# Acwing算法基础课：贪心算法



## 1.区间问题

# 1.1区间选点

## 1.1.1题目描述

给定 $N$个闭区间$[a_i,b_i]$ [][��,��]，请你在数轴上选择尽量少的点，使得每个区间内至少包含一个选出的点。

输出选择的点的最小数量。

位于区间端点上的点也算作区间内。

**输入格式**

第一行包含整数 $N$，表示区间数。

接下来 $N$ 行，每行包含两个整数 $a_i,b_i$，表示一个区间的两个端点。

**输出格式**

输出一个整数，表示所需的点的最小数量。

**数据范围**

$1 \le N \le 10^5$
$-10^9 \le a_i \le b_i \le 10^9$

**输入样例**

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

**输出样例**

```
2
```

## 1.1.2题目分析

首先，要明白题目的意思：

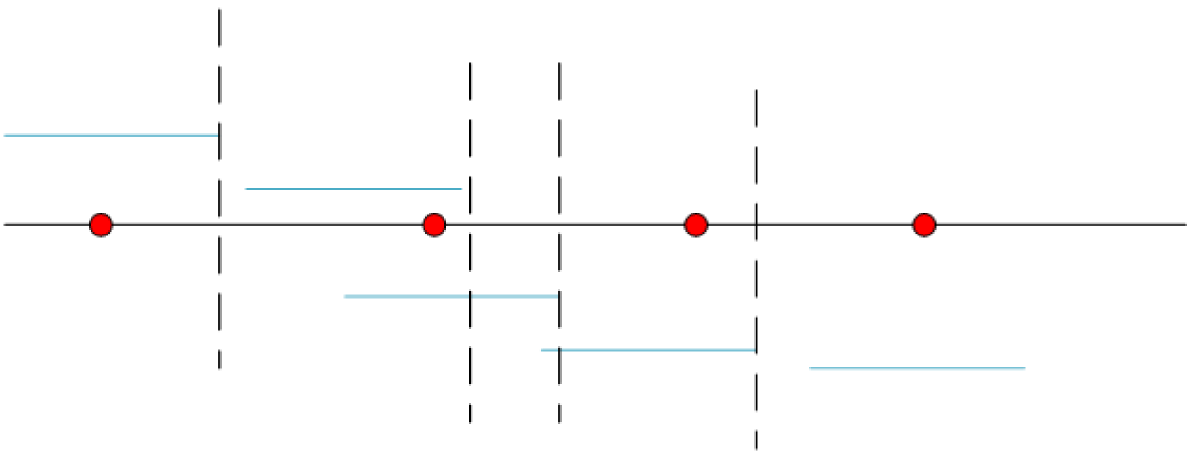

图1-1

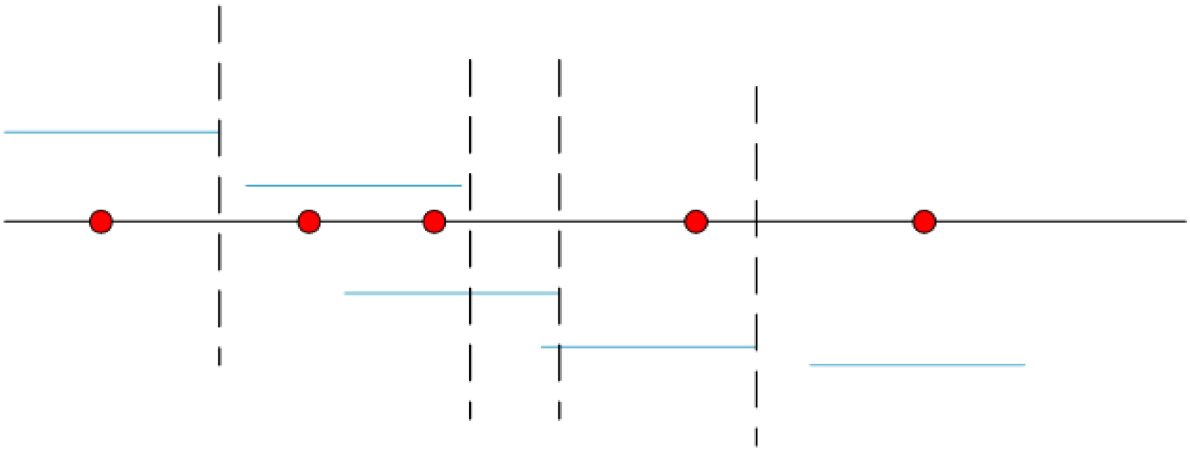

图1-2

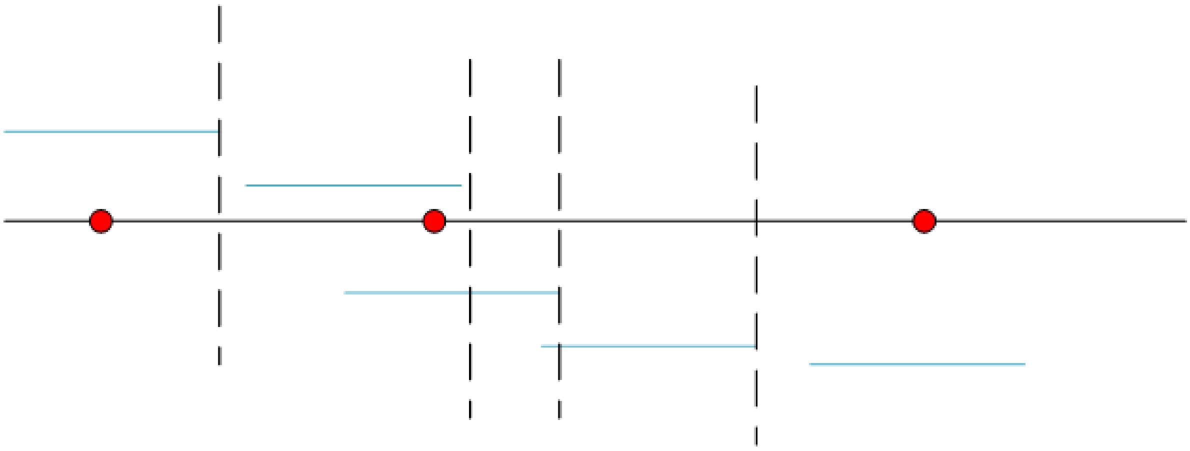

图1-3

黑线表示数轴，蓝线表示区间。对于图1-1和图1-2，都是符合题意的点：对于每个区间，总是有落在其上的点。对于图1-3，第4个区间没有点落在其上。但是图1选择的点更少，那么题目的意思就是，给定一组区间，找到最少数量的一组点，使得每个区间都有点落在其上。

### 1.1.3算法描述

对于贪心算法来说，其实没有固定的格式..只能说大概思想为，每次选择当前(局部)的最优解，最后证明，在经过一系列的贪心选择后，得到的就是全局最优解。

根据闫老师的解释是，那区间问题就先排序(真的嘛hh~)，并且很自然给出了下列的算法：

1.将区间按照右端点的大小进行排序

2.检索每个区间的右端点，那么会有以下的两种情况：

(1)选择的上一个点**没有**落在该区间，那么将该区间的右侧端点加入选择的点集***S***中。(第一个区间的右端点直接加入)

