Dynamic Programming (Ib)

Kelvin Chow {Lrt1088}

2021-05-15

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

Table of Contents

Recap of DP(Ia)

- Introduction to DP
- Some classical DP problems

Today we continues to discuss more DP problems.

Some classical DP problems

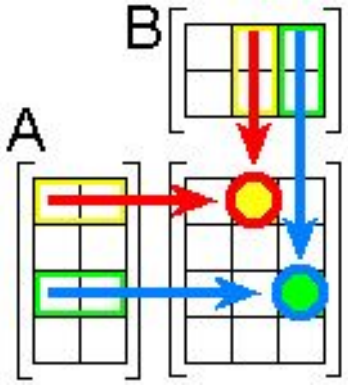
Matrix Multiplication

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

A $\mathbf{m \times n}$ 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** $\mathbf{A_2}$

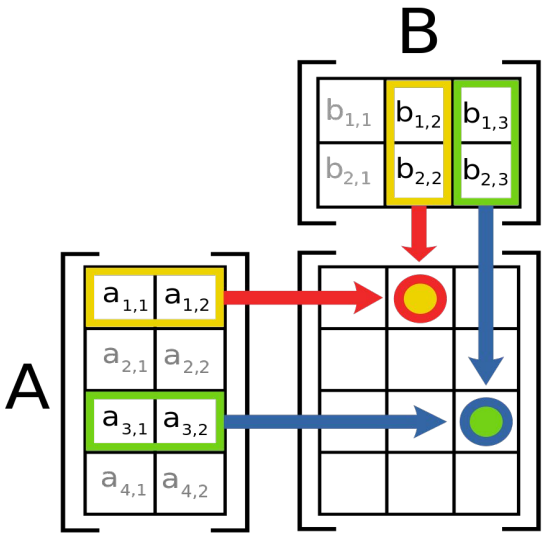
Let $\mathbf{A_1}$ is an $\mathbf{m \times p}$ matrix and $\mathbf{A_2}$ is an $\mathbf{p \times n}$ matrix, by some operation, $\mathbf{A_1A_2}$ will be a $\mathbf{m \times n}$ matrix.



(from Wikipedia)

Matrix Multiplication

(from Wikipedia)



Matrix Multiplication

Properties of matrix multiplication:

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

Matrix Multiplication

HKOJ 01054 Matrix-chain Multiplication

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

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

Find the minimum cost to finish this matrix multiplication.

Remind that Matrix multiplication is **associative.**

Matrix Multiplication

Sample test case: 1

Input	Output
3 5 10 15 5	1000

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

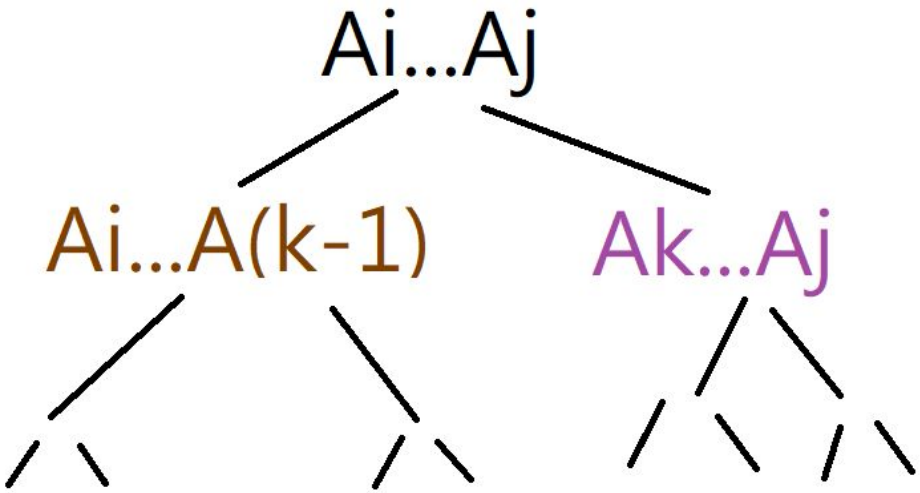
	Matrices			Total cost
1.	5x10	10x15	15x5	0
2.	5x10	10x5		750
3.	5x5			1000

Matrix Multiplication

Brute Force:

- Let's think about it, if we to perform the Multiplication of a chain $A_i...A_j$, where **i < j**, the **final** operation must be the Multiplication of the **resultant** matrix of chain $A_i...A_{k-1}$ and the **resultant** matrix of chain $A_k...A_j$, 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 $A_i...A_j$, $A_i...A_{k-1}$ and $A_k...A_j$ can obtain in a same process as above.
- We can solve it recursively!

