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

Dynamic Programming (I)

Kelvin Chow {Lrt1088}

2023-04-01

Table of Contents

• Introduction to DP
  ◦ Why DP?
  ◦ How to DP?
  ◦ Fibonacci Sequence
  ◦ Top-down & Bottom-up DP
	• Some classical DP problems
	  ◦ Maximum Subarray Sum
	  ◦ Jumper
	  ◦ Longest Common Subsequence
	  ◦ Knapsack Problem
	  ◦ Matrix Multiplication
	  ◦ Longest Increasing Subsequence
	  ◦ Longest Palindrome Subsequence

• Round-up	• More Practice Problems

Tasks on HKOJ will be discussed today

01010 Diamond Chain

01054 Matrix-chain Multiplication

M1222 Longest Increasing Subsequence

If you are too strong then

(Provided by Benson)

JOI18_asceticism Asceticism

JOI21_ho_t3 Group Photo

IOI20_biscuits Packing Biscuits

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 3 DP sessions in ADV this year
  - DP (I) - 18/02 // Introduction
  - DP (II) - 11/03 // Some common techniques
  - DP (III) - 01/04 // Optimization tricks
- In this session, we will discuss about what is DP and then get us familiar with DP by solving some DP problems.

Week 7 2023-02-18	2450	Dynamic Programming (I) Chow King Wang
Week 8 2023-02-25		
Week 9 2023-03-04	2450	Graph (II) Cheung Cheuk Nam
Week 10 2023-03-11	2450	Dynamic Programming (II) Ng Yau Fu
Week 11 2023-03-18	2450	Data Structures (III) Liu Man Kai
Week 12 2023-03-25	2450	Graph (III) Chow King Wang
Week 13 2023-04-01	2450	Dynamic Programming (III) Ng Yau Fu

How to DP?

Prerequisites:

- Recursion
- Divide & Conquer
- Big O notation - to analyze Time and Memory complexities

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 **reused** to constructed a larger subproblems

