算法设计

作　　者: 罗裕辉

时　　间: 2024 年上

目录

一、 前置知识

1.1 常用优化

1.1.1 头文件简写

由于写题目时可能需要用到多个头文件，而 bits 目录下的 stdc++.h 文件中存在多个常用头文件，尽管它不是 c++ 标准头文件，但一般情况下都支持它。

1.1.2 输入输出优化

cin、cout 和 scanf、printf 默认是同步的，我们可以关掉同步，以及 cin、cout 和其他流的绑定，从而加快输入输出。

1.1.3 数据溢出问题

int 是 32 位，范围是 -2^{31} $2^{31}-1$，最大大概是 21 亿，有可能会溢出。因此我们可以使用宏定义将 int 替换成 long long 减少溢出风险 (当然此时 main 函数的返回类型要改成 signed)。

1.1.4 无穷大的设定

由于数据范围有大小限制，有时我们需要设定一个无穷大的数值，通常是每个字节设置成 0x3f，long long 类型就是 8 个 0x3f。

1.1.5 最终模板

最终代码模板如下:

```
#include<bits/stdc++.h>
#define int long long
#define INF 0x3f3f3f3f3f3f3f3f

using namespace std;

signed main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);

```

```
12
13    return 0;
14 }
```

1.2 打表

1.3 STL

二、 基础算法

2.1 离散化

离散化通常用于要开辟一个较大的数组，索引的范围可能比较大 (比如超过 1e9)，但实际存储数据的的索引的个数很少，也就是比较稀疏。而重要的只是这些索引的相对大小，那么可以将这些索引中间的数舍去，进行压缩，也就是离散化。

离散化首先需要排序 (确定相对大小)，然后去重 (注意这步不是必须的，因为后面二分是得到 >=x 的边界，相同数二分的结果相同，只是可以稍微减少长度)，然后当我们需要用到某个索引时可以使用二分快速查找它对应的新索引 (通常 +1，从 1 开始)。

本质上来说，离散化就是将索引的绝对大小映射成相对大小，并且绝对大小变成相对大小不影响题目求解。

代码如下：

```
#include<bits/stdc++.h>
#define int long long
#define INF 0x3f3f3f3f3f3f3f3f
using namespace std;

vector<int> nums;//离散化后的数组
int find(x){
    return lower_bound(nums.begin(), nums.end(), x) - nums.begin() + 1;
}
signed main(){
    //...已经将目标值放在nums中
    sort(nums.begin(), nums.end());
    nums.erase(unique(nums.begin(), nums.end()), nums.end());
    return 0;
}
```

▶ 区间和

- 思路分析：这里目标数据是数轴下标，但是下标的范围非常大，是 2e9，实际上用到的确实 3e5，这时我们可以离散化处理将用到的下标进行离散化，映射到从 1(方便前缀和) 开始的新下标当对某个数进行操作时找到其对应下标，再二分转换成对对应数的操作，节省空间开销
- 代码实现

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
const int N=3e5+10;
vector<int>
    ↪ nums;//原始数组，实际上是一个哈希表了，为了持续查询，一般离散化后就不动了
int a[N],s[N];//转换后的数组，下标来自nums
vector<pair<int,int>>add,query;//存储操作
int n,m;

int find(int x){
   int l=0,r=nums.size()-1;
   while(l<r){
      int mid=l+r>>1;
      if(nums[mid]>=x)r=mid;
      else l=mid+1;
   }
   return r+1;//下标从1开始，方便前缀和
}
int main(){
   cin>>n>>m;
   int index=0;
   while(n--){
      int x,c;
      cin>>x>>c;
      nums.push_back(x);//凡是要对原数据进行操作的都要进行离散化
      add.push_back({x,c});
   }
```

```
    while(m--){
        int l,r;
        cin>>l>>r;
        nums.push_back(l);
        nums.push_back(r);
        query.push_back({l,r});
    }
    sort(nums.begin(),nums.end());//排序
    nums.erase(unique(nums.begin(),nums.end()),nums.end());//去重
    for(auto item : add){//处理加法操作
        int index=find(item.first);
        a[index]+=item.second;
    }
    for(int i=1;i<=nums.size();i++){//计算前缀和
        s[i]=s[i-1]+a[i];
    }
    for(auto item : query){//处理查询操作
        int l=find(item.first);
        int r=find(item.second);
        cout<<s[r]-s[l-1]<<endl;
    }
    return 0;
}
```

2.2 排序

2.2.1 快速排序

快速排序顾名思义，很快，平均复杂度 O(logn)，当每次选择的基准元素都是当前序列中的最大或最小值，最差是 $O(n^2)$，不过能够通过随机选取和三数取中的方法减少发生概率, 且空间复杂度是 O(1)，可以原地排序。

基本思想是选取基准元素，头尾各准备一个指针，前指针找到第一个大于等于基准元素，后指针找到第一个小于等于基准元素，两数交换，继续移动，这样排好一个元素的位置，然后左右两边递归排序。

▶ 第 k 个数

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

const int N = 100010;
int num[N];
void quickSort(int l,int r){
   if(l>=r)return;
   int base = num[l+r>>1];//选择中点作为pivot
   int i=l-1,j=r+1;//准备两个指针
   while(i<j){
      do{i++;}while(num[i]<base);
      do{j--;}while(num[j]>base);
      if(i<j)swap(num[i],num[j]);
   }
   quickSort(l,j);
   quickSort(j+1,r);
}
int main(){
   ios::sync_with_stdio(false);
   int n;
   cin>>n;
   for(int i=0;i<n;i++){
      cin>>num[i];
   }
   quickSort(0,n-1);
   for(int i=0;i<n;i++){
      cout<<num[i]<<" ";
   }
   return 0;

}
```

利用快排思想，每次可确定一个大小范围的区间，由此衍生出快速选择算法。

► 第 k 个数 (快速选择)

```cpp
#include<iostream>
```

```cpp
using namespace std;

const int N = 100010;
int num[N];
int quickSort(int l,int r,int k){
   if(l>=r)return num[l];
   int base = num[l+r>>1];//选择中点作为pivot
   int i=l-1,j=r+1;//准备两个指针
   while(i<j){
      do{i++;}while(num[i]<base);
      do{j--;}while(num[j]>base);
      if(i<j)swap(num[i],num[j]);
   }
   int s1=j-l+1; //注意边界num[j]不一定就是base,也就不一定是j-l+1小的数
   if(k<=s1)return quickSort(l,j,k);//如果k小于等于左区间长度,在左区间找
   else return quickSort(j+1,r,k-s1);//否则在右区间找,注意这里的k变成k-s1
}
int main(){
   ios::sync_with_stdio(false);
   int n,k;
   cin>>n>>k;
   for(int i=0;i<n;i++){
      cin>>num[i];
   }
   int ret=quickSort(0,n-1,k);
   cout<<ret;
   return 0;

}
```

2.2.2 归并排序

归并排序复杂度均是 O(logn)，因为它用的是二分思想，是一种稳定的排序，只不过空间复杂度是 O(n)。

基本思想是先二分，递归左右两边，再将两个已经有序的序列合并。

▶ 归并排序

```cpp
#include<iostream>
using namespace std;
void mergeSort(int* arr,int l,int r){
   if(l>=r)return;
   int mid = l+r>>1;
   mergeSort(arr,l,mid);
   mergeSort(arr,mid+1,r);//先递归
   int* temp = new int[r-l+1];
   int i=l,j=mid+1,k=0;
   while(i<=mid&&j<=r){//再合并
      if(arr[i]<=arr[j])temp[k++]=arr[i++];
      else temp[k++]=arr[j++];
   }
   while(i<=mid)temp[k++]=arr[i++];
   while(j<=r)temp[k++]=arr[j++];
   for(i=l,k=0;i<=r;i++,k++)arr[i]=temp[k];//赋回
}

int main(){
   int n;
   cin>>n;
   int* arr=new int[n];
   for(int i=0;i<n;i++){
      cin>>arr[i];
   }
   mergeSort(arr,0,n-1);
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
   return 0;
}
```

▶ 逆序对的数量

- 思路分析：利用归并过程中需要比较两个数大小，这时可以记录逆序对数量。

- 代码实现：

```
#include<iostream>
using namespace std;
//本质qiu
const int N = 100010;
int num[N];
int temp[N];
void mergeSort(int l,int r,long long &ret){ //引用传参
   if(l>=r)return;
   int mid=l+r>>1;
   mergeSort(l,mid,ret);
   mergeSort(mid+1,r,ret);
   int i=l,j=mid+1,k=0;

   while(i<=mid&&j<=r){
      if(num[i]<=num[j]){
         temp[k++]=num[i++];
      }
      else{
         ret +=
            ↪ mid-i+1;//对于每个num[j],比它大的的是num[i]的后面mid-i+1个
         temp[k++]=num[j++];
      }
   }
   while(i<=mid)temp[k++]=num[i++];
   while(j<=r)temp[k++]=num[j++];//这里相当于i=mid+1,逆序对为0
   for(int i=l,k=0;i<=r;i++,k++)num[i]=temp[k];
}
int main(){
   ios::sync_with_stdio(false);//加快读取
   int n;
   cin>>n;
   for(int i=0;i<n;i++){
      cin>>num[i];
   }
```

```
    long long ret=0;//不开long long见祖宗
    mergeSort(0,n-1,ret);
    cout<<ret;
    return 0;
}
```

2.3 二分

二分缩小了搜索空间，一次减少一半，二分不一定要求单调，只要左右两边满足的性质不同即可。

基本思想是先选取分界点，然后根据需要二分，一般有两种

- 左边满足某种性质，右边不满足，得到满足左边性质的最右边界。
- 右边满足某种性质，左边不满足，得到满足右边性质的最左边界。

2.3.1 整数二分

整数二分是最常见的二分，需要注意的是分界点的选取。

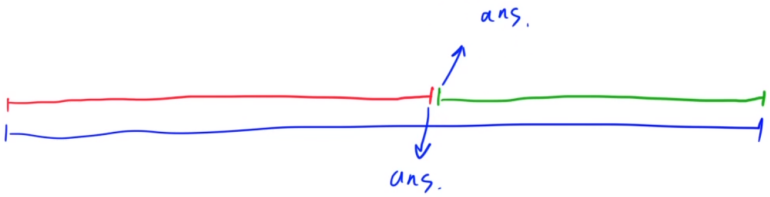


图 1

```
#include<iostream>
using namespace std;
int binarySearch1(int* arr,int l,int r){
   while(l<r){
      int mid = l+r>>1;
      if(arr[mid]>=3)r=mid;
         ↪ //如果mid满足性质，这里是寻找满足右边性质的最左边界,更新r,
      else l=mid+1;
         ↪ //比如>=某个数，mid和l,r的设置是配套的，否则可能死循环
   }
```

```
    return l;
        ↪ //l=r最后可能还要检查，因为整个数组可能不存在这样的边界
}
int binarySearch2(int* arr,int l,int r){
    while(l<r){
        int mid=l+r+1>>1;
        if(arr[mid]<=3)l=mid;//如果mid满足性质，这里是寻找满足左边性质的最右边界,更新l,
        else r=mid-1;
            ↪ //比如<=某个数，mid和l,r的设置是配套的，否则可能死循环
    }
    return l;
        ↪ //l=r最后可能还要检查，因为整个数组可能不存在这样的边界
}
int main(){
    int n;
    cin>>n;
    int* arr = new int[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    int left = binarySearch1(arr,0,n-1);
    int right = binarySearch2(arr,0,n-1);
    cout<<left<<" "<<right;
    return 0;
}
```

▶ 数的范围

- 思路分析：利用对于元素 k, 利用二分寻找 >=k 的左边界和 <=k 的右边界。
- 代码实现：

```
#include<iostream>
using namespace std;
const int N = 1e5+10;
int num[N];

int searchLeft(int k,int n){
```

```
    int l=0,r=n-1;
    while(l<r){
        int mid = l+r>>1;
        if(num[mid]>=k)r=mid;
        else l=mid+1;
    }
    return l;
}
int searchRight(int k,int n){
    int l=0,r=n-1;
    while(l<r){
        int mid = l+r+1>>1;
        if(num[mid]<=k)l=mid;
        else r=mid-1;
    }
    return l;
}

int main(){
    ios::sync_with_stdio(false);
    int n,q,k;
    cin>>n>>q;
    for(int i=0;i<n;i++){
        cin>>num[i];
    }
    while(q--){
        cin>>k;
        int left = searchLeft(k,n);
        if(num[left]!=k){
            cout<<-1<<" "<<-1<<endl;
        }
        else{
            int right = searchRight(k,n);//减少二分次数
            cout<<left<<" "<<right<<endl;
        }
```

```
    }
    return 0;
}
```

2.3.2 浮点数二分

浮点数二分相对简单一点，不用考虑边界问题，不过终止条件需要控制精度，下面以求一个大于 1 的正数平方根为例。

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



int main(){
   double n;
   cin>>n;
   double l=-10000,r=10000;
      ↪ //l,r的设置需要满足答案一定在[l,r]之间,一般两个分别负数和正数大点合适
   while(r-l>1e-8){
      ↪ //判断条件是精度控制，这里精度控制和小数点有关，一般精度数比要求小数点数多两位
      double mid = (l+r)/2;
      if(mid*mid>=n)r=mid;
      else l=mid;
   }
   cout<<fixed<<setprecision(6)<<l;
   return 0;
}
```

▶ 数的三次方根

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <iomanip>
using namespace std;

int main()
```

```
{
   double n;
   cin>>n;
   double l=-10000,r=10000;
   while(r-l>1e-8){
      double mid = (r+l)/2;
      if(mid*mid*mid>=n)r=mid;
      else l=mid;
   }
   printf("%.6f",l);
   return 0;
}
```

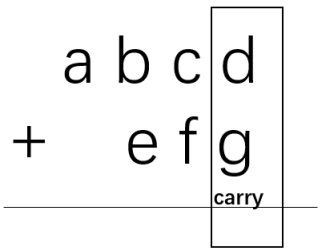
2.4 高精度

对于一些高精度的大整数，一般的数据类型无法存储，需要额外用数组存储，同时额外定义它们的四则运算方式。(Java 有内置的大整数类型，而 python 默认为大整数，不需要自己写高精度)。

2.4.1 高精度加法

▶ 高精度加法

- 思路分析：一般是两个高精度大整数相加。



前一个进位:carry
当前和:(d+g+carry)%10
当前进位:(d+g+carry)/10

图 2

- 代码实现

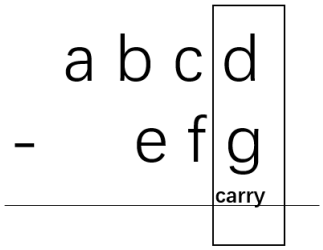
```
#include<iostream>
```

```cpp
#include <vector>

using namespace std;

vector<int> add(vector<int> &A,vector<int>
    ↪ &B){//给定高精度A和B，返回相加结果
   if(A.size()<B.size())return add(B,A);
   int carry=0;//进位，初始为0
   vector<int> C;
   for(int i=0;i<A.size();i++){//加的时候从低字节低位相加
      carry+=A[i];
      if(i<B.size())carry+=B[i];
      C.push_back(carry%10);
      carry/=10;
   }
   if(carry)C.push_back(carry);//最后清尾
   return C;
}

int main(){
   string a,b;
   cin>>a>>b;//以字符串形式读入
   vector<int>A,B,C;
   for(int
      ↪ i=a.size()-1;i>=0;i--)A.push_back(a[i]-'0');//小端序，低位在低字节，便于高
   for(int i=b.size()-1;i>=0;i--)B.push_back(b[i]-'0');
   C=add(A,B);
   for(int i=C.size()-1;i>=0;i--)cout<<C[i];
   return 0;
}
```

2.4.2 高精度减法

▶ 高精度减法

- 思路分析：一般是两个高精度大整数相减



前一个借位:carry
当前差:(d-g-carry+10)%10
当前借位:看d-g-carry的正负

图 3

- 代码实现

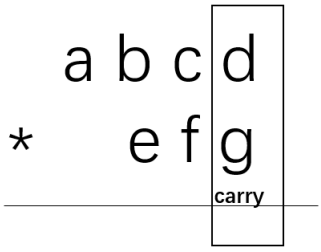
```
#include<iostream>
#include<cstring>
#include<vector>
using namespace std;

vector<int> sub(vector<int> &A,vector<int>
    ↪ &B){//给定高精度A和B,满足A>B,返回相减结果
   vector<int> C;
   int carry=0;//低位向当前位的借位
   for(int i=0;i<A.size();i++){
      carry=A[i]-carry;
      if(i<B.size())carry-=B[i];
      C.push_back((carry+10)%10);//可能需要向高位借位
      if(carry>=0)carry=0;//判断下一位的借位
      else carry=1;
   }
   while(C.size()>1&&C.back()==0)C.pop_back();//去掉前导0
   return C;

}
int main(){
   string a,b;
   cin>>a>>b;
   vector<int>A,B,C;
```

```
    for(int i=a.size()-1;i>=0;i--)A.push_back(a[i]-'0');//转换成整数
    for(int i=b.size()-1;i>=0;i--)B.push_back(b[i]-'0');
    if(a.size()<b.size()||(a.size()==b.size()&&a<b)){
        cout<<"-";//A<B时答案是负值
        C=sub(B,A);
    }
    else C=sub(A,B);
    for(int i=C.size()-1;i>=0;i--)cout<<C[i];
    return 0;
}
```

2.4.3 高精度乘法

► 高精度乘法

- 思路分析：一般是一个高精度乘以一个一般精度。



前一个进位:carry
当前积:(d*efg+carry)%10
当前进位:(d*efg+carry)/10

图 4

- 代码实现

```
#include<iostream>
#include <vector>

using namespace std;

vector<int> mul(vector<int> &A,int
    ↪ b){//给定高精度正整数和一般精度b，返回相乘结果
    vector<int> C;
```

```
    int carry=0;//低位向当前位的进位
    for(int i=0;i<A.size();i++){
        carry+=A[i]*b;
        C.push_back(carry%10);
        carry/=10;//更新进位
    }
    if(carry)C.push_back(carry);//扫尾
    return C;
}
int main(){
    string a;
    int b;//b一般是正常精度的数
    cin>>a>>b;
    if(b==0){//0提前返回
        cout<<0;
        return 0;
    }
    vector<int>A,C;
    for(int i=a.size()-1;i>=0;i--)A.push_back(a[i]-'0');
    C=mul(A,b);
    for(int i=C.size()-1;i>=0;i--)cout<<C[i];
    return 0;

}
```

2.4.4 高精度除法

▶ 高精度除法

- 思路分析：一般是一个高精度除以一个一般精度。

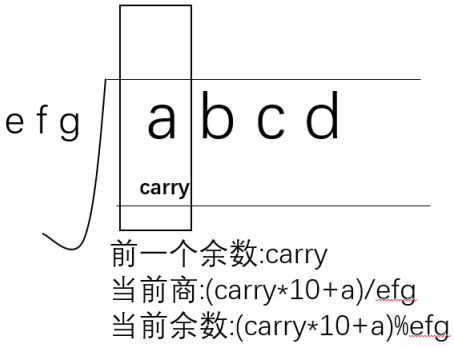


图 5

- 代码实现

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

vector<int> div(vector<int> &A,int
    ↪ b){//给定高精度A和一般精度b,返回相除结果(商和余数)
   vector<int> C;
   int carry=0;//商前一位除法留下的余数
   for(int i=A.size()-1;i>=0;i--){//除法需要从高位开始算
      carry=carry*10+A[i];
      C.push_back(carry/b);
      carry%=b;
   }
   reverse(C.begin(),C.end());//逆序，因为计算是先存储高位
   while(C.size()>1&&C.back()==0)C.pop_back();//去除前导0
   C.push_back(carry);//压入余数
   return C;


}
int main(){
   string a;
   int b;
```

```
    cin>>a>>b;
    vector<int>A,C;
    for(int i=a.size()-1;i>=0;i--)A.push_back(a[i]-'0');
    C=div(A,b);
    for(int i=C.size()-2;i>=0;i--)cout<<C[i];
    cout<<endl<<C[C.size()-1];
    return 0;

}
```

2.5 前缀和

2.5.1 一维前缀和

- 一维前缀和的构造：对于一维数组 s, 构造其前缀和公式如下，其中，i 从 1 到 n：

$$s_i = s_{i-1} + a_i$$

- 一维前缀和的应用：一维前缀和可以计算得到原数组任意连续项的和，比如要计算 [l,r]，公式如下：

$$s_r - s_{l-1}$$

▶ 前缀和

```
#include<iostream>

using namespace std;
const int N = 100010;

int a[N];
int s[N];

int main(){
    ios::sync_with_stdio(false);
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++){//下标从1开始使公式通用
```

```
        cin>>a[i];
        s[i]=s[i-1]+a[i];
    }
    while(m--){
        int l,r;
        cin>>l>>r;
        cout<<s[r]-s[l-1]<<endl;
    }
    return 0;
}
```

2.5.2 二维前缀和

- 二维前缀和的构造：要根据二维数组构造二维前缀和，就是要计算每一个 s(i,j)，如图：

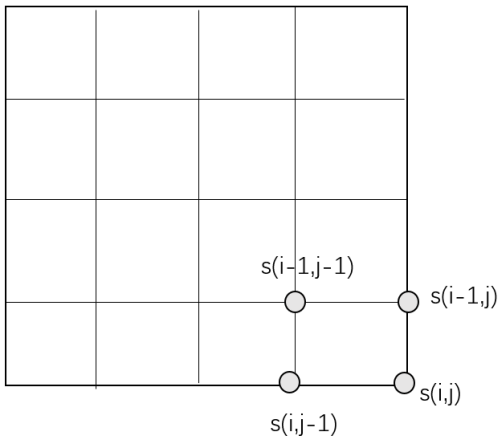


图 6

根据该点与其周围三点的关系，根据容斥原理，有以下公式：

$$s(i,j) = s(i,j-1) + s(i-1,j) - s(i-1,j-1) + a(i,j)$$

- 二维前缀和的应用: 根据二维前缀和，可以计算任意一个子矩阵内的数的和，比如计算 s(x1,y1) 到 s(x2,y2) 的子矩阵的和，如图：

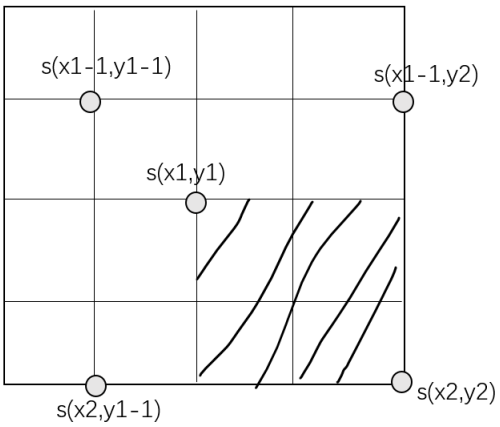


图 7

同样，根据容斥原理，公式如下；

$$ret = s(x2, y2) - s(x2, y1 - 1) - s(x1 - 1, y2) + s(x1 - 1, y1 - 1)$$

► 二维前缀和

```
#include<iostream>

using namespace std;
const int N = 1010;
int a[N][N];
int s[N][N];

int main(){
   int n,m,q;
   cin>>n>>m>>q;
   for(int i=1;i<=n;i++){
      for(int j=1;j<=m;j++){
         cin>>a[i][j];
         s[i][j]=s[i][j-1]+s[i-1][j]-s[i-1][j-1]+a[i][j];
      }
   }
   while(q--){
      int x1,y1,x2,y2;
      cin>>x1>>y1>>x2>>y2;
```

