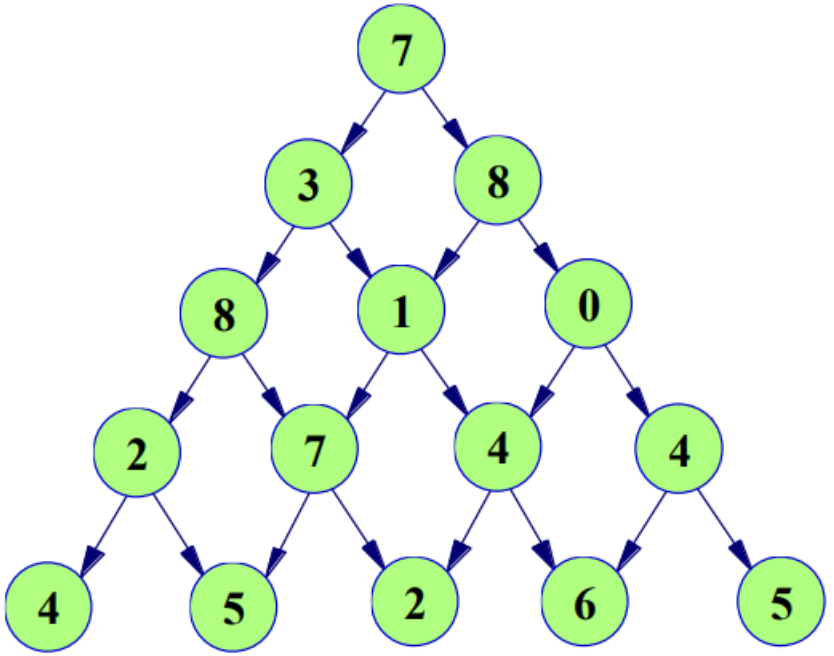
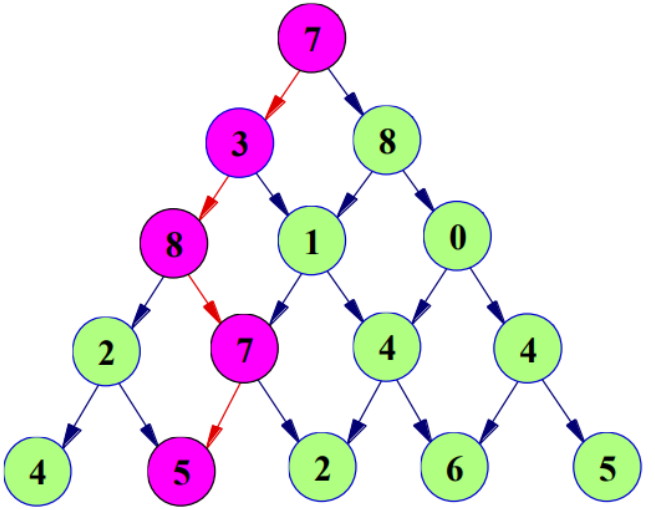
动态规划基础

引例：数字三角形

给出一个数字三角形。请编一个程序计算从顶至底的某处的一条路径，每一步可沿左斜线向下或右斜线向下走，使该路径所经过的数字的总和最大。



答案：



路径数字最大和：7+3+8+7+5=30

引例：数字三角形

我会搜索！

但是，每个位置都会被计算多次，n<=100时必定超时。

思考：

（1）能否改进使得搜索能够通过

（2）是否有其他做法

- 1
- 1 1
- 1 2 1
- 1 3 3 1
- 1 4 6 4 1
- 1 5 10 10 5 1
- 1 6 15 20 15 6 1
- 1 7 21 35 35 21 7 1
- 1 8 28 56 70 56 28 8 1
- 1 9 36 84 126 126 84 36 9 1

本章主要内容

一、动态规划应用的前提条件

二、动态规划的状态设计

三、线性规划的两种实现方法

四、线性动态规划

五、01背包

六、多重背包

七、完全背包

一、动态规划应用的前提条件

DP能够处理的问题的特征：

1. 可以将问题分解为若干个相同形式的子问题
2. 可以通过子问题的答案得到问题的答案
3. 子问题中有很多是重复的，求解时记录所有可能出现的子问题的答案，遇到重复的子问题时直接查表得到答案，使得所有出现的问题只被求解一次。
4. 某一阶段的状态只受前一阶段的状态的影响，不受更早阶段的影响。

