Dynamic Programming (Ia)

Kelvin Chow {Lrt1088}

2021-04-10

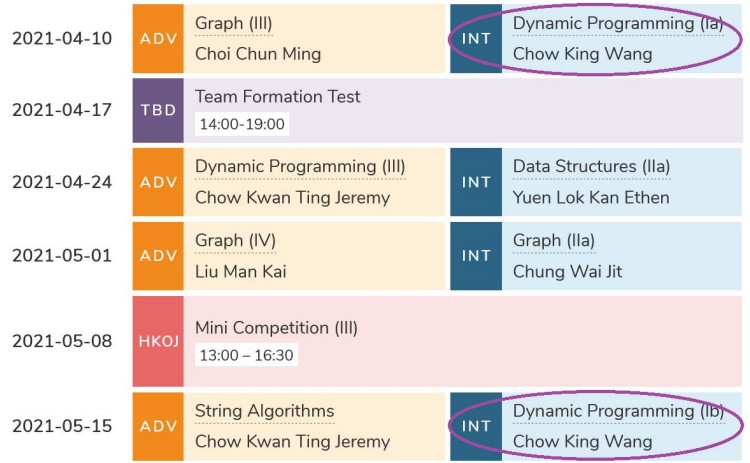
香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Table of Contents

Introduction to DP

Why DP?

- DP is a very common technique in OI
- Some tasks may divide subtasks into different levels of DP.
- We schedule 2 DP sessions in INT this year
  - DP (Ia) - Today
  - DP (Ib) - 15/05
- DP (Ia) & DP (Ib) are basically DP(I) cut into half and add more explanation.
- In this session, we will discuss about what is DP and then get us familiar with DP by solving some DP problems.



How to DP?

Prerequisites:

- Recursion
- Divide & Conquer

  ^20/02 session 2^
- Big O notation - to analyze Time and Memory complexities

  ^06/02 session 1^

How to DP?

Memorization - The Key of DP

- Basically, DP is ‘exhaustion’ but ‘don’t calculate something that has been calculated again’.
- How? ‘Remember’ what have been calculated.
- We often care Time complexity more than Memory complexity.
- Memorization is a method that “trades-off” Time with Memory.

How to DP?

What problems can DP?

- Have optimal substructure
  - The optimal solution of a problem can be constructed **efficiently** from the optimal solutions of its **sub**problems.
- Have overlapping subproblems
  - Some the optimal solutions a subproblems can be **reuse** to constructed a larger subproblems

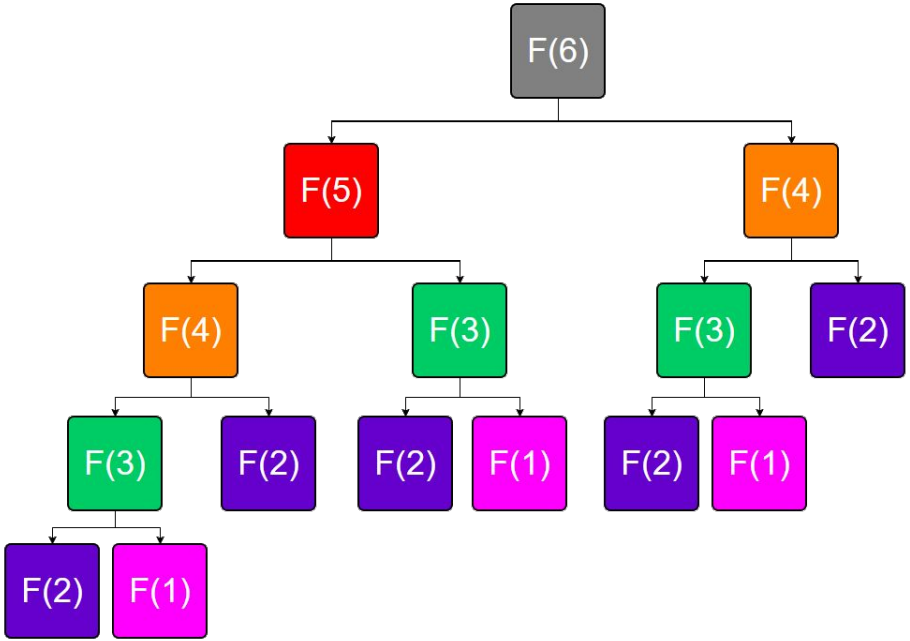
Fibonacci Sequence

Use plain text to understand DP is painful. So let's talk about some examples.

Fibonacci Sequence, a recursive function:

- F(1) = 1
- F(2) = 1
- F(n) = F(n-1) + F(n-2) for n > 2

Fibonacci Sequence



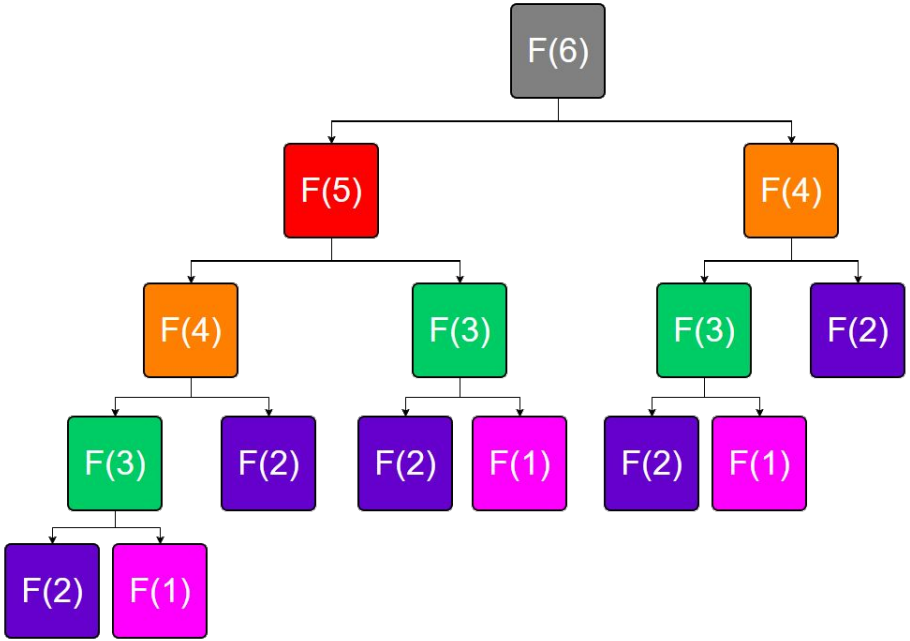
Fibonacci Sequence

Use recursion to solve a recursive function.

```
int F(int n) {
     if(n == 1 || n == 2)
          return 1;
     int tmp = F(n-1);
     return tmp + F(n-2);
}
```

(ps: If you really want to write a program to calculate Fibonacci Sequence, please be reminded that F(n) grow really fast, F(50) ~= 1e10, you may use some method to store it or just use other language.)

Fibonacci Sequence



Fibonacci Sequence

How efficient? Let's count how many times `int F()` have been called.

```
int count = 0;
int F(int n) {
     count++;
     if(n == 1 || n == 2)
          return 1;
     int tmp = F(n-1);
     return tmp + F(n-2);
}
```

Fibonacci Sequence

```
F(n) : count


F(10): 109
F(20): 13529
F(30): 1664079
F(40): 204668309
F(50): TLE :(
```

Fibonacci Sequence

It's very slow! How to speed up???