(2)选择的上一个点落在该区间，那么检索下一个区间。

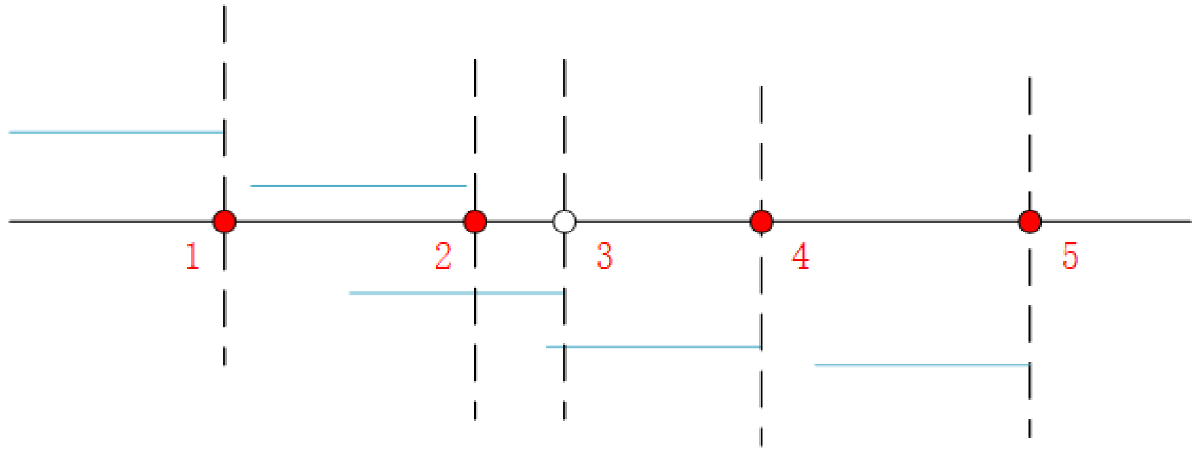


图1-4

这样，如图1-4所示，我们用$x_i$表示第$i$个点的坐标，第$i$个区间表示为[$a_i$ ,$b_i$ ]。按照算法，会经过下列的步骤：

1.检索第一个区间的右端点***1***，作为第一个区间，将其加入点集***S***中。事实上，在代码实现的过程中，肯定会维护一个变量，记录上一个选择的点的坐标***last***，将其初始为较小的负数就好。$last$=$x_1$ $S$={***1***}。

2.检索第二个区间的右端点***2***，由于$last < a_2$ ,也就是上一个选择的点没有落入该区间中，将***2***加入点集***S***：$last$=$x_2$ $S$={***1***,***2***}。

3.检索第二个区间的右端点***3***，由于$last$>$a_3$ ,即上一个选择的点在该区间中，跳过。

4.检索第三个区间的右端点***4***，***5***，情况和2相同，最终***S***={***1***，***2***，***4***，***5***}。

**证明**

下来证明局部最优解最终可以获得全局最优解。

这里的证明是比较简单的，对于每个区间，我们有将其右端点加入或不加入点集***S***两种选择。


图1-5

如图1-5所示，若此时检索的区间选择了右端点，**那么两个区间一定是不重合的。**因而，选择***n***个点后，就会对应***n***个没有交集的***n***个区间。很显然，至少需要***n***个点，才可以保证每个区间都有点与之对应。

## 1.1.4代码实现

代码也很简单：

1.用***pair<int,int>***表示区间。

2.对区间进行排序，我们用***sort***函数，因而要自己写一个***cmp***函数。事实上，这个在算法题中会经常遇到，函数内只有***return***语句。

```
bool cmp(II a,II b)
{
    return a.second<b.second;
}
```

这里是升序，降序只需要将<改为>即可。

3.用***last***维护上一个选择点的坐标，因为需要和当前区间的左侧端点的值进行比较。

#### 完整代码

```
#include<iostream>
#include<algorithm>
using namespace std;
#define II pair<int,int>
int N;
II section[100010];
//按照右端点进行排序，重写sort函数
bool cmp(II a,II b)
{
    return a.second<b.second;
}
int main(){

    //input
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d %d",&section[i].first,&section[i].second);
    }

    //sort
    sort(section,section+N,cmp);

    int res = 0,last=-1e10;//last,上一个点
    for(int i=0;i<N;i++){
        //cout<<section[i].first<<" "<<section[i].second<<endl;
        if(section[i].first>last){//需要将点加入
            res++;
            last=section[i].second;
        }
    }
    //output res
    cout<<res;
}
```

---

## 1.2最大不相交区间数量

### 1.2.1题目描述

给定 $N$ 个闭区间 $[a_i,b_i]$ ，请你在数轴上选择若干区间，使得选中的区间之间互不相交（包括端点）。

输出可选取区间的最大数量。

#### 输入格式

第一行包含整数 $N$ ，表示区间数。

接下来 $N$ 行，每行包含两个整数 $[a_i,b_i]$，表示一个区间的两个端点。

**输出格式**

输出一个整数，表示可选取区间的最大数量。

**数据范围**

$1 \le N \le 10^5$
$-10^9 \le a_i \le b_i \le 10^9$

**输入样例**

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

**输出样例**

```
2
```

## 1.2.2题目分析

写到这里的时候，其实还没有看到闫老师的分析哦。但是感觉上，很容易想到还是将区间按照右端点的大小进行排序，然后检索每一个区间和上一个区间是否有重合：若有重合，就不选这个区间了；若不重合，将该区间加入最后的选择结果中。

那我们需要证明这样的算法是正确的。我认为一种可能的说明方式是：

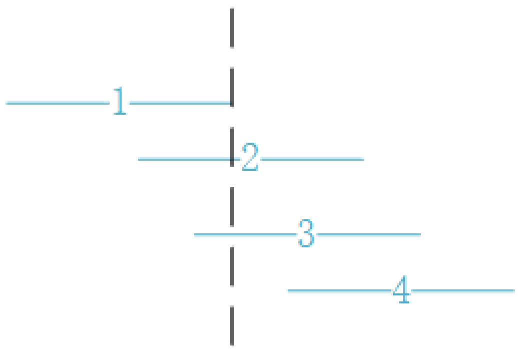


图2-1

如图2-1所示，区间已经按照右端点大小进行排序了，对于最终选择的区间：区间1、2、3至多只可以选择一个，因为3个区间有重合的部分；而区间1、2、3必定会选择一个，因为假设都不选，那无论区间4是否选择，区间1都是可以加入的，因为它和4没有重合部分。那这里我们也可以看到，假设最终结果中包含2或3，那么我一定可以用区间1进行替换，也就是说，**最优解中一定包括区间1！** 同样的，区间4到区间$n$($n$≥4)，是会和区间4有重叠部分，那么最优解中，一定会包括区间4，而不包含其他区间。所以按照该种方法一定可以找到最优解。(也就是我们可以把区间划分为几个部分来考虑)

## 1.2.3代码实现

基本上和上一题的代码一致，甚至更简单些。

```
#include<iostream>
#include<algorithm>
using namespace std;
#define II pair<int,int>
int N;
```

```
II sec[100010];
bool cmp(II a,II b){
    return a.second<b.second;
}
int main(){
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d %d",&sec[i].first,&sec[i].second);
    }
    sort(sec,sec+N,cmp);
    int res=0,last=-1e10;
    for(int i=0;i<N;i++){
        if(sec[i].first>last){
            res++;
            last=sec[i].second;
        }
    }
    cout<<res;
}
```

---

## 1.3区间分组

### 1.3.1题目描述

给定 $N$个闭区间 $[a_i,b_i]$，请你将这些区间分成若干组，使得每组内部的区间两两之间（包括端点）没有交集，并使得组数尽可能小。

输出最小组数。

**输入格式**

第一行包含整数 $N$，表示区间数。

接下来 $N$ 行，每行包含两个整数 $[a_i,b_i]$，表示一个区间的两个端点。

**输出格式**

输出一个整数，表示最小组数。

**数据范围**

$1 \le N \le 10^5$

$-10^9 \le a_i \le b_i \le 10^9$

**输入样例**

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

**输出样例**

```
2
```

### 1.3.2题目分析

同样的，我们先不看闫老师的讲解，自己先尝试解答一下。我们仍然拿第二题的图进行分析：

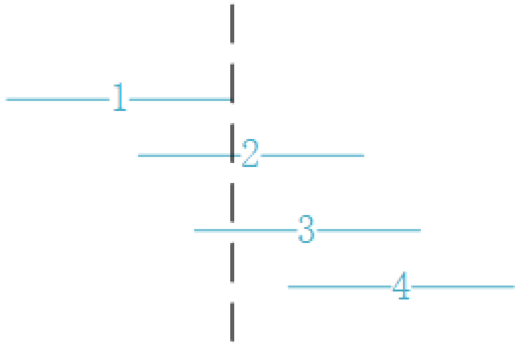


图3-1

需要清楚的是，每一个区间必须在一个组中。所以很显然，区间1、2、3可以将其分别归入组1、2、3。而4可以加入组1。区别于第2题，可以很容易想到对应的算法：

用***last***来记录当前存在的组，最新加入的区间中最小的右端点坐标，并更新***last***：

检索每一个区间，若$a_i > last$,说明至少可以加入到***last***对应的那个组中。若$a_i < last$，说明和当前的组中的区间都会有交集，那么需要创建一个新的组，加入该区间，由于我们会将区间按照右端点的大小进行排序，所以并不会影响***last***，但是我们要做好记录。

