区间类动态规划

信智学院　陈琰宏

区间动态规划的特点

区间DP属于线性DP的一种，以区间长度作为DP的阶段，以区间的左右端点作为状态的维度。一个状态通常由被它包含且比它更小的区间状态转移而来。阶段（长度）、状态（左右端点）、决策三者按照由外到内的顺序构成三层循环。

求解：对整个问题设最优值，枚举合并点，将问题分解成为左右两个部分，最后将左右两个部分的最优值进行合并得到原问题的最优值。有点类似分治算法的解题思想。

典型试题：石子合并、整数划分，能量项链、凸多边形划分、多边形合并、等。

[2826] 石子合并

在一个操场上一排地摆放着N堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的2堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

计算出将N堆石子合并成一堆的最小得分。

输入格式

第一行为一个正整数N（2≤N≤100）；

以下N行，每行一个正整数，小于10000，分别表示第i堆石子的个数(1≤i≤N)。

输出格式

一个正整数，即最小得分。

7 13 7 8 16 21 4 18	4 1 3 5 2
239	22

贪心法

N=7　石子数分别为 13　7　8　16　21　4　18。

用贪心法的合并过程如下：

第一次 13 7 8 16 21 4 18 得分 15

第二次 13 15 16 21 4 18得分 22

第三次 13 15 16 21 22 得分28

第四次 28 16 21 22 得分37

第五次 28 37 22 得分59

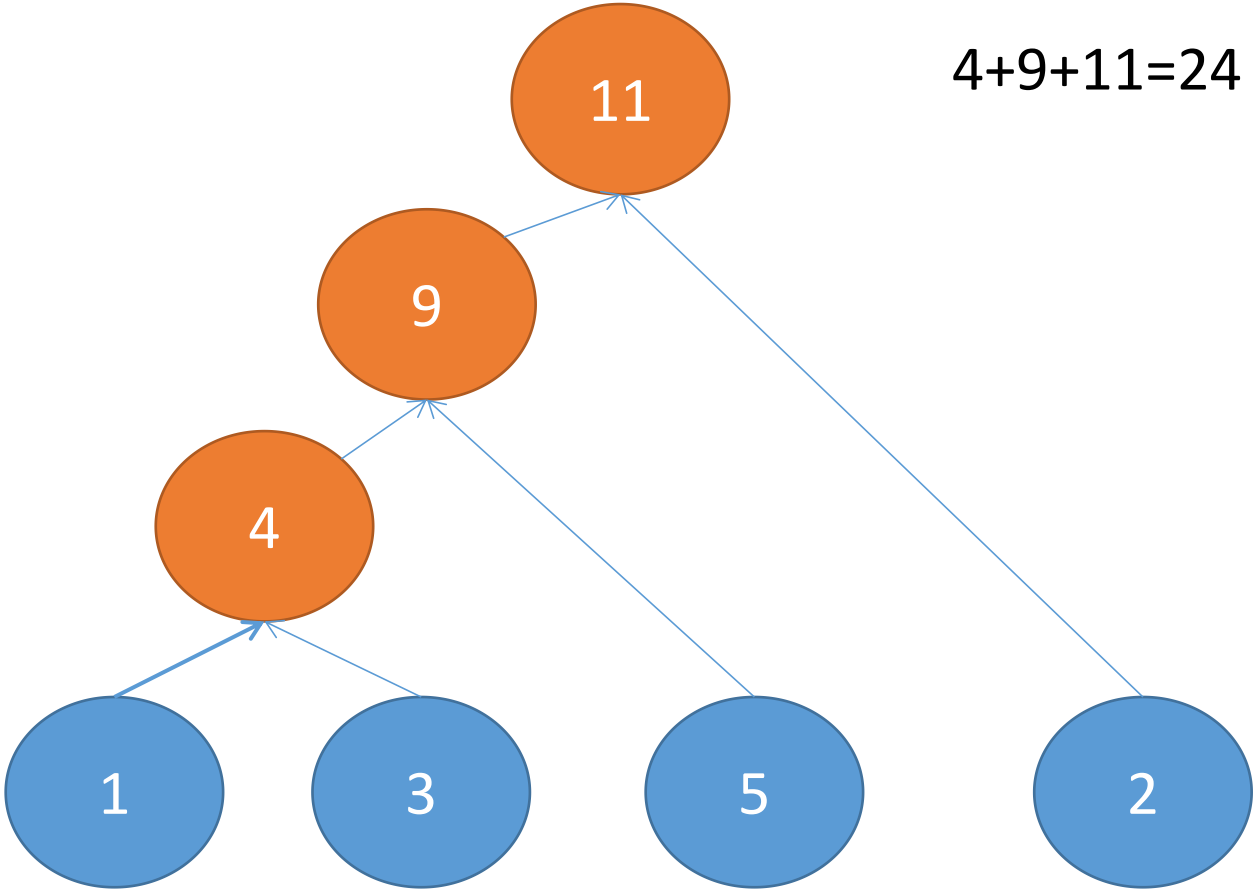
第六次 28 59 得分87

总分：15+22+28+37+59+87=248>239

显然，贪心法是错误的。

第一种合并方式

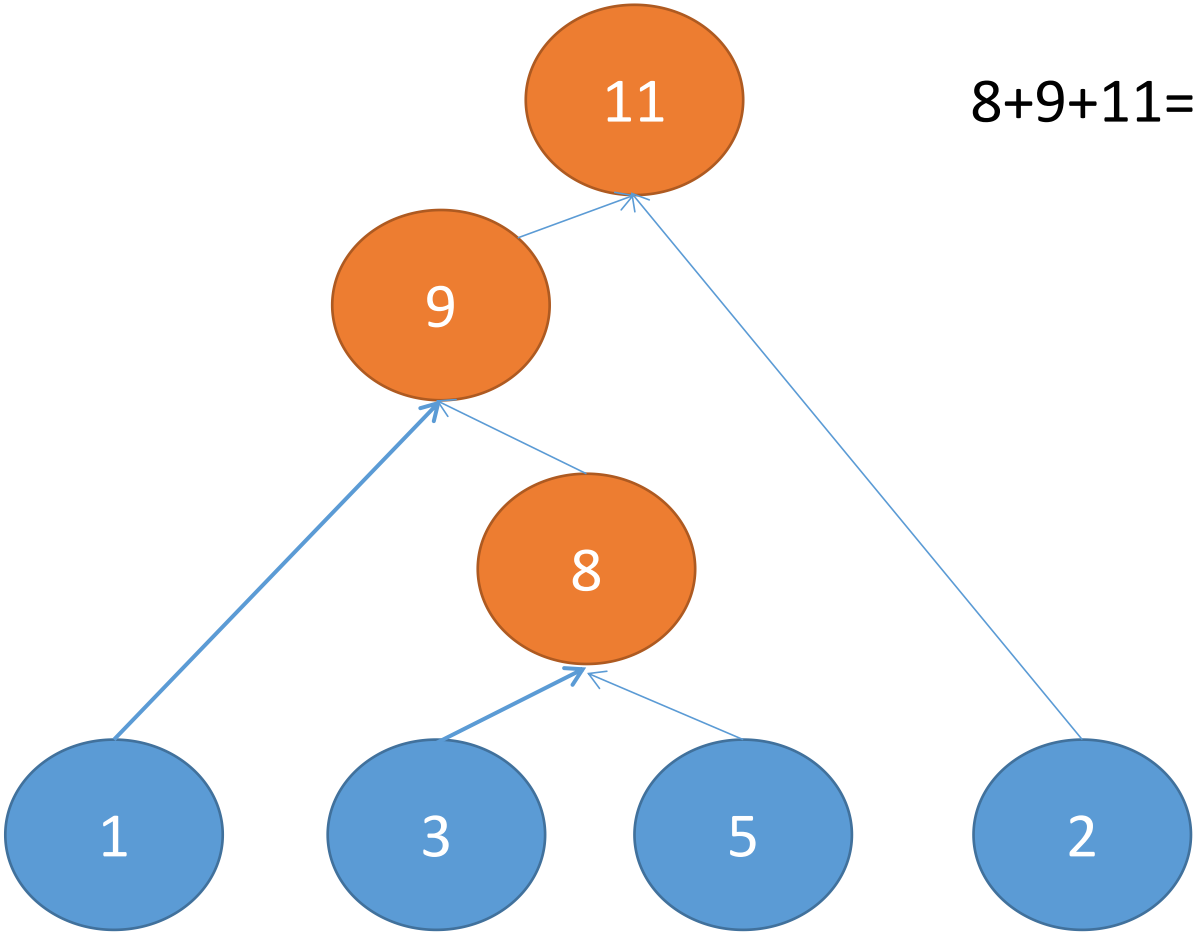
4
1 3 5 2
22



4+9+11=24

第二种合并方式

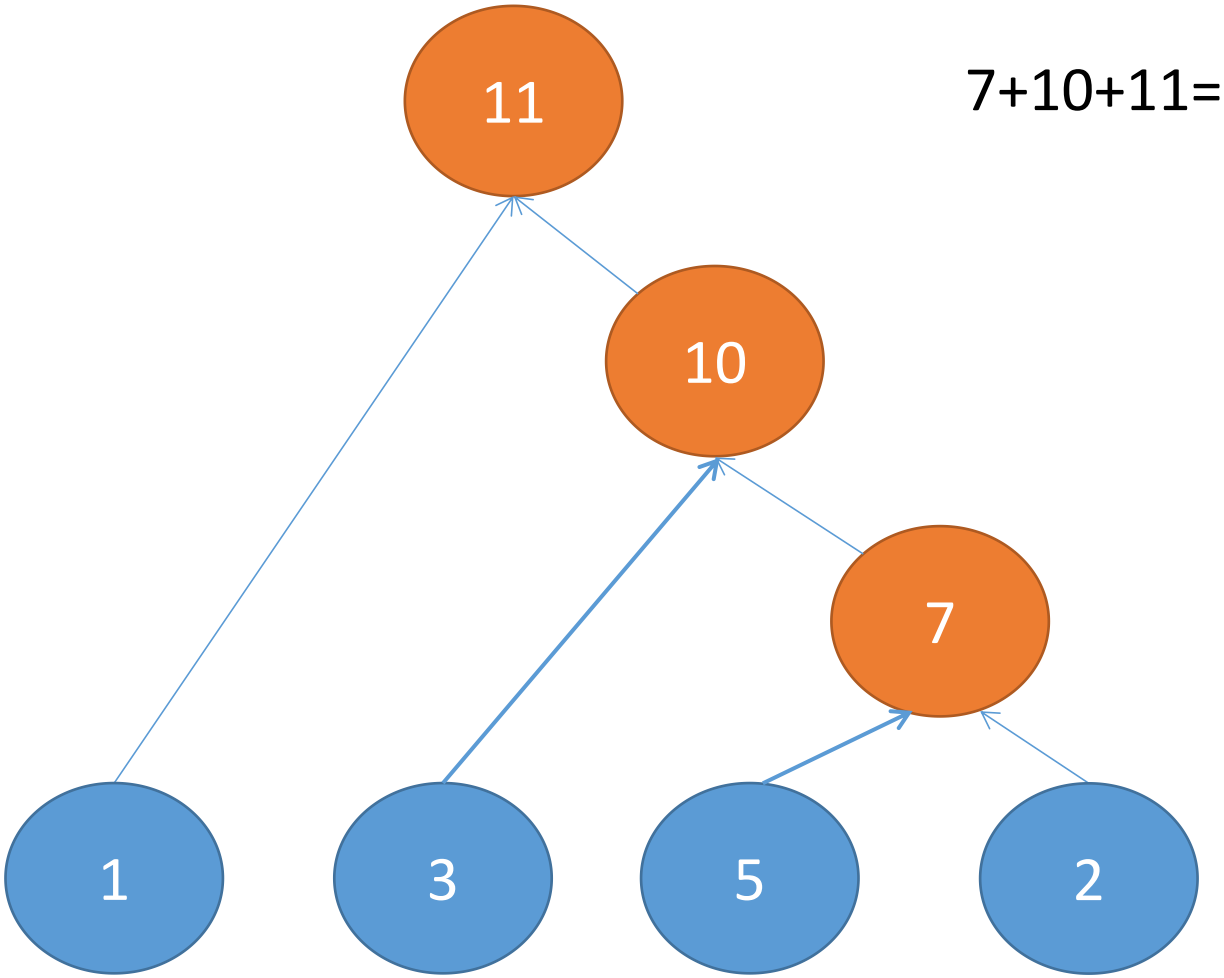
