



浙江财经大学

Zhejiang University Of Finance & Economics



贪心

信智学院 陈琰宏



教学内容



01

熟练应用贪心法解决一些实际问题

02

体验贪心法的审题分析和细节测试

什么是贪心?

实际生活中，经常需要求一些问题的“可行解”和“最优解”，这就是所谓的“最优化”问题。

一般来说，每个最优化问题都包含一组“限制条件”和一个“目标函数”，符合限制条件的问题求解方案称为可行解，使目标函数取得最佳值（最大或最小）的可行解称为最优解。

求解最优化问题的算法很多，例如穷举、搜索、动态规划等。贪心法也是求解这类问题的一种常用方法。

1 贪心理解

贪心法是从问题的某个初始解出发，采用逐步构造最优解的方法，向给定的目标前进。在每一个局部阶段，都做一个“看上去”最优的决策，并期望通过每一次所做的局部最优选择产生出一个全局最优解。即在对问题求解时，**总是做出在当前看来是最好的选择**。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

与递推不同的是，贪心严格意义上说只是一种策略或方法，而不是算法。**推进的每一步不是依据某一个固定的递推式，而是做一个当时“看似最佳”的贪心选择（操作）**，不断将问题归纳为更小的相似子问题。所以，归纳、分析、选择正确合适的贪心策略，是解决贪心问题的关键。



1 贪心理解

【例1】在N行M列的正整数矩阵中，要求从每行中选出1个数，使得选出的总共N个数的和最大。

【算法分析】

要使总和最大，则每个数要尽可能大，自然应该选每行中最大的那个数。因此，我们设计出如下算法：

读入N, M, 矩阵数据；

Total = 0;

for (int i = 1; i <= N; i++)

{ //对N行进行选择

 选择第i行最大的数,记为K;

 Total += K;

}

$$\begin{bmatrix} 5 & 6 & 19 & 23 & 56 \\ 32 & 123 & 5 & 1 & 23 \\ 66 & 1 & 23 & 567 & 2 \\ 100 & 23 & 4 & 231 & 3 \end{bmatrix}$$


1 贪心理解

【例2】部分背包问题

给定一个**最大载重量为 M** 的卡车和 **N 种食品**，有食盐，白糖，大米等。已知第 i 种食品的最多拥有 W_i 公斤，其商品价值为 V_i 元/公斤，编程确定一个装货方案，使得装入卡车中的所有物品总价值最大。

【算法分析】

因为每一个物品都可以分割成单位块，单位块的利益越大显然总收益越大，所以它局部最优满足全局最优，可以用贪心法解答，方法如下：先将单位块收益按从大到小进行排列，然后用循环从单位块收益最大的取起，直到不能取为止便得到了最优解。



1 贪心理解

问题初始化;

//读入数据

按 V_i 从大到小将商品排序;

$i=1$;

do

{

if ($m==0$) break; //如果卡车满载则跳出循环

$m=m-w[i]$;

if ($m>=0$) 将第 i 种商品全部装入卡车

else 将($m-w[i]$) 重量的物品 i 装入卡车;

$i=i+1$; //选择下一种商品

}while ($m>0 \& \& i \leq n$);

在解决上述问题的过程中, 首先根据题设条件, **找到贪心选择标准**
(V_i), 并依据这个标准直接逐步去求最优解, 这种解题策略被称为贪心法。



1 贪心理解

因此，利用贪心策略解题，需要解决两个问题：

首先，确定问题是否能用贪心策略求解；一般来说，**适用于贪心策略求解的问题具有以下特点：**

可通过局部的贪心选择来达到问题的全局最优解。运用贪心策略解题，一般来说需要一步步的进行多次的贪心选择。在经过一次贪心选择之后，原问题将变成一个相似的，但规模更小的问题，而后的每一步都是当前看似最佳的选择，且每一个选择都仅做一次。



2.1 [3867] 加勒比海盗船

海盗们截获了一艘装满各种各样古董的货船，每一件古董都价值连城，一旦打碎就失去了它的价值。虽然海盗船足够大，但载重量为 C ，每件古董的重量为 w_i ，海盗们该如何把尽可能多数量的宝贝装上海盗船呢？

输入：第一行是一个整型数 m ($m < 100$) 表示共有 m 组测试数据。

每组测试数据的第一行是两个整数 c, n ($1 < c, n < 10000$) 表示该测试数据载重量 c 及古董的个数 n 。

输出

对于每一组输入，输出能装入的古董最大数量。

每组的输出占一行

样例输入

2

30 8

4 10 7 11 3 5 14 2

45 10

5 12 7 3 20 9 15 11 8 32

样例输出

5

6

问题分析

根据问题描述可知这是一个可以用贪心算法求解的**最优装载问题**，要求装载的物品的数量尽可能多，而船的容量是固定的，那么优先把重量小的物品放进去，在容量固定的情况下，装的物品最多。**采用重量最轻者先装的贪心选择策略**，从局部最优达到全局最优，从而产生最优装载问题的最优解。

(1) 当载重量为定值 c 时， w_i 越小时，可装载的古董数量 n 越大。只要依次选择最小重量古董，直到不能再装为止。

(2) 把 n 个古董的重量从小到大（非递减）排序，然后根据贪心策略尽可能多地选出前 i 个古董，直到不能继续装为止，此时达到最优。

问题分析

初始： 海盗船的载重量 c 为 30

重量	$w[i]$	4	10	7	11	3	5	14	2
----	--------	---	----	---	----	---	---	----	---

排序后：

重量	$w[i]$	2	3	4	5	7	10	11	14
----	--------	---	---	---	---	---	----	----	----

$i=0$, 选择排序后的第 1 个, 装入重量 $tmp=2$, 不超过载重量 30, $ans=1$ 。
 $i=1$, 选择排序后的第 2 个, 装入重量 $tmp=2+3=5$, 不超过载重量 30, $ans=2$ 。
 $i=2$, 选择排序后的第 3 个, 装入重量 $tmp=5+4=9$, 不超过载重量 30, $ans=3$ 。
 $i=3$, 选择排序后的第 4 个, 装入重量 $tmp=9+5=14$, 不超过载重量 30, $ans=4$ 。
 $i=4$, 选择排序后的第 5 个, 装入重量 $tmp=14+7=21$, 不超过载重量 30, $ans=5$ 。
 $i=5$, 选择排序后的第 6 个, 装入重量 $tmp=21+10=31$, 超过载重量 30, 算法结束。

