动态规划入门与基础

目录:

1.动态规划是一种思想!

2.按时间或空间划分顺序，依次求子问题（一次）！最后到达目的。

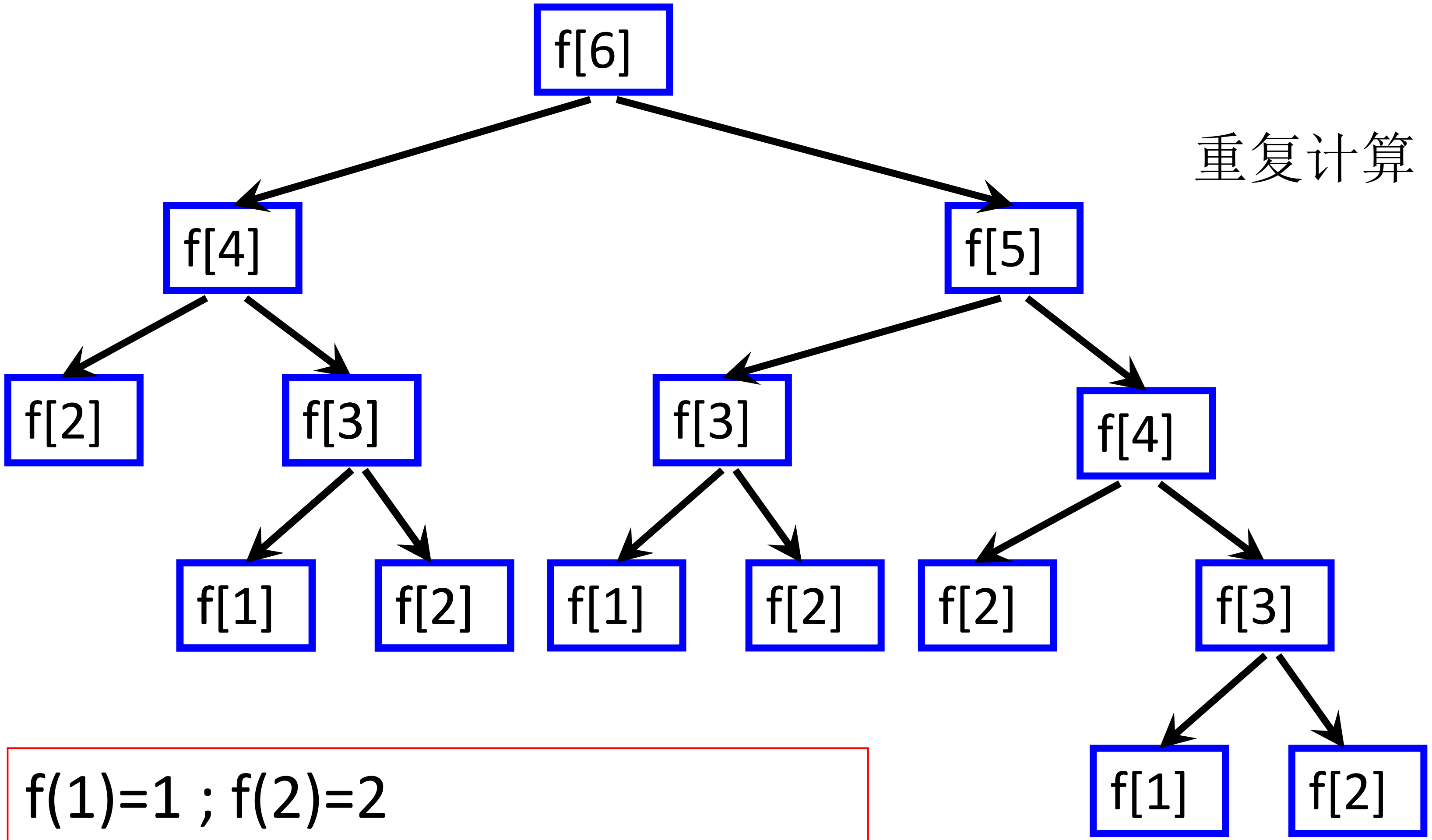
3.求解过程是一个填表（关键是如何设计表）过程。

例1:

数列 f(n)，递推关系：
f(1)=1
f(2)=2
f(n)=f(n-1)+f(n-2)
输入n，求f(n)的值。

递归实现1：

```
#include<cstdio>
#include<iostream>
using namespace std;
int n;
int dfs(int i){
    if(i==1)return 1;
    if(i==2)return 2;
    return dfs(i-2)+dfs(i-1);
}
int main(){
    cin>>n;
    cout<<dfs(n)<<endl;
    return 0;
}
```