二、动态规划的状态设计

名词解释：

状态：描述子问题

转移：通过子问题的答案求出母问题的答案

初始量：最小的子问题（解很显然或可以通过人工计算获得）

最终答案：最终状态对应的答案

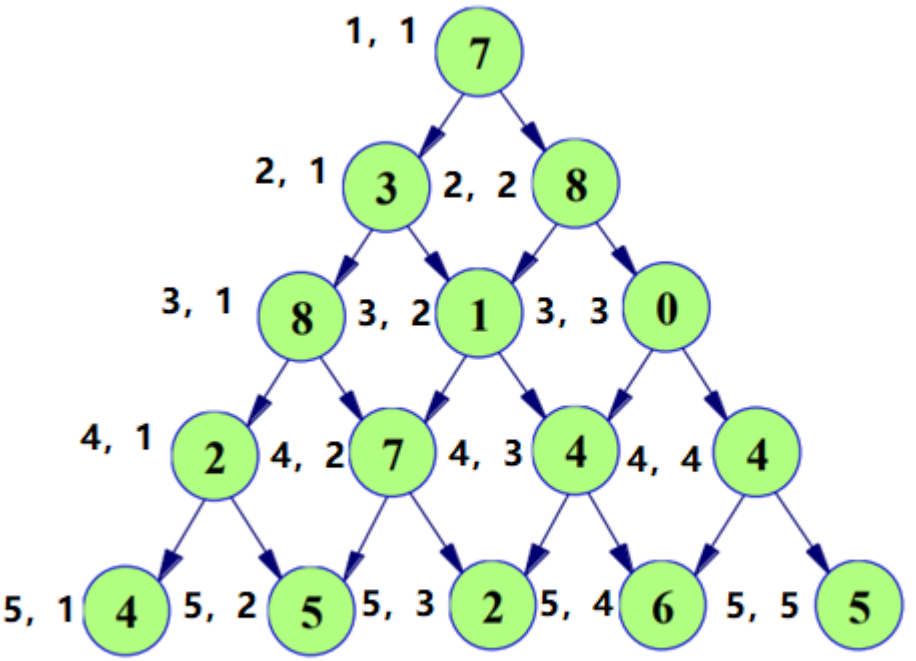
二、动态规划的状态设计

以引例为例

最终答案：

从顶层结点沿着路径走到底层结点的权值和的最大值

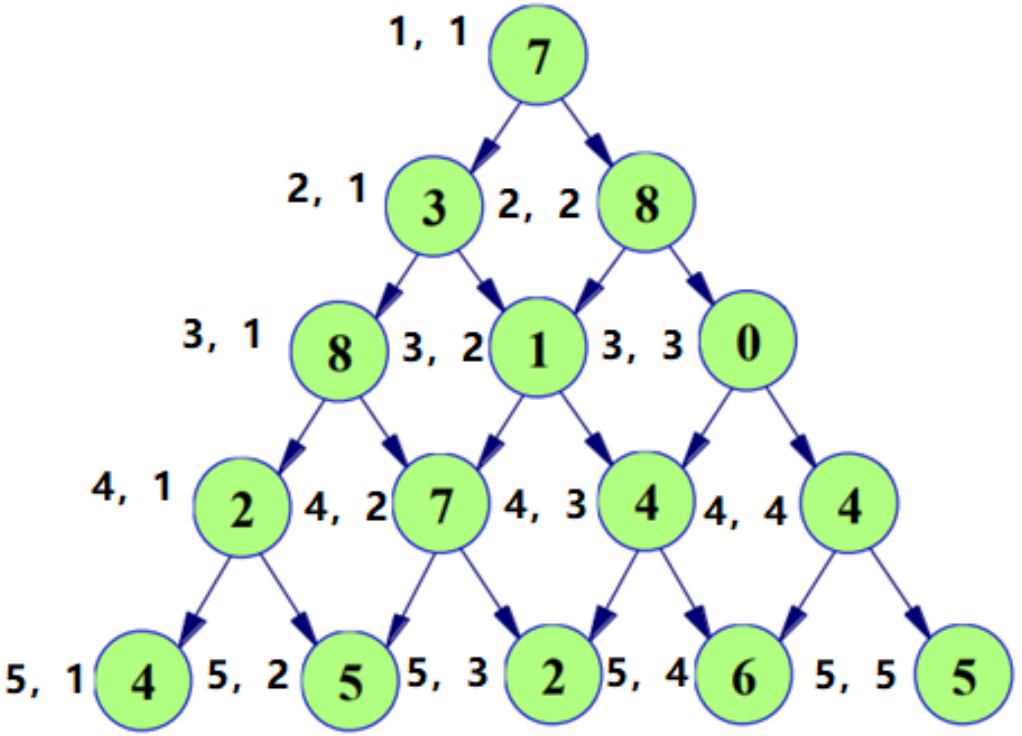
答案可以表示成什么？



二、动态规划的状态设计

设f(x)为到达底层节点5,x时权值和的最大值，那么答案为

max{f(1),f(2),f(3),f(4),f(5)}



二、动态规划的状态设计

状态：

f(x)到达底层节点(5,x)