即放入古董的个数为 $ans=5$ 个

代码

```
1  #include<iostream>
2  #include<algorithm>
3  using namespace std;
4  const int N=10005;
5  int w[N];//定义古董的重量数组
6
7  int main()
8  {
9      int n,m;
10     int c,tmp,ans; //tmp临时重量, ans表示求得的件数
11     cin>>m;
12     for(int i=0;i<m;i++){
13         cin>>c>>n;//输入重量c及古董数量
14         for(int j=0;j<n;j++)
15             cin>>w[j];
16         sort(w,w+n);//按照古董重量升序排序
17         tmp=0,ans=0;// 初始化
18         for(int k=0;k<n;k++){
19             tmp=tmp+w[k];//选取古董
20             if(tmp<=c)ans++;//未装满
21             else
22                 break;
23         }
24         cout<<ans<<endl;
25     }
26     return 0;
27 }
```

填空

```
4  const int N=10005;
5  int w[N]; //定义古董的重量数组
6
7  int main()
8  {
9      int n,m;
10     int c,tmp,ans; //tmp临时重量, ans表示求得的件数
11     cin>>m;
12     for(int i=0;i<m;i++){
13         cin>>c>>n; //输入重量c及古董数量
14         for(int j=0;j<n;j++){
15             cin>>w[j];
16             _____1_____ //按照古董重量升序排序
17             _____2_____; // 初始化
18         }
19         for(int k=0;k<n;k++){
20             _____3_____ //选取古董
21             if(_____4_____) ans++; //未装满
22             else
23                 _____5_____;
24         }
25         cout<<ans<<endl;
26     }
27     return 0;
28 }
```

2.2 [3868] 阿里巴巴与四十大盗

假设山洞中有 n 种宝物，每种宝物有一定重量 w 和相应的价值 v ，毛驴运载能力有限，只能运走 m 重量的宝物，一种宝物只能拿一样，宝物可以分割。那么怎样才能使毛驴运走宝物的价值最大呢？

输入：每组测试数据的第一行是两个整数 n, c ($1 < n, c < 10000$) 表示该测试数据宝物数量及驴子的承载重量。随后的 n 行，每行有两个正整数 w_i, v_i 分别表示第 i 个宝物的重量和价值 ($1 < w_i, v_i < 100$)。

输出：对于每一组输入，输出毛驴运走宝物的最大价值。每组的输出占一行，结果保留一位小数。

样例输入

6 19

2 8

6 1

7 9

4 3

10 2

3 4

样例输出

24.6

问题分析

我们可以尝试贪心策略：

- (1) 每次挑选价值最大的宝物装入背包，得到的结果是否最优？
- (2) 每次挑选重量最小的宝物装入，能否得到最优解？
- (3) 每次选取单位重量价值最大的宝物，能否使价值最高？

如果选价值最大的宝物，但重量非常大，也是不行的，因为运载能力是有限的，所以第1 种策略舍弃；如果选重量最小的物品装入，那么其价值不一定高，所以不能在总重限制的情况下保证价值最大，第2 种策略舍弃；而第3 种是每次选取单位重量价值最大的宝物，也就是说每次选择性价比（价值/重量）最高的宝物，如果可以达到运载重量 m 那么一定能得到价值最大。因此采用第 3 种贪心策略，每次从剩下的宝物中选择性价比最高的宝物。

问题分析

(1) 数据结构及初始化。将 n 种宝物的重量和价值存储在结构体 `three` (包含重量、价值、性价比 3 个成员) 中, 同时求出每种宝物的性价比也存储在对应的结构体 `three` 中, 将其按照性价比从高到低排序。采用 `sum` 来存储毛驴能够运走的最大价值, 初始化为 0。

(2) 根据贪心策略, 按照性价比从大到小选取宝物, 直到达到毛驴的运载能力。每次选择性价比高的物品, 判断是否小于 m (毛驴运载能力), 如果小于 m , 则放入, `sum` (已放入物品的价值) 加上当前宝物的价值, m 减去放入宝物的重量; 如果不小于 m , 则取该宝物的一部分 $m * p[i]$, $m=0$, 程序结束。 m 减少到 0, 则 `sum` 得到最大值。

问题分析

假设现在有一批宝物，价值和重量如表所示，毛驴运载能力 $m=30$ ，那么怎么装入最大价值的物品？

宝物	i	1	2	3	4	5	6	7	8	9	10
重量	w[i]	4	2	9	5	5	8	5	4	5	5
价值	v[i]	3	8	18	6	8	20	5	6	7	15

因为贪心策略是每次选择性价比（价值/重量）高的宝物，可以按照性价比降序排序，排序后如表所示。

问题分析

排序后如表如下所示：

宝物	i	2	10	6	3	5	8	9	4	7	1
重量	w[i]	2	5	8	9	5	4	5	5	5	4
价值	v[i]	8	15	20	18	8	6	7	6	5	3
性价比	p[i]	4	3	2.5	2	1.6	1.5	1.4	1.2	1	0.75