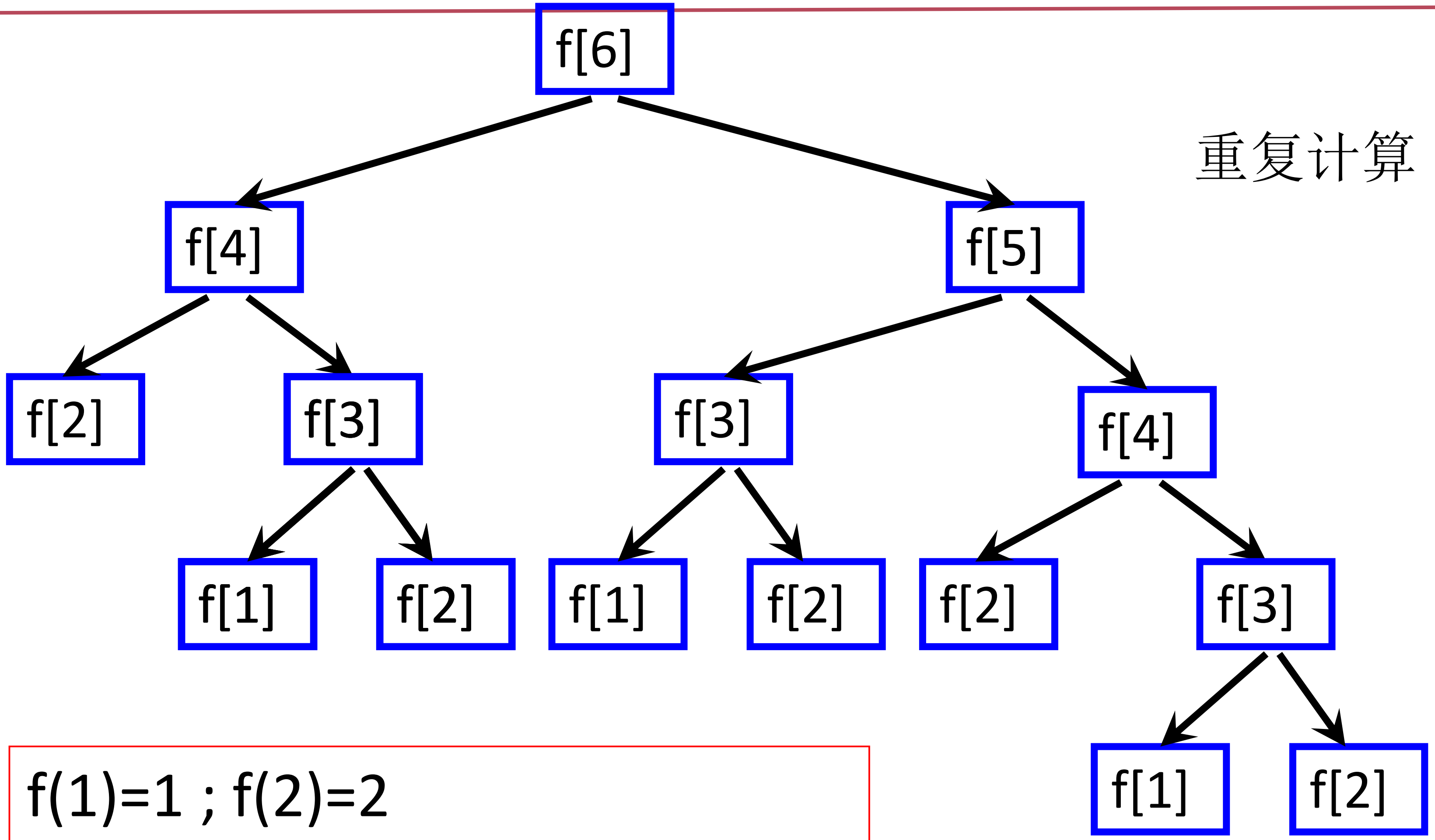
重复计算

$f(1)=1$; $f(2)=2$

$f(n)=f(n-2)+f(n-1)$

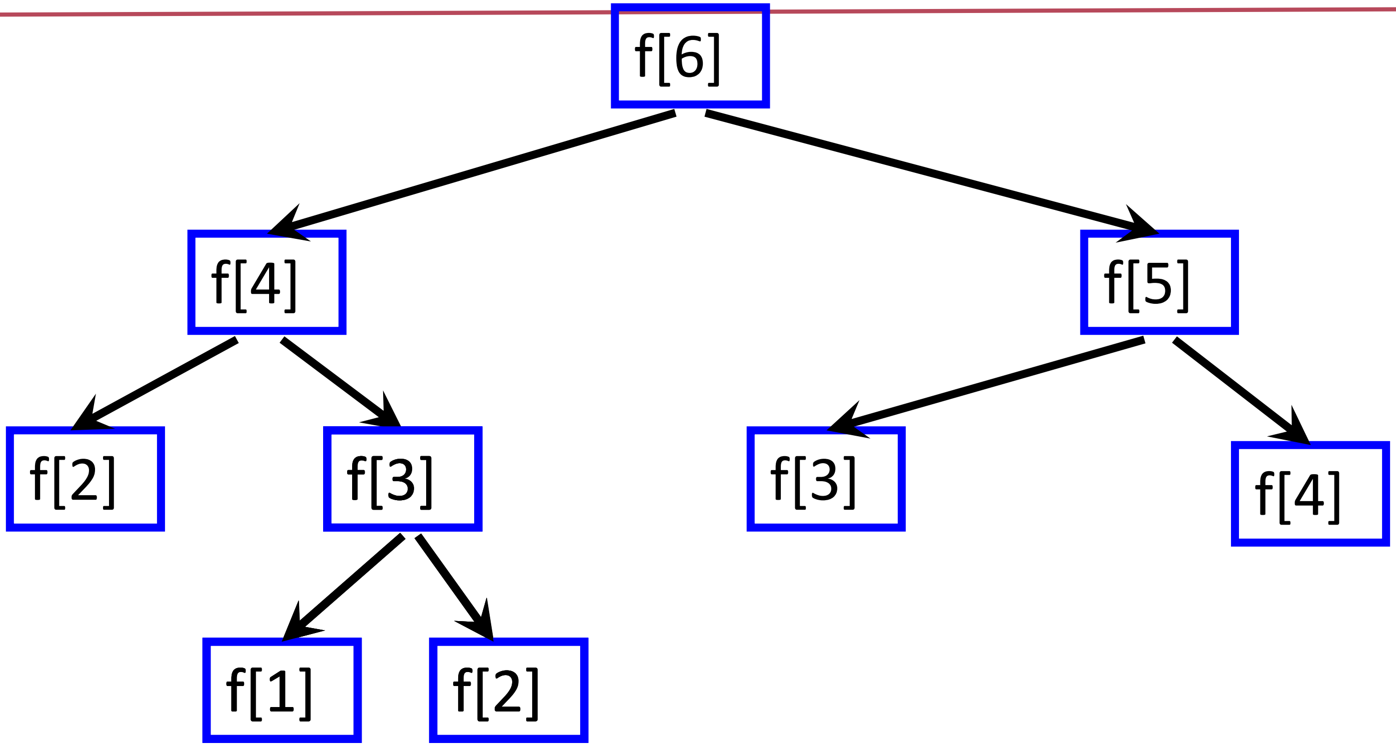
改进方法1：记忆化搜索

```
long long f[60];
int n;
long long dfs(int i){
    if(i==1)return f[1]=1;
    if(i==2)return f[2]=2;
    if(f[i]>0)return f[i];
    return f[i]=dfs(i-2)+dfs(i-1);
}
int main(){
    cin>>n;
    cout<<dfs(n)<<endl;
    return 0;
}
```



重复计算

f(1)=1 ; f(2)=2
f(n)=f(n-2)+f(n-1)



去掉重复的计算

f(1)=1 ; f(2)=2
f(n)=f(n-2)+f(n-1)

改进方法2：递推

```
long long f[60];
int n;
int main(){
	cin>>n;
	f[1]=1;
	f[2]=2;
	for(int i=3;i<=n;i++)
		f[i]=f[i-2]+f[i-1];
	cout<<f[n]<<endl;
	return 0;
}
```

两种方法效率提高的原因：

1.重复的任务只是第一次遇到时计算一次，记下来，后面再遇到直接查表使用，不再做重复的计算；

2.重复的任务，每次遇到都是一样的，不会改变；

3.计算按照一定的顺序执行；

4.备忘录

5.目标是f(n),需要求依次求出子问题f(1)..f(n-1)。

1.记忆化搜索

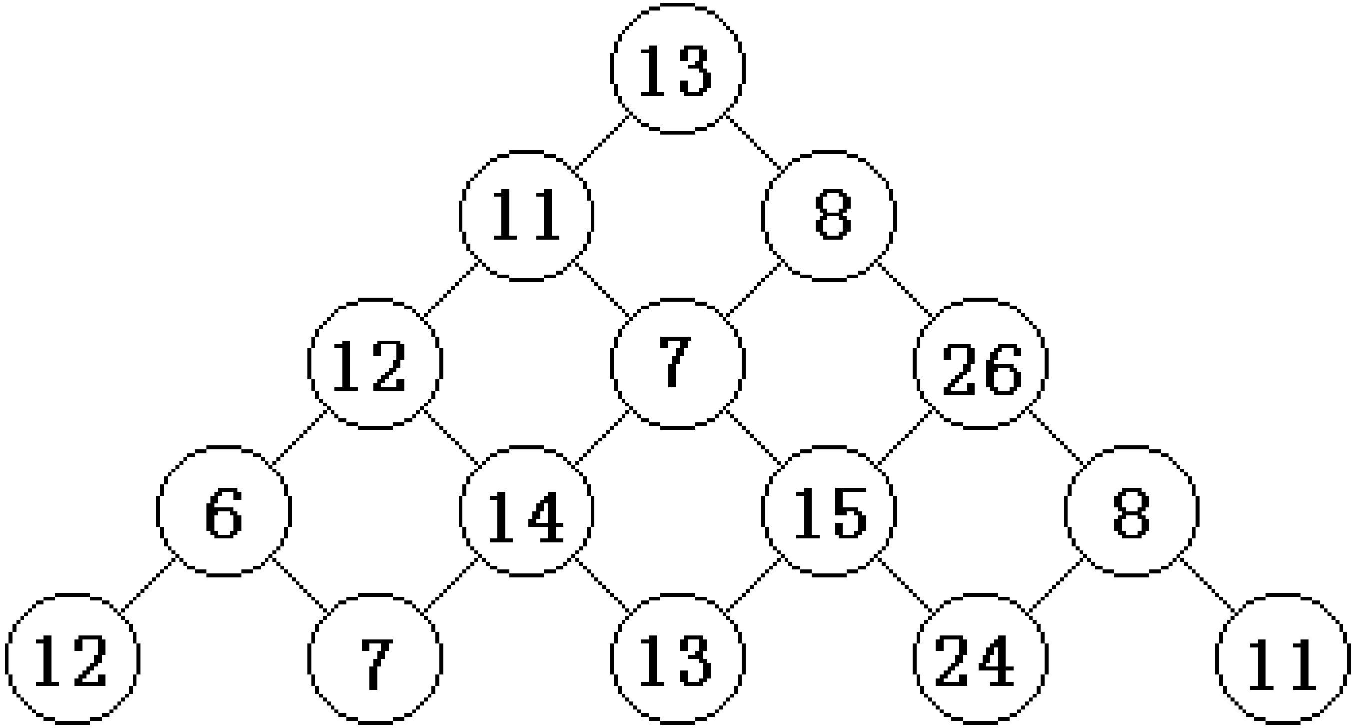
```
long long dfs(int i){
    if(i==1)return f[1]=1;
    if(i==2)return f[2]=2;
    if(f[i]>0)return f[i];//已经求过直接使用
    return f[i]=dfs(i-2)+dfs(i-1);
}
```

2.递推

```
f[1]=1;
f[2]=2;
for(int i=3;i<=n;i++)
    f[i]=f[i-2]+f[i-1];
cout<<f[n]<<endl;
```