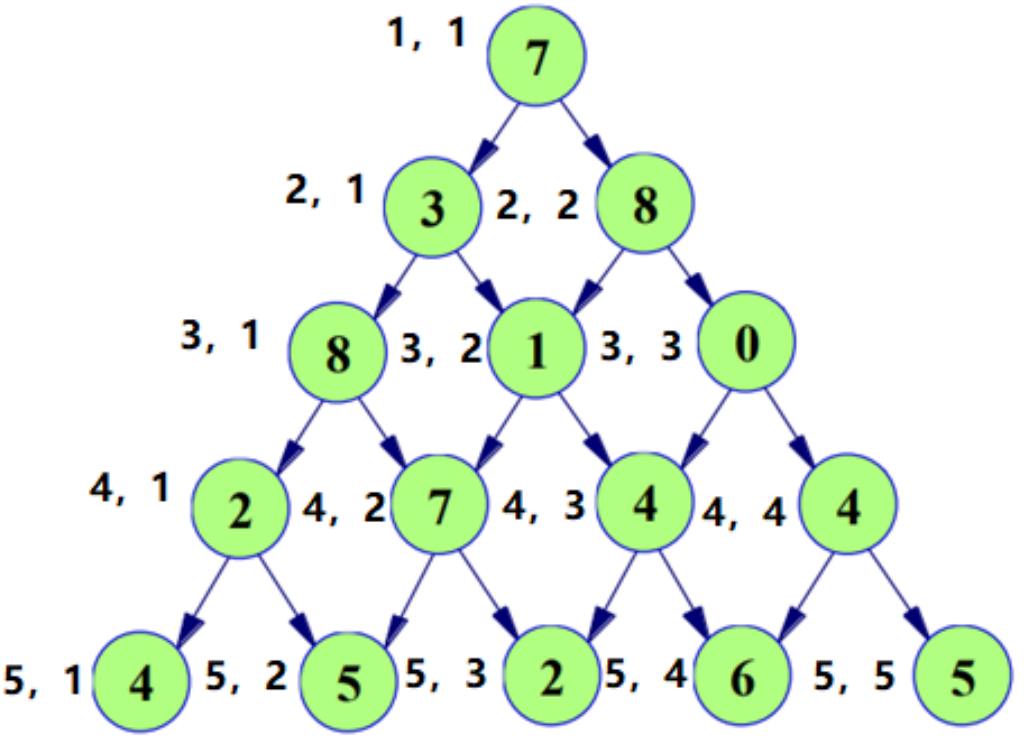
的权值和的最大值

因为数字三角形有多层，

那么我们给f(x)增加一维

dp[i][j]：到达点(i,j)的

权值和的最大值



二、动态规划的状态设计

转移：

dp[i][j]：到达点(i,j)的

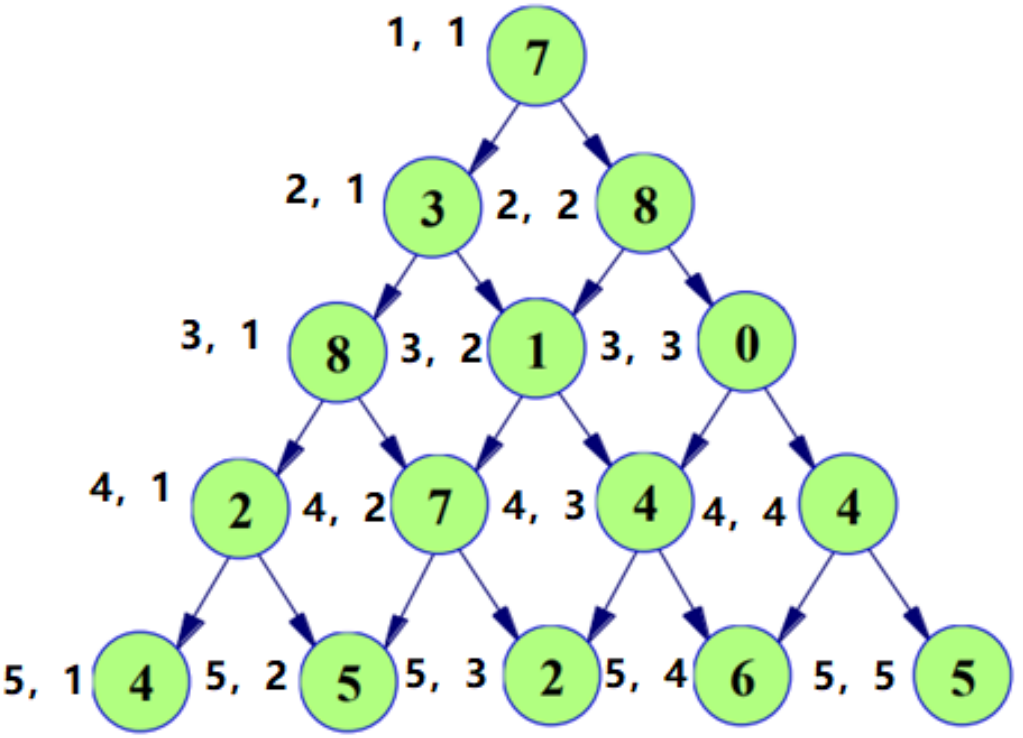
权值和的最大值

点(i,j)可以由哪些点到达？

(i-1,j-1)(i-1,j)

dp[i-1][j-1]

dp[i-1][j]