(2) 按照贪心策略，每次选择性价比高的宝物放入：

第 1 次选择宝物 2，剩余容量 $30-2=28$ ，目前装入最大价值为 8。

第 2 次选择宝物 10，剩余容量 $28-5=23$ ，目前装入最大价值为 $8+15=23$ 。

第 3 次选择宝物 6，剩余容量 $23-8=15$ ，目前装入最大价值为 $23+20=43$ 。

第 4 次选择宝物 3，剩余容量 $15-9=6$ ，目前装入最大价值为 $43+18=61$ 。

第 5 次选择宝物 5，剩余容量 $6-5=1$ ，目前装入最大价值为 $61+8=69$ 。

第 6 次选择宝物 8，发现上次处理完时剩余容量为 1，而 8 号宝物重量为 4，无法全部放入，那么可以采用部分装入的形式，装入 1 个重量单位，因为 8 号宝物的单位重量价值为 1.5，因此放入价值 $1 \times 1.5 = 1.5$ ，你也可以认为装入了 8 号宝物的 $1/4$ ，目前装入最大价值为 $69+1.5=70.5$ ，剩余容量为 0。

代码

```
1  #include<iostream>
2  #include<algorithm>
3  using namespace std;
4  const int M=1000005;
5  □ struct three{
6      double w;// 每个宝物的重量
7      double v;// 每个宝物的价值
8      double p;// 性价比
9  }s[M];
10 bool cmp(three a,three b)
11 □ {
12     return a.p>b.p;// 根据宝物的单位价值从大到小排序
13 }
```

代码

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  struct bao_wu{
4      int w;//每个宝物的重量
5      int v;//每个宝物的价值
6      double p;//性价比
7  }bw[10001];
8  int cmp(bao_wu a, bao_wu b){
9      return a.p>b.p;
10     //根据性价比从大到小排序
11 }
12 int main(){
13     int n,c;
14     double sum=0;
15     cin>>n>>c;
16     for(int i=0;i<n;i++){
17         cin>>bw[i].w>>bw[i].v;//输入
18         bw[i].p=1.0*bw[i].v/bw[i].w;//计算性价比
19     } //输入数据, 初始化
20     sort(bw,bw+n,cmp);//对决策参数作排序操作
21     for(int i=0;i<n;i++){
22         if(bw[i].w<=c){
23             sum=sum+bw[i].v; //获得价值
24             c=c-bw[i].w;//装入背包, 重量减小
25         }
26         else{//没法把当前物品整个装入, 进行分割
27             sum=sum+c*bw[i].p;
28             break;
29         }
30     }
31     printf("%.11f",sum);
32 }
33 }
```

填空

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  struct bao_wu{
4      _____1_____ ;
5  }bw[10001];
6  int cmp(bao_wu a, bao_wu b){
7      _____2_____ ;
8      //根据性价比从大到小排序
9  }
10 int main(){
11     int n,c;
12     double sum=0;
13     cin>>n>>c;
14     for(int i=0;i<n;i++){
15         cin>>bw[i].w>>bw[i].v;//输入
16         bw[i].p=_____3_____ ;//计算性价比
17     } //输入数据, 初始化
18     sort(bw,bw+n,cmp);//对决策参数作排序操作
19     for(int i=0;i<n;i++){
29         printf("%.11f",sum);
30         return 0;
31     }
19     for(int i=0;i<n;i++){
20         if(_____4_____){
21             _____5_____ ;//获得价值
22             _____6_____ ;//装入背包, 重量减小
23         }
24         else{//没法把当前物品整个装入, 进行分割
25             _____7_____ ;
26             _____8_____ ;
27         }
28     }
29     printf("%.11f",sum);
```

2.3 [3407] 看电视-会议安排问题

会议安排的目的是能在有限的时间内召开更多的会议（任何两个会议不能同时进行）。

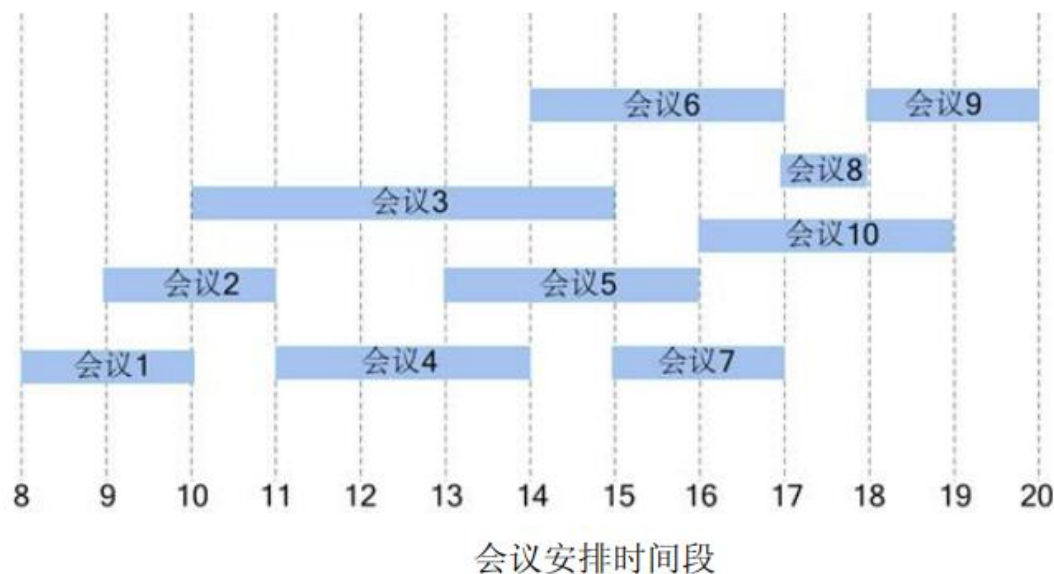
在会议安排中，每个会议 i 都有起始时间 b_i 和结束时间 e_i ，且 $b_i < e_i$ ，即一个会议进行的时间为半开区间 $[b_i, e_i)$ 。如果 $[b_i, e_i)$ 与 $[b_j, e_j)$ 均在“有限的时间内”，且不相交，则称会议 i 与会议 j 相容的。也就是说，当 $b_i \geq e_j$ 或 $b_j \geq e_i$ 时，会议 i 与会议 j 相容。会议安排问题要求在所给的会议集合中选出最大的相容活动子集，即尽可能在有限的时间内召开更多的会议。