- F(n) can be constructed easily by F(n-1) and F(n-2).
  - F(n-1) and F(n-2) are **sub**problems of F(n)!
- Both F(n) & F(n-1) will call F(n-2).
  - F(n-2) can be reused!

DP can be used! But how?

Fibonacci Sequence

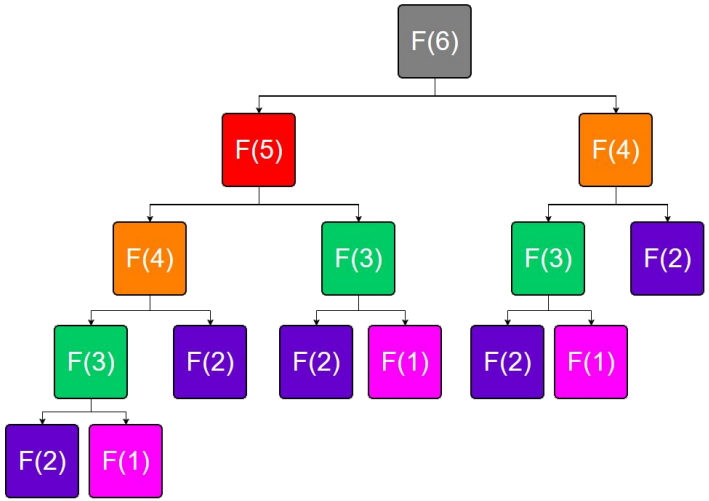
If F(n) is not calculated, calculate it. Else, just return the value F(n) stored.

```
bool caled[101]; // calculated?, let's assume caled[1] = caled[2] = 1, else = 0
int f[101]; // values of F(n), let's assume f[1] = f[2] = 1
int F(int n) {
     if(!caled[n]) {
          f[n] = F(n-1);
          f[n] += F(n-2);
          caled[n] = 1;
     }
     return f[n];
}
```

Fibonacci Sequence

Let's dry run together!

Initial State

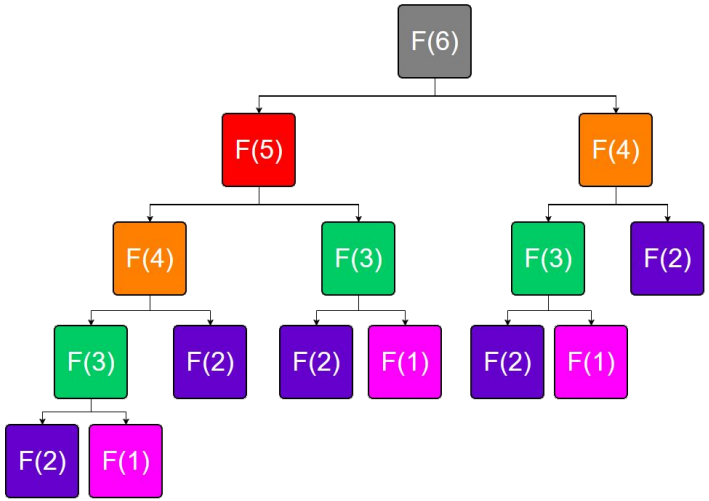


	1	2	3	4	5	6
`f[]`	1	1	-	-	-	-
`caled[]`	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!

Called F(6)

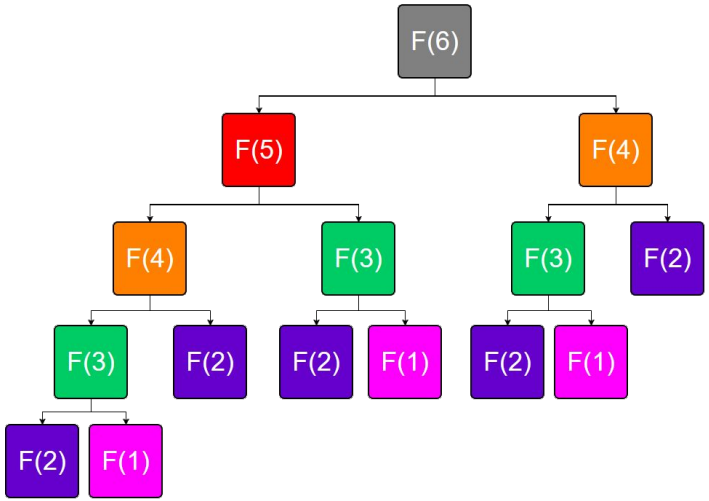


	1	2	3	4	5	6
f[]	1	1	-	-	-	-
caled[]	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!

Called F(6) -> F(5)

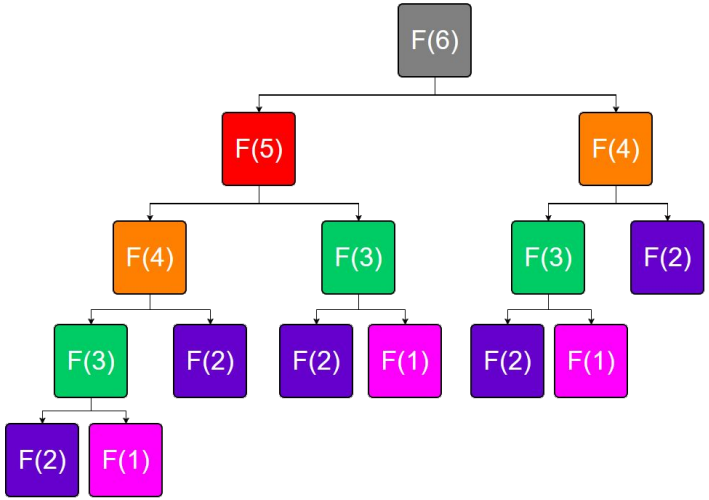


	1	2	3	4	5	6
f[]	1	1	-	-	-	-
caled[]	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!

Called F(6) -> F(5) -> F(4)

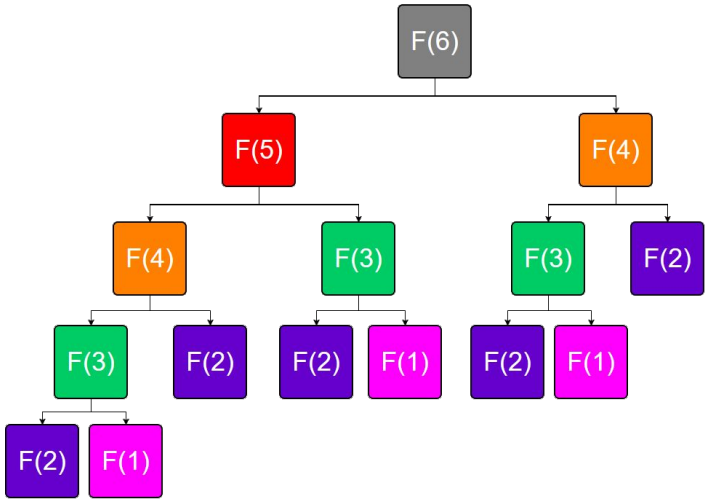


	1	2	3	4	5	6
f[]	1	1	-	-	-	-
caled[]	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!