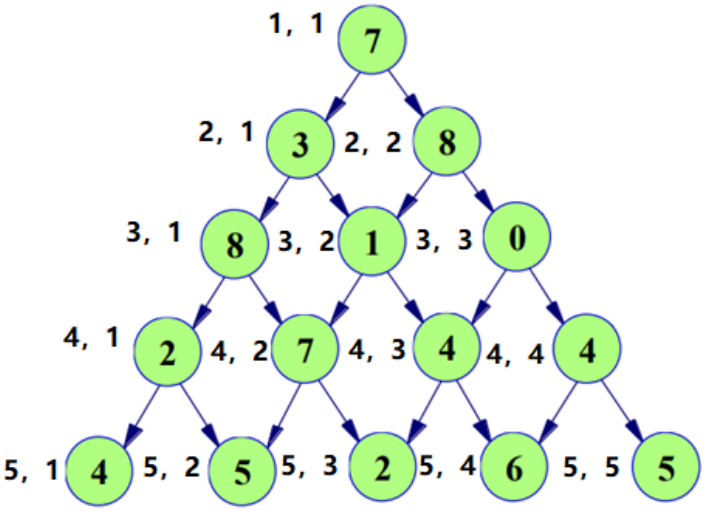


二、动态规划的状态设计

转移：

dp[i][j]：到达点(i,j)的

权值和的最大值



如何转移？

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

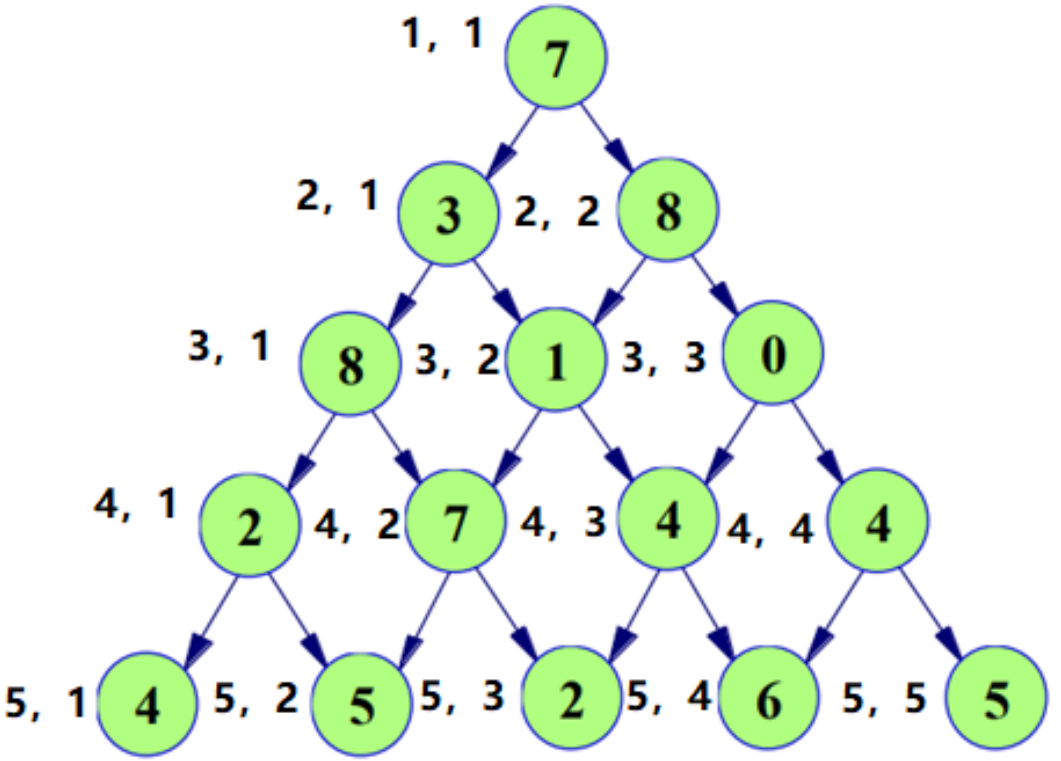
二、动态规划的状态设计

初始值：

dp[i][j]：到达点(i,j)

权值和的最大值

dp[1][1]=a[1][1]



三、动态规划的两种实现方法

递推实现：

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

三、动态规划的两种实现方法

递归实现（记忆化搜索）：

```
int dfs(int i,int j){
  if(j<1||j>i) return 0;
  if(i==n) return a[i][j];
  if(f[i][j]) return f[i][j];
  return f[i][j]=max(dfs(i+1,j),dfs(i+1,j+1))+a[i][j];
}
```

四、线性动态规划

- 线性动态规划状态是一维的（f[i]）。
- 正推：第i个元素的最优值只与前i-1个元素的最优值有关。
- 倒推：第i个元素的最优值只第i+1个元素之后的最优值有关。
- 经典的线性DP题目有最长上升子序列、最大连续子序列和、最长公共子序列等

四、线性动态规划