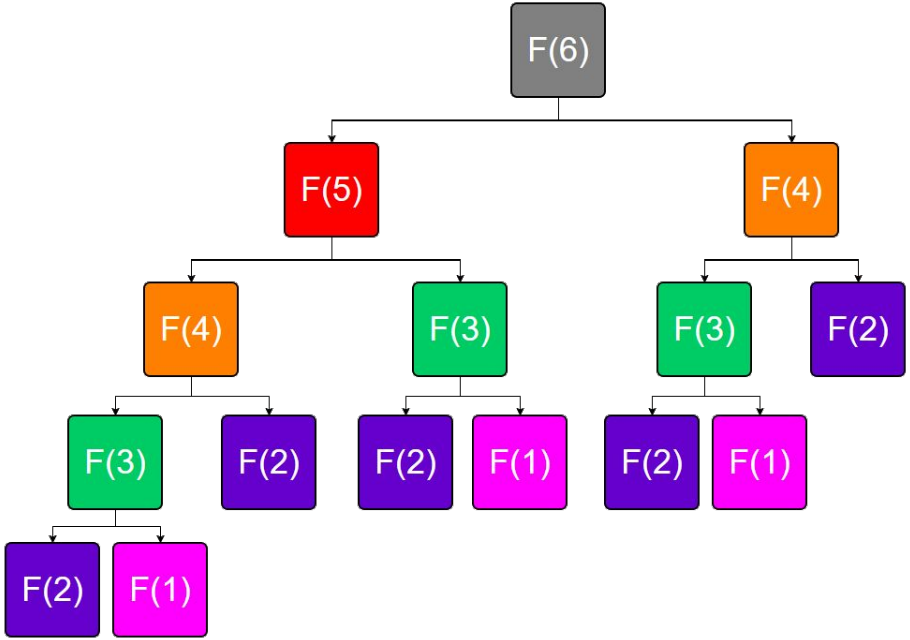
Fibonacci Sequence

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



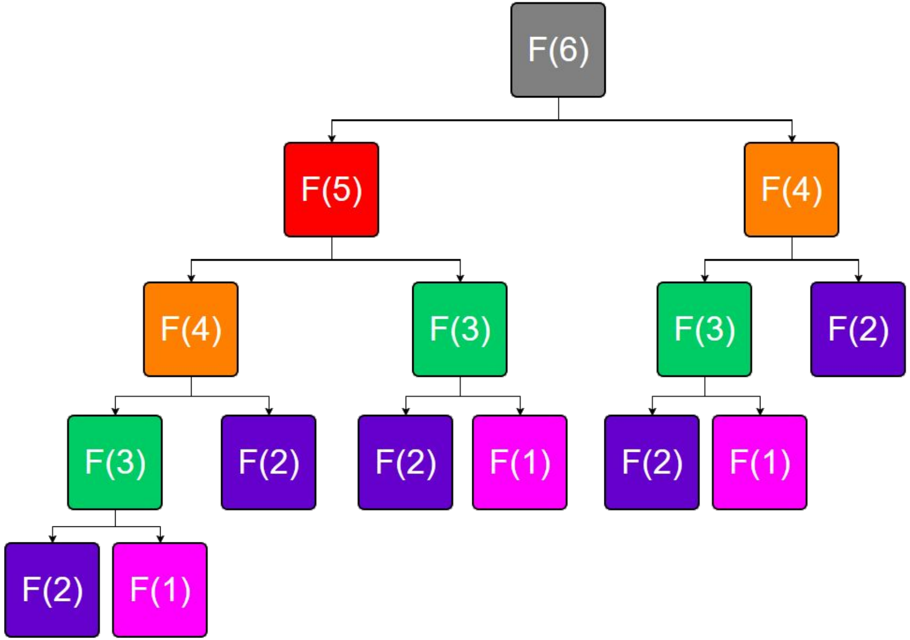
Fibonacci Sequence

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?

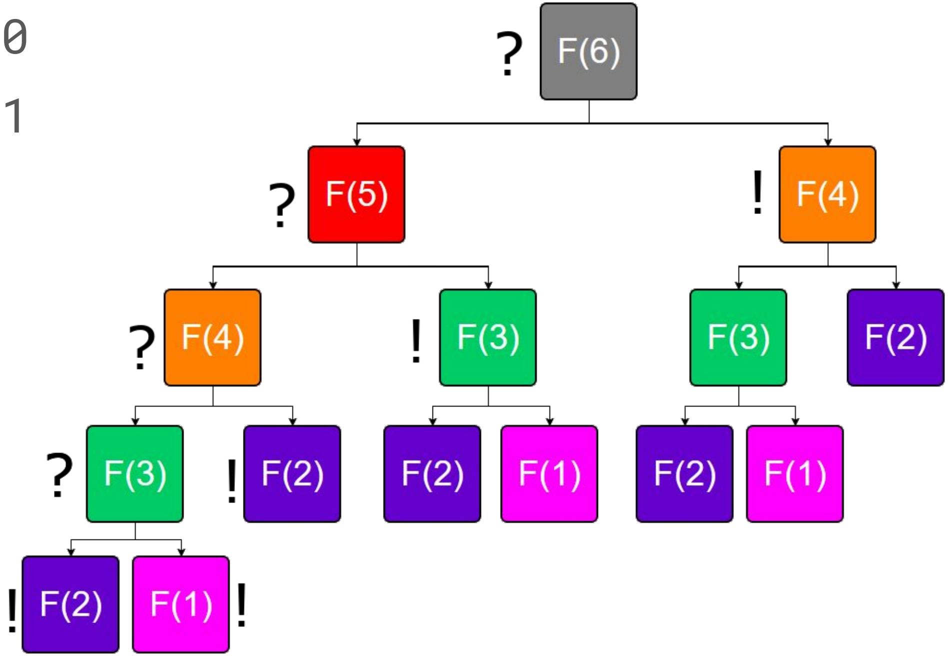
Fibonacci Sequence

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

```
?: 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];
}
```

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**
- 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.

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, 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

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

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

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

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 $|\mathbf{h_i} - \mathbf{h_j}|$ 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. $\mathbf{j - i <= k}$.

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

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 $\mathbf{S_1 \ldots S_i}$ 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 $\mathbf{S_1 \ldots S_i}$ and $\mathbf{T_1 \ldots T_j}$.

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

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. :)

Let’s have a Break

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 **0...(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 **0…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][w[0]] = v[0], else 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.

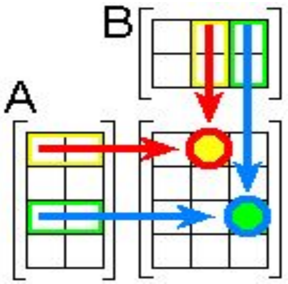
Matrix Multiplication

For whom may not know what Matrix Multiplication is, Let's briefly talk about it first.

