贺菜狗机考准备

> **开始时间:** 2023/03/11
>
> **准备目标:** 在各大高校的机试中尽量少丢分，不被其他竞争者拉开差距
>
> **目前成果：** **厦门大学**计科系夏令营机试**450**分(满分470分)，机试排名前20。

知识点积累

数理基础

最大公约数和最小公倍数

一般来说，是**通过求两个数的最大公约数**来求最小公倍数。

至于最大公约数的求法，一般是用**辗转相除法**。其步骤如下：

> 即对于a和b，选其中**较大**的数做**被除数x**，**较小**的做**除数y**，二者除法的**余数**为c。
>
> - **c = x%y**；
> - 若c!=0，则x=y; y=c。转第一步。
> - 若c==0，则y就是最大公约数。

当我们求得两个数的最大公约数后，即可利用最大公约数求出最小公倍数。

> **基本理念**： **a * b = 最小公倍数*最大公约数**

快速排序

快排是非常经典且基础的算法，虽然日常用sort就能够节省大量的代码空间，但如果说学校限制只能使用C语言，那么不得不手撸快排。

算法思路

快排的思路很简单，这里采用**挖坑法**来实现。具体按下面的步骤进行。

> 1. 对下标范围为 **[0,n-1]** 的区间进行排序，选定区间**最左**或**最右**或**中间值**作为**基准值**。
> 2. 进入一个循环，当 左值针(ll)小于右指针(rr) 时，循环继续进行。
> 3. 从**右侧**开始，在**保证左右指针合法**的情况下，找到**第一个** 右指针指向的值**小于基准值**的位置，用这个值**覆盖**掉 左指针所指位置的元素。
> 4. 再从**左侧**开始，同上限制，找到第一个 左指针指向的值大于基准值的位置，用这个值覆盖掉 右指针所指位置的元素。
> 5. 当循环结束，此时左右指针是**相等**的，并且指向的位置就是基准值应该放置的位置。
> 6. 判断该位置左边元素个数是否大于1，若大于则进入**左区间**[1，ll-1]进行**递归**。

> 7. 判断该位置左边元素个数是否大于1，若大于则进入**右区间**[ll+1 , r]进行**递归。**

代码实现

```
void quick_sort(int l,int r){
    int ll = l, rr = r;
    // 选定基准值,这里采用区间最左端点
    int key = q[l];
    while(ll<rr){
        while(ll<rr&&q[rr]>=key) rr--;
        q[ll] = q[rr];
        while(ll<rr&&q[ll]<=key) ll++;
        q[rr] = q[ll];
    }
    q[ll] = key;
    if(l<ll) quick_sort(l,ll-1);
    if(r>ll) quick_sort(ll+1,r);
}
```

C语言

输入多组数据

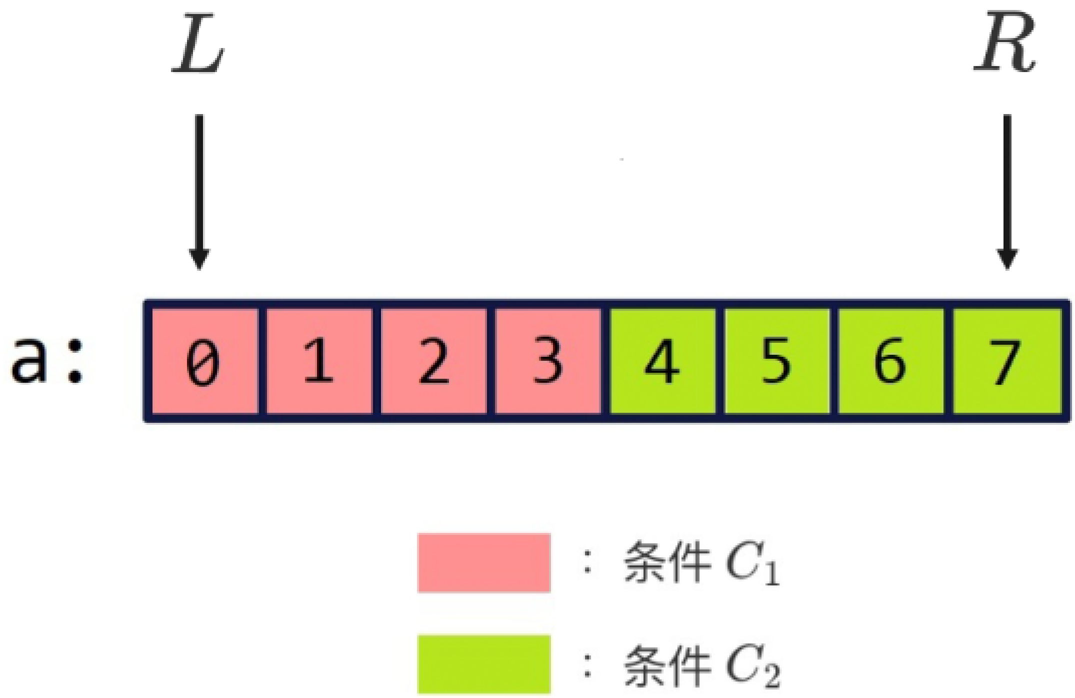
有的题目要求能够输入多组数据，而不是每次输入只有一组数据。为了应对这种要求，需要使用下面的代码来完成。

```
while(scanf("%d",&n)!=EOF){
    // 操作
}
```

整数二分

二分查找的过程中，会在每次查找的区间的中点处划分为两个子区间，下次查找将会在**符合条件**的那个子区间中继续进行。

注：二分查找只能在 *有序* 的序列内进行。



也就是上图这样，C1和C2是**互斥**的，区间只可能满足其中一个条件。

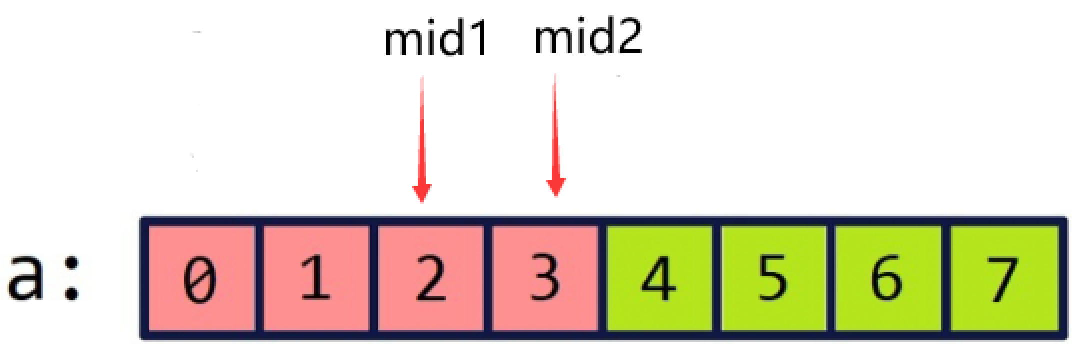
在进行二分之前，会根据题意定义一个**check函数**，该函数接收传入的mid，**判断其代表的区间是否符合条件**。而mid如何代表一个区间，就有下面的两种二分查找了。

寻找右边界的二分查找

思路

寻找右边界，就是指我们索要寻找的元素是区间的右边界。

当我们根据当前区间左右端点确定了mid后，**如果check(mid)==true**，那么mid是落在红色区的，有可能是右边界，此时通过让 **l = mid**，即可缩短查找范围；**若check(mid)==false**，那么mid是落在绿色区的，一定不会是我们要找的右边界，因此让 **r = mid-1**即可。



当 l == r 时，l 指向的位置就是我们要查找的元素的位置。

需要注意的是，**如果采用mid = (l+r)/2**，那么由于**C语言整数除法向下取整**的特点，可能出现 **r - l = 1**的情况，此时**(l+r)/2 = (l+l+1)/2 = l**，也就是说区间仍为[l，r]从而陷入死循环。

