Lesson 2

DYNAMIC PROGRAMMING I

Did you take a snack?

https://forms.office.com/r/t24xVfAPRx

Quick Announcements

Resume Book

https://tinyurl.com/cpresumes

Meta Hacker Cup

September 20: Meta Hacker Cup Practice (10am PST, 72 hours)
October 5: North American Qualifiers
October 5: Meta Hacker Cup R1 (10am PST, 3 hours)
October 19: Meta Hacker Cup R2 (10am PST, 3 hours)
November 2: Meta Hacker Cup R3 (10am PST, 3 hours)
November 16: Southeast NA Regionals (Registration deadline TBD)
December 7: Meta Hacker Cup Finals (6am PST 4 hours)

Definition of Dynamic Programming (DP)

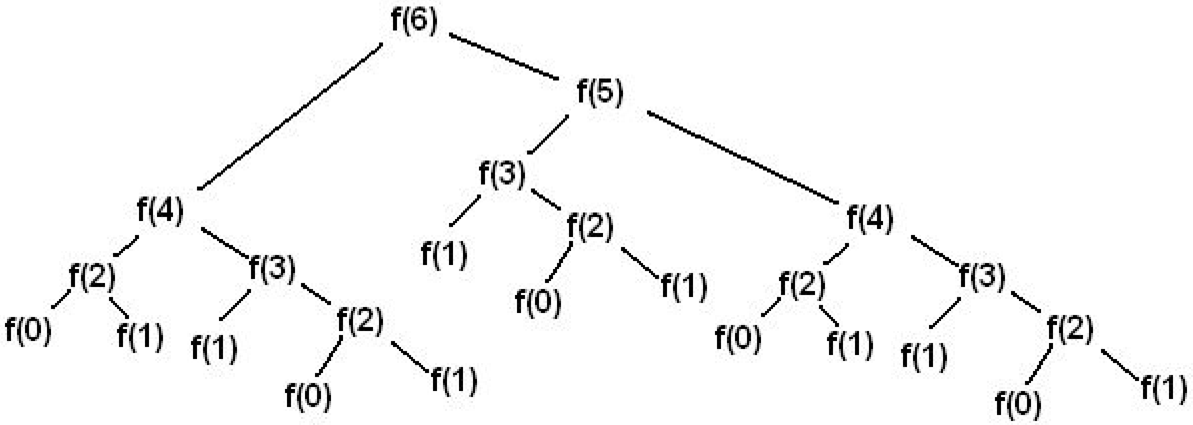
Solve big problems by breaking them down into smaller problems, repeatedly

1. Overlapping subproblems
2. Optimal substructure

Fibonacci

F[0] = F[1] = 1

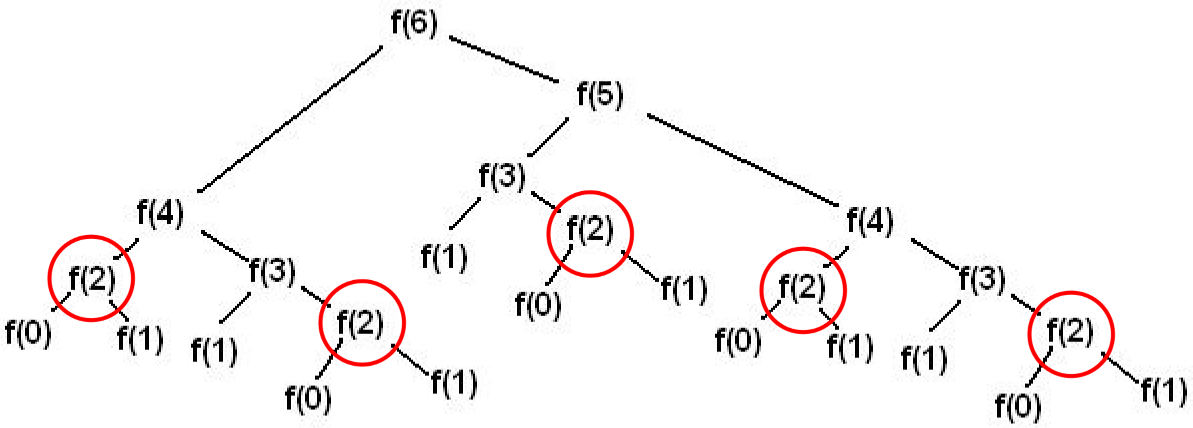
F[n] = F[n-1] + F[n-2]