在这个问题中，“有限的时间内（这段时间应该是连续的）”是其中的一个限制条件，也应该是有一个起始时间和一个结束时间（简单化，起始时间可以是会议最早开始的时间，结束时间可以是会议最晚结束的时间）。



2.3 [3407] 看电视-会议安排问题

会议	i	1	2	3	4	5	6	7	8	9	10
开始时间	bi	8	9	10	11	13	14	15	17	18	16
结束时间	ei	10	11	15	14	16	17	17	18	20	19



2.3 [3407] 看电视-会议安排问题

暑假到了，小明终于可以开心的看电视了。但是小明喜欢的节目太多了，他希望尽量多的看到完整的节目。

现在他把他喜欢的电视节目的转播时间表给你，你能帮他合理安排吗？

输入

输入包含多组测试数据。每组输入的第一行是一个整数 n

($n \leq 100$)，表示小明喜欢的节目的总数。

接下来 n 行，每行输入两个整数 s_i 和 e_i ($1 \leq i \leq n$)，表示第 i 个节目的开始和结束时间，为了简化问题，每个时间都用一个正整数表示。

当 $n=0$ 时，输入结束。

输出

对于每组输入，输出能完整看到的电视节目的个数。

样例输入

12
1 3
3 4
0 7
3 8
15 19
15 20
10 15
8 18
6 12
5 10
4 14
2 9
0

样例输出

5



分析

要让看的节目数最多，我们需要选择最多的不相交时间段。我们可以尝试贪心策略：

- （1）每次从剩下未安排的节目中选择会议具有**最早开始时间**且与已安排的会议相容的节目安排，以增大时间资源的利用率。
- （2）每次从剩下未安排的会议中选择**持续时间最短**且与已安排的节目相容的节目安排，这样可以安排更多一些的节目。
- （3）每次从剩下未安排的节目中选择具有**最早结束时间**且与已安排的会议相容的节目安排，这样可以尽快安排下一个会议。



分析

(1) 根据贪心策略就是选择第一个具有最早结束时间的节目，用 **last** 记录刚选中节目的结束时间；

(2) 选择第一个节目之后，依次从剩下未安排的节目中选择，如果会议 i 开始时间大于等于最后一个选中的节目的结束时间 **last**，那么节目 i 与已选中的节目相容，可以安排，更新 **last** 为刚选中会议的结束时间；否则，舍弃会议 i ，检查下一个会议是否可以安排。



贪心过程模拟

原始会议时间表

会议	num	1	2	3	4	5	6	7	8	9	10	11	12
开始时间	beg	1	3	0	3	15	15	10	8	6	5	4	2
结束时间	end	3	4	7	8	19	20	15	18	12	10	14	9

排序后的会议时间表:

会议	num	1	2	3	4	12	10	9	11	7	8	5	6
开始时间	beg	1	3	0	3	2	5	6	4	10	8	15	15
结束时间	end	3	4	7	8	9	10	12	14	15	18	19	20



贪心过程模拟

- (1) 首先选择排序后的第一个会议即最早结束的会议（编号为 1），用 `last` 记录最后一个被选中会议的结束时间，`last=3`。
- (2) 检查余下的会议，找到第一个开始时间大于等于 `last`（`last=3`）的会议，子问题转化为从该会议开始，余下的所有会议。如表所示

会议	num	1	2	3	4	12	10	9	11	7	8	5	6
开始时间	beg	1	3	0	3	2	5	6	4	10	8	15	15
结束时间	end	3	4	7	8	9	10	12	14	15	18	19	20



```

1  #include<bits/stdc++.h>
2  using namespace std;
3  struct meeting{
4      int s;//开始
5      int e;//结束
6  }meet[101];
7
8  int cmp(meeting a, meeting b){
9      return a.e<b.e;
10 }
11 int main(){
12     int last,n;
13     int sum;
14     while(cin>>n){
15         if(n==0)break;
16         for(int i=0;i<n;i++){
19             sort(meet,meet+n,cmp);
20             sum=1;//选择第一个会议
21             last=meet[0].e;
22             //当前会议的结束时间
23             for(int j=1;j<n;j++){
30                 cout<<sum<<endl;
31             }
32             return 0;
33     }

```

```

14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

```

```

while(cin>>n){
    if(n==0)break;
    for(int i=0;i<n;i++){
        cin>>meet[i].s>>meet[i].e;
    }//输入数据
    sort(meet,meet+n,cmp);
    sum=1;//选择第一个会议
    last=meet[0].e;
    //当前会议的结束时间
    for(int j=1;j<n;j++){
        //扫描除第一个会议外的所有会议
        if(meet[j].s>=last){
            sum++;
            last=meet[j].e;//更新last
        }
    }
    cout<<sum<<endl;
}

```

```

2  using namespace std;
3  _____1_____//定义一个会议 meet[101];
4
5  int cmp(meeting a, meeting b){
6      _____2_____
7  }
8  int main(){
9      int last,n;
10     int sum;
11     while(cin>>n){
12         if(n==0)break;
13         for(int i=0;i<n;i++){
14             cin>>meet[i].s>>meet[i].e;
15         }//输入数据
16         _____3_____//排序
17         sum=1;//选择第一个会议
18         last=_____4_____;//当前会议的结束时间
19         for(int j=1;j<n;j++){//扫描除第一个会议外的所有会议
20             if(_____5_____t){
21                 sum++;
22                 last=_____6_____;//更新last
23             }
24         }
25         cout<<sum<<endl;
26     }
27     return 0;
28 }

