longest subsequence where every element is larger than the previous one.

A subsequence is a sequence that can be derived from the array by deleting some elements without changing the order of the remaining elements.

Input

The first line contains an integer n: the size of the array.

After this there are n integers $x_1, x_2, \ldots, x_n$: the contents of the array.

Output

Print the length of the longest increasing subsequence.

Constraints

- $1 \leq n \leq 2 \cdot 10^5$
- $1 \leq x_i \leq 10^9$

Example

Input:
```
8
7 3 5 3 6 2 9 8
```

Output:
```
4
```

Longest Increasing Subsequence Solution

- Let **dp[i]** be the length of the longest increasing subsequence ending with element **i**

Longest Increasing Subsequence Solution

- Let **dp[i]** be the length of the longest increasing subsequence ending with element **i**
- For each index ***i***, we can check all indices ***j*** that are on the left of ***i*** (***j*** < ***i***)
    - Try to extend **dp[j]** with element ***i***

Longest Increasing Subsequence Solution

- Let **dp[i]** be the length of the longest increasing subsequence ending with element **i**
- For each index ***i***, we can check all indices ***j*** that are on the left of ***i*** (***j*** < ***i***)
    - Try to extend **dp[j]** with element ***i***
- Call the given array ***a***
- If $a_i > a_j$ then we can set dp[i] = dp[j]+1

Longest Increasing Subsequence Solution

- Let **dp[i]** be the length of the longest increasing subsequence ending with element **i**
- For each index ***i***, we can check all indices ***j*** that are on the left of ***i*** (***j*** < ***i***)
    - Try to extend **dp[j]** with element ***i***
- Call the given array ***a***
- If $\mathbf{a_i} > \mathbf{a_j}$ then we can set **dp[i]** = **dp[j]**+1
- Same as before, we want **dp[i]** to be the **maximum length** longest increasing subsequence
    - So $dp[i] = \max_{0 \le j < i} (dp[j] + 1)$ only if $\mathbf{a_i} > \mathbf{a_j}$

Longest Increasing Subsequence Solution

- Base case
  - Each index ***i*** can be its own increasing subsequence of length 1
  - So our base case is dp[i] = 1 for all $0 \leq i < n$

Longest Increasing Subsequence Solution

- Do example on board
- n = 8
- a = [7, 3, 5, 3, 6, 2, 9, 8]

Longest Increasing Subsequence Solution

- Do example on board
- n = 8
- a = [7, 3, 5, 3, 6, 2, 9, 8]
- Answer will be 4
- final dp array = [1, 1, 2, 1, 3, 1, 4, 4]

Cut Ribbon

- https://codeforces.com/contest/189/problem/A
- Problem A on this week's problemset

Polycarpus has a ribbon, its length is n. He wants to cut the ribbon in a way that fulfils the following two conditions:

- After the cutting each ribbon piece should have length a, b or c.
- After the cutting the number of ribbon pieces should be maximum.

Help Polycarpus and find the number of ribbon pieces after the required cutting.

Input

The first line contains four space-separated integers n, a, b and c ($1 \le n, a, b, c \le 4000$) — the length of the original ribbon and the acceptable lengths of the ribbon pieces after the cutting, correspondingly. The numbers a, b and c can coincide.

Output

Print a single number — the maximum possible number of ribbon pieces. It is guaranteed that at least one correct ribbon cutting exists.

Examples

input	Copy
5 5 3 2	
output	Copy
2	

input	Copy
7 5 5 2	
output	Copy
2	

Note

In the first example Polycarpus can cut the ribbon in such way: the first piece has length 2, the second piece has length 3.

In the second example Polycarpus can cut the ribbon in such way: the first piece has length 5, the second piece has length 2.

Cut Ribbon Solution

- Let's think of the question a different way
  - Instead of "cutting" a ribbon of length ***n*** into pieces of length **a, b, c**
    - Create a ribbon of length ***n*** by combining pieces of **ONLY** lengths **a, b, c**

Cut Ribbon Solution

- Let dp[i] be the maximum number of ribbon pieces you can combine to make a ribbon of length ***i***
- Base case: dp[0] = 0, set rest to zero?