```
        cout<<s[x2][y2]-s[x2][y1-1]-s[x1-1][y2]+s[x1-1][y1-1]<<endl;
    }
    return 0;

}
```

2.6 差分

2.6.1 一维差分

▶ 差分

- 思路分析：对于原数组 a，差分数组 d 的定义是相邻两项之差，公式如下

$$d[i] = a[i] - a[i-1]$$

可以看出，前缀和和差分其实是逆运算，d 是 a 的差分，反过来 a 也是 d 的前缀和，当然，构造可以这样构造，但是，为了和它的应用相结合，我们将它视为一种特殊的应用，方便记忆。

对于一维差分的应用，构造一维差分数组后，可以快速让原数组某一连续子数组同时增加一个数，比如要使得 [l,r] 同时加 c, 那么只需要对差分数组进行如下操作：

$$d[l] += c;$$

$$d[r+1] -= c;$$

当然，这样操作完差分数组后还需要计算一遍前缀和得到原数组，但如果对原数组这种操作频繁的话只需最后计算一次，能快很多。

同时，对于差分数组的初始化也可据此公式，只不过 l=r，可以视为分别插入 a[i]。

- 代码实现

```
#include<iostream>

using namespace std;

const int N = 100010;
int a[N];
int d[N];//差分数组

```

```
int n,m;

void insert(int l,int r,int c){//对差分数组的操作，使得原数组[l,r]+c
   d[l]+=c;
   d[r+1]-=c;
}
int main(){
   ios::sync_with_stdio(false);
   cin>>n>>m;
   for(int i=1;i<=n;i++){
      cin>>a[i];
      insert(i,i,a[i]);//初始化差分数组
   }
   while(m--){
      int l,r,c;
      cin>>l>>r>>c;
      insert(l,r,c);
   }
   for(int i=1;i<=n;i++){
      a[i]=a[i-1]+d[i];//对差分数组再计算一次前缀和，更新原数组
      cout<<a[i]<<" ";
   }
   return 0;
}
```

2.6.2 二维差分

▶ 差分矩阵

- 思路分析：现在来看二维差分，直接根据定义推导不像前缀那样并不容易 (当然可以根据前缀和的公式逆推)，我们依然根据应用着手，构造本质上就是一个特殊的应用。二维差分的应用就是让某个矩阵内所有的数都加上同一个数，假设 d 是 a 的二维差分数组，a 是 d 的前缀和，我们看一下如何更新 d 才能让 a 某个矩阵内所有数都加上同一个数，比如对于 x1,y1 和 x2,y2 内的矩阵。
  显然，d 每一个数的增加都会使得右下角所有对应的 a 增加，要达到目标，根据容斥

原理，公式如下：

$$d(x1,y1)+=c;$$

$$d(x2+1,y1)-=c;$$

$$d(x1,y2+1)-=c;$$

$$d(x2+1,y2+1)+=c;$$

对于初始化，显然可以看作特殊的应用,x1=x2,y1=y2

- 代码实现

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

const int N = 1010;
int a[N][N];
int d[N][N];
int n,m,q;
void insert(int x1,int y1,int x2,int y2,int
    ↪ c){//对二维差分数组的操作，使得子矩阵同时+c
   d[x1][y1]+=c;
   d[x2+1][y1]-=c;
   d[x1][y2+1]-=c;
   d[x2+1][y2+1]+=c;
}
int main(){
   ios::sync_with_stdio(false);
   cin>>n>>m>>q;
   for(int i=1;i<=n;i++){
      for(int j=1;j<=m;j++){
         cin>>a[i][j];
         insert(i,j,i,j,a[i][j]);//初始化差分数组
      }
   }
   while(q--){
      int x1,y1,x2,y2,c;
      cin>>x1>>y1>>x2>>y2>>c;
```

```
        insert(x1,y1,x2,y2,c);
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            a[i][j]=a[i][j-1]+a[i-1][j]-a[i-1][j-1]+d[i][j];//对差分数组再计算一次前缀和
            cout<<a[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

2.7 前缀和及差分的小结

对前缀和及差分做一个小结，重点是理解它们的应用。

假设给定原数组 a

- 如果要多次快速得到 a 数组某一个连续的子数组的和，可以构造其前缀和数组 s，前缀和数组两数之差就是答案
- 如果要多次快速对 a 数组某一个连续的子数组同时进行操作，可以构造其差分数组 d，对差分数组操作，最后还需要进行一次前缀和才是答案

具体关系图下：

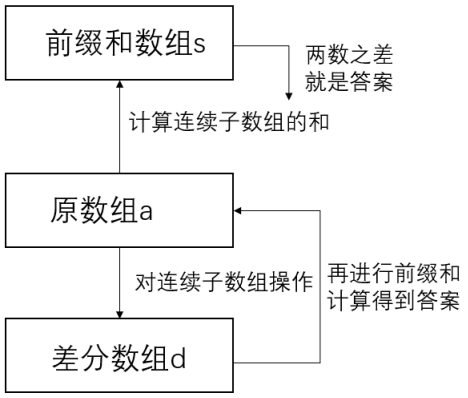


图 **8**

2.8 双指针

双指针是解决要求维护一种区间问题的暴力遍历的优化算法，传统暴力数字 i,j 指针，对每个 i 需要遍历整个 j，但双指针的关键在于利用 i 和 j 之间的单调关系，实现指针不回退的效果，一般可将 $O(n^2)$ 的复杂度优化成 O(n)

单数组一般模板如下 (双数组类似)：

```

//暴力：
for(int i=0;i<n;i++){
   for(int j=0;j<n;j++)
}

//双指针：
for(int i=0,j=0;i<n;i++){
   while(i<=j&&check(i,j))j++;//check是根据题目要求i和j的关系
   //其他逻辑;
}
```

▶ 最长连续不重复子序列

- 思路分析：使用双指针算法解决。
  首先设置指针 i，从前向后遍历，意图求以 i 结尾的最长连续不重复子序列
  接着设置指针 j，是子序列的左边界，只有当 [i,j] 区间内有重复元素时 j 才向后移动，可以证明，随着 i 的向右移动 j 也只会向右移动，否则 i-1 对应的 j 可以更靠左，但我们定义的 j 是只有重复元素才向右移动，已经是最左的了。
- 代码实现

```
#include<iostream>
using namespace std;
const int N = 1e5+10;
int a[N],count[N];//count记录区间内每种数对应的个数
int n;
int main(){
   ios::sync_with_stdio(false);
   cin>>n;
   int ret=0;
   for(int i=0;i<n;i++)cin>>a[i];
```

```
    for(int i=0,j=0;i<n;i++){//i是区间右端点，j是区间左端点
        count[a[i]]++;//插入a[i]
        while(j<=i&&count[a[i]]>1){//如果插入a[i]使得区间有重复，那么移动j去重
            count[a[j]]--;
            j++;
        }
        ret=max(ret,i-j+1);
    }
    cout<<ret;
    return 0;
}
```

▶ 数组元素的目标和

- 思路分析：和上一题不同，这里是两个数组的双指针，不过依然要利用单调性 i 指向 A 的开头，j 指向 B 的结尾，根据 A[i]+B[j] 和 x 的大小进行指针移动，复杂度 O(n)
- 代码实现

```
#include<iostream>
using namespace std;
const int N = 1e5+10;

int A[N];
int B[N];

int n,m,x;
int main(){
    ios::sync_with_stdio(false);
    cin>>n>>m>>x;
    for(int i=0;i<n;i++){
        cin>>A[i];
    }
    for(int i=0;i<m;i++){
        cin>>B[i];
    }
    for(int i=0,j=m-1;i<n&&j>=0;){
        if(A[i]+B[j]<x)i++;
```

```
        else if(A[i]+B[j]>x)j--;
        else{
            cout<<i<<" "<<j;
            return 0;
        }
    }
    return 0;
}
```

2.9 位运算

位运算巧妙运用了移位操作以及位运算符，可以高效实现一些功能。

- 判断奇偶性。

```
if(n&1==1) //奇数

if(n&1==0) //偶数
```

- 判断一个整数第 k 位是否是 1

```
n>>k&1
```

- 求一个整数二进制表示最后一个 1 代表的数

```
n&-n  //简称lowBit运算
```

▶ 二进制中 1 的个数

```
#include<iostream>
using namespace std;
const int N = 1e5+10;
int a[N];
int n;

int main(){
    ios::sync_with_stdio(false);
    unsigned int a = -10;
    for(int i=31;i>=0;i--)cout<<(a>>i&1);
```

```
    return 0;
}
```

2.10 区间合并

▶ 区间和

- 思路分析：问题需要将有交集的区间进行合并，输出一共有多少个合并后的区间
  区间问题一般使用贪心算法
  步骤如下：
  1. 将所有区间按照左端点排序
  2. 维护上一个合并后的区间 [start,end]，从前往后遍历每一个区间，并更新维护的区间
  3. 输出合并后的区间的数量
  关键在第二步
  对于当前区间，只需考虑是否和上一个合并后的区间关系，一共有三种情况：
  2.1 当前区间完全被上一个包含，这个时候不用更新, 不变
  2.2 当前区间和 [start,end] 有部分交集，更新 end
  2.3 当前区间和 [start,end] 无交集，也就是左端点也大于 end,[start,end] 设置成当前区间，同时也意味着上一个合并后的区间就是最终情况，计数 +1
- 代码实现

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 1e5+10;
struct Range{
    int l,r;
    bool operator<(const Range &r){
        return l<r.l;//按照左端点排序
    }
}range[N];
int n;
int main(){
    ios::sync_with_stdio(false);
    cin>>n;
    for(int i=0;i<n;i++){
```

```
        int l,r;
        cin>>l>>r;
        range[i]={l,r};
    }
    sort(range,range+n);
    int start=-2e9,end=-2e9;
    int count=0;
    for(int i=0;i<n;i++){
        Range &t = range[i];
        if(t.l<=end&&t.r>end)end=t.r;
        else
            ↪ if(t.l>end){//这种方法实际上统计的是初始值dao倒数第二个区间的数量，结果是
            start=t.l;
            end=t.r;
            count++;
        }
    }
    cout<<count;
    return 0;
}
```

2.11 递归与递推

2.11.1 递归

递归是一种更基本的实现技巧，自己调用自己，实现递归最重要的是确定一种序，能够通过解决子问题解决大问题，可以通过递归树理解递归过程。

递归树：递归树表示了递归函数的调用顺序，一般是一棵深度搜索树，f(i) 对应规模为 i 的调用函数。

▶ 递归实现指数型枚举

- 思路分析：问题：n 个整数选任意多个
  大致思路: 从前往后枚举每一个位置是否选，可以利用递归实现
  需要参数:st 状态数组各个位置选还是不选,u 标识当前位置
  当前层函数: 当前位置选还是不选
  下一层函数: 分支考虑下一个位置

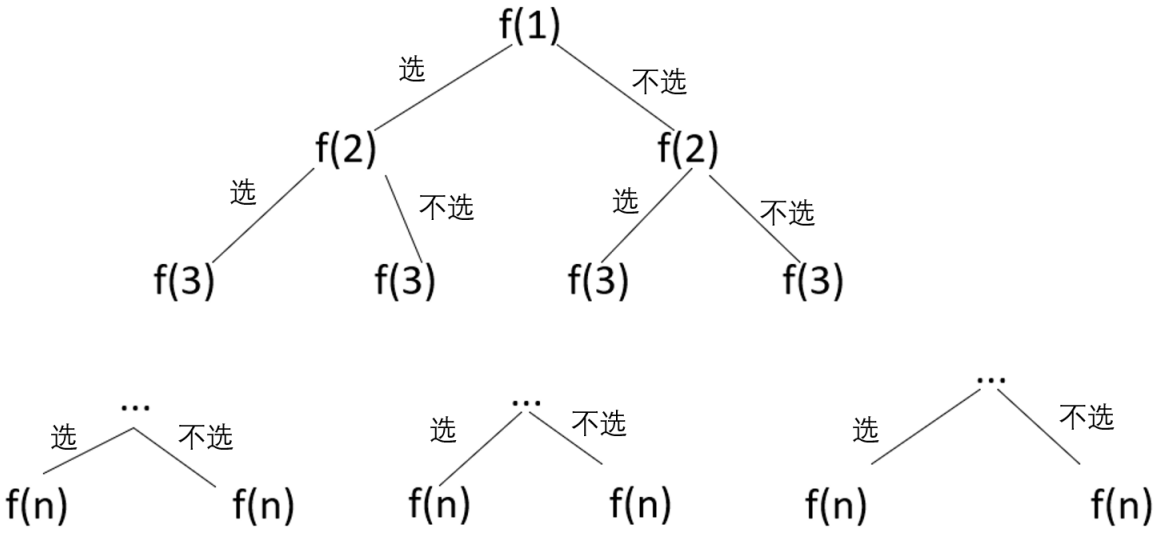


图 **9**

- 复杂度分析

  这里 f(n) 是考虑第 n 个方案取还是不取，复杂度是 O(1)。而递归树第 i 层宽度是 2^{i-1}。因此递归树的复杂度为 $1+2+4+...+2^{n-1}=O(2^n)$。

  此外，由于还要输出每个方案，复杂度是深度 O(n)。因此 $T(n)=O(2^n+n)=O(2^n)$

- 代码实现

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

const int N = 16;
int
    ↪ st[N];//状态数组，1表示当前数选择，0表示当前数不选择,-1表示选不选还不确定
int n;

void dfs(int u){
   if(u>n){//终止条件,输出结果
      for(int i=1;i<=n;i++){
         if(st[i]==1)cout<<i<<" ";
      }
      cout<<endl;
```

```
        return;
    }
    st[u]=1;
    dfs(u+1);
    st[u]=-1;//恢复现场，符合递归过程，但这里不需要

    st[u]=0;
    dfs(u+1);
    st[u]=-1;//恢复现场，但这里不需要
}
int main(){
    ios::sync_with_stdio(false);
    cin>>n;
    dfs(1);
    return 0;
}
```

▶ 递归实现排列型枚举

- 思路分析。

  问题：n 个整数顺序任意

  大致思路: 从前往后枚举每一个位置，选一个没有被选过的数，可以利用递归实现

  需要参数:st 状态数组每个位置选哪个数，used 标识每个数选还是没选,u 标识当前位置

  当前层函数: 当前位置选哪个每选过的数 (一般从前往后)

  下一层函数: 分支考虑下一个位置

- 复杂度分析

```
计算复杂度f(n)时都需要遍历st数组，复杂度为O(n)
总宽度为:1+n+n*(n-1)+...+n!
递归树复杂度为n(1+n+n*(n-1)+...+n!)
1+n+n*(n-1)+...+n! = (n!+n!/1+n!/2!+n!/3!+...+n!/n!)
    ↪ //利用n!>2^(n-1)将分母放缩
<=n!+n!/1+n!/2+n!/4+...+n!/2^(n-1)
<=n!(1+1+1/2+1/4+...+1/2^(n-1))
<=3n!
因此递归树复杂度为O(n*n!)，此外还有输出具体方案，复杂度是O(n!)
```

```
总共复杂度T(n)=O(n*n!)+O(n!)=O(n*n!)
```

- 代码实现

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

const int N = 10;

int st[N];//每个位置放哪个数，-1表示还未考虑
bool used[N];//由于每个数只能放一次，需要记录
int n;
void dfs(int u){
   if(u>n){
      for(int i=1;i<=n;i++){//结束条件
         cout<<st[i]<<" ";
      }
      cout<<endl;
      return;
   }
   for(int i=1;i<=n;i++){
      if(!used[i]){//从前往后寻找第一个未使用的数，保证字典序
         st[u]=i;
         used[i]=true;
         dfs(u+1);//枚举下一个位置
         st[u]=-1;//恢复现场，符合递归过程
         used[i]=false;

      }
   }
}
int main(){
   ios::sync_with_stdio(false);
   cin>>n;
   dfs(1);
   return 0;
```

```
}
```

▶ 递归实现组合型枚举

- 思路分析。

  问题：n 个数取出 m 个

  大致思路: 从前往后枚举每一个位置 (m)，选一个没有被选过的数同时保证最后的方案和已有的不重

  需要参数:st 状态数组每个位置选哪个数,u 标识当前位置，此外需要保证组合的性质可以用一个 prev 记录前一个位置选的数，保证选出的数升序排列，这样所有的方案不重不漏 (或者作为 dfs 的参数)

  当前层函数: 当前位置选哪个

  下一层函数: 分支考虑下一个位置

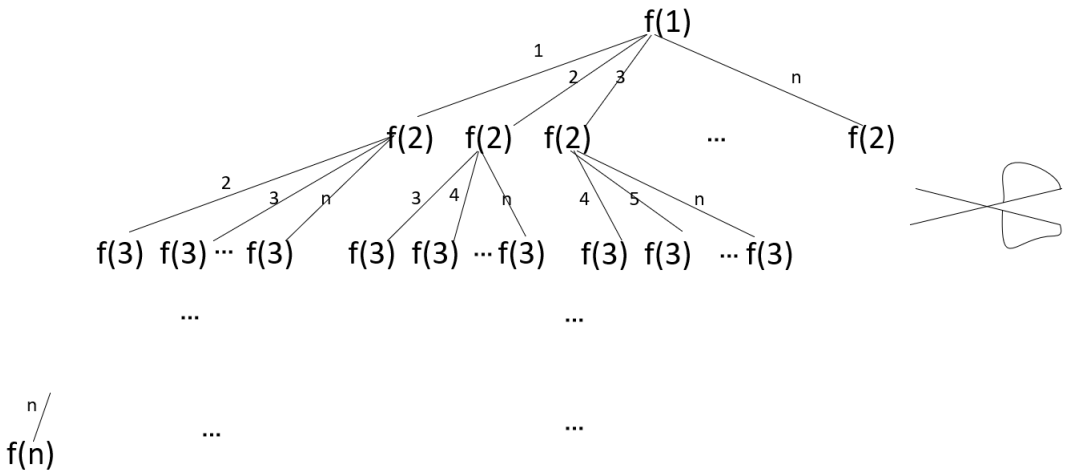


图 **10**

- 代码实现

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

const int N = 30;

int st[N];//每个位置放哪个数, -1表示还未考虑
int n,m;
void dfs(int u,int start){
   if(u>m){
```

```
        for(int i=1;i<=m;i++){//结束条件
            cout<<st[i]<<" ";
        }
        cout<<endl;
        return;
    }
    for(int i=start;i<=n;i++){
        st[u]=i;
        dfs(u+1,start+1);//枚举下一个位置
        st[u]=-1;//恢复现场，符合递归过程

    }
}
int main(){
    ios::sync_with_stdio(false);
    cin>>n>>m;
    dfs(1,1);
    return 0;
}
```

- 剪枝优化

  原始做法需要遍历到叶子节点才会返回，然而，由于题目需要保证后面枚举的数对于前面的大，因此对于一些节点 (某些子树的根) 已经不可能满足该条件，可以提前终止返回，实现剪枝的效果

  对于 dfs(u,start) 当前正在枚举第 u 个位置 (已经枚举了 u-1 个位置)，从 start 开始剩下需要枚举的位置是 m-u+1, 剩下可用的数是 n-start+1 如果 m-u+1>n-start+1，可以提前终止 (当然，这里其实是判断该子树深度最深的叶子节点有没有可能符合)

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

const int N = 30;

int st[N];//每个位置放哪个数，-1表示还未考虑
int n,m;
void dfs(int u,int start){
```

```
    if(m-u+1>n-start+1)return;
    if(u>m){
        for(int i=1;i<=m;i++){//结束条件
            printf("%d ",st[i]);
        }
        printf("\n");
        return;
    }
    for(int i=start;i<=n;i++){
        st[u]=i;
        dfs(u+1,i+1);//枚举下一个位置，保证比枚举过的数大
        st[u]=-1;//恢复现场，符合递归过程

    }
}
int main(){
    scanf("%d%d", &n, &m);
    dfs(1,1);
    return 0;
}
```

▶ 带分数

- 思路分析。暴力枚举: 由于需要满足 a,b,c 加起来是 1 9 9 个数字各出现 1 次，使得 n=a+b/c

  因此可以首先先求出 1 9 的全排列然后对每种排列枚举 a,b,c(隔板法) 最后看是否满足 a*c+b==n*c(/是整除，我们需要的是彻底除)
- 代码实现

```
#include<iostream>
#include<cstring>
#include <cmath>
using namespace std;

int count;
int a[10];//全排列的方案
bool used[10];
```

```
int n;
int getNum(int i,int j){//返回a中i~j位置对应的十进制数
   int res=0;
   for(int k=i;k<=j;k++){
      res=res*10+a[k];//用pow会超时
   }
   return res;
}
void dfs(int u){
   if(u>9){
      for(int i=1;i<=7;i++){ //_ _ _ _ _ _ _ | _ | _
         ↪ , 然后枚举挡板, 也就是枚举a,b,c
         for(int j=i+1;j<=8;j++){
            int a=getNum(1,i);
            int b=getNum(i+1,j);
            int c=getNum(j+1,9);
            if(a*c+b==n*c)count++;//如果这种a,b,c符合条件, 计数+1
         }
      }


   }
   for(int i=1;i<=9;i++){//首先得出1~9这9个数字的全排列
      if(!used[i]){
         a[u]=i;
         used[i]=true;
         dfs(u+1);
         a[u]=0;
         used[i]=false;
      }
   }
}
int main(){
   scanf("%d",&n);
   dfs(1);
```

```
    printf("%d",count);
    return 0;
}
```

- 小剪枝优化

  首先，这里需要先得出全排列公式再枚举 a,b,c 实际上可以不用这样，直接对 a，b,c 进行 dfs，对每个 a 再 dfs_b，对每个 b 再 dfs_c 这样可以结合题意进行小剪枝，比如对于 a 剪枝，也就是 a>=n 提前返回，因为需要保证 n=a+b/c，显然 a<n，而不是要枚举完才返回同时 a>0 才 dfs_b，b>0 才 dfs_c，做这些优化要比纯粹的暴力要快些，大概快 1s

```
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;
bool used[10];
int n;
int cnt;

void dfs_c(int u,int a,int b,int c){
    if(u>9){
        if(c&&c*n==a*c+b)cnt++;//只需保证c!=0，前面已经确定a和b不为0
        return;
    }
    for(int i=1;i<=9;i++){
        if(!used[i]){
            used[i]=true;
            dfs_c(u+1,a,b,c*10+i);
            used[i]=false;
        }
    }
}
void dfs_b(int u,int a,int b){
    if(u>9)return;
    if(b>0)dfs_c(u,a,b,0);
```

```
    for(int i=1;i<=9;i++){
        if(!used[i]){
            used[i]=true;
            dfs_b(u+1,a,b*10+i);
            used[i]=false;
        }
    }
}
void dfs_a(int u,int
    ↪ a){//u是当前枚举的位置，只是为了和模板保持一致，实际没用，a是枚举的a
    if(a>=n)return;//a>=n提前返回，而不必等到u>9
    if(a>0)dfs_b(u,a,0);
    for(int i=1;i<=9;i++){
        if(!used[i]){
            used[i]=true;
            dfs_a(u+1,a*10+i);
            used[i]=false;
        }
    }
}
int main(){
    scanf("%d", &n);
    dfs_a(1,0);
    printf("%d",cnt);
    return 0;
}
```

- 大剪枝优化

  上面虽然进行了简单的剪枝，但依然需要枚举 a,b,c，实际上可以只枚举 a,b,c 其中两个，这样通过等式 c*n==a*c+b 可以直接计算另一个变量 (而且是唯一的)，直接看剩下可用的数是否存在满足这种条件的情况