因此，查找右边界的二分法的mid计算公式如下：

$$(l+r+1)/2$$

以此来进行**向上取整**，从而避免死循环。

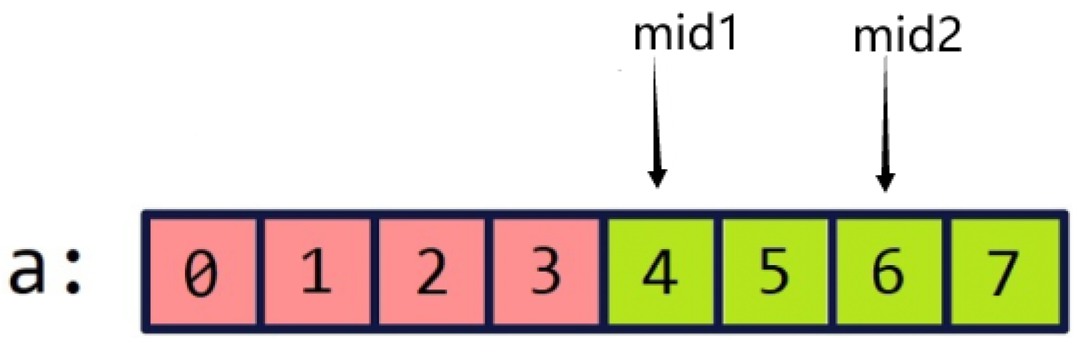
代码

```
int b_search1(int l,int r){
    // 右移一位,实际上就是除以2
    int mid;
    while(l<r){
        mid = l+r+1>>1;
        if(check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

寻找左边界的二分查找

大体思路是差不多的，只不过这次查找的是区域的左边界，也就是绿色区域的左边界。

如果符合条件，那么 mid 就是在绿色区间的，有可能是左端点。我们要查找的是这个区间的左边界，那么应该**让 r = mid**；**若不符合条件**，那么mid一定是在红色区域，那么一定不是我们要找的左边界，因此**让 l = mid + 1**；



而由于我们就是要找到左边界，而向下取整不会导致我们错过左边界，因此mid就正常取中心点向下取整即可。

代码

```
int b_search2(int l,int r){
    int mid;
    while(l<r){
        mid = l+r>>1;
        if(check(mid)) r = mid;
        else l = mid+1;
    }
    return l;
}
```

注：由于两种二分查找所查找的区间是不一样的，因此check函数也是不一样的，写题时要注意！

例题

Acwing789.数的范围

题目会给出一个**递增**的，**可重复**的序列。接下来会进行若干次查询，每次查询要求输出所查询元素所在位置的**起始下标**和终止下标。

思路

> 基于上面提到的两种写法，如果采用寻找左边界，那么我们第一轮可以找出**第一个**等于所查询元素的位置下标；接下来再通过寻找右边界，找到**最后一个**等于所查询元素的位置下标，即可解决此题。

代码

```c
#include<stdio.h>
int arr[100010],k;

int main(){
    int n,q,i,mid,l,r;
    scanf("%d%d",&n,&q);
    for(i=0;i<n;i++) scanf("%d",&arr[i]);
    while(q--){
        scanf("%d",&k);
        l = 0;
        r = n-1;
        // 寻找左边界
        while(l<r){
            // >>1表示右移1位,实际上就是除以2
            mid = l+r>>1;
            if(arr[mid]>=k) r = mid;
            else l = mid+1;
        }
        if(arr[l]!=k){
            // 说明数组里根本就没有
            puts("-1 -1");
        }else{
            // 寻找右边界
            printf("%d ",l);
            l = 0,r = n-1;
            while(l<r){
                mid = l+r+1>>1;
                if(arr[mid]<=k) l = mid;
                else r = mid-1;
            }
            printf("%d\n",r);
        }
    }
    return 0;
}
```

链表

动态链表往往适用于面试答题，但对于机试就不适用了，因为分配空间十分耗时。所以为了能够稳定应对机试，应通过静态链表的方式来实现链表，即使用**数组**模拟。

单链表

单链表的结构，即每个节点存放**值**和**指向下一节点的指针**。如果要用数组来模拟，则需要两个数组。

部分操作代码

```
// val[i] 存放第i个插入的节点，ne[i] 存放第i个插入节点的下一个节点的插入位序(在数组中的位置)
int val[N],ne[N];
// head指向队首，idx指向下一个可插入的数组单元(和队列的tt同理)
int head,idx;

void init(){
    // 链表为空
    head = -1;
    idx = 0;
}

// 在链表头部插入一个节点
void insert_to_head(int x){
    val[idx] = x;
    ne[idx] = head;
    head = idx;
    idx++;
}

// 在第k个插入的节点后面插入一个节点
void insert_after_k(int x,int k){
    // 第k个插入的数在数组的下标是k-1
    // 取得它的下一个节点位序
    int nex = ne[k-1];
    val[idx] = x;
    ne[idx] = nex;
    ne[k-1] = idx;
    idx++;
}

// 删除第k个插入的数后面的数
void delete_after_k(int k){
    // k==0 删除头节点
    if(k==0){
        head = ne[head];
        return ;
    }
    int nex = ne[k-1];
    ne[k-1] = ne[nex];
}
```

理解了数组怎么模拟链表后，按照链表各种操作的特性就可以很轻易的写出操作的函数，机试尽可能还是用这种办法来做。

双链表

双链表即每个节点新增一个指向上一节点的指针，那么用数组模拟的话就新开一个数组，用来存放上一节点指针。

部分操作代码

```
// 舍弃前两个单元用作首尾指针，且二者相连，即r[0] = 1,l[1] = 0;
int l[N],r[N],val[N];
int idx;
void init(){
    l[1] = 0;
    r[0] = 1;
    idx = 2;// 前两个空间占用了，故从2开始
}

// 在第k个插入的数右侧加入一个新值x
// 若让在第k个插入的数左侧新增值x，则就是在k的左侧的值的右侧新增值，即调用add(l[k],x)即可
void add(int k,int x){
    // 得出真实下标 +2是因为第一个数放在 下标为2的单元内
    k = k-1+2;
    int ne = r[k];
    val[idx] = x;
    r[idx] = ne;
    l[idx] = k;
    r[k] = idx;
    idx++;
}

void remove(int k){
    // 获取正确下标
    k = k-1+2;
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}
```

队列

定义

> **队列**是一个**只**能在**一端插入**，**一端删除**的数据结构，且队列中的元素维持着**先进先出**的特点。在做题中，一般选择采用**双指针**+**数组**来模拟，从而降低时间复杂度。

代码实现

```
// hh==tt时表示队列为空,tt指向下一个要插入的位置
int hh=0,tt=0;
// 数组模拟队列
int q[N];
/*
    当然,也可以采取下面的方案2,使tt指向队尾元素。
    int hh=0,tt=-1;
*/
void insert(int val){
    q[tt++] = val;
    // 若采取方案2,则是q[++tt] = val;
```

```
}
// 弹出队头元素
int pop(){
    return q[hh++];
}
// 弹出队尾元素
int pop_back(){
    return q[--tt];
    // 若采用方案2,则为return q[tt--];
}
```

应用

宽度优先搜索BFS

图和树的遍历算法中，宽度优先算法的思想是在访问一个点时，将其所有的点都进行访问，再通过这些点继续扩展，直到所有节点均访问完成。队列**先进先出**的特点就能够很好的解决此问题。

单调队列

定义

单调队列，就是保证队列**内部元素**时刻保持**单调递增或单调递减**，从而能够针对某些问题达到优化的效果。

例题

滑动窗口

题目描述

> 题目给定窗口大小，要求能够输出每个窗口下的最小值和最大值。由于窗口大小固定，且不断右移，那么可以**使用队列来模拟窗口**，但要求窗口内部的最大最小值则需要每次移动都全部遍历一遍，这会产生巨大的时间开销。
>
> 如果这个队列时刻能够保持单调，那么**队头元素**就是窗口内元素的最小(大)值，则无需遍历，从而优化了时间复杂度。
>
> **思路**：维护一个单调队列，每次考虑的都是当前窗口的最后一个元素。在插入新元素前，先判断队头元素是否还在以当前元素为结尾的窗口内，若不在，则删除队头元素；接下来再从队尾开始向前遍历，把所有比当前值**大或等于**的值都从队尾出队，然后再将**当前值**插入队列**尾部**。

代码