Cut Ribbon Solution

- Let dp[i] be the maximum number of ribbon pieces you can combine to make a ribbon of length ***i***
- Base case: dp[0] = 0, set rest to zero?
- dp [i] = max(dp[i-a], dp[i-b], dp[i-c]) + 1

Cut Ribbon Solution

- Let dp[i] be the maximum number of ribbon pieces you can combine to make a ribbon of length ***i***
- Base case: dp[0] = 0, set rest to zero?
- dp [i] = max(dp[i-a], dp[i-b], dp[i-c]) + 1
    - Be careful!
    - You should only be able to update dp[i] if it's possible to actually make a ribbon of length i-a, i-b, or i-c

Cut Ribbon Solution

- Let dp[i] be the maximum number of ribbon pieces you can combine to make a ribbon of length ***i***
- Base case: dp[0] = 0, set all other indices to -1
    - It might not be possible to make a ribbon of that specific length!
- dp [i] = max(dp[i-a], dp[i-b], dp[i-c]) + 1
    - Be careful!
    - You should only be able to update dp[i] if it's possible to actually make a ribbon of length i-a, i-b, or i-c

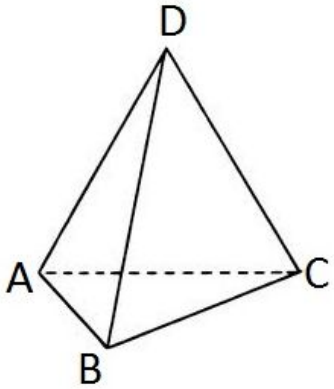
Cut Ribbon Solution

- Let dp[i] be the maximum number of ribbon pieces you can combine to make a ribbon of length ***i***
- Base case: dp[0] = 0, set all other indices to -1
    - It might not be possible to make a ribbon of that specific length!
- dp [i] = max(dp[i-a], dp[i-b], dp[i-c]) + 1
    - Only if dp[i-a] ≠-1, otherwise ignore it
    - dp[i-b] ≠-1, otherwise ignore it
    - dp[i-c] ≠-1, otherwise ignore it

Tetrahedron

- https://codeforces.com/problemset/problem/166/E
- Problem B on this week's problemset (optional but the code is in these slides lmao)

You are given a tetrahedron. Let's mark its vertices with letters A, B, C and D correspondingly.



An ant is standing in the vertex D of the tetrahedron. The ant is quite active and he wouldn't stay idle. At each moment of time he makes a step from one vertex to another one along some edge of the tetrahedron. The ant just can't stand on one place.

You do not have to do much to solve the problem: your task is to count the number of ways in which the ant can go from the initial vertex D to itself in exactly n steps. In other words, you are asked to find out the number of different cyclic paths with the length of n from vertex D to itself. As the number can be quite large, you should print it modulo 1000000007 $(10^9 + 7)$.

Input

The first line contains the only integer n $(1 \le n \le 10^7)$ — the required length of the cyclic path.

Output

Print the only integer — the required number of ways modulo 1000000007 $(10^9 + 7)$.

Examples

input	Copy
2	
output	Copy
3	

Tetrahedron

- **Key observation:** this tetrahedron is the same as a complete graph with 4 vertices, $\mathbf{K_4}$
- Let's remap [A, B, C, D] to [0, 1, 2, 3]
    - Vertex 3 is now the starting point D (the top tip of the tetrahedron)

Tetrahedron

- **Key observation:** this tetrahedron is the same as a complete graph with 4 vertices, $\mathbf{K_4}$
- Let's remap [A, B, C, D] to [0, 1, 2, 3]
    - Vertex 3 is now the starting point D (the top tip of the tetrahedron)
- How many ways to get to a particular vertex **v** in **k** steps?
    - Every vertex except for **v** has an edge to **v**

