Dynamic Programming

Knapsack

You are a thief who has a knapsack.You are robbing the bank.
there are some things:
weight: 2kg value: 3 million dollars
weight: 3kg value: 4 million dollars
weight: 4kg value: 5 million dollars
weight: 5kg value: 6 million dollars
But your knapsack can only hold 5 kg ! ! !
So how should you make a choice to maximize the total value?

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int kanpsack_main(int order,int
most_weight_now,vector<pair<int,int>>things,vector<vector<int>>&now)
{
    if(order==things.size()||most_weight_now<=0)
    {
        return 0;
    }
    if(now[order][most_weight_now]!=-1)
    {
        return now[order][most_weight_now];
    }
    if(things[order].first>most_weight_now)
    {
        return kanpsack_main(order + 1, most_weight_now, things, now);
    }
    else
    {
        int choose_this = things[order].second + kanpsack_main(order + 1,
most_weight_now - things[order].first, things, now);
        int not_choose_this = kanpsack_main(order + 1, most_weight_now,
```

```
things, now);
        now[order][most_weight_now] = max(choose_this, not_choose_this);
        return now[order][most_weight_now];
    }
}

int knapsack(vector<pair<int,int>>things,int most_weight)//pair<weight,value>
{
    int size_of_things = things.size();
    vector<vector<int>> now(size_of_things, vector<int>(most_weight+1,-1));
    return kanpsack_main(0,most_weight,things,now);
}

int main() {
    vector<pair<int, int>> things = {{2, 3}, {3, 4}, {4, 5}, {5, 6}};
    int most_weight = 5;
    int max_value = knapsack(things, most_weight);
    cout << "The maximum value that can be obtained is: " << max_value <<
endl;
    return 0;
}
```

longest_subset

We now have a set {1,2,3,4,5,1}.Please choose the longest_incremental_subset

```
#include<iostream>
#include<vector>
using namespace std;

int DP(int last,int
now,vector<int>original_set,vector<vector<int>>&length_of_DP)
{
    int length = original_set.size();
    if(now==length)
    {
        return 0;
    }
    else if(original_set[now]<original_set[last])
    {
```

```
        return DP(last, now + 1, original_set,length_of_DP);
    }
    else
    {
        if(length_of_DP[last][now]!=INT_MAX)
        {
            return length_of_DP[last][now];
        }
        else
        {
            length_of_DP[last][now]= max(DP(last, now + 1, original_set,
length_of_DP), 1+DP(now, now + 1, original_set, length_of_DP));
            return length_of_DP[last][now];
        }
    }
}

int longset_subset(vector<int>original_set)
{
    original_set.insert(original_set.begin(), INT_MIN);
    vector<vector<int>> length_of_DP(original_set.size(),vector<int>
(original_set.size(),INT_MAX));

    return DP(0, 1, original_set,length_of_DP);
}

int main()
{
    vector<int> original_set = {1, 2, 3, 4, 5,1};
    int result = longset_subset(original_set);

    // 输出结果
    cout << "Result: " << result << endl;