对于图3-1，分析过程为：

1.区间1加入组***1***，$last=b_1$

2.$a_2 < last$,区间2加入组***2***，$last=b_1$

3.$a_3 < last$,区间3加入组***3***，$last=b_1$

4.$a_4 > last$,区间4加入组***1***，$last=b_2$

**但是在提交过后是WA！！**

那为什么会这样？事实上，按照闫老师的讲解，将区间按照左端点的大小进行排序，就是正确的算法了。而两者的区别，在自己尝试之后好像没有找到一个合适的例子，但是Acwing上一个同学给了解释：[AcWing 906. 一张图弄明白为什么不能右端点排序]

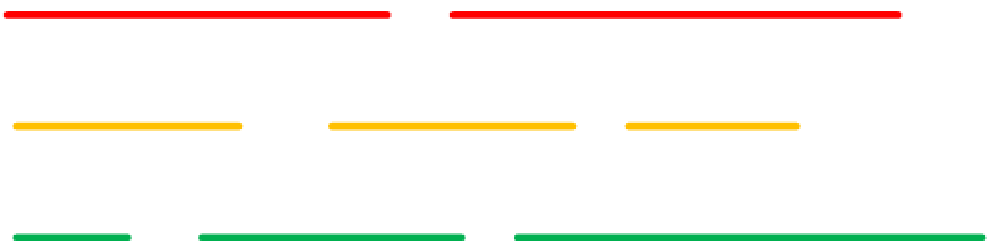

图3-2

**若按照右端点排序：**

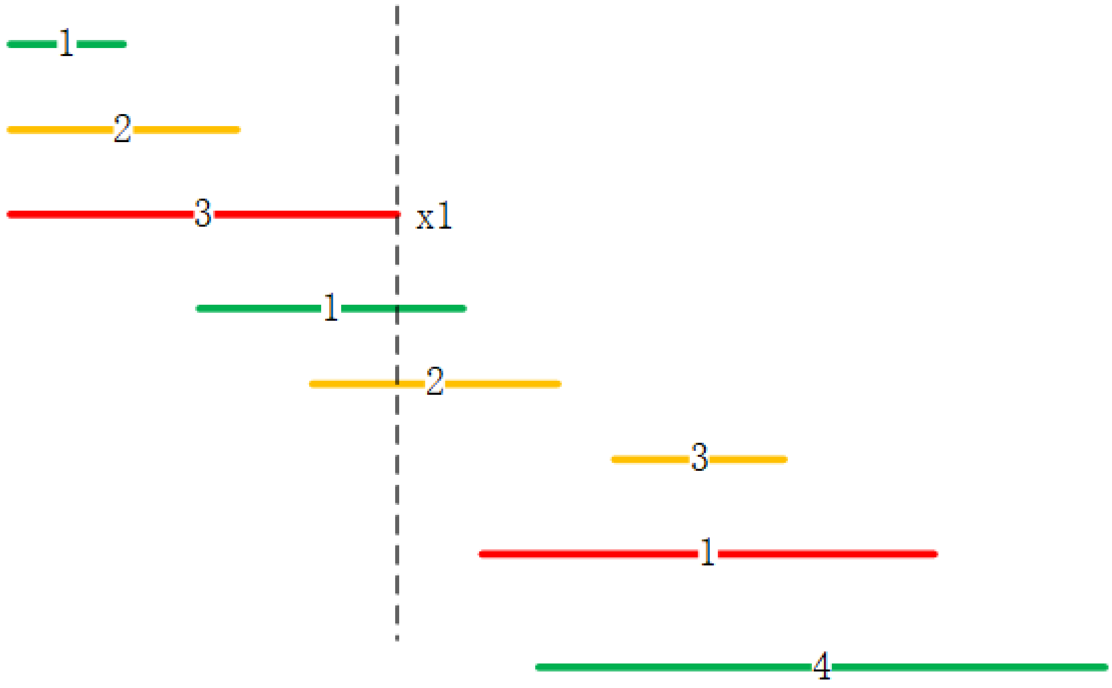


图3-3

那分组情况已经在图中进行了说明，在检索最后一个绿色的区间时，其左端点的值比分组1、2、3中最新插入的区间的右端点的值都大，按照算法，我们必须将其分作一组。但事实上，图3-2只用了3组就完成了划分。

**那这样的原因又是什么呢？**我们看第3个黄色的区间，事实上，该区间可以加入分组1、2、3，但很显然加入分组2是最好的选择：

因为后续的区间只要左端点的值大于$x_1$，就可以不必创建一个新的分组。可是按照算法，第3个黄色的区间加入了分组3，使得第3个红色的区间被迫加入分组1，第三个绿色的区间需要划分一个新的分组。

我们可以得出结论，加入***last***所在的分组并不是最好的一个选择，而是需要加入所有**可插入的**分组中，新插入的区间右端点最大的那个分组。**但是换一种想法**，我们担心的是，将区间***sec***插入了***last***所在的分组，而后续还有分组***sec1***的左端点在***sec***和***last***之间，这样发现***sec***会是更适合插入***last***所在的分组的区间。因而，我们将区间按照左端点进行排序，这样不就解决了问题吗！？

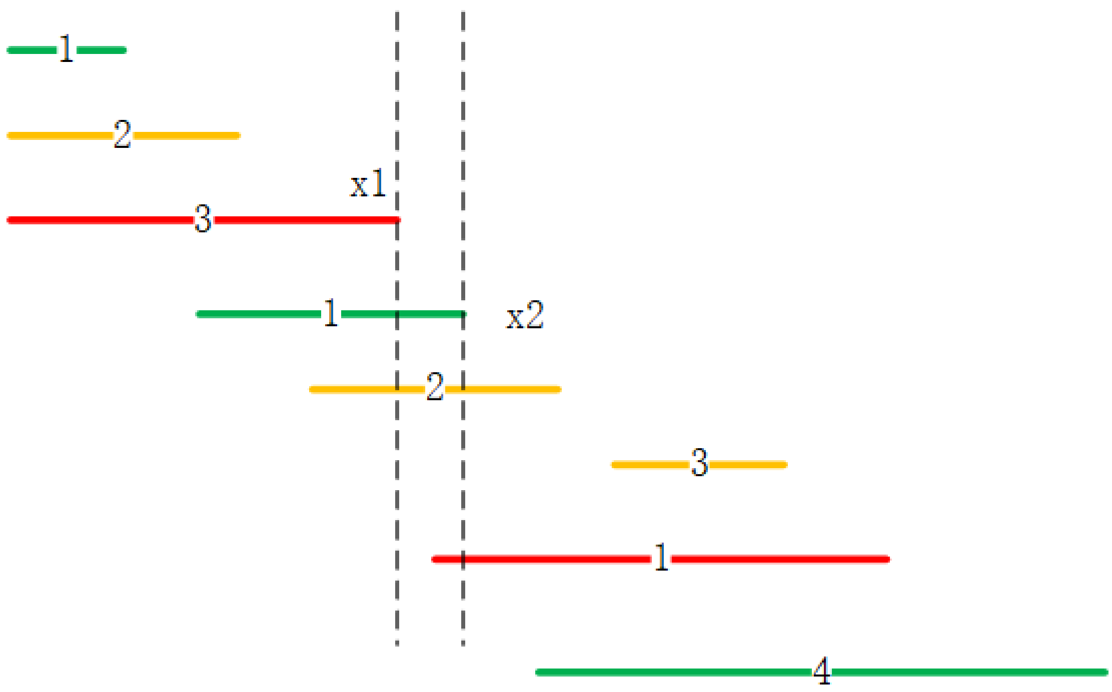


图3-4

（如图3-4,第2个红色区间会比第三个黄色区间更适合加入分组3）

### 1.3.3代码实现

**按照左端点进行排序：**

```
#include<iostream>
#include<algorithm>
#include<queue>
using namespace std;
#define II pair<int,int>
int N;
II sec[100010];
priority_queue<int,vector<int>,greater<int> > heap;//小顶堆
bool cmp(II a,II b){
    return a.first<b.first;
}
int main(){
    int N;
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d %d",&sec[i].first,&sec[i].second);
    }
    sort(sec,sec+N,cmp);//sort
    int res=1;
    //将一个很小的数插入小顶堆
    heap.push(-1e10);
    for(int i=0;i<N;i++){
        if(sec[i].first>heap.top()){//可以加入该组,heap.top()就是last
            heap.pop();
            //既然加入了该组，右端点的值一定是更大的，要更新last
            heap.push(sec[i].second);
        }
        else{
            //需要创建一个新的组
            res++;
            //并且将该区间的右端点的值加入到小顶堆中
            heap.push(sec[i].second);
        }
    }
    //数据结构为什重要...
    cout<<res;
}
```

