马上开始

35tang-C++竞赛系列四阶课程

《 35tang-C++竞赛系列四阶课程》

内容

区间动态规划

复习动态规划

区间动态规划

- 动态规划：从前面的状态（最优解）推导出后面的状态（最优解）。
- 区间动态规划：二维的动态规划。从小的区间推导出大的区间。比如计算出小的长度的区间的最优解，然后通过合并这些小的区间，得到大的区间的最优解。注意：不一定是合并，其实就是要找分割点，看大的区间从哪两个小的区间可以得到最优值。

```
for(int len=1;len<=n;len++)//枚举长度
{
   for(int i=1; i+len-1<=n;i++)//枚举区间左端点
   {
      //do something
     int j=i+len-1;//右端点
      for(int k=i;k<j;k++)
    //枚举左右区间的分割点
      {
         //get dp[i][j] from dp[i][k]  and dp[k+1][j]
      }
   }
}
```

例题：

石子合并

有n堆石子排成一排，每堆石子有一定的数量。现要将n堆石子并成为一堆。合并的过程只能每次将相邻的两堆石子堆成一堆，每次合并花费的代价为这两堆石子的数目的和，经过n-1次合并后成为一堆。求出总的代价最小值。

输入

测试数据第一行有一个整数n，表示有n堆石子。

接下来的一行有n（0< n <200）个数（每个数不超过100），分别表示这n堆石子的数目，用空格隔开

输出

输出总代价的最小值，占单独的一行

样例输入1

3

1 2 3

样例输出

9

样例输入2

7

13 7 8 16 21 4 18

样例输出2

239

验证理解—比划一下：示例数据显然是两个可能合并的顺序中找最小

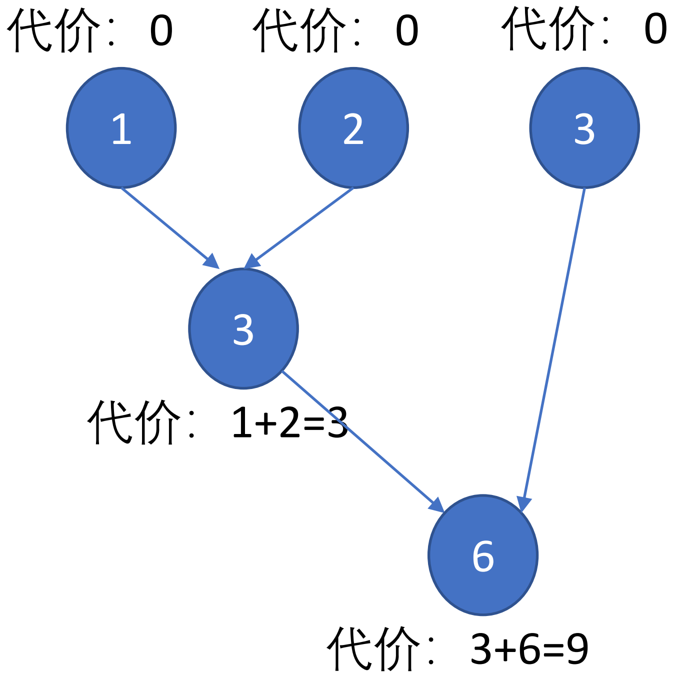
样例输入1

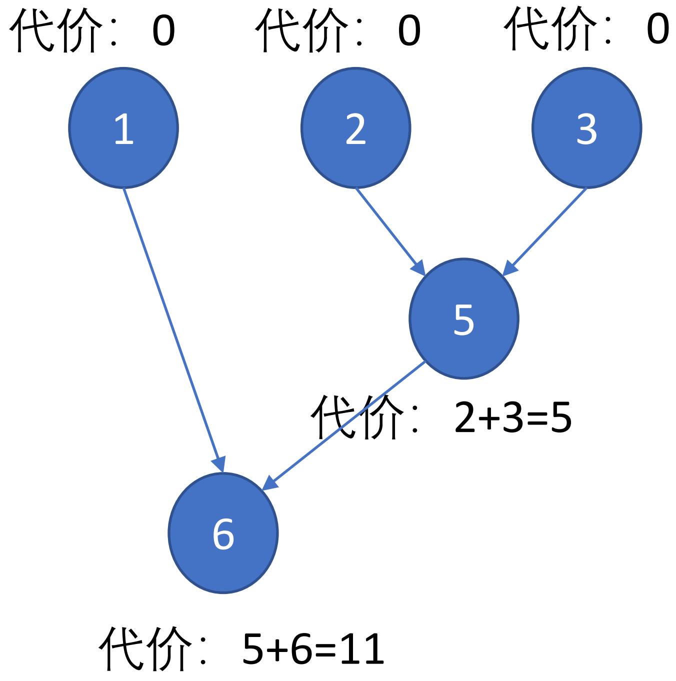
3

1 2 3

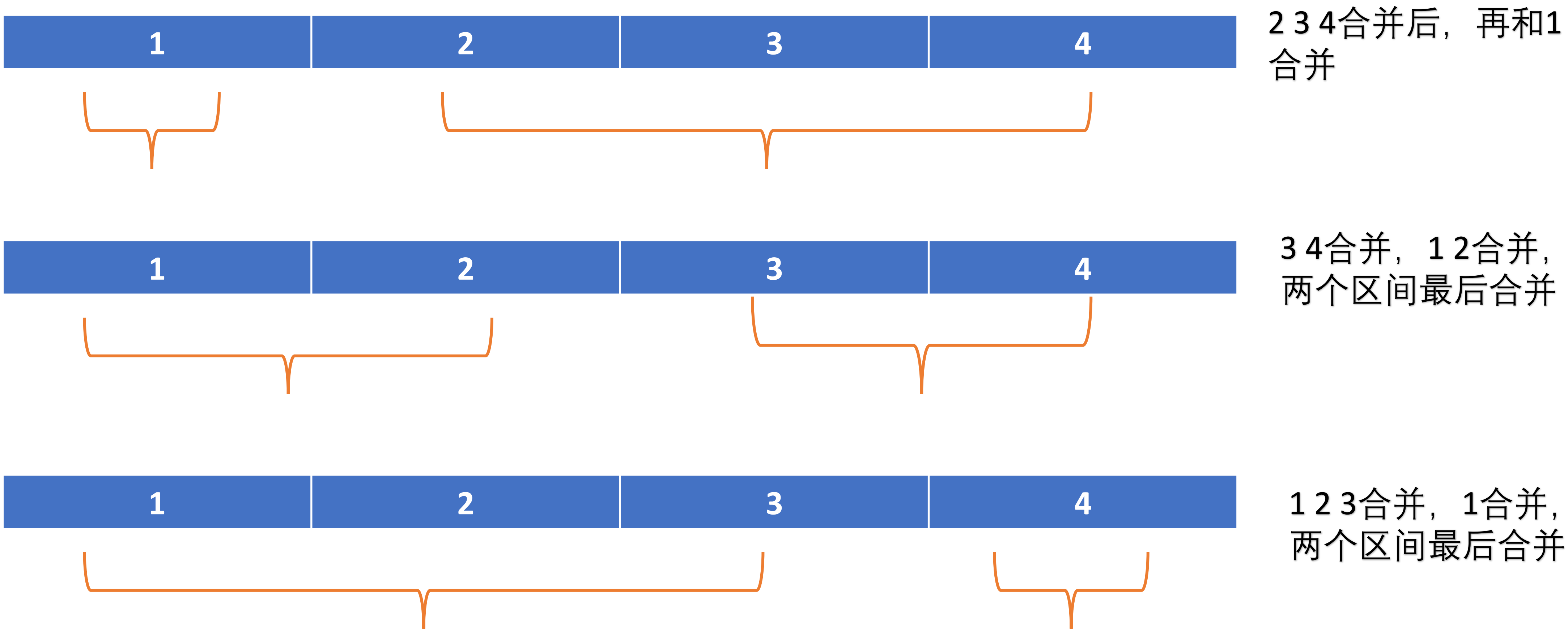
样例输出

9





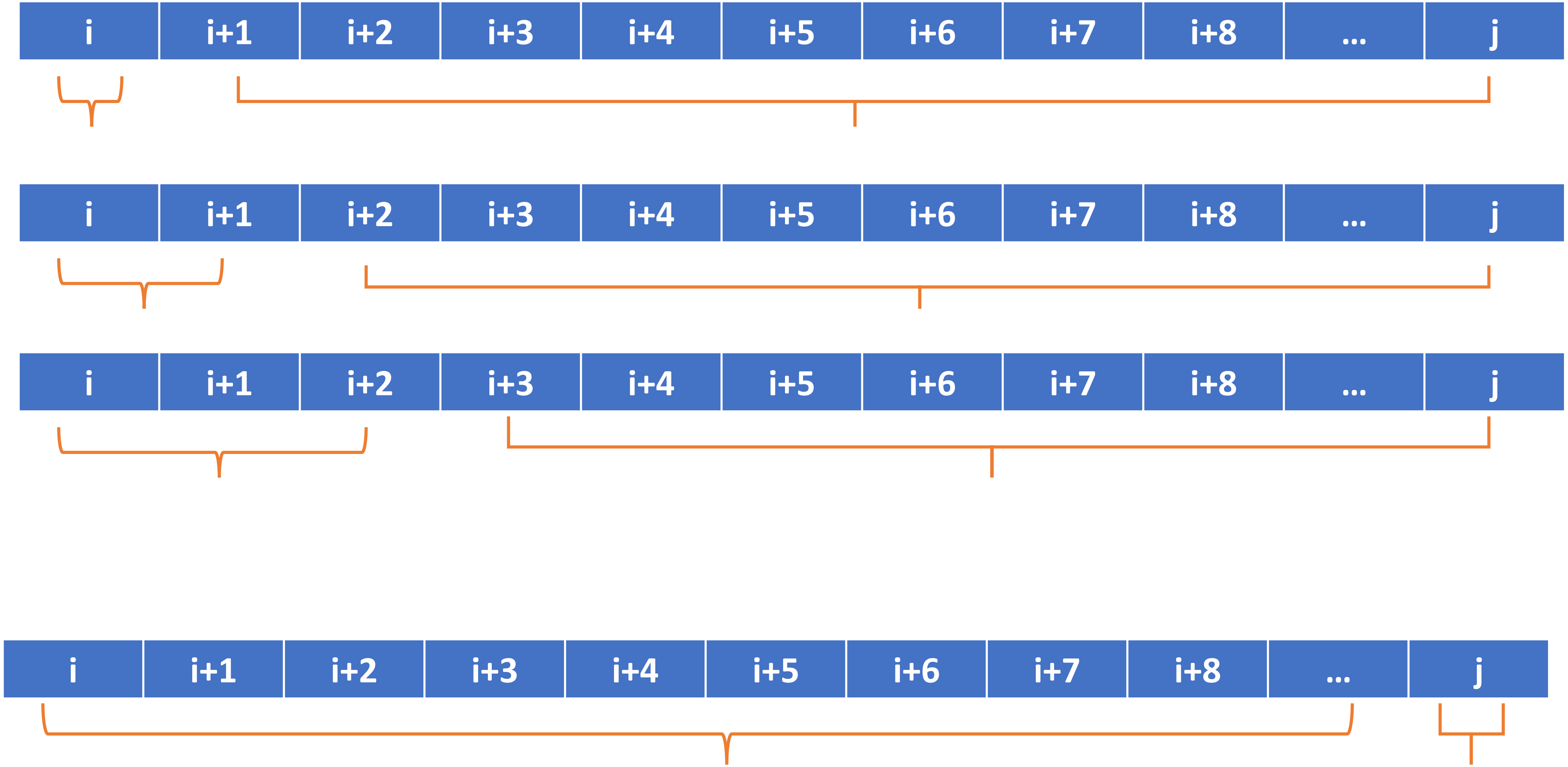
多几个石头看看，比如四堆石头分别1 2 3 4个，那么4个石头这个大的区间一定是两个小区间合并来的，这两个小区间有好几个可能，而最终1到4组成的区间的最小合并代价就是这若干个可能中最小的一种



扩展到i和j位置：打擂找最小

k为区间分割点，k从i到j-1

每一次分隔出两个小区间，[i,k]和[k+1,j]



k=i

k=i+1

k=i+2

k=...

k=j-1

公式

- 设置dp[i][j]表示区间i到j的石头都合并了的最小代价
- dp[i][j] = min{dp[i][j], dp[i][k] + dp[k + 1][j] + sum(i,j) };
  - 其中，k在i和j之间。sum(i,j)就是i到j之间所有的石头的总数目，可以遍历数组元素计算。注意k+1应该小于等于j。
  - 也就是说区间i到j之间合并的最小价值={i和j中间每一个位置作为分隔点分开的两个区间分别合并，然后两个区间再进行合并的代价}中最小的一个
- 用区间动态规划：当我们计算i到j区间的最小代价的时候，所有区间长度小于i到j这个区间长度的区间的最小代价都已经计算出来了，就可以完成推导了