4
1 3 5 2
22



8+9+11=28

第三种合并方式

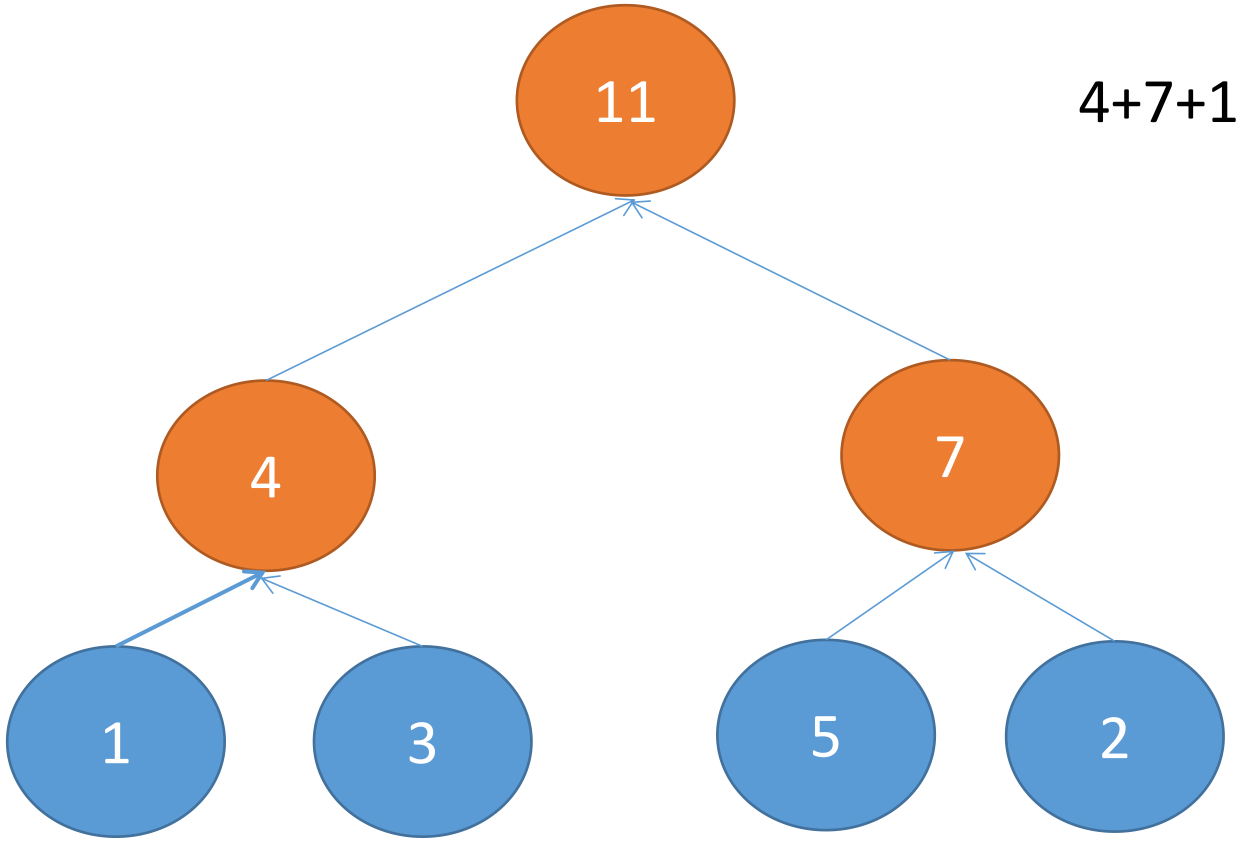
4
1 3 5 2
22



7+10+11=28

第四种合并方式

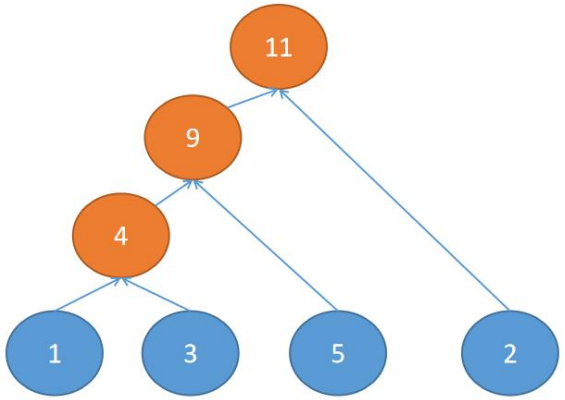
4
1 3 5 2
22

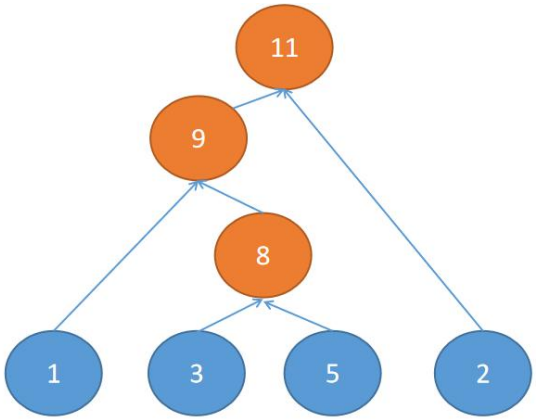


4+7+11=22

分析

将连续的两个点合并在一起，枚举所有情况，选出一个最大的



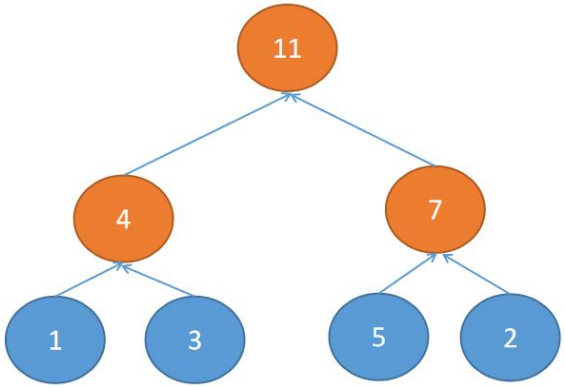


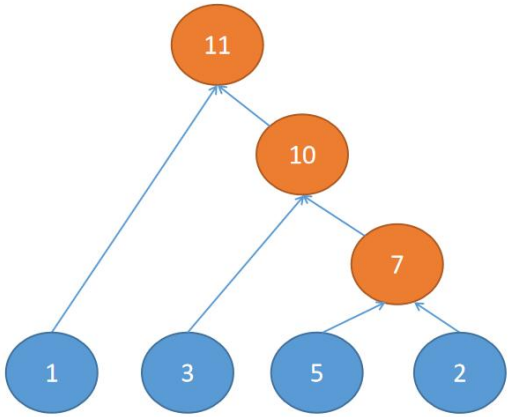
4+9+11=24

8+9+11=28

4+7+11=22

7+10+11=28

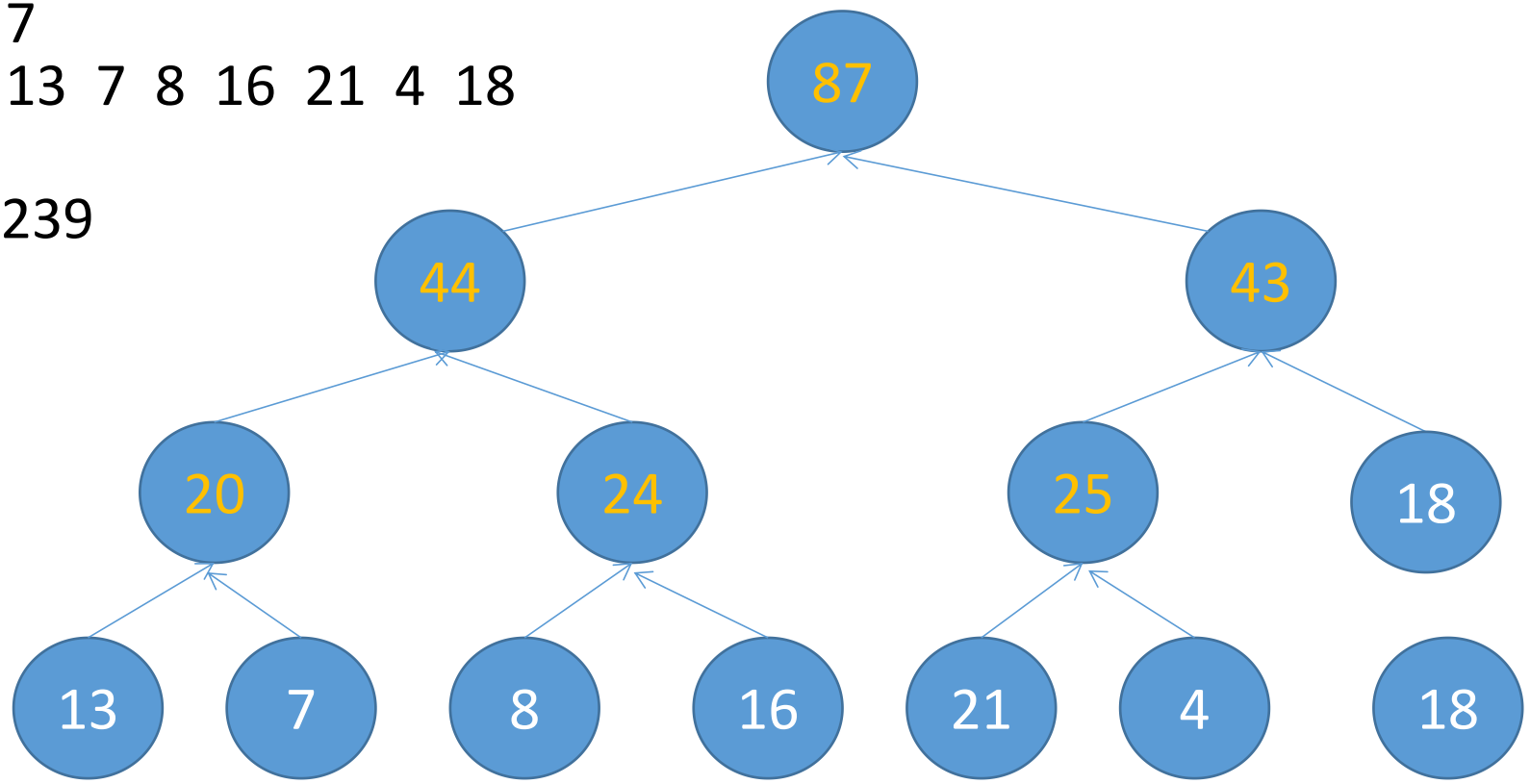




分析：如何枚举?