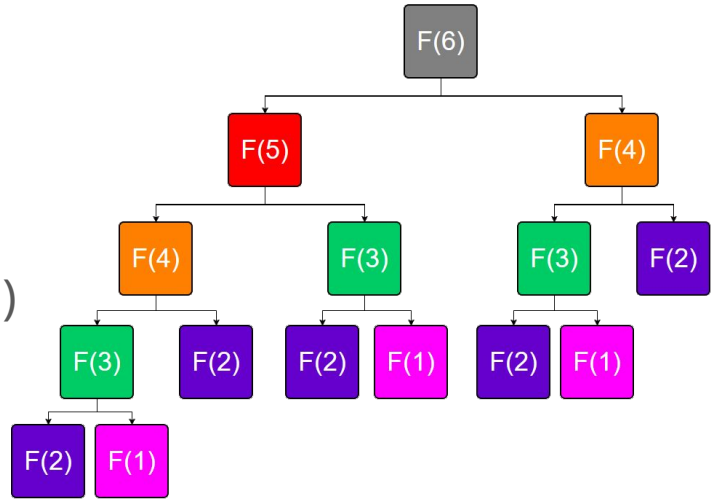
Called F(6) -> F(5) -> F(4) -> F(3)



	1	2	3	4	5	6
f[]	1	1	-	-	-	-
caled[]	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!



Called F(6) -> F(5) -> F(4) -> F(3) -> F(2)

caled[2] is 1, return f[2]

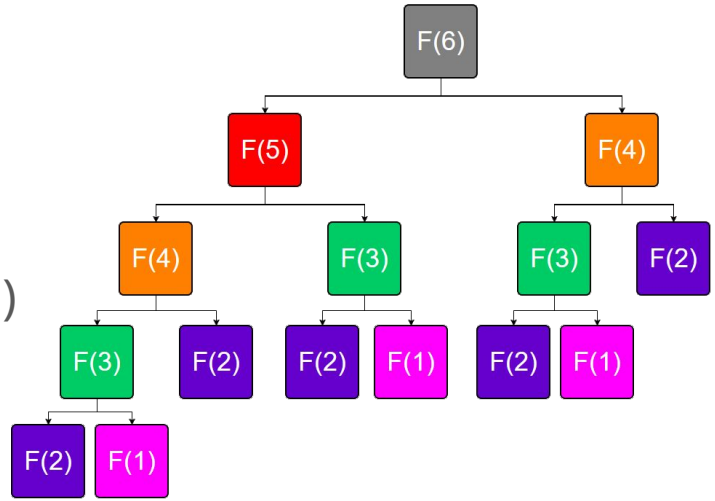
	1	2	3	4	5	6
f[]	1	1	-	-	-	-
caled[]	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!

Called F(6) -> F(5) -> F(4) -> F(3) -> F(1)

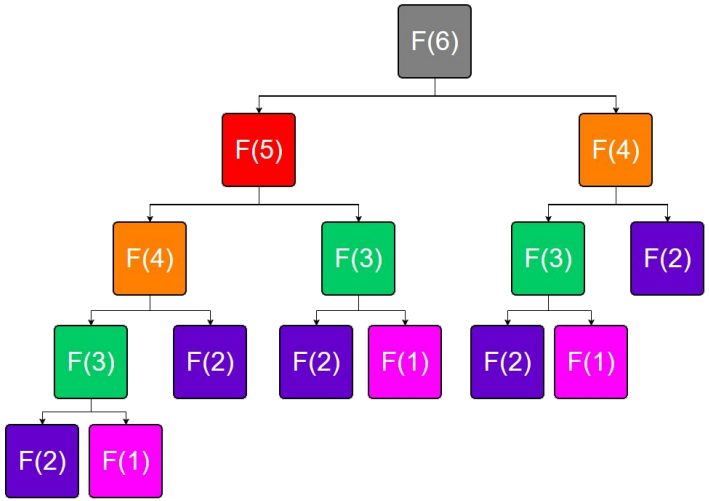
caled[1] is 1, return f[1]



	1	2	3	4	5	6
f[]	1	1	-	-	-	-
caled[]	1	1	0	0	0	0

Fibonacci Sequence

Let's dry run together!



Called F(6) -> F(5) -> F(4) -> F(3)

f[3] = F(2) + F(1)

caled[3] = 1

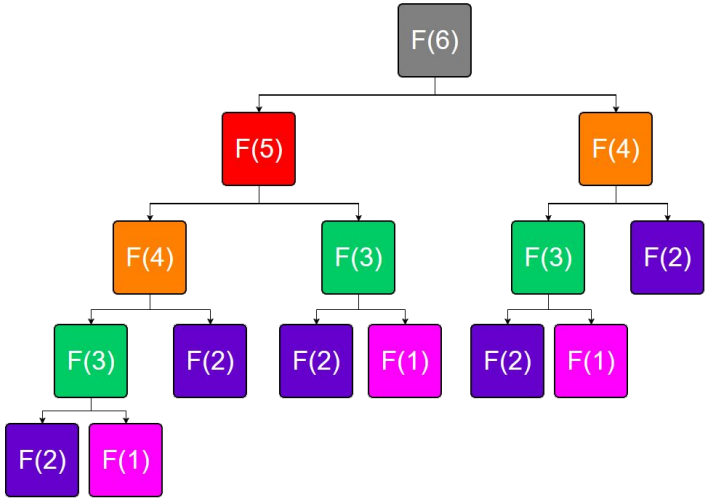
	1	2	3	4	5	6
f[]	1	1	2	-	-	-
caled[]	1	1	1	0	0	0

Fibonacci Sequence

Let's dry run together!

Called F(6) -> F(5) -> F(4) -> F(2)

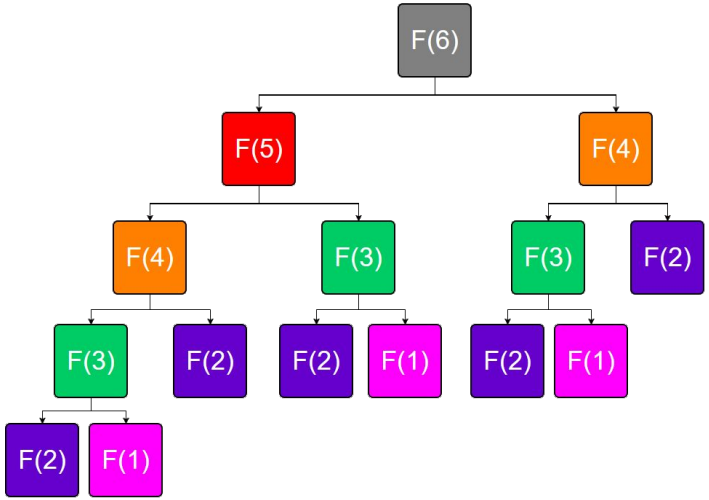
caled[2] is 1, return f[2]



	1	2	3	4	5	6
f[]	1	1	2	-	-	-
caled[]	1	1	1	0	0	0

Fibonacci Sequence

Let's dry run together!