Fibonacci

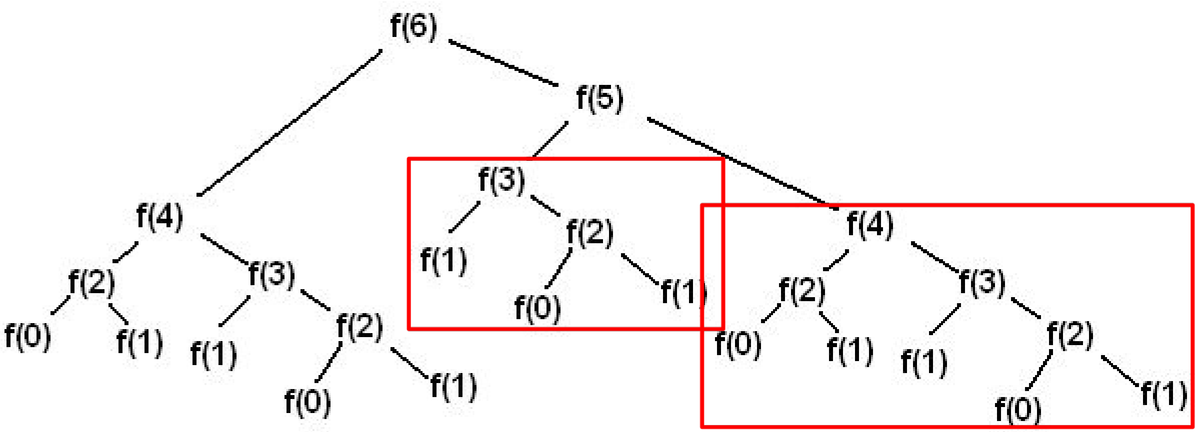
Overlapping subproblems:

The same values of F [n] appear many times



Fibonacci

Optimal substructure: We can find bigger answers (F_n) from smaller answers ($F_{n-1} + F_{n-2}$)



DP Terminology

State : Parameters of the problem instances
State space : Set of all possible instances
Transition : Rule to get bigger answers from smaller answers
Base case : Smallest possible instances that define all the other answers

DP Terminology

State : F(n) = nth Fibonacci number
State space : Integer n from 0 onwards
Transition : F(n) = F(n - 1) + F(n - 2)
Base case : F(0) = F(1) = 1

Fibonacci - Recursive

```
#include <bits/stdc++.h>
using namespace std;
int fib(int x) {

     if (x==0 || x==1) return 1; //base case (why 2?)
     else return fib(x-1)+fib(x-2); //recursive formula
}
int main () {
     printf("%d\n", fib(5));
}
```

Fibonacci - Top Down

```
int F[45]; // outside of main(): F[] is all 0s.
// No Fibonacci number is 0, so we can treat 0 as "not calculated before".


int fib(int n) {
     if (n <= 1) return 1;
     if (F[n] != 0) return F[n]; //calculated before
     return F[n] = fib(n - 1) + fib(n - 2); //save to array
}
```

Fibonacci - Bottom Up

```
int N, F[45]; // F(45) is the largest Fibonacci number that fits in an int


F[0] = F[1] = 1;
for (int i = 2; i <= N; i++)
    F[i] = F[i - 1] + F[i - 2];
```

Pros and Cons

Top Down	Bottom Up
Never visits unnecessary states Faster if the required states are sparse in the state space	Visits all the states Need to be able to iterate in the correct order
Function call overhead, watch out for stack overflow	No function call overhead

Classic DP Problems

Integer partition (coin change)

Change-making problem

0-1 Knapsack

1D, 2D maximum sum

Longest Increasing Subsequence

1D-Maxsum

1D-Maxsum

Given N integers in an array, find a *contiguous* subarray such that its sum is maximized and output the sum.