一起套模板写

```
for(int len=1;len<=n;len++)//枚举长度
{
   for(int i=1; i+len-1<=n;i++)//枚举区间区间左端点
   {
       //do something
      int j=i+len-1;//右端点
       for(int k=i;k<j;k++)
     //枚举左右区间的分割点
       {
          //get dp[i][j] from dp[i][k]  and dp[k+1][j]
       }
   }
}
```

```
int main()
{
        cin>>n;
        for (int i=1;i<=n;i++)
        {
                cin>>a[i];
        }
        for(int len=2;len<=n;len++)//枚举长度，从2开始，长度1的区间都是0
        {
           for(int i=1; i+len-1<=n;i++)//枚举区间区间左端点
           {
              int j=i+len-1;//右端点
              dp[i][j]=INT_MAX;//为打擂初始化最大
              for(int k=i;k<j;k++)//看下面的公式，k+1需要小于等于j
              //枚举左右区间的分割点
              {
                 dp[i][j]=min(dp[i][j],dp[i][k] + dp[k + 1][j] + sum(i,j));
              }
           }
        }

        cout<<dp[1][n];//输出1到n这个区间的结果
        return 0;
}
```

优化---前缀和

提前计算（预处理），把主运算中的一次循环省略掉（提前算）

这里的时间复杂度是$O(n^4)$.超时了，怎么优化?我们看sum(i,j)每一次取计算i到j之间的数的和。这里可以用前缀和的方法（后面专门讲，这里先试着看看）。我们定义一个数组presum[], 让presum[i]表示前i个数的和，那么sum(i,j)是不是就等于presum[j]-presum[i]+a[i]?

- presum[]数组可以在dp之前准备一下，一次循环搞定：presum[i]=presum[i-1]+a[i];

```
#include <bits/stdc++.h>
using namespace std;
int n,a[201],dp[201][201],presum[201];
int main(){
        cin>>n;
        for (int i=1;i<=n;i++){
                cin>>a[i];
                presum[i]=presum[i-1]+a[i];//初始化前缀和数组
        }
        for(int len=2;len<=n;len++) {//枚举长度，从2开始，长度1的区间都是0
          for(int i=1; i+len-1<=n;i++) {//枚举区间区间左端点
            int j=i+len-1;//右端点
            dp[i][j]=INT_MAX;//为打擂初始化最大
            for(int k=i;k<j;k++) {//枚举左右区间的分割点,看下面的公式，k+1需要小于等于j
              // dp[i][j]=min(dp[i][j],dp[i][k] + dp[k + 1][j] + sum(i,j));
               dp[i][j]=min(dp[i][j],dp[i][k] + dp[k + 1][j] + presum[j]-presum[i]+a[i]);
            }
          }
        }
        cout<<dp[1][n];//输出1~n
        return 0;
}
```

验证程序

- 出了用给出的样例数据测试，还应该构造特殊的数据来测试
- 对于这道题，由于n是大于0的，我们至少可以尝试构造n==1和2的数据去验证一下。

Multiplication Puzzle

- 给出n个数，每次从中抽取一个数出来与相邻两个数相乘，直到抽到只剩两个数字，第一个数和最后一个数不能抽，求这些乘积的和的最小值。
- 输入格式

第一行一个整数n(3 <=n <= 100)，第二行空格分开的n个整数（1到100）

- 输出格式

一个整数，最小值

- 示例输入

5

10 1 50 20 5

- 示例输出

1150

示例说明：如果依次选择1 20 50，那么和为：
10∗1∗50+50∗20∗5+10∗50∗5=500+5000+2500=8000

如果选择 50 20 1，和最小，为1∗50∗20+1∗20∗5+10∗1∗5=1000+100+50=1150.

注意

- 如果给出n个数字最后剩下的一定是边上的两个
- 那么，如果在第i和第j个数字中间按照这个规则选择，最后剩下的一定是第i和第j个数。
- 用例子体会一下：
  - 10 1 50 20 5 无论如何选，最后剩下最前面的10和最后面的5
  - 10 1 50 20 无论如何选，最后剩下的是最前面的10和最后面的20
  - 10 1 50最后一定剩下10和50