Called F(6) -> F(5) -> F(4)

f[4] = F(3) + F(2)

caled[4] = 1

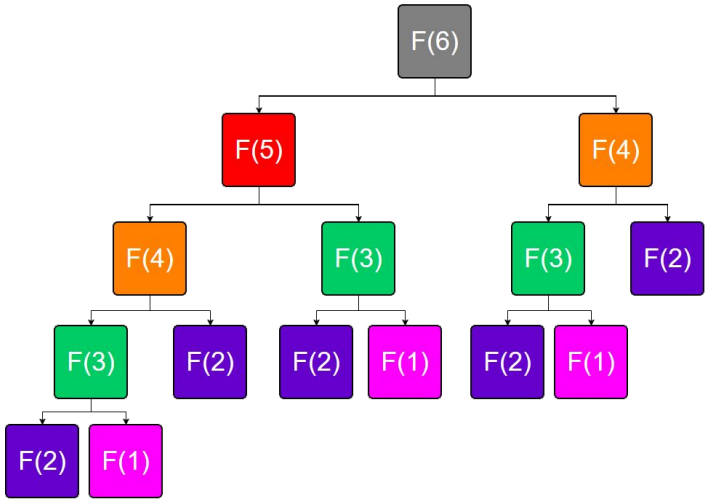
	1	2	3	4	5	6
f[]	1	1	2	3	-	-
caled[]	1	1	1	1	0	0

Fibonacci Sequence

Let's dry run together!

Called F(6) -> F(5) -> F(3)

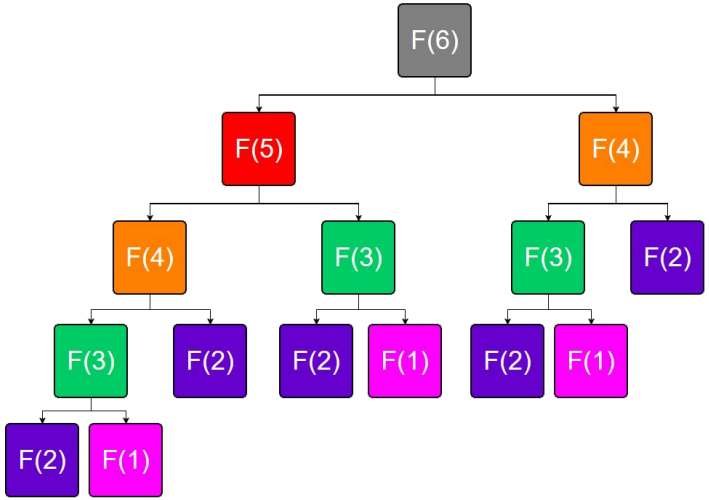
caled[3] is 1, return f[3]



	1	2	3	4	5	6
f[]	1	1	2	3	-	-
caled[]	1	1	1	1	0	0

Fibonacci Sequence

Let's dry run together!



Called F(6) -> F(5)

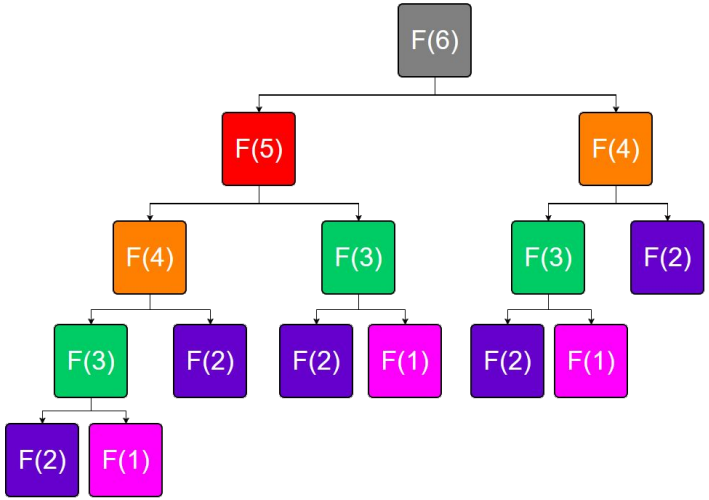
f[5] = F(4) + F(3)

caled[5] = 1

	1	2	3	4	5	6
f[]	1	1	2	3	5	-
caled[]	1	1	1	1	1	0

Fibonacci Sequence

Let's dry run together!



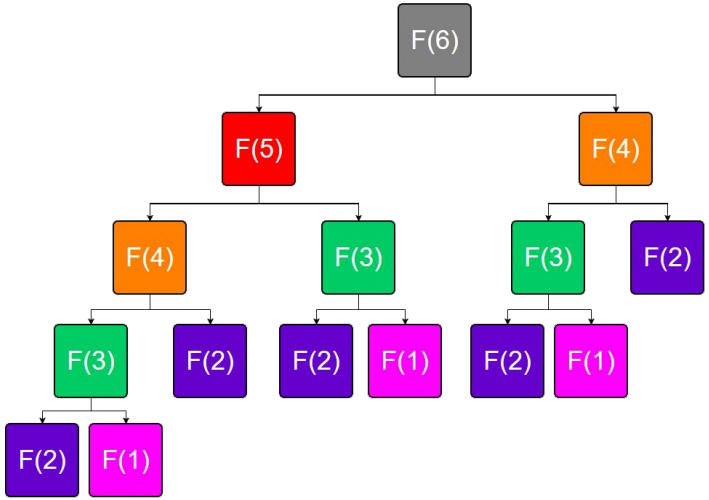
Called F(6) -> F(4)

caled[4] is 1, return f[4]

	1	2	3	4	5	6
`f[]`	1	1	2	3	5	-
`caled[]`	1	1	1	1	1	0

Fibonacci Sequence

Let's dry run together!



Called F(6)

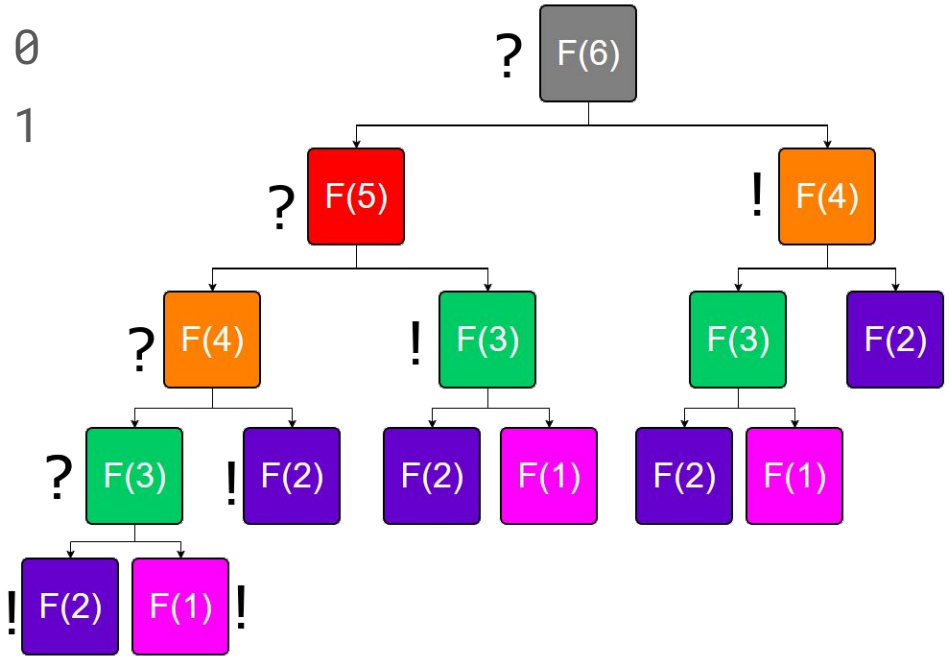
f[6] = F(5) + F(4)

caled[6] = 1

	1	2	3	4	5	6
f[]	1	1	2	3	5	8
caled[]	1	1	1	1	1	1

Fibonacci Sequence

```
?: caled[n] = 0
!: caled[n] = 1
```



Fibonacci Sequence

There are some time saved, but how much?