Array = { 2, -3, 2, -1, 2, -5, 1 }

Maxsum = 3

If all numbers are non-negative?

If all numbers are non-negative?

Maxsum is the sum of all numbers.

If all numbers are non-negative?

Maxsum is the sum of all numbers.

If all numbers are negative?

If all numbers are negative?

Maxsum is the least negative number.

If all numbers are negative?

Maxsum is the least negative number.

Kadane's Algorithm

Let maxsum(*i*) be the maximum sum that ends at A_i and must include A_i.

Kadane's Algorithm

Let maxsum(i) be the maximum sum that ends at A_i and must include A_i.

$$\mathrm{maxsum}(0) = A_0$$

$$\mathrm{maxsum}(i) = \max \begin{cases} \mathrm{maxsum}(i-1) + A_i \\ A_i \end{cases}$$

Kadane’s Algorithm

Let maxsum(*i*) be the maximum sum that ends at A_i and must include A_i.

$$\text{maxsum}(0) = A_0$$

$$\text{maxsum}(i) = \max \begin{cases} \text{maxsum}(i-1) + A_i \\ A_i \end{cases}$$

$$\text{ans} = \max_{0 \le i < n} \text{maxsum}(i)$$

Kadane's Algorithm

Let maxsum(*i*) be the maximum sum that ends at A_i and must include A_i.

$$\text{maxsum}(0) = A_0$$

$$\text{maxsum}(i) = \max \begin{cases} \text{maxsum}(i-1) + A_i \\ A_i \end{cases}$$

$$\text{ans} = \max_{0 \le i < n} \text{maxsum}(i)$$

O(1) transition,
O(N) states

Time complexity: O(N)

Kadane’s Algorithm

```
int A[N], maxsum[N];
int ans = maxsum[0] = A[0];
for (int i = 1; i < N; ++i) {

    maxsum[i] = max(maxsum[i-1] + A[i], A[i]);
    ans = max(ans, maxsum[i]);
}
```

On the FLY

Kadane's Algorithm

```
int A[N];
int ans = A[0], cursum = A[0];
for (int i = 1; i < N; ++i) {
    if (cursum < 0) cursum = 0; /* cursum is effectively maxsum[i-1] */
    cursum += A[i];
    ans = max(ans, cursum);
}
```

Longest **I**ncreasing **S**ubsequence

LIS

Given an array A of N numbers, find the longest subsequence such that the numbers within it are increasing.

Subsequences can be non-contiguous.

Array: {2, 6, 1, 3, 5}
LIS: 3

Greedy?

Just take the next bigger number?

No.

Just take the next bigger number?

No. Array: {2, 6, 1, 3, 5}

Try all 2^N subsets?

Try all 2^N subsets?

Yes.

Try all 2^N subsets?

Yes.
But takes O(N·2^N) time.

LIS

If we have an increasing subsequence that ends with A[i], we can add a number bigger than A[i] to get a longer subsequence.

LIS

a	3	5	2	4	8	1	5	7
lis(i)	1	2	1	2	3	1	?	

LIS

a	3	5	2	4	8	1	5	7
lis(i)	1	2	1	2	3	1		

Don't Care!

LIS

a	3	5	2	4	8	1	5	7
lis(i)	1	2	1	2	3	1		

LIS

a	3	5	2	4	8	1	5	7
lis(i)	1	2	1	2	3	1		

Max lis(i) = 2

LIS

a	3	5	2	4	8	1	5	7
lis(i)	1	2	1	2	3	1	3	

Max lis(i) = 2

LIS

Let lis(i) = LIS that ends at A_i.