A **m**x**n** matrix is like a 2D array of numbers with **m** rows and **n** columns.

Let $\mathbf{A_1}$ and $\mathbf{A_2}$ be matrices, $\mathbf{A_1A_2}$ only exist **if and only if** no. of **columns** of $\mathbf{A_1}$ = no. of **rows** $\mathbf{A_2}$

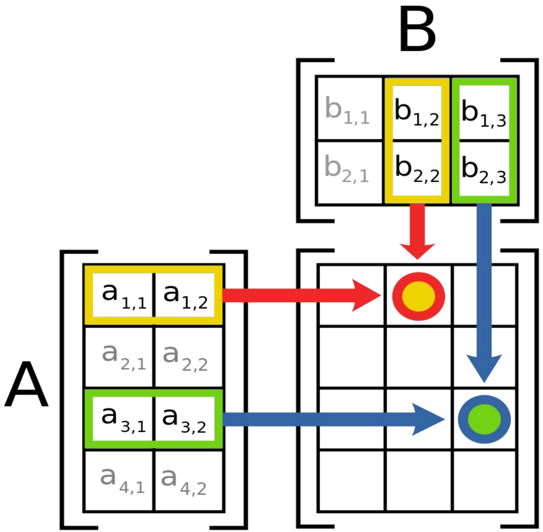
Let $\mathbf{A_1}$ is an **m**x**p** matrix and $\mathbf{A_2}$ is an **p**x**n** matrix, by some operation, $\mathbf{A_1A_2}$ will be a **m**x**n** matrix.



(from Wikipedia)

Matrix Multiplication

(from Wikipedia)



Matrix Multiplication

Properties of matrix multiplication:

- Matrix multiplication is **associative**, i.e. $A_1(A_2A_3) = (A_1A_2)A_3$
- But matrix multiplication is **not commutative**, i.e. $A_1A_2 \mathrel{!=} A_2A_1$ generally.

Matrix Multiplication

HKOJ 01054 Matrix-chain Multiplication

Let define the cost of performing multiplication of two matrix $\mathbf{A_1}$ - $\mathbf{m{\times}p}$ matrix and $\mathbf{A_2}$ - $\mathbf{p{\times}n}$ matrix is equal to $\mathbf{m*p*n}$.

There **N+1** number $\mathbf{p_i}$ which the **i-th** matrix $\mathbf{A_i}$ is a $\mathbf{p_i{\times}p_{i+1}}$ matrix.

Find the minimum cost to finish this matrix multiplication.

Remind that Matrix multiplication is **associative.**

Matrix Multiplication

Sample test case:

	Input	Output
1	3 5 10 15 5	1000

	Matrices			Total cost
1.	5x10	10x15	15x5	0
2.	5x15	15x5		750
3.	5x5			1125

	Matrices			Total cost
1.	5x10	10x15	15x5	0
2.	5x10	10x5		750
3.	5x5			1000

Matrix Multiplication

Brute Force:

- Let's think about it, if we to perform the Multiplication of a chain $A_i \ldots A_j$, where **$i < j$**, the **final** operation must be the Multiplication of the **resultant** matrix of chain $A_i \ldots A_{k-1}$ and the **resultant** matrix of chain $A_k \ldots A_j$, for some **k** where **$i < k <= j$**. Which is the Multiplication of between an $p_i \times p_k$ matrix and an $p_k \times p_{j+1}$ matrix, the resultant matrix would be $p_i \times p_{j+1}$ with cost $p_i * p_k * p_{j+1}$.
- Same as $A_i \ldots A_j$, $A_i \ldots A_{k-1}$ and $A_k \ldots A_j$ can obtain in a same process as above.
- We can solve it recursively!

Matrix Multiplication

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

Matrix Multiplication

The time complexity is large by having a look at it. :(

Again, sol() had been call so many time.

Let's define **dp[i][j]** like before.

Matrix Multiplication

Let **dp[i][j]** = the minimum cost to perform the multiplication to the chain of matrices $\mathbf{A_i \ldots A_j}$.

Similar to the brute force solution, the transition formula is:

dp[i][j] = min(dp[i][k-1] + dp[k][j] + p[i]*p[k]*p[j+1]) for all **k** where **i < k <= j**.

dp[i][i] (i.e. the chain have $\mathbf{A_i}$ only) = 0

Matrix Multiplication

```
for(int i = 1; i <= N; i++) {
    for(int j = 1; j <= N; j++) {
        dp[i][j] = INF; // some really big number
        for(int k = i+1; k <= j; k++) {
            dp[i][j] = min(dp[i][j], dp[i][k-1] + dp[k][j] + p[i] * p[k] * p[j+1]);
        }
    }
}
```

Matrix Multiplication

WA why??? :(

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**
- We will mainly focus on Bottom-up DP.