已经算出了所有比较小的区间的最小值

如果我们已经算出了比较小的区间的最小值，如何得到大区间的最小值？

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

区间(长度6)最小值 + 3*1*2 + 区间(长度7)最小值

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

区间(长度7)最小值 + 3*2*2 + 区间(长度6)最小值

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

打擂

思路

- 如果有很多数（放在数组a中）去合并，比如从第1个到第n个，最后一个合并的数可能是从第2个到第n-1个都有可能，这些可能中我们要找最小的一个，比如我们选择第k个数，如果我们已经知道了前面第1到k个数的最优解，并且知道了第k到n个数的最优解。那么这种情况的和就是a[k]*a[i]*a[j]+ (第1到k个数的最优解)+第k到n个数的最优解。注意：每一个区间合并到最后剩下左右两个数。
- 我们设置dp[i][j]表示第i个数到第j个数的最优解，那么递推公式：dp[i][j]=min{dp[i][k]+dp[k][j]+a[i]*a[k]*a[j]}, 其中k大于i小于j。
- 初始化dp值为0。
- 可以看出，我们找dp[i][j]也就是i到j区间的值，需要先算出所有i和j之间的各种区间的值。所以用区间动态规划，按照长度（从3开始）从小到大计算各个区间的值，保证在求i和j之间值的时候，所有小于i和j长度的小区间都已经计算出来了。

一起套模板写

```
for(int len=1;len<=n;len++)//枚举长度

{

   for(int i=1; i+len-1<=n;i++)//枚举区间区间左端点

   {

       //do something

      int j=i+len-1;//右端点

       for(int k=i;k<j;k++)

     //枚举左右区间的分割点

       {

          //get dp[i][j] from dp[i][k]  and dp[k+1][j]

       }

   }

}
```

```
#include <bits/stdc++.h>
using namespace std;
int n,a[101],dp[101][101];//如果每一个数太大，就要long long了
int main()
{
        cin>>n;
        for (int i=1;i<=n;i++)  cin>>a[i];
        for(int len=3;len<=n;len++)//枚举长度，从3开始，长度1，2区间都是0
        {
           for(int i=1; i+len-1<=n;i++)//枚举区间区间左端点
           {
              int j=i+len-1;//右端点
              dp[i][j]=INT_MAX;//为打擂初始化最大
              for(int k=i+1;k<j;k++)//需要保证k-1>=i,k+1<=j
              //枚举左右区间的分割点，打擂找最小的就是dp[i][j]
              {
                 dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]+a[i]*a[k]*a[j]);
              }
           }
        }

        cout<<dp[1][n];//输出1~n
        return 0;
}
```

验证程序

- 出了用给出的样例数据测试，还应该构造特殊的数据来测试
- 对于这道题，由于n是3到100之间的，我们至少可以尝试构造n==3的数据
- 输入

  3

  1 2 3
- 输出

  6
- 测试一下

区间动态规划总结

- 动态规划：从前面的状态（最优解）推导出后面的状态（最优解）。
- 区间动态规划：二维的动态规划。从小的区间推导出大的区间。计算出所有的小的长度的区间的最优解，然后通过合并这些小的区间，得到大的区间的最优解。注意：不一定是合并，其实就是要找分割点，看大的区间从哪两个小的区间可以得到最优值。

```
for(int len=1;len<=n;len++)//枚举长度，从小到大也就是先计算小区间
{
   for(int i=1; i+len-1<=n;i++)//枚举区间区间左端点，计算出所有小区间
   {
      //do something
     int j=i+len-1;//根据长度和左端点计算出右端点
      for(int k=i;k<j;k++)
    //枚举左右区间的分割点，从中寻找最优解
      {
         //get dp[i][j] from dp[i][k]  and dp[k+1][j]
      }
   }
}
```

[CSP-J2020] 方格取数——一起去网站看看题目

设有 n×m 的方格图，每个方格中都有一个整数。现有一只小熊，想从图的左上角走到右下角，每一步只能向上、向下或向右走一格，并且不能重复经过已经走过的方格，也不能走出边界。小熊会取走所有经过的方格中的整数，求它能取到的整数之和的最大值。