7
13 7 8 16 21 4 18

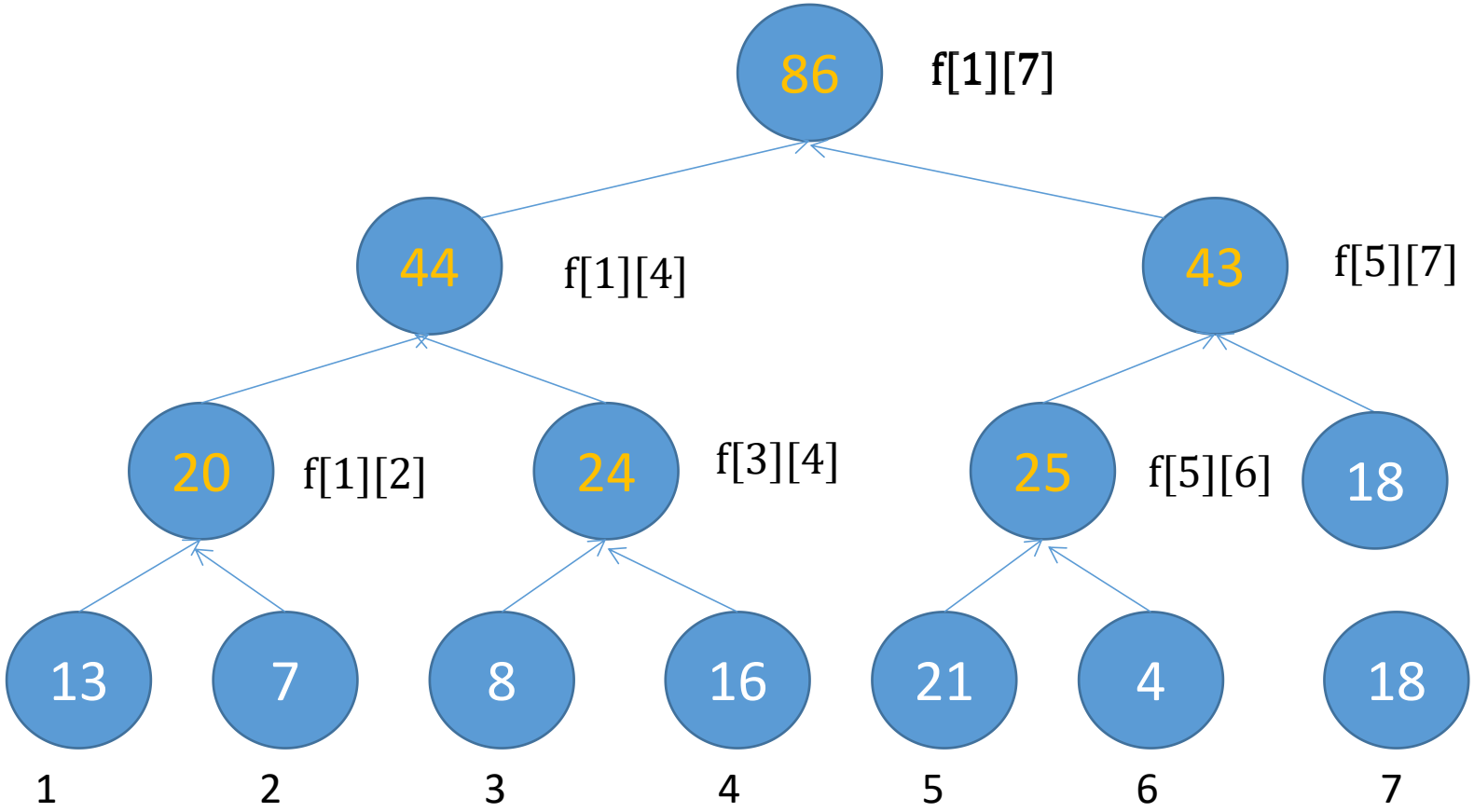
239



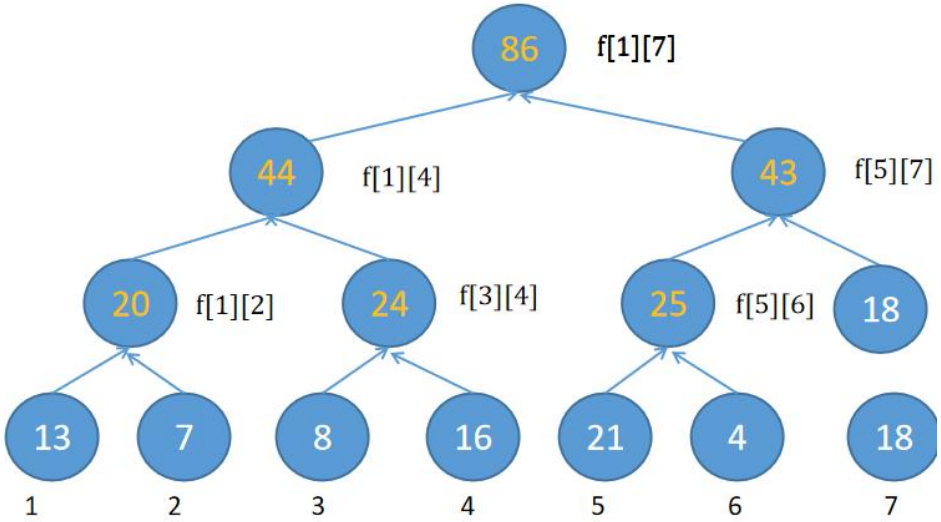
20+24+25+44+43+87=243

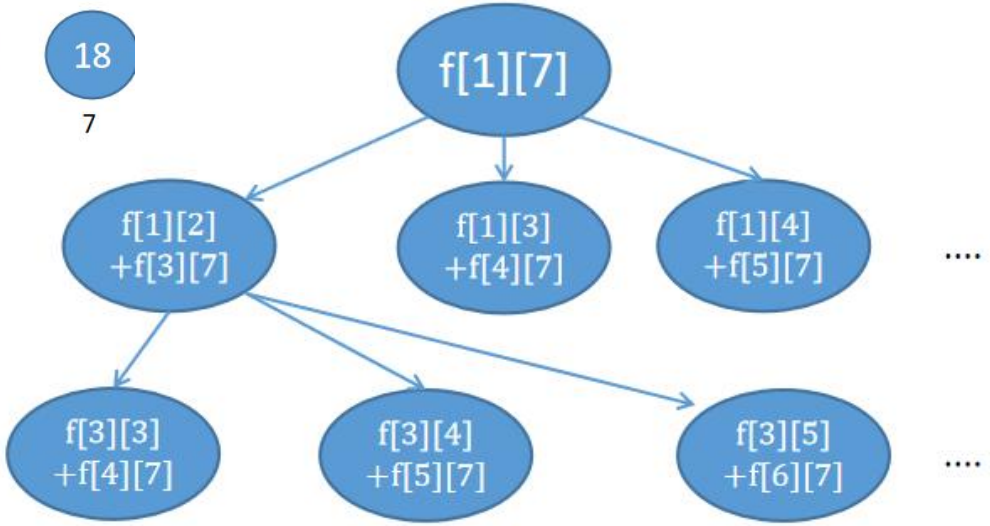
分析：状态描述

f[i][j] 表示将 i 到 j 合并成一堆的方案的得分



分析：状态描述



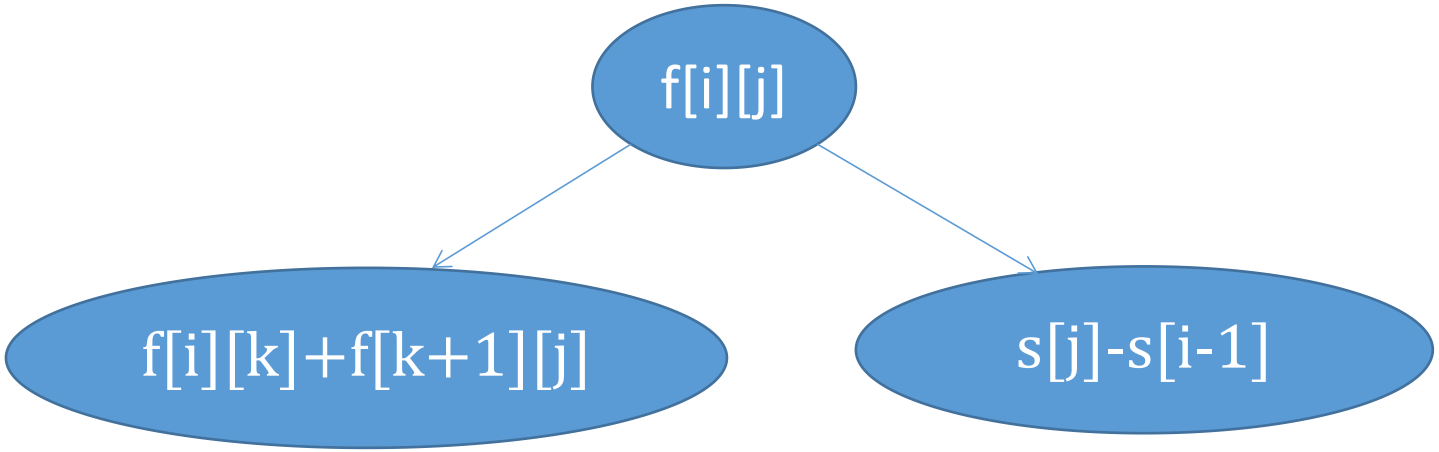


$$f[1][7]=f[1][2]+f[3][7]$$
$$f[1][7]=f[1][3]+f[4][7]$$
$$f[1][7]=f[1][4]+f[5][7]$$
$$\ldots \quad \ldots \quad \ldots$$

$$f[i][j] = f[i][k] + f[k+1][j]$$

分析：状态转移方程

$$f[i][j] = \min_{i \le k \le j-1} (f[i][k] + f[k+1][j] + s[j] - s[i-1])$$



两堆合并的代价，即从i到j的和

$s[j]$表示前缀和

分析：总结

核心：最后一次合并一定是左边连续的一部分和右边连续的一部分进行合并

状态表示：f[i][j] 表示将 i 到 j 合并成一堆的方案的集合，属性 Min

(1) $i < j$ 时，$f[i][j] = \min_{i \le k \le j-1} f[i][k] + f[k+1][j] + s[j] - s[i-1]$

(2) $i = j$ 时，$f[i][i] = 0$ （合并一堆石子代价为 0）

核心代码

算法实现：

阶段（区间长度）

状态（区间起点和终点）

决策（状态转移方程）

核心代码

所有的区间dp问题，第一维都是枚举区间长度，一般 len = 1 用来初始化，枚举从 len = 2 开始，第二维枚举起点 i （右端点 j 自动获得，j = i + len - 1）

```
for (int i = 1; i <= n; i++) {
    dp[i][i] = 初始值
}
for (int len = 2; len <= n; len++)              //区间长度
    for (int i = 1; i + len - 1 <= n; i++) {    //枚举起点
        int j = i + len - 1;                    //区间终点
        for (int k = i; k < j; k++) {           //枚举分割点，构造状态转移方程
            dp[i][j] = max(dp[i][j], dp[i][k] + dp[k + 1][j] + w[i][j]);
        }
    }
```

源代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int n,a[101],f[101][101],s[101];
int main(){
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        s[i]=s[i-1]+a[i];
    }
    for(int len=2;len<=n;len++){
        for(int i=1;i+len-1<=n;i++){
            int j=i+len-1;
            f[i][j]=0x3f3f3f;
            for(int k=i;k<j;k++){
                f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]+s[j]-s[i-1]);
            }
        }
    }
    cout<<f[1][n]<<endl;
}
```

记忆化搜索写法：

```
9  // 记忆化搜索：dp的记忆化递归实现
10 int dp(int i, int j) {
11     if (i == j) return 0; // 判断边界
12         //一个数，不需要合并
13     if (f[i][j] != -1) return f[i][j];
14
15     f[i][j] = 1e8;
16     for (int k = i; k <= j - 1; k ++)
17         f[i][j] = min(f[i][j], dp(i,k)+dp(k+1, j)+s[j]-s[i-1]);
18
19     return f[i][j];
20 }
```

记忆化搜索写法：