Matrix Multiplication

It will be somewhat complicated to design the subproblems to be solved first in this problem. (You may try it).

Don’t forget we still have top-down approach.

Matrix Multiplication

```
int sol(int i, int j) {
     if(i == j)
          return 0;
     if(!caled[i][j]) {
          dp[i][j] = INF; // some really big number
          for(int k = i+1; k <= j; k++) {
               dp[i][j] = min(dp[i][j], sol(i, k-1) + sol(k, j) + p[i]*p[k]*p[j+1]);
          }
          caled[i][j] = 1;
     }
     return dp[i][j];
}
```

Matrix Multiplication

Time Complexity:

- Each state have a time complexity of O(N)
- There are O(N^2) states
- The resultant time complexity: O($N*N^2$) = O(N^3)

Longest Increasing Subsequence

HKOJ M1222 Longest Increasing Subsequence

Given a sequence $\mathbf{a_1, a_2, a_3, \ldots, a_N}$, find the length of the Longest Increasing Subsequence

Subsequence again.

Increasing Subsequence is the subsequence of a, such that for elements of this subsequence b, **b[i] < b[i+1]** for all applicable i.

E.g. the LIS of **{3, 4, 2, 9, 1, 9, 6, 7}** is **{3, 4, 6, 7}**

Longest Increasing Subsequence

Let **dp[i]** = the LIS of **a[1], ..., a[i]** that **ended with a[i]**.

Is this enough to compute **dp[i]** by **dp[i-1]** only?

No, it is because **a[i]** may not be larger than **a[i-1]**.

We need to consider all **dp[j]** where **1 <= j < i**.

Longest Increasing Subsequence

```
for(int i = 1; i <= N; i++) {
    dp[i] = 1; // a[i] itself is a increasing subsequence
    for(int j = 1; j < i; j++) {
        if(a[i] > a[j]) {
            dp[i] = max(dp[i], dp[j] + 1);
        }
    }
}
```

Longest Increasing Subsequence

Time Complexity: $O(N^2)$

Is not hard right?

CONSTRAINTS

In all test cases, $1 \le N \le 100000$, $1 \le a_i \le 10^9$.

In 50% test cases, $1 \le N \le 3000$.

Oh no! Let's improve it.

Longest Increasing Subsequence

Imagine there are some Increasing Subsequence **S** of $\mathbf{a_1, ..., a_{i-1}}$ with length k.

If we want to append $\mathbf{a_i}$ to the one of the **S** to form an Increasing Subsequence

of $\mathbf{a_1, ..., a_i}$ with length **k+1**. What will we choose?

$\mathbf{A_i}$ can only append to the Subsequence that the **last** element of that is **smaller** than $\mathbf{A_i}$. If it is possible, the **last** element of the **new** Subsequence become $\mathbf{A_i}$.

Because only the last element can only affect our choice, we can consider the S with the **smallest last element only** right? In which the elements is not the lastest doesn't matter right?

Longest Increasing Subsequence

Let **f[i][k]** = the smallest last element of the Increasing Subsequence of $\mathbf{a_1}$**, ..., $\mathbf{a_i}$** such that the length this Increasing Subsequence is **k**.

For every **f[i][k]**, if **f[i-1][k-1]** is smaller than **a[i]**, we can append **a[i]** to it and form a Increasing Subsequence with length k. That mean we can compare **a[i]** to **f[i-1][k]**, i.e. **f[i][k]** = min(**a[i]**, **f[i-1][k]**).

If **f[i-1][k-1]** is not smaller than **a[i]**, **f[i][k]** can only be **f[i-1][k]**,

i.e. **f[i][k]** = **f[i-1][k]**.

At start, **f[i][0]** = -INF(Always possible to append a[i] in to an empty subsequence.), and other = INF(Indicate there no such subsequence)

Longest Increasing Subsequence

i		1	2	3	4	5	6	7	8
a[i]		3	4	2	9	1	9	6	7
f[i][0]	-INF	-INF	-INF	-INF	-INF	-INF	-INF	-INF	-INF
f[i][1]	INF	3	3	2	2	1	1	1	1
f[i][2]	INF	INF	4	4	4	4	4	4	4
f[i][3]	INF	INF	INF	INF	9	9	9	6	6
f[i][4]	INF	INF	INF	INF	INF	INF	INF	INF	7

Longest Increasing Subsequence

Each column is increasing right? Why?