Each F(n) will be call almost 2 times. (called by F(n+1) & F(n+2))

F() was called about n*2 times, which is O(n)

We successfully reduce the time complexity to O(n) by using O(n) amount of memory!

Fibonacci Sequence

Let's change the direction of thinking.

If we know we have to calculate smaller F(n) first to get a bigger F(n),

why don't we start from F(3) to F(n)?

Fibonacci Sequence

```
int f[101]; // values of F(n)
int F(int n) {
     f[1] = f[2] = 1;
     for(int i = 3; i <= n; i++) {
          f[i] = f[i-1] + f[i-2];
     }
     return f[n];
}
```

Fibonacci Sequence

Let's dry run together!

Initial State

	1	2	3	4	5	6
f[]	1	1				

Fibonacci Sequence

Let's dry run together!

i = 3,

f[i] = f[i-1] + f[i-2]

	1	2	3	4	5	6
f[]	1	1	2			

Fibonacci Sequence

Let's dry run together!

i = 4,

f[i] = f[i-1] + f[i-2]

	1	2	3	4	5	6
f[]	1	1	2	3		

Fibonacci Sequence

Let's dry run together!

i = 5,

f[i] = f[i-1] + f[i-2]

	1	2	3	4	5	6
f[]	1	1	2	3	5	

Fibonacci Sequence

Let's dry run together!

i = 6,

f[i] = f[i-1] + f[i-2]

	1	2	3	4	5	6
f[]	1	1	2	3	5	8

Top-down & Bottom-up DP

We have just seen two approach of DP.

The First one is call Top-down DP, and

The second one is call Bottom-up DP.

What's the difference?

Top-down & Bottom-up DP

Top-down:

```
bool caled[101]; // calculated?, let's assume caled[1] = caled[2] = 1, else = 0
int f[101]; // values of F(n), let's assume f[1] = f[2] = 1
int F(int n) {
     if(!caled[n]) {
          f[n] = F(n-1);
          f[n] += F(n-2);
          caled[n] = 1;
     }
     return f[n];
}
```

Top-down & Bottom-up DP

Bottom-up:

```
int f[101]; // values of F(n)
int F(int n) {
     f[1] = f[2] = 1;
     for(int i = 3; i <= n; i++) {
          f[i] = f[i-1] + f[i-2];
     }
     return f[n];
}
```

Top-down & Bottom-up DP

Top-down DP

- Intuitive (**I** think)
- No need to care the order of computation.
- Messy (**I** think) and harder to debug.

Top-down & Bottom-up DP

Bottom-up DP

- So clean
- The subproblems must be solved first when transitioning
- Some techniques and tricks will be discussed in DP (II) & (III) can only been done easily in Bottom-up DP - **important**
- Usually run much faster - **important when time limit is narrow**
- We will mainly focus on Bottom-up DP.

Some classical DP problems

Maximum Subarray Sum

HKOJ 01010 Diamond Chain

Let a[] be a array with n values of a_1, a_2,..., a_n.

A subarray of a[] is define by delete some prefix of a[] and delete some suffix of a[].

- If a[] = [0, 1, 2, 3];

  [0, 1, 2, 3], [0, 1], [2] and [] are the subarray of a[] but [0, 3] isn't.

Find the maximum subarray from all of the possible subarray.

Maximum Subarray Sum

HKOJ 01010 Diamond Chain

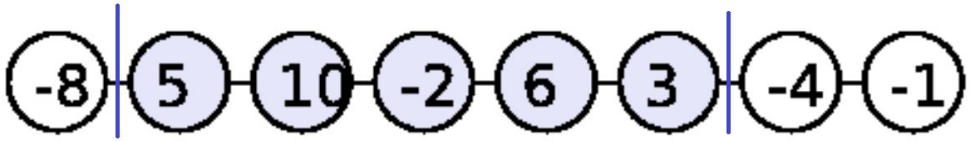
Let's see the sample.

The optimal solution of this sample is to delete (-8) from the front and (-4) & (-1) from the end.

SAMPLE TESTS

	Input	Output
1	8 -8 5 10 -2 6 3 -4 -1	22

EXPLANATION



Maximum Subarray Sum

What is the naive solution? Just loop all the possible subarrays in this array.

```
int ans = 0;
for(int i = 1; i <= n; i++) { // all subarray start with index i
    for(int j = i; j <= n; j++) { // all subarray end with index j
        int sum = 0;
        for(int k = i; k <= j; k++) { // calculate the sum from i to j
            sum += a[k];
        }
        ans = max(ans, sum);
    }
}
```

Maximum Subarray Sum

Time complexity: $O(n^3)$

Way to slow for n = 100000.

If you know partial sum before (27/02 session 2), you may reduce it to $O(n^2)$ by replacing the innermost loop.

Still TLE.

Maximum Subarray Sum

(**A[i : j]** = the subarray of **A** that start at **i** and end at **j**)

For example if the length of **A** is **2**.

We can choose the answer from A[1 : 1], A[1 : 2] or A[2 : 2].

What if we want to add another element end the back of **A**.

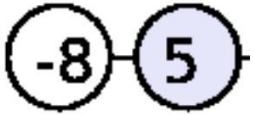
We can choose the answer from A[1 : 1], A[1 : 2], A[2 : 2], A[1 : 3], A[2 : 3] or A[3 : 3].

Note that of those new options, A[1 : 3] and A[2 : 3] can be calculated easily by A[1 : 2] and A[2 : 2] by adding $\mathbf{A}_3$, and A[3 : 3] is just $\mathbf{A}_3$ itself.

Maximum Subarray Sum

If |A| = 2, the possible subarray:

A[1 : 1] 

A[1 : 2] 

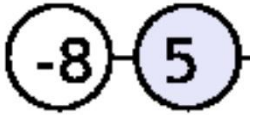
A[2 : 2] 

Maximum Subarray Sum

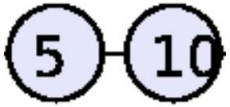
If |A| = 3, the possible subarray:

A[1 : 1] 

A[1 : 3] 

A[1 : 2] 

A[2 : 2] 

A[2 : 3] 

A[3 : 3] 

Maximum Subarray Sum

To know A[1 : 3] or A[2 : 3] is better, we only need to care which is the better one between A[1 : 2] and A[2 : 2].

That mean we only need to store the better one between A[1 : 2] and A[2 : 2].

And that compare it to the value A[3 : 3] = $\mathbf{A_3}$.