  比如这里我们可以先枚举 a 和 b(a 一定选上，可以 a>=n 剪枝，至于 b 和 c 似乎区别不大)，再计算 c，看剩余的数是否可能存在叶子节点是 c 的情况 (这里就是填补 used，使得它全是 true，不重不漏), 这样直接减去大概 1/3 子树的规模，剪枝更彻底

```
#include<iostream>
```

```cpp
#include<algorithm>
#include<cstring>

using namespace std;
bool used[10],temp[10];
int n;
int cnt;

bool check(int a,int b){
   if(b%(n-a)!=0)return
      ↪ false;//如果b不是n-a的倍数，直接返回，减少后面的计算，比枚举a和c，不过这里
   int c=b/(n-a);//貌似y总枚举a和c可能会溢出，但这里就不会
   memcpy(temp,used,sizeof(used));//先copy一下，避免对原数据的修改
   while(c){
      int low=c%10;//取最后一位
      if(low==0||temp[low]==true)return
         ↪ false;//low不能是0，同时不能已经用过，不重
      temp[low]=true;
      c=c/10;
   }
   for(int i=1;i<=9;i++){
      if(!temp[i])return false;//如果还有没用过的，返回false，不漏

   }
   return true;//temp全是true，返回true
}


void dfs_b(int u,int a,int b){
   if(u>9)return;
   if(b>0){
      if(check(a,b))cnt++;//直接剪去了c的子树，直接看是否可能存在,注意这里可不能提前
   }
   for(int i=1;i<=9;i++){
      if(!used[i]){
```

```
            used[i]=true;
            dfs_b(u+1,a,b*10+i);
            used[i]=false;
        }
    }
}
void dfs_a(int u,int
    ↪ a){//u是当前枚举的位置，只是为了和模板保持一致，实际没用，a是枚举的a
    if(a>=n)return;//a>=n提前返回，而不必等到u>9
    if(a>0)dfs_b(u,a,0);
    for(int i=1;i<=9;i++){
        if(!used[i]){
            used[i]=true;
            dfs_a(u+1,a*10+i);
            used[i]=false;
        }
    }
}
int main(){
    scanf("%d", &n);
    dfs_a(1,0);
    printf("%d",cnt);
    return 0;
}
```

2.11.2 递推

▶ 费解的开关

- 思路分析。

  题目要求是否能够最少的改变灯的操作使得所有灯都亮，每次改变一盏灯会使得它自身以及上下左右共 5 个位置的状态都发生改变

  要完成这道题，首先需要明确以下性质：

  1. 操作之间的顺序并不重要，灯和灯之间的操作并不影响最终的结果

  2. 每盏灯要么不动，要么改变一次，不会改变两次及以上，这是因为灯只有两种状态，改变偶数次相当于不变

既然灯的操作顺序并不重要，同时每盏灯最多只会改变一次，因此这道题最暴力的做法就是穷举每盏灯，不过会超时，我们来看看有没有什么优化的地方

优化关键是找到一种解决的顺序，找到问题内部的依赖关系，而不是盲目的穷举，每盏灯的状态实际上只涉及它上下左右以及自身的操作，因此可以这样考虑

1. 首先枚举第一行每盏灯的状态，这里第一行有 5 盏灯，共 32 种状态

2.(关键) 再考虑第二行，由于第一行的每盏灯的操作已经确定，而我们的目标是要使得最终所有的灯都亮，因此第二行灯的操作是固定的，它必须使得第一行的所有灯都亮，第二行的状态是唯一确定的

3. 以此类推，上一行完全决定了下一行 (其实有点类似贪心)

4. 最后确定的是最后一行的状态，可以保证前面所有行的灯都是亮的，只需要再看最后一行所有灯是否都亮，并统计总共的操作步数

- 代码实现

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<string>
using namespace std;
const int N = 510;
char a[5][6],b[5][6];
int n;
int dx[5] = {-1, 0, 1, 0, 0}, dy[5] = {0, 1, 0, -1, 0};

void turn(int x0,int y0){
   for(int i=0;i<5;i++){
      int x=x0+dx[i],y=y0+dy[i];
      if(x<0||x>4||y<0||y>4)continue;//判断边界
      b[x][y]^=1;//0和1字符的ASCII码最后一位不同，通过异或实现
   }
}

void solve(){
   int res=10;
   for(int i=0;i<32;i++){//遍历第一行的32种状态
      memcpy(b,a,sizeof(a));//拷贝a到b，防止修改原始数据
      int cnt=0;//记录操作数
```

```
24      for(int k=0;k<5;k++){//每种状态二进制表示就是第一行灯的改变与否
25          if(i>>k&1){
26              turn(0,k);
27              cnt++;
28          }
29      }
30      for(int j=1;j<5;j++){//依次确定下一行状态
31          for(int m=0;m<5;m++){
32              if(b[j-1][m]=='0'){
33                  turn(j,m);
34                  cnt++;
35              }
36          }
37      }
38      bool flag=true;
39      for(int j=0;j<5;j++){
40          if(b[4][j]=='0')flag=false;
41      }
42      if(flag)res=min(res,cnt);//当解可行时才更新答案
43    }
44    if(res>6)printf("-1\n");
45    else printf("%d\n",res);
46
47  }
48
49  int main(){
50    scanf("%d",&n);
51    while(n--){
52      for(int i=0;i<5;i++)scanf("%s",&a[i]);
53      solve();
54    }
55    return 0;
56  }
```

2.12 KMP

KMP 算法是一种改进的字符串匹配算法，也就是在目标字符串 S 寻找给定模式字符串 P 匹配的第一个字串的首字符位置，它是以三个提出者的名字命名的算法。

- 思路分析。
  - 暴力做法。

    暴力做法下的模式匹配可以直接穷举 S 的所有子串，逐个与 P 进行匹配，遇到不相等的就提前退出，否则匹配成功，代码如下：

    注意，这里字符串习惯上从 1 号开始存储，0 号位置存储字符串长度

```cpp
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
#include<string>
using namespace std;

int main(){
   char* S = " ajslfdjaswaz";
   char* P = " asw";
   for(int i = 1;i < strlen(S);i++){
      bool flag = true;
      for(int j = 1;j < strlen(P);j++){
         if(S[i+j-1] != P[j]){
            flag = false;
            break;
         }
      }
      if(flag){
         cout<<i<<endl;
         break;
      }
   }
   return 0;
}
```

– 复杂度分析：

- 时间复杂度：二重循环，假设 T 长度是 m，P 长度是 n，则时间复杂度是 O(mn)

- 空间复杂度：有限几个变量，复杂度 O(1)

– kmp 朴素的算法当模式串 P 某个字符不匹配时，依然要从头开始和 S 的下一个字符匹配。但前一次匹配其实已经有额外的信息，kmp 算法就是利用已匹配的信息，使得下一次匹配时 S 的指针不回退，这是通过一个 next 数组实现的。

next 数组

- 当模式串和主串某一字符不匹配时，模式串要回退的位置

- next[j]：第 j 位字符前 j-1 位字符串的最长前后缀字符数 +1(因为从 1 开始)

– next 数组的求法

next 数组的求法最为关键，也就是求每个位置前面最长前后缀长度，这个当然可以暴力，但我们也可以优化，利用之前的 next 信息。

首先来看一个基本的模式串：

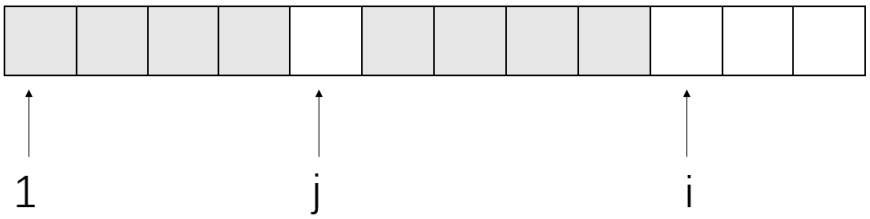


图 **11**

这是求 next 数组中的一般情形，其中 j 是前缀末尾，i 是后缀末尾，求的是 next[i+1]，也就是 i+1 前面的最长前后缀。

1. 当 P[i]==P[j] 时，显然最长前后缀长度 +1，则 nex[++i] = ++j

2. 当 P[i] != P[j] 时，显然不能以 j 为前缀末尾，但是我们已经求过 next[j]，可以证明这里就是让 j = next[j]，重复比较 P[i] 和 P[j]。

– 证明为什么当 P[i] != P[j] 时要让 j=next[j]：

1. 首先这是一种可能的相等前后缀

对于 next[j]，它是 j 前面最长公共字符 +1，下面两个蓝色块相等。

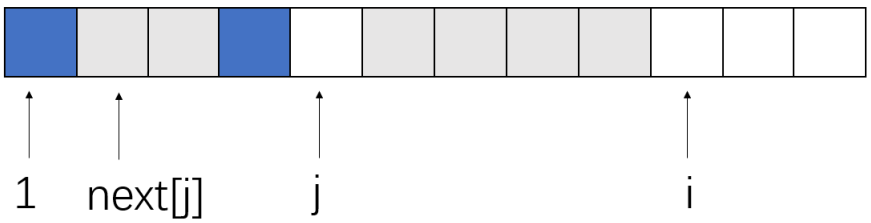


图 **12**

由于已经有 1 j-1 和 i-j+1 i-1 相等，由此可得到**下面四块子串连等 (下面四块蓝色块)**，即有 1 next[j]-1 和 i-next[j]+1 i-1 相等

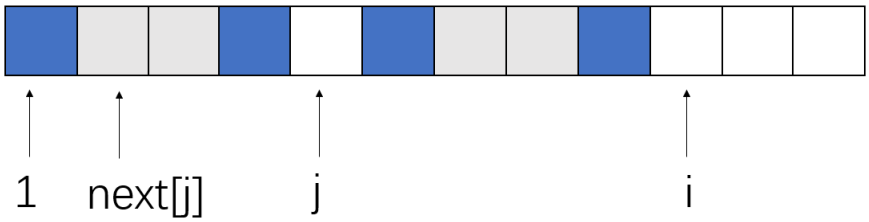


图 **13**

而如果 P[next[j]]==P[i]，则求得一个前后缀，有 next[++i]=++next[j] 也就是最长前缀以 next[j] 结尾

2. 其次这种是最长的

因为本身 next[j] 就是 j 前面最长的，如果 i+1 前面的最长前后缀比上面所求的还长，那么至少可以得到 j 前面的最长前后缀可以加 1(**根据连等的性质**)，与 next[j] 矛盾

– 以上是理解求 next 操作的一种方法，另一种更感性，我们可以将求 next 数组本身看作是将**模式串前缀字符串 (模式串) 和后缀字符串 (主串) 相匹配**的过程 (**只不过这里是求前缀的末尾索引 +1**)，当 i 和 j 不匹配时显然根据 next 本身的定义需要令 j=next[j]，也就是让 j 回退，而 i 不回退。

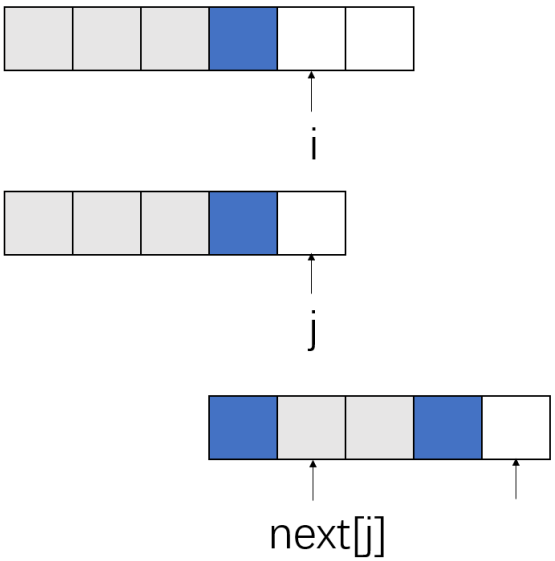


图 **14**

- 代码实现

```
#include<iostream>
```

```cpp
#include<cstdio>
#include<cstring>
#include<string>
#include<algorithm>

using namespace std;
const int N = 1e5+10;
const int M = 1e6+10;
int nex[N];
char S[M],P[N];//S是主串，P是模式串
int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   int n,m;
   cin>>n>>P+1;
   cin>>m>>S+1;

   //求nex数组
   nex[1] = 0;//第1号位置对应的nex就设为0,作为第一个字符不匹配的记号
   for(int i = 1,j = 0;i <=
      ↪ n-1;){//i从1开始，是后缀末尾，j从0开始，是前缀末尾
      if(j == 0 || P[i] == P[j])nex[++i] =
         ↪ ++j;//这里j==0包括两种情况，一是初始j=0，会让nex[2]=1(显然正确，因为2号字
         ↪ = 1; 此外，如果P[i]==P[j]，代表相等前后缀子串长度+1。
      else j = nex[j];//否则回退j
   }

   //nex的优化
   for(int i = 2;i <= n-1;i ++){
      if(P[i] == P[nex[i]])nex[i] =
         ↪ nex[nex[i]];//如果P[i]==P[nex[i]]，也就是你nex之后要比较的字符不变，那肯
   }
   //进行模式匹配
   for(int i = 1,j = 1;i <= m && j <= n;){
```

```
33      if(j ==0 || S[i] ==
            ↪ P[j]){//当P第一个字符都不匹配(j==0)或者S[i]和P[j]匹配时后移i和j
34          i++;
35          j++;
36      }
37      else j = nex[j];//否则回退j
38      if(j > n){//匹配成功
39          cout<<i-n-1<<" ";//输出下标，这里却是按0开始计
40
41          //还可以继续找可能的所有匹配成功的下标
42          i--;
43          j--;
44          j = nex[j];
45      }
46  }
47  return 0;
48 }
```

三、 基础数据结构

3.1 链表

链表是链式存储的线性表，可以是静态的，也可以是动态的。

- 动态链表的节点结构体如下：

```
1 struct ListNode
2 {
3     double value;
4     ListNode *next;
5 };
```

- 然而，这种方法如果多次 new 申请的话笔试中耗时较高，而笔试一般已知数据范围，通常使用的是数组实现的静态链表。
- 常用：通常实现的是邻接表，存储树和图

3.1.1 数组实现单链表

数组实现单链表最简单的方式就是使用两个数组，分别是每个节点的值和 next 指针，此外，增加一个 head 头指针指向链表的头，同时，增加一个 index 指向最后一个节点的后一个节点的索引，用于分配数组空间。

实现方式如下：

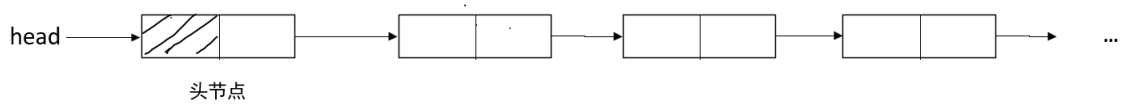


图 15

```cpp
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 1e5+10;
int head,idx,val[N],nex[N]; //0号下标记为头节点
int m;
void init(){ //初始化
   head = 0;
   nex[0] = -1;
   idx = 1;//从1开始便于对应
}
void insert(int k,int x){//在第k个序号位置后面插入x，方便操作
   val[ind] = x;
   nex[ind] = nex[k];
   nex[k] = ind++;
}
void insert_head(int x){ //头插
   insert(0,x);
}
void del(int
   ↪ k){//删除第k个序号位置后面的节点(因为是单链表，必须知道前一个节点)
```

```
    nex[k] = nex[nex[k]];
}
```

▶ 单链表

- 思路分析。

  考察单链表的数组实现，直接写，关键是处理输入。

- 代码实现

```
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 1e5+10;
int head,idx,val[N],nex[N]; //0号下标记为头节点
int m;
void init(){ //初始化
    head = 0;
    nex[0] = -1;
    idx = 1;//从1开始便于对应
}
void insert(int k,int x){//在第k个序号位置后面插入x，方便操作
    val[ind] = x;
    nex[ind] = nex[k];
    nex[k] = idx++;
}
void insert_head(int x){ //头插
    insert(0,x);
}
void del(int
    ↪ k){//删除第k个序号位置后面的节点(因为是单链表，必须知道前一个节点)
    nex[k] = nex[nex[k]];
}

int main(){
```

```
    scanf("%d",&m);
    init();
    while(m--){
       char op;
       int k,x;
       cin>>op;
       if(op=='H'){
          cin>>x;
          insert_head(x);
       }
       else if(op=='D'){
          cin>>k;
          del(k);
       }
       else{
          cin>>k>>x;
          insert(k,x);
       }
    }
    for(int i = nex[head];i != -1;i = nex[i])cout<<val[i]<<" ";
       ↪ //从第一个节点开始
    return 0;
}
```

3.1.2 数组实现双链表

- 双链表就是每个节点有两个指针，指向前一个节点和后一个节点，可以通过用两个数组 pre 和 nex 实现
- 作用：优化某些问题

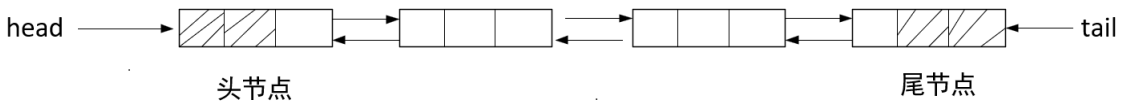


图 16

```
#include<iostream>
```

```cpp
#include<cstdio>
#include<algorithm>
#include<string>
#include<cstring>
using namespace std;

const int N = 1e5+10;
int head,val[N],pre[N],nex[N],idx; //0号作为头节点,1号作为尾节点

void init(){
   head = 0;
   pre[0] = -1;
   nex[0] = 1;
   pre[1] = 0;
   nex[1] = -1;
   idx = 2;
}
void insert(int k,int x){ //插入到第k个分配的节点的后面
   val[ind] = x;
   pre[ind] = k;
   nex[ind] = nex[k];
   nex[k] = ind;
   pre[nex[ind]] = ind;
   idx++;
}
void insert_head(int x){ //插入到头节点后
   insert(0,x);
}

void del(int k){ //删除第k个分配的节点
   nex[pre[k]] = nex[k];
   pre[nex[k]] = pre[k];
}

int main(){
```

```
    ios::sync_with_stdio(false);
    int m;
    cin>>m;
    init();
    while(m--){
        string op;
        cin>>op;
        int k,x;
        if(op=="L"){
            cin>>x;
            insert_head(x);
        }
        else if(op=="R"){
            cin>>x;
            insert(pre[1],x);
        }
        else if(op=="D"){
            cin>>k;
            del(k+1);
        }
        else if(op=="IL"){
            cin>>k>>x;
            insert(pre[k+1],x);
        }
        else{
            cin>>k>>x;
            insert(k+1,x);
        }
    }
    for(int i = nex[head];i != 1;i = nex[i])cout<<val[i]<<" ";
        ↪ //注意输出条件
    return 0;
}
```

3.2 栈

栈是一种后进先出的数据结构，类似叠盘子。

3.2.1 数组模拟栈

▶ 模拟栈

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

const int N = 1e5+10;
int s[N],top_idx = -1;

void push(int x){
   s[++top_idx] = x;
}
int pop(){
   return s[top_idx--];
}
bool empty(){
   return top_idx < 0;
}
int top(){
   return s[top_idx];
}

int main(){
   ios::sync_with_stdio(false);
   int m;
   cin>>m;
   while(m--){
      string op;
```

```
30      cin>>op;
31      int x;
32      if(op=="push"){
33         cin>>x;
34         push(x);
35      }
36      else if(op=="pop"){
37         pop();
38      }
39      else if(op=="empty"){
40         if(empty())cout<<"YES"<<endl;
41         else cout<<"NO"<<endl;
42      }
43      else cout<<top()<<endl;
44   }
45   return 0;
46 }
```

3.2.2 单调栈

单调栈是一种特殊的栈，基本模型一般是一个序列找到每个数左边最近且比它小的数。

之所以叫做单调栈，是因为这种栈里面的元素满足某种单调性。

▶ 单调栈

- 思路分析。
  - 暴力做法。最一般做法是双指针，对序列中每一个元素 i，遍历 j 从 i-1 到 0，找到比 i 小的元素输出。
  - 单调栈

    然而，如果深入挖掘问题性质，可以进一步优化。

    对于元素 i，其实不必逐个遍历左边元素，可以对搜索空间进行缩减，可以证明，只需用栈存储一个连续的递增序列，它就是搜索空间。

    证明：如果栈不是单调递增的，存在元素 i1，i2，a[i1] >= a[i2]，然而题目是找最近且小于 i 的数，如果 i2 不满足，那么 i1 更不满足，因此可以删去 a[i1]，减小搜索空间。

    此外，最关键的是在求每个元素答案时保持了栈的单调性操作。对于 a[i]，依次

比较栈顶元素，如果比 a[i] 大或相等，弹出 (对于后面的元素 a[i] 比该元素更优)；如果比 a[i] 小，它就是答案，输出，并将 a[i] 入栈 (a[i] 可能是后面元素的答案)，这样求每个元素的答案时栈的单调性不变。

总的来说，过程一般是：暴力-> 剔除元素-> 单调性-> 优化

- 代码实现

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
#include<cstdio>
using namespace std;

const int N = 1e5+10;
int s[N],top_idx = -1;

int main(){
   ios::sync_with_stdio(false); //加快输入输出
   cin.tie(0);
   cout.tie(0);
   int m,x;
   cin>>m;
   while(m--){
      cin>>x;
      while(top_idx >=0 && s[top_idx] >= x)top_idx--;
         ↪ //大于等于的都出栈
      if(top_idx >=0){
         cout<<s[top_idx]<<" ";//还有元素它就是答案
      }
      else cout<<"-1 ";
      s[++top_idx] = x;//最后入栈当前元素
   }
   return 0;
}
```

3.3 队列

队列是一种先进先出的数据结构，类似排队。

3.3.1 数组模拟队列

▶ 模拟队列

```
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 1e5+10;
int q[N],head,tail = -1;
   ↪ //数组从左边开始，head是队首，tail是队尾，从队尾插入，从队首弹出

void push(int x){
   q[++tail] = x;
}
int pop(){
   return q[head++];
}
bool empty(){
   return tail < head;
}
int top(){
   return q[head];
}

int main(){
   ios::sync_with_stdio(false);
   int m;
   cin>>m;
   while(m--){
      string op;
```

```
        cin>>op;
        int x;
        if(op=="push"){
            cin>>x;
            push(x);
        }
        else if(op=="pop"){
            pop();
        }
        else if(op=="empty"){
            if(empty()) cout<<"YES"<<endl;
            else cout<<"NO"<<endl;
        }
        else cout<<top()<<endl;
    }
    return 0;
}
```

3.3.2 单调队列

单调队列基本模型是维护一个滑动窗口，求滑动窗口的最大值/最小值。关注突出点。

▶ 滑动窗口

- 思路分析。
  - 暴力做法：求每个窗口最小值的暴力做法就是遍历这个窗口，复杂度 O(nk)
  - 单调队列：可以对窗口里面的数进行剔除，可以证明，当窗口最右边的数是 a[i] 时，可以剔除掉前面比 a[i] 大于等于的数 (它们当前不会被考虑，并且比 a[i] 先出队)，窗口实际维护的是一个单调递增的序列。
    这样，对于元素 i，如果队尾比它大于等于，就出队 (这里是双端队列)，否则入队，最终队头就是答案。
    此外，由于要保证滑动窗口的长度不变，存储索引而不是值。
    最终每个元素至多入队和出队一次，复杂度 O(n)
- 代码实现