```

2.4 [3780]排队打水问题

有 n 个人排队到 m 个水龙头去打水，他们装满水桶的时间 $t_1, t_2, \dots, t_n$ 为整数且各不相同，应如何安排他们的打水顺序才能使他们花费的总时间最少？

输入 $n\ m$ 输出 所有人的花费时间总和

样例输入

4 2

2 6 4 5

样例输出

23



分析

由于排队时，越靠前面的计算的次数越多，显然越小的排在越前面得出的结果越小，所以这道题可以用贪心法解答，基本步骤：

- (1)将输入的时间按从小到大排序；
- (2)将排序后的时间按顺序依次放入每个水龙头的队列中；
- (3)统计，输出答案。

样例输入

4 2

2 6 4 5

样例输出

23

总时间=每个人的打水时间+等待时间

总时间=2+4+7+3=23



分析

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int main(){
4      int i,j,n,m,sum=0;
5      int a[1001],cost[1001];
6      memset(cost,0,sizeof(cost)); //初始化每个人的花费时间为0
7      cin>>n>>m; //n表示 人数 m表示水龙头个数
8      for(i=1;i<=n;i++)
9          cin>>a[i];
10     sort(a+1,a+1+n);
11     j=0,sum=0; //j表示当前排的水龙头的编号
12     for(i=1;i<=n;i++){
13         j++;
14         if(j==m+1)
15             j=1; //排队打水回到第一个水龙头的位置
16         cost[j]=cost[j]+a[i]; //上一个人的花费时间就是下一个人的等待时间
17         sum=sum+cost[j]; //;
18     }
19     cout<<sum<<endl;
20     return 0;
21 }
```

填空

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int main(){
4      int i,j,n,m,sum=0;
5      int a[1001],cost[1001];
6      memset(cost,0,sizeof(cost)); //初始化每个人的花费时间为0
7      cin>>n>>m; //n表示 人数 m表示水龙头个数
8      for(i=1;i<=n;i++)
9          cin>>a[i];
10     sort(__1__);
11     __2__ //j表示当前排的水龙头的编号
12     for(i=1;i<=n;i++){
13         j++;
14         if(j==__3__)
15             j=__4__; //排队打水回到第一个水龙头的位置
16         __5__; //上一个人的花费时间就是下一个人的等待时间
17         sum=sum+__6__;
18     }
19     cout<<sum<<endl;
20     return 0;
21 }
```

2.5 [3066]均分纸牌

有 N 堆纸牌，编号分别为 $1, 2, \dots, N$ 。每堆上有若干张，但纸牌总数必为 N 的倍数。可以在任一堆上取若干张纸牌，然后移动。

移牌规则为：在编号为 1 堆上取的纸牌，只能移到编号为 2 的堆上；在编号为 N 的堆上取的纸牌，只能移到编号为 $N-1$ 的堆上；其他堆上取的纸牌，可以移到相邻左边或右边的堆上。

现在要求找出一种移动方法，用**最少的移动次数使每堆上纸牌数都一样多**。

输入：每个测试文件只包含一组测试数据，每组输入的第一行输入一个整数 N ($1 \leq N \leq 100$)，表示有 N 堆纸牌。
接下来一行输入 N 个整数 $A_1 A_2 \dots A_n$ ，表示每堆纸牌初始数， $1 \leq A_i \leq 10000$ 。

输出：对于每组输入数据，输出所有堆均达到相等时的最少移动次数。



2.5 [3066]均分纸牌

例如 $N=4$,

4 堆纸牌数分别为: ① 9 ② 8 ③ 17 ④ 6

移动3次可达到目的:

1. 从 ③ 取4张牌放到④ (9 8 13 10)
2. 从③取3张牌放到 ② (9 11 10 10)
3. 从②取1张牌放到① (10 10 10 10)

[输入格式]

N (N 堆纸牌, $1 \leq N \leq 100$)

$A_1 A_2 \cdots A_n$ (N 堆纸牌, 每堆纸牌初始数, $1 \leq A_i \leq 10000$)

[输出格式】

所有堆均达到相等时的最少移动次数。

[样例输入]

4
9 8 17 6

[样例输出]

3

分析

把每堆牌的张数减去平均张数，题目就变成移动正数，加到负数中，使大家都变成0，那就意味着成功了一半！

平均张数为10,原张数9, 8, 17, 6, 变为
-1, -2, 7, -4, 其中没有为0的数,

我们从左边出发：

1. 要使第1堆的牌数-1变为0，只须将-1张牌移到它的右边（第2堆）-2中；
2. 结果是-1变为0，-2变为-3，各堆牌张数变为0, -3, 7, -4。





分析

	9,	8,	17,	6
	-1,	-2,	7,	-4
1.	0,	-3,	7,	-4
2.	0	0	4	-4
3.	0	0	0	0

如果张数中本来就有为0，怎么办呢？如0， -1， -5， 6，还是从左算起（从右算起也完全一样），第1堆是0，无需移牌，余下与上相同；再比如-1， -2， 3， 10， -4， -6，从左算起，第1次移动的结果为0， -3， 3， 10， -4， -6；第2次移动的结果为0， 0， 0， 10， -4， -6，现在第3堆已经变为0了，可节省1步，余下继续。