    return 0;
}
```

shortLength_A_to_B

graph = {
{{1, 3}, {2, 1}},
{{3, 7}},
{{3, 11}},
{}
};
What is the shortest way from A to B?

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

vector<vector<pair<int,int>>> reverseGraph(vector<vector<pair<int,int>>>graph)
{
    vector<vector<pair<int, int>>> rev_graph;
    rev_graph.resize(graph.size());
    for (int i = 0; i < graph.size();i++)
    {
        for(pair<int,int>edge:graph[i])
        {
            int end=edge.first;
            int weight=edge.second;
            rev_graph[end].push_back({i, weight});
        }
    }
    return rev_graph;
}

int shortLength_A_to_B_main(vector<vector<pair<int,int>>>regraph,int
start_point,int end_point,vector<int>&shortestLength,vector<int>&choices)
{
    if(start_point==end_point)
    {
        return 0;
    }
    else if (regraph[end_point].empty())
    {
        return INT_MAX;
    }
```

```
    else if(shortestLength[end_point]!=-1)
    {
        return shortestLength[end_point];
    }
    else
    {
        int min_length = INT_MAX;
        for(pair<int,int>edge:regraph[end_point])
        {
            int prepoint = edge.first;
            int weight=edge.second;
            int length_start_to_prepoint = shortLength_A_to_B_main(regraph,
start_point, prepoint, shortestLength,choices);
            if(length_start_to_prepoint!=INT_MAX)
            {
                if(length_start_to_prepoint+weight<min_length)
                {
                    min_length = length_start_to_prepoint + weight;
                    choices[end_point] = prepoint;
                }
            }
        }
        shortestLength[end_point] = min_length;
        return min_length;
    }
}

int shortestLength_from_A_to_B(vector<vector<pair<int,int>>>graph,int
start_point,int end_point)
{
    vector<int> shortestLength(graph.size(), -1);
    shortestLength[start_point] = 0;
    vector<int> choices(graph.size(), -1);
    int distance = shortLength_A_to_B_main(reverseGraph(graph), start_point,
end_point, shortestLength,choices);

    if(distance==INT_MAX)
    {
        cout << "there is no way from start_point to end_point." << endl;
    }
```

```
    else
    {
        vector<int> points_in_the_way;
        int at = end_point;
        while(at!=start_point)
        {
            points_in_the_way.push_back(at);
            at = choices[at];
        }
        points_in_the_way.push_back(start_point);
        reverse(points_in_the_way.begin(), points_in_the_way.end());

        cout << "The shortest path from " << start_point << " to " <<
end_point << " is: ";
        for (int point : points_in_the_way) {
            cout << point << " ";
        }
        cout << endl;

    }

    return distance;
}

int main() {
    // 示例图的邻接表表示
    vector<vector<pair<int, int>>> graph = {
        {{1, 3}, {2, 1}},
        {{3, 7}},
        {{3, 11}},
        {}
    };

    int start_point = 0; // 起始点
    int end_point = 3;   // 终点

    int shortest_length = shortestLength_from_A_to_B(graph, start_point,
end_point);
    if (shortest_length == INT_MAX) {
        cout << "There is no path from " << start_point << " to " << end_point
```

```
<< endl;
    } else {
        cout << "The shortest path from " << start_point << " to " <<
end_point << " is " << shortest_length << endl;
    }

    return 0;
}
```