动态规划实现的两种常用方法。

例2: 数字三角形（n<=1000）



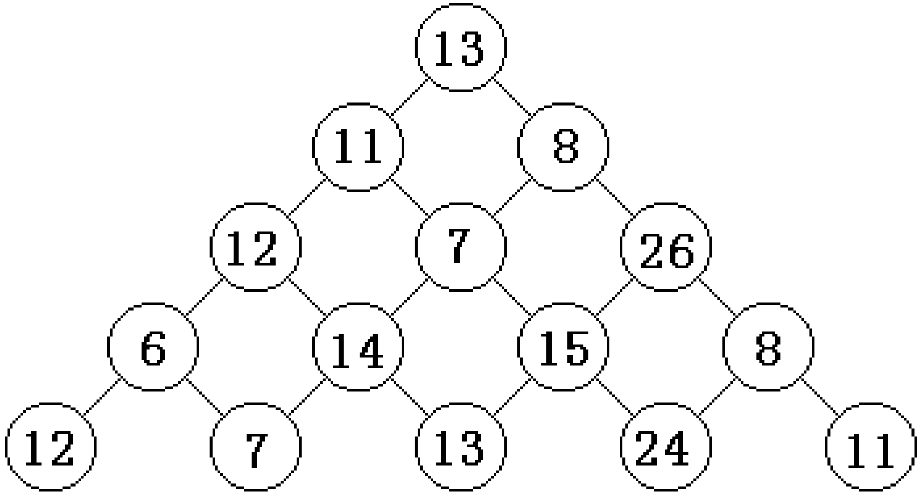
方法1：递归（爆搜）

定义：函数dfs(x,y)从(x,y)走到最后一行(n,i)得到的最大值。

d(x,y)=max(dfs(x+1,y),dfs(x+1,y+1))+a[x][y]

边界：x=n

目标：dfs(1,1);



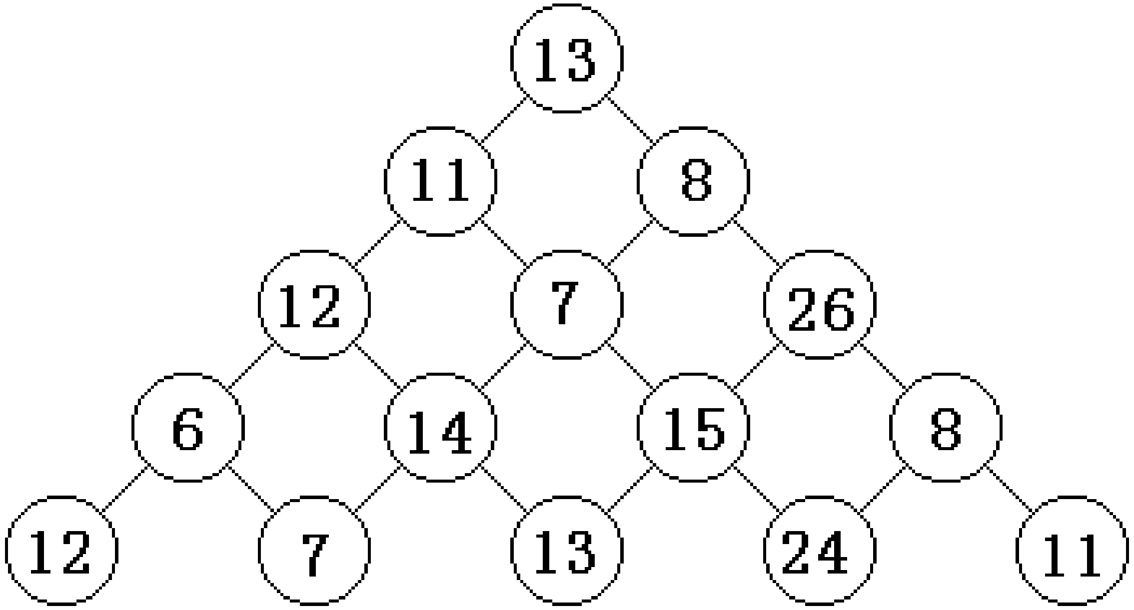

```
int dfs(int x,int y){
    //从(x,y)走到最后一行(n,i)得到的最大值
    if(x==n)return a[x][y];
    return max(dfs(x+1,y),dfs(x+1,y+1))+a[x][y];
}
```

方法2:分析太慢的原因?

重复计算了很多的dfs(x,y);

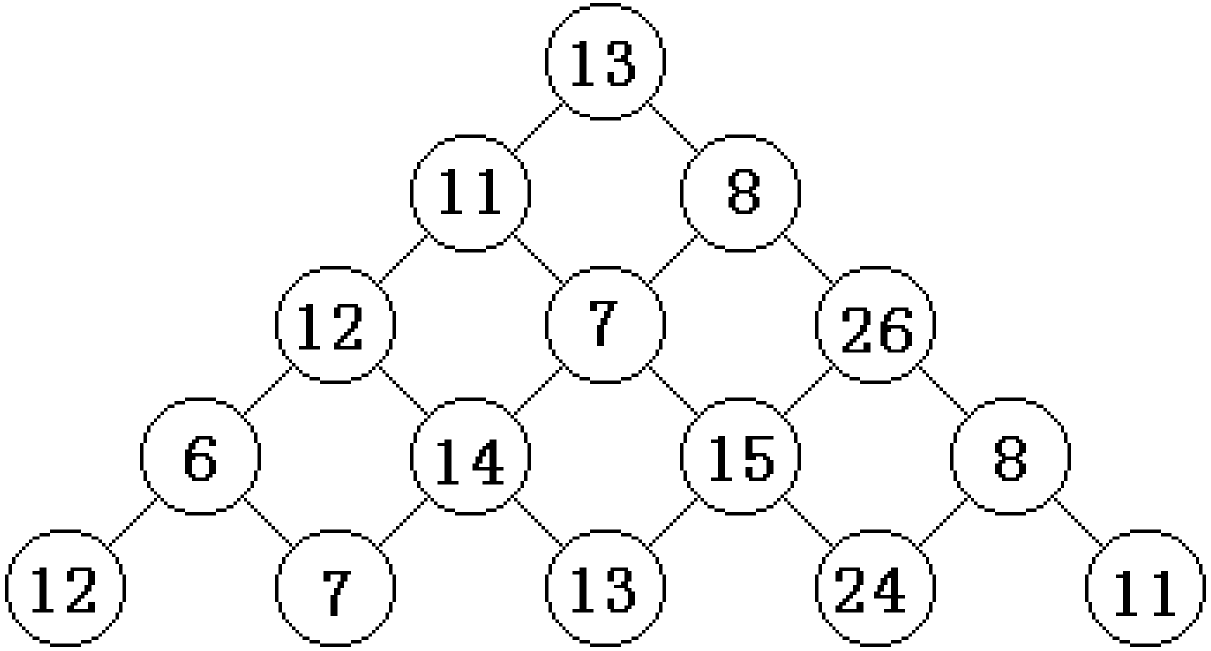
改进:

f[x][y]:记录dfs(x,y),因为(x,y)走到最后一行的最大值是唯一的，第一次走时记录下最优值，以后再用到时直接用f[x][y]即可，不需要再递归求解。



记忆化搜索

```
int dfs(int x,int y){
    if(f[x][y]>0)return f[x][y];
    if(x==n)return f[x][y]=a[x][y];
    return f[x][y]=max(dfs(x+1,y),dfs(x+1,y+1))+a[x][y];
}
```



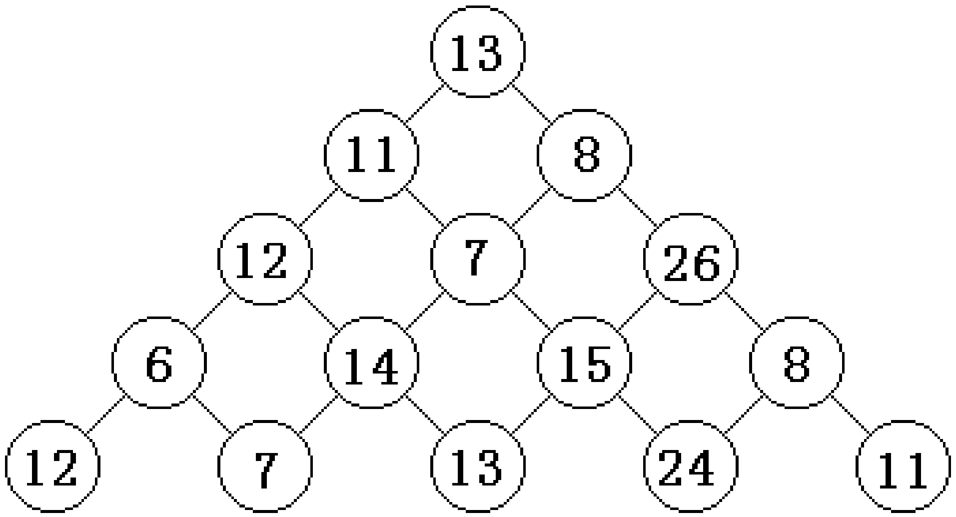
dfs(1,1):往下走，向上返回：真正计算是倒着从下向上计算的。依次计算第n,n-1,n-2,..,1行。