分析

本题有**3个关键点**：

一是善于将每堆牌数减去平均数，简化了问题；

二是要过滤掉**0**（不是所有的**0**，如**-2, 3, 0, -1**中的**0**是不能过滤的）；

三是负数张牌也可以移动，这是关键中的关键。



代码

```
3 int main(){
4     int n,ave,step,a[101];
5     int i,j;
6     cin>>n;
7     ave=0,step=0;
8     for (i=1;i<=n;++i)
9         cin>>a[i],ave+=a[i]; //读入各堆牌张数, 求总张数ave
10    ave/=n; //求牌的平均张数ave
11    for (i=1;i<=n;++i) a[i]-=ave; //每堆牌的张数减去平均数
12    i=1;j=n;
13    while (a[i]==0&& i<n) ++i; //过滤左边的0
14    while (a[j]==0&& j>1) --j; //过滤右边的0
15    while (i<j)
16    {
17        a[i+1]+=a[i]; //将第i堆牌移到第i+1堆中去
18        a[i]=0; //第i堆牌移走后变为0
19        ++step; //移牌步数计数
20        ++i; //对下一堆牌进行循环操作
21        while (a[i]==0&& i<j) ++i; //过滤移牌过程中产生的0
22    }
23    cout<<step<<endl;
24 }
```

```

1  #include<bits/stdc++.h>
2  using namespace std;
3  int main(){
4      int n,ave,step,a[101];
5      int i,j;
6      cin>>n;
7      ave=0,step=0;
8      for (i=1;i<=n;++i)
9          cin>>a[i],ave+=a[i]; //读入各堆牌张数, 求总张数ave
10     ave/=n; //求牌的平均张数ave
11     for (i=1;i<=n;++i) a[i]-=ave; //每堆牌的张数减去平均数
12     _____1_____ ; //i,j初始化
13     _____2_____ ; //过滤左边的0
14     _____3_____ ; //过滤右边的0
15     while ( _____4_____ )
16     {
17         _____5_____ ; //将第i堆牌移到第i+1堆中去
18         _____6_____ ; //第i堆牌移走后变为0
19         _____7_____ ; //移牌步数计数
20         _____8_____ ; //对下一堆牌进行循环操作
21         _____9_____ ; //过滤移牌过程中产生的0
22     }
23     cout<<step<<endl;
24 }

```

2.6 [2872] 删数问题

输入一个高精度的正整数 N ，去掉其中任意 S 个数字后剩下的数字按原左右次序组成一个新的正整数。编程对给定的 N 和 S ，寻找一种方案使得剩下的数字组成的新数最小。

输出新的正整数。（ N 不超过240位）输入数据均不需判错。

【输入】

n

s

【输出】

最后剩下的最小数。

【样例输入】

175438

4

【样例输出】

13



分析

由于正整数 n 的有效数位为240位，所以很自然地采用字符串类型存贮 n 。

那么如何决定哪 s 位被删除呢？是不是最大的 s 个数字呢？

例如： $n=175438$

$s=4$

删数的过程如下：

$n=175438$	//删掉7
<u>15438</u>	//删掉5
<u>1438</u>	//删掉4
<u>138</u>	//删掉8
13	//解为13



分析

这样，删数问题就与**如何寻找递减区间首字符**问题对应起来。不过还要注意一个细节性的问题，就是可能会出现字符串串首有若干0的情况，甚至整个字符串都是0的情况。

如 900067800

贪心策略为：每一步总是选择一个使剩下的数最小的数字删去，即按高位到低位的顺序搜索，若各位数字递增，则删除最后一个数字；否则删除第一个递减区间的首字符，这样删一位便形成了一个新数字串。然后回到串首，按上述规则再删下一个数字。重复以上过程s次为止，剩下的数字串便是问题的解了。



代码

```
4  char str[260];
5  int main(){
6      int len,i,j,s;
7      cin>>str>>s;
8      len=strlen(str);
9      while(s--){//删除s个数字
10         for(i=0;i<=len-2;i++)
11             if(str[i]>str[i+1]){//找到满足字符
12                 for(j=i;j<=len-2;j++)
13                     str[j]=str[j+1];//删除
14                     break;
15             }
16         len--;
17     }
18     i=0;
19     while(i<=len-1&&str[i]=='0')i++;//处理前导0
20     if(i==len)printf("0");
21     else
22         for(j=i;j<=len-1;j++)
23             cout<<str[j];
24     return 0;
25 }
```

填空

```
4 char str[260];
5 int main(){
6     int len,i,j,s;
7     cin>>str>>s;
8     len=strlen(str);
9     while(____1____){//删除s个数字
10         for(i=0;i<=____2____;i++)
11             if(____3____){//找到满足字符
12                 for(j=i;j<=len-2;j++)
13                     ____4____//删除
14                 break;
15             }
16         len--;
17     }
18     i=0;
19     ____5____;//处理前导0
20     if(i==len)printf("0");
21     else
22         for(j=i;j<=len-1;j++)
23             cout<<str[j];
24     return 0;
25 }
```

2.7 [2961] 拦截导弹问题

某国为了防御敌国的导弹袭击，开发出一种导弹拦截系统，但是这种拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都**不能高于前一发的高度**。某天，雷达捕捉到敌国的导弹来袭，由于该系统还在试用阶段。所以一套系统有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度不大于30000的正整数）。计算要拦截所有导弹最小需要配备多少套这种导弹拦截系统。

【输入格式】

n颗依次飞来的高度（ $1 \leq n \leq 1000$ ）。

【输出格式】

要拦截所有导弹最小配备的系统数k。

【输入样例】

389 207 155 300 299 170 158 65

【输出样例】

2

分析

1989 1220 616 1619 1606 1981 118 123 456 789

1	1989	1220	616	118	
2	1619	1606	123		
3	1981	456			
4	789				
5					

按照题意，被一套系统拦截的所有导弹中，最后一枚导弹的高度最低。

若导弹*i*的高度高于所有系统的最低高度，则断定导弹*i*不能被这些系统所拦截，应增设一套系统来拦截导弹*i*

若导弹*i*低于某些系统的最低高度，那么导弹*i*均可被这些系统所拦截。



分析

设：

k 为当前配备的系统数；

$len[k]$ 为被第 k 套系统拦截的最后一枚导弹的高度，简称系统 k 的最低高度（ $1 \leq k \leq n$ ）。

我们首先设导弹1被系统1所拦截（ $k \leftarrow 1, len[k] \leftarrow$ 导弹1的高度）。然后依次分析导弹2， $\dots$ ，导弹 n 的高度。