```
#include <iostream>
```

```cpp
#include <cstring>
#include <algorithm>
#include<cstdio>

using namespace std;

const int N = 1e6+10;
int a[N];//存储序列，因为要索引
int q[N],head,tail =
    ↪ -1;//用两个滑动窗口，因为题目要求输出最小值以及最大值

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int n,k;
    cin>>n>>k;
    for(int i = 0;i < n;i++)cin>>a[i];
    for(int i = 0;i < n;i++){
        while(head<=tail&&i-q[head]+1>k)head++;//如果窗口内元素大于k，出队首
        while(head<=tail&&a[q[tail]]>=a[i])tail--;//队尾元素出队
        q[++tail]=i;//入队的是索引
        if(i>=k-1)cout<<a[q[head]]<<" ";
    }
    cout<<endl;

    head=0,tail = -1;
    for(int i = 0;i < n;i++){
        while(head<=tail&&i-q[head]+1>k)head++;//如果窗口内元素大于k，出队首
        while(head<=tail&&a[q[tail]]<=a[i])tail--;//队尾元素出队
        q[++tail]=i;//入队的是索引
        if(i>=k-1)cout<<a[q[head]]<<" ";
    }
    cout<<endl;
    return 0;
```

```
36
37 }
```

3.4 Trie

Trie 是一种高效存储和查找字符串集合的多叉树的数据结构。关键是它利用字符串的前缀进行了优化。这种多叉树是这样的

- 二叉树的节点存储的是编号 (从 0 开始，0 是根节点)，边对应字符串中的字符 (ascii 码)
- 二维数组 child[p] [c]，child [p] [c] = j，意思是节点 p 有一条 c 对应的字符可以到 j
- 计数数组 cnt[p]，代表以 p 为结尾字符串的个数
- 节点编号 idx，给节点编号，从 0 开始自增
- 一个示意图如下：

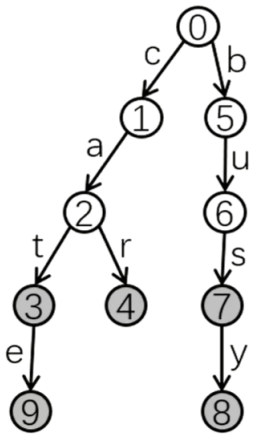


图 **17**

▶ Trie 字符串统计

- 思路分析。

  就是 Trie 的模板题关键实现两个函数，一个是插入操作，一个是查询操作
- 代码实现

```
1 #include<iostream>
2 #include<cstdio>
3 #include<algorithm>
4 #include<cstring>
```

```
5
6  using namespace std;
7
8  const int N =
       ↪ 1e5+10;//节点最大个数，边是字符，而一个字符必定指向一个节点，其实一一对应
9
10 int child[N][26],cnt[N],idx;//child二维数组,cnt计数数组,idx用于分配节点
11 char s[N];//每个字符串
12
13 void insert(char s[]){//插入字符串s
14    int p = 0;//从0号根节点开始
15    for(int i = 0;s[i] != 0;i ++){//插入每个字符
16       int c = s[i] - 'a';//取该字符对应ascii码
17       if(!child[p][c])child[p][c] = ++
             ↪ idx;//如果没有给字符c分配对应节点，那就创建
18       p = child[p][c];//p移动到子节点
19    }
20    cnt[p] ++;//代表以p节点结尾的字符串计数+1
21 }
22 int query(char s[]){//查询字符串s的个数
23    int p = 0;//同样从0号根节点开始
24    for(int i = 0;s[i] != 0 ;i++){
25       int c = s[i] - 'a';
26       if(!child[p][c])return 0;//如果没找到某个字符返回0
27       p = child[p][c];//找下一个字符
28    }
29    return cnt[p];//返回个数
30 }
31 int main(){
32    ios::sync_with_stdio(false);
33    cin.tie(0);
34    cout.tie(0);
35    int n;
36    cin>>n;
37    while(n--){
```

```
        char op;
        cin>>op>>s;
        if(op=='I')insert(s);
        else cout<<query(s)<<endl;
    }
    return 0;
}
```

▶ 最大异或对

- 思路分析

```
这道题要求求一组数中的最大的两个数的异或值。
首先最容易想的是暴力做法，遍历所有情况取最大值，但需要注意的是，由于异或满足交换律，
for(int i = 1;i <= n;i ++)cin>>a[i];
for(int i = 1;i <= n;i ++){
    for(int j = 1;j < i;j ++){
        res = max(res,a[i]^a[j]);
    }
}
然而，这道题数据范围是1e5，时间要求是1s，c++1s可以计算1e7~1e8，因此至少需要nlogn算法

优化需要深入题目本身性质，题目求得是最大异或值，异或是按二进制位异或，相同为0，不同为
而题目规定每个元素最多31位，因此我们可以求出每个数的31位的二进制表示，对每个数，从高

这道题表明Trie不仅可以存储字符串，也可以存储整数(准确来说是整数的二进制表示)
```

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>

using namespace std;

const int N = 1e5+10;
```

```cpp
int child[N*31][2],idx;//N个数，每个数31位，最多N*31个节点

void insert(int x){
   int p = 0;
   for(int k = 30;k >=0;k --){
      int b = x >> k & 1;
      if(!child[p][b])child[p][b] = ++ idx;
      p = child[p][b];
   }
}
int query(int x){
   int p = 0,val = 0;
   for(int k = 30;k >= 0;k--){
      int b = x >> k & 1;//对每一个数，从高位到低位依次取出
      if(child[p][!b]){//如果存在和当前位不同的路径，进入
         p = child[p][!b];
         val = val * 2 + 1;//更新异或值
      }
      else{
         p = child[p][b];//否则该位只能相同
         val = val * 2 + 0;//更新异或值
      }
   }
   return val;
}
int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   int n,res = 0;
   cin>>n;
   int x;
   cin>>x;
   insert(x);//先插入一个数
   for(int i = 2;i <= n;i ++){
```

```
        cin>>x;
        insert(x);//插入该数到Trie中
        res = max(res,query(x));
    }
    cout<<res;
    return 0;
}
```

3.5 并查集

并查集是一种树形的数据结构，它将每个集合用一棵树表示，每个集合用根节点作为代表元，代表元的编号就是集合的编号，每个节点存储它的父节点 p[x]，如果 p[x]=x 说明 x 是代表元。

它主要支持两种操作：

- 合并操作：将两个子集合合并成一个集合
  将一个集合的代表元的父节点改成另一个集合的父节点。
  如果给的条件是元素所在集合，那么还需要先查找元素所在集合的代表元 (这一步就需要使用下面的查找操作)
- 查找操作：确定某个元素属于哪个集合
  从该元素出发，寻找其代表元。这一步可以进行路径压缩优化，也就是在查找过程中将经过的节点的父节点修改为最终的代表元，使得之后在查找这些元素时能近乎 O(1) 的复杂度查询。

▶ 合并集合

- 思路分析。
  模板题，关键是合并和查询两个操作
- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
using namespace std;

const int N = 1e5+10;
int p[N];//存储每个编号的父节点号
```

```
9
10 void init(int n){//初始化，每个编号自成一个集合，父节点是它们自己
11    for(int i = 1;i < N;i ++)p[i] = i;
12 }
13 int find(int i){//查找编号i的元素的所在集合
14    if(p[i] != i)p[i] =
      ↪ find(p[i]);//如果不是代表元，递归寻找，并且路径压缩
15    return p[i];//返回查询结果
16 }
17 void union_set(int i,int
   ↪ j){//合并两个元素所在集合，这里直接找到代表元再改父节点，需要指出的是，
18    //尽管可以用按秩合并优化，然而路径压缩的优化效率已经足够
19    p[find(i)] = find(j);
20 }
21 int main(){
22    ios::sync_with_stdio(false);
23    cin.tie(0);
24    cout.tie(0);
25    int n,m;
26    cin>>n>>m;
27    init(n);
28    while(m--){
29       char op;
30       int x,y;
31       cin>>op>>x>>y;
32       if(op == 'M')union_set(x,y);
33       else{
34          if(find(x) == find(y))cout<<"Yes"<<endl;
35          else cout<<"No"<<endl;
36       }
37    }
38    return 0;
39 }
```

▶ 连通块中点的数量

- 思路分析。

```
依然是模板题目，只不过需要维护每个集合的元素个数
由于集合的元素个数只会在合并两个集合时发生改变，而合并集合是通过将一个集合的代表元的
```

- 代码实现

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cstdio>
#include<string>

using namespace std;

const int N = 1e5+10;
int
    ↪ p[N],s[N];//size存储每个编号所在集合的元素个数，当且仅当它是代表元才有意义

void init(int n){//初始化
   for(int i = 1;i <= n;i ++){
      p[i] = i;
      s[i] = 1;//初始每个集合都是1个元素
   }
}
int find(int i){
   if(p[i] != i)p[i] = find(p[i]);
   return p[i];
}
void union_set(int i,int j){
   int x1 = find(i);
   int x2 = find(j);
   if(x1 == x2)return;//同一个集合不重复合并
   p[x1] = x2;//由于提前取出两个代表元，因此这两行顺序任意，否则需要注意
   s[x2] += s[x1];
}

```

```
30
31 int main(){
32    ios::sync_with_stdio(false);
33    cin.tie(0);
34    cout.tie(0);
35    int n,m;
36    cin>>n>>m;
37    init(n);
38    while(m--){
39       string op;
40       cin>>op;
41       int x,y;
42       if(op=="C"){
43          cin>>x>>y;
44          union_set(x,y);
45       }
46       else if(op=="Q1"){
47          cin>>x>>y;
48          if(x == y){
49             cout<<"Yes"<<endl;
50             continue;
51          }
52          if(find(x) == find(y))cout<<"Yes"<<endl;
53          else cout<<"No"<<endl;
54       }
55       else{
56          cin>>x;
57          cout<<s[find(x)]<<endl;
58       }
59    }
60    return 0;
61 }
```

▶ 食物链

- 思路分析。

题目相对复杂一点，简单的说就是要求假话的个数，最重要的是判断当且说法是否和之前的说法

根据题意，维护动物之间的关系可以通过一棵树表示，父节点指向子节点表示子节点可以吃父节
然而，这些节点明确属于三类，因此需要对这些节点划分，一种好的方法是将第0、1、2层作为这
而计算每个节点到根节点的路径长度是通过路径压缩实现的(因为初始只知道到自己父节点的路径

- 代码实现

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>

using namespace std;

const int N = 5e4+10;
int
    ↪ p[N],dis[N];//p是并查集，dis维护的是各个动物到根节点的距离(初始自己是根节点，

int find(int
    ↪ x){//并查集的find操作寻找x属于哪个集合(代表元)，附带路径压缩以便计算到根节点距
   if(p[x] != x){
      int t = find(p[x]);//递归，返回根节点
      dis[x] += dis[p[x]];//更新距离
      p[x] = t;
   }
   return p[x];
}

int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   int n,k;
   cin>>n>>k;
```

```
    int cnt = 0;
    for(int i = 1;i <= n ;i++)p[i] = i;//初始根节点是自己
    while(k--){
        int d,x,y;
        cin>>d>>x>>y;
        if(x > n || y > n)cnt++;//先简单判断明显是假话的
        else if(d == 1){//对于是同类的说法
            int px = find(x);//先路径压缩一下计算到根节点距离
            int py = find(y);
            if(px == py && (dis[x] - dis[y]) %
                ↪ 3)cnt++;//x和y已经在同一个并查集中，但x和y到根节点距离不相等(同余意义
            else if(px !=
                ↪ py){//合并x和y所在集合，并使得它们到根节点距离相同
                p[px] = py;//将p[y]作为p[x]的根节点
                dis[px] = dis[y] -
                    ↪ dis[x];//设置p[x]到p[y]的距离，由于是在同余意义下，这里负数也不影响
            }
        }
        else{//对于x吃y的说法
            int px = find(x);//先路径压缩一下计算到根节点距离
            int py = find(y);
            if(px == py && (dis[x] - dis[y] - 1) %
                ↪ 3)cnt++;//x和y已经在同一个并查集中，但x到根节点距离没有比y多1(同余意义
            else if(px != py){//合并x和y所在集合,并使得x到根节点距离比y多1
                p[px] = py;
                dis[px] = dis[y] - dis[x] + 1;
            }

        }

    }
    cout<<cnt;
    return 0;

}
```

3.6 堆

堆，又称为优先队列 (队列中的元素有不同的优先级，有最小/最大优先级的元素)，是一种完全二叉树，并且满足父节点大于等于/小于等于两个子节点，分为大根堆和小根堆，能快速求一组数中的最大值/最小值。

手写一个堆支持以下操作 (STL 只支持前 4 种)

- 建立一个堆

可以通过逐个插入一个元素实现，然而复杂度是 O(nlogn)，如果已经一次性给定目标数组，可以从最后一个非叶子节点开始进行 down 操作实现，复杂度是 O(n)

- 插入一个元素
- 求最大/最小值
- 删除堆顶元素
- 删除任意一个元素
- 修改任意一个元素

要实现这些操作，需要两个更基本的操作

- 向上调整 up
- 向下调整 down

▶ 简单堆

- 思路分析

  模拟堆的 3 种操作

- 代码实现

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>

using namespace std;

const int N = 1e5+10;
int heap[N],heap_size;//heap存储堆，从1开始，左孩子2*i+1，右孩子2*i+2
void up(int i){//对i号位置元素进行向上调整
   int p = i/2;
   if(p >= 1 && heap[p] > heap[i]){
      swap(heap[p],heap[i]);
      up(p);
```

```
    }
}
void down(int i){//对i号位置元素向下调整
    int l = 2*i,r = 2*i+1;
    int t = i;
    if(l <= heap_size && heap[t] > heap[l])t = l;
    if(r <= heap_size && heap[t] > heap[r])t = r;
    if(t != i){
        swap(heap[t],heap[i]);//将根节点与孩子节点中的小者进行交换
        down(t);
    }
}


void create_heap(int n){//建堆
    heap_size = n;
    for(int i = n/2;i >= 1;i
        ↪ --)down(i);//从最后一个非叶子节点进行down操作建堆，这里建立小根堆
}
void insert(int x){//插入元素
    heap[++heap_size] = x;
    up(heap_size);
}
int get_top(){//返回堆顶元素
    return heap[1];
}
int remove_top(){//删除堆顶元素
    int x = heap[1];
    heap[1] = heap[heap_size--];
    down(1);
    return x;
}


int main(){
```

```
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int n,m;
    cin>>n>>m;
    for(int i = 1;i <= n;i ++)cin>>heap[i];
    create_heap(n);
    while(m--){
        cout<<remove_top()<<" ";
    }
    return 0;
}
```

▶ 模拟堆

- 思路分析。

  模拟堆的 5 种操作，需要额外 2 个数组

- 代码实现

  关键在两个数组 ih，hi

  - ih，第 k 个插入的数在堆中的下标

  - hi，堆中某个下标是第几个插入的数

  在交换堆中两个数时，同时也要修改这两个数组，图解如下

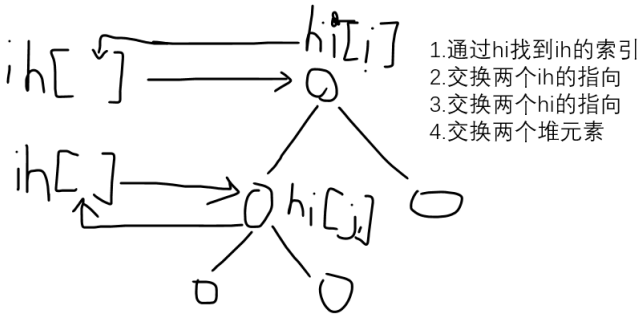


图 **18**

- 代码实现

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
```

```cpp
#include<string>

using namespace std;

const int N = 1e5+10;
int heap[N],heap_size;//heap存储堆，从1开始，左孩子2*i+1，有孩子2*i+2
int ih[N],hi[N],idx;//为了拓展功能，需要额外存储第k个插入的数在堆中的下标
//以及堆中某个下标是第几个插入的
void heap_swap(int i,int j){//额外定义一个堆交换操作
   swap(ih[hi[i]],ih[hi[j]]);//交换两个指针
   swap(hi[i],hi[j]);//交换两个指针
   swap(heap[i],heap[j]);
}
void up(int i){//对i号位置元素进行向上调整
   int p = i/2;
   if(p >= 1 && heap[p] > heap[i]){
      heap_swap(p,i);
      up(p);
   }
}
void down(int i){//对i号位置元素向下调整
   int l = 2*i,r = 2*i+1;
   int t = i;
   if(l <= heap_size && heap[t] > heap[l])t = l;
   if(r <= heap_size && heap[t] > heap[r])t = r;
   if(t != i){
      heap_swap(t,i);
      down(t);
   }
}



void create_heap(int n){//建堆
```

```
39    heap_size = n;
40    for(int i = 1;i <= n;i ++){
41       ih[i] = i;
42       hi[i] = i;
43    }
44    for(int i = n/2;i >= 1;i --){
45       down(i);//从最后一个非叶子节点进行down操作建堆，这里建立小根堆
46    }
47 }
48 void insert(int x){//插入元素
49    heap[++heap_size] = x;
50    ih[++idx] = heap_size;
51    hi[heap_size] = idx;
52    up(heap_size);
53 }
54 int get_top(){//返回堆顶元素
55    return heap[1];
56 }
57 int remove_top(){//删除堆顶元素
58    int x = heap[1];
59    heap_swap(1,heap_size--);
60    down(1);
61    return x;
62 }
63 int remove(int k){//删除第k个插入的数
64    int index = ih[k];//先求下标
65    int x = heap[index];
66    heap_swap(index,heap_size--);
67    up(index);
68    down(index);
69    return x;
70 }
71 void modify(int k,int x){//修改第k个元素为x
72    int index = ih[k];//先求下标
73    heap[index] = x;
```

```
74      up(index);
75      down(index);
76  }
77
78
79
80
81  int main(){
82      ios::sync_with_stdio(false);
83      cin.tie(0);
84      cout.tie(0);
85      int m;
86      cin>>m;
87      while(m--){
88          string op;
89          cin>>op;
90          int k,x;
91          if(op == "I"){
92              cin>>x;
93              insert(x);
94          }
95          else if(op == "PM"){
96              cout<<get_top()<<endl;
97          }
98          else if(op == "DM"){
99              remove_top();
100         }
101         else if(op == "D"){
102             cin>>k;
103             remove(k);
104         }
105         else{
106             cin>>k>>x;
107             modify(k,x);
108         }
```

```
    }
    return 0;
}
```

3.7 哈希表

哈希表，一种映射表，它利用哈希函数将原有数据映射 (映射后通常是数组的索引，这样可以快速定位到相应的值)，由于哈希函数的特性，可以实现平均 0(1) 的复杂度进行查找、插入和删除操作。

哈希函数的性质如下：

- 同一数据哈希后的值相同
- 不同数据哈希后的值不同

当然，即使哈希冲突的概率理论上很低，但依然可能发生，在发生哈希冲突时，根据存储结构有不同的方式解决。

3.7.1 整数哈希

▶ 模拟散列表

- 哈希函数：整数哈希通常是通过模一个大于预计数据区间长度的质数 (越远离 2 的次幂越好)
- 存储方式：通常有拉链法和开放寻址法两种方式
- 哈希冲突：不同的存储方式冲突解决不同

拉链法

- 算法思想

  存储方式：拉链法就是首先用一个槽对应不同哈希值，每个哈希值下面接着一条链，存储对应哈希值的不同数据，并且算法题中为了效率通常用数组模拟，索引：哈希值，链：哈希值对应的原数据，图如下：

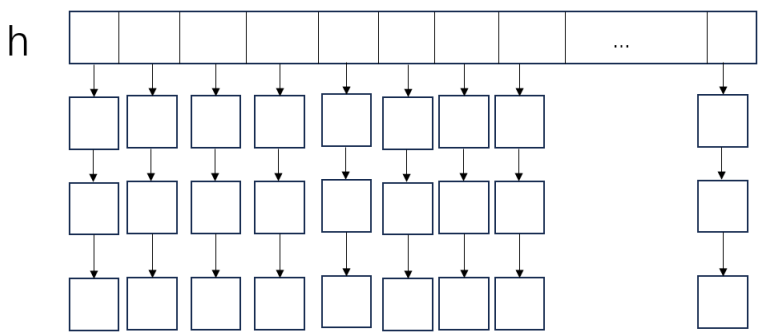


图 19

哈希冲突：当遇到冲突时直接利用头插法插入到相应哈希值下的链即可，同时注意如果要删除的话通常不会直接删除对应节点，而是打上标记。

- 代码实现

```cpp
#include<iostream>
#include<algorithm>
#include<cstring>
#include<cstdio>

using namespace std;

//拉链法实现哈希表
const int N = 1e5+3;//既是数组的长度，也是模数
int h[N];//存储不同哈希值对应链的头节点的位置
int val[N],nex[N],idx;//数组链表，实际上是将其拆成不同链


void insert(int x){
   int t = (x % N + N) %
      ↪ N;//哈希函数，c++中负数取模还是负数，所以需要这样
   val[idx] = x;
   nex[idx] = h[t];
   h[t] = idx++;
}

bool query(int x){
   int t = (x % N + N) %
      ↪ N;//哈希函数，c++中负数取模还是负数，所以需要这样
   for(int i = h[t];i != -1;i = nex[i]){
      if(val[i] == x)return true;
   }
   return false;
}
int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
```

```
    cout.tie(0);
    int n;
    cin>>n;
    memset(h,-1,sizeof(h));//将h数组初始化为-1，也就是空链
    while(n--){
        char op;
        int x;
        cin>>op>>x;
        if(op == 'I')insert(x);
        else{
            if(query(x))cout<<"Yes"<<endl;
            else cout<<"No"<<endl;
        }

    }
    return 0;
}
```

开放寻址法

- 算法思想

  - 存储方式

  相比拉链法每个槽拉一条链，开放寻址法是利用一个长度是预计存储的数据范围区间的 2 3 倍的数组直接存储不同哈希值对应的原始数据。索引：哈希值，值：哈希值对应的原数据

  - 哈希冲突

  发生哈希冲突时向后移直到有空位为止再填入，并且移动到末尾后要返回最前端 (整个过程一定有空位)

- 代码实现

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
#define INF 0x3f3f3f3f //大于10^9，作为无穷大

using namespace std;
```

```cpp

const int N = 2e5+3;//N比实际要存储的数据个数大2~3倍，并且要是个质数
int h[N];//只用一个数组存储

void insert(int x){
   int t = (x % N + N) % N;//哈希函数
   while(h[t] != INF && h[t] !=
      ↪ x){//寻找一个没有被占的位置插入,加后面这个只是为了和查询统一(相比拉链法还去
      t = (t + 1) % N;//注意是模N意义下
   }
   h[t] = x;
}

bool query(int x){
   int t = (x % N + N) % N;
   while(h[t] != INF && h[t] != x){//跳过其他存储的数据
      t = (t + 1) % N;//注意是模N意义下
   }
   return h[t] == x;
}


int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   int n;
   cin>>n;
   memset(h,0x3f,sizeof(h));//先初始化h每个字节为0x3f(避免和要插入的数据冲突)
   while(n--){
      char op;
      int x;
      cin>>op>>x;
      if(op == 'I')insert(x);
      else{
```

```
        if(query(x))cout<<"Yes"<<endl;
        else cout<<"No"<<endl;
      }
   }

   return 0;
}
```