LIS

Let lis(i) = LIS that ends at A_i.

$$\mathrm{lis}(0) = 1$$

$$\mathrm{lis}(i) = \operatorname*{argmax}_{j \le i} \begin{cases} 1 & j = i, \\ 1 + \mathrm{lis}(j) & j < i, A_j < A_i, \\ 0 & \text{otherwise.} \end{cases}$$

LIS

Let lis(i) = LIS that ends at A_i.

$$\text{lis}(0) = 1$$

$$\text{lis}(i) = \underset{j \le i}{\text{argmax}} \begin{cases} 1 & j = i, \\ 1 + \text{lis}(j) & j < i, A_j < A_i, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{ans} = \underset{i}{\text{argmax}}\ \text{lis}(i)$$

LIS

Let lis(i) = LIS that ends at A_i.

$$\operatorname{lis}(0) = 1$$

$$\operatorname{lis}(i) = \underset{j \le i}{\operatorname{argmax}} \begin{cases} 1 & j = i, \\ 1 + \operatorname{lis}(j) & j < i, A_j < A_i, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{ans} = \underset{i}{\operatorname{argmax}} \operatorname{lis}(i)$$

Time complexity: O(N^2)

(Spoiler alert: we can do better than N^2)

LIS

Let lis(i) = LIS that ends at A_i.

$$\text{lis}(0) = 1$$

$$\text{lis}(i) = \underset{j \le i}{\text{argmax}} \begin{cases} 1 & j = i, \\ 1 + \text{lis}(j) & j < i, A_j < A_i, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{ans} = \underset{i}{\text{argmax}}\ \text{lis}(i)$$

Time complexity: O(N^2)
(Spoiler alert: we can do better than N^2)

We can solve this in O(n log n) time using **Binary Search**. The idea is to traverse the given sequence and maintain a separate list of sorted subsequence so far. For every new element, find its position in the sorted subsequence using Binary Search.

LIS

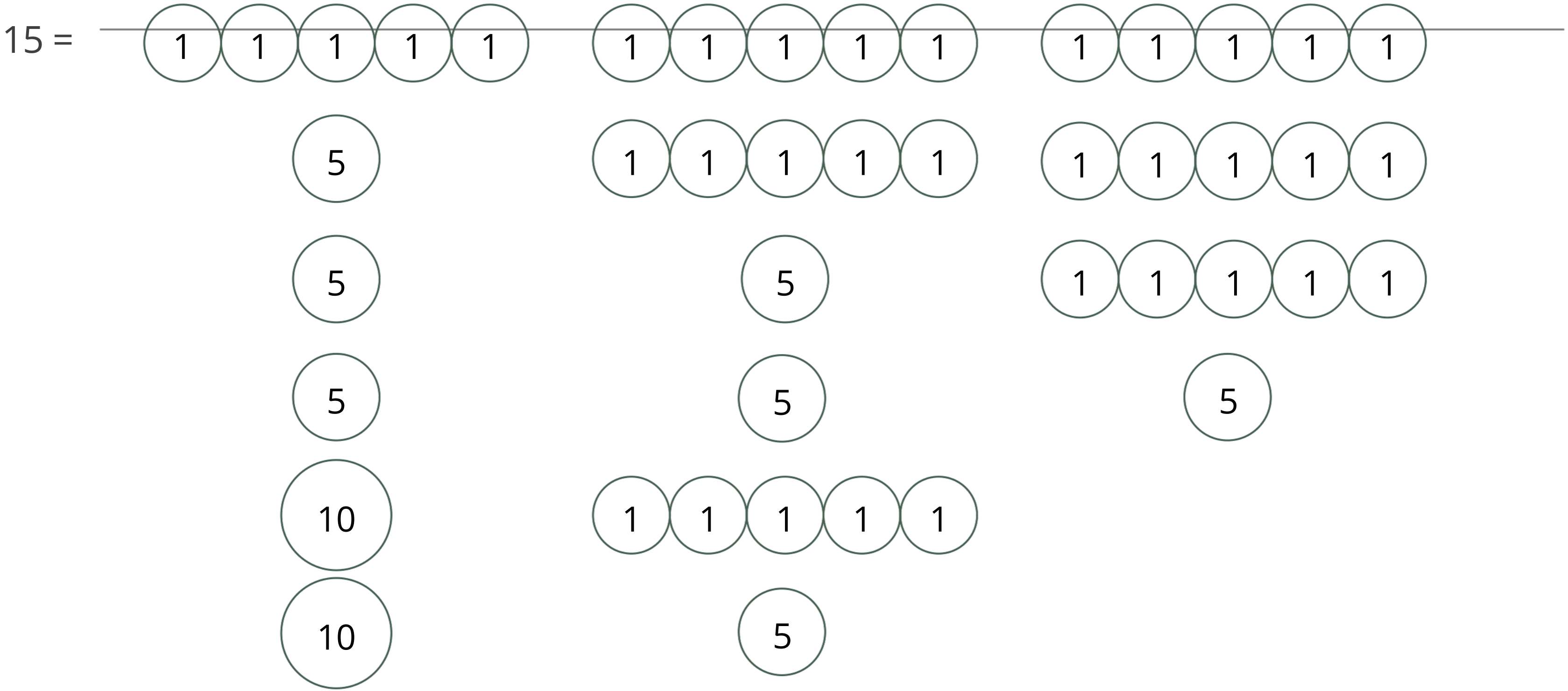
```
int N, A[1000], lis[1000], ans = 0;
for (int i = 0; i < N; ++i) {

     lis[i] = 1;
     for (int j = 0; j < i; ++j)
          if (A[j] < A[i])
               lis[i] = max(lis[i], 1 + lis[j]);

     ans = max(ans, lis[i]);
}
```