```
#include<stdio.h>
int n,k,hh,tt;
int arr[1000010],q[1000010];
// 输出窗口最小值,维护一个单调增队列,队首为最小值
void method1(){
    int i,j;
    hh=0;tt=0;
    for(i=0;i<n;i++){
        // 判断队首元素是否还在当前窗口内,若不在则弹出
        if(hh<tt&&i-k+1>q[hh]) hh++;
        // 从末尾向前把队列里所有比当前值大的值都清除
        while(hh<tt&&arr[q[tt-1]]>=arr[i]) tt--;
        // 将当前值的下标存入队列
```

```
        q[tt++] = i;
        if(i>=k-1) printf("%d ",arr[q[hh]]);
    }
    puts("");
}
// 输出窗口最大值
void method2(){
    int i,j;
    hh=0;tt=0;
    for(i=0;i<n;i++){
        // 判断队首元素是否还在当前窗口内,若不在则弹出
        if(hh<tt&&i-k+1>q[hh]) hh++;
        // 从末尾向前把队列里所有比当前值大的值都清除
        while(hh<tt&&arr[q[tt-1]]<=arr[i]) tt--;
        // 将当前值的下标存入队列
        q[tt++] = i;
        if(i>=k-1) printf("%d ",arr[q[hh]]);
    }
    puts("");
}
int main(){
    int i;
    scanf("%d%d",&n,&k);
    for(i=0;i<n;i++) scanf("%d",&arr[i]);
    method1();
    method2();
    return 0;
}
```

栈

定义

> **栈**是一个**先进后出**的数据结构，根据这个特性，可以解决很多的问题。
>
> **栈**是一个**线性**的结构，只支持**查看栈顶元素**、**弹出栈顶元素(pop)**、**入栈**操作。

代码实现

```
// top指向当前栈顶元素
int top=0;
int st[N];
// 入栈
void push(int x){
    st[++top] = x;
}
// 出栈操作,我这里还顺便返回了栈顶元素
int pop(){
    return st[top--];
}
```

当然，栈并不局限于整型数据，任何种类的数据都可以，取决于具体问题的应用，这里仅是为了方便展示代码。

应用

括号匹配

> **问题描述**：括号匹配是一个经典的问题了，说的是给你一串**只包含括号**的字符串，让你判断这个字符串是否合法。而每个括号要想合法，就必须**左右括号相匹配**，而最后出现的左括号往往要与最先出现的右括号进行匹配，因此可以应用栈的**先进后出**特点。
>
> 例如: (((<{}>)))合法、{{(}}不合法...
>
> **思路**：遇到任何一种**左括号**都**入栈**，遇到一个**右括号**就拿它跟栈顶元素进行**匹配**，如果匹配则栈顶**出栈**，否则直接结束判断并输出不匹配。

单调栈

所谓单调栈，就是保证**栈内**元素呈**单调**递增或递减，是针对某些特殊问题的一种**优化**。具体做法就是，在每个元素入栈的时候，先判断栈内元素在加入这个新元素后是否还"**有用**"，若没用，则**将所有没用的元素弹出**。

这样一来，任意时刻的栈内都一定能够保持单调，从而能够优化时间复杂度。

例题

Acwing830.单调栈

这题要求输出每个元素**左侧第一个比它小**的数，没有则输出-1。根据上面单调栈的定义，不难想到：如果能够将每个数左侧的所有数维护成一个单调栈，那么栈空就表示没有比当前数更小的，栈不空则输出栈顶元素即可。

```c
#include<stdio.h>
int N;
int arr[100010],result[100010],cnt = 0;
int st[100010],top=0;
int main(){
    int i,x;
    scanf("%d",&N);
    st[0] = 100000000;
    for(i = 0;i<N;i++){
        scanf("%d",&arr[i]);
        while(top>0&&st[top]>=arr[i]){
            top--;
        }
        if(top==0){
            result[cnt++] = -1;
        }else{
            result[cnt++] = st[top];
        }
        st[++top] = arr[i];
    }
    for(i=0;i<cnt;i++) printf("%d ",result[i]);
    printf("\n");
    return 0;
}
```

并查集

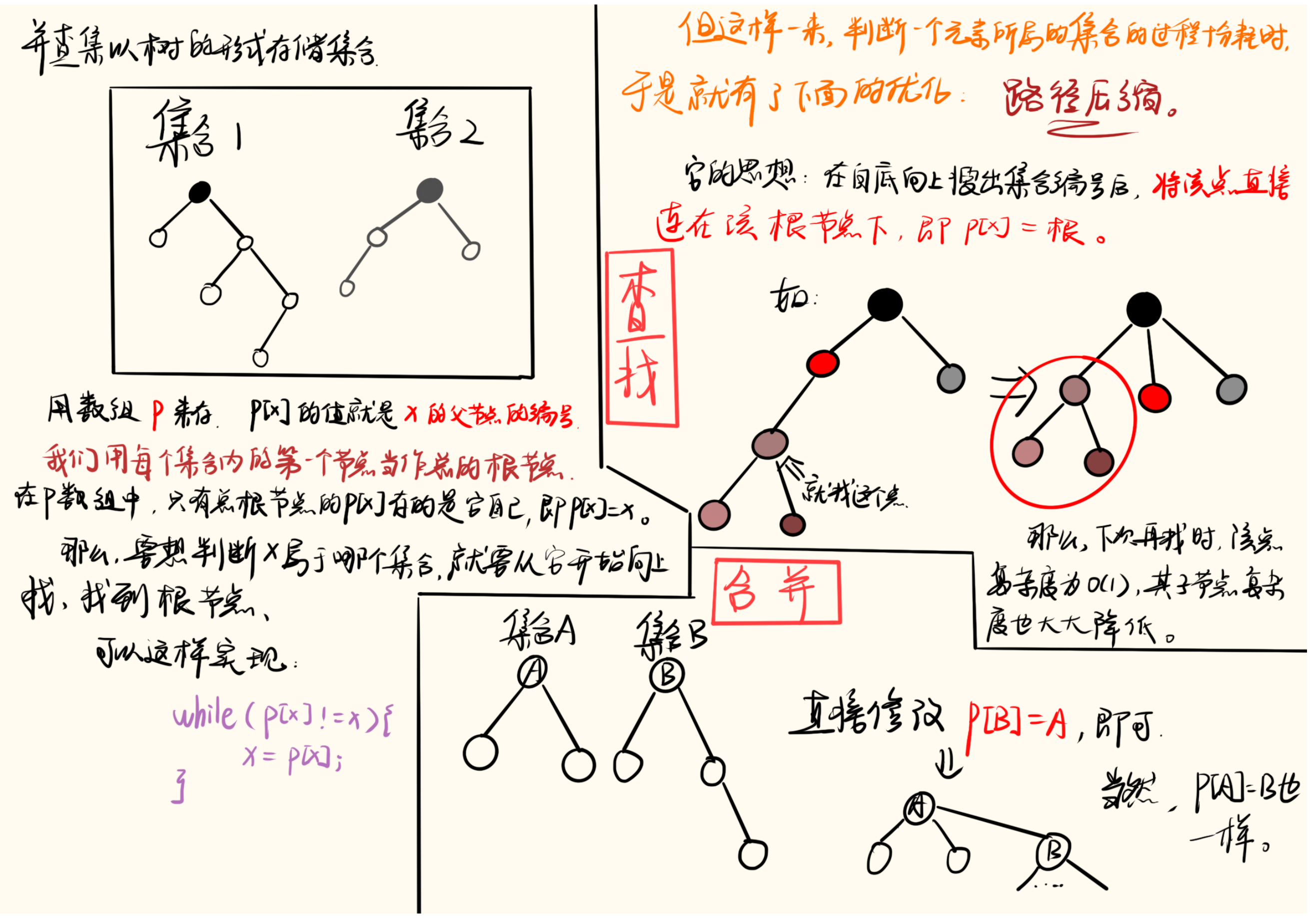
并查集解决的问题

并查集是通过**树**的数据结构来存储集合的，即每一个集合对应一个树，而每个集合内的第一个数的编号就充当这棵树的根节点。并查集的查询和合并操作的时间复杂度均 **近似O(1)**。

常见的并查集用法

> - 查询两个元素是否属于同一集合
> - 合并两个集合
>   (其实所谓**集合**，在**图**中也可以理解为一个**连通子图**)
> - **查询**一个图中的**连通子图个数**
> - **查询**一个连通子图中的**节点个数**

并查集图解



并查集的基本代码写法

存储结构

我们使用一个**p数组**来存，p[x] 表示元素x在集合树中的**父节点编号。**

那么，如果 **p[x]** = x ，则x**就是这个集合树的根节点。**

初始化