这里需要注意的是，用到了优先队列，也就是小顶堆。因为我们需要维护每个分组右端点的最小值，而每次在更新的时候，刚好是把堆顶删除，插入新的值，而新的堆顶，我们并不需要管，优先队列会给我们排好，我们只要用***pop()***返回就好了。

事实上，前面也提到了，按照右端点进行排序，也是可以的，但是此时就不仅要判断是否需要创建新的分组，还需要插入右端点离我最近的分组，时间复杂度就很大了。

# 1.4区间覆盖

## 1.4.1题目描述

给定 $N$个闭区间 $[a_i,b_i]$以及一个线段区间 $[s,t]$，请你选择尽量少的区间，将指定线段区间完全覆盖。

输出最少区间数，如果无法完全覆盖则输出 $-1$。

**输入格式**

第一行包含两个整数 $s$ 和 $t$，表示给定线段区间的两个端点。

第二行包含整数 $N$，表示给定区间数。

接下来 $N$ 行，每行包含两个整数$a_i$，$b_i$表示一个区间的两个端点。

**输出格式**

输出一个整数，表示所需最少区间数。

如果无解，则输出 $-1$。

**数据范围**

$1 \le N \le 10^5$
$-10^9 \le a_i \le b_i \le 10^9$

$-10^9 \le s \le t \le 10^9$

**输入样例**

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

**输出样例**

```
2
```

## 1.4.2题目分析

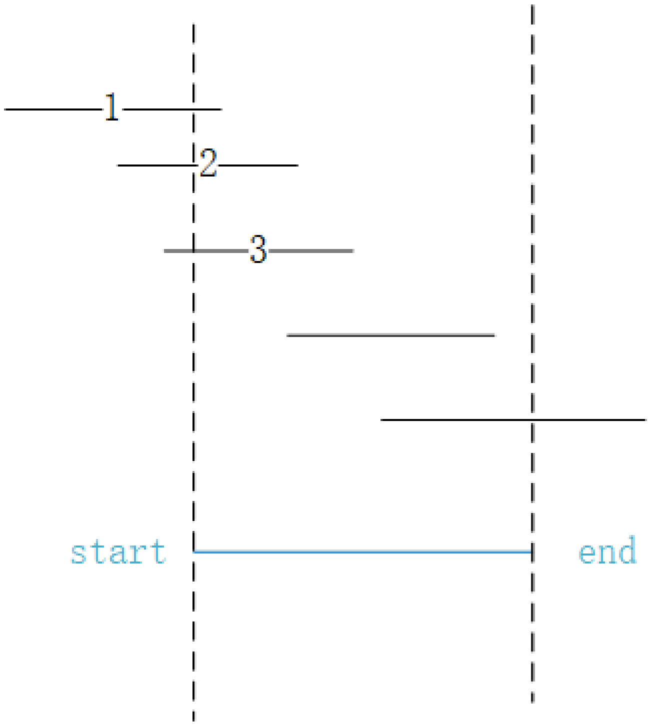


图4-1

这题的分析，还是看了闫老师的讲解才想到的。同样的，我们还是对区间的左端点进行排序。对于线段区间[$start$,$end$]，要想覆盖点$start$，**那么区间1、2、3必须要选择一个**。而在这里，我们的**贪心选择**就是选择右端点值最大的那个区间[$a_i$,$b_i$]。并将$start$更新为$b_i$，事实上这是显然的，因为选择的已经覆盖了[$start$,$b_i$]，我们会得到一个子问题，覆盖区间[$b_i$,$end$]。

那么我们需要解决两个问题：

**1.为什么这样可以得到全局最优解?**

事实上，在前面已经大概分析过了。我们的第一次选择，假设有$n$个区间是覆盖了$start$的，而第$i$个区间的右端点最大。那么对于全局最优解，在这$n$个区间中至少会选择一个。若区间$i$[$a_i$,$b_i$]未被选中，选择了区间$j$[$a_j$,$b_j$]，由于$b_j < b_i$,那么讲区间$i$代替区间$j$，也同样可以覆盖[$s$,$t$]，并且区间数并没有增多，也同样是最优解。事实上，若在这$n$个区间选择了多个区间而不包含区间$i$，区间$i$也同样可以替代，但这与其本身是全局最优解是矛盾的。

因而，我们可以得出结论，对于全局最优解，在$n$个覆盖了$start$的区间中，一定只唯一的选择了右端点最大的那个区间。

这样，我们将原本的问题转化了一个子问题：在剩下的区间中，选择若干个区间，覆盖[$b_i$,$end$]，我们可以递归的证明，每一步选择的区间，一定是在全局最优解当中的。那么最后我们只需要判断$b_i \geq end$，算法就结束了。

当然，在算法结束时，若仍然有$start \leq end$，我们就认为不存在这样的区间覆盖。

**2.为什么要按照左端点进行排序?**

在这里应该还是比较好理解的。我们在进行每一步选择时，需要将覆盖了当前$start$的区间选择出来，而判断的依据就是左端点的值小于$start$。按照左端点排序时，我们就可以通过**依次检索每个区间的左端点**，发现$a_i > start$时，前面的所有区间就是选择的范围了。但按照右端点进行排序时，并不能保证。

### 1.4.3代码实现

这体现的是我的一个调试过程啦hh~开始时代码是这样的：

```
#include<iostream>
#include<algorithm>
using namespace std;
#define II pair<int,int>
int s,t;//covered section
int N;//the sum of sections
II sec[100010];
bool lcmp(II a,II b){// sort by left point of section
    return a.first<b.first;
}
bool rcmp(II a,II b){
    return a.second<b.second;
}
int main(){
    cin>>s>>t;
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d %d",&sec[i].first,&sec[i].second);
    }
    //sort
    sort(sec,sec+N,lcmp);
    int start=s;//记录每一步的start
    int beg=0;//记录选择的区间范围
    int res=0;//结果
    while(start<t && beg<N){
        // while(sec[beg].second<start) beg++;
        int end = beg;
        while(sec[end].first<=start){
            end++;
            if(end == N) break;
        }
        //现在在sec[beg:end]中选择右端点最大的
        sort(sec+beg,sec+end-1,rcmp);
        //cout<<sec[end-1].first<<" "<<sec[end-1].second<<endl;
        beg = end;
        res++;
        start=sec[end-1].second;//更新start
    }
    if(start<t) cout<<-1;
    else cout<<res;
}
```

但是在如下的输入时，就是WA了：

```
-98 19
30
-66 53
-54 -13
41 48
-46 69
-6 59
```

```
-22 -12
-21 61
-8 6
-92 -54
-35 -4
-62 -4
-8 81
-86 -43
-49 31
-2 31
-85 -41
-63 -24
87 87
-47 -7
-67 -10
16 98
10 59
-54 18
-97 -41
-1 95
-81 64
-85 99
-42 79
-46 77
46 70
```

但是我们很容易发现，**线段区间(即要被覆盖的区间)的左端点是-98，但是所有待选的区间左端点最小值才为-97**，按照上述算法的执行过程，在遍历完所有的区间，我们都发现不了一个区间满足***sec[end].first<=start***，所以最后***end=0***，那你执行***sort(sec+beg,sec+end-1,rcmp)***当然就会报错***Segmentation Fault***了呀。