输入格式

第一行有两个整数 n, m。

接下来 n 行每行 m 个整数，依次代表每个方格中的整数。

输出格式

一个整数，表示小熊能取到的整数之和的最大值。

对于 100% 的数据，$1 \le n,m \le 10^3$。方格中整数的绝对值不超过 10^4

看示例数据

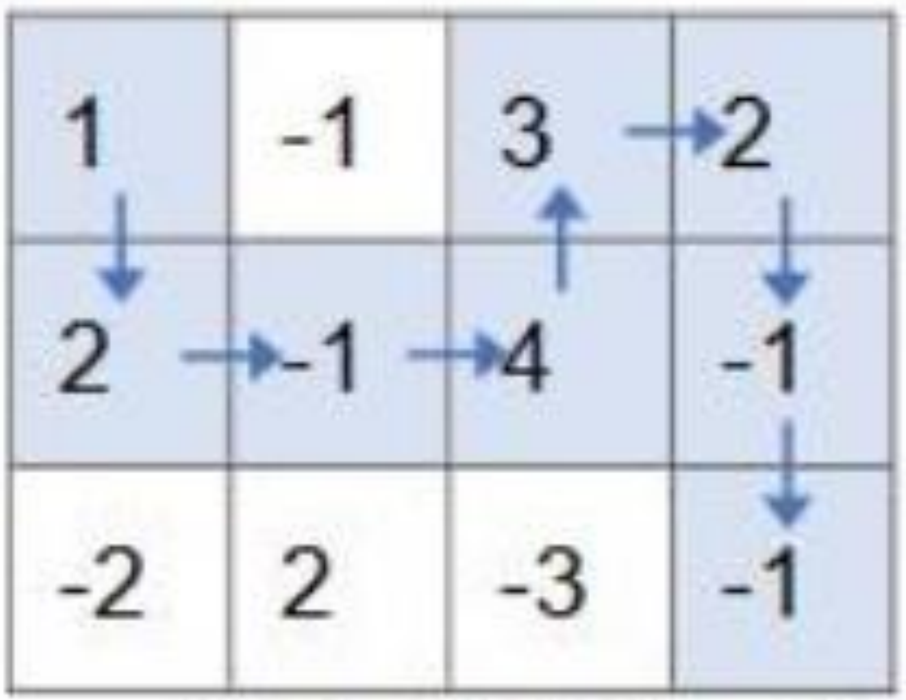
3 4

1 -1 3 2

2 -1 4 -1

-2 2 -3 -1

答案： 9



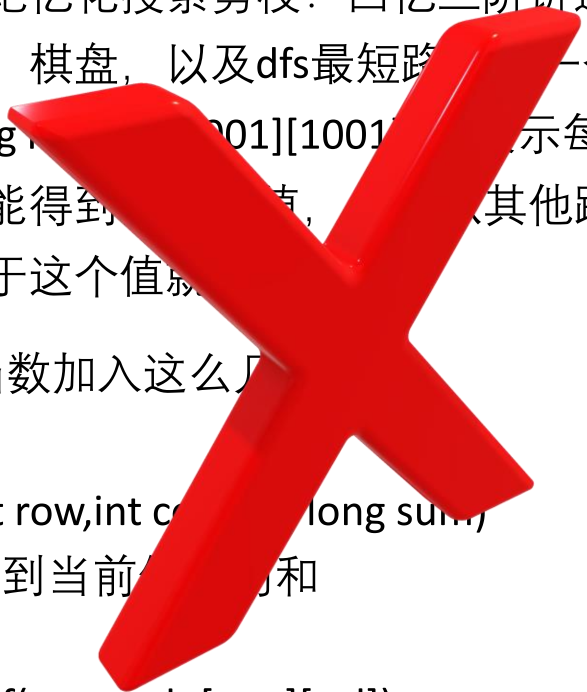
思路1

- 暴力dfs：需要走完所有的可能路径，至少得了20分左右。切记，如果考试想不到其他的方法，暴力！得多少分都可以。
- 暴力搜索的程序特别简单，由于我们搜索所有路径，所以visited状态需要手工回溯，因为每一个位置可能出现在不同得路线上。
- 注意：要用long long而不是int来表示和，因为最坏情况这些位置加起来超过了10的9次方。

思路2

- 能否用记忆化搜索剪枝？回忆三阶讲过得2017年的真题：棋盘，以及dfs最短路[illegible]一个二维数组比如long [illegible]01][1001][illegible]示每一个位置当前所能得到[illegible]值，[illegible]其他路线走过来如果小于这个值就[illegible]
- 在DFS函数加入这么几[illegible]

```
void dfs(int row,int c[illegible]long su[illegible])
//sum是走到当前[illegible]和
{
        if(sum<=dp[row][col]) return;
        dp[row][col]=sum;
```



当绿色线路走到红色位置的时候，值是6，比7小，但是，这条路才是最大的一条路

1	-1	3	2
2	-1	4	-1
-2	2	-3	-1

怎么办?