贪心策略：选择其中最低高度最小（即导弹 i 的高度与系统最低高度最接近）的一套系统 $p(len[p]=\min\{len[j] \mid len[j]>\text{导弹}i\text{的高度}\})$ 。这样可使得一套系统拦截的导弹数尽可能增多。依次类推，直至分析了 n 枚导弹的高度为止。此时得出的 k 便为应配备的最少系统数。



框架

```
k=1;len[k]=导弹1的高度;
for (i=2;i<=n;++i)
{
    p=0;//一套系统p
    for (j=1;j<=k;j++)
        if (len[j]>=导弹i的高度) {
            if (p==0) p=j;
            else if (len[j]<len[p]) p=j;
        } //贪心
    if (p==0) {
        k++;len[k]=导弹i的高度;
        //增加一套新系统
    }
    else
        len[p]=导弹i的高度; //贪心,更新第p套系统的最低高度
}
输出应配备的最少系统数K。
```

1989 1220 616 1619 1606 1981 118 123 456 789

1	1989	1220	616	118	
2	1619	1606	123		
3	1981	456			
4	789				
5					



程序

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int s[1005]; //存储导敌国的导弹高度
4  int len[1005]; //第k套系统拦截的最后一枚导弹的高度
5  int main()
6  {
7      int n;
8      n=1;
9      memset(s,0,sizeof(s));
10     memset(len,0,sizeof(len));
11     while(cin>>s[n]){
12         n++; //计算导弹数量
13     }
14     int k=1; //当前配备的系统数
15     len[k]=s[1]; //把第一个导弹的高度作为第一套系统的初始高度
16     int p; //表示某一套系统
17     for(int i=2;i<n;i++)
18     { //遍历每一颗导弹
19         p=k;
20         for(int j=k+1;j<=n;j++)
21         {
22             if(s[j]>len[j-k])
23             {
24                 k=j-k;
25                 len[k]=s[j];
26             }
27         }
28     }
29     cout<<k<<endl;
30     return 0;
31 }
```

程序

```
17 for(int i=2;i<n;i++)
18 { //遍历每一颗导弹
19     p=0; //记录当前导弹j属于哪套系统
20     for(int j=1;j<=k;j++) //一共有k套系统
21     {
22         if(len[j]>=s[i]) //从k套系统选择
23         {
24             if(p==0)p=j; //p赋值为第一个满足条件的第j套系统
25             else if(len[j]<len[p]) p=j;
26             //贪心，选择最小系统中最小的值，
27             //然后后面会把这个比最小系统还小的值赋值给
28             //这个最小系统的最小值
29         }
30     }
31     if(p==0)
32     { //没有找到合适的系统，生成新的导弹系统
33         k++;
34         len[k]=s[i]; //把s[i]作为第k套系统的最小值
35     }
36     else
37         len[p]=s[i]; //到合适的系统，把s[i].....
38 }
39 cout<<k<<endl;
```

填空

```
17 for(int i=2;i<n;i++)
18 { //遍历每一颗导弹
19     p=0; //记录当前导弹j属于哪套系统
20     for(int j=1;j<=k;j++) //一共有k套系统
21     {
22         if(_____) //从k套系统选择
23         {
24             if(p==0) _____; //p赋值为第一个满足条件的第j套系统
25             else if(_____) p=j;
26             //贪心，选择最小系统中最小的值，
27             //然后后面会把这个比最小系统还小的值赋值给
28             //这个最小系统的最小值
29         }
30     }
31     if(p==0)
32     { //没有找到合适的系统，生成新的导弹系统
33         _____
34         _____ //把s[i]作为第k套系统的最小值
35     }
36     else
37         _____ //到合适的系统，把s[i].....
38 }
39 cout<<k<<endl;
```



2873 活动选择

学校在最近几天有 n 个活动，这些活动都需要使用学校的大礼堂，在同一时间，礼堂只能被一个活动使用。由于有些活动时间上有冲突，学校办公室人员只好让一些活动放弃使用礼堂而使用其他教室。

现在给出 n 个活动使用礼堂的起始时间 $begin_i$ 和结束时间 end_i ($begin_i < end_i$)，请你帮助办公室人员安排一些活动来使用礼堂，要求安排的活动尽量多。

【输入】 第一行一个整数 n ($n \leq 1000$)； 接下来的 n 行，每行两个整数，第一个 $begin_i$ ，第二个是 end_i ($begin_i < end_i \leq 32767$)

【输出】 输出最多能安排的活动个数。

【样例输入】

```
11
3 5
1 4
12 14
8 12
0 6
8 11
6 10
5 7
3 8
5 9
2 13
```

【样例输出】

```
4
```





2874】整数区间

请编程完成以下任务：

1. 从文件中读取闭区间的个数及它们的描述；
2. 找到一个含元素个数最少的集合, 使得对于每一个区间, 都至少有一个整数属于该集合, 输出该集合的元素个数。

【输入】

首行包括区间的数目 n , $1 \leq n \leq 10000$, 接下来的 n 行, 每行包括两个整数 a, b , 被一空格隔开, $0 \leq a \leq b \leq 10000$, 它们是某一个区间的开始值和结束值。

【输出】

第一行集合元素的个数, 对于每一个区间都至少有一个整数属于该区间, 且集合所包含元素数目最少。

【样例输入】