Matrix Multiplication



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 $\mathbf{A_i...A_j}$.

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.

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

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 $O(N^2)$ state
- 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

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	-	-	-	-	-	-	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	-	-	-	-	-	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	-	-	-	-	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	-	-	-	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	-	-	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	3	-	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	3	3	-

Longest Increasing Subsequence

Let's dry run together!

```
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);
	}
}
```

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	3	3	4

Longest Increasing Subsequence

Time Complexity: $O(N^2)$

Is not hard right?

CONSTRAINTS

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

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

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

a[8] = 7 can append to the back of i = 1, 2, 3, 5, 7

We only need to know there is a LIS ended with a[i] that its length = dp[i].

We don't need to how to construct this LIS.

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	3	3	4

Longest Increasing Subsequence

For all dp[i] = 3, i.e. 4, 6, 7, to maximize the chance of successful append, we should only consider the one with the minimum a[i].

In this case a[7] = 6.

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	3	3	4

Longest Increasing Subsequence

For each value of dp[i], if consider the minimum value of a[i] only and store those values, we can see it's increasing.

	1	2	3	4	5	6	7	8
a[ ]	3	4	2	9	1	9	6	7
dp[ ]	1	2	1	3	1	3	3	4

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

f[i] 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 **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(lgN)** every query since g[] is **increasing**.

Total Time Complexity: O(N*lgN)

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. $\mathbf{A_i = A_{N-i+1}}$ 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!)

Some DP problems on HKOJ

Diamond Chain II

M0822 Diamond Chain II

- Similar to Maximum Subarray Sum on DP(Ia)
- Expect we need to form two disjoint sub-arrays.

Input	Output
12 -8 5 10 -2 6 3 -7 -2 3 5 -10 2	30

Diamond Chain II

Let's divide our "sub-arrays forming process" into different stages.

1. Did not start forming the first sub-array
2. Forming the first sub-array
3. After finishing forming the first one but not start to form the second one
4. Forming the second sub-array
5. Finished

Input	Output
12 -8 5 10 -2 6 3 -7 -2 3 5 -10 2	30

Diamond Chain II

Generally, for each element **a[i]**, it can be on those stages when forming the sub-arrays. Let's rephrase the stages.

1. Not adding **a[i]** to the first sub-array
2. Add **a[i]** to the first sub-array
3. Not adding **a[i]** to the first sub-array and the second sub-array
4. Add **a[i]** to the second sub-array
5. Not adding **a[i]** to the second sub-array

We should be able to find the optimal value for each i and stage.

Diamond Chain II

Let dp[i][j] = the optimal value when we arrive **i** in stage **j**.

Let's discuss how those stages can be transit from others.

Diamond Chain II

1. Not adding **a[i]** to the first sub-array

We should simply skip a[i], so dp[i][1] = dp[i-1][1].

Diamond Chain II

2. Add **a[i]** to the first sub-array

There two possibility, the first sub-array start from **a[i]** or the first sub-array start before **a[i]**. For the later one, we can append **a[i]** on the back of **a[i-1]**

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

Diamond Chain II

3. Not adding **a[i]** to the first sub-array and the second sub-array

The possibility are:

1. The first sub-array is empty
2. The first sub-array is end with **a[i-1]**
3. The first sub-array is end before **a[i-1]**

dp[i][3] = max(dp[i-1][1], dp[i-1][2], dp[i-1][3])

Diamond Chain II

4. Add **a[i]** to the second sub-array

The possibility are:

1. The first sub-array is empty, and the second start with **a[i]**
2. The first sub-array is end with **a[i-1]**, and the second start with **a[i]**
3. The first sub-array is end before **a[i-1]**, and the second start with **a[i]**
4. The second sub-array start before **a[i]**

dp[i][4] = max(dp[i-1][1]+a[i], dp[i-1][2]+a[i], dp[i-1][3]+a[i], dp[i-1][4]+a[i])

Diamond Chain II

5. Not adding **a[i]** to the second sub-array

Actually, we don't need to consider this stage because for each dp[i][j], it already skip every elements afterward.

ans = max(dp[i][j])

Time complexity: O(N*4) = O(N)

Diamond Chain II

When you facing other problems, you may consider separate it into different cases as different states like this, and then think how to transit it from each other.