30多分了！

2分别表示从上面过来，下面过来，左面过来，dfs函数调用的时侯加上参数表示从哪个方向过来。这样剪枝的时侯就可以根据不同的方向单独剪了。

```
void dfs(int row,int col,long long sum,int from)
{
    if(sum<=dp[row][col][from]) return;
    dp[row][col][from]=sum;
    if (row==n && col==m) {
        ans=max(ans,sum);
...
ow<1
l>m)
            continue;
        visited[newrow][newcol]=true;
        dfs(newrow,newcol,sum+a[newrow][newcol],i);
        //i这个下标正好表示了方向
        visited[newrow][newcol]=false;
    }
}
```

思路3：记忆化搜索

- 现在long long dp[1001][1001][3];表示当前位置(从某一个特定方向过来)到后面能够取到的最大值。按照记忆化搜索的思路，dfs函数现在返回当前位置到达最后的最大值，那么如果b是a的相邻节点，dfs(a)的结果就应该是到达a的值加上dfs(b)的结果。这个dfs(a),dfs(b)都记忆在dp数组里面。
- 小tip：这里最小数字好几个地方用到，干脆定义一个

```
const long long minlong = -0x7ffffffffffffff;
```

```
long long dfs(int row,int col ,int from)//表示当前row,col位置从某一个特定方向过来，到后面能够取到的最大值
{
        if(dp[row][col][from]!=minlong) return dp[row][col][from];
        if (row==n && col==m) {
                return dp[row][col][from]=a[n][m];
        }
        //dp[row][col][from]应该等于3个可能走相邻节点的最大值，并且返回
        for (int i=0;i<3;i++)
        {
                int newrow=row+rdir[i];
                int newcol=col+cdir[i];
                if (visited[newrow][newcol] ||newrow<1 ||newcol<1 || ne
                visited[newrow][newcol]=true;
                dp[row][col][from]=max(dp[row][col][from],a[row][col]+dfs(newrow,newcol,i));//i这个下标正好表示了方向
                visited[newrow][newcol]=false;
        }
        //dp[row][col][from]+=sum;
        return  dp[row][col][from];
}
//注意：dp数组初始化所有值为minlong最小数
```

主程序中调用dfs函数：

```
cout<<dfs(1,1 ,0);//开始可以认为是从上面过来，
```

100!!!

这里访问相邻节点也可以不用方向数组和visited数组来控制，实际上根据from我们是知道下一步可能访问哪个位置的，比如如果从上面来，下一次就只可能两个分支：右和下。

换个方向，从出口往前推，并且优化不用dir数组和visited数组

```
long long dfs(int row,int col ,int from)//dp[row][col][from]表示row col位置到入口的从from方向来的最大值
{
        if (   row<1 ||col<1 || row>n || col>m) return minlong;
        if(dp[row][col][from]!=minlong) return dp[row][col][from];
        //dp[row][col][from]应该等于3个可能的方向的最大值，并且返回;
        if (row==1 && col==1) {
                return dp[row][col][from]=a[1][1];
        }
        if(from==0)  dp[row][col][from]=a[row][col]+max(dfs(row-1,col,0),dfs(row-1,col,2));
        //从上面来的，那么上一个位置只能从左面或者上面来
        else if(from==1) dp[row][col][from]=a[row][col]+max(dfs(row+1,col,1),dfs(row+1,col,2));
        //从下面来的，那么上一个位置只能从左面或者下面来
        else dp[row][col][from]=a[row][col]+max(max(dfs(row,col-1,1),dfs(row,col-1,0)),dfs(row,col-1,2));
        //从左面来的，上一个可能从3个位置来
        return  dp[row][col][from];
}
```

主程序调用： cout<<max(dfs(n,m ,0),dfs(n,m ,2));

最后一个位置可能是从上面来的，也可能是从左面来的，所以取最大。

就是把记忆化搜索反过来，循环递推。从小的行列到大的行列。
动态规划，就是把上面的记忆化搜索反过来用循环递推。我们注意到：

```
if(from==0)  dp[row][col][from]=a[row][col]+max(dfs(row-1,col,0),dfs(row-1,col,2));
//从上面来的，那么上一个位置只能从左面或者上面来
else if(from==1) dp[row][col][from]=a[row][col]+max(dfs(row+1,col,1),dfs(row+1,col,2));
//从下面来的，那么上一个位置只能从左面或者下面来
else dp[row][col][from]=a[row][col]+max(max(dfs(row,col-1,1),dfs(row,col-1,0)),dfs(row,col-1,2));
//从左面来的，上一个可能从3个位置来
```