- 例题：最长上升子序列
- 给定n个元素的数列，求最长上升子序列长度
- 子序列：从原序列中选出若干个元素，不改变其原有顺序形成的新序列成为原序列的子序列
- 上升序列：对于任意1<=i<j<=n , ai<aj
- 例：8 2 7 1 9 10 1 4 3

四、线性动态规划

- 正推：
- 状态设计：f[i]表示以序列中第i个元素结尾的最长上升子序列长度
- 初始值：f[1]=1;
- 最终答案:ans=max{f[i]}
- 转移：f[i]=max{f[j]}+1(j<i , a[i]>a[j])

四、线性动态规划

- 正推：
- for(int i=1;i<=n;i++){
- f[i]=1;
- for(int j=1;j<i;j++){
- if(a[j]<a[i])
- f[i]=max(f[i],f[j]+1);
- }
- }

四、线性动态规划

- 逆推：
- 状态设计：f[i]表示以序列中第i个元素作为开头的最长上升子序列长度
- 初始值：f[n]=1;
- 最终答案:ans=max{f[i]}
- 转移：f[i]=max{f[j]}+1(j>i , a[i]<a[j])

四、线性动态规划

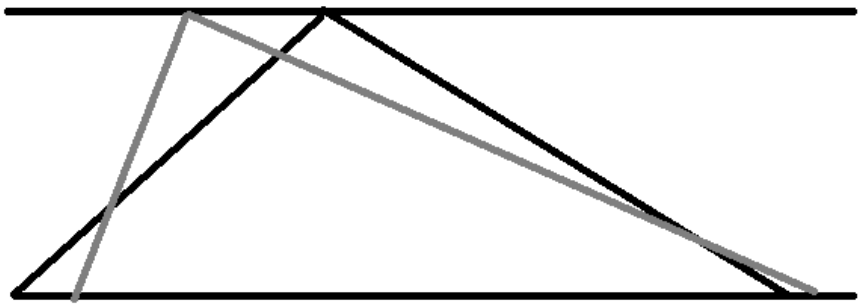
- 逆推：
- for(int i=n;i>=1;i--){
- f[i]=1;
- for(int j=i+1;j<=n;j++){
- if(a[i]<a[j])
- f[i]=max(f[i],f[j]+1);
- }
- }