3.7.2 字符串哈希

字符串哈希就是用哈希表存储字符串，便于快速存储和查找字符串。

- 算法思想

  - 哈希函数

  整数哈希是用取模的方式得到数组的下标，而字符串需要先转换成数字，通常我们会先计算一个字符串的所有前缀的哈希值，进而计算任意一个子串的哈希值。

  哈希值的计算：将字符串看成一个 P 进制数 (P 取 131 或 133331)，而每个字符以它的 ascii 码作为基数，计算该字符串的十进制数值作为哈希值。（当然，可能会发生溢出，因此使用 unsigned long long 存储，溢出后会从 0 开始，相当于模 2^64)

  - 存储方式

  用一个数组存储目标字符串的所有前缀的哈希值，索引是前缀的字符串的长度，值是哈希值，并且 h[0] = 0，这样计算任意一个子串 [l,r] 的哈希值公式如下（为了方便，字符串 s 从 1 开始存储):

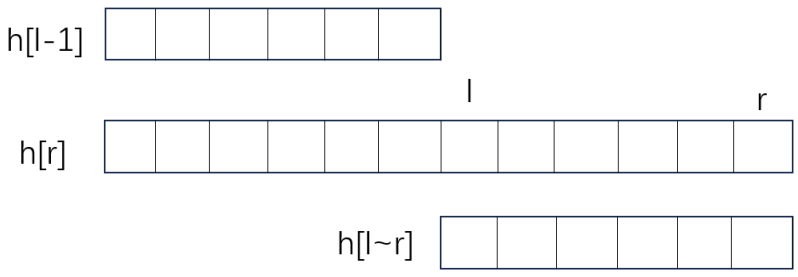


图 **20**

```
h[l-1] = s[1]*p^{l-2} + s[1]*p^{l-3}+...+s[l-1]*p^{0} \\
h[r] = s[1]*p^{r-1}+s[1]*p^{r-2}+...+s[r]*p^{0}\\
h[l:r] = s[l]*p^{r-l}+s[l+1]*p^{r-l-1}+...+s[r]*p^{0}\\
```

```
h[l:r] = h[r] - h[l-1]*p^{r-l+1} \\
思想：h[l-1]最后一个数的权是p^{0},h[r]除去h[l:r]最后一个数的权是p^{r-l+1}
    ↪ \\
因此将h[l-1]的权增大p^{r-l+1}，从而让h[r]减去前l-1位对应的数值
```

-哈希冲突

经验表明，当 P 取 131 或 133331 时，哈希冲突的概率极低，也就是说一个字符串和一个哈希值一一对应，这样我们实际上可以用哈希值代表字符串 (这和整数直接哈希不同)，查找和插入实际操作的是这个哈希值，由此可实现字符串的比较相等等操作 (很多情况下比 kmp 更方便)。

- 代码实现

```cpp
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>

using namespace std;

typedef unsigned long long ull;//使用无符号整数，并起别名
const int N = 1e5 + 10, P = 131;
char s[N];//字符串
ull h[N],p[N];//h存储字符串所有前缀的哈希值，p存储P在不同位的权重值

void init(int n){//计算所有前缀的哈希值
   p[0] = 1;//p^0 = 1
   for(int i = 1; i <= n; i++){
      h[i] = h[i-1] * P + s[i];//前缀计算
      p[i] = p[i-1] * P;//同时计算不同位的权重，方便后面
   }
}
ull get(int l, int r){//计算[l,r]对应字符串的哈希值
   return h[r] - h[l - 1] * p[r - l + 1];
}


```

```
int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   int n,m;
   cin>>n>>m;
   cin>>s+1;
   init(n);
   while(m--){
      int l1,r1,l2,r2;
      cin>>l1>>r1>>l2>>r2;//比较[l1,r1]和[l2,r2]字符串是否相等实际上就是比较它们的哈
      if(get(l1,r1) == get(l2,r2))cout<<"Yes"<<endl;
      else cout<<"No"<<endl;
   }
   return 0;

}
```

四、 搜索和图论

4.1 DFS

深度优先搜索，很执着，朝一条路径走完，不能走了再回溯，用系统栈和递归实现，相比 BFS 优点是空间少。

实现 DFS 关键是确定解决问题的顺序，同时注意回溯时恢复现场。

▶ 排列数字

- 思路分析。

  顺序：对每一个位置枚举所有可能的数字，并且从前往后枚举每个位置

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

```

```cpp
bool used[10];//记录可用的数字
int a[10];//存储摆放结果
int n;
void dfs(int u){//对于当前位置
   if(u > n){
      for(int i = 1; i <= n; i++)cout<<a[i]<<" ";
      cout<<endl;
      return;
   }
   for(int i = 1; i <= n; i++){//从前往后遍历所有可用数字
      if(!used[i]){
         used[i] = true;
         a[u] = i;//放置该数字
         dfs(u+1);//考虑下一个位置
         used[i] =
            ↪ false;//回溯回来时恢复现场(因为分支结束需要释放可用数字)
         a[u] = 0;
      }
   }
}


int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   cin>>n;
   dfs(1);
   return 0;
}
```

▶ n-皇后问题

- 思路分析。

```
同样可以用dfs
方法一:暴力
```

```
枚举每个位置是否可以放皇后，左下开始向右，超出就上一行逐个格子判断(上面这种是一次跨一

方法二:优化
实际上一行只需考虑一个位置
我们可以考虑每行皇后放在哪一列，并且从前往后枚举所有行
由于要保证不冲突条件，我们可以在枚举时候判断，进行剪枝
```

- 代码实现

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>

using namespace std;
int n;
bool
    ↪ row[10],col[10],dg[20],bdg[20];//判断行、列、正对角线、反对角线是否冲突
char g[10][10];//存放皇后的拜访位置

void dfs(int r){//考虑当前r行皇后放在哪一列
   if(r > n){//当r大于n时说明已经放完，是一种方案
      for(int i = 1; i <= n; i++){
         for(int j = 1; j <= n; j++){
            cout<<g[i][j];
         }
         cout<<endl;
      }
      cout<<endl;
   }
   for(int c = 1; c <= n; c++){//找一个可以放置的列位置
      if(!col[c] && !dg[c - r + n] && !bdg[c +
          ↪ r]){//不能列、正对角线(由于可能是负数，整体+n)、反对角线冲突
         g[r][c] = 'Q';//放置该皇后
         col[c] = true;
         dg[c - r + n] = true;
```

```
26          bdg[c + r] = true;
27
28          dfs(r + 1);//考虑下一行
29
30          g[r][c] = '.';
31          col[c] = false;
32          dg[c - r + n] = false;
33          bdg[c + r] = false;
34       }
35    }
36
37
38 }
39
40 int main(){
41    ios::sync_with_stdio(false);
42    cin.tie(0);
43    cout.tie(0);
44    cin>>n;
45    for(int i = 1; i <= n; i++){//初始化g数组为.
46       for(int j = 1; j <= n; j++){
47          g[i][j] = '.';
48       }
49    }
50    dfs(1);
51    return 0;
52 }
```

4.1.1 剪枝优化

- 可行性剪枝：当前节点的解非法或者加上剩余的搜索子树不可能有解，提前返回
- 最优性剪枝：通常我们的目标是寻找一个最优解，如果当前节点的解的最优性已经超过或者加上剩余的搜索子树的最优性小于等于已知的最优解，那么可以提前返回。
- 贪心剪枝：尽可能先枚举可能性较大的，比如通过排序、调整 dfs 的顺序等实现。
- 深度限制剪枝 (必有)：一般是搜索完了一条路径或者达到上限情况下，提前返回。

- 双向 DFS(相对复杂)：牺牲空间换时间，先对一半情况进行 DFS，哈希表存储结果，再对后一半进行 DFS，结合前一半结果判断。
- 对称性剪枝：问题具有对称性，只需搜索一条路径，另外一条提前返回。
- 其他：根据题目确定。

▶ 买瓜

这道题是一道经典的 DFS 问题。

- 思路分析

  题目要求切最少的刀，从而买到目标重量的瓜，从题目数据范围 n=30 可知，大概是用 dfs 枚举实现，初步估计每个瓜有三种情况：拿整个瓜，拿一半的瓜，不拿，这样，复杂度是 $O(3^n)$，很明显，需要剪枝。

  剪枝优化

  剪枝优化有很多方法，有些是基本的，有些需要根据题目确定，这道题的剪枝方法如下：

  - 可行性剪枝：我们要使得最终的重量等于目标重量，那么，如果当前的重量 w 大于 m，或者剩下的瓜的总重量 (这里要求个后缀和) 小于 m - w，不存在解，直接返回。
  - 最优性剪枝：这道题是求最少方案，如果当前的切瓜次数大于等于已知的最少次数，那么可以提前返回 (当然，对于剩下的搜索子树判断是否会有更优的解貌似不容易)。
  - 贪心剪枝：这里我们可以贪心，也就是将 n 个瓜的重量从大到小排序，先枚举重量大的瓜，使得我们开始搜索的节点尽量少些，尽快达到最优解，并且可以提前剪枝。
  - 深度限制剪枝：我们只有 n 个瓜，因此搜索完这 n 个瓜后必须要返回
  - 注：使用以上剪枝后在 AcWing 上只能过 8/10 的数据，如果要全部通过需要双向 DFS

- 代码实现

```
#include<bits/stdc++.h>
#define int long long
#define INF 0x3f3f3f3f3f3f3f3f

using namespace std;
int n, m;
int res = INF;
double a[50];
```

```cpp
double s[50];

void dfs(int u, double w, int c){
   if(w == m){//更新答案，注意也要提前返回
      res = min(res, c);
      return;
   }
   if(w > m || m - w > s[u] || c >= res || u >= n)return;//剪枝
   dfs(u + 1, w + a[u], c);//选当前瓜
   dfs(u + 1, w + a[u] / 2.0, c + 1);//选当前瓜并劈一半
   dfs(u + 1, w, c);///不选当前瓜
}
bool cmp(int x, int y){
   return x > y;
}
signed main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   cin>>n>>m;
   for(int i = 0; i < n; i ++){
      cin>>a[i];
   }
   sort(a, a + n, cmp);//先从大到小排序，贪心优化，进一步剪枝
   for(int i = n - 1; i >= 0; i --){//后缀和数组，辅助剪枝
      s[i] = a[i] + s[i + 1];
   }

   dfs(0, 0.0, 0);
   if(res == INF)cout<<"-1";
   else cout<<res;

   return 0;
}
```

4.2 BFS

宽度优先搜索，一层一层，队列和迭代实现，相比 DFS 所需空间大，但对于一些特殊的问题比如求宽度、” 最短路径”(第一次搜索到的路径一定是最短的，但要求边权重相同) 有用。

▶ 走迷宫

- 思路分析。

  需要求解 (1,1) 到 (n,m) 的最短路径长度可以使用 BFS，从起点 (1,1) 开始搜索，用队列存储，每次搜索直接相通的点，进而更新到 (1,1) 的最短距离 (因为是第一次遍历)，最后输出结果即可

  当然也可以用 DFS 暴力，DFS 只能求解到达 (n,m) 的路径，由于需要最短，因此还需要对所有的路径取 min, 但当 n 和 m 很大时会超时，不推荐

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
#include <queue>


using namespace std;



typedef pair<int, int> PII;
const int N = 110;

int g[N][N];//存储图
int dis[N][N];//存储路径长度
//PII q[N * N];//数组实现队列
queue<PII> q;//STL队列
int n,m;


int bfs(){//bfs求到(1,1)的最短路径
```

```
23    memset(dis, -1, sizeof(dis));//首先将dis初始化为-1
24
25    q.push({1,1});//入队点(1,1)
26    dis[1][1] = 0;//初始(1,1)到(1,1)的距离是0
27
28    int dx[4] = {-1, 1, 0, 0};//提前用数组存储偏移量，这样就不用写4个循环
29    int dy[4] = {0, 0, -1, 1};
30    while(!q.empty()){
31       auto t = q.front();
32       q.pop();
33       for(int i = 0; i < 4; i++){//找可以过的点
34          int x = t.first + dx[i];
35          int y = t.second + dy[i];
36          if(x >= 1 && x <= n && y >= 1 && y <= m && !g[x][y] &&
               ↪ dis[x][y] == -1){//不冲突就更新距离并入队
37             dis[x][y] = dis[t.first][t.second] + 1;
38             q.push({x, y});
39
40          }
41       }
42
43    }
44    return dis[n][m];
45
46
47 }
48 int main(){
49    ios::sync_with_stdio(false);
50    cin.tie(0);
51    cout.tie(0);
52    cin>>n>>m;
53    for(int i = 1; i <= n; i++)
54    for(int j = 1; j <= m; j++)
55    cin>>g[i][j];
56    cout<<bfs();
```

```
    return 0;
}
```

4.3 树和图的存储

- 树是一种特殊的图，因此只考虑图的存储。
- 无向图是特殊的有向图，只需要有向图多存一条边

通常用邻接表存储，每个点拉出来一个单链表，存储这个点能走到哪个点

```
#include<iostream>
#include<algorithm>
#include<cstdio>
#include<cstring>

using namespce std;

const int N = ;
int h[N],val[N * N],nex[N * N],idx;

void add(int a, int b){//添加边a—>b
    val[idx] = b, nex[idx] = h[a], h[a] = idx++;
}


int main(){
    memset(h, -1, sizeof(h));
    return 0;
}
```

4.4 树和图的遍历

树是一种特殊的图，因此只考虑图的遍历

4.4.1 深度优先

```
#include<iostream>
```

```cpp
#include<algorithm>
#include<cstdio>
#include<cstring>

using namespce std;

const int N = ;
int h[N],val[N * N],nex[N * N],idx;
bool vis[N];

void add(int a, int b){//添加边a—>b
   val[idx] = b, nex[idx] = h[a], h[a] = idx++;
}

void dfs(int u){//深度优先搜索
   vis[u] = true;//先搜索当前节点
   for(int i = h[u]; i != -1; i =
      ↪ nex[i]){//对周围没有搜索到的节点再递归搜索
      int v = val[i];
      if(!vis[v])dfs(v);
   }
}
void bfs(){//宽度优先搜索


}


int main(){
   memset(h, -1, sizeof(h));
   return 0;
}
```

▶ 树的重心

- 思路分析

目前只根据图的情况貌似没有方法直接判断哪个是树的重心

可以考虑直接计算删去每个点后的最大连通块的点数，然后在这些点里取最小的就是答案

删去每个点后图变成了几个连通块，要求各个连通块的点数可以使用DFS求解
这里DFS可以从一个点出发按照深度优先遍历每一个点，同时由于可以回溯因此可以计算出以当前

DFS函数功能：
函数体:可以获得当前节点的所有子树的节点个数，由此可以求得各个连通块的点个数，这些求ma
返回值:当前节点为根节点的子树的节点个数
也就是说，返回值不仅用来累加下一层的返回值，也用来更新答案

图解如下：

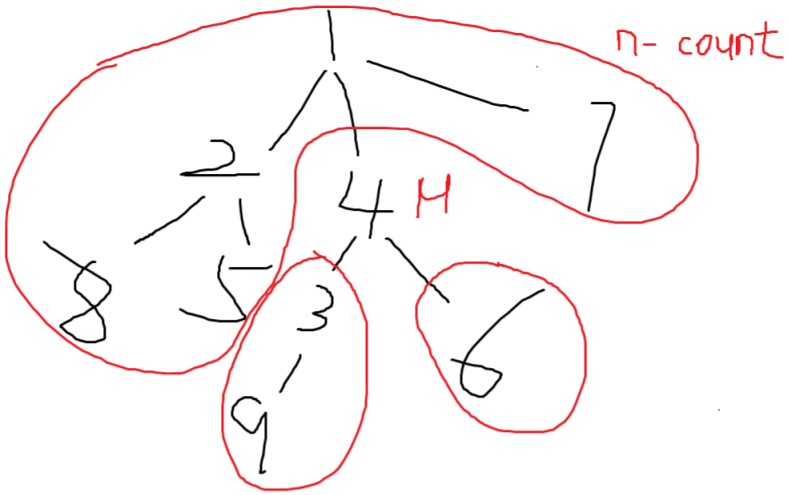


图 **21**

- 代码实现

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<cstdio>

using namespace std;

const int N = 2e5 +
```

```
    ↪ 10;//由于这颗树的边是无向的，况且没有指明根节点，开两倍大小
int n;
int h[N], val[N], nex[N], idx;//存储图的数组
bool vis[N];
int res = N;

void add(int a, int b){//加边
   val[idx] = b, nex[idx] = h[a], h[a] = idx++;
}
void create(){//构建图
   memset(h, -1, sizeof(h));
   for(int i = 1; i < n; i++){
      int a, b;
      cin>>a>>b;
      add(a, b);//两条边都加，这样任何一个节点作为根节点都可以
      add(b, a);
   }
}

int dfs(int u){//求出以当前节点为根节点的子树的节点个数(回溯时需要用到)

   vis[u] = true;//防止重复访问

   int count = 1;
   int max_count = 0;
   for(int i = h[u]; i != -1; i = nex[i]){//对于所有出边递归搜索
      if(!vis[val[i]]){
         int t = dfs(val[i]);//每次求出一棵子树的节点个数
         count +=
            ↪ t;//累加一下以u为根节点的整棵子树的节点个数(后面需要返回，同时还要计算
         max_count = max(max_count,
            ↪ t);//以val[i]为根节点的子树能否更新最大连通块点个数
      }
   }
   //cout<<u<<endl;
```

```
    max_count = max(max_count, n -
        ↪ count);//删除当前节点u后，另一边连通块点个数是n -
        ↪ count，到此更新max_count完成
    res = min(res,
        ↪ max_count);//更新最终结果res，也就是最小的最大连通块点个数
    return count;//返回以当前节点为根节点的子树的节点个数(回溯时需要用到)
}



int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n;
    create();//构建图
    dfs(1);
    cout<<res;
    return 0;
}
```

4.4.2 宽度优先

▶ 图中点的层次

- 思路分析。

  模板题，图边的权重都是 1，因此 BFS 第一次遍历到的就是最短路径。

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
#include <queue>

using namespace std;

```

```cpp
const int N = 1e5 + 10;

int h[N], val[N], nex[N], idx;
int dis[N];//存储各个点到1号点的距离
bool vis[N];
int n, m;


void add(int a, int b){
   val[idx] = b, nex[idx] = h[a], h[a] = idx++;
}

int bfs(){

   queue<int> q;//队列
   q.push(1);//首先压入起始点1
   vis[1] = true;
   dis[1] = 0;
   while(!q.empty()){
      int t = q.front();
      q.pop();


      for(int i = h[t]; i != -1; i = nex[i]){
         int p = val[i];
         if(!vis[p]){//当然这里也可以用dis判断是否遍历过
            vis[p] = true;
            dis[p] = dis[t] + 1;//第一次搜索到是最短的，更新距离
            q.push(p);
         }
      }

   }
   return dis[n];
}
```

```

int main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   cin>>n>>m;

   memset(h, -1, sizeof(h));
   memset(dis, -1, sizeof(dis));
   while(m--){
      int a, b;
      cin>>a>>b;
      add(a, b);
   }
   cout<<bfs();

   return 0;
}
```

▶ 八数码

- 思路分析。

```
乍一看，这道题很难，也和BFS没什么关系
但是，题目要求的是最小移动步数，同时一次只能移动一步。
如果这样看，每种状态(图)都是一个点，如果可以从一种状态一次移动
到另一种状态那么就有一条边，而我们
最终要求的就是到最终目标状态点的距离，这样就可以用BFS逐层遍历邻接点，
直到第一次遍历到最终点路径长度就是答案

但这道题更难的是在于如何实现，如何描述每种状态，以及如何记录到初状态的距离
(简单的数组显然是不够的)
一个简单的表示是用字符串存储每种状态，用哈希表记录各状态到初始状态的距离，
然后一维字符串和二维表根据坐标进行转换
```

- 代码实现

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
#include <string>
#include <queue>
#include <unordered_map>


using namespace std;


queue<string> q;//队列中的元素是字符串，对应每种状态
unordered_map<string, int>
    ↪ dis;//哈希表存储各个状态到初始状态的最小操作次数

int bfs(string s){//宽度优先求最短步长，s是初始状态
   q.push(s);//初始化队列和哈希表
   dis[s] = 0;

   while(!q.empty()){//队列不为空
      auto t = q.front();//取出队首元素
      q.pop();

      int d = dis[t];
      if(t == "12345678x") return d;//如果最终状态已经遍历过了，提前返回

      //扩展所有的邻点
      int index = t.find('x');//首先找到x在字符串中的下标
      int x = index / 3, y = index % 3;//转换成二维坐标
      int dx[4] = {-1, 0, 0, 1}, dy[4] = {0, 1, -1,
          ↪ 0};//用向量存储4个方向的偏移量
      for(int i = 0; i < 4; i++){
         int x1 = x + dx[i], y1 = y + dy[i];
         if(x1 >= 0 && x1 <= 2 && y1 >=0 && y1 <=
```

```
            ↪ 2){//将x和周围可能的数进行交换
34          swap(t[index], t[x1 * 3 + y1]);//交换
35          if(!dis.count(t)){//如果交换后的状态没有更新过，那么就更新，同时入队
36             dis[t] = d + 1;
37             q.push(t);
38          }
39          swap(t[index], t[x1 * 3 +
               ↪ y1]);//注意还原回来，因为可能有其他方向的交换
40       }
41    }
42 }
43
44 return -1;//没找到就返回-1
45
46 }
47
48
49 int main(){
50    ios::sync_with_stdio(false);
51    cin.tie(0);
52    cout.tie(0);
53    string s;
54    for(int i = 0; i < 9; i++){
55       char c;
56       cin>>c;
57       s += c;
58    }
59    cout<<bfs(s);
60
61    return 0;
62
63 }
```

▶ 拓扑序列

1. 思路分析。

拓扑序列就是说序列满足边的起点在前，终点在后每次取出一个入度为 0 的点，并减去它的出点的入度

2. 代码实现

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<cstdio>
#include <queue>

using namespace std;

const int N = 1e5 + 10;
int in[N];//存储各个点的入度
int res[N], length;//存储拓扑序列
int h[N], val[N], nex[N], idx;
int n, m;
void add(int a, int b){
   val[idx] = b, nex[idx] = h[a], h[a] = idx++;
   in[b] += 1;
}

void bfs(){
   queue<int> q;
   for(int i = 1; i <= n; i++){
      if(!in[i])q.push(i);//将所有入度是0的点入队
   }
   while(!q.empty()){
      int t = q.front();
      q.pop();
      res[++length] = t;

      for(int i = h[t]; i != -1; i = nex[i]){
         int p = val[i];
         in[p] --;
         if(!in[p]){
```