对于dp里面的3个值，列都是从小到大推的，比如从col-1推导出col，可以作为外循环。

而行有的是从大到小，有的是从小到大。所以需要不同的循环。道理上也和容易明白，由于是只能往右不能往左，所以列从左到右也就是从小到大进入下一列。对于当前到达的每一列，该位置可以从上一行或者下一行来，所以从大到小或者从小到大需要不同的循环。

注意，这里边界处理比较麻烦，因为实际上第一列不能从左面来，第一行不能从上面来，最后一行不能从下面来。所以几个循环的开始值有所不同。

思路4：动态规划

```
for(int i=1;i<=n;i++)  dp[i][1][0] = dp[i-1][1][0] + a[i][1]; //初始化第一列的值，第一列只能从上往下走，所以后面的dp推到中从第2列开始推导

for(int j=2;j<=m;j++)   {

    for(int i=1;i<=n;i++)    {

            // form ==2  dp[row][col][from]=a[row][col]+max(max(dfs(row,col-1,1),dfs(row,col-1,0)),dfs(row,col-1,2));  左面来的

            dp[i][j][2]=max( max(dp[i][j-1][0],dp[i][j-1][1]), dp[i][j-1][2]) +a[i][j];

    }

    for(int i=2;i<=n;i++)    {//注意从上往下从第2行开始推导，第一行没有上面过来的

            //if(from==0)  dp[row][col][from]=a[row][col]+max(dfs(row-1,col,0),dfs(row-1,col,2)); 上面来的

            dp[i][j][0]=max(dp[i-1][j][0],dp[i-1][j][2])+a[i][j];

    }

   for(int i=n-1;i>=1;i--)   {//从下往上从n-1行开始，因为最后一行不能从下上来

            //if(from==1) dp[row][col][from]=a[row][col]+max(dfs(row+1,col,1),dfs(row+1,col,2)); 下面来的

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

    }

}

cout<<max(dp[n][m][0],dp[n][m][2]);
```

复习背包

三种背包问题，没有压缩之前，都是二维的dp，思路其实是一致的，就是设置dp[i][j]为容量是j的时候前i个物品里面去选择的最优解。

0/1背包：第i个物品选择放0个和放1个，其中找最优。双循环。

多重背包：第i个物品选择放0,1,2…s[i]个，其中找最优。其中s[i]表示i物品有多少个可以选择。三重循环。

完全背包：第i个物品选择放0,1,2….个，直到背包放不下。三重循环。

可以理解多重背包是完全背包的特例（每个物品数量有限制），而0/1背包是更特殊的情况（每一个物品只有1个）。

复习：完全背包简化版本

对于n个物品任选，每一个容量j的最大值dp[j], 等于max{选择一个物品i之后剩下的容量可以得到的最大值},注意容量从前往后选，从而保证i物品小的容量都计算过了。

```
for(int i=1;i<=n;i++)
//一共n个物品
 {
      for(int j=w[i];j<=c;j++)
     //c是背包容量，w[i]是第i个物品的重量
      {
                    dp [j]=max(dp[j],dp[j-w[i]]+ v[i]);
       }
 }
cout<<dp[c]<<endl;
```

挑战作业1

- P4170 [CQOI2007]涂色
- https://www.luogu.com.cn/problem/P4170
- 提示：如果定义dp[i][j]就是字符串i位置到j位置之间的区间的最小染色次数。那么：
  - 如果字符串i位置的字符和j位置字符不同，如何推导dp[i][j]，假设k为i到j之间的任意一个位置，dp[i][j]和dp[i][k],dp[k+1][j]什么关系？
  - 如果相同呢？和dp[i+1][j], dp[i][j-1]什么关系？
- 注意：区间长度从2开始，那么长度为1的也就是表示只有一个字母的，需要初始化为1，因为一个字母最少需要1次染色。

挑战选做作业 2

- NOIP 2019纪念品 https://www.luogu.com.cn/problem/P5662
- 提示："当日购买的纪念品也可以当日卖出换回金币"这句很重要，告诉我们实际上每天都可以是一个新的背包问题，背包就要找value，weight，volume三个。不难发现value就是纪念品的价格差（负的不要），weight就是纪念品价格，volume就是总金额。这个volume每天都要更新（就是类似每天全部先卖出去）。

由易到难，思维体系训练
实战结合，创新协作培养
兴趣导向，未来职业引领