四、线性动态规划

- **例题：合唱队形**
  - n 位同学站成一排，音乐老师要请其中的 n−k 位同学出列，使得剩下的 k 位同学排成合唱队形。
  - 合唱队形是指这样的一种队形：设 k 位同学从左到右依次编号为 1,2, … ,k他们的身高分别为 t1,t2, … ,tk则他们的身高满足 t1<…< ti > ti+1 > … > tk (1≤i≤k)。
  - 你的任务是，已知所有 n 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

四、线性动态规划

- **例题：合唱队形**
  - 计算最少需要几位同学出列，即最多能留下多少同学。
  - 合唱队形应当形似下图：



  - 队形中间为最高点，左侧为上升子序列，右侧为下降子序列，当长度和最大时为最优合唱队形。

四、线性动态规划

- **例题：合唱队形**
  - 对于某一点为最高点的最优解为左侧选择以该点结尾的最长上升子序列，右侧选择以该点开始的最长下降子序列。
  - 整个问题的最优解为枚举每一点最为最高点，取最大值。
  - 所以，只需正推求一次最长上升子序列(f[i])
  - 倒推求一次最长下降子序列(g[i])
  - 枚举最高点取最大值，ans=max(f[i]+g[i]-1)
  - 最小出队人数:n-ans

四、线性动态规划

- 例题：合唱队形

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

四、线性动态规划

- **例题：**最长公共子序列
- 给出1,2,…,n(n<=10^3)的两个排列P1和P2，求它们的最长公共子序列。
- 若一个序列S分别是序列A和序列B的子序列，则将序列S称为序列A和序列B的子序列
- 例如：
- A={3,2,1,4,5}
- B={1,2,3,4,5}
- 最长公共子序列为{3,4,5}（不唯一）

四、线性动态规划

- **例题：**最长公共子序列
- 状态设计：设计两个子序列的比较，状态也应当增加一维，f[i][j]表示序列A的前i个元素与序列B的前j个元素的最长公共子序列长度
- 初始值：f[0][0]=0;
- 最终答案:f[n][n]
- 转移：if(ai==bj) f[i][j]=f[i-1][j-1]+1;
- else f[i][j]=max{f[i][j-1],f[i-1][j]}

四、线性动态规划

- **例题：**最长公共子序列
- 另一种做法？
- 考虑将序列重新标号:
- A={3,2,1,4,5}->{a,b,c,d,e}
- B={1,2,3,4,5}->{c,b,a,d,e}
- 重新编号后，答案显然不会改变，且一定是A的子序列
- A本身是单调递增的，则答案也一定是单调递增的
- 同时，B中单调递增的子序列也一定是A的子序列
- 转化为求B中的最长递增子序列

四、线性动态规划

- **例题：**最长公共子序列
- 有何好处？
- 最长递增子序列转移为：f[i]=max{f[j]}+1(j<i , a[i]>a[j])
- 该过程可以从O(n)复杂度优化（如使用线段树）到O(logn)，算法时间复杂度可以从O(n^2)优化至O(nlogn)解决数据范围更大的问题（n<=10^5）
- ~~但是超过了今天的授课范围所以不讲~~

五、01背包

例题：01背包

有N件物品和一个容量为V的背包，每种物品只有一个，放入第i件物品消耗的空间是Ci，得到的价值是Wi。求解将哪些物品装入背包可使价值总和最大？

状态设计：f[i][v]表示前i件物品恰好放入一个容量为v的背包可以获得的最大价值。

初始值：f[0][0…v]=0;

最终答案：f[N][V]

转移？

五、01背包

例题：01背包

每种物品仅有一件，可以选择放或不放。

如果不放第i件物品,f[i][v]=f[i-1][v]

如果放第i件物品f[i][v]=f[i-1][v-Ci]+Wi

两者取最优f[i][v]=max{f[i-1][v],f[i-1][v-Ci]+Wi}

```
for(int i=0;i<V;i++) f[0][i]=0;
for(int i=1;i<=n;i++)
        for(int j=c[i];j<=v;j++)
                f[i][j]=max(f[i-1][j],f[i-1][j-c[i]]+w[i]);
```