Edit_Distance

We have a “cat" and a "dog".
We can add a letter to "cat" or delete a letter from "cat" or change a letter in "cat".
What is the shortest "distance" from "cat" to "dog"?(At least, we should use how many steps to change "cat" to "dog"?)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int edit_distance_main(string s1, string s2, vector<vector<int>>
&distance_now)
{
    if (s1.empty() && s2.empty())
    {
        return 0;
    }
    else if (s1.empty() && !s2.empty())
    {
        return s2.size();
    }
    else if (!s1.empty() && s2.empty())
    {
        return s1.size();
    }
    else if (distance_now[s1.size()][s2.size()] != INT_MAX)
    {
        return distance_now[s1.size()][s2.size()];
```

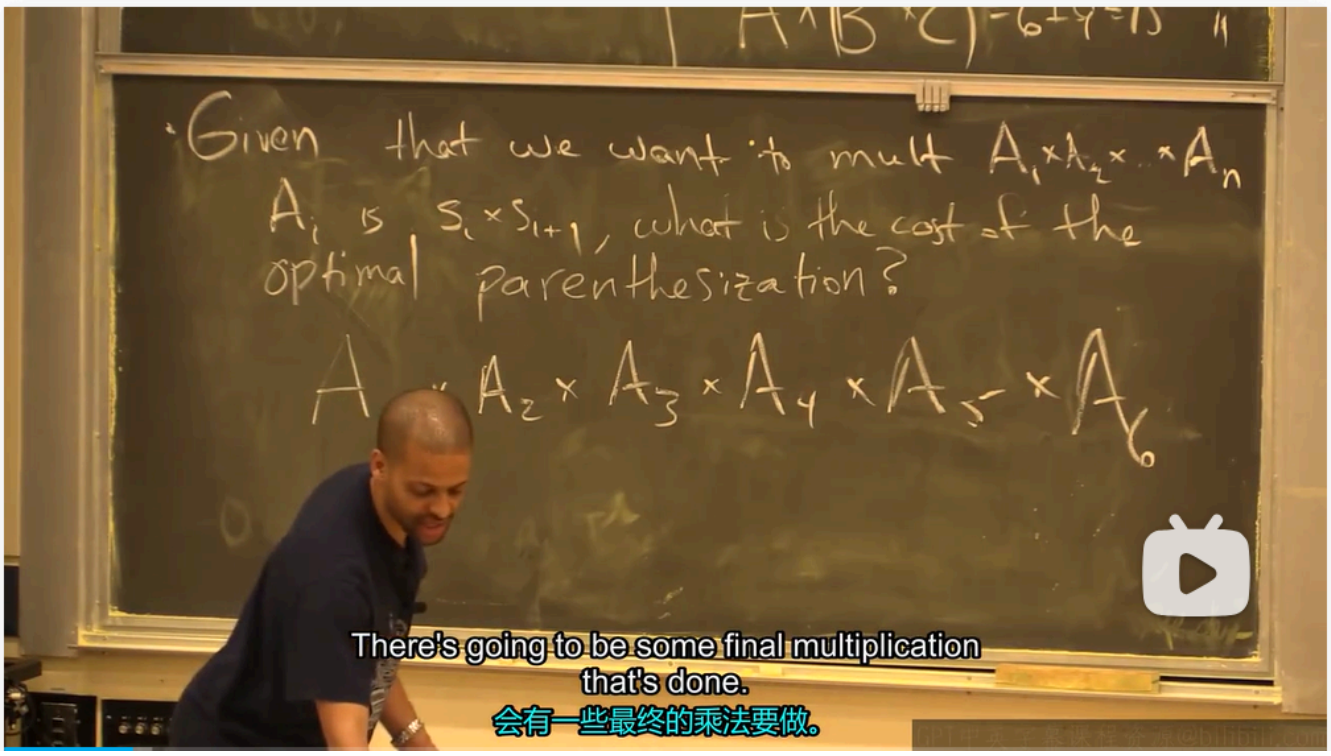
```
    }
    else
    {
        if (s1[0] == s2[0])
        {
            distance_now[s1.size()][s2.size()] =
edit_distance_main(s1.substr(1), s2.substr(1), distance_now);
            return distance_now[s1.size()][s2.size()];
        }
        else
        {
            // add a letter to s1
            int distance_add = edit_distance_main(s1, s2.substr(1),
distance_now) + 1;
            // delete a letter fron s1
            int distance_delete = edit_distance_main(s1.substr(1), s2,
distance_now) + 1;
            // replace a letter from s1
            int distance_replace = edit_distance_main(s1.substr(1),
s2.substr(1), distance_now) + 1;
            distance_now[s1.size()][s2.size()] = min(min(distance_add,
distance_delete), distance_replace);
            return distance_now[s1.size()][s2.size()];
        }
    }
}

int edit_distance(string s1, string s2)
{
    vector<vector<int>> distance_now(s1.size() + 1, vector<int>(s2.size() + 1,
INT_MAX));
    return edit_distance_main(s1, s2, distance_now);
}