方法3：直接倒着从第n行开始向上推（递归的回退过程）：

f[i][j]:从(i,j)走到最后一行的最大值。

目标：f[1][1]

初始：f[n][i]=a[n][i]



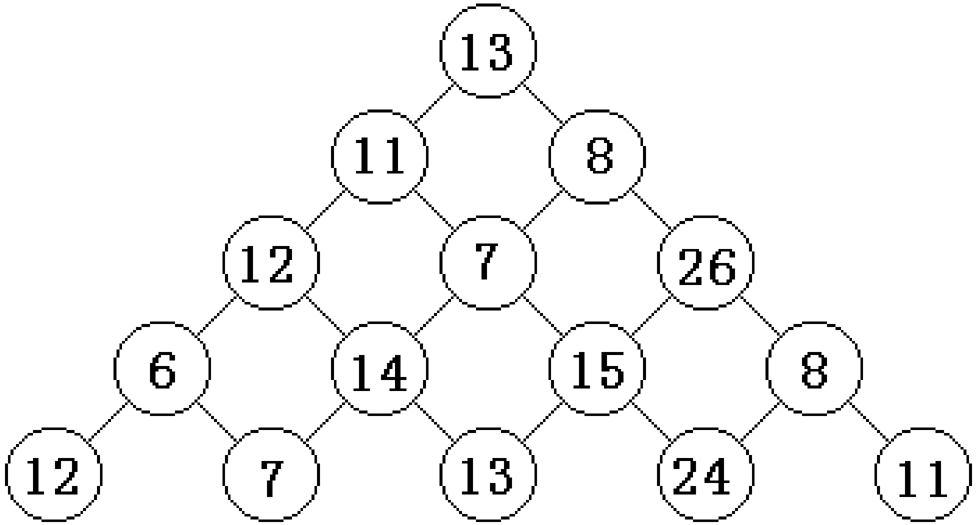

```
for(int i=1;i<=n;i++)f[n][i]=a[n][i];
for(int i=n-1;i>0;i--)
    for(int j=1;j<=i;j++)
        f[i][j]=max(f[i+1][j],f[i+1][j+1])+a[i][j];
cout<<f[1][1]<<endl;
```

能否正向求？怎么定义？

方法4：正推（从第一行到最后一行）

f[i][j]:从(1,1)走到(i,j) 的最大值。

目标：max(f[n][i])
初始：f[1][1]=a[1][1]




```
f[1][1]=a[1][1];
for(int i=2;i<=n;i++)
    for(int j=1;j<=i;j++)
        f[i][j]=max(f[i-1][j-1],f[i-1][j])+a[i][j];
int ans=0;
for(int i=1;i<=n;i++)ans=max(ans,f[n][i]);
cout<<ans<<endl;
```

有明确方向的问题：一般正向和逆向都可以求，选择一种即可，

建议：初学者尽量都写一遍以加深理解

动态规划的基本概念

动态规划（Dynamic Programming 简称DP）。

解决“多阶段决策问题”的一种高效算法。

通过合理组合子问题的解从而解决整个问题解的一种算法。其中的子问题并不是独立的，这些子问题又包含有公共的子子问题。……

动态规划算法就是对每个子问题只求一次，并将其结果保存在一张表中(数组)，以后再用到时直接从表中拿过来使用，避免重复计算相同的子问题。

“不做无用功”的求解模式，大大提高了程序的效率。

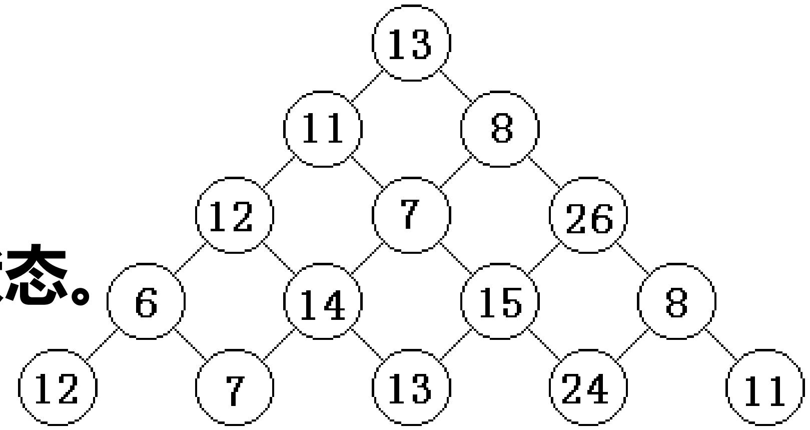
动态规划算法常用于解决统计类问题（统计方案总数）和最优值问题（最大值或最小值），尤其普遍用于最优化问题。

动态规划的术语：

1、 阶段：

把所给求解问题的过程恰当地分成若干个相互联系的阶段，以便于按一定的次序去求解，过程不同，阶段数就可能不同．描述阶段的变量称为阶段变量。在多数情况下，阶段变量是离散的，用k表示。