```
// 即初始状态是让每个元素自己是一个集合
for(int i=1;i<=N;i++) p[i] = i;
```

find函数

```
// 并查集查找某个元素所属集合的编号
int find(int x){
    // 这里的思想是路径压缩
    // 在对每个元素完成一次查找后 我们已经找到这个元素所属集合树的根节点了
    // 直接就将这个元素连接在根节点下
    // 那么下一次查找的复杂度就是 o(1) 同时当前节点的子节点查找时间也将减少
    if(p[x]!=x) p[x] = find(p[x]);
    return p[x];
}
```

合并操作

```
cin>>a>>b;
if(find(a)!=find(b)){
    // a和b不是一个子图里的,子图数-1
    subG--;
    // 由于a和b的边已经被考虑了,所以把a和b所属的集合合并
    // 由于上面已经执行了find(a) 那么这里执行find(a)就是o(1),不用担心时间复杂度的问题
    p[find(a)] = find(b);
}
```

关于上面的合并，其实谁合并进谁的集合树，并不重要，**只要能够合并**即可。上面的例子就是**将a所在的集合合并进了b所在的集合**下。

新的认识

上面说到，并查集是能够进行合并和查找的。通过这几天的做题，我对合并的过程有了新的理解。

我们先看下下面这步**合并**的代码写法：

```
if(find(a)!=find(b)){
    p[find(a)] = find(b);
}
```

上例中，a和b所属集合不一样，所以我们将a所属的集合合并到b所属的集合下。

也就是说，从a向上走，走到a所属集合的根节点时，将这个根节点直接连在b所属集合的根节点下面。但是find函数是递归的过程，采用了路径压缩的想法，那么这句话执行以后，a所属集合下的所有元素**通过find函数**找到的根都将**等于**find(b)。

那么，如果我们要解决的问题可以被视作寻找子图的问题的话，通过若干次上面的合并，我们就已经把整个图划分成了多个子图。每个子图内所有元素通过find函数找到的根都是一样的，即**find函数返回值是一样的。**

那么，我们可以记录find函数有多少个不同的返回值，那么每个返回值都将代表一个子图，那么**子图数**就出来了；同时，我们还可以记录每种返回值出现的次数，这样我们又知道了**每个子图的元素个数**；如果我们再把find返回值相同的元素记录在一个容器里，我们也就知道了**每个子图内有哪些元素**了...以此类推，会发现并查集能够很好的解决这些复杂的划分问题，PAT上面也有很多类似的题目。

那么，解决**划分子图**问题的**思路**大致如下：

> - 根据题意，构建图(有向图或者无向图都可以，自己根据图结构调整代码)
> - 将所有节点都视为一个单独的集合，即p[i] = i，设置子图数为N
> - 对于所有存在边的两个点，判断是否已经已经合并，如果没合并就合并
> - 合并后将子图数-1，即**N--**
> - 全部执行完后，我们就得到了N个子图，每个子图内的所有元素的find返回值均相同。
> - 根据题意利用划分好的子图来做出解答

查询某个集合的元素个数

上面大部分是在合并，或者是根据合并结果统计各个子图的元素数，但如果我指定要**查询某元素所在集合的元素数**，又该怎么办呢？

这两天做题得到了新思路，维护一个**元素个数数组 size**，**size[find(i)]** 表示 i 所在集合的元素个数，即**只有每个集合根节点所对应的size值才是有效的。**

维护方法：在合并时动态维护

```
if(find(a)!=find(b)){
    size[find(b)] += size[find(a)];
    p[find(a)] = find(b);
}
```

树

根据中、后序遍历创建树

对应题目：PAT甲级真题1020（当然，这题还要求输出此树的层序遍历结果。）

思想：

> 通过**递归**来构建树，每一个递归区间都是一棵**子树**。
>
> 那么在这棵子树中，后序遍历中的最后一个元素就是这棵**子树**的根节点。
>
> 接下来我们需要找到这个根节点在中序遍历中的位置，它的左侧元素就是它的左子树，它的右侧元素就是它的右子树，再进入左右子树完成**递归建树**。

代码实现

```
int N;
int inorder[40];
int postorder[40];
// 存储后序遍历中元素在中序遍历里对应的下标
unordered_map<int,int> post_in_mid;
// 存储每个节点的左右子树的根节点
```

```
unordered_map<int,int> l_son,r_son;

// 递归建树,返回值为根节点
int build(int il,int ir,int pl,int pr){
    // 每个根节点都是这个子树区间的后序遍历的最后一个值
    int root = postorder[pr];
    // 取得根节点在中序遍历中的下标
    int k = post_in_mid[root];
    if(il<k){
        // 说明存在左子树
        // pl+(k-1-il) 是 因为无论是中序还是后序，子树节点个数是一样的
        // 那么区间长度也是一样的，是推导出来的
        l_son[root] = build(il,k-1,pl,pl+(k-1-il));
    }
    if(k<ir){
        // 存在右子树
        r_son[root] = build(k+1,ir,pl+(k-1-il)+1,pr-1);
    }
    return root;
}
```

这种类型的问题比较常见，在leetcode上也见到过类似的题，不过那题是用先、中序遍历来建树，再通过层序遍历输出树的每个节点。不过换汤不换药，都是一样的道理。

只不过，每个子树的根节点是先序遍历对应区间内的第一个值(因为 “根左右”)，然后同样找到根节点在中序遍历里对应的位置，左侧即为左子树，右侧即为右子树。

树的层序遍历

思想：

> **层序遍历**，对应的思想是bfs，即**宽度优先搜索**。我们需要维护一个**队列**，这个队列内存储层序遍历得到的节点排序。
>
> 我们利用两个指针来处理这个队列：**hh** 和 **tt**。
>
> 其中，**hh** 指向的是**当前要处理的根节点**，**tt** 指向的是队列中**下一个要填入的位置**。
>
> 在每一轮循环中，我们先通过**hh**指针取得当前需要处理的根节点，然后判断其**是否有左右子树**，如果有，那么我们取出左右子树的根节点，按**先左后右**的顺序将节点存入队列。
>
> 那么，当所有节点都进入队列后，**tt将不再变化**，当hh增加到比tt还大的时候，说明**没有需要处理的节点了**，此时循环结束，并且队列中的顺序就是层序遍历的顺序。

代码实现：

```
// 存储后序遍历中元素在中序遍历里对应的下标
unordered_map<int,int> post_in_mid;
// 存储每个节点的左右子树的根节点
unordered_map<int,int> l_son,r_son;
// 层序遍历队列
int q[40];

void bfs(int root){
```

```
    // 两个指针
    int hh=0,tt=0;
    int t;
    q[0] = root;
    while(hh<=tt){
        // 拿到当前队首的根节点 取完后队首指针后移
        t = q[hh++];
        // 如果左子树map里有key值为当前根节点
        // 说明该节点有左子树
        if(l_son.count(t)) q[++tt] = l_son[t];
        if(r_son.count(t)) q[++tt] = r_son[t];
    }
    cout<<q[0];
    for(int i=1;i<N;i++) cout<<" "<<q[i];
    cout<<endl;
}
```

组合题

Acwing1620,PAT1127:

"Z"型遍历，即奇数行从右向左层序遍历，偶数行从左到右做层序遍历。

这道题首先考察根据某两种遍历方式创建树，然后再考察层序遍历。题目很有意思，在3月份的PAT中也有一个遍历树的题，但当时时间不够，题目还没看就结束了。

整体思路就是先根据第一个知识点把树建出来，建的过程中要记录每一层节点的个数，树的结构我是用l,r数组模拟的。然后建好了就进行层序遍历，记录层序遍历的结果。

然后，针对层序遍历的结果，按照"Z"型遍历的规则来输出。我是按照层进行遍历，先判断这个层是奇数还是偶数，然后确定这一层的所有元素在层序遍历中的下标范围，然后从前往后或者从后往前输出即可。

解题新思路积累

关于树的题目有很多种考法，只记知识点难以应付多变的题型，这里将用于记录做题得到的新思路。

模拟获取完全二叉树某子树节点个数

Acwing 3471.二叉树

这道题的意思是，默认有一棵**已经建好的**，含有无数个点的**完全二叉树**。题目会给出一个n和m，n代表截断的位置，即限制完全二叉树的最大节点序号是n，m代表的是某个子树的根节点。我们要输出**以m为根**，且**最大节点标号不超过**n的这棵**子树的节点个数**。

一开始，我直接撸了个dfs，求每个节点左右子树节点个数，然后加和返回结果。这个代码能过8个测试点，距离AC还差俩测试点超时。代码如下：