```
4   using namespace std;
5   const int N = 307;
6   int a[N], s[N];
7   int f[N][N];
8
9   // 记忆化搜索：dp的记忆化递归实现
10  int dp(int i, int j) {
21
22  int main() {
23      int n;
24      cin >> n;
25      for (int i = 1; i <= n; i ++) {
26          cin >> a[i];
27          s[i] += s[i - 1] + a[i];
28      }
29      memset(f, -1, sizeof f);
30      cout << dp(1, n) << endl;
31      return 0;
32  }
```

[2827] 乘积最大

设有一个长度为N的数字串，要求选手使用K个乘号将它分成K+1个部分，找出一种分法，使得这K+1个部分的乘积最大。

同时，为了帮助选手能够正确理解题意，主持人还举了如下的一个例子：

有一个数字串：312， 当N=3，K=1时会有以下两种分法：

1）3*12=36

2）31*2=62

这时，符合题目要求的结果是：31*2=62。

现在，请你帮助你的好朋友XZ设计一个程序，求得正确的答案。

分析：

本道题是一道区间DP问题，而且对阶段数有限制 k。可以将其按照插入乘号数来进行划分。如果插入k个乘号，可以把问题看做k个阶段的决策问题。

1、然要用一个数组用来记录组成结果的值，在合并石子中是前i堆石子的个数，在本道题中，是数字的大小，也就是用a[j][i]表示从第j位到第i位所组成的自然数 。

2、用f[j][k] 存储阶段k的每一个状态，表示前 j 位的数字用 k 个乘号能够得到的最大的结果（划分成小问题）

3、边界条件，f[j][0] 当k=0 时，表示没有乘号，则 f[j][0] 就是前 j 位组成的数字 f[j][0] = a[1][j]。

分析：

注意：还是阶段、状态、决策三层循环。

在考虑加一个乘号的时候要考虑这个乘号应该放在当前面有k-1的乘号时，后面的哪个位置比较好。也就是说要确定前j位用k个乘号f[i][k]（j>=k+1）的结果，应该从 有k-1个乘号的各种方案(前几位中已经包括了k-1个乘号)中选择哪种方案将第k个乘号加在后面之后结果最大。

要明白有k个乘号时，数字至少要有k+1位。有k-1个乘号时，数字至少要有k位。

状态转移方程：

$$f[i][k] = \max(f[i][k], f[j][k-1] * a[j+1][i])（j < i）$$

核心代码：

```
2   for(int len=1;len<=k;len++)
3   {
4       for(int i=len+1;i<=n;i++)//k个乘号要有k+1位的数字
5       {
6           for(int j=len;j<i;j++)//当有k-1个乘号的时候，最少有k个数字
7               f[i][len]=max(f[i][len],f[j][len-1]*a[j+1][i]);
8       }
9   }
```

源代码：

```
4   #include<string.h>
5   #include<math.h>
6   using namespace std;
7   //仍然要用一个数组用来记录组成结果的值，在合并石子中是前i堆石子的个数
8   //在本道题中，是数字的大小，也就是用a[j][i]表示从第j位到第i位所组成的自然数
9   //k个阶段的决策问题
10
11  int main()
12  {
13      int n,k;
14      cin>>n>>k;
15      long long int a[10][10];//表示从第j位到第i位的数值大小
16      char s[11];
17      for(int i=1;i<=n;i++)
18          cin>>s[i];
19      for(int i=1;i<=n;i++)//先将自己一个数的时候初始化一下
20          a[i][i]=s[i]-48;
21      for(int j=1;j<n;j++)
22      {
23          for(int i=j+1;i<=n;i++)
24              a[j][i]=a[j][i-1]*10+s[i]-48;
25      }
```

源代码：

```
27      long long int f[11][7];//f[i][k]表示在前i位数中插入k个乘号所得到的的最大值
28      for(int i=0;i<=10;i++)//初始化 因为是乘积 所以先初始化为1
29          for(int j=0;j<=6;j++)
30              f[i][j]=1;
31      for(int j=1;j<=n;j++)//初始化
32          f[j][0]=a[1][j];//边界条件  当没有乘号的时候，结果值就是数值
33
34      //过程
35      for(int len=1;len<=k;len++)
36      {
37          for(int i=len+1;i<=n;i++)//k个乘号要有k+1位的数字
38          {
39              for(int j=len;j<i;j++)//当有k-1个乘号的时候，最少有k个数字
40                  f[i][len]=max(f[i][len],f[j][len-1]*a[j+1][i]);
41          }
42      }
43      cout<<f[n][k]<<endl;
```

[3694] 环形石子合并

在一个圆形操场的四周摆放N堆石子，现要将石子有次序地合并成一堆. 规定每次只能选相邻的2堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分.

试设计出一种算法，计算出将N堆石子合并成一堆的最小得分和最大得分。

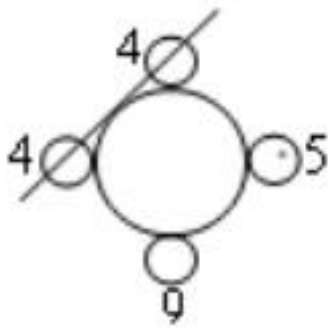
输入格式

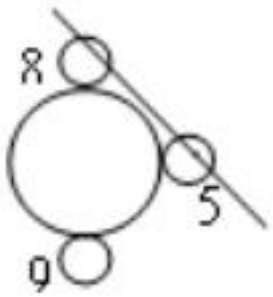
数据的第1行试正整数N，1≤N≤100，表示有N堆石子. 第2行有N个数，分别表示每堆石子的个数

输出格式

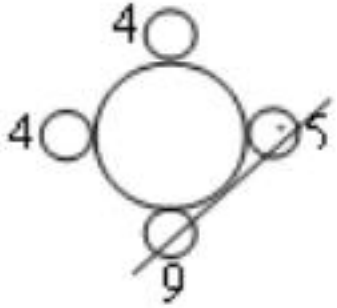
输出共2行，第1行为最小得分，第2行为最大得分

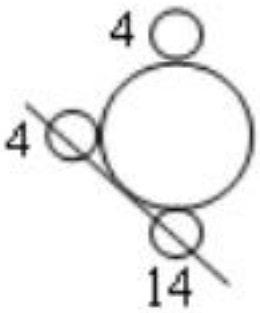
示例





总得分=8+13+22=43





总得分=14+18+22=54

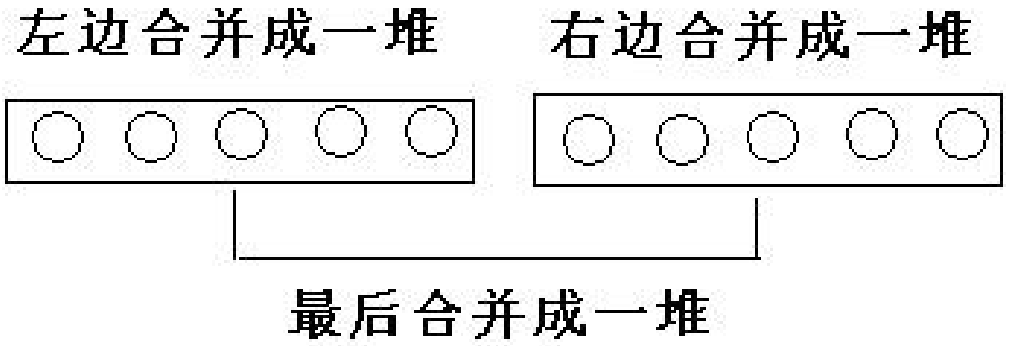
分析

假设只有2堆石子，显然只有1种合并方案

如果有3堆石子，则有2种合并方案，((1,2),3)和(1,(2,3))

如果有k堆石子呢？

不管怎么合并，总之最后总会归结为2堆，如果我们把最后两堆分开，左边和右边无论怎么合并，都必须满足最优合并方案，整个问题才能得到最优解。如下图：



状态转移方程

- 设t[i,j]表示从第i堆到第j堆石子数总和。
  Fmax(i,j)表示将从第i堆石子合并到第j堆石子的最大的得分
  Fmin(i,j)表示将从第i堆石子合并到第j堆石子的最小的得分

$$F\max(i,j)=\max_{i\le k\le j-1}\{F\max(i,k)+F\max(k+1,j)+t[i,j]\}$$

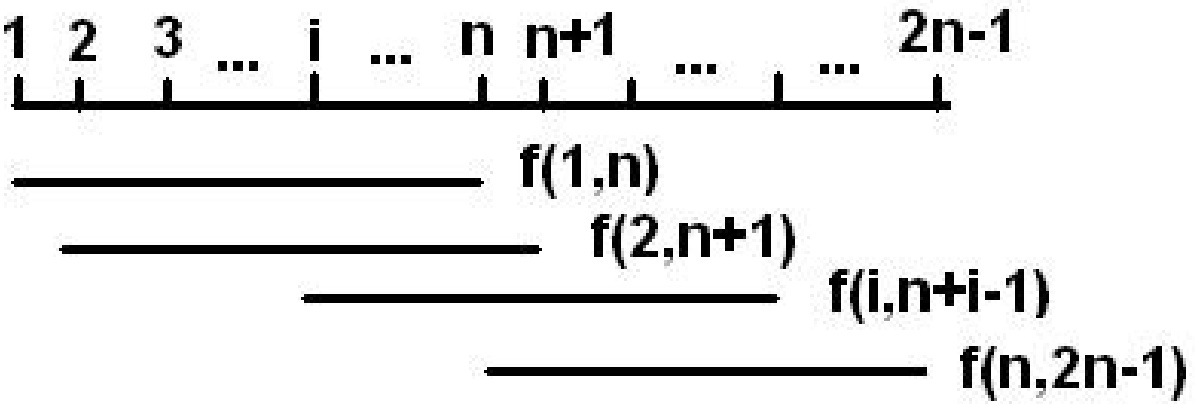
同理，

$$F\min(i,j)=\min_{i\le k\le j-1}\{F\min(i,k)+F\min(k+1,j)+t[i,j]\}$$

- Fmax[i,i] = 0, Fmin[i,i] = 0
- 时间复杂度为O(n^3)

优化

- 将这个圈转化为链，即将这条链延长2倍，扩展成2n-1堆，其中第1堆与n+1堆完全相同，第i堆与n+i堆完全相同，这样我们只要对这2n堆动态规划后，枚举f(1,n),f(2,n+1),…,f(n,2n-1)取最优值即可即可。
- 时间复杂度为$O(8n^3)$,如下图：



猜想

- 合并第i堆到第j堆石子的最优断开位置s[i,j]要么等于i+1，要么等于j-1，也就是说最优合并方案只可能是：

{ (i) (i+1… j) }

或者

{ (i… j-1) (j) }