所以我们需要对这种情况进行单独的判断，若从最开始可选择的那个区间，其左端点就已经大于当前的***start***了，那肯定是不能无法覆盖了，算法结束，输出***-1***。这需要每一次更新***start***后都进行判断：

```
if(sec[end].first>start){
            cout<<-1;
            return 0;
        }
```

还有一点需要注意的是，***sort***排序，下标是从***0***到***n-1***，写成***sort(sec,sec+N,cmp)***，那在所有可选区间中，按照右端点排序，也应该是***sort(sec+beg,sec+end,cmp)***，并不需要***-1***。

这样，我们修改过后，再次进行提交，在下面的用例又提示为**WA**:

```
1 1
1
1 1
```

此时只需要考虑这一种特殊情况即可。它只有一个区间，按照我们的算法，***start=t***，所以事实上并没有进入***while***语句，自然就输出***0***了。我们并不能直接判断当***start=t***时，就输出***1***，因为有可能所有的区间并不会覆盖这一个点。我们需要遍历每个区间，判断有***sec[i].first<=t && sec[i].second>=t***时，那这个区间就覆盖了这个点(也就是题目的线段区间)，算法结束，输出***1***。

最后完整的代码为：

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
#define II pair<int,int>
int s,t;//covered section
int N;//the sum of sections
II sec[100010];
bool lcmp(II a,II b){// sort by left point of section
    return a.first<b.first;
}
bool rcmp(II a,II b){
    return a.second<b.second;
}
int main(){
    cin>>s>>t;
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d %d",&sec[i].first,&sec[i].second);
    }
    //sort
    sort(sec,sec+N,lcmp);
    int start=s;//记录每一步的start
    int beg=0;//记录选择的区间范围
    int res=0;//结果
    if(start==t){
        for(int i=0;i<N;i++){
            if(sec[i].first<=t && sec[i].second>=t){
                cout<<1;
                return 0;
            }
        }
    }
    else{
        while(start<t && beg<N){
        // while(sec[beg].second<start) beg++;
        int end = beg;
        if(sec[end].first>start){
            cout<<-1;
            return 0;
        }
        while(sec[end].first<=start){
            end++;
            if(end == N) break;
        }
        //现在在sec[beg:end]中选择右端点最大的
        sort(sec+beg,sec+end,rcmp);
        //cout<<sec[end-1].first<<" "<<sec[end-1].second<<endl;
        beg = end;//下次循环，但是也有可能已经超出区间的范围了
        res++;
        if(sec[end-1].second>start) start=sec[end-1].second;//更新start
        }
        if(start<t) cout<<-1;
        else cout<<res;
    }
}
```

那么在第一次WA时，也给我有一点启发，按照其用例，我们将所有区间表示出来：

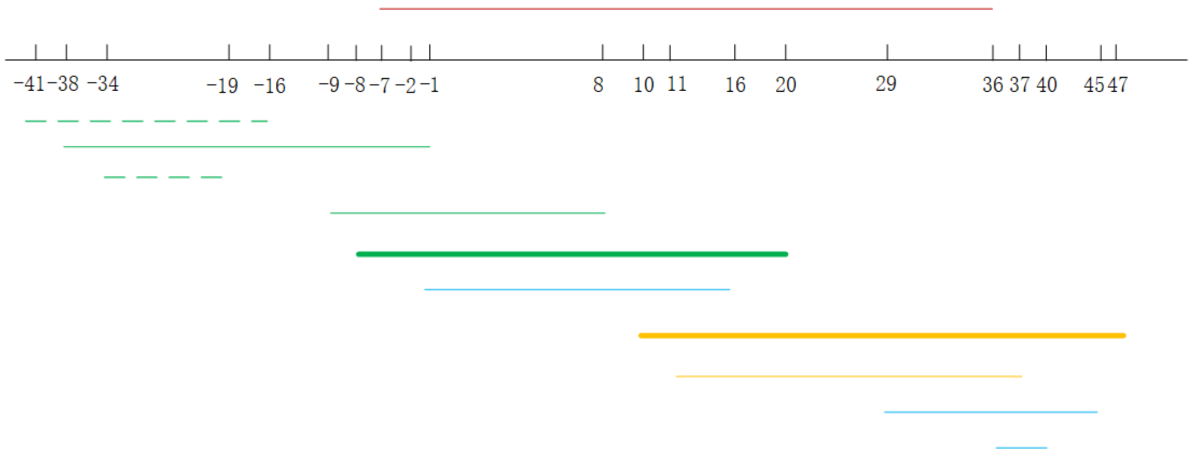


图4-2

在***start=-7***，也就是第一次进行选择时，绿色表示的区间，按照我们的算法，都是可选择的区间。但事实上，用虚线表示的两个区间，理应是不能够“被纳入考虑的”，因为其并不满足右端点$b_i > start$。但这是没有关系的，有两种情况：

1.右端点最大的那个区间，其$b_i > start$，就像图4-2这样，那反正也不会选择前面的区间，其右端点的值是什么也无关紧要了。

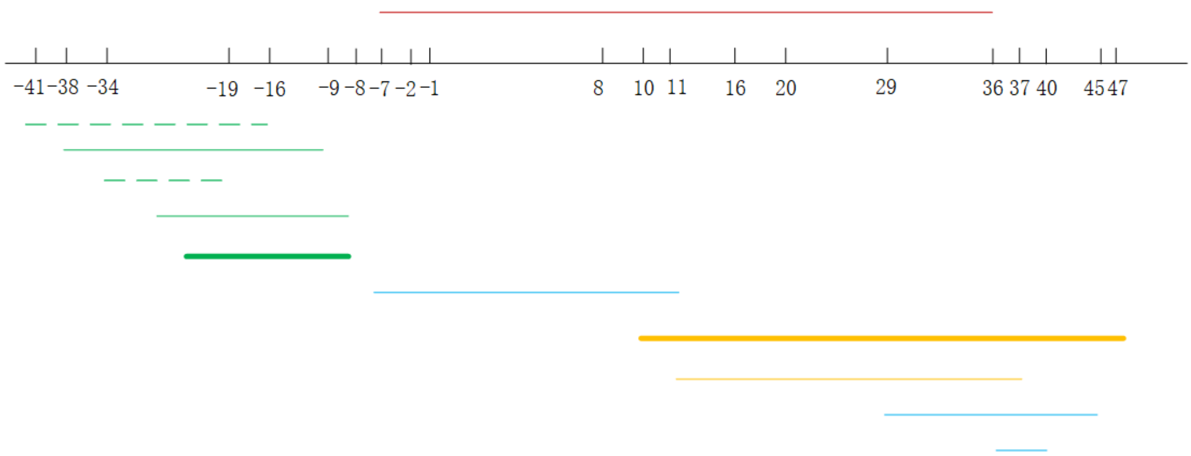


图4-3

2.就连右端点最大的那个区间，仍有$b_i < start$，如图4-3所示，按照我们的算法，还是选择加粗的绿色区间，但是此时若执行***start=sec[end-1].second***，那就是把要覆盖的区间扩大了，如图4-4中**红色被加粗的部分**，范围扩大了，就有可能会**误判**为不能够实现覆盖。所以一定要加上***if(sec[end-1].second>start) start=sec[end-1].second;***（但是在Acwing平台上好像并没有考虑这种情况hh~）

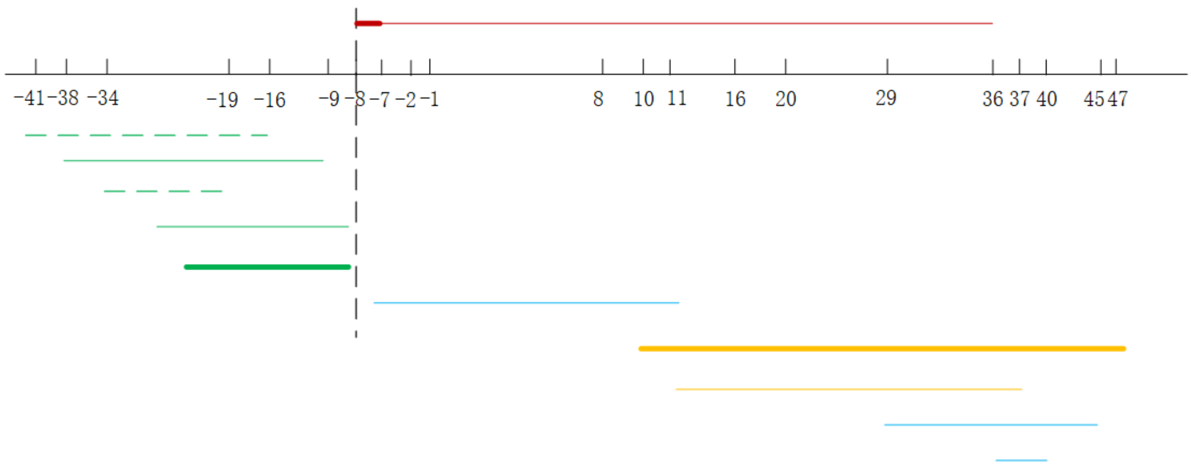


图4-4

# 2.Huffman树

## 2.1合并果子

### 2.1.1题目描述

在一个果园里，达达已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。

达达决定把所有的果子合成一堆。

每一次合并，达达可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。

可以看出，所有的果子经过$n-1$ 次合并之后，就只剩下一堆了。

达达在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以达达在合并果子时要尽可能地节省体力。

假定每个果子重量都为 $1$，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使达达耗费的体力最少，并输出这个最小的体力耗费值。

例如有 $3$ 种果子，数目依次为 $1,2,9$。

可以先将$1$、$2$ 堆合并，新堆数目为 $3$，耗费体力为 $3$。

接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 $12$，耗费体力为 $12$。

所以达达总共耗费体力$=3+12=15$。

可以证明 $15$ 为最小的体力耗费值。

**输入格式**

输入包括两行，第一行是一个整数 $n$，表示果子的种类数。

第二行包含 $n$ 个整数，用空格分隔，第 $i$ 个整数 $a_i$ 是第 $i$ 种果子的数目。

**输出格式**

输出包括一行，这一行只包含一个整数，也就是最小的体力耗费值。

输入数据保证这个值小于 $2^{31}$。

**数据范围**

$1 \le n \le 10000$,
$1 \le a_i \le 20000$

**输入样例**