```
int dfs(int root){
    if(root>n) return 0;
    int l = dfs(root*2);
    int r = dfs(root*2+1);
    return l+r+1;
}
```

可以看出，思路可以，但数据太大就一定会超时。我去看了其他同学的解题思路，得到了一种全新的思路。

其实根本不需要细化到每一个节点对应的子树都去算结点数，**只需要考虑子树的这一层是不是满的就行。**

也就是说，推算出每一层的**左右边界**节点序号 l 和 r ，**若 l 和 r 均小于等于n，说明这一层是满的**，而第i层最多节点个数为 2 ^(i-1)，加上即可。接下来不断向下推算下一层的左右边界，只要右边界还小于等于n，就可以继续向下延申。延申过程按照下述公式进行：

$$l = l * 2, r = r * 2 + 1, r <= n$$

当右边界超出n的范围后，延申结束，此时判断左边界是否还在范围内。若左边界在范围内，则最后一层仍有有效节点，且最后一个节点序号就是n。则此层的节点个数就是：n-l+1，加上即可。

图

图的存储

邻接矩阵就不说了，时间空间开销都较大，只是易于理解，做题一般不采用。

邻接表

邻接表，思想与树的**孩子表示法**一致。即每个节点拥有一个单链表，这个单链表里存放的就是图中与当前节点相连的所有节点。

其实在C++中用vector进行嵌套，也是邻接表的思想，但无奈于机考不能使用c++，故要掌握数组模拟方法。

代码实现

```
// 其实跟单链表的实现大致相同，只是邻接表有N个头，即每个节点都会在总链表中占据一段区域
int val[N],ne[N],he[N],idx;

// 头插法,将a与b之间连起一条边,这里是有向边
void add(int a,int b){
    val[idx] = b;
    ne[idx] = he[a];
    he[a] = idx++;
}
```

上面的实现方式十分巧妙，每个节点都拥有自己的链表，但这个链表只是公共链表的某个区域，因此存储空间不是O(N^2)而只是O(N)。

一般使用前要将所有节点的头指针置为-1 ，表示链表为**空**。

```
// 用memset为he内的所有空间赋值为-1
memset(he,-1,sizeof he);
```

图的最短路算法

DFS（深度优先搜索）

> - 适用范围比较宽泛，对于很多操作步骤比较奇怪的问题都可以考虑用DFS，没有固定的模板
> - **数据结构**：**栈**（递归函数栈）
> - **包含思想**：**回溯、剪枝**，这俩是用于优化递归过程，减少不必要计算的方法
> - **缺点：**可以搜到解，但解不一定是最优解，如下面的走迷宫问题，如果非要用DFS求最优解，则需要考虑所有情况，取出全局最优，时间开销大
> - **优点**：相比于BFS，空间要求小

很多经典的问题都是DFS思想，如树的前中后序遍历或建树，以及23年6月的PAT甲级的机器人走迷宫问题，都可以用DFS来做。

例题

全排列问题

> 即对于正整数n，按增序输出1~n的全排列。
>
> **思路**：通过DFS，在**每一层递归确定一位数字**，确定的方法就是遍历所有**未使用**的数字，然后将其设置为当前位数字，并标记已访问，进入下一位数字的递归；回到当前层时将该数字设为未放问，即**回溯**的思想。

代码

```c
#include<stdio.h>
int n;
int result[7],visited[8];

// t为当前确定第几位
void dfs(int t){
    if(t==n+1){
        // 说明n位数字已确定
        for(int i=1;i<=n;i++){
            printf("%d ",result[i]);
        }
        printf("\n");
        return ;
    }
    for(int i=1;i<=n;i++){
        if(visited[i]!=1){
            result[t] = i;
            visited[i] = 1;
            dfs(t+1);
            visited[i] = 0;
        }
    }
}

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

BFS（宽度优先搜索）

> - **适用情况：当所有边的权重相同，求单源最短路时适用。**
> - **关键词**：**“最短路”、“最短距离”、“最少步骤”**...等，若每一步所需的距离相同，则BFS一般是适用的。
> - **数据结构：队列**
> - **缺点**：相比于DFS，空间要求比较高
> - **优点**：当**图内所有边的长度均为1**时，那么第一次遍历到某个点时，一定是一条从起点到当前点的**最短路**，因为BFS是逐层扩展的，距离逐渐加1。

代码的写法同树层序遍历的写法，**执行思路**如下：

> 1. 起点入队，标记为已访问
> 2. 将队首元素的所有**未访问**的相邻节点入队，并标记为已访问；
> 3. 队首元素出队，若**队列为空**则结束执行，否则返回步骤2

例题

> **走迷宫问题**
>
> **原题链接**：https://www.acwing.com/problem/content/846/
>
> **题目概述**：起点是(1,1)，迷宫大小为n*m，求从起点出发走到终点(n,m)的最短路径(步数)，每个节点之间距离相同。
>
> **思路：**以BFS的思想，每次将队首元素周围可行的点入队，直到队首元素是终点时结束。用结构体维护每个节点的累计步数，那么**终点的累计步数就一定是最短路的长度。**

代码

```
#include<iostream>
using namespace std;
struct Node{
    int x,y,st;
};

int n,m;
int graph[110][110];
int way[4] = {-1,1,-1,1};
int visited[110][110];
int step;
Node allNodes[10010];
int hh = 0,tt = 0;
void bfs(){
    int x1,y1,x2,y2,ts;
    Node one;
    one.x = 1;
    one.y = 1;
    one.st = 0;
    allNodes[tt++] = one;
    visited[1][1] = 1;
    while(hh<tt){
        x1 = allNodes[hh].x;
        y1 = allNodes[hh].y;
        ts = allNodes[hh].st;
        if(x1==n&&y1==m) break;
```

```
        for(int i=0;i<4;i++){
            if(i<2){
                x2 = x1+way[i];
                y2 = y1;
            }else{
                y2 = y1+way[i];
                x2 = x1;
            }
            if(x1<=0||y2<=0||x2>n||y2>m) continue;
            if(visited[x2][y2]!=1&&graph[x2][y2]!=1){
                Node temp;
                temp.x = x2;
                temp.y = y2;
                temp.st = ts+1;
                allNodes[tt++] = temp;
                visited[x2][y2] = 1;
            }
        }
        hh++; // 出队
    }
}

int main(){
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            scanf("%d",&graph[i][j]);
    bfs();
    printf("%d\n",allNodes[hh].st);
    return 0;
}
```

朴素Dijkstra算法

思路

- dis[i]: 从起点到i号点的最短路径长度。(dis[起点]=0，其他的等于正无穷)
- ok[i]: 表示i号节点是否已经找到最短路。
- 采用**邻接矩阵**存储**稠密图**，采用**邻接表**存储**稀疏图。**

> 1. dis[起点]=0，dis[其他]=正无穷;
> 2. 共进行**n轮**循环，每轮能够确定一个点的最短路;
> 3. 找到当前**未处理**的点中，**dis值最小**的点，记为 t 。
> 4. 将t标记为已找到最短路，同时**考虑t作为中转节点**，以此**更新**所有点的dis值。公式为：**dis[j] = min(dis[j],dis[t]+w[t] [j]);**
> 5. 转至2，直至n轮结束