```
            q.push(p);
        }
    }
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m;
    memset(h, -1, sizeof(h));
    while(m--){
        int a, b;
        cin>>a>>b;
        add(a,b);
    }

    for(int i = 1; i <= n; i ++){
        bfs();
        if(length < n)cout<<-1;//注意如果所有点不能构成拓扑序列输出-1
        else{
            for(int i = 1; i <= n; i++)cout<<res[i]<<" ";
        }

        break;
    }

    return 0;
}
```

4.5 最短路径

这类问题大概思路图如下：

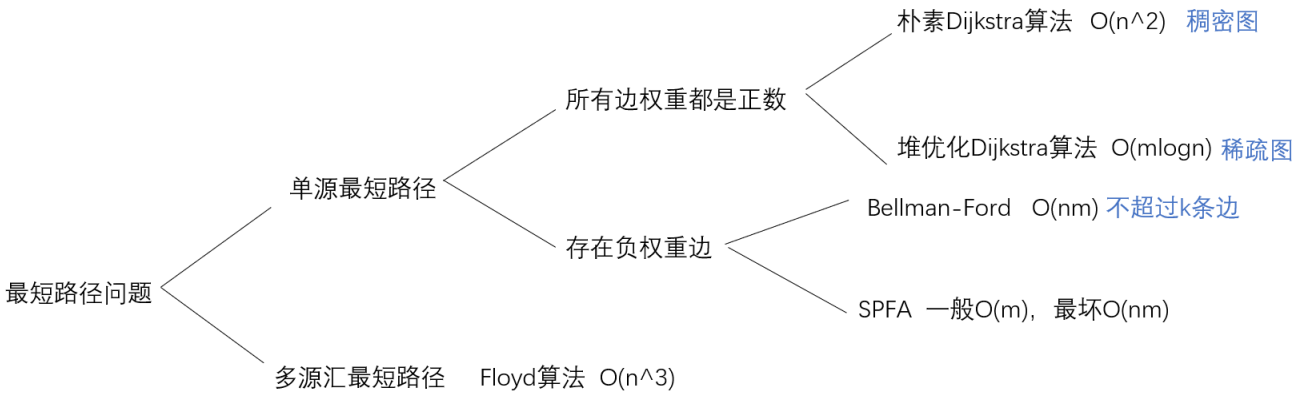


图 22

4.5.1 单源最短路径

朴素 Dijkstra ▶ Dijkstra 求最短路 I

- 思路分析。

```
朴素的Dijkstra算法是贪心的思想，其思路如下
1.初始化:假设V是所有点集，S是已经确定最短路径的点的集合，初始是空集，dis存储各个点到
2.循环执行:将V-S中距离最小的点加入到S中，同时看能否通过该点更新起点到V-S中其他点的距
3.最终的dis就存储了起点到其他所有点的距离
```

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>

using namespace std;

const int N = 510;
int n,m;
int g[N][N];//由于是稠密图，因此用邻接矩阵存储
int dis[N];//存储各个点到起点的距离
bool vis[N];//存储已经更新过的点，初始都没 更新

int Dijkstra(){//朴素版本的Dijkstra算法
```

```
    memset(dis, 0x3f, sizeof(dis));//其他点到起点的距离是正无穷
    dis[1] = 0;//起点到它自己的距离是0

    for(int i = 1; i <= n; i++){//进行n次更新
        int min_index = -1;
        for(int j = 1; j <= n ; j++){//选出未更新过的距离最小的点
            if(!vis[j] && (min_index == -1 || dis[j] < dis[min_index])){
                min_index = j;
            }
        }
        vis[min_index] = true;//加入到确定距离的集合中
        if(min_index == n)break;//可以提前退出，因为我们的目标就是n
        for(int j = 1; j <= n; j++){//更新到其他点的距离
            if(!vis[j])dis[j] = min(dis[j], dis[min_index] +
                ↪ g[min_index][j]);
        }

    }

    if(dis[n] == 0x3f3f3f3f)return
        ↪ -1;//如果dis没有更新过，路径都不存在，返回-1
    else return dis[n];
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m;
    memset(g, 0x3f, sizeof(g));//初始设定边长是无穷大
    while(m--){
        int a, b, w;
        cin>>a>>b>>w;
        g[a][b] = min(g[a][b], w);//由于可能有重边，取最小的
    }
```

```
48
49    cout<<Dijkstra();
50    return 0;
51
52 }
```

堆优化 Dijkstra

▶ Dijkstra 求最短路 II

- 思路分析

```
1 这是Dijkstra算法的堆优化
2 朴素算法中，在V-S中寻找到源点距离最短的点需要遍历整个数组，复杂度O(n)。
3 可以使用小根堆进行存储，其中的元素是每个点到源点的距离及其对应的点(因为我们要寻找到最
4 而还要更新和寻找到的点的邻点的距离，需要修改堆中元素，复杂度O(logn)，有m条边，总共O(
5
6 需要注意的是，由于需要修改堆中元素(我们需要根据扩展的点修改最短距离)，因此理论上需要
7
8
9 相比朴素版本的O(n^2)，Dijkstra算法通常更好，当然如果是稠密图,m接近n^2的量级，那么使用
```

- 代码实现

```
1  #include<iostream>
2  #include<algorithm>
3  #include<cstring>
4  #include<cstdio>
5  #include <queue>
6
7
8  using namespace std;
9
10 typedef pair<int, int> PII;
11
12 const int N = 1e6 + 10;
13 int n, m;
14 int h[N], val[N], w[N], nex[N], idx;//因为是稀疏图，所以用邻接表存储，
```

```
    ↪ 其中w存储边的权重
int dis[N];//存储各个点到源点的距离
int vis[N];

void add(int a, int b, int z){
   val[idx] = b, nex[idx] = h[a], w[idx] = z, h[a] = idx++;
}

int Dijkstra(){//堆优化版本的Dijstra算法
   priority_queue<PII, vector<PII>, greater<PII>>
      ↪ min_heap;//创建小根堆，其中的元素是(距离，点)
   min_heap.push({0, 1});
   memset(dis, 0x3f, sizeof(dis));
   dis[1] = 0;//初始化1号点到它自己的距离是0

   while(!min_heap.empty()){
      auto t = min_heap.top();
      min_heap.pop();

      int v = t.second, d = t.first;//每次在V-S中选出一个距离最短的点
      if(vis[v])continue;//由于我们之后会往堆中插入重复点，因此如果该点已经在S中，就
      vis[v] = true;//否则加入S中

      for(int i = h[v]; i != -1; i = nex[i]){
         int point = val[i], weight = w[i];//取出邻点和邻边
         if(!vis[point] && dis[point] > dis[v] + weight){
            dis[point] = dis[v] + weight;//更新距离
            min_heap.push({dis[point],
               ↪ point});//堆中的距离也要更新，通过插入新元素的方式
         }
      }

   }

   if(dis[n] == 0x3f3f3f3f)return -1;
```

```
    else return dis[n];
}


int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m;
    memset(h, -1, sizeof(h));
    while(m--){
        int a, b, z;
        cin>>a>>b>>z;
        add(a, b, z);
    }
    cout<<Dijkstra();

    return 0;

}
```

Bellman-Ford

▶ 有边数限制的最短路

- 思路分析。

```
Belllman-Ford算法非常简单
初始化dis[1] = 0, dis[other] = 无穷大
然后外层循环n次，内层循环遍历所有边(a,b,w)，更新dis[b] = min(dis[b],
    ↪ dis[a] + w)，看能否缩短最短距
离
循环n次代表的实际意义是从源点经过不超过n条边到各点的最短距离

由于这种更新策略，因此最终满足dis[b] <= dis[a] +
    ↪ w，可以处理含负权重的边，但不一定能处理负环(负环不在目标路径上依然存在最短路径
因为第n次循环如果还更新了距离说明有一条路径是n条边，经过了n+1个点，必然有两个点重合，

```

当然，其实一般存在负权边的情况多用SPFA，但在有边数限制的情况下用Bellman-Ford

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>

using namespace std;

const int N = 1e5 + 10;

int n, m, k;
int h[N], val[N], wei[N], nex[N], idx;
int dis[N];//存储各个点到源点的距离
int backup[N];//备份，用来存储每次迭代前的状态

void add(int a, int b, int z){
   val[idx] = b, wei[idx] = z, nex[idx] = h[a], h[a] = idx++;
}

int Bellman_Ford(){
   memset(dis, 0x3f, sizeof(dis));
   dis[1] = 0;
   for(int i = 0; i < k; i++){//由于题目要求不超过k条边，因此循环k次
      memcpy(backup, dis,
         ↪ sizeof(dis));//为什么要备份呢，因为我们要确保下面整个遍历边的过程之间不会
      for(int j = 1; j <= n; j++){//每次遍历所有的边
         for(int x = h[j]; x != -1; x = nex[x]){
            int b = val[x], z = wei[x];
            dis[b] = min(dis[b], backup[j] + z);
         }

      }

```

```
    }
    if(dis[n] > 0x3f3f3f3f / 2)return
        ↪ 0x3f3f3f3f;//由于有负权，负无穷大可能会减小一些
    else return dis[n];
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m>>k;
    memset(h, -1, sizeof(h));
    while(m--){
        int a, b, z;
        cin>>a>>b>>z;
        add(a, b, z);
    }
    int ans = Bellman_Ford();
    if(ans == 0x3f3f3f3f)cout<<"impossible";
    else cout<<ans;
    return 0;

}
```

SPFA

▶ spfa 求最短路

- 思路分析。

```
SPFA是对Bellman-Ford算法的优化，Bellman-Ford算法每次都需要遍历所有的边，判断dis[b]
    ↪ = min(dis[b], dis[a] +
    ↪ w)，看看能否做松弛，然而实际上只有dis[a]变小时，dis[b]才会更新。
因此我们利用BFS，队列存储变小更新了的点，每次取出队首元素，更新邻点，再把邻点加入队列

需要指出的是,SPFA虽然通常效率较高，但也可能因为网格数据而被特意限制，需要视情况而定。
```

- 代码实现

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
#include <queue>

using namespace std;

const int N = 1e5 + 10;
int n, m;
int h[N], val[N], nex[N], wei[N], idx;
int dis[N],
    ↪ vis[N];//存储各个点到源点的距离，同时vis判断是否对应的距离变小

void add(int a, int b, int z){
   val[idx] = b, nex[idx] = h[a], wei[idx] = z, h[a] = idx++;
}

int spfa(){
   memset(dis, 0x3f, sizeof(dis));
   dis[1] = 0;
   queue<int> q;//队列存储距离变小的点

   q.push(1);
   vis[1] = true;
   while(!q.empty()){
      auto t = q.front();
      q.pop();
      vis[t] = false;
      for(int i = h[t]; i != -1; i =
          ↪ nex[i]){//当前点更新了，那么它的邻点也要更新
         int point = val[i], weight = wei[i];
         if(dis[point] > dis[t] + weight){
            dis[point] = dis[t] + weight;
```

```
            if(!vis[point]){//point可能已经在队列里，如果不再，那么需要把它加入队列中
                q.push(point);
                vis[point] = true;
            }
        }
    }

    }
    if(dis[n] == 0x3f3f3f3f)return 0x3f3f3f3f;//判断是否存在最短路径
    else return dis[n];
}


int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m;
    memset(h, -1, sizeof(h));
    while(m--){
        int a, b, z;
        cin>>a>>b>>z;
        add(a, b, z);
    }
    int ans = spfa();
    if(ans == 0x3f3f3f3f)cout<<"impossible";
    else cout<<ans;

    return 0;


}
```

► **spfa 判断负环**

- 思路分析

当没有负环时最短路径最多只会经过n-1条边，当边数超过n-1时便可推断出存在负环，这便是判

Bellman-Ford判断负环是根据第n次迭代有没有更新

同理，SPFA由于每次不确定当前经过了几条边，因此需要额外一个数组统计当前点到源点的边数
当某次+完后超过n-1时便可判断存在负环

- 代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cstdio>
#include <queue>

using namespace std;

const int N = 1e5 + 10;

int n, m;
int h[N], val[N], nex[N], wei[N], idx;
int dis[N], vis[N], cnt[N];//cnt存储各个点到源点的边长
queue<int> q;

void add(int a, int b, int z){
   val[idx] = b, nex[idx] = h[a], wei[idx] = z, h[a] = idx++;
}

bool spfa(){
   memset(dis, 0x3f, sizeof(dis));
   dis[1] = 0;
   for(int i = 1; i <= n;
      ↪ i++){//由于题目求的是是否存在负环，因此我们需要把所有点都考虑进去
      q.push(i);
      vis[i] = true;
```

```
    }
    //q.push(1);//首先将源点入队
    //vis[1] = true;
    while(!q.empty()){
        auto t = q.front();
        q.pop();
        vis[t] = false;//便于下次可能再次入队
        for(int i = h[t]; i != -1; i = nex[i]){
            int point = val[i], weight = wei[i];
            if(dis[point] > dis[t] +
                ↪ weight){//如果取出的点可以使得它的邻点到源点的距离变小，就更新
                dis[point] = dis[t] + weight;
                cnt[point] = cnt[t] + 1;//经过的边长+1
                if(cnt[point] > n - 1)return
                    ↪ true;//如果边长超过n-1，说明存在负环
                if(!vis[point]){//如果邻点不再队列中，就加入到队列中，否则不用加(防止重复
                    vis[point] = true;
                    q.push(point);
                }

            }
        }
    }
    return false;
}


int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m;
    memset(h, -1, sizeof(h));
    while(m--){
        int a, b, z;
```

```
        cin>>a>>b>>z;
        add(a, b, z);
    }

    bool ans = spfa();
    if(ans)cout<<"Yes";
    else cout<<"No";
    return 0;
}
```

4.5.2 多源最短路径-Floyd

▶ Floyd 求最短路

- 思路分析。

```

floyd是处理多源最短路径问题，它是一种动态规划算法，每次考虑经过的点是1~k时能否更新最
for k 1 to n
dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])
时间复杂度是O(n^3)
```

- 代码实现

```
#include <iostream>
#include <algorithm>
#include <cstdio>
#include <cstring>

using namespace std;

const int N = 300;
int d[N][N];
int n, m, k;


void floyd(){
```

```
    for(int k = 1; k <= n; k++){//三层循环
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
}


int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n>>m>>k;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            if(i == j)d[i][j] =
                ↪ 0;//需要设置自己到自己距离是0，否则后面floyd难以更新
            else d[i][j] = 0x3f3f3f3f;
        }
    }
    while(m--){
        int a, b, z;
        cin>>a>>b>>z;
        d[a][b] = min(d[a][b], z);
    }
    floyd();
    while(k--){
        int a, b;
        cin>>a>>b;
        if(d[a][b] > 0x3f3f3f3f / 2)cout<<"impossible"<<endl;
        else cout<<d[a][b]<<endl;
    }

```

```
48    return 0;
49 }
```

五、 数学

六、 贪心

6.1 贪心分析

贪心策略一般是直觉

贪心证明有两种

一种是数学归纳法，对贪心算法步骤或者输入规模归纳，归纳过程中证明最优性另一种是替换，设法将最优解替换成算法解，并且替换过程中保持最优性

6.2 区间问题

▶ 区间选点

- 思路分析

```
贪心策略
1.将每个区间按照右端点排序
2.从前到后枚举每个区间,如果当前区间已经有点,跳过;否则选择区间右端点
3.输出总共选择的点

贪心证明
直觉上是既然每个区间必须有点，那就选右端点，使每个点尽可能地覆盖更多的区间
但也可以简要证明
首先这种贪心是一种合法方案,同时假设最终最少的点是min,贪心求出来的是ret
首先min是最少的,min<=ret
接下来只需证明min>=ret
根据贪心策略,问题等价于我们只考虑从前往后哪些没有跳过的区间(跳过的区间对于点的选择没
综上min==ret,贪心策略正确
```

- 代码实现

```
#include<iostream>
#include<algorithm>
```

```cpp
using namespace std;
const int N = 100010;
struct Range{
   int l,r;
   bool operator<(const Range &g)const{
      return r<g.r;
   }
}range[N];
int main(){
   int n,l,r;
   cin>>n;
   for(int i=0;i<n;i++){
      cin>>l>>r;
      range[i]={l,r};
   }
   int ret = 0,end = -2e9;//end存储上一个选择的点
   sort(range,range+n);
   for(int i=0;i<n;i++){
      if(range[i].l>end){//如果当前区间没有点,选择右端点
         ret++;
         end=range[i].r;
      }
   }
   cout<<ret;
   return 0;
}
```

▶ 最大不相交区间数量

- 思路分析

```
贪心策略
1.将每个区间按照右端点排序
2.从前到后枚举每个区间,如果选择当前区间会存在相交情况,跳过;否则选择当前区间
3.输出总共选择的区间数

贪心证明
```

```
直觉上是使选择的区间尽可能对后面的区间选择没有影响
但也可以简要证明
首先这种贪心是一种合法方案,同时假设最终最大的区间数量是max,贪心求出来的是ret,max>=re
接下来只需证明max<=ret
同样问题等价于我们只考虑从前往后哪些没有跳过的区间(跳过的区间对于区间数没有贡献),这些
综上max==ret, 贪心策略正确

这里其实和区间选点问题代码一样,可以看成一个点对应一个区间,只不过区间选点求的是下界,这
```

- 代码实现

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 100010;
struct Range{
   int l,r;
   bool operator<(const Range &g)const{
      return r<g.r;
   }
}range[N];
int main(){
   int n,l,r;
   cin>>n;
   for(int i=0;i<n;i++){
      cin>>l>>r;
      range[i]={l,r};
   }
   int ret = 0,end = -2e9;//end存储上一个选择的区间的右端点
   sort(range,range+n);
   for(int i=0;i<n;i++){
      if(range[i].l>end){//如果选择当前区间不会存在相交,选择,并更新右端点
         ret++;
         end=range[i].r;
      }
   }
```

```
    cout<<ret;
    return 0;
}
```

▶ 区间分组

- 思路分析

```
方法1:
可以类比给课程安排教室，每个课程的开始时间和结束时间分别是Si和Fi，每个时间段一个教室
先按照时间先后排序
遇到课程开始时间就教室+1，遇到结束时间教室-1，中间最大的教师数就是最少需要的教室

方法2：贪心
贪心策略：
1.将区间按照左端点排序
2.从前往后枚举每一个区间，如果当前区间和已存在的某个分组没有交集，就加入该分组，否则
3.输出分组数量

贪心证明：
比较当前区间是否和某个组有交集是根据当前区间的左端点和所有组的最大区间右端点
证明类似最大不相交区间数量,只不过为什么按左端点排序是个问题
```

- 代码实现

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 2e5+10;
int point[N];
int main(){
    int n,a,b;
    cin>>n;
    for (int i = 0; i < n; i ++ ){
        cin>>a>>b;
        point[2*i]=2*a; //标记区间左端点为偶数
        point[2*i+1]=2*b+1;//标记区间有端点为奇数
```

```
    }
    sort(point,point+2*n);
    int ret = 0,count=0;
    for (int i = 0; i < 2*n; i ++ ){
        if(point[i]%2==0)count++; //遇到左端点就加1
        else count--; //遇到右端点就少1
        ret = max(ret,count);
    }
    cout<<ret<<endl;
    return 0;
}
```

▶ 区间覆盖

- 思路分析

```
贪心策略
1.将所有区间按照左端点排序，设置end等于目标区间的左端点
2.从前往后枚举所有区间，选择能够覆盖end的右端点最大的区间,更新end
3.继续重复进行第2步，如果最终end小于目标区间右端点说明无法覆盖；否则可以覆盖

贪心证明
从前往后可以逐个将区间的最优解替换成算法解，并且保持最优性
```

- 代码实现

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 1e5+10;

struct Range{
    int l,r;
    bool operator<(const Range&r){
        return l<r.l;
    }
}range[N];
```

```
12
13 int main(){
14    int a,b,n;
15    cin>>a>>b>>n;
16    for(int i=0;i<n;i++){
17       cin>>range[i].l>>range[i].r;
18    }
19    sort(range,range+n);
20    int end = a,ret=0;//end为要覆盖的目标端点
21    for(int i=0;i<n;i++){
22       int j = i,r=-2e9;
23       for(;j<n&&range[j].l<=end;j++){ //寻找能覆盖端点的最大区间右端点
24          r = max(r,range[j].r);
25       }
26       ret++;
27       end = r;
28       if(end<a){ //找不到能覆盖的区间，输出-1
29          cout<<-1<<endl;
30          return 0;
31       }
32       if(end>=b)break; //如果完成目标能覆盖右端点,退出
33       i=j-1;
34    }
35    if(end<b)cout<<-1<<endl;//整个循环退出依然无法完成目标，输出-1
36    else cout<<ret<<endl;
37    return 0;
38 }
```

6.3 Huffman 树

▶ 合并果子

- 思路分析

```
1 贪心策略
2 每次选择两个权值最小的果子合并，新果子的体力消耗是合并的果子体力消耗之和，直到果子只
```

- 代码实现

```
#include<iostream>
#include <queue>

using namespace std;

int main(){
   int n;
   cin>>n;
   priority_queue<int,vector<int>,greater<int>>minHeap;//使用小根堆
   for (int i = 0; i < n; i ++ ){
      int a;
      cin>>a;
      minHeap.push(a);
   }
   int ret = 0;
   while(minHeap.size()>1){
      int a = minHeap.top();//每次取出两个最小体力消耗果子合并
      minHeap.pop();
      int b = minHeap.top();
      minHeap.pop();
      minHeap.push(a+b);
      ret += a+b;
   }
   cout<<ret;
   return 0;
}
```

6.4 排序不等式

▶ 排队打水

- 思路分析

```
贪心策略
将每个人打水的时间从小到大排序，打水顺序就是该顺序
```

```
贪心证明
假设最优解的打水顺序不是按照打水时间从小到大顺序，那么必定存在两个相邻的逆序对(否则两
```

- 代码实现

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 1e5+10;
int num[N];
int main(){
   int n;
   cin>>n;
   long long ret = 0;
   for(int i=0;i<n;i++){
      cin>>num[i];
   }
   sort(num,num+n);
   for(int i=0;i<n;i++){
      ret += num[i]*(n-i-1);//每个打水时间的权重是后面的人数之和
   }
   cout<<ret<<endl;
   return 0;
}
```

6.5 绝对值不等式

▶ 货仓选址

- 思路分析

```
贪心策略
如果n是奇数，选在中位数
如果n是偶数，选在中间两个数之间即可

证明
```

```
利用绝对值不等式|a|+|b|>=|a+b|, 当且仅当ab>=0时等号成立
假设选取的点是x, 数轴上从小到大共有n个目标点x1,x2,x3,...,xn
距离和为f(x)=|x1-x|+|x2-x|+...+|xn-x|
=(|x-x1|+|xn-x|)+(|x-x2|+|xn-1-x2|)+...
>=xn-x1+xn-1-x2+...
当且仅当x1<=x<=xn,x2<=x<=xn-1,...时等号成立
从而选择中位数作为货仓最好
```

- 代码实现

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

const int N=100010;
int num[N];
int main(){
   int n;
   cin>>n;
   for(int i=0;i<n;i++)cin>>num[i];
   sort(num,num+n);
   long long ret = 0;
   for(int i=0;i<n;i++)ret += abs(num[i]-num[n/2]);//选择中位数点
   cout<<ret<<endl;
   return 0;
}
```

6.6 推公式

▶ 耍杂技的牛

- 思路分析

```
贪心策略
从上到下按照重量w和强壮程度s之和排序堆叠

贪心证明
```

```
假设最优解不是按照这种序，则必然存在两头牛w(i),s(i)和w(i+1),s(i+1)
    ↪ w(i)+s(i)>w(i+1)+s(i+1)
风险分别为w(1)+w(2)+...+w(i-1)-s(i) 和 w(1)+w(2)+...+w(i)-s(i+1)
交换后两头牛分别为w(i+1),s(i+1)和w(i),s(i)
风险分别为 w(1)+w(2)+...+w(i-1)-s(i+1) 和
    ↪ w(1)+w(2)+...+w(i-1)+w(i+1)-s(i)

舍去前i-1项
交换前风险分别为 -s(i) 和 w(i)-s(i+1)
交换后风险分别为 -s(i+1) 和 w(i+1)-s(i)
w(i)-s(i+1)>w(i+1)-s(i)
w(i)-s(i+1)>-s(i+1)
因此交换后的最大风险只可能不变或者变xiao
```

- 代码实现

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

const int N = 50010;
int w[N],s[N];
pair<int,int> PII[N];

int main(){
   int n;
   cin>>n;
   for(int i=0;i<n;i++){
      cin>>w[i]>>s[i];
      PII[i]={w[i]+s[i],s[i]};
   }
   sort(PII,PII+n);
   long long ret=-2e9,sum=0,t;
   for(int i=0;i<n;i++){
      t = sum-PII[i].second;
      ret = max(ret,t); //求最大风险值
```