阶段的划分一般根据时间和空间来划分的。



2、状态：

某一阶段的出发位置成为状态，通常一个阶段有多个状态。

状态通常可以用一个或一组数来描述，称为状态变量。

3、决策：

一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择（行动）称为决策。描述决策的变量称决策变量

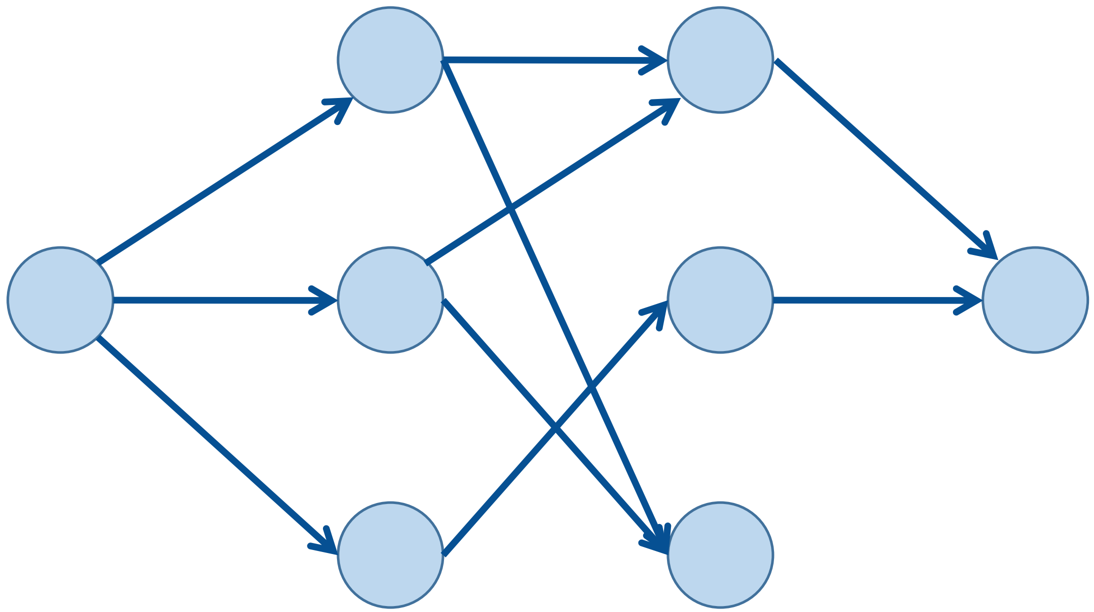
4、策略和最优策略

所有阶段的决策有序组合构成一个策略。

最优效果的策略叫最优策略。

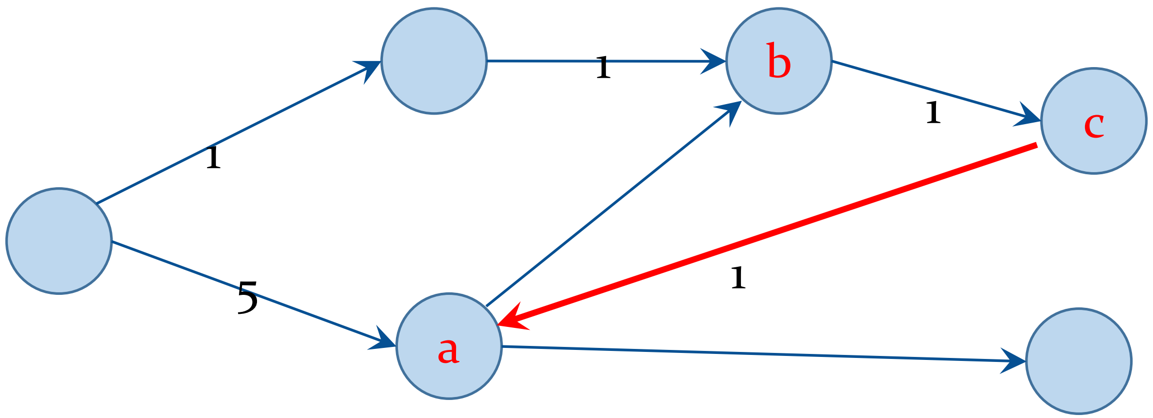
动态规划的条件：

- 拓扑图（有向无环图）

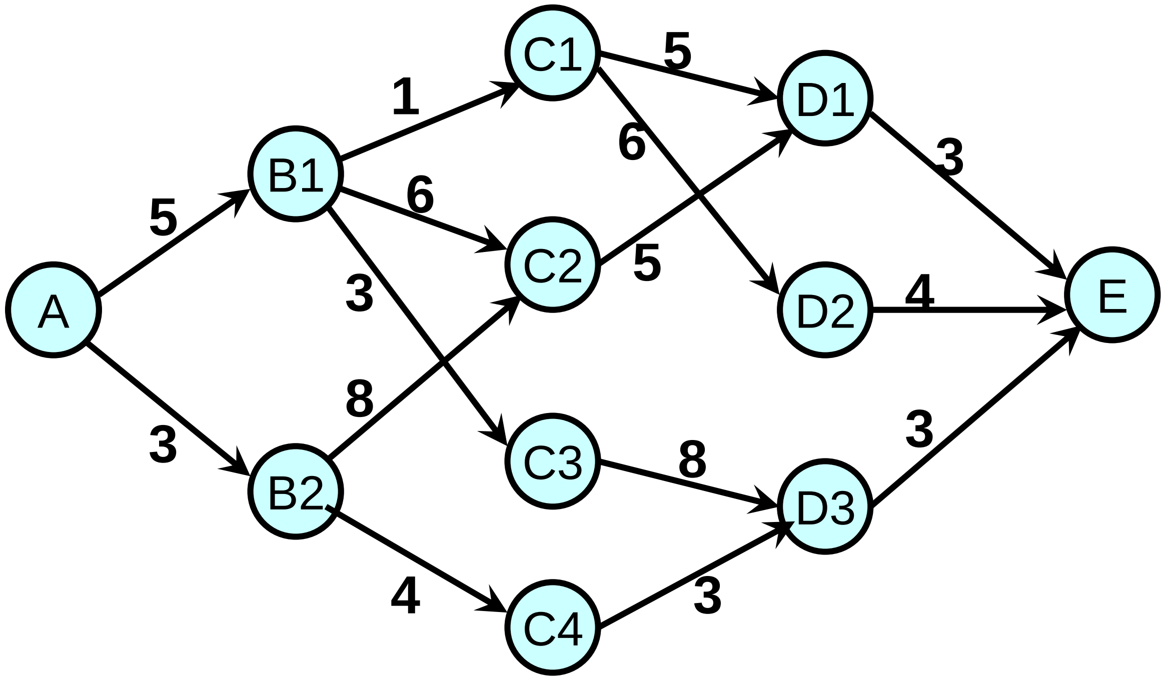


非拓扑图（可能有环）

有后效性 a→b→c？ b→c→a?