Everytime we want to append **a[i+1]**, we would like to greedily find the **largest f[i][k]** such that **f[i][k] < a[i+1]**, which give us **f[i+1][k+1] = a[i+1]** which is not larger than **f[i][k+1]**. (f[i][k] < a[i+1] <= f[i][k+1])

Otherwise, let's say **f[i][k-1] < f[i][k] < a[i+1]**, if we choose **f[i][k-1]**, considering **f[i+1][k]** = min(**a[i+1]**, **f[i][k]**), **f[i+1][k]** would not be smaller than **a[i+1]** since **f[i][k] < a[i+1]**.

As I'd said before, we want to leave the smallest last element of each length **K** **f[i]** to compute **f[i+1]**.

Longest Increasing Subsequence

Wait. We need to copy every elements of **f[i]** to **f[i+1]**. It still cost us at least $O(n^2)$!

Actually we only need to consider **f[i]** to compute **f[i+1]**, and there will be **almost 1 element** will be change from **f[i]** to **f[i+1]**, since we only greedily find the **largest f[i][k]** such that **f[i][k] < a[i+1]**.

We can use a 1D array **g[]** to scan through **a[i]**.

g[0] = -INF

Longest Increasing Subsequence

a[]		3	4	2	9	1	9	6	7
g[0]	-INF	-INF	-INF	-INF	-INF	-INF	-INF	-INF	-INF
g[1]		3	3	2	2	1	1	1	1
g[2]			4	4	4	4	4	4	4
g[3]					9	9	9	6	6
g[4]									7

Longest Increasing Subsequence

```
int search_g(int t); // return the index of largest g[k] which is smaller than t
bool is_last_g(int i); // return is i the last index of g
void push_g(int t); // append t to back of g
int sol() {
    for(int i = 1; i <= N; i++) {
        int tar = search_g(a[i]);
        if(is_last_g(tar))
            push_g(a[i]);
        else
            g[tar+1] = a[i];
    }
}
```

Longest Increasing Subsequence

The Time Complexity is depend on how you implement search_g().

Linear Search will give us O(N) every query.

But Binary Search will give us **O(log N)** every query since g[] is **increasing**.

Total Time Complexity: O(N*log N)

The detail of the code is left as a exercise.

Longest Palindrome Subsequence

Yes, It's subsequence again.

Given a string **S** with length **N**.

The target is to find the longest Palindrome Subsequence.

Here, Palindrome is a string **A** with length **N** such that the reverse of the **A** is equal to the **A** itself. I.e. $A_i = A_{N-i+1}$ for all **i**. (1-based)

E.g. Longest Palindrome Subsequence of "abcabca" is "abcba"

Longest Palindrome Subsequence

Quite similar to Longest Common Subsequence discussed before, right?

Let’s define the states and the transitional formal like that.

Let **dp[i][j]** = the length of the Longest Palindrome Subsequence for the substring **S[i], S[i+1], ..., S[j-1], S[j]**.

Longest Palindrome Subsequence

For each **dp[i][j]**, we can either concatenate **S[i]** to **S[i+1], S[i+2], ..., S[j]** or concatenate **S[j] to S[i], ..., S[j-2], S[j-1]**. The values won't add up.

It give us **dp[i][j]** = max(**dp[i+1][j]**, **dp[i][j-1]**).

If **S[i] == S[j]**, other than above, we can also concatenate **S[i]** and **S[j]** to **S[i+1], ..., S[j-1]**, and the values will be incremented by 2, which give us

dp[i][j] = max(**dp[i][j]**, dp[i+1][j-1] + 2)

The base cases:

- **dp[i][j] = 0** if **i > j** // empty string
- **dp[i][j] = 1** if **i = j** // a character itself is a palindrome.

Longest Palindrome Subsequence

Time Complexity: O($\mathbf{N}^2$)

Try to code it as a exercise.

You may find Top-Down approach is more suitable, because it is hard to find the order to compute.

Longest Palindrome Subsequence

(ps: I’d said the problem is similar Longest Common Subsequence, well actually this problem can be solve be find the Longest Common Subsequence of S and it reverse. Try to find out why again!)

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, the transitional formula and the base cases.

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