Coin Change

Coin Change

We have N coins with values C1, C2, ..., CN.
How many ways can we form the value V?

Coin Change



Coin Change

f(V) = number of ways to make V
f(V) = sum(f(V-c) for all denominations c < V)
f(0) = 1 since there is one way to make nothing which is nothing

Easy right?

Coin Change

f(V) = number of ways to make V
f(V) = sum(f(V-c) for all denominations c < V)
f(0) = 1 since there is one way to make nothing which is nothing

Easy right? NO!!
Let’s take a look at an example, say there are only 2 denominations, 1c and 2c.

Coin Change

Based on our transition, we have that $f(v) = f(v-1) + f(v-2)$, just like the Fibonacci sequence.

$f(0) = 1$, $f(1) = 1$
$f(2) = f(1) + f(0) = 2$
$f(3) = f(2) + f(1) = 3$

Coin Change

Based on our transition, we have that $f(v) = f(v-1) + f(v-2)$, just like the Fibonacci sequence.

$f(0) = 1$, $f(1) = 1$
$f(2) = f(1) + f(0) = 2$
$f(3) = f(2) + f(1) = 3$
But wait! There are only 2 ways to make 3c, namely 1c+1c+1c and 1c+2c.

Coin Change

Based on our transition, we have that f(v) = f(v-1) + f(v-2), just like the Fibonacci sequence.

f(0) = 1 , f(1) = 1
f(2) = f(1) + f(0) = 2
f(3) = f(2) + f(1) = 3
But wait! There are only 2 ways to make 3c, namely 1c+1c+1c and 1c+2c. So what’s wrong?

Coin Change

f(3) = 1 1 1

2 1

Counted twice!