五、01背包

例题：01背包

空间复杂度的优化：

观察转移方程可以发现，所有i状态只和i-1的状态有关，状态设计可以改为f[0/1][V] 0/1交替表示当前组和上一组。

能否更优？

我们发现，转移时只与容量更小的状态有关，所以可以考虑直接去掉第一维，枚举重量时从大到小枚举，转移方程变为

f[v]=max{f[v],f[v-ci]+wi}等式右侧的f[v]和f[v-ci]+wi都是本次更新前的，即f[i-1][…]对应的值。

五、01背包

例题：01背包

```
for(int i=0;i<V;i++) f[i]=0;
for(int i=1;i<=n;i++)
        for(int j=V;j>=c[i];j--)
                f[j]=max(f[j],f[j-c[i]]+w[i]);
```

六、完全背包

例题：完全背包

有N种物品和一个容量为V 的背包，每种物品都有无限件可用。放入第i种 物品的耗费的空间是Ci，得到的价值是Wi。求解：将哪些物品装入背包，可使 这些物品的耗费的空间总和不超过背包容量，且价值总和最大。

与01背包不同，所有物品都有无限件，每一种物品的策略也不再是取或不取两种，而是可以取0,1,2,……,V/Ci种多种策略，按照01背包的思路，只需要再枚举选择几件物品即可：

F[i][v]=max{f[i-1][V-kCi]+kWi}(0<=kCi<=V)

状态依然是O(VN)个，但转移不再是O(1)。优化？

六、完全背包

例题：完全背包

```
dfs(int now,int v){
        if(now==n+1) return 0;
        if(f[now][v]!=-1) return f[now][v];
        int ans=-INF;
        for(int k=0;k*c[i]<=v;k++){
                ans=max(ans,dfs(now+1,v-k*c[i])+k*w[i]);
        }
        return f[now][v]=ans;
}
```

方便后续讲解，先看一下记忆化搜索的写法

六、完全背包

时间复杂度优化瓶颈在于循环。

现在转移时一次性枚举了第i个物品的所有选择策略，但是实际上，一次放入3件物品和分3次放入1件物品是相同的，所以：

```
dfs(int now,int v){
        if(now==n+1) return 0;
        if(f[now][v]!=-1) return f[now][v];
        int ans=-INF;
        ans=max(ans,dfs(now+1,v));
        if(v>=c[i]) ans=max(ans,dfs(now,v-c[i])+w[i]);
        return f[now][v]=ans;
}
```

六、完全背包

这样，转移方程就变成了：

f[i][v]=max{f[i-1][v],f[i][v-Ci]+Wi};

写成递推形式：

```
for(int i=0;i<V;i++) f[0][i]=0;
for(int i=1;i<=n;i++)
        for(int j=c[i];j<=V;j++)
                f[i][j]=max(f[i-1][j],f[i][j-c[i]]+w[i]);
```

空间复杂度能否优化?

六、完全背包

```
for(int i=0;i<V;i++) f[i]=0;
for(int i=1;i<=n;i++)
        for(int j=c[i];j<=V;j++)
                f[j]=max(f[j],f[j-c[i]]+w[i]);
```

和01背包优化空间复杂度后的代码只有内层循环顺序不同

因为转移时用的是更新后的答案。

七、多重背包

例题：多重背包

有N种物品和一个容量为V 的背包。第i种物品最多有Mi件可用，每件耗费 的空间是Ci，价值是Wi。求解将哪些物品装入背包可使这些物品的耗费的空间 总和不超过背包容量，且价值总和最大。

与完全背包类似，只需要在枚举选择物品数量时将选择数量的上限改为Mi即可

f[i][j]=max{f[i-1][v-k*Ci]+k*Wi}(0<=k<=Mi&& 0<=k*Ci<=V)

能否优化？

七、多重背包

例题：多重背包

将第i种物品拆成Mi件物品，就转化成了01背包问题。但是，复杂度依然没有改变。

这么拆正确的原因是Mi个1自然可以表示0~Mi中的所有数

但是，这并不是最优的，如3个1可以表示0~3中的所有整数，但一个1和一个2也可以表示0~3中的所有整数。

我们可以将物品拆成2^0,2^1,2^2,…,2^(k-1),Mi-2^k+1件一组的物品，其中k时满足Mi-2^k+1>0的最大整数。这样也可以表示0-Mi中的所有整数。