```
        sum += PII[i].first-PII[i].second;
    }

    cout<<ret;
    return 0;
}
```

七、 动态规划

7.1 分析模板

动态规划算是最难的一类了，没有固定的模板，但有其思想，闫式 DP 分析法如下

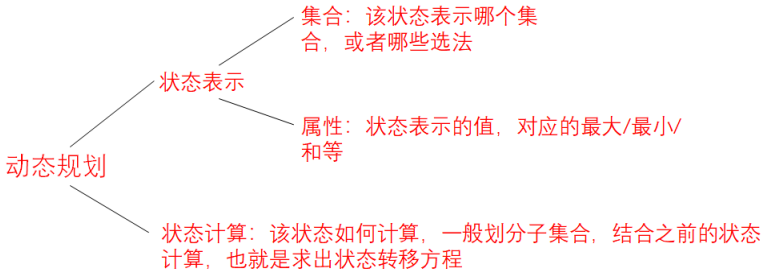


图 **23**

状态表示是一个数代表一个集合，并且有一个具体属性，优化搜索

状态计算依据一般是寻找最后一个不同点

```
先得出状态表示(一个表达式)，一维/二维/...，这个需要一定经验
集合：状态表示哪个集合
属性：状态表示这个集合具体什么值，最小/最大/和

然后进行状态计算
集合划分:根据最后一步xxx划分,各个子集分别对应哪个状态(不一定一一对应，一个状态可能包含或
状态转移:转移方程是这些状态取最小/最大/求和

最后按流程求解
初始化:需要设定什么初始值
转移过程:如何编写代码对应状态转移
答案:最终答案是哪个状态或者怎么得到
```

下面以各类题目为例

7.2 背包问题

背包问题是经典的问题了，有很多种解法，最经典的还是动态规划

7.2.1 01 背包问题

▶ 01 背包问题

- 思路分析

```
状态表示:dp(i,j)
集合:前i件物品，体积不超过j的背包物品的选择方式
属性:最大价值

状态计算
状态划分:
根据第i件物品选与不选划分
选第i件物品,可由前i-1件物品递推得到,dp(i-1,j-v[i])+w[i]
不选第i件物品，就是前i-1件物品,dp(i-1,j)
状态转移:
两者取最小值
```

图 24

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE = 1010;
int dp[SIZE][SIZE]; //SIZE设大点方便，dp二维数组初始化为0
int main(){
   int N,V;
   cin>>N>>V;
   int* v = new int[N];
```

```
    int* w = new int[N];
    for(int i=1;i<=N;i++){
        cin>>v[i]>>w[i];
    }
    for(int i=1;i<=N;i++){
        for(int j=0;j<=V;j++){
            dp[i][j]=dp[i-1][j];
            if(j>=v[i])dp[i][j]=max(dp[i][j],dp[i-1][j-v[i]]+w[i]);
        }
    }
    cout<<dp[N][V];
    return 0;
}
```

- 01 背包的优化—使用滚动数组

  朴素的 01 背包使用了两维数组，但实际上根据递推公式，只是第 i 层和第 i-1 层不断迭代，那么我们可以使用滚动数组，更新前的旧数组为第 i-1 层，当前更新的为第 i 层，利用这个差完成优化。(有一个无脑的降维技巧就是直接先删去第一维再考虑要不要改)

```
#include<iostream>
using namespace std;
const int SIZE = 1010;
int dp[SIZE];
int main(){
    int N,V;
    cin>>N>>V;
    int* v = new int[N];
    int* w = new int[N];
    for(int i=1;i<=N;i++){
        cin>>v[i]>>w[i];
    }
    for(int i=1;i<=N;i++){
        for(int j=V;j>=v[i];j--){
            ↪ //这里j从大到小，是为了保证j-v[i]还没有更新也就是属于i-1层,以后直接一步到
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
```

```
    }
  }
  cout<<dp[V];
  return 0;
}
```

7.2.2 完全背包问题

▶ 完全背包问题

- 思路分析

  和 01 背包的区别是每件物品无限件可用，区别在于集合划分。

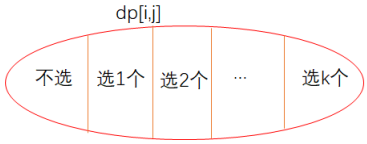


图 25

  得出递推公式：$dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i]] + w[i], dp[i-1][j-2*v[i]] + 2*w[i], \ldots, dp[i-1][j-k*v[i]] + k*w[i])$

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE = 1010;
int dp[SIZE][SIZE];
int main(){
  int N,V;
  cin>>N>>V;
  int* v = new int[N];
  int* w = new int[N];
  for(int i=1;i<=N;i++){
    cin>>v[i]>>w[i];
  }
  for(int i=1;i<=N;i++){
    for(int j=1;j<=V;j++){
      for(int k=0;k<=j/v[i];k++){
```

```
            dp[i][j]=max(dp[i][j],dp[i-1][j-k*v[i]]+k*w[i]);
        }
    }
}
cout<<dp[N][V];
return 0;
}
```

- 完全背包的优化-基于递推关系
  朴素的完全背包似乎需要枚举每种的所有可能性，但是，真的如此吗，会不会有重复

```
以dp[i][j]和dp[i][j-v[i]]为例,尝试找出关系
dp[i][j]=max(dp[i-1][j],dp[i-1][j-v[i]]+w[i],dp[i-1][j-2*v[i]]+2*w[i],...,dp[i-1][
dp[i][j-v[i]]=max(   dp[i-1][j-v[i]],  d[i-1][j-2*v[i]]+w[i],...,
    ↪ dp[i-1][j-k*v[i]+(k-1)*w[i])
可以看出dp[i][j]相比于dp[i][j-v[i]]，后面多了w[i]
因此简化成dp[i][j]=max(dp[i-1][j],dp[i][j-v[i]]+w[i])
```

```
#include<iostream>
using namespace std;
const int SIZE=1010;
int dp[SIZE][SIZE];
int main(){
    int N,V;
    cin>>N>>V;
    int* v = new int[N];
    int* w = new int[N];
    for(int i=1;i<=N;i++){
        cin>>v[i]>>w[i];
    }
    for(int i=1;i<=N;i++){
        for(int j=0;j<=V;j++){
            dp[i][j]=dp[i-1][j];
            if(j>=v[i])dp[i][j]=max(dp[i][j],dp[i][j-v[i]]+w[i]);
        }
```

```
    }
    cout<<dp[N][V];
    return 0;
}
```

进一步，可以使用滚动数组降成一维

```
#include<iostream>
using namespace std;
const int SIZE=1010;
int dp[SIZE];
int main(){
    int N,V;
    cin>>N>>V;
    int* v = new int[N];
    int* w = new int[N];
    for(int i=1;i<=N;i++){
        cin>>v[i]>>w[i];
    }
    for(int i=1;i<=N;i++){
        for(int j=v[i];j<=V;j++){
            ↪ //这里j从小到大，是因为dp[j-v[i]]是在第i层，与01背包也只有这里不同
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
                ↪ //记忆的话就是说完全背包能不能多要一件使得价值更大
        }
    }
    cout<<dp[V];
    return 0;
}
```

7.2.3 多重背包问题

▶ 多重背包问题 I

- 思路分析

  和完全背包的区别只在于每件物品有限，首先正常思路仍然可以找出递推关系

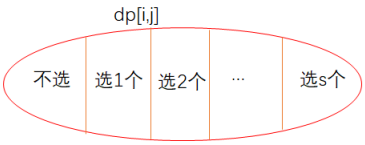


图 **26**

递推关系式为：$dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i]]+w[i], dp[i-1][j-2*v[i]]+2*w[i], ..., dp[i-1][j-s*v[i]]+s*w[i])$

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE=1010;
int dp[SIZE][SIZE];
int main(){
   int N,V;
   cin>>N>>V;
   int* v = new int[N];
   int* w = new int[N];
   int* s = new int[N];
   for(int i=1;i<=N;i++){
      cin>>v[i]>>w[i]>>s[i];
   }
   for(int i=1;i<=N;i++){
      for(int j=0;j<=V;j++){
         for(int k=0;k<=s[i]&&k<=j/v[i];k++){
            dp[i][j]=max(dp[i][j],dp[i-1][j-k*v[i]]+k*w[i]);
         }
      }
   }
   cout<<dp[N][V];
   return 0;
}
```

- 多重背包的优化-基于二进制

```
首先可以发现，各个物品的数量不相同，单纯递推关系不能再优化，但可以使用二进制进行优化。

```

```
对于任意一个整数k,0<=k<=s，可以表示成k=0+1+2+4+...+a,

证明：对于s，0+1+2+4+8+...+2^i+2^j+...
显然s必然属于某个左闭右开集合，比如[2^i,2^j)
s = 0+1+2+4+8+...+2^i+a;
从而我们不必列举0~s所有可能性，而只需要列举0,1,2,...,a，复杂度降到logs
实际上我们将一件数量为s的物品转换成若干个数量为1物品，它们的体积和价值是原来相应的倍
这样实际变成了01背包问题，下面给出代码，直接上一维了。
```

```
#include<iostream>
using namespace std;
const int SIZE=14000;
int dp[SIZE];
int main(){
   int N,V;
   cin>>N>>V;
   int* v = new int[N*14];
   int* w = new int[N*14];
   int index = 0;
   int a,b,s;
   for(int i=1;i<=N;i++){ //初始化v和w
      cin>>a>>b>>s;
      for(int k=1;k<=s;k*=2){
         index++;
         v[index] = a*k;
         w[index] = b*k;
         s-=k;
      }
      if(s>0){
         index++;
         v[index] = a*s;
         w[index] = b*s;
      }
   }
   for(int i=1;i<=index;i++){
```

```
27      for(int j=V;j>=v[i];j--){
28         dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
29      }
30   }
31   cout<<dp[V];
32   return 0;
33}
```

7.2.4 分组背包问题

▶ 分组背包问题

- 思路分析

```
对于dp[i][j]，表示前i组体积不超过j的所有选法中的最大价值
dp[i][j] =
   ↪ max(dp[i-1][j],dp[i-1][j-v[i][1]]+w[i][1],...,dp[i-1][j-v[i][s]]+w[i][s])
   ↪
```

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE = 120;
int dp[SIZE];
int v[SIZE][SIZE];
int w[SIZE][SIZE];
int s[SIZE];
int main(){
   int N,V;
   cin>>N>>V;
   for(int i=1;i<=N;i++){
      cin>>s[i];
      for(int j=1;j<=s[i];j++){
         cin>>v[i][j]>>w[i][j];
      }
   }
```

```
    for(int i=1;i<=N;i++){
        for(int j=V;j>=0;j--){
            for(int k=1;k<=s[i];k++){
                ↪ //注意这里j和k的顺序不能变，每次列举的k对应j-v[i]层
                if(j>=v[i][k])dp[j]=max(dp[j],dp[j-v[i][k]]+w[i][k]);
            }
        }
    }
    cout<<dp[V];
    return 0;
}
```

7.3 线性 DP

所谓线性 dp，就是指状态之间有线性关系的动态规划问题。

▶ 数字三角形

- 思路分析

```
从动态规划角度考虑

首先定义状态，根据三角形数据状况定义dp(i,j)，i表示行数，j表示列数(注意对应关系)

集合：从起点出发到（i，j）位置处的所有走法

属性：所有走法中数字之和的最大值

然后，找出状态转移方程

对于(i,j)位置，它可由(i-1,j-1)和(i-1,j)位置转移而来
也就是dp[i][j]=max(dp[i-1][j-1],dp[i-1][j])+arr[i][j]
当然要注意边界情况

最后，注意遍历方式，显然i，j都从小到大遍历即可
```

- 代码实现

```cpp
#include<iostream>
using namespace std;
const int SIZE = 510;
int dp[SIZE][SIZE];
int num[SIZE][SIZE];
int main(){
   int n;
   cin>>n;
   for(int i=1;i<=n;i++){
      for(int j=1;j<=i;j++){
         cin>>num[i][j];
      }
   }
   for(int i=1;i<=n;i++){
      for(int j=1;j<=i;j++){
         if(j==1)dp[i][j]=dp[i-1][j]+num[i][j];
         else if(j==i)dp[i][j]=dp[i-1][j-1]+num[i][j];
         else{
            dp[i][j]=max(dp[i-1][j-1],dp[i-1][j])+num[i][j];
         }
      }
   }
   int maxPathNum=dp[n][1];
   for(int j=2;j<=n;j++){
      maxPathNum=max(maxPathNum,dp[n][j]);
   }
   cout<<maxPathNum;
   return 0;
}
```

- 自底向上的优化一般数字三角形是从上往下进行递推的，然而这样需要判断边界条件并且答案还需要遍历最后一行，实际上我们可以从下向上进行递推，最终规约到第一行一个数，直接输出，方便很多，当然复杂度还是一样。

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

```

const int SIZE = 510;
int dp[SIZE][SIZE];
int num[SIZE][];

int main(){
   int n;
   cin>>n;
   for(int i=1;i<=n;i++){
      for(int j=1;j<=i;j++){
         cin>>num[i][j];
      }
   }
   for(int i=n;i>=1;i--){
      for(int j=1;j<=i;j++){
         dp[i][j]=max(dp[i+1][j],dp[i+1][j+1])+num[i][j];
      }
   }
   cout<<dp[1][1];
   return 0;
}
```

▶ 最长上升子序列

- 思路分析

```
首先定义状态dp(i)，表示以i结尾的严格递增的子序列(集合)的最长值(属性)、

然后进行状态计算，dp(i)和前面dp(1),dp(2),...,dp(i-1)都有关，也就是

dp[i]=max(dp[1]+num[i]>num[1],dp[2]+num[i]>num[2]+...+dp[i-1]+num[i]>num[i-1])
```

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE = 1000;
```

```
int dp[SIZE];
int num[SIZE];
int main(){
   int N;
   cin>>N;
   for(int i=1;i<=N;i++){
      cin>>num[i];
   }
   for(int i=1;i<=N;i++){
      dp[i]=1;
      for(int j=1;j<i;j++){
         if(num[i]>num[j])dp[i]=max(dp[i],dp[j]+1);

      }
   }
   int ret = dp[1];
   for(int i=1;i<=N;i++){
      ret = max(ret,dp[i]);
   }
   cout<<ret;
   return 0;
}
```

- 优化朴素的动态规划需要遍历当前位置之前的每个元素，然而实际上并不需要，可以进行优化。

  我们原来 dp 数组存储的是以 i 结尾的上升子序列的最大长度，也就是我们可能存储了重复的长度，实际上每种长度只需存一个它的最小结尾。

  比如当前来到 i 位置，对于 1,2,...,i-1 位置，上升子序列长度最大为 i-1，用 dp[1,2,3,...i-1] 存储每种长度的最小结尾数，因为如果 i 位置能够接在最小结尾数后面，那也一定能接在更大结尾数后面，比如对于 num[i]，它应该替换小于它的最大结尾数后一位 (使用二分)，比如下标 k，使得 dp[k+1] 变成更小的 num[i]，从而完成状态转移，最终 dp 的长度就是答案。

  总之就是相比 key 为下标，值为长度，key 为长度，值为下标可以使用二分优化到 logn

```
#include<iostream>
```

```
using namespace std;
const int SIZE = 100010;
int dp[SIZE];
int num[SIZE];
int binarySearch(int base,int length){
   int l=0,r=length;
   while(l<r){
      int mid = l+r+1>>1;
      if(dp[mid]<base)l=mid;
      else r=mid-1;
   }
   return l;
}
int main(){
   int N;
   cin>>N;
   for(int i=1;i<=N;i++){
      cin>>num[i];
   }
   dp[0]=-2e9; //为了确保能二分，增设dp[0]
   int length=0;
   for(int i=1;i<=N;i++){
      int index = binarySearch(num[i],length)+1;
      dp[index]=num[i];
      length=max(length,index);
   }
   cout<<length;
   return 0;
}
```

▶ 最长公共子序列

- 思路分析

```
状态表示:定义状态dp(i,j)，集合表示A的前i个字符和B的前j个字符的子序列，属性是长度最长
状态计算:dp(i,j)
根据最后一个字符是否在子序列中分为四类
```

```
情况1:第i个字符不在,第j个字符不在,此时为dp(i-1,j-1)
情况2:第i个字符不在,第j个字符在,此时为dp(i-1,j)
情况3:第i个字符在,第j个字符不在,此时为dp(i,j-1)
情况4:第i个字符在,第j个字符在,此时为dp(i-1,j-1)+1
    ↪ 注意条件需要满足str[i]==str[j]

初步想法如上，但是仔细思考会发现dp(i-1,j)和dp(i,j-1)超出对应情况
dp(i-1,j)只是表示B前j个字符在子序列中，第j个在不在不确定，因此包括情况1和情况2
同样dp(i,j-1)包括情况1和情况3

但是由于我们求的是最长公共子序列，因此重复并不影响，最终我们只选取dp(i-1,j),dp(i,j-1
```

- 代码实现

```cpp
#include<iostream>
using namespace std;
const int SIZE = 1010;
char A[SIZE];
char B[SIZE];
int dp[SIZE][SIZE];
int main(){
   int N,M;
   cin>>N>>M;
   cin>>A+1;
   cin>>B+1;
   for(int i=1;i<=N;i++){
      for(int j=1;j<=M;j++){
         dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
         if(A[i]==B[j])dp[i][j]=max(dp[i][j],dp[i-1][j-1]+1);
      }
   }
   cout<<dp[N][M];
   return 0;
}
```

▶ 最短编辑距离

- 思路分析

```
状态表示:dp(i,j)
集合:将A的前i个字符变成B的前j个字符的操作数
属性:最小值

状态计算
集合划分:
根据最后一次操作的不同划分
情况1:删除,将A[i]删除后A变成B,意味A前i-1和B前j完成操作，dp(i-1,j)+1
情况2:插入,插入B[j]到A末尾后后A变成B，意味着A前i和B前j-1完成操作，dp(i,j-1)+1
情况3:替换,将A[i]替换成B[j],意味着A前i-1和B前j-1完成操作，dp(i-1,j-1)+A[i]==B[j],注

状态转移:
三种情况取最小值
```

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE = 1010;
char A[SIZE];
char B[SIZE];
int dp[SIZE][SIZE];
int main(){
   int N,M;
   cin>>N;
   cin>>A+1;
   cin>>M;
   cin>>B+1;
   for(int i=0;i<=N;i++){
      dp[i][0]=i;//B无字符时A只能删除
   }
   for(int j=0;j<=M;j++){
      dp[0][j]=j;//A无字符时A只能插入
   }
   for(int i=1;i<=N;i++){
```

```
    for(int j=1;j<=M;j++){
        dp[i][j]=min(dp[i-1][j]+1,dp[i][j-1]+1);
        dp[i][j]=min(dp[i][j],dp[i-1][j-1]+1-(A[i]==B[j]));
    }
  }
  cout<<dp[N][M];
  return 0;
}
```

▶ 编辑距离

- 思路分析

```
只是求多次最短编辑距离
```

- 代码实现

```
#include<iostream>
#include<string.h>
using namespace std;
const int SIZE = 1010;
char A[SIZE][SIZE];
char B[SIZE];
int dp[SIZE][SIZE];
int lowestDistance(char* A,char* B){
  int lA = strlen(A+1),lB=strlen(B+1);
  for(int i=0;i<=lA;i++){
    dp[i][0]=i;
  }
  for(int i=0;i<=lB;i++){
    dp[0][i]=i;
  }
  for(int i=1;i<=lA;i++){
    for(int j=1;j<=lB;j++){
      dp[i][j]=min(dp[i-1][j]+1,dp[i][j-1]+1);
      dp[i][j]=min(dp[i][j],dp[i-1][j-1]+1-(A[i]==B[j]));
    }
```

```
    }
    return dp[lA][lB];
}
int main(){
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++){
        cin>>A[i]+1;
    }
    int bound;
    for(int i=0;i<m;i++){
        cin>>B+1;
        cin>>bound;
        int cnt = 0;
        for(int j=1;j<=n;j++){
            int num = lowestDistance(A[j],B);
            if(num<=bound)cnt++;
        }
        cout<<cnt<<endl;
    }
    return 0;
}
```

7.4 区间 DP

所谓区间 dp，指在一段区间上进行动态规划，一般做法是由长度较小的区间往长度较大的区间进行递推，最终得到整个区间的答案，而边界就是长度为 1 以及 2 的区间。

▶ 石子合并

- 思路分析

```
状态表示:dp(i,j)
集合:将第i堆到第j堆石子合并的顺序
属性:合并的最小代价

状态计算
集合划分:根据最后一次石子合并的顺序划分
```

```
i-j一共 j-i+1堆
最后一次合并可以是[i,k]和[k+1,j]合并,其中k取i到j-1, 代价为dp(i,k)+dp(k+1,j)+cost(i,
状态转移:取最小
for(int k=i;k<j;k++)
dp(i,j)=min(dp(i,j),dp(i,k)+dp(k+1,j)+cost(i,j))
```

- 代码实现

```
#include<iostream>
using namespace std;
const int SIZE=310;
int num[SIZE];
int preCost[SIZE];
int dp[SIZE][SIZE];
int main(){
   int N;
   cin>>N;
   for(int i=1;i<=N;i++){
      cin>>num[i];
   }
   for(int i=1;i<=N;i++){
      preCost[i]=num[i]+preCost[i-1]; //使用前缀数组
   }
   for(int len=1;len<N;len++){
      for(int i=1;i<=N-len;i++){
         int j=i+len;
         dp[i][j]=dp[i][i]+dp[i+1][j]+preCost[j]-preCost[i-1];
         for(int k=i;k<j;k++){
            dp[i][j]=min(dp[i][j],dp[i][k]+dp[k+1][j]+preCost[j]-preCost[i-1]);
         }
      }
   }
   cout<<dp[1][N];
   return 0;
}
```

7.5 状态压缩 DP

▶ 蒙德里安的梦想

- 思路分析

```
状态压缩其实就是将一组状态压缩成一个整数的二进制表示

首先这里只需考虑横方块的放置方案数，因为横方块只要合法放置，那么竖放块将剩下的填充即
状态表示:dp(i,j)
集合:对于横向方块,前i-1列已经放好,第i列的放置的状态为j(放置的意思是方块放置的起点为当
属性:方案数之和

状态计算
集合划分:根据第i列的状态不同划分
对于第i列来讲,有N行,理论上有2^N种不同状态,如下
dp(i,00...00),dp(i,00...01),...,dp(i,11...11)