```
3
1 2 9
```

**输出样例**

```
15
```

## 2.1.2题目分析

其实这里的大标题就是Huffman树，记得在数据结构中，Huffman树是应用在Huffman编码。都已经学过 数据结构啦，我们要记住，哈夫曼树也可以称作是**最优二叉树**，可以使得**带权路径最短**。并且我们也必须要清楚构造Huffman树的算法，就是每次选择权重最小的两个节点构造新树。

我们可以从两方面来考虑这个问题：

**角度1**

我们已经知道，通过Huffman树构造算法得到的带权路径长度一定是最小的。**一个搬运的次序方案，就对应了一个二叉树，并且方案最后花费的体力，也就是带权路径长度，其中叶子节点就对应了不同的堆。**

**因为叶子节点的深度，就代表了其要搬运的次数，所以最后花费的体力，就是所有叶子节点和其深度相乘之和，刚好就是带权路径长度**

所以我们贪心选择就是，**每次选择权重最小的两个节点进行合并**，Huffman树构造算法是正确的，这样我们也一定能得到全局最优解。

**角度2**

或许我们真的可以证明，哈夫曼算法的正确性。**可以参见《算法分析与设计(第4版)》**

设第***c***堆果子的重量为***f(c)***,又设***x***和***y***是所有果堆中重量最小的两堆。假设在全局最优解中，第一步选择的两个节点是***b***和***c***，对应的二叉树为***T***，那么如图1-1所示，我们在树***T***中交换叶子***b***和***x***的位置，得到树***T***″：

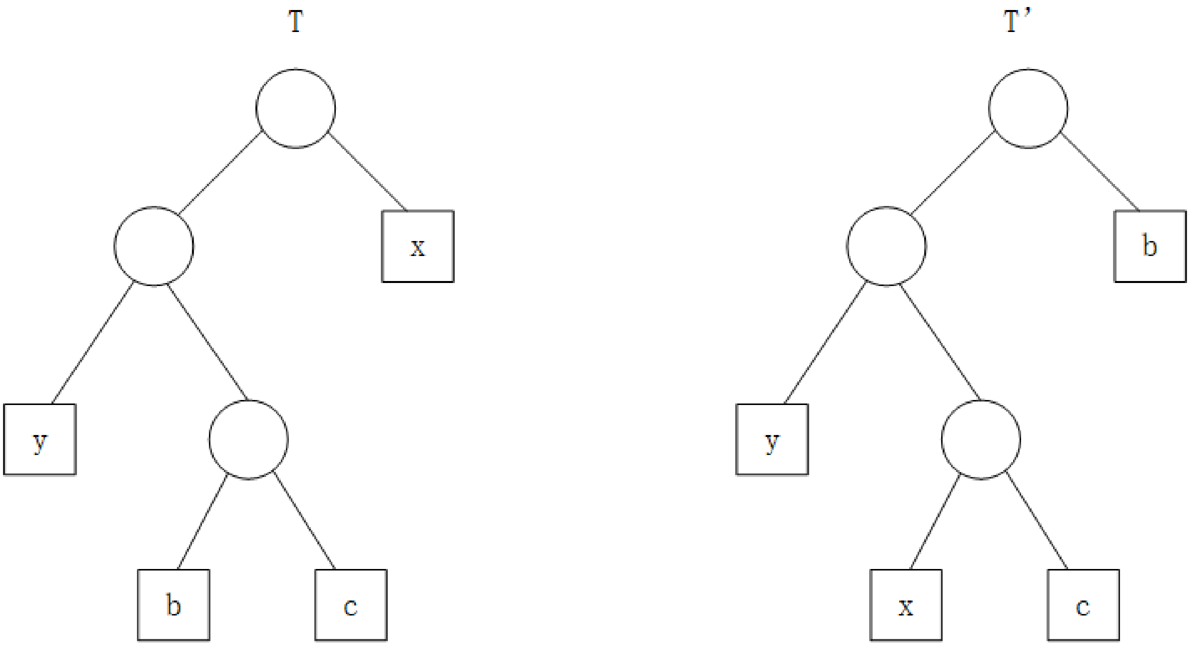


图1-1

$$B\left(T\right)=f\left(x\right)d_T\left(x\right)+f\left(y\right)d_T\left(y\right)+f\left(b\right)d_T\left(b\right)+f\left(c\right)d_T\left(c\right)$$
$$B\left(T^{'}\right)=f\left(x\right)d_{T^{'}}\left(x\right)+f\left(y\right)d_T\left(y\right)+f\left(b\right)d_{T^{'}}\left(b\right)+f\left(c\right)d_T\left(c\right)$$

带权路径之差为：

$$B\left(T\right)-B\left(T^{'}\right)=f\left(x\right)\left(d_T\left(x\right)-d_{T^{'}}\left(x\right)\right)+f\left(b\right)\left(d_T\left(b\right)-d_{T^{'}}\left(b\right)\right)$$

由于我们是交换了***x***和***b***的位置，因而有：

$$
\begin{gathered}
d_T(x) = d_{T'}(b) \\
d_T(b) = d_{T'}(x)
\end{gathered}
$$

所以有：

$$
\begin{aligned}
B(T) - B\left(T^{'}\right) &= f(x)(d_T(x) - d_T(b)) + f(b)(d_T(b) - d_T(x)) \\
&= (f(x) - f(b))(d_T(x) - d_T(b))
\end{aligned}
$$

那在***T***树中，我们首次选择了堆***b***和***c***，它们具有更大的深度，并且堆***x***和***y***是权重最小的两个节点，这样一定有：

$$
\begin{gathered}
d_T(x) < d_T(b) \\
f(x) < f(b)
\end{gathered}
$$

所以：

$$
B(T) - B\left(T^{'}\right) \geq 0
$$

即事实上，***T'***树会比***T***树具有更小的带权路径长度。

同样的，我们可以再次交换节点***y***和***c***得到树***T''***，用类似的方法，我们可以证明

$$
B\left(T^{'}\right) - B\left(T^{''}\right) \geq 0
$$

因此我们可以获得结论，对于全局最优解，若第一步选择的不是节点***x***和***y***，而是***b***和***c***，那么我们可以交换***x***和***b***、***y***和***c***，并且由于

$$
B(T) - B\left(T^{''}\right) \geq 0
$$

交换后的结果也是最优的，即第一步选择权重值最小的两个节点，一定在全局最优解中。而我们也可以证明，在后续的每一步选择权重最下哦的两个节点合并，也都在全局最优解中。事实上，这是该问题的**最优子结构**性质，这里先不进行说明了。

### 2.1.2代码实现

关于代码实现，我们每次要获得最小的权重，其实就很容易想到**优先队列**，刚好在合并子节点后，再用***heap.push()***插入。并且刚好STL也提供了***heap.size()***返回当前节点的数量，当***heap.size()=1***时，算法结束。