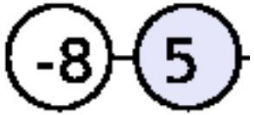
There we have the optimal subarray that end with $\mathbf{A_3}$.

In general, we only need to store the optimal value of all subarray that subarray that end with $\mathbf{A_i}$ to compute the optimal value of **i+1** one!

Maximum Subarray Sum

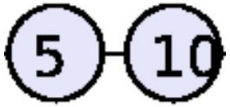
If |A| = 3, the possible subarray:

A[1 : 1] 

A[1 : 2] 

A[2 : 2] 

A[1 : 3] 

A[2 : 3] 

A[3 : 3] 

Maximum Subarray Sum

For each $\mathbf{a_i}$, we can decide added it to some subarrays that end with $\mathbf{a_{i-1}}$.

Or we just start a new subarray that start with $\mathbf{a_i}$.

For each a_i, we can decide added it to some subarrays that end with a_{i-1}.

What if we somehow know the optimal subarray that end with $\mathbf{a_{i-1}}$?

We can choose connecting $\mathbf{a_i}$ to the optimal subarray that end with $\mathbf{a_{i-1}}$ or just $\mathbf{a_i}$ itself become the subarray that end with $\mathbf{a_i}$.

Maximum Subarray Sum

Ok, but how?

Let dp[i] = the Maximum Subarray Sum that end with $\mathbf{a_i}$.

For i = 1, **dp[1]** is $\mathbf{a_1}$ itself only.

For i = 2, **dp[2]** is the choose between adding **dp[1]** to $\mathbf{a_2}$ and $\mathbf{a_2}$ itself only.

For i > 2, **dp[i]** is the choose between adding **dp[i-1]** to $\mathbf{a_i}$ and $\mathbf{a_i}$ itself only.

Maximum Subarray Sum

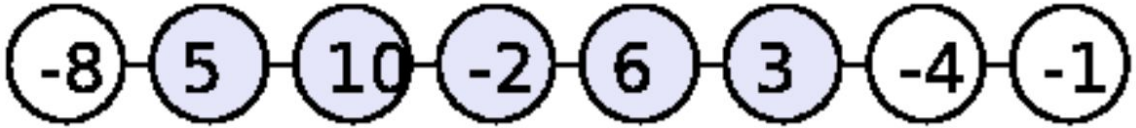
- **dp[i]** can be constructed easily by **dp[i-1]**
- **dp[i]** will be used by **dp[i+1]**

It can DP!

```
dp[1] = a[1];
int ans = max(0, dp[1]);
for(int i = 2; i <= n; i++) {
    dp[i] = dp[i-1] + a[i]; // connecting a[i] to the optimal subarray before
    dp[i] = max(dp[i], a[i]); // leave a[i] alone
    ans = max(ans, dp[i]); // choose the best subarray
}
```

Maximum Subarray Sum

Let's dry run together!

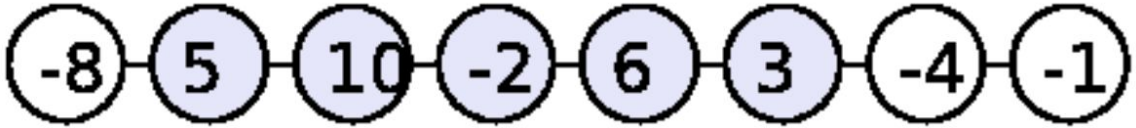


Initial state

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	-	-	-	-	-	-	-

Maximum Subarray Sum

Let's dry run together!



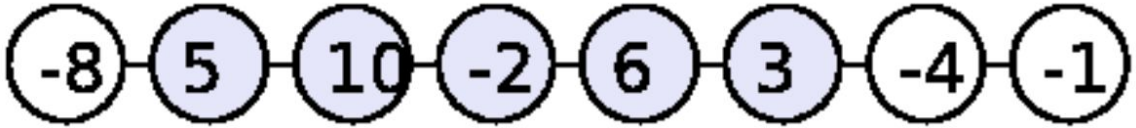
i = 2

```
dp[i] = dp[i-1] + a[i]; // -3
dp[i] = max(dp[i], a[i]); // max(-3, 5)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	-	-	-	-	-	-

Maximum Subarray Sum

Let's dry run together!



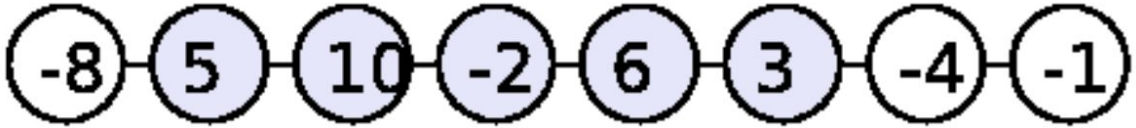
i = 3

```
dp[i] = dp[i-1] + a[i]; // 15
dp[i] = max(dp[i], a[i]); // max(15, 10)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	15	-	-	-	-	-

Maximum Subarray Sum

Let's dry run together!



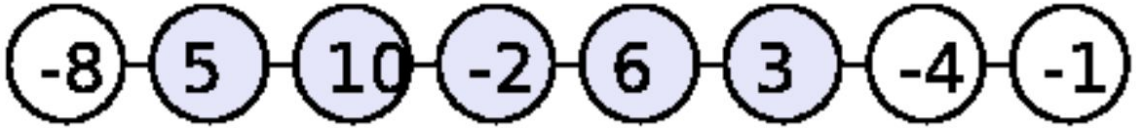
i = 4

```
dp[i] = dp[i-1] + a[i]; // 13
dp[i] = max(dp[i], a[i]); // max(13, -2)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	15	13	-	-	-	-

Maximum Subarray Sum

Let's dry run together!



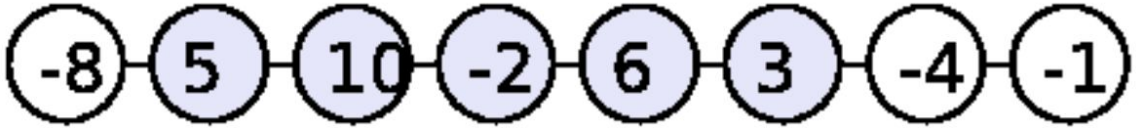
i = 5

```
dp[i] = dp[i-1] + a[i]; // 19
dp[i] = max(dp[i], a[i]); // max(19, 6)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	15	13	19	-	-	-

Maximum Subarray Sum

Let's dry run together!