1 2

Coin Change

Let *ways(i, v)* = the number of ways to form the value v with the coins C1, C2,..., Cn
ways(i, 0) = 1 (there is only 1 way to form 0)
ways(0, v) = 0 (there is no way to form with 0 coins)

v < C i: *ways(i, v)* = *ways(i - 1, v)* + 0
else: *ways(i, v)* = *ways(i - 1, v)* + *ways(i, v -* C_i)

Coin Change

Let *ways(i, v)* = the number of ways to form the value v with the coins C1, C2,..., Cn
ways(i, 0) = 1 (there is only 1 way to form 0)
ways(0, v) = 0 (there is no way to form with 0 coins)

At *ways(i, v)*

1. Use coin Ci (if possible → *Ci* <= *v*)
2. Don't use coin Ci

Coin Change

ways(i, 0) = 1 (there is only 1 way to form 0)
ways(0, v) = 0 (there is no way to form with 0 coins)

v < C i: *ways(i, v)* = *ways(i - 1, v)* + 0
else: *ways(i, v)* = *ways(i - 1, v)* + *ways(i, v - C i)*

Time complexity: O(NV)
Space complexity: O(NV)

N = number of coins, V = value to form

Coin Change

We don’t really need to remember ways for all i.

ways(i - 1, •)				(i - 1, v)		
ways(i, v)		(i, v - c_i)		(i, v)		

Space complexity: O(N + V)

Coin Change

We don't really need to remember ways for all i.

ways(i - 2, v)	DUNNID	DUNNID	DUNNID	DUNNID	DUNNID	DUNNID
ways(i - 1, v)	DUNNID	DUNNID	DUNNID	(i - 1, v)		
ways(i, v)		(i, v - C i)		(i, v)		

Space complexity: O(N + V)

Coin Change

We don’t really need to remember ways for all i.

ways(i - 2, v)	DUNNID	DUNNID	DUNNID	DUNNID	DUNNID	DUNNID
ways(i - 1, v)	DUNNID	DUNNID	DUNNID	(i - 1, v)		
ways(i, v)		(i, v - C i)		(i, v)		

Space complexity: O(N + V)

Coin Change - Top Down

```
int N, V, coins[1001], mug[1001][1001]; // coins are 1-indexed
int ways(int i, int v) { //how many ways to form v cents with first i coins

    if (v == 0) return 1; //no value
     if (i == 0) return 0; //no coins
     if (mug[i][v] != -1) return mug[i][v]; //computed before, return ans
     if (v >= coins[i]) mug[i][v] = ways(i-1, v)+ ways(i, v-coins[i]);
     else mug[i][v] = ways(i-1, v);
     return mug[i][v];
}
int main() {
    for (int i = 0; i <= N; i++)
        for (int j = 0; j <= V; j++)
            mug[i][j] = -1; //-1 means not computed
    cout << ways(N, V) << endl;
}
```

Coin Change - Bottom Up

```
int N, V, coins[1001], ways[1001][1001]; // coins are 1-indexed
ways[0][0] = 1; // initialise values
for (int v = 0; v <= V; ++v) ways[0][v] = 0;

for (int i = 1; i <= N; ++i) {
     ways[i][0] = 1;
     for (int v = 1; v <= V; ++v)
          if (v >= coins[i]) ways[i][v] = ways[i - 1][v] + ways[i][v - coins[i]];
          else ways[i][v] = ways[i - 1][v];
}
```

Coin Change - Bottom Up, Less Memory

```
int N, V, coins[1001], ways[1001]; // coins are 1-indexed


ways[0] = 1;
for (int i = 1; i <= N; ++i)// note order of loops!
    for (int v = 1; v <= V; ++v)
        if (v >= coins[i])
            ways[v] += ways[v - coins[i]];
```

Let’s try some problems!

Raymond’s Magical Donut

Raymond’s Magical Donut

Raymond has a magical donut. Being a magical donut, this donut is a very special donut. Firstly, this donut consists of N units. Note that because donuts are circular, the “first” unit in the donut is next to the “last” unit. Secondly, each bite of the donut has a certain satisfaction value. Raymond can open his mouth very wide, so he can eat as many units as he wants. When Raymond takes a bite of the donut, he gains the amount of satisfaction that bite has, which is the sum of satisfaction value of all units in the bite.