```
#include<iostream>
#include<queue>
using namespace std;
priority_queue<int,vector<int>,greater<int> > heap;
int N;
int main(){
    cin>>N;
    //将所有元素插入小顶堆
    while(N--){
        int w;
        scanf("%d",&w);
        heap.push(w);
    }
    int res = 0;//花费的体力数
```

```
    while(1){
        //若只剩下一个节点，算法结束
        if(heap.size()==1) break;
        else{
            int h1 = heap.top();
            heap.pop();
            int h2 = heap.top();
            heap.pop();
            int h3 = h1 + h2;//合并
            heap.push(h3);
            res+=h3;
        }
    }
    cout<<res;
}
```

注意我们这里用***res***记录最终合并花费的体力数，在每一次合并***heap.push()***后，更新***res***的值。

# 3.排序不等式

## 3.1排队打水

### 3.1.1问题描述

有 $n$个人排队到 $1$ 个水龙头处打水，第 $i$ 个人装满水桶所需的时间是 $t_i$，请问如何安排他们的打水顺序才能使所有人的等待时间之和最小？

**输入格式**

第一行包含整数 $n$。

第二行包含 $n$个整数，其中第 $i$个整数表示第 $i$ 个人装满水桶所花费的时间$t_i$。

**输出格式**

输出一个整数，表示最小的等待时间之和。

**数据范围**

$1 \le n \le 10^5$,

$1 \le t_i \le 10^4$

**输入样例**

```
7
3 6 1 4 2 5 7
```

**输出样例**

```
56
```

### 3.1.2题目分析

题目描述非常简单，在操作系统里面的CPU调度算法里面，其实我们已经知道了最短作业优先调度算法$SJF$会有最短的平均调度时间。

就像给的样例中，我们安排的次序为 1、2、3、4、5、6、7，那么平均等待的时间为：

$$
\begin{aligned}
Avg = 1+(1+2)+(1+2+3)+(1+2+3+4) \\
=(1+2+3+4+5)(1+2+3+4+5+6) = 1+3+6+10+15+21 = 56
\end{aligned}
$$

那输出样例就是56。

所以算法其实很简单，我们每次选择最小的装满水桶的时间即可。我们可以简单证明一下，为什么这样可以得到全局最优解：

假设现在有$n$个人，其打水花费的时间分别为$A_1$,$A_2$,......，$A_n$，并且其中最短的花费的时间为$A_i$。现在对于全局最优解$S$，假设其第一步的选择是$A_j$，那么有$A_i \le A_j$。现在我们交换$A_i$和$A_j$的次序得到一个序列$S'$。这样，$S$：$A_j$，$A_{s_1}$ ,$A_{s_2}$,......$A_i$......;$S'$：$A_i$，$A_{s_1}$,$A_{s_2}$,......$A_j$......。对于$S$中$A_i$后面的所有人，交换前后等待的时间是不变的。那对于$S$中从$A_{s_1}$到$A_i$中的人，其等待时间会增加$A_i - A_j$,但是有$A_i \le A_j$，也就是等待时间会减少$A_j - A_i$，那么对于序列$S'$，其花费的时间会减少$x(A_j - A_i)$，其中$x$就是$S$中从$A_{s_1}$到$A_i$中的人数。所以我们可以得出两个结论：要么$S$并不是全局最优解，因为$S'$花费了更少的时间；要么$S'$也是全局最优解，其花费和$S$相同。

因此我们总是可以交换$A_i$和$A_j$，也就是，第一步我们选择时间最短的$A_i$，其总是在全局最优解中。在进行第一步选择过后，我们得到了一个子问题，我们仍然选择剩余的人中，装满水桶的最短时间。

### 3.1.3代码实现

在最开始的时候，还在想要使用优先队列，但是其实并不需要。对于给定的花费时间的序列，我们只要进行排序，再求**前缀和**就好了。求前缀和在Acwing的第一章就讲过，也很简单。

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
vector<int> w;
int N;
int main(){
    cin>>N;
    for(int i=0;i<N;i++){
        int cost;
        cin>>cost;
        w.push_back(cost);
    }
    //排序
    sort(w.begin(),w.end());
    //求前缀和
    int s[100010];
    long long  res = 0;
    s[0]=0;
    for(int i=1;i<=N;i++){
        s[i]=s[i-1]+w[i-1];
        if(i!=N) res+=s[i];
    }
    cout<<res;
```

```
}
```

需要注意的两点:

1.在数据量增大的时候，最后的结果会很大，需要把***res***定义为***long long***类型

2.在求解前缀和过后，我们只需要累计前***N-1***个前缀和

# 4.绝对值不等式

## 4.1货仓选址

### 4.1.1题目描述

在一条数轴上有 ***N*** 家商店，它们的坐标分别为 $A_1$ ~ $A_N$。

现在需要在数轴上建立一家货仓，每天清晨，从货仓到每家商店都要运送一车商品。

为了提高效率，求把货仓建在何处，可以使得货仓到每家商店的距离之和最小。

**输入格式**

第一行输入整数 ***N*** 。

第二行 ***N*** 个整数 $A_1$ ~ $A_N$。

**数据范围**

$1 \le N \le 100000$,
$0 \le A_i \le 40000$

**输入样例**

```
4
6 2 9 1
```

**输出样例**

```
12
```

### 4.1.2题目分析

**自己的一点想法**

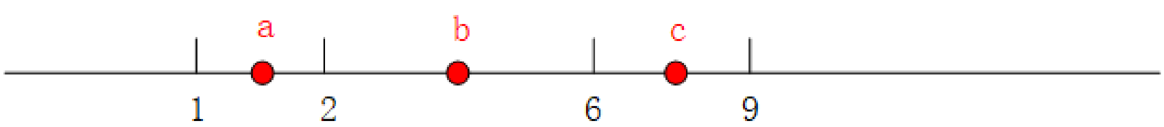


图1-1

如图1-1所示，我们并不会将点选在区间[1,9]之外。我们可以注意到，题目也没有问你具体货仓的位置，因为就像给的样例一样，在[1,2]中任意的位置，最后距离之和都是相同的。

于是，货仓的位置也就只有3种情况，分别为***a***、***b***、***c***，但是很显然我们应该选***b***，代价就是区间[1,9]和[2,6]距离之和，***(9-1)+(6-2)=12***。于是很自然而然的想到，若只有奇数个点，比如***1***、***6***、***9***,那我们货仓的位置就选在***6***，距离之和就是***(9-1)=8***。

因而初步的想法为，我们给坐标进行排序，若为偶数个点，从外到内，两两组成一个区间，货仓应该选在最内部的区间中；若为奇数个点，货仓和中间的点位置重合。这个证明我认为是显然的...，因为在区间内部，到两端点之和是固定且最小的，那这和贪心选择又有什么关系呢..