例：最短路径问题



现在，我们想从城市A到达城市E。
怎样走才能使得路径最短，最短路径的长度是多少？

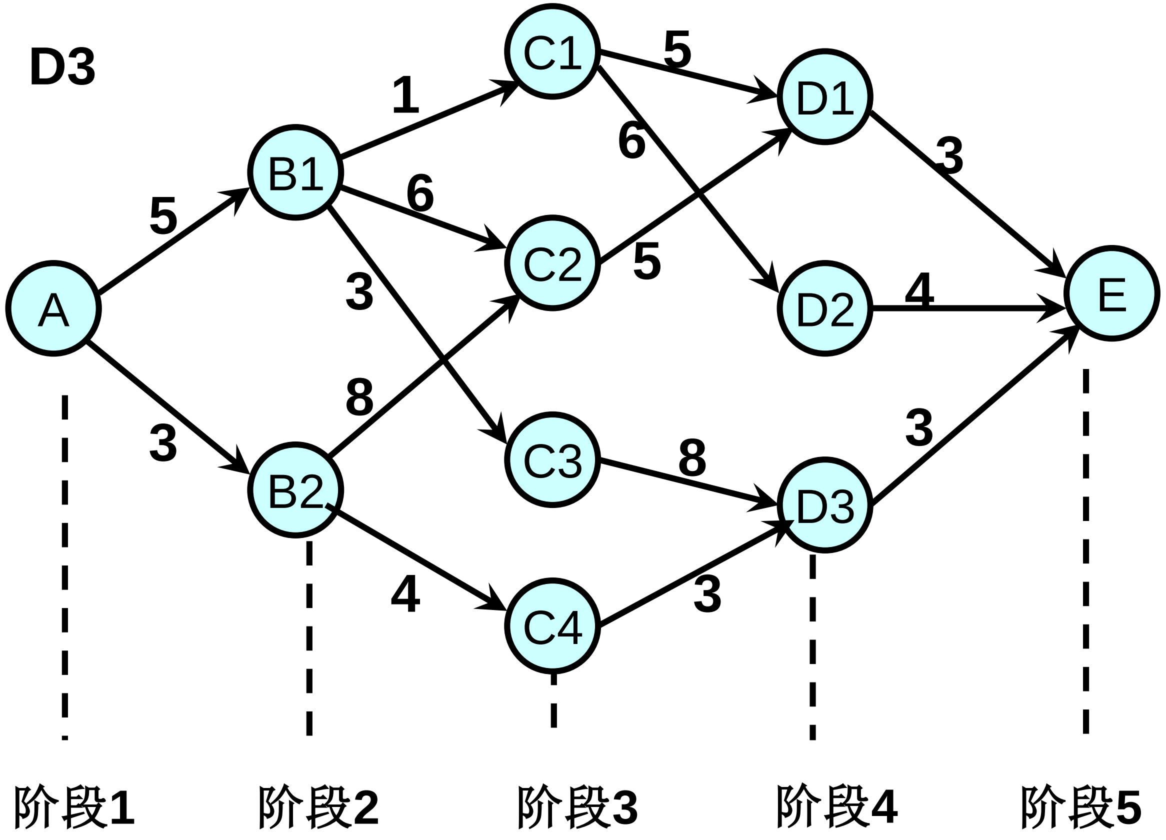
第1部分：状态 A

第2部分：状态 B1，B2

第3部分：状态 C1，C2，C3，C4

第4部分：状态 D1，D2，D3

第5部分：状态 E

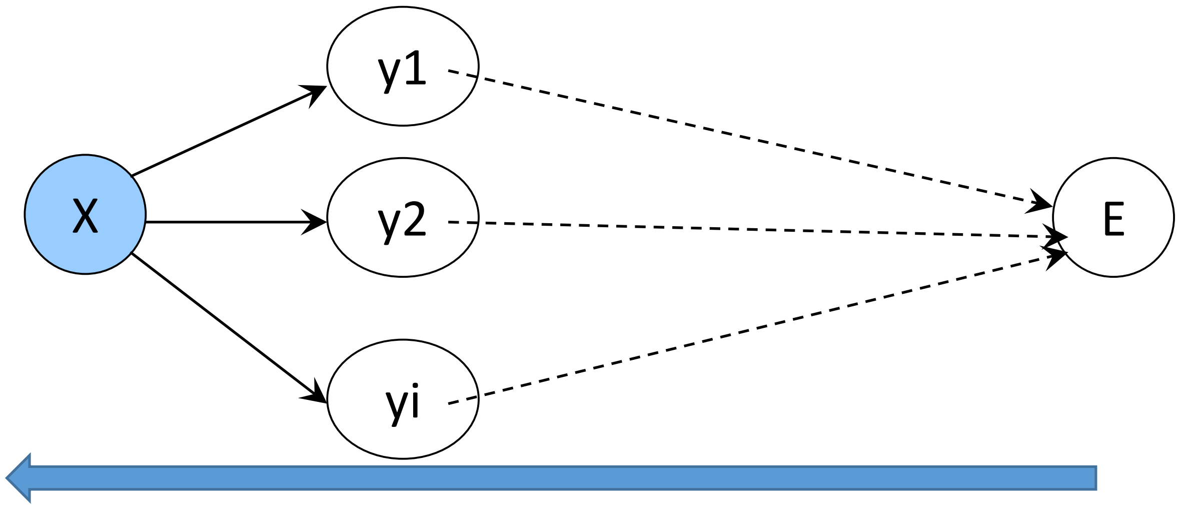


状态转移方程：

由已求得的状态来求未知状态
递推关系式称为状态转移方程。

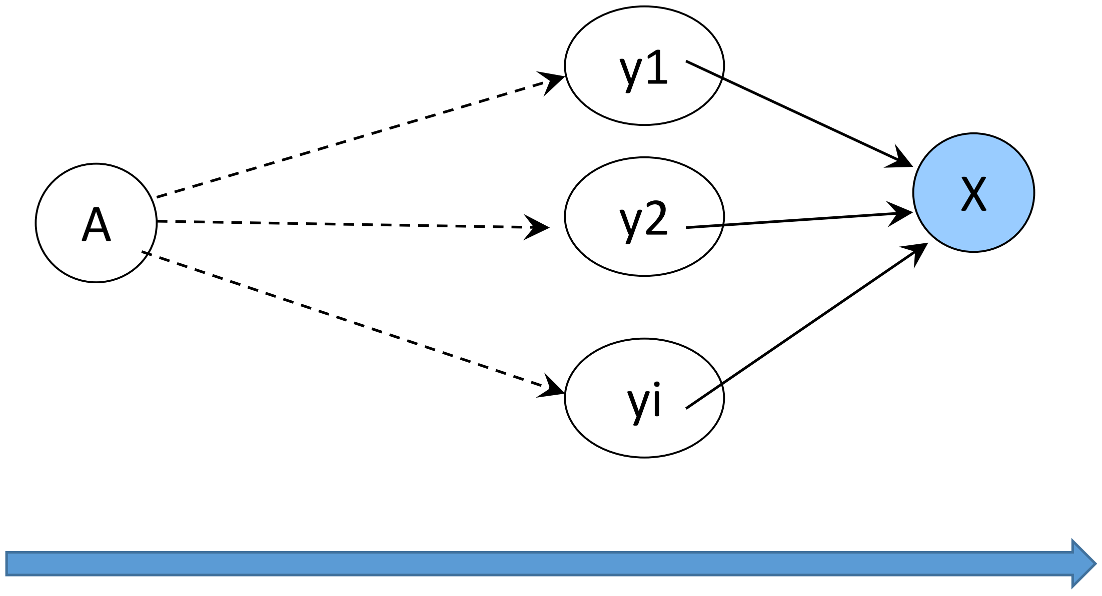
倒推： **f[x]:x到终点E的最短距离。f[E]=0;目标是f_1[A]**

$$f_k[x]=\min\{f_{k+1}[y_i]+d[x,y_i]\}$$



顺推： f(x)为A点到x的最短距离， f(A)=0;目标f(E)

$$f_k[x]=\min\{f_{k-1}[y_i]+d[y_i,x]\}$$



倒推格式为：

$f[U_n]$=初始值；

for k←n-1 downto 1 do {枚举阶段}

for U取遍所有状态 do {枚举状态}

for X取遍所有决策 do {枚举决策}

$f[U_k]=opt\{f[U_{k+1}]+L[U_k,X_k]\}$；

//$L[U_k,X_k]$}：状态U_k通过策略X_k到达状态U_{k+1}的费用输出：

$f[U_1]$:目标

顺推格式为：

$f[U_1]$=初始值;

for k←2 to n do {枚举每一个阶段}

for U取遍所有状态 do

for X取遍所有决策 do

$f[U_k]=opt\{f[U_{k-1}]+L[U_{k-1},X_{k-1}]\}$;

//$L[U_{k-1},X_{k-1}]$}：状态U_{k-1}通过策略X_{k-1}到达状态U_k 的费用

输出：$f[U_n]$:目标

设计动态规划法的步骤：

1、找出最优解的性质，并刻画其结构特征；
2、递归地定义最优值（写出动态规划方程）；
3、以自底向上的方式计算出最优值；
   记忆化搜索（树型）、递推

4、根据计算最优值时得到的信息，构造一个最优解。

解决问题：

1.求最优值问题

2.统计问题

一.坐标型

在二维坐标系内，规定了方向，求最优值问题。

比较容易根据方向写出动态规划方程：

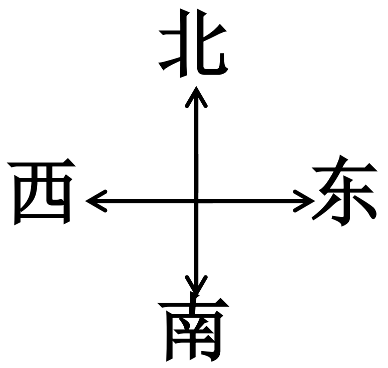
一般方程也是二维的f[i][j]

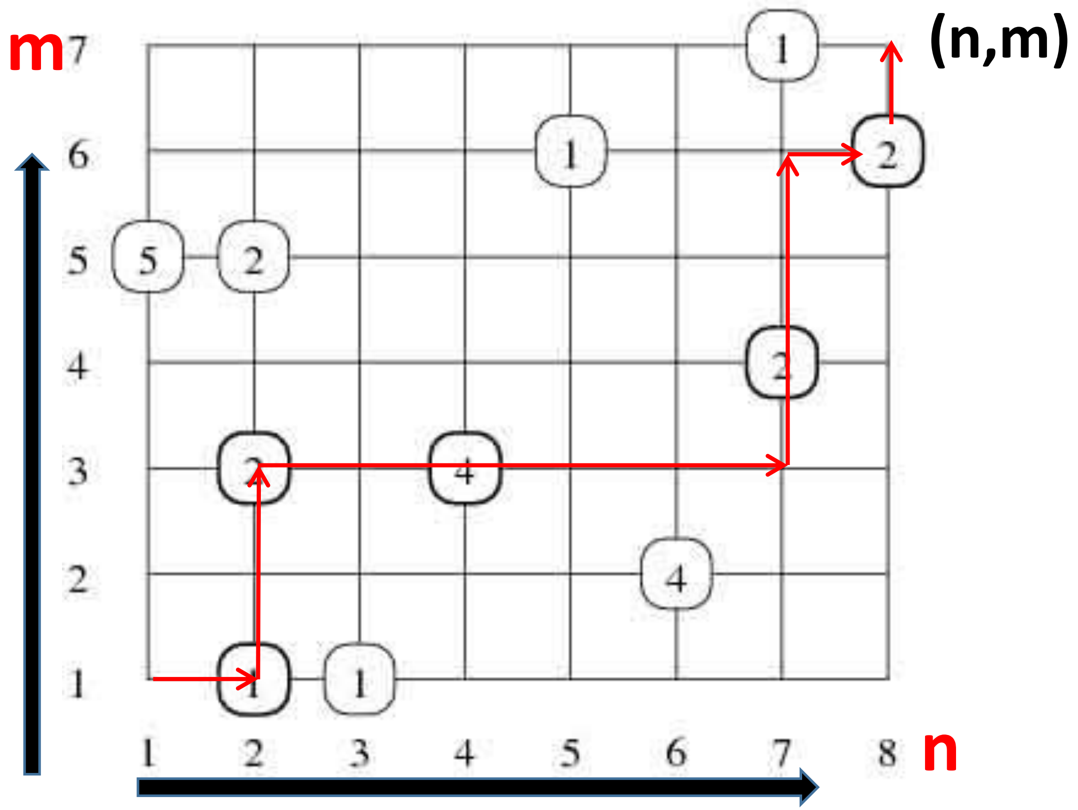
例：公共汽车

【问题描述】

一个城市的道路，南北向的路有n条，并由西向东从1标记到n,东西向的路有m条，并从南向北从1标记到m,每一个交叉点代表一个路口，有的路口有正在等车的乘客。一辆公共汽车将从(1,1)点驶到（n,m）点，车只能向东或者向北开.