Raymond wants to take at most 2 bites. What is his maximum possible satisfaction?

Raymond’s Magical Donut

Sample input

8

5 4 -2 1 3 5 -7 1

Sample output

19

Raymond’s Magical Donut

Sample input

8

5 4 -2 1 3 5 -7 1

Sample output

19

Raymond can eat the following segments: [1 5 4], [1 3 5]

- Consider a straight line
- If we take two distinct segments, we can divide the donut

- Consider a straight line
- If we take two distinct segments, we can divide the donut

Hence, we can precompute the maxsum starting from both ends and try every possible dividing line

Now, let’s consider the case of a circle.

Now, let’s consider the case of a circle.

There are 2 cases to consider:

- Either one of the max-sums wrap around the “end” of the array,
- or neither do.

Now, let’s consider the case of a circle.

There are 2 cases to consider:

- Either one of the max-sums wrap around the “end” of the array, (???)
- or neither do. (Done)

- If one of the max-sums wrap around,
- Then, we would effectively want to remove two min-sums from the array to get our answer.

palindromes (IOI 00)

())(is obviously a palindrome.

IOI '00 P1 - Palindrome

IOI '00 - Beijing, China

A palindrome is a symmetrical string, that is, a string read identically from left to right as well as from right to left. You are to write a program which, given a string, determines the minimal number of characters to be inserted into the string in order to obtain a palindrome.

As an example, by inserting 2 characters, the string `Ab3bd` can be transformed into a palindrome (`dAb3bAd` or `Adb3bdA`). However, inserting fewer than 2 characters does not produce a palindrome.

Input Specification

The first line contains one integer: the length of the input string N, $3 \le N \le 5\,000$. The second line contains one string with length N. The string is formed from uppercase letters from `A` to `Z`, lowercase letters from `a` to `z` and digits from `0` to `9`. Uppercase and lowercase letters are to be considered distinct.

Output Specification

The first line contains one integer, which is the desired minimal number.

Sample Input

```
5
Ab3bd
```

Copy

Sample Output

```
2
```

Copy

Problem TLDR

How many characters to add to make a string a palindrome?

Solution

State: dp(i, j) is how many letters you need to make the substring with indices from [i, j) a palindrome.
State space: i = 0 to i = N to j=0 to j=N, i<=j
Transition: ???
Base case: dp(i, i) = dp(i, i+1) = 0

O(N^2) dynamic programming.

Observation

There are a few cases for substring [i, j):

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same so we only have to consider the substring [i+1, j-1), so dp(i, j) = dp(i+1, j-1)

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same so we only have to consider the substring [i+1, j-1), so dp(i, j) = dp(i+1, j-1)

Case 2: x____y,

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same so we only have to consider the substring [i+1, j-1), so dp(i, j) = dp(i+1, j-1)

Case 2: x____y,

Case 2a: We add x to the right
x____yx,

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same so we only have to consider the substring [i+1, j-1), so dp(i, j) = dp(i+1, j-1)

Case 2: x____y,

Case 2a: We add x to the right
x____yx, so dp(i ,j) = dp(i+1, j)+1

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same so we only have to consider the substring [i+1, j-1), so dp(i, j) = dp(i+1, j-1)

Case 2: x____y,

Case 2a: We add x to the right
x____yx, so dp(i ,j) = dp(i+1, j)+1

Case 2b: We add y to the left
yx____y,

Observation

There are a few cases for substring [i, j):

Case 1:
x____x
Ends are same so we only have to consider the substring [i+1, j-1), so dp(i, j) = dp(i+1, j-1)

Case 2: x____y,

Case 2a: We add x to the right
x____yx, so dp(i ,j) = dp(i+1, j)+1

Case 2b: We add y to the left
yx____y, so dp(i, j) = dp(i, j-1)+1

Transition

We combine all the possible cases.

```
if(s[i] == s[j-1]){
    dp[i][j] = min(min(dp[i+1][j] + 1, dp[i][j-1] + 1), dp[i+1][j-1]);
    //either don’t add OR add left/right
} else {
    dp[i][j] = min(dp[i+1][j] + 1, dp[i][j-1] + 1);      //add on left OR add on right
}
//hence, transition is O(1)


printf("%d\n", dp[0][(int)s.length()]);             //answer is stored here
```

PROBLEM!