Subsequence Product

M1712 Subsequence Product

- Subsequence seems to be the friend of DP.
- Find the total number of Subsequences that their product = **M**

Input	Output
4 6 1 2 2 3	4

Subsequence Product

Let dp[i][j] = the number of subsequences of 1...i that their product = j.

(very rough work below)

For each dp[i][j]:

```
dp[i][j] = 0;
if(a[i] == j) dp[i][j]++; //a[i] itself as a subsequences
dp[i][j] += dp[i-1][j]; //skip a[i]
if(j % a[i] == 0) dp[i][j] += dp[i-1][j / a[i]]; //add a[i]
```

Subsequence Product

a[]	1	2	2	3
j\i	1	2	3	4
1	1	1	1	1
2	0	2	4	4
3	0	0	0	1
4	0	0	1	1
5	0	0	0	0
6	0	0	0	4

Subsequence Product

Time complexity: O(NM)

For all cases: $1 \leq N \leq 500, 1 \leq M, x_i \leq 10^{12}$

Well, let's improve it.

Subsequence Product

M is too big, so let's focus on M.

Let there two positive number **a** and **b** such that **a*b = M**.

What's the property of **a, b**?

They are the factors of M.

If **a and/or b** aren't the factors of **M**, there is **no** chances that **a*b = M**.

That's mean we only need to consider dp[i][j] where **j** is the factors of **M**.

Subsequence Product

To find the factors of M, we can use following algorithm:

```
vector<long long> f;
for(long long i = 1; i*i <= M; i++) {
     if(M % i == 0) {
          f.push_back(i);
          if(i*i != M) {
               f.push_back(M / i);
          }
     }
}
sort(f.begin(), f.end());
```

Subsequence Product

Let dp[i][j] =

the number of subsequences of 1...i that their product = the j^{th} factors of M.

(very rough work below)

For each dp[i][j]:

```
dp[i][j] = 0;
if(a[i] == f[j]) dp[i][j]++; //a[i] itself as a subsequences
dp[i][j] += dp[i-1][j]; //skip a[i]
if(f[j] % a[i] == 0) dp[i][j] += dp[i-1][find_pos(f[j]/a[i])]; //add a[i]
```

Subsequence Product

	a[]	1	2	2	3
f[]	j\i	1	2	3	4
1	0	1	1	1	1
2	1	0	2	4	4
3	2	0	0	0	1
6	3	0	0	0	4

Subsequence Product

To implement `find_pos( )`, we can binary search on f[].

Time complexity: $O(N*|f|\log|f| + M^{0.5})$

The number of factors should be much less than $M^{0.5}$. (I'm very poor in Math)

Subsequence Product

When you discover the constraint of your DP formula is too large, you may consider use similar method to reduce, or you may use std::map.

Coins

05022 Coins

Given some coins that are strictly in the set of {0.1, 0.2, 0.5, 1, 2, 5, 10}, find the largest subset that the sum is exactly equal to **v**.

Note sum of the coins {0.1, 0.2, 0.5} in the subset cannot be larger than 2.

Coins

First of all, floating point number cause trouble so let's multiply everything by 10. i.e. {1, 2, 5, 10, 20, 50, 100}

Since we to find the sum of coins' value with the restriction for {1, 2, 5}, we need to care of them.

Coins

Let dp[i][j][k] = the largest subset of 1...i that the sum of the values of all coins is j and the sum of the values of {1, 2, 5} is k.

(very rough work below)

For each dp[i][j][k]:

```
dp[i][j][k] = dp[i-1][j][k]; //skip c[i]
if(c[i] <= 5)
    dp[i][j][k] = max(dp[i][j][k], 1+dp[i-1][j-c[i]][k-c[i]]);
else
    dp[i][j][k] = max(dp[i][j][k], 1+dp[i-1][j-c[i]][k]);
```

Coins

Time complexity: O(|c|*(v*10)*(2*10))

This problem require backtracking so you may try it.

If you encounter problem some restrictions like this problem, you may try to add more dimension if it is fast enough.

Round-up

In this session, we had discussed about another problems. I hope that you can understand how we define the states and the transitional formals. Which are the main elements of DP.

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