例如，最多取13件的物品可分为四组各1,2,4,6件物品

七、多重背包

例题：多重背包

正确性？

考虑二进制，1=(0001) 2=(0010) 4=(0100)

显然，1,2,4能表示(0000)~(0111)，即0~2^k-1中的所有整数

先将所有物品全部选择，在放弃除最后一组外的若干组物品，显然能够表示(Mi-2^k+1)~(Mi)中的所有整数

Mi<2^(k+1) = 2*k^2

Mi-2^k+1<k^2+1

Mi-2^k+1<=k^2 所以能表示0~Mi中的所有整数

七、区间DP

区间DP是通过短区间的答案推出长区间的答案的一种DP。

例题：石子合并

有N堆石子(N≤100)排成一排。现要将石子合并 成一堆.规定每次只能选相临的两堆合并成一堆新的 石子,并将新的一堆的石子数,记为该次合并的得分.选择一种合并石子的方案,使得做N-1次合并,得分的总和最少。

如：

1 3 5 2

22

七、区间DP

例题：石子合并

贪心？

7 4 4 7

8+15+22=45

11+11+22=44

状态设计f[i][j]：区间i~j的最优解

初始值：f[i][i]=0

转移？

七、区间DP

例题：石子合并

三堆石子的合并方案：

8 3 6

11+(11+6)=30

9+(8+9)=26

Ans=min(30,26)=26

七、区间DP

例题：石子合并

四堆石子的合并方案：

8 5 5 8

最后一步合并的情况：

8+(5+5+8)

(8+5)+(5+8)

(8+5+5)+8

可以发现，是先把小区间合并好，再把两个合并后的区间合并

七、区间DP

例题：石子合并

DP转移也可以如此进行

f[i][j]=min{

f[i][j]+f[i+1][j]，

f[i][i+1]+f[i+2][j]，

……

f[i][j-1]+f[j][j]}+sum[i][j]

其中sum[i][j]=a[i]+a[i][i+1]+……+a[j]

七、区间DP

例题：石子合并

写成dp方程就是

f[i][j]=min{f[i][k],f[k+1][j]}+sum[i][j]

其中sum[i][j]根据前缀和思想可转化为sum[j]-sum[i-1]

f[i][j]=min{f[i][k],f[k+1][j]}+ sum[j]-sum[i-1]

最终答案：f[1][n]

七、区间DP

例题：石子合并

```
memset(f,0x3f,sizeof(f));
for(int i=1;i<=n;i++) f[i][i]=0;
for(int p=2;p<=n;p++){
        for(int i=1;i<=n-p+1;i++){
                int j=i+p-1;
                for(int k=i;k<j;k++){
                        f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]+sum[j]-sum[i-1]);
                }
        }
}
```

七、区间DP

例题：环形石子合并

有N堆石子(N≤100)排成环形。现要将石子合并 成一堆.规定每次只能选相临的两堆合并成一堆新的 石子,并将新的一堆的石子数,记为该次合并的得分.选择一种合并石子的方案,使得做N-1次合并,得分的总和最少。

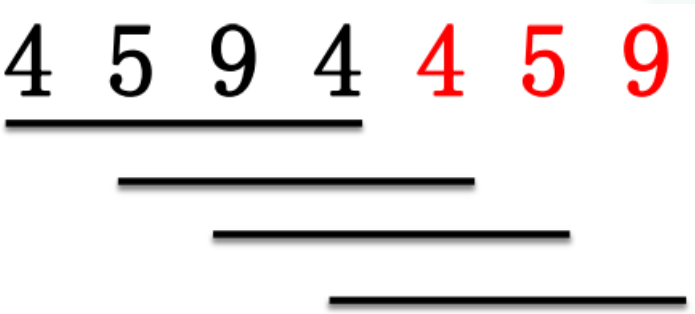
和上一题不同的地方在于是环形的

4 5 9 4可以先合并4和4

但是，实际上我们把这个序列写两遍

七、区间DP

例题：环形石子合并



我们先合并4 4，再合并8 9，最后合并17 5

实际上是对序列5 9 4 4进行普通的石子合并

同理4 4，8 5，9 13是对9 4 4 5的石子合并

七、区间DP

例题：环形石子合并

所以我们只要把序列写两遍，变成。

a1,a2,…,a[n],a[n+1],…,a[2n-1]

其中a[n+i]=a[i]

然后对整个区间做一次普通石子合并。

ans=min{f[1][n],f[2][n+1],…,f[n][2n-1]}

E N D