状态转移
对于第i列的每一种状态,并不完全合法，不能直接计算,需要满足以下条件
1.当前列状态自身必须合法,连续的0必须是偶数，以便放置竖方块
2.前一列延展到当前列的方块和当前列放置方块不能有重合,也就是如果前一列对应行放了,当前
这样第i列的状态其实由第i-1列决定,也就是某个状态可能对应前一列多种状态,可由前一列的状
用vector<int> transfer[1<<12]
    ↪ 存储当前列对应的前一列的合法状态,index:当前列状态
    ↪ value:前一列状态
则转移方程为:dp(i,j)=dp(i-1,k1)+dp(i-1,k2)+...+dp(i-1,kn)
k1,k2,...,kn为状态j对应的前一列合法状态
```

- 代码实现

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

long long
    ↪ dp[12][1<<12];//状态表示dp[i][j],前i-1列方块已放好(横向方块放置起点列号小于等
```

```
//注意这里的状态j是二进制表示,第几位是1代表当前列第几行放了方块
//以下为预处理部分,这样不必枚举dp每列状态时都穷举前一列所有状态
vector<int>
    ↪ transfer[1<<12];//transfer[j],列状态为j,可由前一列的哪些状态转移而来
bool
    ↪ isConstEven[1<<12];//isisConstEven[j],列状态为j时是否有合法,是否有连续个偶数0

int main(){
   int N,M;
   while(cin>>N>>M,N&&M){//逗号表达式,处理多组用例,遇到0 0截止
      for(int
          ↪ j=0;j<1<<N;j++){//转移条件1:对于列状态j,计算哪些是有连续个偶数0
         isConstEven[j]=true;
         int cnt = 0;//状态j二进制表示0的连续个数
         for(int s=0;s<N;s++){
            if(!(j>>s&1)){//状态j从后往前第s位是0,计数+1
               cnt++;
            }
            else{//遇到1
               if(cnt&1){//判断连续的0是否是偶数个,不是不合法,枚举下一个状态
                  isConstEven[j]=false;
                  break;
               }
               cnt = 0;//是的话重置,当然也可以不重置,偶数+偶数还是偶数
            }
         }
         if(cnt&1)isConstEven[j]=false;//状态j的第一位(倒数最后一位)可能是0,缺少终止
      }
      for(int
          ↪ j=0;j<1<<N;j++){//转移条件2:对于当前列状态j,摆好后不能使得当前列重叠
         transfer[j].clear();
         for(int k=0;k<1<<N;k++){
            if(isConstEven[j|k]&&!(j&k)){//两个转移条件都要满足
               transfer[j].push_back(k);
            }
```

```
        }
      }
      memset(dp, 0, sizeof dp);
      dp[0][0]=1;//第0列不存在,不能放方块, 方案数为1
      for(int i=1;i<=M;i++){
         for(int j=0;j<1<<N;j++){
            for(auto k : transfer[j]){
               dp[i][j] += dp[i-1][k];
            }
         }
      }
      cout<<dp[M][0]<<endl;
   }
}
```

▶ 最短 Hamilton 路径

- 思路分析

```
状态表示:dp(i,j)
集合:经过的点是i(二进制表示),目的地是j(j在i中)的路径
属性:最短路径

状态计算
集合划分:根据倒数第二个点的不同划分,假设倒数第二个点是k
则看能否先从起点到达k,再由k到达终点使得路径更短

状态转移:取最小
dp(i,j)=min(dp(i,j),dp(i-(1<<j),k)+weight(k,j))
需要注意,这里k遍历所有点同时需要满足k在i除去j的集合中

有点类似图的最短路径,但是具体过程是不一样的
这里是从小到大对于每一个经过的点集合,枚举集合里每一个目的地j,枚举集合中
    ↪ 倒数第二个点k,是否能缩短距离到j的距离,需要保证每个点都当且仅当经过一次,每个集合

初始化:dp(1,0)=0 起点为0号点,其他状态都是无穷大
答案:dp((1<<n)-1,n-1)
```

```

tips:这里集合保证每个点都会经过,除了dp(1,0)其他没有更新过的集合都是无穷大
    ↪
```

- 代码实现

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 21;
int
    ↪ dp[1<<N][N];//状态表示dp[i][j],经过的点是i(二进制表示),目的地是j的所有路径(j在
    ↪ 最小距离
int weight[N][N];//所有点之间两两之间的距离

int main(){
   int n;
   cin>>n;
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         cin>>weight[i][j];
      }
   }
   memset(dp,0x3f,sizeof dp);//初始设置成无穷大
   dp[1][0]=0;//从0号点出发,因此经过0号点到0号点距离是0
   for(int i=1;i<1<<n;i++){//对于每一个经过的点集合
      for(int j=0;j<n;j++){//当以j为目的地时
         if(i>>j&1){//j需要在i中
            for(int k=0;k<n;k++){//枚举倒数第二个点,是否能缩短距离
               if(i-(1<<j)>>k&1){//k也需要在除去j的i中
                  dp[i][j]=min(dp[i][j],dp[i-(1<<j)][k]+weight[k][j]);
               }

```

```
            }

        }

    }
}
cout<<dp[(1<<n)-1][n-1];
return 0;
}
```

7.6 树形 DP

▶ 没有上司的舞会

- 思路分析

```
状态表示:dp(i,0)和dp(i,1)
集合:以i为节点,i去或不去
属性:最大快乐值

状态计算
集合划分+状态转移
根据i去不去划分
i去,子节点不去的最大值相加,dp(i,1)=dp(i_sub1,0)+dp(i_sub2,0)+...dp(i_subn,0)+happy(
i不去，子节点可去可不去
dp(i,0)=max(dp(i_sub1,0),dp(i_sub1,1))+...+max(dp(i_subn,0),dp(i_subn,1))

答案:max(dp(0,1),dp(0,0))
```

- 代码实现

```
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;
const int N = 6010;
int n;
```

```cpp
int happy[N]; //每个职工的高兴度
int f[N][2]; //上面有解释哦~
int e[N],ne[N],h[N],idx; //链表，用来模拟建一个树
bool has_father[N]; //判断当前节点是否有父节点
void add(int a,int b){ //把a插入树中
   e[idx] = b,ne[idx] = h[a],h[a] = idx ++;
}
void dfs(int u){ //开始求解题目
   f[u][1] = happy[u]; //如果选当前节点u，就可以把f[u,1]先怼上他的高兴度
   for(int i = h[u];~i;i = ne[i]){ //遍历树
      int j = e[i];
      dfs(j); //回溯
      //状态转移部分，上面有详细讲解~
      f[u][0] += max(f[j][1],f[j][0]);
      f[u][1] += f[j][0];
   }
}
int main(){
   scanf("%d",&n);
   for(int i = 1;i <= n;i ++) scanf("%d",&happy[i]);
      ↪ //输入每个人的高兴度
   memset(h,-1,sizeof h); //把h都赋值为-1
   for(int i = 1;i < n;i ++){
      int a,b; //对应题目中的L,K,表示b是a的上司
      scanf("%d%d",&a,&b); //输入~
      has_father[a] = true; //说明a他有爸爸（划掉）上司
      add(b,a); //把a加入到b的后面
   }
   int root = 1; //用来找根节点
   while(has_father[root]) root ++; //找根节点
   dfs(root); //从根节点开始搜索
   printf("%d\n",max(f[root][0],f[root][1]));
      ↪ //输出不选根节点与选根节点的最大值
   return 0;
}
```

7.7 计数类 DP

▶ 整数划分

- 思路分析

```
状态表示:dp(i,j)
集合:
类比完全背包:从前i个物品中选,体积不超过j, 每种物品可以取无限个
从前i个整数(1~i)选,和恰好是j,每个数可以取无限个
属性:方案数之和

状态计算
集合划分:根据第i个数选几个划分
选0个:dp(i-1,j)
选1个:dp(i-1,j-i)
选2个:dp(i-1,j-2*i)
...
选k个:dp(i-1,j-k*i)

状态转移:求和
dp(i,j)=dp(i-1,j)+dp(i-1,j-i)+dp(i-2,j-2*i)+...+dp(i-k,j-k*i)
化简:
dp(i,j-i)=      dp(i-1,j-i)+dp(i-2,j-2*i)+...+dp(i-k,j-k*i)

dp(i,j)=dp(i-1,j)+dp(i,j-i)

初始化:dp(i,0)=1 前i个数中选,和为0,只有全不选1种
答案: dp(n,n)
```

- 代码实现

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

const int N = 1010;
const int mod = 1e9+7;
int dp[N];
```

```cpp
int main(){
   int n;
   cin>>n;
   for(int i=0;i<=n;i++){
      dp[0]=1;
   }
   for(int i=1;i<=n;i++){
      for(int j=i;j<=n;j++){ //j从小到大枚举,优化到一维
         dp[j] += dp[j-i];
         dp[j] %= mod;
      }
   }
   cout<<dp[n];
   return 0;
}
```

7.8 数位统计 DP

▶ 计数问题

- 思路分析

```
首先为了方便，统计[a,b]数字j出现的次数,可以使用前缀统计[1,b]数字j出现的次数,[1,b]-[1
对于[1,b],要求数字j出现的次数

根据j出现的倒数位数i划分,对于数字j,枚举i从1到b最高位
假设j出现在第i位,[1,b]满足条件数形式必为:left j  right,
b表示为:b_left b_i b_right
1.0<=left<b_left:第i位可取j,第0~i-1位任取, b_left*10^(i-1)
2.left==b_left
2.1 j<b_i:j比b的第i位小,第i位可取j,共10^(i-1)
2.2
   ↪ j==b_i:j比b的第i位小,第i位可取j, 但后几位必须小于b_right,共b_right+1
2.3 j>b_i,j比b的第i位大,第i位不可取j,共0种
共b_left*10^(i-1)+10^(i-1)/b_right+1/0

但是需要注意j为0的情况,j为0那么前面的数不能为0,同时j不能出现在最高位,也就是讨论如下:
```

```
1.0<left<b_left:第i位可取j,第0~i-1位任取, (b_left-1)*10^(i-1)
2.left==b_left
2.1 j<b_i:j比b的第i位小,第i位可取j,共10^(i-1)
2.2
    ↪ j==b_i:j和b的第i位想等,第i位可取j, 但后几位必须小于b_right,共b_right+1
2.3 j>b_i,j比b的第i位大,第i位不可取j,共0种
(b_left-1)*10^(i-1)+10^(i-1)/b_right,共b_right+1/0
综合相比j!=0情况直接减去一个10^(i-1)

答案:对于j,累加i就是数字j出现的次数
```

- 代码实现

```cpp
#include<iostream>
#include <cmath>

using namespace std;

int digit(int n){

   int ret = 0;
   while(n>0){
      n/=10;
      ret++;
   }
   return ret;
}
int getLeft(int n,int i){
   while(i--){
      n/=10;
   }
   return n;
}
int getRight(int n,int i){
   int shift = 1;
   while(--i){
```

```
24      shift*=10;
25   }
26   return n%shift;
27}
28int num(int n,int i){
29   while(--i){
30      n/=10;
31   }
32   return n%10;
33}
34int count(int n,int j){ //返回[1,n]当中j出现的次数
35   int ret = 0;
36   int dig = digit(n);
37   if(j!=0){
38      for(int i=1;i<=dig;i++){
39         int n_left = getLeft(n,i);
40         int n_right = getRight(n,i);
41         ret += n_left*pow(10,i-1);
42         int n_i = num(n,i);
43         if(j<n_i)ret += pow(10,i-1);
44         else if(j==n_i) ret += n_right+1;
45      }
46      return ret;
47   }
48   for(int i=1;i<dig;i++){ // 0不能是最高位,并且前面的数不能全是0
49      int n_left = getLeft(n,i);
50      int n_right = getRight(n,i);
51      ret += (n_left-1)*pow(10,i-1);
52      int n_i = num(n,i);
53      if(j<n_i)ret += pow(10,i-1);
54      else if(j==n_i) ret += n_right+1;
55   }
56   return ret;
57}
58int main(){
```

```
    int a,b;
    while(cin>>a>>b,a||b){
        if(a>b)swap(a,b);
        for(int j=0;j<10;j++)cout<<count(b,j)-count(a-1,j)<<" ";
        cout<<endl;
    }
    return 0;
}
```

- 简化

  算 n_left 和 n_right 可以更快，并且可以将 0 和其他情况合并

  j=0, 考虑到高位不能是 0, 直接减去 1 个 pow(10,i-1)

```
#include<iostream>
#include <cmath>

using namespace std;

int digit(int n){

    int ret = 0;
    while(n>0){
        n/=10;
        ret++;
    }
    return ret;
}
int count(int n,int j){ //返回[1,n]当中j出现的次数
    int ret = 0;
    int dig = digit(n);
    for(int i=1;i<=dig;i++){
        int t = pow(10,i-1);
        int n_left = n/t/10;
        int n_right = n%t;
        int n_i = n/t%10;
        ret += n_left*pow(10,i-1);
```

```
        if(j==0) ret -= pow(10,i-1);//j=0时高位不能是0,减去1个
        if(j<n_i)ret += pow(10,i-1);
        if(j==n_i)ret += n_right+1;
    }
    return ret;
}
int main(){
    int a,b;
    while(cin>>a>>b,a||b){
        if(a>b)swap(a,b);
        for(int j=0;j<10;j++)cout<<count(b,j)-count(a-1,j)<<" ";
        cout<<endl;
    }
    return 0;
}
```

7.9 记忆化搜索

▶ 滑雪

- 思路分析

```
状态表示:dp(i,j)
集合:从第i行第j列开始的滑雪路径
属性:最长滑雪轨迹

状态计算
集合划分:根据第一步的方向不同计算
上:dp(i+1,j)
下:dp(i-1,j)
左:dp(i,j-1)
右:dp(i,j+1)

状态转移:求最大
dp(i,j)=max(dp(i-1,j),dp(i+1,j),dp(i,j-1),dp(i,j+1))+1,当然这里需要满足num(i,j)>nu

为了方便这里实际上是用递归实现
```

- 代码实现

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

const int N = 310;
int R, C;
int dp[N][N];
int num[N][N];
int dx[4]={-1,0,0,1};
int dy[4]={0,-1,1,0};
int getDp(int i,int j){
   if(dp[i][j]!=-1)return dp[i][j];
      ↪ //如果已经计算过，直接返回答案，防止死循环
   dp[i][j]=1; //当前区域最小也是1
   for(int k=0;k<4;k++){
      if(num[i][j]>num[i+dx[k]][j+dy[k]]&&i+dx[k]>=1&&i+dx[k]<=R&&j+dy[k]>=1&&j+dy[
         dp[i][j]=max(dp[i][j],getDp(i+dx[k],j+dy[k])+1);
      }
   }
   return dp[i][j];
}
int main(){
   cin>>R>>C;
   for(int i=1;i<=R;i++){
      for(int j=1;j<=C;j++){
         cin>>num[i][j];
      }
   }
   int length = 0;
   memset(dp,-1,sizeof dp);
   for(int i=1;i<=R;i++){
      for(int j=1;j<=C;j++){
```

```
32          length = max(length,getDp(i,j));
33        }
34    }
35    cout<<length;
36    return 0;
37 }
```

八、 高级算法

8.1 FFT

九、 高级数据结构

9.1 树状数组

树状数组是一种用于高效处理动态数据集的数据结构，通常用于处理动态数组的单点更新和区间查询 (均为 $\log n$)。

假设目标区间是 [1,R], 对区间右端点根据二进制拆分，有

$R = 2^{x_1} + 2^{x_2} + 2^{x_3} + ... + 2^{x_k}, x_1 < x_2 < ... < x_k$

这样，区间右端点 R 可以这样划分:

$$[1, 2^{x_1}]$$

$$[2^{x_1} + 1, 2^{x_1} + 2^{x_2}]$$

$$...$$

$$[2^{x_1} + 2^{x_2} + ... + 2^{x_{k-1}} + 1, 2^{x_1} + 2^{x_2} + ... + 2^{x_k}]$$

其中每个区间长度刚好是右端点的 lowbit 值，所以我们可以用一个数组 tr 存储这段区间和，它就是树状数组。

原数组 a[i]，树状数组 tr[i] = sum((a[i] - lowbit(a[i]), a[i]])(左开右闭)。

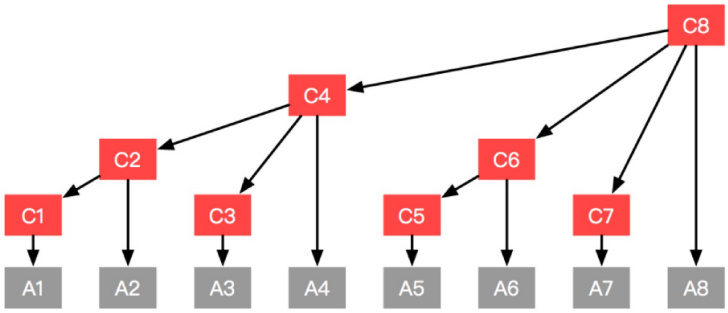


图 **27**

9.1.1 单点修改，区间查询

单点修改和区间查询操作是树状数组最基本的应用，由于复杂度均为 $\log n$，因此一般前缀和的题目它都能做，甚至更快。

- **单点修改**

如果要修改 a[i] 的值，显然，它对应的树状数组 tr[i] 以及所有会累加 tr[i] 的值都要修改，而 tr[i] 的父节点是 tr[i] + lowbit(tr[i])，我们需要从 tr[i] 向后迭代修改，代码如下：

```
void add(int k, int v){//原数组k位置增加v对应的操作
   for(int i = k; i <= n; i += lowbit(i))tr[i] += v;
}
```

初始建立树状数组也可以通过单点修改 (修改可以转换成设置) 得到。

- **区间查询**

区间查询通常是查询原数组 [1,i] 的区间和，由于 tr[i] 存储的是 a[i] - lowbit(a[i]) 到 a[i] 的左开右闭区间和，我们可以从 tr[i] 向前迭代累加，代码如下：

```
int query(int k){//返回原数组[1,k]的和
   int res = 0;
   for(int i = k; i > 0; i -= lowbit(i))res += tr[i];
   return res;
}
```

其余任意区间和可由前缀和得出。

▶ 逆序对的数量

这道题想了很久很久，不过终于还是明白了，网上很多帖子没有讲清而且有的还是错误的，重要的一点是还是要有深入思考和批判性思维的能力

- 思路分析

  这道题要求的是数组中逆序对的数量，我们由浅入深思考。首先最简单的做法就是暴力枚举，比如我们从前往后看每个数它后面的数有多少个比它小，统计，但我更倾向于从前往后看每个数它前面有多少个数比它大，尽管目前暴力枚举它们两个复杂度一样，但是后者可以动态计算，也就是我不必预先确定整个数组，可以逐个加入逐个计算。

  然而，这种暴力穷举的复杂度通常是不可接受的，对于当前数 a[i]，我需要枚举前面每个数 a[j]，看看多少个数比 a[i] 大。我们想能不能直接得出，也就是预先存储已经统计过的每个数的出现次数，再求一个前缀和 b[i]，表示 [1,i] 个数出现的次数，这样遇到 a[i] 的时候不就可以直接通过 a[N] - a[i] 得到吗。但是，现在问题是一般的前缀和查询虽然是 O(1)，但是它每次更新复杂度是 O(n)，这就对我们的数据结构提出了时间复杂度的要求。

  因而，明确到这道题，首先可以前缀和实现，并且涉及动态求前缀和，因而可以用树状数组做，它修改和查询的复杂度都是 O($\log n$)。

- 代码实现

```cpp
#include<bits/stdc++.h>
#define int long long
#define INF 0x3f3f3f3f3f3f3f3f
using namespace std;

const int N = 1e5 + 10;

int n;
int a[N], tr[N];//tr是树状数组
vector<int> nums;
int res;

int lowbit(int x){
   return x & -x;
}

void add(int k, int v){
   for(int i = k; i <= n; i += lowbit(i))tr[i] += v;
}
int query(int k){
```

```
    int res = 0;
    for(int i = k; i > 0; i -= lowbit(i))res += tr[i];
    return res;
}

int find(int x){
    return lower_bound(nums.begin(), nums.end(), x) - nums.begin() + 1;
}
signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n;
    for(int i = 1; i <= n; i ++)cin>>a[i], nums.push_back(a[i]);

    sort(nums.begin(), nums.end());
    nums.erase(unique(nums.begin(), nums.end()), nums.end());
    for(int i = 1; i <= n; i ++){//每次从前往后枚举原数组的每个数
        int t = find(a[i]);//查询当前枚举的数的相对大小排名
        res += query(n) - query(t); //前面的n是树状数组的长度

        cout<<a[i]<<" "<<find(a[i])<<endl;

        add(t, 1);//将当前数的名次加入
              //实际上这里隐含了树状数组的原数组存储的是每个数的相对大小
    }
    cout<<res;
    return 0;
}
```

9.1.2 单点查询，区间修改

树状数组也可以做到单点查询，区间修改。这是通过差分的思想得到的。我们要使得原数组 [l,r] 的值 +c，或者要求某个值，可以维护一个差分 b_i 的前缀和，[l,r]+c 相当于差分数组 [l] + c，[r + 1] - c，查询某个值相当于求其前缀和。

▶ 题目：一个简单的整数问题

```
#include<bits/stdc++.h>
#define int long long
#define INF 0x3f3f3f3f3f3f3f3f

using namespace std;

const int N = 1e5 + 10;
int n, m;
int a[N], tr[N];//a是原数组，tr是树状数组，对应的原数组是差分数组

int lowbit(int x){
   return x & -x;
}
void add(int k, int v){
   for(int i = k; i <= n; i += lowbit(i))tr[i] += v;
}

int query(int k){
   int res = 0;
   for(int i = k; i > 0; i -= lowbit(i))res += tr[i];
   return res;
}


signed main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   cin>>n>>m;
   for(int i = 1; i <= n; i ++){
      cin>>a[i];
      add(i, a[i] - a[i - 1]);
   }
   while(m --){
```

```
35      char op;
36      cin>>op;
37      if(op == 'C'){
38         int l, r, d;
39         cin>>l>>r>>d;
40         add(l, d);
41         add(r + 1, -d);
42      }
43      else{
44         int x;
45         cin>>x;
46         cout<<query(x)<<endl;
47      }
48   }
49
50   return 0;
51}
```

9.1.3 区间查询，区间修改

树状数组借助差分数组实现区间修改，此外，进一步递推，也可实现区间查询。

在上面的基础上，假设我们要求出 [1,r] 的区间和，b_i 是差分数组，则有

$$\sum_{i=1}^{r} a_i = \sum_{i=1}^{r}\sum_{j=1}^{i} b_j = (r+1)\sum_{i=1}^{r} b_i - \sum_{i=1}^{r} i * b_i$$

(这一步需要用互补思想，将 b1,b1+b2,...b1+b2+...+br 对称补上再减去)

这样，我们需要维护 b_i 和 $i * b_i$ 两个差分数组的前缀和也就是树状数组，求区间的和的话前面乘以一个 $(r+1)$ 再相减。

▶ 一个简单的整数问题 2

```
1 #include<bits/stdc++.h>
2 #define int long long
3 #define INF 0x3f3f3f3f3f3f3f3f
4
5 using namespace std;
6
```

```cpp
const int N = 1e5 + 10;
int n, m;
int a[N], tr1[N], tr2[N];//a是原数组，tr1是bi的前缀和，tr2是i * bi的前缀和

int lowbit(int x){
   return x & -x;
}
void add(int tr[], int k, int v){
   for(int i = k; i <= n; i += lowbit(i))tr[i] += v;
}



int query(int tr[], int k){
   int res = 0;
   for(int i = k; i > 0; i -= lowbit(i))res += tr[i];
   return res;
}
int sum(int k){

   return (k + 1) * query(tr1, k) - query(tr2, k);
}

signed main(){
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   cin>>n>>m;
   for(int i = 1; i <= n; i ++){
      cin>>a[i];
      add(tr1, i, a[i] - a[i - 1]);
      add(tr2, i, i * (a[i] - a[i - 1]));
   }
   while(m --){
      char op;
```

```
        cin>>op;
        if(op == 'C'){
            int l, r, d;
            cin>>l>>r>>d;
            add(tr1, l, d);
            add(tr1, r + 1, -d);
            add(tr2, l, l * d);//按照a[i]的修改而修改
            add(tr2, r + 1, (r + 1) * (-d));
        }
        else{
            int l, r;
            cin>>l>>r;
            cout<<sum(r) - sum(l - 1)<<endl;
        }
    }
    return 0;
}
```

9.2 线段树

参考文献

[1] AcWing