代码

```
int dijkstra(){
    int i,j,t,x;
    memset(dis,0x3f,sizeof dis);
    dis[1] = 0;
    for(i=1;i<=n;i++){
        t = -1;
```

```
        // 找到当前没处理的节点中距离最短的
        for(j=1;j<=n;j++){
            if(ok[j]!=1&&(t==-1||dis[t]>dis[j])){
                t = j;
            }
        }
        ok[t] = 1;
        for(j=1;j<=n;j++){
            x = w[t][j]+dis[t];
            dis[j] = x<dis[j]?x:dis[j];
        }
    }
    if(dis[n]==0x3f3f3f3f) return -1;
    return dis[n];
}
```

拓扑序列

拓扑排序适用于**有向无环图**，带有宽度优先搜索的基本思想，按照如下规则生成遍历序列:

> 1. 将所有**入度为0**的节点全部入队;
> 2. 当队列非空时，获得队首元素，并将队首元素出队;
> 3. 将队首元素向外伸出的所有边删除(让延申到的点的入度-1即可);
> 4. 若入度减1后，小于0了，则将该点也加入队列;
> 5. 回到2步;

等到遍历完成，队列中存放的便是拓扑序列。

Floyd算法

思路

> 该算法源自**动态规划**的思想，通过**邻接矩阵**的方式来定义图结构，然后枚举每个点作为中间节点，更新邻接矩阵的每个值，从而使邻接矩阵变成一个**多源最短路径**矩阵。

什么意思呢?

上面的最短路径，都是**同源**，即对于每一条最短路径，数组记录的是每个点在这条路上距离起点的最短距离。

而**Floyd算法**能够推算**多源最短路径**，即这个算法能够推算出多条不同源头的最短路径，g[i] [j]存的是从i到j的最短路径长度。

推算更新过程按照如下公式进行:

$$g[i][j] = min(g[i][j], g[i][k] + g[k][j])$$

k就是**中转节点**，**枚举每一个节点作为中转节点**，看是否能够缩短从i到j的最短距离。

代码

```
void floyd(){
    int i,j,k;
    for(k=1;k<=n;k++)
        for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                g[i][j] = min(g[i][j],g[i][k]+g[k][j]);
}
```

可以看到，此算法实现起来需要O(n^3)的时间复杂度。如果某道题给你一个图，还让你求出**多组** i 到 j的最短路径长度，可以采用这个算法。

最小生成树

朴素Prim算法

思路

dis[i]: **节点i**到以确定节点**集合**的最短距离。

> 1. 初始化所有**dis数组**值为MAX；
> 2. 找到当前**未确定**最小dis值的点，用它来更新所有其他点到**集合**的距离(注：集合包含的是所有已经确定最小dis的点)；
> 3. 把该节点设置为**已确定**，即加入到集合中去；
> 4. 回到2，共进行**n轮**循环；

代码

```
#include<stdio.h>
int n,m;
int w[510][510],st[510];
int dis[510];
// 朴素Prim算法
void prim(){
    int res=0;
    int i,j,t;
    // 全都初始化为0x3f3f3f3f，这是一个很大的值，用来标记未更新
    memset(dis,0x3f,sizeof dis);
    for(i=0;i<n;i++){
        t = -1;
        for(j=1;j<=n;j++){
            // 找到未确定的最小dis值点，这个点离集合最近
            if(st[j]!=1&&(t==-1||dis[t]>dis[j])){
                t = j;
            }
        }
        //如果已经经过一轮处理，还有点的dis[t]为初始值，说明不连通，不可能存在生成树
        if(i!=0 && dis[t]==0x3f3f3f3f){
            printf("impossible\n");
            return ;
        }
        //如果已经更新过一轮，则集合里已经有了一个确定的点，那么可以开始累计记录权重和了
        if(i!=0) res += dis[t];
        // 用t更新所有其他的点，规则就是看看t加入集合以后能不能缩短各点到集合的最短距离
        for(j=1;j<=n;j++) dis[j] = dis[j]<w[t][j]?dis[j]:w[t][j];
        st[t] = 1;
```

```
    }
    printf("%d\n",res);
}
```

Kruskal算法(常用)

思路

Kruskal算法是以边为单位进行考虑的，每次抽取出**未选中**的，且**左右端点不连通**的**最短**的边，将其加入边集合中。这其实就能应用到**并查集**的思想，对于给定的一个边，如果左右端点不在集合中，就将二者合并进同一集合，即**用并查集来维护这个构造最小生成树的过程。**

算法流程如下：

> 1. 将所有边**按照其权值**大小进行排序，排成**从小到大**的顺序。
> 2. 依次取出每一条边ab，若a和b是连通的则跳过，若不连通，则选中，并将a和b合并到同一集合中去。
> 3. 距离累加器加上新增的边，重复第2步。

假设有m条边，则该算法的**时间复杂度**为：

$$\text{排序}: O(mlogn) + \text{构造}: O(m)$$

前面Prim算法主要考虑的是点，以点来更新点，进行堆优化又比较复杂。因此，**在面对稀疏图时，优先使用Kruskal算法。**

代码

```
// 定义边结构体
typedef struct edge{
    int a;
    int b;
    int val;
}edge;
edge E[200010];
int n,m;
int p[200010];

int find(int x){
    if(p[x]!=x) p[x] = find(p[x]);
    return p[x];
}

void Kruskal(){
    int i,res=0,cnt=0;
    // 将边进行排序,可以手动实现
    sort(E,E+m,cmp);
    // 并查集初始化
    for(i=1;i<=n;i++) p[i]=i;
    for(i=0;i<m;i++){
        // 若左右不连通
        int a = E[i].a,b=E[i].b,w=E[i].val;
        a = find(a),b=find(b);
        if(a!=b){
            p[a] = b;
            res+=w;
            cnt++;
```

```
            }
        }
    if(cnt!=n-1) puts("impossible");
    else printf("%d\n",res);
}
```

例题

859.Kruskal算法求最小生成树

代码

这里我采用了**纯c语言**实现，如果可以使用C++，则调用sort即可，就不必手写快排了，时间花销是差不多的。

```
#include<stdio.h>
#include<string.h>
int p[200010],n,m;
typedef struct Edge{
    int a,b,val;
}Edge;

Edge edges[200010];

// 快速排序
void q_sort(int l,int r){
    int ll = l,rr = r;
    Edge t = edges[l];
    //printf("%d-%d-%d\n",t.a,t.b,t.val);
    while(ll<rr){
        while(ll<rr&&edges[rr].val>=t.val) rr--;
        edges[ll] = edges[rr];
        while(ll<rr&&edges[ll].val<=t.val) ll++;
        edges[rr] = edges[ll];
    }
    edges[ll] = t;
    if(ll>l) q_sort(l,ll-1);
    if(ll<r) q_sort(ll+1,r);
}

// 并查集find操作
int find(int x){
    if(p[x]!=x) p[x] = find(p[x]);
    return p[x];
}

void Kruskal(){
    int i,j,k,a,b,v,res=0,cnt=0;
    // 先将所有边按其权值从小到大进行排序
    q_sort(0,m-1);
    //for(i=0;i<m;i++) printf("%d ",edges[i].val);
    for(i=0;i<m;i++){
        a = edges[i].a;
        b = edges[i].b;
        v = edges[i].val;
        a = find(a),b = find(b);
        if(a!=b){
            p[a] = p[b];
            res += v;
```

```
            cnt++;
        }
    }
    // 最小生成树的边数为n-1
    if(cnt<n-1) puts("impossible");
    else printf("%d\n",res);
}

int main(){
    int u,v,w,i;
    scanf("%d%d",&n,&m);
    for(i=1;i<=n;i++) p[i] = i;
    for(i=0;i<m;i++){
        scanf("%d%d%d",&u,&v,&w);
        edges[i].a = u,edges[i].b = v,edges[i].val = w;
    }
    Kruskal();
    return 0;
}
```

堆

基本理解

堆是一种特别的数据结构，它在形状上满足**完全二叉树**的定义。即它的存储是可以按照完全二叉树那样，存在一个数组里的。

> **完全二叉树存储**
>
> 节点i(下标为i) 的左儿子下标是**2i**，右儿子下标是**2i+1**；
>
> 同理，节点i的父节点则是**i/2**(向下取整)；

堆的种类

> - **大根堆**：每个节点都要**大于等于**它的左右子树的所有值。
> - **小根堆**：每个节点都要**小于等于**它的左右子树的所有值。

那么很明显，如果我们将一个序列构建成大根堆(小根堆)，那么这个堆的顶点就一定是序列的最大值(最小值)。

堆的操作

y总笔记

如何手写一个堆?