```cpp
#include<bits/stdc++.h>
using namespace std;
int dp1[205][205],dp2[205][205],sum[205],a[205];
int main(){
    int n;
    cin>>n;
    sum[0]=0;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        a[i+n]=a[i];//把环拆成长度为2n的链
    }
    for(int i = 1; i <= 2*n; i++)
        sum[i]=sum[i-1]+a[i];
    for(int len=2;len<=n;len++){//len表示合并区间的长度
```

```
14      for(int len=2;len<=n;len++){//len表示合并区间的长度
15          for(int i=1;i+len-1<=2*n;i++) {
16              int j=i+len-1;
17              dp1[i][j]=1e9+7;
18              dp2[i][j]=0;
19              for(int k=i;k<j;k++){//枚举
20                  dp1[i][j]=min(dp1[i][j],
21                                dp1[i][k]+dp1[k+1][j]+sum[j]-sum[i-1]);
22                  dp2[i][j]=max(dp2[i][j],
23                                dp2[i][k]+dp2[k+1][j]+sum[j]-sum[i-1]);
24              }
25          }
26      }
27      int ans1=1e9+7;
28      int ans2=0;
29      for(int i = 1; i <= n; i++){
30          ans1=min(ans1,dp1[i][i+n-1]);
31          ans2=max(ans2,dp2[i][i+n-1]);
32      }
33      cout<<ans1<<"\n"<<ans2<<endl;
```

[3695] 能量项链

在Mars星球上，每个Mars人都随身佩带着一串能量项链。在项链上有N颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。因为只有这样，通过吸盘（吸盘是Mars人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为m，尾标记为r，后一颗能量珠的头标记为r，尾标记为n，则聚合后释放的能量为 m*r*n （Mars单位），新产生的珠子的头标记为m，尾标记为n。

[3695] 能量项链

需要时，Mars人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。

例如：设N=4，4颗珠子的头标记与尾标记依次为(2，3) (3，5) (5，10) (10，2)。我们用记号⊕表示两颗珠子的聚合操作，(j⊕k)表示第j，k两颗珠子聚合后所释放的能量。则第4、1两颗珠子聚合后释放的能量为：

(4⊕1)=10*2*3=60。

这一串项链可以得到最优值的一个聚合顺序所释放的总能量为

((4⊕1)⊕2)⊕3）=10*2*3+10*3*5+10*5*10=710。

[3695] 能量项链

输入文件的第一行是一个正整数N（4≤N≤100），表示项链上珠子的个数。第二行是N个用空格隔开的正整数，所有的数均不超过1000。第i个数为第i颗珠子的头标记（1≤i≤N），当1≤i＜N时，第i颗珠子的尾标记应该等于第i+1颗珠子的头标记。第N颗珠子的尾标记应该等于第1颗珠子的头标记。

至于珠子的顺序，你可以这样确定：将项链放到桌面上，不要出现交叉，随意指定第一颗珠子，然后按顺时针方向确定其他珠子的顺序。

输出文件只有一行，是一个正整数E（E≤2.1*109），为一个最优聚合顺序所释放的总能量。

[3695] 能量项链

输入
4
2 3 5 10

输出：
710

[3695] 能量项链

给定N个能量石，每个能量石有一个二元属性$(W_{i,1}, W_{i,2})$

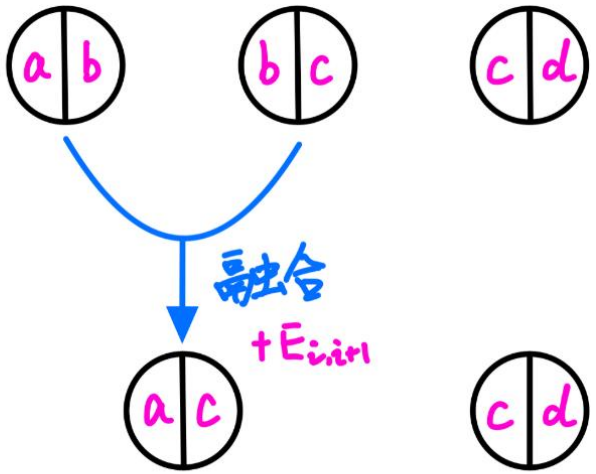
其中$W_{i,1}$表示第i个能量石和第i-1个能量石融合产生的能量的其中一个参数当然；

$W_{i,2}$表示第i个能量石和第i+1个能量石融合产生的能量的其中一个参数魔法石是顺序且环形摆放的，每次可以融合相邻两个魔法石

融合两个能量石i, i+1所产生的能量为

$$E_{i,i+1} = w_{i,1} \times (w_{i,2}或1$$

融合后左侧魔法石的第一个合并成为一颗新的魔法石



求最终把所有石头融合成一个石头时，产生的最大能量值

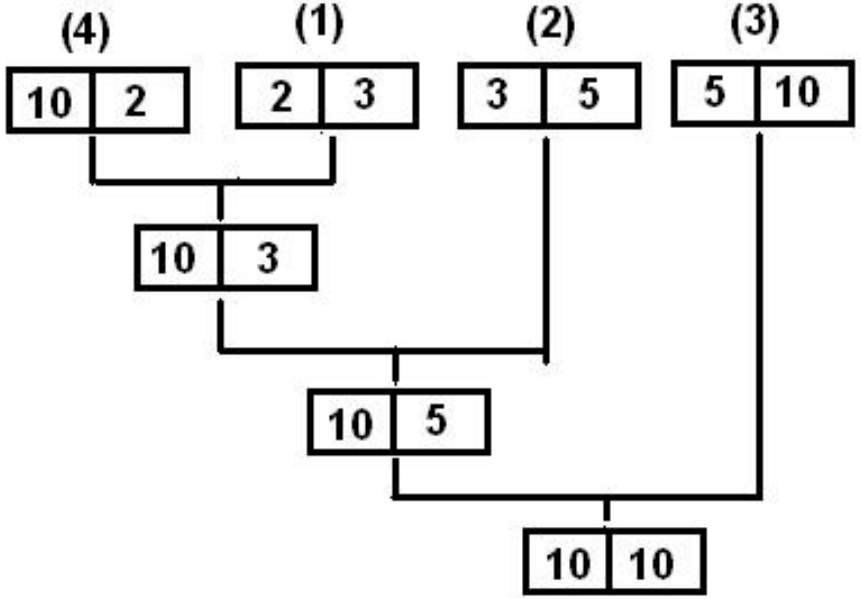
- 分析样例：

  N=4，4颗珠子的头标记与尾标记依次为

  (2，3) (3，5) (5，10) (10，2)。

- 我们用记号⊕表示两颗珠子的聚合操作，释放总能量：

  ((4⊕1)⊕2)⊕3）=10*2*3+10*3*5+10*5*10=710



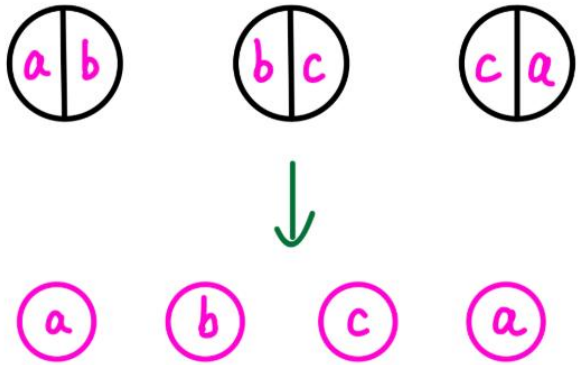
分析

本题可以把区间长度作为搜索的阶段来进行记忆话搜索，因此我们也可以采用区间DP 的方式来处理，和 环形石子合并 十分相似，但又不尽相同。

在 环形石子 中，每个石头只有单一的参数，而本题有两个参数，也就意味着我们需要在细节上做出改变

经过观察我们发现，合并两个石头 (a,b) (b,c)的操作就像是矩阵乘法一样，合并完后就变成了(a,c).

因此我们可以离散的来存储每个参数，具体如下所示：

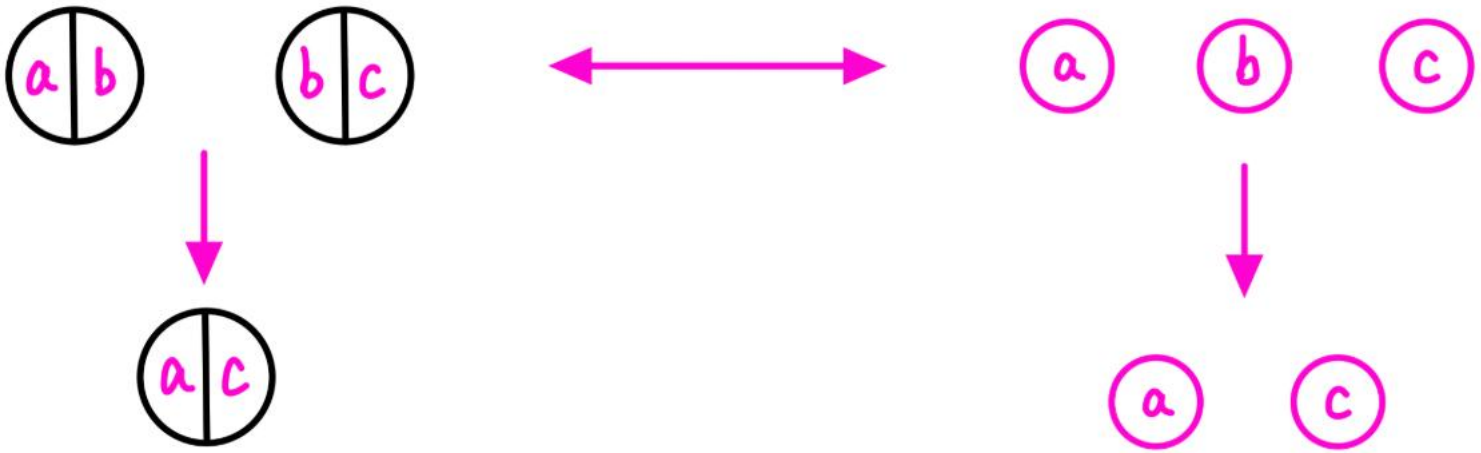


[3695] 能量项链

这样 状态表示 就更新为：当前合并的石子堆的左端石头的左参数是 l，右端石头的右参数是 r 的方案

这样对应的 初始状态 本来应该是一个有着 二元属性 的石头，现在就变成了长度为 2的区间

这样合并区间后，需要记录的新石头的参数也刚好是 区间的两端 对应的参数，如下图所示：



[3695] 能量项链

而且这里我们的转移方程也要修改为 $f_{l,r} = max(f_{l,k} + f_{k,r} + E_{l,r})$

以往的 **区间DP** 我们是把区间 $[a, b]$ 拆分为 $[a, k]$和$[k + 1, b]$

因为 **同一个石子** 只会被合并到 **一个石子堆** 里

但本题合并魔法石时，分割点 k 要被分到 **左侧石子堆的右端点** 和 **右侧石子堆的左端点** 中

因此，参数 k 要作为两个区间的共同端点来使用，即 $[a, k]$ 和 $[k, b]$

此外我们原来只需要合并 n 个石头，这样转换后就要合并 $n + 1$ 个石头了

[3695] 能量项链

状态表示—集合$f_{l,r}$：当前合并的石子堆的左端石头的左参数是 l，右端石头的右参数是 r 的方案

状态表示—属性$f_{l,r}$：方案的费用最大

状态计算—$f_{l,r}$：

$$f_{l,r} = max(f_{l,k} + f_{k,r} + E_{l,r}) \quad (l < k < r)$$

初始状态： $f_{l,l+1} = 0 \quad (1 \le l \le n)$

目标状态： $f_{1,n+1}$

[3695] 能量项链