$N = 5000 \rightarrow N^2 = 25000000$

PROBLEM!

$N = 5000 \rightarrow N^2 = 25000000$

Memory given for the problem is 32MB, so your memo array which is currently memo[5005][5005] will MLE.

PROBLEM!

N = 5000 → N^2 = 25000000

Memory given for the problem is 32MB, so your memo array which is currently memo[5005][5005] will MLE.

How?

On the FLY

Transition

We combine all the possible cases.

```
if(s[i] == s[j-1]) {

    dp[i][j] = min(min(dp[i+1][j] + 1, dp[i][j-1] + 1), dp[i+1][j-1]);
    // either don’t add OR add left/right
} else {
    dp[i][j] = min(dp[i+1][j] + 1, dp[i][j-1] + 1); //add on left OR add on
right
}
//hence, transition is O(1)

printf(“%d\n”, dp[0][(int)s.length()]);                          //answer is
stored here
```

Transition

In other words, the row dp[i][ ] will depend on solely values from dp[i+1][ ].
So we only need to store 2 rows of information at each time.

Examining the code below:
dp[i][j] = min(min(dp[i+1][j] + 1, dp[i][j-1] + 1), dp[i+1][j-1]);

Transition

In other words, the row dp[i][ ] will depend on solely values from dp[i+1][ ].
So we only need to store 2 rows of information at each time.

Examining the code below:
dp[i][j] = min(min(dp[i+1][j] + 1, dp[i][j-1] + 1), dp[i+1][j-1]);

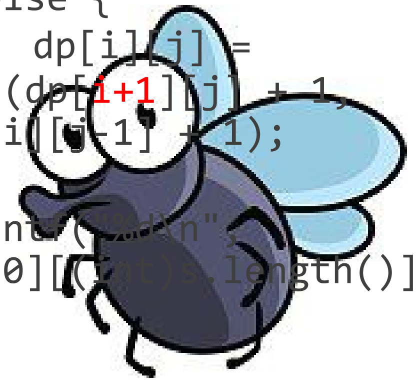
This means when you build your memo array, you must do it in:

- decreasing i (because you need dp[i+1][j] and dp[i+1][j-1])
- increasing j (because you need to use dp[i][j-1])

Do you want the on the fly code?

Code on the FLY

```
if(s[i] == s[j-1]){
    dp[i][j] =
min(min(dp[i+1][j] + 1,
dp[i][j-1] + 1),
dp[i+1][j-1]);
} else {
    dp[i][j] =
min(dp[i+1][j] + 1,
dp[i][j-1] + 1);
}
printf("%d\n",
dp[0][(int)s.length()]);
```



Code (on the fly)

```
int dp[2][5005];
for (int i = s.length(); i >= 0; i--) {

      for (int j = i; j <= s.length(); j++) {
            int ii = i%2;
            if (i == j || i + 1 == j){                                        //setting base cases
                  dp[ii][j] = 0;
                  continue;
            }
            if(s[i] == s[j-1]) {                                               //transitions
                  dp[ii][j] = min(min(dp[1 - ii][j] + 1, dp[ii][j-1] + 1), dp[1 - ii][j-1]);
            } else {
                  dp[ii][j] = min(dp[1 - ii][j] + 1, dp[ii][j-1] + 1);
            }
      }
}
printf("%d\n", dp[0][(int)s.length()]);
```

More DP coming soon...

Stay Tuned!

1. Digit DP
2. Lexicographic DP
3. DP on graphs
4. DP + Binary Search
5. State manipulation
6. Speed ups

Mash Up Time!