int main()
{
    string s1 = "cat";
    string s2 = "dog";
    int distance = edit_distance(s1, s2);
    cout << distance << endl;
```

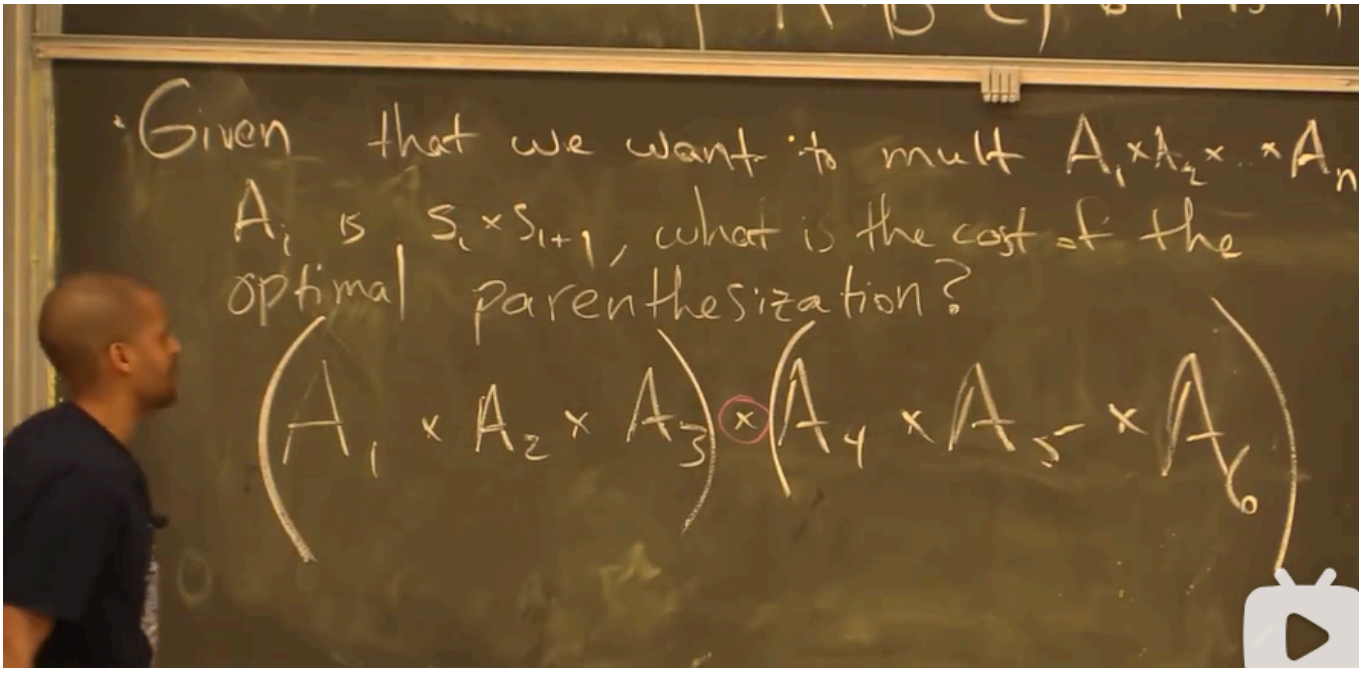
```
    return 0;
}
```

Matrix chain multiplication



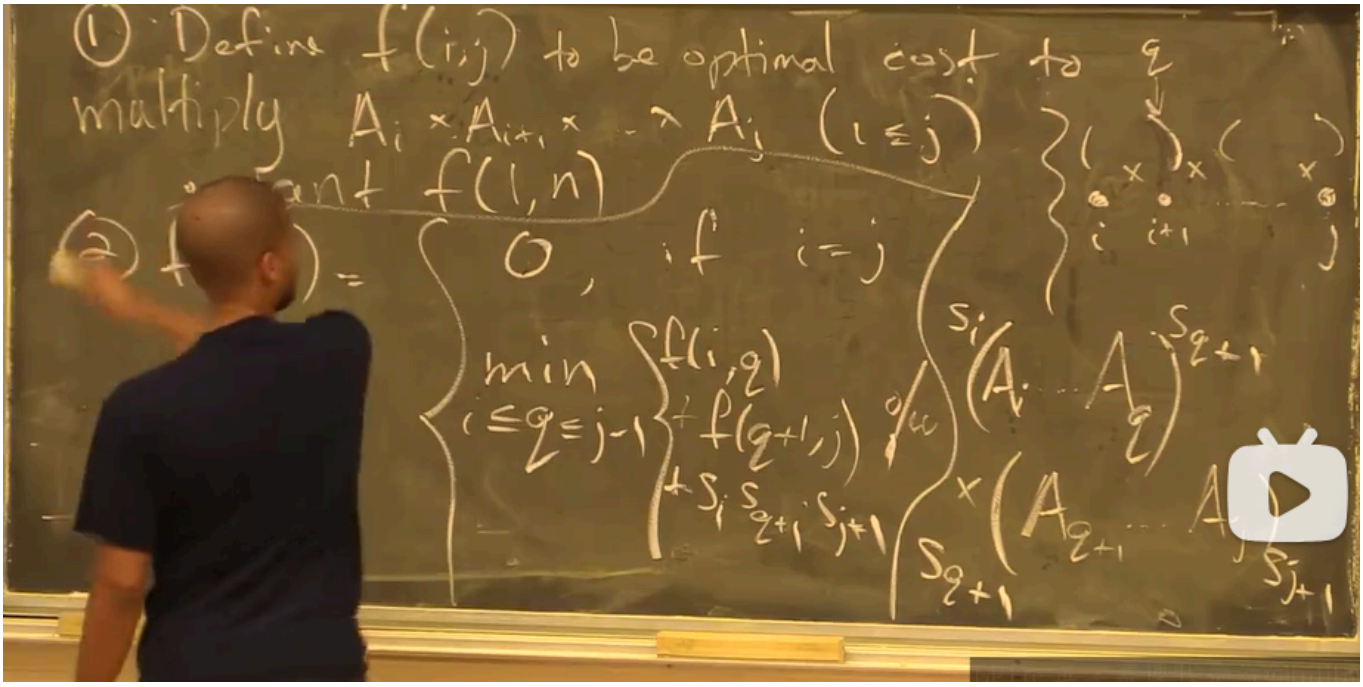
We should multiply A1(1,2),A2(2,4),A3(4,3),.....