```
3  int n,w[210],dp[210][210];
4  int main()
5  {
6      int n;
7      cin>>n;
8      for(int i=1;i<=n;i++){
9          cin>>w[i];
10         w[i+n]=w[i];//huan->lian
11     }
12
13     for(int len=3;len<=n+1;len++)
14     for(int l=1,r;l+len-1<=2*n;l++){
15         r=l+len-1;
16         for(int k=l+1;k<r;k++)    //3
17             dp[l][r]=max(dp[l][r],dp[l][k]+dp[k][r]+w[l]*w[k]*w[r]);//2
18     }
19
20     int ans=0;
21     for(int l=1;l<=n;l++)
22         ans=max(ans,dp[l][l+n]);//1
23     cout<<ans;
24     return 0;
```

[3695] 能量项链

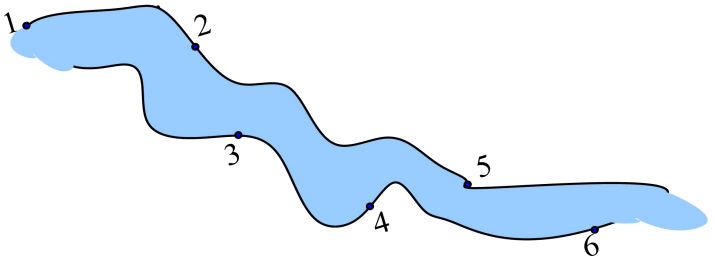
[3695] 能量项链

[3695] 能量项链

[3695] 能量项链

[3889] 长江一日游——游艇租赁

长江游艇俱乐部在长江上设置了n个游艇出租站，游客可以在这些出租站租用游艇，在下游的任何一个游艇出租站归还游艇。游艇出租站i到游艇出租站j之间的租金为$r(i,j)$。现在要求出从游艇出租站1到游艇出租站n所需的最少的租金。



当要租用游艇从一个站到另外一个站时，中间可能经过很多站点，不同的停靠站策略就有不同的租金。

如果穷举所有的停靠策略，例如一共有10个站点，当求子问题4个站点的停靠策略时，子问题有

（1，2，3，4），（2，3，4，5），（3，4，5，6）

（4，5，6，7），（5，6，7，8），（6，7，8，9）

（7，8，9，10）。

如果再继续求解子问题，会发现有大量的子问题重叠，其算法时间复杂度为2^n，暴力穷举的办法是不可取的。

分析

（1）确定状态。 dp[i][j]表示第i个站点到第j个站点的最少租金。

（2）划分阶段。 区间长度。

（3）决策选择。 原问题与子问题之间的关系。

i，i+1，…，k　　k，k+1，…，j

状态转移方程：$dp[i][j]=\min_{i<k<j}\{dp[i][k]+dp[k][j],dp[i][j]\}$

（4）边界条件。$i<j$，dp[i][j]=r[i][j]。

（5）求解目标。dp[1][n]。

边界条件：$i<j$，$dp[i][j]=r[i][j]$。

状态转移方程：

$$dp[i][j]=\min_{i<k<j}\{dp[i][k]+dp[k][j],dp[i][j]\}$$

r[ ][ ]	1	2	3	4	5	6
1		2	6	9	15	20
2			3	5	11	18
3				3	6	12
4					5	8
5						6
6						

dp[ ][ ]	1	2	3	4	5	6
1		2	6	9	15	20
2			3	5	11	18
3				3	6	12
4					5	8
5						6
6						

求解目标： dp[1][6]=15

dp[][]	1	2	3	4	5	6
		2	5	7	11	15
2			3	5	9	13
3				3	6	11
4					5	8
5						6
6						

算法实现：

阶段（区间长度）

状态（区间起点和终点）

决策（状态转移方程）

算法实现：

```
void rent(){
    for(int d=3;d<=n;d++){//区间长度d
        for(int i=1;i<=n-d+1;i++){//状态起点i，终点j
            int j=i+d-1;
            for(int k=i+1;k<j;k++){//枚举决策点k
                if(dp[i][j]>dp[i][k]+dp[k][j])
                    dp[i][j]=dp[i][k]+dp[k][j];
            }
        }
    }
}
```

算法分析

时间复杂度： 3层for循环中嵌套，执行次数为$O(n^3)$，时间复杂度为$O(n^3)$。

空间复杂度： 数组dp[n][n]，空间复杂度为$O(n^2)$。

延伸思考

上面得到的最优值只是第1个站点到第n个站点之间的最少租金，并不知道停靠了哪些站点，如果还想知道停靠的站点，怎么办？

状态转移方程：

$$\mathrm{dp}[i][j]=\min_{i<k<j}\{\mathrm{dp}[i][k]+\mathrm{dp}[k][j],\mathrm{dp}[i][j]\}$$

用辅助数组$s[i][j]$记录各个子问题（$i..j$）的最优决策k（停靠站点）。

构造最优解：

dp[][]

	1	2	3	4	5	6
		2	5	7	11	15
2			3	5	9	13
3				3	6	11
4					5	8
5						6
6						

s[ ][ ]

	1	2	3	4	5	6
1		0	2	2	2	2
2			0	0	3	4
3				0	0	4
4					0	0
5						0
6						

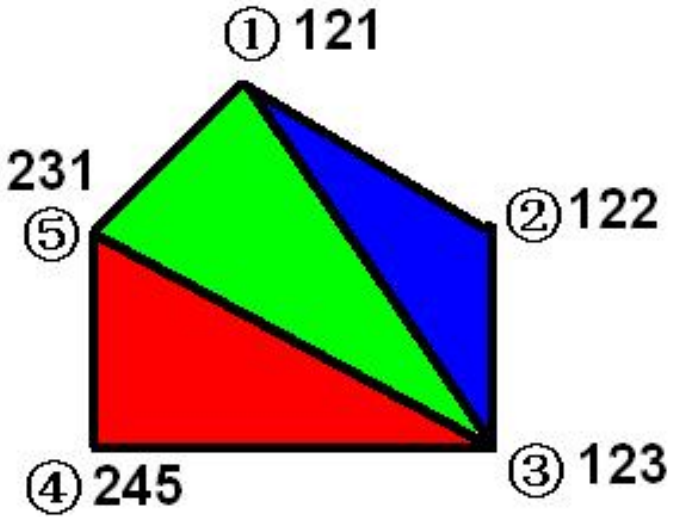
答案：1——2——4——6。

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n,t;
    int m[200][200] = { 0 };
    cin>>t;
    for(int p=1;p<=t;p++){
```

```cpp
    for(int p=1;p<=t;p++){
        cin >> n;
        memset(m,0,sizeof(m));
        for (int i = 1; i <= n - 1; i++)
        {//初始化，将所有r(i,j)都先存在数组m中
            for (int j = i + 1; j <= n; j++)
                cin >> m[i][j];
        }
        for (int r = 2; r <= n; r++)
        {//接着从长度为 2 的开始找较优解（比如说从 1 到 2 就是长度为 2的）
            //直到找到长度为 n 的
            for (int i = 1; i <= n - r + 1; i++)
            {
                int j = i + r - 1;// r(i, j)的长度为 r
                for (int k = i; k <= j; k++)
                {//在 i 到 j 中找某一站 k，使得r(i,k)+r(k,j)最小
                    int t = m[i][k] + m[k][j];
                    if (t < m[i][j])
                        m[i][j] = t;//用较优解替换原来的r(i,j)
                }
            }
        }
        cout << m[1][n] << endl;
```