1. 插入一个数	heap[++ size] = x; up(size);
2. 求集合当中的最小值	heap[1];
3. 删除最小值	heap[1] = heap[size]; szie -- ; down(1);
4. 删除任意一个元素	heap[k] = heap[size]; size -- ; down(k); up(k);
5. 修改任意一个元素	heap[k] = x; down(k); up(k);

注：其实down操作和up操作都只会进行其中一个，因为**二者条件互斥**。

堆排序

利用堆的特点，如果能**动态维护**一个堆结构，那么每次取出**堆顶元素**加入序列，最终得到的序列就一定是有序的。

堆排序实现

下沉(down)操作

所谓down操作，就是指当堆的结构**不满足定义**时，如大顶堆的根节点比它左右儿子小，此时这个节点应该**下沉到一个正确的位置**从而保证堆结构成立。

down操作用来调整堆结构，使其满足堆结构定义

> **思路**
>
> 拿小顶堆来说，当当前节点无法满足堆定义时，需要进行down操作。
>
> 1. 取出当前点与其左右儿子节点中值**最小**的点。
> 2. 如果说这个最小值点不是当前的根节点，说明根节点位置有问题，则将根节点与这个最小值点互换位置。
> 3. 接下来对当前根节点进行**递归**处理。(因为交换以后可能被交换的点的位置也不合理)

代码

```
void down(int u){
    int k = u;
    // 若有左儿子,且左儿子不合理
    if(2*u<=n&&heap[2*u]<heap[k]) k = 2*u;
    // 若有右儿子,且右儿子不合理
    if(2*u+1<=n&&heap[2*u+1]<heap[k]) k = 2*u+1;
    if(k!=u){
        reap[0] = reap[k];
        reap[k] = reap[u];
        reap[u] = reap[0];
        down(k);
    }
}
```

上浮up操作

即将一个节点向上浮动，当当前节点不满足定义时，如小顶堆中当前节点比父节点还小，那当前节点的位置应该和父节点进行调换，也就是所谓的“**上浮**”。

up操作一般用于向堆中插入元素

```
void up(int x){
    while(x/2>=1&&heap[x]<heap[x/2]){
        // 交换父子节点
        heap[0] = heap[x];
        heap[x] = heap[x/2];
        heap[x/2] = heap[0];
        x /= 2;
    }
}
```

删除操作

删除最小值点

```
heap[1] = heap[N--];
down(1);
```

例题

Acwing 838.堆排序

注：堆排序只需要down操作即可完成。

代码

```
#include<stdio.h>
int n,m;
int heap[100010];
int cnt;
// 下降函数
void down(int cur){
    // 先把当前点记录为最小值点
    int t = cur;
    // 有左孩子且左孩子比最小值点小
    if(cur*2<=cnt&&heap[cur*2]<heap[t]){
        t = 2*cur;
    }
    // 有右孩子且右孩子比最小值点小
    if((cur*2+1)<=cnt&&heap[cur*2+1]<heap[t]){
        t = 2*cur+1;
    }
    // 最小的那个点不是根节点，说明根节点有问题，需要换位置
    if(t!=cur){
        heap[0] = heap[t];
        heap[t] = heap[cur];
        heap[cur] = heap[0];
        // 递归使该节点下沉
```

```c
            down(t);
        }
}

int main(){
    int i;
    scanf("%d %d",&n,&m);
    for(i=1;i<=n;i++) scanf("%d",&heap[i]);
    cnt = n;
    // 建立初始堆,从n/2开始是因为,n/2是这棵树最后一个非叶子节点
    for(i=n/2;i>=1;i--) down(i);
    while(m--){
        // 头节点为当前区域最小值
        printf("%d ",heap[1]);
        // 删除头节点，做法就是把当前区域最后一个值拿过来直接覆盖掉根节点
        heap[1] = heap[cnt--];
        // 重新down操作，保证堆结构合理
        down(1);
    }
    printf("\n");
    return 0;
}

```

注意

今天在重刷代码时提交Wrong了，我发现我自己没有注意一个细节，在排序的过程中，我们的思路是每次输出堆顶元素，再将其删掉，从而得到递增序列。但我今天的做法是错误的，**记住，一定要真的把这个值删掉，也就是要把堆节点个数n真的减少**。我今天的错误就出现在我用一个临时变量存了堆的元素个数，结果一直在减少这个临时变量，没有修改n值，实际上就没有真的把元素删掉。

散列(哈希)表

使用哈希表来存放数据，也就是所谓的**散列存储**。意思是根据插入的关键字的值，通过某种映射方式得到这个值应该插入的位置。查找时也一样，通过同样的规则计算出查找关键字存放的位置，再按照一定规则从对应位置处开始查找。

比较常用的映射方式是**使用关键字对存储空间大小进行取模操作**，为了保证获得的下标为正，在c/c++中可以通过下面的方式来实现。

```c
// key为插入或查找的关键字，n是哈希表存储空间大小
c = (key%n+n)%n;
```

往往关键字的范围会远远大于散列表的空间大小，所以难免会出现**冲突**(即两个不同的关键字经过映射函数，得到的存放地址相同)，也就出现了一些处理冲突的方法。

拉链法

拉链法，顾名思义，当出现冲突时，所有冲突的元素都会被存放在一个链表里。类似于邻接表，先确定下标位置，数组中每一个位置都对应着一条链表，所有冲突元素都会被挂在对应数组位置下的链表里。

相关代码实现

```
int MAX = 100010;
int h[100010],val[100010],ne[100010],idx=0;

// 拉链法处理哈希冲突,头插法
void insert_link(int x){
    // 直接取余得到存放位置
    int i = (x%MAX+MAX)%MAX;
    val[idx] = x;
    ne[idx] = h[i];
    h[i] = idx++;
}

// 拉链法查找是否存在元素
int find_link(int x){
    // 直接取余得到存放位置
    int i = (x%MAX+MAX)%MAX;
    // 取得拉链头节点
    for(i = h[i];i!=-1;i=ne[i]){
        if(val[i]==x){
            return 1;
        }
    }
    return 0;
}
```

开放寻址法

基本思想就是，当通过映射得到的位置并不是空闲的时候，就继续向临近的位置探索，直到找到一个可以放的位置。

代码实现

```
// 一般会选择开两倍的空间
int MAX = 0x3f3f3f3f,N = 200020;
int h[200020];
// 开放定址法,如果x存在则返回其存放的位置，否则返回其应该放的位置
int find(int x){
    // 获得映射地址
    int i=(x%N+N)%N;
    // 当这个地址对应的空间不是空闲的，并且也不是要查找的数的时候，就继续向后找
    while(h[i]!=MAX&&h[i]!=x){
        i++;
        // 找到末尾了，此时应该循环从头开始再找
        if(i==N) i=0;
    }
    return i;
}
```

> **注**：**MAX**值是一个**标志值**，这里的作用是用来标注空间是**空闲**的。因为有些题目会规定每个元素的大小，这个**MAX**值一般是比所有规定值都大的，可以用来做标志位。

字符串哈希

字符串哈希，顾名思义，就是要**通过某种方法将一个字符串映射成一个哈希值**，哈希值直接代表这个字符串与其他字符串进行比较。

比较经典的问题就是，让你比较同一字符串的两个子串是否相同。如果不考虑时间，那么substr即可完成字串截取，但只要是上机考试，就存在时间限制，那么这个函数以O(n^2)的时间复杂度很容易就超时。

思路&流程

每个字符串可能含有数字、大小写字母，我们可以将它抽象成一个**p进制数**，从左到右即为从高位到低位。那么字符串的哈希值就是这个p进制数转换成10进制数后的值。

通过一个**哈希数组**来记录这个字符串的**前缀哈希值**，即**前i个字符构成的字串的哈希值**，这样可以通过类似于**前缀和**的方式来获取任意区间字符串的哈希值。

那么，这个p如何定义呢?