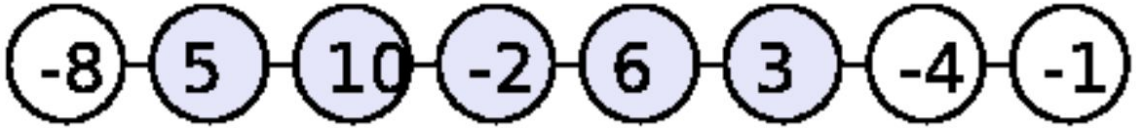
i = 6

```
dp[i] = dp[i-1] + a[i]; // 22
dp[i] = max(dp[i], a[i]); // max(22, 3)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	15	13	19	22	-	-

Maximum Subarray Sum

Let's dry run together!



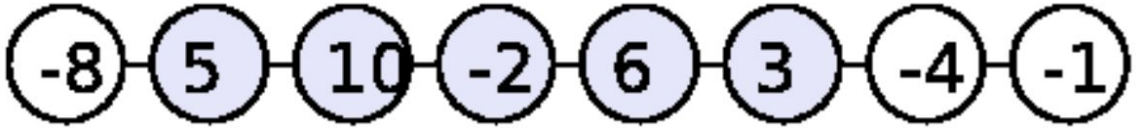
i = 7

```
dp[i] = dp[i-1] + a[i]; // 18
dp[i] = max(dp[i], a[i]); // max(18, -4)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	15	13	19	22	18	-

Maximum Subarray Sum

Let's dry run together!



i = 8

```
dp[i] = dp[i-1] + a[i]; // 17
dp[i] = max(dp[i], a[i]); // max(17, -1)
```

	1	2	3	4	5	6	7	8
a[]	-8	5	10	-2	6	3	-4	-1
dp[]	-8	5	15	13	19	22	18	17

Maximum Subarray Sum

There is a top-down version for your reference.

```
bool caled[100001]; // let's assume caled[1] = 1, else = 0
int dp[100001]; // let's assume dp[1] = a[i]
int DP(int i) {
     if(!caled[i]) {
          dp[i] = DP(i-1) + a[i];
          dp[i] = a[i];
          caled[i] = 1;
     }
     return dp[i];
}
```

Maximum Subarray Sum

Time Complexity become O(n) by using O(n) extra memory. So good.

To discuss DP solution more formally, we can use something call transitional formula:

dp[i] = max(dp[i-1] + a[i], a[i])

Be carefully how to define the transition formula and the base cases(in this case dp[1]).

Maximum Subarray Sum

Be Careful that this DP solution only apply on this “offline” problem. (i.e. the values of a_i won’t change)

“Online” problem like M0923 require some special Data Structure.

Jumper

There **n** building on a line and the height of i-th building is $\mathbf{h_i}$.

There is a man who wants to go to building n from building 1 by jumping rooftop to rooftop.

He can jump really high and each jump cost him $\mathbf{|h_i - h_j|}$ from **i** to **j**.

But he can't jump too far, so the gaps between building he can jump is limited to **k**. I.e. **j - i <= k**.

Also he can only jump forward.

What is the minimum energy costed?

Jumper

The “Naive” solution:

```
int sol(int i) {
     if(i == n)
          return 0;
     int tmp = INF; // some very big number
     for(int j = i+1; j <= min(n, i+k); i++) {
          tmp = min(tmp, sol(j) + abs(h[i] - h[j]));
     }
     return tmp;
}
```

Jumper

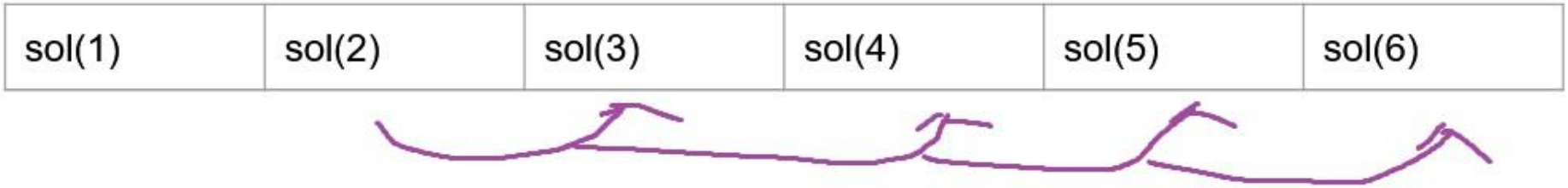
Imagine k = 5

i = 1

Jumper

Imagine k = 5

i = 2



Jumper

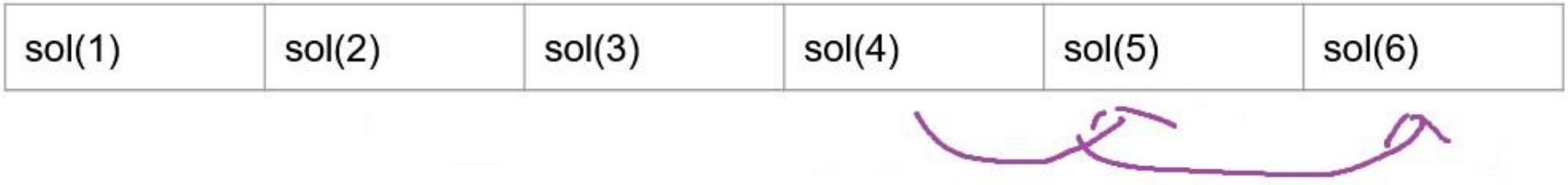
Imagine k = 5

i = 3

Jumper

Imagine k = 5

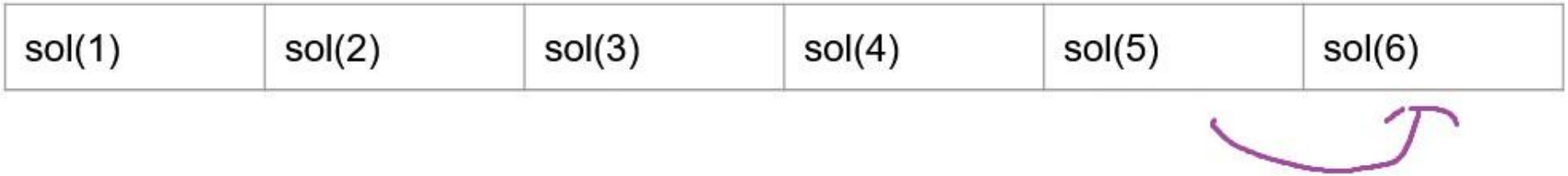
i = 4



Jumper

Imagine k = 5

i = 5



Jumper

It is kind of like the solution of Fibonacci Sequence.

Which mean it is really really really slow.

But it also mean it can be improved. Right?

Jumper

```
int sol(int i) {
     if(i == n)
          return 0;
     int tmp = INF; // some very big number
     for(int j = i+1; j <= min(n, i+k); i++) {
          tmp = min(tmp, sol(j) + abs(h[i] - h[j]));
     }
     return tmp;
}
```

sol() has been called many times.

Jumper

Let’s think about what is sol(i).

sol(i) is the minimum cost jumping from building i to n.

sol(i) will be called by sol(i-k) to sol(i-1).

Why don’t we just memorize the value of sol(i)?

Jumper

Let's **dp[i]** = the minimum cost from **i** to **n**.

dp[n] i.e. the minimum cost from **n** to **n** is **0**.

dp[i] = min(dp[j] + $|h_i - h_j|$) for each **j** he can jump from **i**, i.e. **i+1 to i+k**.

dp[1] will become the answer.