问：司机怎么走能接到最多的乘客。





【输入】

第一行是n,m,和k,其中k是有乘客的路口的个数。以下k行是有乘客的路口的坐标和乘客的数量。已知每个路口的乘客数量不超过1000000。n,m<=1000.

【输出】

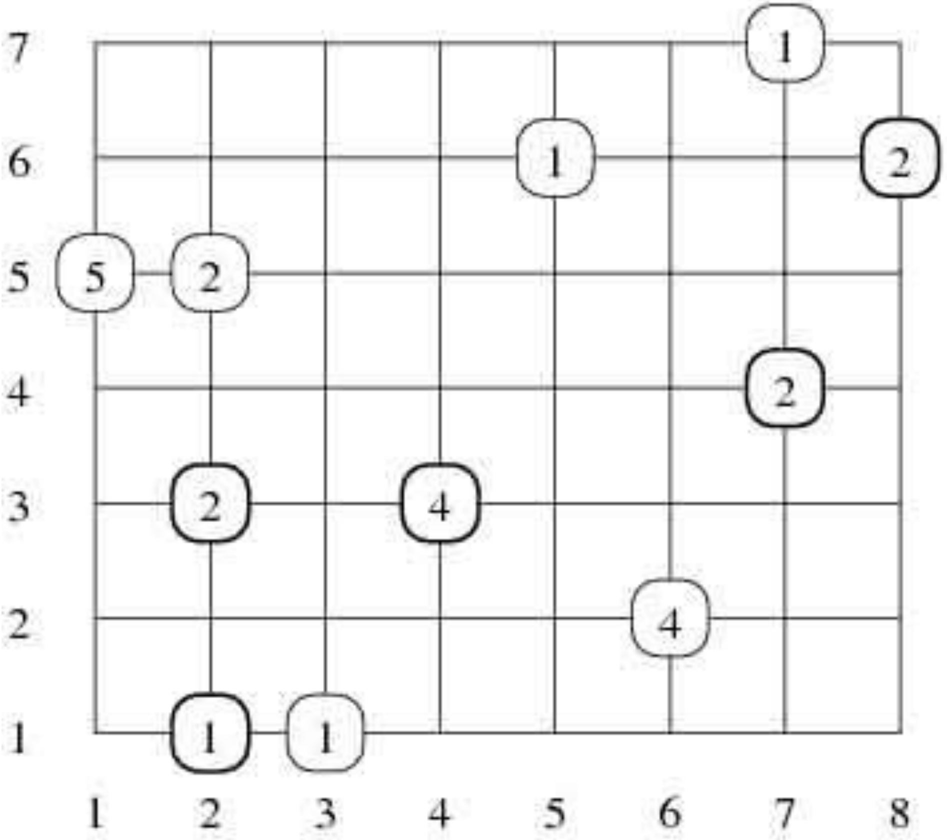
接到的最多的乘客数。

bus.in	bus.out
8 7 11 4 3 4 6 2 4 2 3 2 5 6 1 2 5 2 1 5 5 2 1 1 3 1 1 7 7 1 7 4 2 8 6 2	11

a[i,j] (i,j)位置的人数，

f[i,j]:从（1,1）走到（i, j）能接的最多人数。

f[i,j]:=max{f[i-1,j],f[i,j-1]}+a[i,j]



```
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
        f[i][j]=max(f[i][j-1]+f[i-1][j])+a[i][j];
cout<<f[n][m];
```

训练：一本通：

1284

1287

二.线性模型：

LIS (Longest Increasing Subsequence）最长上升子序列：

给定n个元素的数列，求最长的上升子序列长度(LIS)。

最长上升子序列长度（LIS）：

8 2 7 1 9 10 1 4 3

找出以每个元素为**起点（首元素）**的所有的上升子序列:

8 9 10

2 7 9 10

7 9 10

1 9 10

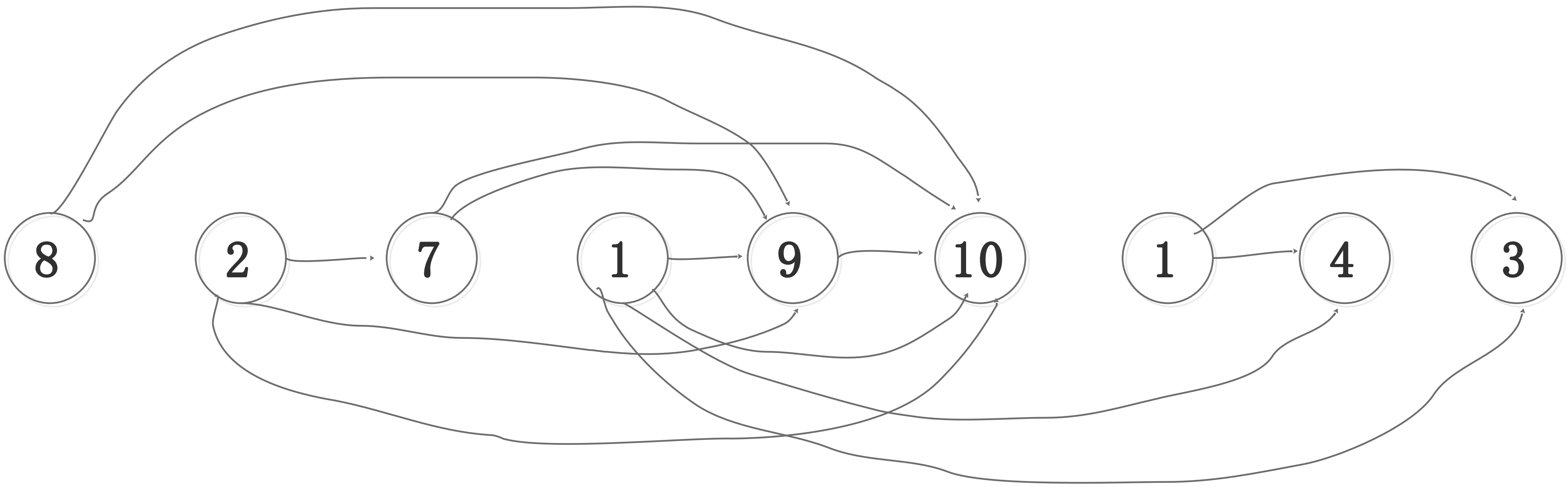
9 10

1 4

4

3

每个数向后面比他大的点建立有向边；
求最长路（顶点数最多）



$f[i]=max(f[j])+1\ (i<j<=n\&\&a[i]<a[j])$

方法1：暴力搜索

8 2 7 1 9 10 1 4 3

找出以每个元素为**起点**的所有的上升子序列；
然后选择最长的即可。

① 8 9 10
② 2 7 9 10
③ 7 9 10
④ 1 9 10
⑤ 9 10
⑥ 10
⑦ 1 4
⑧ 4
⑨ 3

怎么找出这些序列？

```
int dfs(int i){
    //以a[i]为开头的最长递增子序列长度
    int s=0;
    for(int j=i+1;j<=n;j++)
        if(a[i]<a[j])s=max(s,dfs(j));
    s++;
    return s;
}
```

自底向上倒着找

```
ans=0;
for(int i=1;i<=n;i++)
    ans=max(ans,dfs(i));
cout<<ans<<endl;
```

为什么超时?
怎样在此基础上改进搜索?