[3696] 凸多边形的三角剖分

- 给定由N顶点组成的凸多边形
- 每个顶点具有权值
- 将凸N边形剖分成N-2个三角形
- 求N-2个三角形顶点权值乘积之和最小？



上述凸五边形分成△123 ， △135， △345

三角形顶点权值乘积之和为：

121*122*123+121*123*231+123*245*231= 12214884

分析

- 性质：一个凸多边形剖分一个三角形后，可以将凸多边形剖分成三个部分：
  - 一个三角形
  - 二个凸多边形（图2可以看成另一个凸多边形为0）

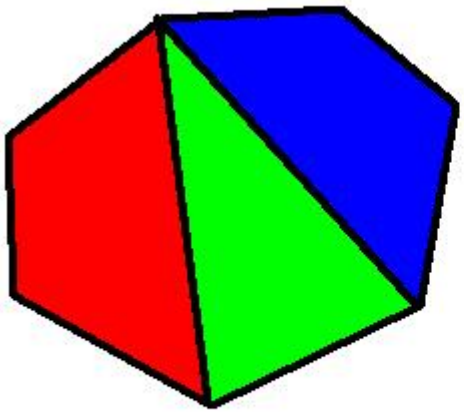


图1

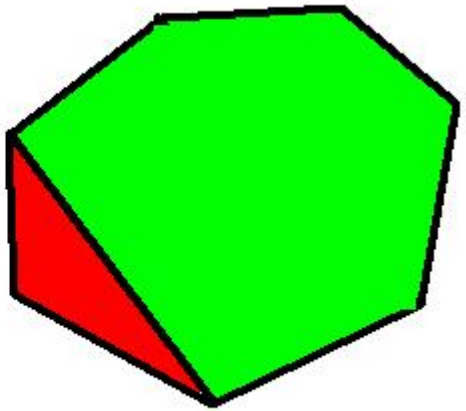


图2

动态规划

如果我们按顺时针将顶点编号，则可以相邻两个顶点描述一个凸多边形。

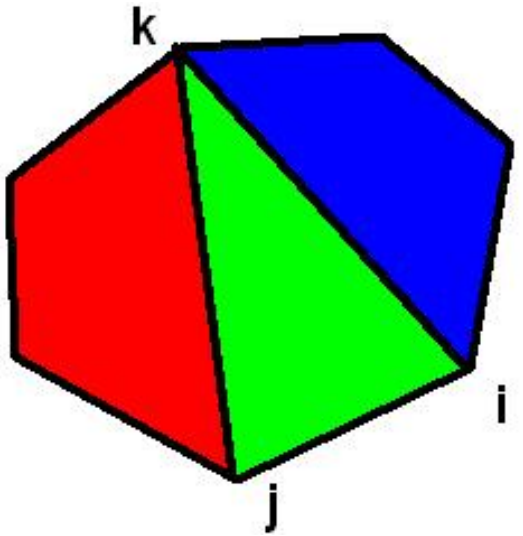
设f(i,j)表示i~j这一段连续顶点的多边形划分后最小乘积

枚举点k，i、j和k相连成基本三角形，并把原多边形划分成两个子多边形，则有

$f(i,j)=\min\{f(i,k)+f(k,j)+a[i]*a[j]$

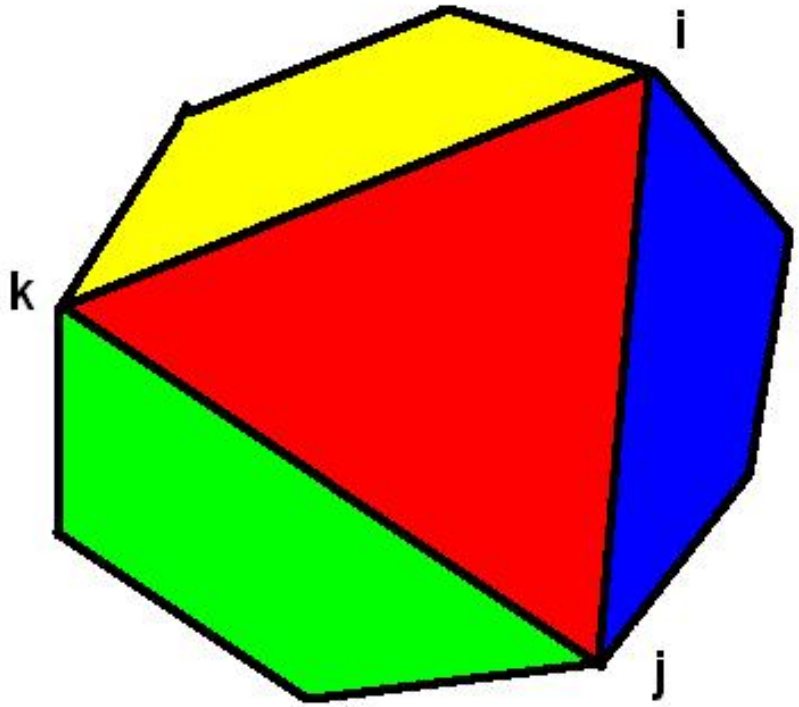
$1<=i<k<j<=n$

时间复杂度$O(n^3)$



讨论

- 为什么可以不考虑这种情况？



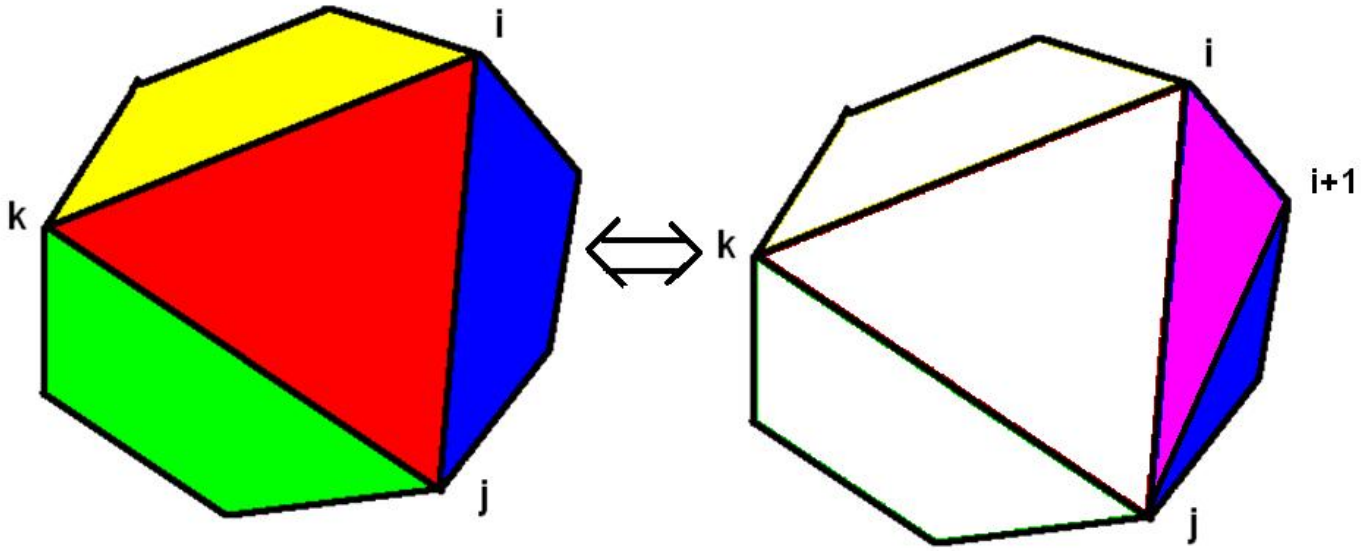


图1　　图2

- 可以看出图1和图2是等价的，也就是说如果存在图1的剖分方案，则可以转化成图2的剖分方案，因此可以不考虑图1的这种情形。