将字符串抽象成p进制数后，里面既有大小写字母，又有数字，甚至可能有字符。那么，我们用的最多的便是**ASCII码**了，基于经验，可以将**p值设定成131**，大部分的数字、字母、字符都能够包含在内。

此外，通过**ASCII码**构成一个**131进制数**，再转换成十进制将十分的庞大。我们需要将这个数进行mod，保证它能够在某种数据类型的范围内正常工作，基于经验，一般**mod 2^64**。在C/C++中，**unsigned long long**范围恰好就是**0~2^64 - 1**，一旦超出范围就会自动取模，因此只需要用这个数据类型的数组来存哈希值即可。

大致流程如下:

PS: 字符串下标从1开始，有效数组元素也从1开始。

> 1. 准备一个 **h 数组**和 **p 数组**。
>
>   **h[i]:** 字符串**前i个字符**构成的字符串的哈希值。
>
>   **p[i]:** 表示p的i次方。
>
> 2. **预处理** p 数组 和 h数组，遵循下面的公式进行推导：**(P=131)**
>
> $$p[i] = p[i-1] * P, h[i] = h[i-1] * P + input[i]$$
>
> 3. 如果要求出某一个字串的哈希值，如字串下标范围是[a,b]，则按下面公式即可定义求指定字串的哈希值函数。
>
> $$getHash(a, b) = h[b] - h[a-1] * p[b-(a-1)]$$
>
> 这里乘上一个$p^{b-(a-1)}$是为了让短的字串进行移位，使二者高位对齐。

应用

[**Acwing 831. KMP**]

此题为KMP算法的经典题，需要输出指定模式串在主串的所有开头下标。

由于字符串哈希解决的就是字符串匹配的问题，当然也可以用来解决KMP算法能够解决的问题。

字符串哈希是通过**将字符串转换成指定的P进制数**，这个P进制数转换成10进制得到的值就是Hash值，从而**用哈希值来代替字符串进行比较**。那么，我们只要按照同样的方法预计算出模式串的哈希值，并预处理得到主串的哈希值数组，在主串中枚举所有长度和模式串一样的子串，比较其与模式串的哈希值是否相等，即可知道是否匹配。

代码

```
#include<string.h>
#include<stdio.h>
typedef unsigned long long ull;
int n,m;
char P[1000010],S[10000010];
ull h[1000010],p[1000010];
int X=131;
// 获取[a,b]子串的哈希值
ull getHash(int a,int b){
    return h[b]-h[a-1]*p[b-(a-1)];
}

int main(){
    int i;
    ull hashP;
    scanf("%d\n",&n);
    scanf("%s",P+1);
    scanf("%d\n",&m);
    scanf("%s",S+1);
    // 计算模式串的哈希值
    hashP = 0;
    p[0]=1;
    for(i=1;i<=n;i++){
        hashP = hashP*X + P[i];
    }
    // 计算总串的哈希值
    for(i=1;i<=m;i++){
        p[i] = p[i-1]*X;
        h[i] = h[i-1]*X + S[i];
    }
    // 在总串中找到哈希值与模式串相同的子串，输出其开始下标
    for(i=1;i+n-1<=m;i++){
        if(hashP==getHash(i,i+n-1)){
            printf("%d ",i-1);
        }
    }
    return 0;
}
```

Acwing841.字符串哈希

```
#include<stdio.h>
#include<string.h>
typedef unsigned long long ull;
char input[100010];
ull h[100010],p[100010];
int P=131;

// 得到[l,r]子串的hash值
```

```
ull getHash(int l,int r){
    return h[r]-h[l-1]*p[r-(l-1)];
}

int main(){
    int m,n,l1,r1,l2,r2,i,j;
    scanf("%d%d",&n,&m);
    // 有效下标从1开始
    scanf("%s",input+1);
    // 预处理两个数组
    p[0]=1;
    for(i=1;i<=n;i++){
        p[i] = p[i-1]*P;
        h[i] = h[i-1]*P+input[i];
    }
    while(m--){
        scanf("%d%d%d%d",&l1,&r1,&l2,&r2);
        if(getHash(l1,r1)==getHash(l2,r2)) puts("Yes");
        else puts("No");
    }
    return 0;
}
```

贪心思想

所谓贪心，是指一种**短见**的思考方式，即对于每一个状态，我都选择**当前状态下最好的路**往下走，而不考虑全局。

这是一种灵活的思想，不存在硬性的模板，因此尽量依靠题目来积累。

区间问题

Acwing 906. 区间分组

题目

给出N个区间，让我们将这些区间进行分组。分组的要求是：**组内**的区间两两**互不相交**，且要求**分组数尽可能少。**

输入样例

```
3
-1 1
2 4
3 5
```

输出样例

```
2
```

Ps: 这里可以将 [-1,1] 和 [2,4]分为一组，[3,5]自己为一组。

思路

想让组数尽可能小，那么我们就得追求每个分组尽可能大。当向某个分组中添加区间时，要尽可能把**所有**能够满足互不相交条件的区间都加入其中。

可以通过维护一个**堆**来实现，堆中存放某个分组的最大右端点，其中这个右端点是堆中所有分组里最小的。

> 1. 先将所有区间按照左端点从小到大排好
> 2. 遍历每个区间
> 3. 若当前区间左端点比堆顶存放的分组的最大右端点要**小**，说明当前区间一定会与堆顶的分组**有交叉**的部分。说明二者**不能在同一个分组**内，因此将当前区间的右端点直接插入堆中，以此来代表一个新的分组。
> 4. 若当前区间左端点比堆顶存放的分组的最大右端点要**大**，说明当前区间**不会**与堆顶分组**交叉**，那么为了追求分组数尽可能少，应将当前区间加入堆顶分组。做法是先将堆顶弹出，再将当前区间右端点加入堆中。
> 5. 最后，循环结束。堆中存放了多少个右端点，就代表有多少个分组。

代码

```
#include<stdio.h>
typedef struct range{
    int begin,end;
}range;
int n;
range all[100010];
// 用一个堆来维护各组内区间的最大右端点
int heap[100010],cnt=0;

void quick_sort(int l,int r){
    range t = all[l];
    int ll = l,rr = r;
    while(ll<rr){
        while(ll<rr&&all[rr].begin>=t.begin) rr--;
        all[ll] = all[rr];
        while(ll<rr&&all[ll].begin<=t.begin) ll++;
        all[rr] = all[ll];
    }
    all[ll] = t;
    if(l<ll) quick_sort(l,ll-1);
    if(r>ll) quick_sort(ll+1,r);
}

void down(int x){
    int t = x;
    if(x*2<=cnt&&heap[x*2]<heap[t]) t = x*2;
    if(x*2+1<=cnt&&heap[x*2+1]<heap[t]) t = x*2+1;
    if(t!=x){
        heap[0] = heap[t];
        heap[t] = heap[x];
        heap[x] = heap[0];
        down(t);
    }
}

void up(int x){
    while(x/2>=1&&heap[x]<heap[x/2]){
        heap[0] = heap[x];
        heap[x] = heap[x/2];
        heap[x/2] = heap[0];
        x/=2;
```

```
    }
}

void push(int x){
    heap[++cnt] = x;
    up(cnt);
}

// 弹出堆顶区间右端点下标
int pop(){
    int res = heap[1];
    heap[1] = heap[cnt--];
    down(1);
    return res;
}

// 要想让分组数最小,就应该让每个组内元素尽可能多
int main(){
    int i,j,a,b;
    scanf("%d",&n);
    for(i=0;i<n;i++){
        scanf("%d%d",&a,&b);
        all[i].begin = a;
        all[i].end = b;
    }
    quick_sort(0,n-1);
    for(i=0;i<n;i++){
        // 如果当前区间左端点比最小右端点的组的右端点还小
        // 则这个区间必须开一个新组,并加入堆中
        if(cnt==0||(cnt>=1&&all[i].begin<=heap[1])){
            push(all[i].end);
        }else{
            // 此时二者可以是一组
            pop();
            push(all[i].end);
        }
    }
    printf("%d\n",cnt);
    return 0;
}
```