if we multiply just two matrix A1(x,y),A2(y,z),It will be **x *y* z** times calculates.
So we could divide it into two matrixs

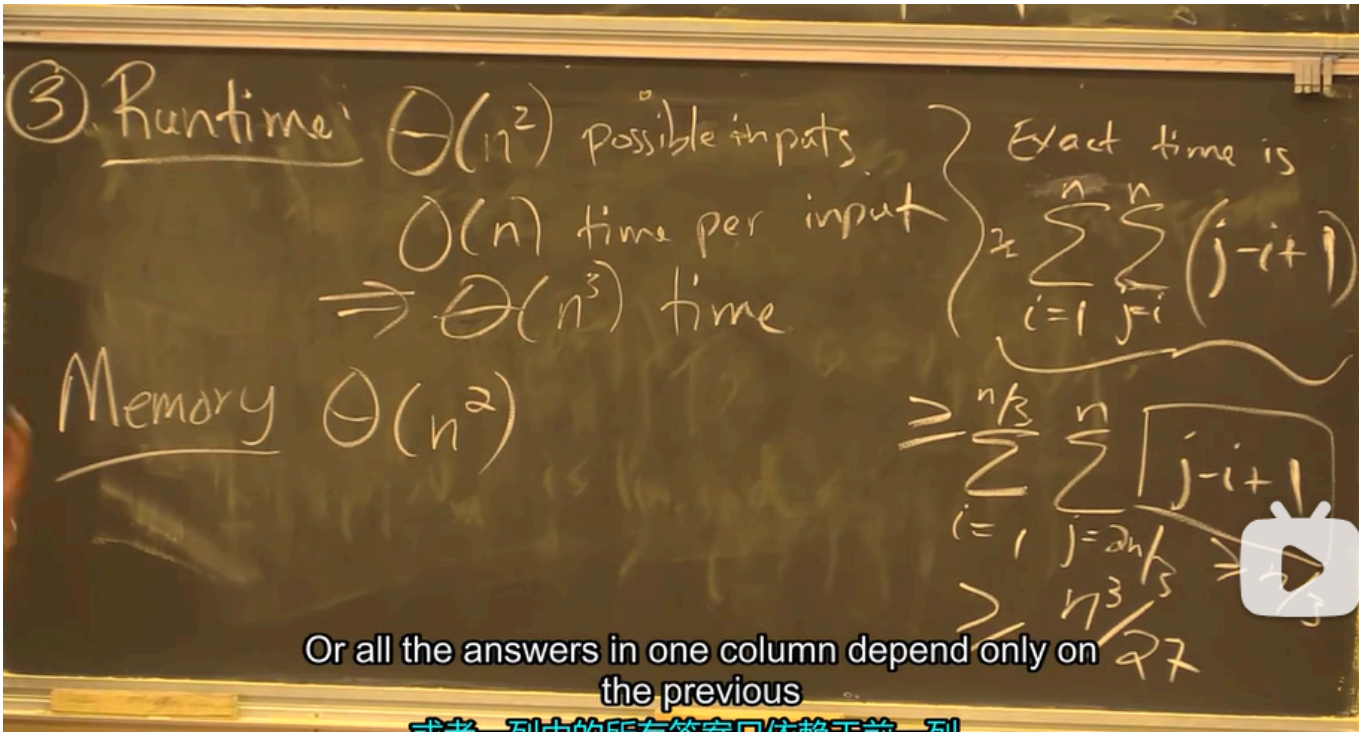


Then we difine F(x,y) to be optional cost to multiply A(x) A(x+1) .....A(y).

So let's Dynamic Programming ! ! !



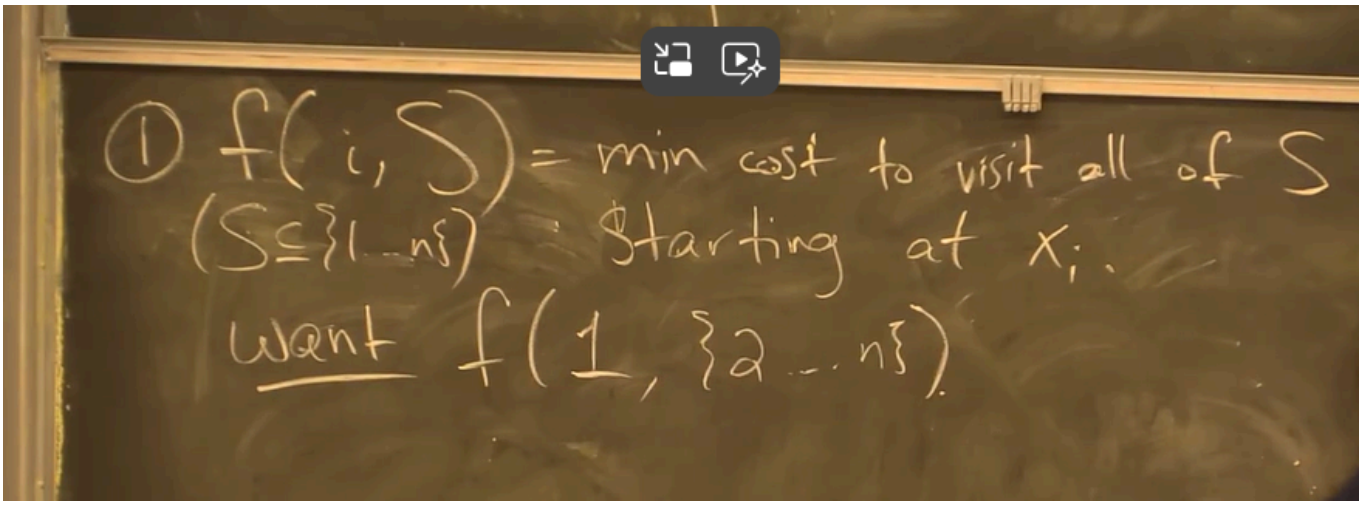
But you could see it will be a bad Runtime and Memory:

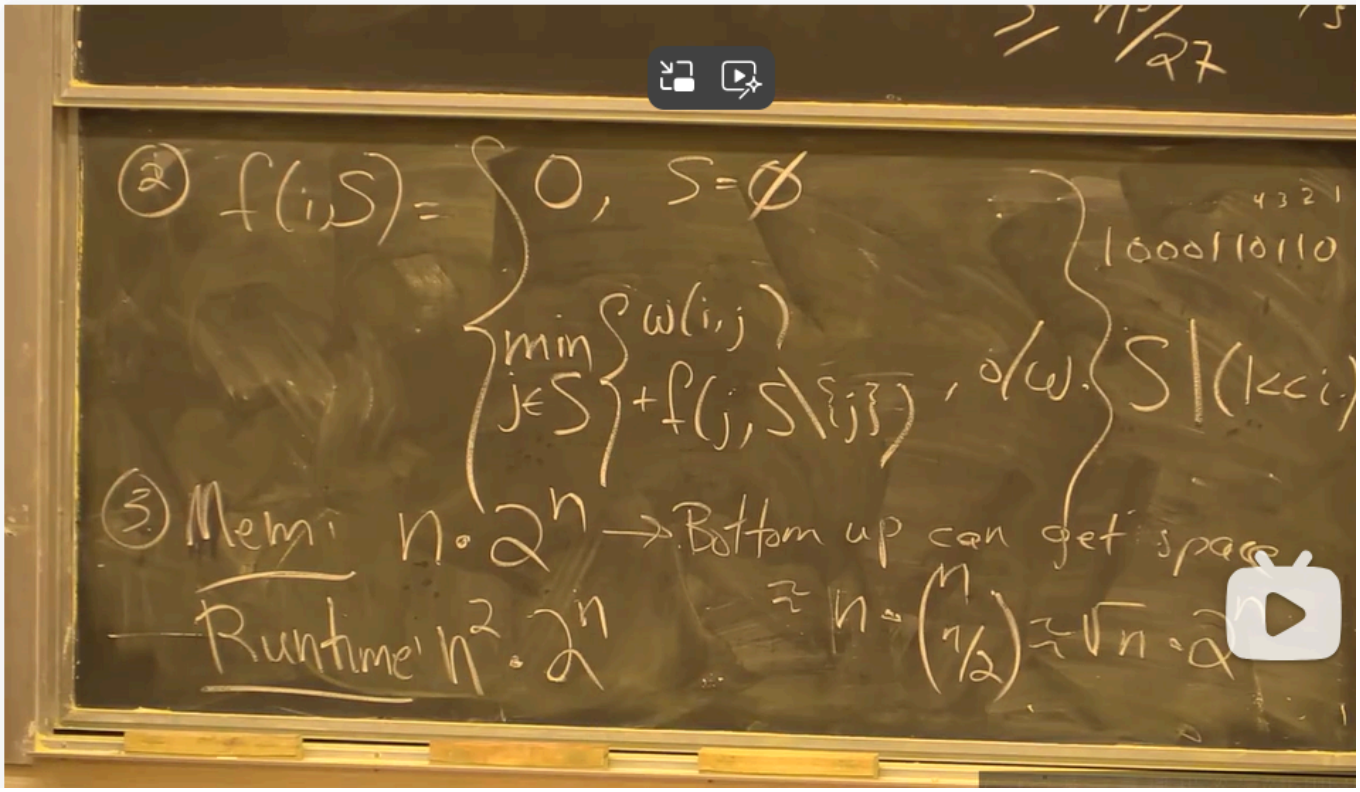


Traveling Salesman

There are n points x1,x2,.....xn,representing citys. You are a traveling salesman. You should start from x1,then travel all other citys.

Of course,you should pay money to travel between any two citys(from different city to different city cost different amount dollars).But you don't want to cost even one dollar ! So try to find a way to cost the least money.


① f(i, S) = min cost to visit all of S
(S ⊆ {1...n}) starting at x_i.
Want f(1, {2...n})


② f(i,S) = 0, S = ∅
min_{j∈S} { ω(i,j) + f(j, S\{j}) }, ω(i,j) ∈ S | (1 << i)
4 3 2 1
1000110110
③ Mem: n · 2^n → Bottom up can get space
Runtime: n^2 · 2^n ≈ n · (n choose n/2) ≈ √n · 2^n