Jumper

```
dp[n] = 0;
for(int i = n-1; i >= 1; i--) {
    dp[i] = INF; // some really big number
    for(int j = i+1; j <= min(n, i+k); j++) {
        dp[i] = min(dp[i], dp[j] + abs(h[i] - h[j]));
    }
}
cout << dp[1];
```

Jumper

There is a top-down version for your reference.

```
bool caled[100001]; // let's assume caled[n] = 1, else = 0
int dp[100001]; // let's assume dp[n] = 0
int DP(int i) {
      if(!caled[i]) {
            dp[i] = INF; // some really big number
            for(int j = i+1; j <= min(n, i+k); j++) {
                  dp[i] = min(dp[i], DP(j) + abs(h[i] - h[j]));
            }
            caled[i] = 1;
      }
      return dp[i];
}
```

Jumper

Time complexity become O(n*k).

Note that you may take dp[i] = the minimum cost from 1 to i, and try work on the transition formula and the base case. Both should work perfectly ok.

Longest Common Subsequence

Given two strings **S** & **T**, find the Longest Common Subsequence.

Let a subsequence of S is the result of deleting some(0 to All) characters in S.

E.g. **S** = “abcde”; “”, “abcde”, “ae” are the subsequences of **S**, but “aeb” is not.

Remind that **Subsequence != Substring.** (Substring is contiguous)

If S = “abcdef”, T = “ebbdaf”,

There Longest Common Subsequence = “bdf”

Longest Common Subsequence

I give up on thinking the brute force solution.

Let's think about a DP solution.

What if we define **dp[i]** = the length of Longest Common Subsequence of $\mathbf{S_1 ... S_i}$ and **T**?

dp[i] doesn't help because we don't know where it end on **T**. :(

Well just add another dimension. :)

Longest Common Subsequence

(1-based)

Let **dp[i][j]** = the length of Longest Common Subsequence of $\mathbf{S_1...S_i}$ and $\mathbf{T_1...T_j}$.

If **S[i] != T[j]**, we consider **dp[i-1][j]** and **dp[i][j-1]**.

I.e. **dp[i][j]** = max(dp[i-1][j], dp[i][j-1]).

If **S[i] == T[j]**, other consider **dp[i-1][j]** and **dp[i][j-1]**, we may add this character to the Longest Common Subsequence of $\mathbf{S_1...S_{i-1}}$ and $\mathbf{T_1...T_{j-1}}$.

I.e. **dp[i][j]** = max(dp[i-1][j], dp[i][j-1], dp[i-1][j-1] + 1).

dp[0][j] = dp[i][0] = 0 for all i, j because of empty string.

Longest Common Subsequence

		T[]	e	b	b	d	a	f
	i\j	0	1	2	3	4	5	6
S[]	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	1	1
b	2	0	0	1	1	1	1	1
c	3	0	0	1	1	1	1	1
d	4	0	0	1	1	2	2	2
e	5	0	1	1	1	2	2	2
f	6	0	1	1	1	2	2	3

Longest Common Subsequence

(Let SL = |S| and TL = |T|)

```
for(int i = 1; i <= SL; i++) {
    for(int j = 1; j <= TL; j++) {
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        if(S[i] == T[j]) {
            dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
        }
    }
}
cout << dp[SL][TL];
```

Longest Common Subsequence

Time complexity: O(|S|*|T|).

This program only give us longest length of the Longest Common Subsequence. What if we want to know the actually Subsequence?

- For each i and j, we **choose** the optimal **dp[i][j]** between **dp[i-1][j]**, dp[i][j-1] and dp[i-1][j-1] + 1.
- Why don't we **memorize** what we had chose for each i and j?
- Finally, **Backtrack** it from i = |S|, j = |T| and record where we choose **dp[i-1][j-1] + 1**.
- You may try it as exercise. :)

Longest Common Subsequence

		T[]	e	b	b	d	a	f
	i\j	0	1	2	3	4	5	6
S[]	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	1	1
b	2	0	0	1	1	1	1	1
c	3	0	0	1	1	1	1	1
d	4	0	0	1	1	2	2	2
e	5	0	1	1	1	2	2	2
f	6	0	1	1	1	2	2	3

Knapsack Problem

There are **N** items. Each with a Weight $\mathbf{w_i}$ and a value $\mathbf{v_i}$.

You have a bag that can only carry **at most** total weight of **K**. I.e. For any set of items that can be put in the bag, the sum of $\mathbf{w_i}$ is smaller than or equal to **K**.

Note that you can't duplicate or divide the items.

We want to find a set of **N** such that **total weight <= K**, the total value is the largest.

Knapsack Problem

Let's Brute force it.

(0-based)

For **i** from **0 to (1 << N)-1 ($00...0_2$ to $11...1_2$)**, if the **j-th bit of i** is **1**, we put the **j-th** item into the bag.

E.g. if **i = 5_{10} (101_2)**, we put item **0** and **2** into the bag.

Knapsack Problem

```
for(int i = 0; i < (1 << N); i++) {
     int sum_w = 0, sum_v = 0;
     for(int j = 0; j < N; j++) {
          if(i & (1<<j) != 0) { // is the j-th bit is 1
               sum_w += w[j];
               sum_v += v[j];
          }
     }
     if(sum_w <= K) // is this set valid
          ans = max(ans, sum_v);
}
```

Knapsack Problem

Time Complexity: $O(2^N)$

So slow, let's improve it.

Intuitively, we may think about put the items with largest v_i/w_i first.

But it will WA on this version of the problem. Why?

Remind that we **can't** divide the items.

You may think about a counter example.

Knapsack Problem

For each Item **i**, we can choose to put it into the bag or not.

Let assume after putting item **i** into the bag, total weight of the set is **j**.

What if we know about the optimal solution of using a set of items **1...(i-1)** which the **total weight = j - w[i]**?

Here we find the subproblem of this problem.

Knapsack Problem

Let **dp[i][j]** = the optimal solution of using a set of items **1…i** which the total weight is **j**.

For each **dp[i][j]**, we consider put item **i** or not.

I.e. **dp[i][j]** = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])

dp[0][i] = 0

The answer will be max(**dp[N][j]**) for each j from 0 to K.

Knapsack Problem

```
for(int i = 1; i <= N; i++) {
    for(int j = 0; j <= K; j++) {
        dp[i][j] = dp[i-1][j];
        if(j >= w[i]) {
            dp[i][j] = max(dp[i][j], dp[i-1][j - w[i]] + v[i]);
        }
    }
}
```

Knapsack Problem

Time Complexity: O(N*K)

There are some variations you may think about it (from 2019 Materials):

- The bag and the items have volume now, both weight and volume can't exceed.
- There are infinite amount of each item. (J181 subtasks 2 & 4)
- There are some amount of each item.
- Some items i can be picked only if item j has been picked before.

Round-up

In this session, we had discussed about a lot of problems. During this, the common theme of the discussion is to define the states and the transitional formula.

When encounter a problem, that you believe there are a DP solution, define states and formula clearly which help you to analyze will it TLE or MLE, and code faster with less bug.

More Practice Problems

(CF = CodeForces)

• Maximum subarray sum ○ 01010 Diamond Chain ○ 01016 Diamond Ring ○ M0822 Diamond Chain II	• Parentheses ○ CF 628C FamilDoor and Brackets
• Knapsack problem ○ 05011 Coin ○ T043 Need for speed	• Combinatorics ○ CF 553A Kyoyaand ColouredBalls
• Palindrome ○ I0011 Palindrome ○ CF 607B Zuma	• Probabilities ○ CF 540D Bad Luck Island