方法2：记忆化搜索

以每个元素为起点的LIS是固定不变的，每次求完可以记录下来，供后面直接使用，避免重复搜索。

f[i]:以a[i]开始的最长上升子序列长度（初始为0）,一旦求过f[i],一定是f[i]>=1,起码有a[i].

```
int dfs(int i){
    //以a[i]开始的最长序列长度
    if(f[i]>0)return f[i];
    f[i]=0;
    for(int j=i+1;j<=n;j++)
        if(a[i]<a[j])f[i]=max(f[i],dfs(j));
    f[i]++;
    return f[i];
}
```

方法3:倒序递推求f[i]

以a[i]为**起点**元素的最长上升子序列长度

每个数向后面比他大的点建立有向边;

求最长路（顶点数最多）

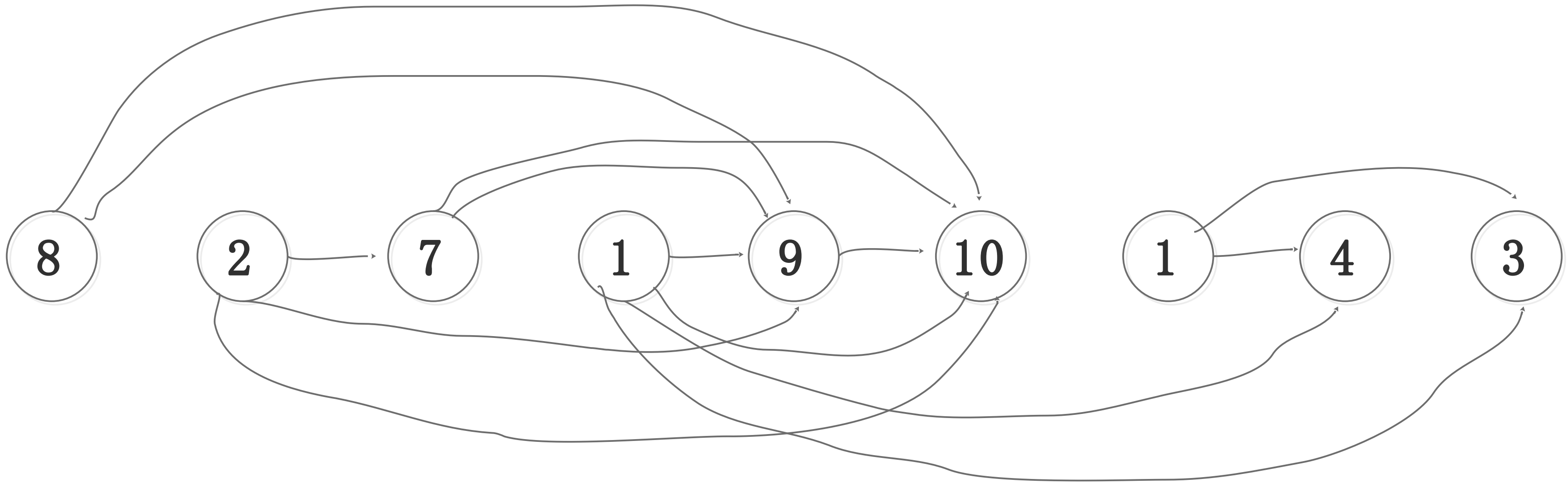
观察：边的顺序：a[i]向后的边i+1,i+1,..,n中选择。

可以直接倒序求即可。

f[n]=1;

f[i]=max(f[j])+1 (i<j<=n&&a[i]<a[j])

ans=max(f[i]);



倒推求f[i]:

以a[i]为**起点**元素的最长上升子序列长度

```
cin>>n;
for(int i=1;i<=n;i++)cin>>a[i];
f[n]=1;
for(int i=n-1;i>=1;i--){
	f[i]=0;
	for(int j=i+1;j<=n;j++)
		if(a[i]<a[j])f[i]=max(f[i],f[j]);
	f[i]++;
}
int ans=0;
for(int i=1;i<=n;i++)
	ans=max(ans,f[i]);
cout<<ans<<endl;
```

方法4：正向递推：

f[i]:以a[i]为**结束（最后一个元素）**元素的最长子序列长度。

8 2 7 1 9 10 1 4 3

① **8**

② **2**

③ 2 **7**

④ **1**

⑤ 2 7 **9**

⑥ 2 7 9 **10**

⑦ **1**

⑧ 1 **4**

⑨ 1 **3**

正推求f[i]：

以a[i]为<u>终点</u>元素的最长子序列长度。

方程：

f[1]=1;

f[i]=max(f[j])+1 (1<=j<i&&a[j]<a[i])

ans=max(f[i]);

正推：

```
f[1]=1;
for(int i=2;i<=n;i++){
    f[i]=0;
    for(int j=1;j<i;j++)
        if(a[j]<a[i])f[i]=max(f[i],f[j]);
    f[i]++;
}
```

输出最优方案:

一本通1259.

求最长不下降序列长度及输出改序列。

【输入样例】
14
13 7 9 16 38 24 37 18 44 19 21 22 63 15

【输出样例】
max=8
7 9 16 18 19 21 22 63

正推: 8 2 7 1 9 10 1 4 3

正推求f[i]:

以a[i]为**终点**元素的最长子序列长度。

方程:

f[1]=1;

f[i]=max(f[j])+1 (1<=j<i&&a[j]<a[i])

ans=max(f[i]);

p[i]记录f[i]去最优值时的j。

找到最大的f[i],然后向**前**找即可（递归实现，类似bfs输出路径，输出父亲结点）。

```
f[1]=1;
p[1]=0;
for(int i=2;i<=n;i++){
    f[i]=0;
    for(int j=1;j<i;j++)
        if(a[j]<=a[i]&&f[j]>f[i]){
            f[i]=f[j];
            p[i]=j;
        }
    f[i]++;
}
int ans=f[1],k=1;
for(int i=2;i<=n;i++)
    if(f[i]>ans)ans=f[k=i];
cout<<"max="<<ans<<endl;
dfs(k);
```

```
void dfs(int i){
    if(p[i]>0)dfs(p[i]);
    cout<<a[i]<<" ";
}
```

LIS的优化：O(n*logn)

以严格递增为例：
f[i]:以a[i]为**终点**元素（序列最后元素）的最长子序列长度。正向求

8	2	7	1	9	10	1	4	3
1	1	2	1	3	4	1	2	2

长度相同的递增序列中最后一元素小的比大的好用（对后面更有用）

① **8**
② **2**
③ 2 **7**
④ **1**
⑤ 2 7 **9**
⑥ 2 7 9 **10**
⑦ **1**
⑧ 1 **4**
⑨ 1 **3**

长度相同的递增序列中最后一元素小的比大的好用（对后面作用）

重新定义：f[i]表示长度为i的上升子序列最后一个数最小是多少.

数组f是单增的

8	2	7	1	9	10	1	4	3
1	1	2	1	3	4	1	2	2

f[]= 1 3 9 10

最后f的长度cnt=4是答案

注意：最后的f[]并不是那个最长的序列元素

长度相同的递增序列中最后一元素小的比大的好用（对后面作用）
重新定义：f[i]表示长度为i的上升子序列最后一个数最小是多少.
数组f是单增的

8	2	7	1	9	10	1	4	3
1	1	2	1	3	4	1	2	2

f[]= 1 3 9 10
最后f的长度cnt=4是答案
注意：最后的f[]并不是那个最长的序列元素

```
9
8  2  7  1  9  10  1  4  3
8
2
2 7
1 7
1 7 9
1 7 9 10
1 7 9 10
1 4 9 10
1 3 9 10
4
```

8

4 6 5 12 15 10 6 1

从左向右依次处理a[i]，把他插在比他小的最大那个元素后面。维护f递增，类似插入排序，不同的是这里是替换（也可能在最后插入新增元素，长度增加了）

f[i]递增，二分查找，时间O(n*logn)

8

4 6 5 12 15 10 6 1

```
8
4 6 5 12 15 10 6 1
4
4 6
4 5
4 5 12
4 5 12 15
4 5 10 15
4 5 6 15
1 5 6 15
4
```

从左向右依次处理a[i]，把他插在最大比他小的那个元素后面。维护f递增，类似插入排序，不同的是这里是替换（也可能在最后插入新增元素，长度增加了）

f[i]递增，二分查找，时间O(n*logn)

查找满足条件的最小值（递增序列中找>=x最小值）：

```
f[1]=a[1];
cnt=1;
for(int i=2;i<=n;i++){
    int l=1,r=cnt+1;
    f[r]=1e9;
    while(l<r){
        int mid=(l+r)>>1;
        if(f[mid]>=a[i])r=mid;
        else l=mid+1;
    }
    f[l]=a[i];
    if(l==cnt+1)cnt++;
}
cout<<cnt<<endl;
```

有没有等号问题，可以自己尝试

6

2 1 4 8 4 4

if(f[mid]>=a[i])r=mid;

严格递增得到f[i]:

1 4 8

if(f[mid]>a[i])r=mid;

非递减得到f[i]:

1 4 4 4

知识扩展：

最长上升子序列长度； <
最长不下降子序列长度； <=
最长下降子序列长度； >
最长不上升子序列长度。 >=
应用广泛！

课后训练：

① p1091/yt1264 【例9.8】合唱队形

② 1283 登山

③ 1286 怪盗基德的滑翔翼

④ **P2782 /yt**1263 【例9.7】友好城市

⑤ p1020/yt1260 拦截导弹NOIP999