Tetrahedron

- **Key observation:** this tetrahedron is the same as a complete graph with 4 vertices, $\mathbf{K_4}$
- Let's remap [A, B, C, D] to [0, 1, 2, 3]
    - Vertex 3 is now the starting point D (the top tip of the tetrahedron)
- How many ways to get to a particular vertex **v** in **k** steps?
    - Every vertex except for **v** has an edge to **v**
- Let dp[i][j] be the number of ways to return to vertex **j** with **i** steps

Tetrahedron

- **Key observation:** this tetrahedron is the same as a complete graph with 4 vertices, $\mathbf{K_4}$
- Let's remap [A, B, C, D] to [0, 1, 2, 3]
  - Vertex 3 is now the starting point D (the top tip of the tetrahedron)
- How many ways to get to a particular vertex **v** in **k** steps?
  - Every vertex except for **v** has an edge to **v**
- Let dp[i][j] be the number of ways to return to vertex **j** with **i** steps
- Then $dp[i][j] = \sum_{k=0}^{3} dp[i-1][k]$ where $k \neq j$
- Do this for all vertices ***j***

Tetrahedron

```
int main(){
    ll n;
    cin>>n;
    vector<vector<ll>> dp(n+1, vector<ll>(4));
    dp[0][3] = 1;
    for(int i = 1; i <= n; i++){
        for(int u = 0; u < 4; u++){
            for(int v = 0; v < 4; v++){
                if(u != v)
                    dp[i][v] = (dp[i][v] + dp[i-1][u]) % mod;
            }
        }
    }
    cout<<dp[n][3]<<"\n";
}
```

Tetrahedron

```
n = int(input())
mod = 1e9+7
dp = [[0]*4 for i in range(n+1)]
dp[0][3] = 1
for i in range(1, n+1):
    for u in range(4):
        for v in range(4):
            if u != v:
                dp[i][v] = (dp[i][v] + dp[i-1][u]) % mod

print(int(dp[n][3]))
```

Tetrahedron

- Code on the previous slide doesn’t work!
  - Memory limit exceeded verdict

Tetrahedron

- Code on the previous slide doesn't work!
  - Memory limit exceeded verdict
- ***n <= 10^7***
- O(n) memory is too much for this problem
  - Memory limit of 256mb

Tetrahedron

- Try the memory saving trick!
- Why do we need to save the number of ways for all path lengths up until ***n***?

Tetrahedron

- Try the memory saving trick!
- Why do we need to save the number of ways for all path lengths up until ***n***?
  - We don't!
- We only need to use the dp values of ***i*** steps to calculate the number of ways with ***i+1*** steps!
  - After we calculate ***i+1***, we can just get rid of it

Tetrahedron Working Solution

```
n = int(input())
mod = 1e9+7
dp = [0]*4
dp[3] = 1
for i in range(1, n+1):
    new_dp = [0] * 4
    for u in range(4):
        for v in range(4):
            if u != v:
                new_dp[v] = (new_dp[v] + dp[u]) % mod
    dp = new_dp
print(int(dp[3]))
```

Tetrahedron Note

- Python is reallyyy slow
  - My python solution is about 16 times slower than the C++ one
- Just a reminder than some problems are not really tested on Python (on codeforces)
- Worth knowing C++ for competitive programming purposes
  - You'll be fine without it for interview purposes

Codeforces Example

- https://codeforces.com/contest/1096/problem/D

Codeforces Example

We can consider a “state” of how many characters in the string we’ve already looked at, and how many characters of “hard” we’ve already included as a subsequence.

haaarrrhaarrdharhar

^ for example, one state could be the string in bold, with three of the characters of “hard” (h, a, and r), already having been included.

Codeforces Example

We can define F(i, j) as the minimum possible changes to make to the first i characters of the string, such that it contains the first j characters of “hard” as a subsequence (but no more).

`F(i, j) = min(F(i - 1, j - 1), F(i - 1, j) + 1)` (if the string has the next character in “hard” at position i+1)

`F(i, j) = F(i - 1, j)` (otherwise)

The answer will be `min(F(N, 0), F(N, 1), F(N, 2), F(N, 3))`

Attendance