**代码实现**

```
#include<iostream>
#include<algorithm>
using namespace std;
int N;
int aix[100010];
int main(){
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d",&aix[i]);
    }
    sort(aix,aix+N);
    long long res = 0;
    for(int i=0;i<N/2;i++){
        int j = N-1-i;
        res += aix[j]-aix[i];
    }
    cout<<res;
}
```

然后居然Accept了？那这题到底想干嘛...在题目分析里面我们已经说了，就是由外到内两两个点形成一个区间，求其差值的绝对值。左边从***0***递增，右边从***N-1***递减，直到左端点为***N/2-1***。当为偶数个点时，***N/2-1***也是一个整数,最中间的区间就是***[(N/2)-1,N/2]***。而***N***是奇数时，事实上，在***i=(N/2)-1(向下取整)***时退出，此时***j=N-1-i=N/2(向上取整)***，最后没有管最中间的点了。

### 4.1.3代码实现

在4.1.2已经给出。

# 5.推公式

## 5.1耍杂技的牛

### 5.1.1题目描述

农民约翰的***N*** 头奶牛（编号为 ***1..N***）计划逃跑并加入马戏团，为此它们决定练习表演杂技。

奶牛们不是非常有创意，只提出了一个杂技表演：

叠罗汉，表演时，奶牛们站在彼此的身上，形成一个高高的垂直堆叠。

奶牛们正在试图找到自己在这个堆叠中应该所处的位置顺序。

这 ***N*** 头奶牛中的每一头都有着自己的重量 $W_i$ 以及自己的强壮程度 $S_i$。

一头牛支撑不住的可能性取决于它头上所有牛的总重量（不包括它自己）减去它的身体强壮程度的值，现在称该数值为风险值，风险值越大，这只牛撑不住的可能性越高。

您的任务是确定奶牛的排序，使得所有奶牛的风险值中的最大值尽可能的小。

**输入格式**

第一行输入整数 $N$，表示奶牛数量。

接下来 $N$ 行，每行输入两个整数，表示牛的重量和强壮程度，第 $i$ 行表示第 $i$ 头牛的重量 $W_i$以及它的强壮程度 $S_i$。

**输出格式**

输出一个整数，表示最大风险值的最小可能值

**输入样例**

```
3
10 3
2 5
3 3
```

**输出样例**

```
2
```

## 5.1.2题目分析

嗯..在这里有两个变量啊..所以我们肯定不能简单地想，越重的放在最下面，或者是越强壮的放在最下面。~~感觉从题目的意思来看，若强壮程度大于上面所有牛的重量，风险值取0~~。

我们可以考虑第一步贪心选择，若选择第$i$个牛，我们假设所有牛的总重量是$W$，这是固定不变的，那这头牛上面的总重量就是$W\text{-}W_i$ ,风险值就是$W\text{-}W_i\text{-}S_i\text{=}W\text{-}(W_i\text{+}S_i)$。到这里我们就会想，是不是每一次的贪心选择，是选择重量和身体强壮程度之和**最大**的牛。

可以简单进行证明一下。我们假设一个全局最优解$S$，其第一个选择为第$j$个牛，而第$i$个牛的重量和身体强壮程度之和最大，即$W_i\text{+}S_i\text{>}W_j\text{+}S_j$。那我们交换第$i$个牛和第$j$个牛的位置，得到另外一个排序$S'$。在$S$中，第$i$个牛上面的牛，并没有影响。对于第$i$个牛下面的牛，由于上面有一个牛的重量从$W_i$变为了$W_j$，所以风险值会增加$x(W_j\text{-}W_i)$,其中$x$是$S$中第$i$个牛下面的数量。但是好像$S$没有变?

**嗷题目理解错了，题目是说所有奶牛的风险值中的最大值最小**。像给的样例中，从下到上按照体重为$10$、$2$、$3$的重量进行排序，那么风险值依次为$(3+2)\text{-}3\text{=}2$、$3\text{-}5\text{=-}2$、$0\text{-}3\text{=-}3$，那$max(2,\text{-}2,\text{-}3)\text{=}2$。

那还是这么考虑，对于全局最优解$S$，若其第一个选择为第$j$个牛，但是其在第$x$个奶牛的风险值最大。有两种情况：

1.$j\text{=}x$，恰好是第一个牛的风险值最大，那我们交换第$i$个和第$j$个奶牛的位置。那么编号为$j$的风险值就从$W\text{-}(W_j\text{+}S_j)$变成了$W'\text{-}S_j$，其中$W'$是其上面牛的重量。编号为$i$的风险值变为了$W\text{-}(W_i\text{+}S_i)$。对于其中间的奶牛，其风险值都会增加$W_j\text{-}W_i$。**但是我并不能保证$W_j \le W_i$,也就是并不能说明所有奶牛的风险值的最大值变成了$W\text{-}(W_i\text{+}S_i)$，会更小**，好像证明不下去了。

那在Acwing中给出这样的证明，事实上我们在上面的分析过程中，交换编号为$i$和$j$的奶牛，中间的奶牛我们不好分析，所以我们改成相邻的奶牛：

我们给出的贪心选择是，每次选择$W_i\text{+}S_i$最大的奶牛，从下往上排序。我们令按照该方法得到的一个排序为$S'$。对于全局最优解$S$，若$S\neq S'$，那么必定存在相邻的两头奶牛，位置为$i$和$i\text{+}1$，并且$W_i\text{+}S_i \ge W_{i+1}\text{+}S_{i+1}$。那么我们交换两头奶牛的位置，该操作对其他奶牛并不会带来任何影响。那么对于奶牛$i$，其增加了一个第$i\text{+}1$个奶牛的重量，所以其风险值会增加$W_{i+1}$，而对于第$i\text{+}1$个奶牛，其风险值会减小$W_i$。

但是我们要注意的是，我们假设原先第$i$头奶牛上面的重量为$x$，原先第$i$头奶牛的风险值为$x\text{-}S_i$，现在第$i$头奶牛的风险为$x+W_{i+1}\text{-}S_i$。第$i+1$头奶牛的风险值为$x+W_i\text{-}S_{i+1}$，现在变为$x\text{-}S_{i+1}$.对于这四个表达式，我们同时加上$S_i$，得到$x$，$x+W_{i+1}$，$x+W_i+S_i\text{-}S_{i+1}$，$x+S_i\text{-}S_{i+1}$。再加上$S_{i+1}$，得到$x+S_{i+1}$，$x+W_{i+1}+S_{i+1}$，$x+W_i+S_i$ ,$x+S_i$.由于$x+W_i+S_i \geq x+W_{i+1}+S_{i+1}$，所以在交换过后，风险的最大值一定不会增大。(注意，我们不需要去理会$x+S_{i+1}$和$x+S_i$，因为当$x+S_{i+1} > x+W_i+S_i$，有$x+W_{i+1}+S_{i+1} < x+W_i+S_i < x+S_{i+1}$,以及$x+S_i < x+W_i+S_i < x+S_{i+1}$ )。

所以对于$W_i+S_i$最大的奶牛，在路途上，总是最大的，我们总是可以进行交换且不会增大风险值的最大值，那么最底下的奶牛一定是有最大的$W_i+S_i$。同理，我们可以把$W_i+S_i$放到倒数第二个位置。最后我们可以获得全局最优解：按照$W_i+S_i$从小到大的顺序排序。

（我真的晕了）

### 5.1.3代码实现

```
#include<iostream>
#include<algorithm>
using namespace std;
#define II pair<int,int>
II cow[50010];
int N;
bool cmp(II a,II b){
    //按照w+s排序
    return a.first+a.second<b.first+b.second;
}
int main(){
    cin>>N;
    for(int i=0;i<N;i++){
        scanf("%d %d",&cow[i].first,&cow[i].second);
    }
    sort(cow,cow+N,cmp);
    //求前缀和
    int s[50010];
    s[0]=0;
    for(int i = 1;i<N;i++){
        s[i]=s[i-1]+cow[i-1].first;
    }
    int res = -1e9;//最大值
    for(int i=0;i<N;i++){
        if(s[i]-cow[i].second>res) res=s[i]-cow[i].second;
    }
    cout<<res;
}
```

涉及到前缀和，但是前面都已经提及过了，很简单。

# 总结

今天已经是7.19了，记得写这篇记录是上上上礼拜周末，也有十多天了..当然，中间还在小学期，而且结束后玩了些天，所以按正常来说，8道题是用不着花这么久时间的，尽管写了一些笔记。

对于贪心算法， 在《算法分析与设计》的课程中接触过，可能最主要的还是得想到该如何做贪心选择，以及由局部最优解得到全局最优解的证明，一般会用交换法。但是在最后一题里面，证明稍微有一些不同，并没有着手于每一步的选择，而是直接从经过贪心选择后得到的最终结果入手，证明其可以通过全局最优解交换得到。可能还要刷一下题把...，后续碰到也会放在这个目录下的！