```
4
3 6
2 4
0 2
4 7
```

【样例输出】

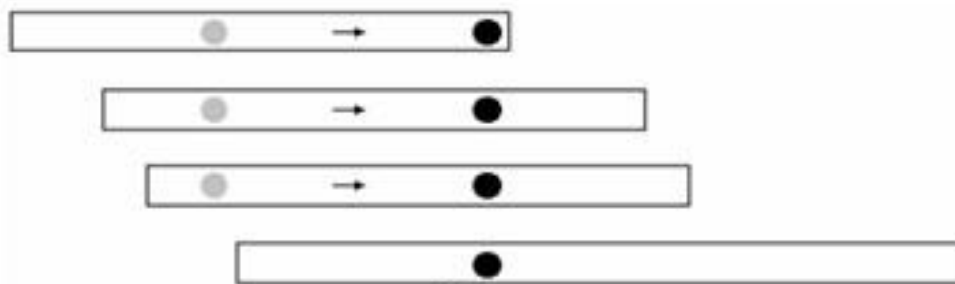
```
2
```

【算法分析】

算法模型：给 n 个闭区间 $[a_i, b_i]$ ，在数轴上选尽量少的点，使每个区间内至少有一个点。

算法：首先按 $b_1 \leq b_2 \leq \dots \leq b_n$ 排序。每次标记当前区间的右端点 x ，并右移当前区间指针，直到当前区间不包含 x ，再重复上述操作。

如下图，如果选灰色点，移动到黑色点更优。





[2007_p2]纪念品分组

元旦快到了，校学生会让乐乐负责新年晚会的纪念品发放工作。为使得参加晚会的同学所获得的纪念品价值相对均衡，他要把购来的纪念品根据价格进行分组，但每组最多只能包括两件纪念品，并且每组纪念品的价格之和不能超过一个给定的整数。为了保证在尽量短的时间内发完所有纪念品，乐乐希望分组的数目最少。

你的任务是写一个程序，找出所有分组方案中分组数最少的一种，输出最少的分组数目。

输入描述：输入文件包含 $n+2$ 行：

第1行包括一个整数 W ，为每组纪念品价格之和的上限

第2行为一个整数 n ，表示购来的纪念品的总件数 G

第3- $n+2$ 行每行包含一个正整数 P_i ($5 \leq P_i \leq w$) w 表示所对应纪念品的价格。

输出描述：

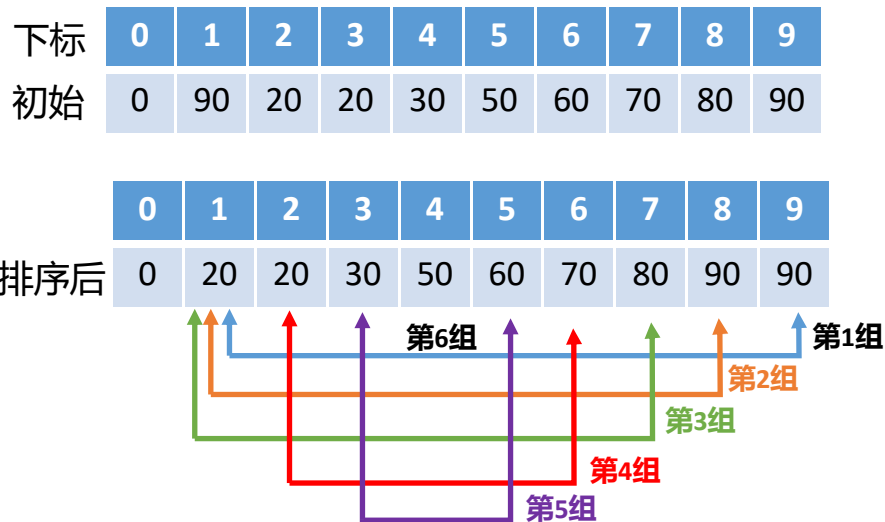
输出文件仅一行，包含一个整数， ep 最少的分组数目合

分析

样例输入：

100
9
90
20
20
30
50
60
70
80
90

根据问题描述，我们知道这里有一个条件，每组最多有两件纪念品，那么我们可以这样去思考，将价格进行排序，最大的加上最小的，如果没有超过上限，则他们两个一组；否则，说明超过了，最大的单独一组，如此往内推进。



样例输出：

6

贪心策略：先按n件纪念品的价格进行排序，将价格最小的与价值最大的分为一组，这样分组是最少的。 $i=1, j=n$, 条件 $i \leq j$

代码实现

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int main(){
4      int w,n,a[100000];
5      cin>>w>>n;
6      for(int i=1;i<=n;i++){
7          cin>>a[i];
8      }
9      sort(a+1,a+n+1);
10     int i=1,j=n,ans=0;
11     while(i<=j){
12         if(a[i]+a[j]<=w){
13             ans++;
14             i++;
15             j--;
16         }
17         if(a[i]+a[j]>w){
18             ans++;
19             j--;
20         }
21     }
22     cout<<ans;
23     return 0;
```

```
10
11
12
13
14
15
16
17
18
19
20
```

```
while(i<=j){
    if(a[i]+a[j]<=w){
        ans++;
        i++;
        j--;
    }
    if(a[i]+a[j]>w){
        ans++;
        j--;
    }
}
```

[1206] 母舰

题目描述：

在小A的星际大战游戏中，一艘强力的母舰往往决定了一场战争的胜负。一艘母舰的攻击力是普通的MA(Mobile Armor)无法比较的。对于一艘母舰而言，它是由若干个攻击系统和若干个防御系统组成的。两艘母舰对决时，一艘母舰会选择用不同的攻击系统去攻击对面母舰的防御系统。**当这个攻击系统的攻击力大于防御系统的防御力时，那个防御系统会被破坏掉。**当一艘母舰的防御系统全部被破坏掉之后，所有的攻击都会攻击到敌方母舰本身上去造成伤害。这样说，一艘母舰对对面的伤害在一定程度上是取决于选择的攻击对象的。在瞬息万变的战场中，选择一个最优的攻击对象是非常重要的。所以需要写出一个战斗系统出来，判断出你的母舰最多能对对手造成多少伤害并加以实现。

1 母舰

输入描述：输入第一行两个整数M和N，表示对方母舰的防御系统数量和你的母舰的攻击系统数量。接着M行每行一个整数每一个表示对方防御系统的防御力是多少。接着N行每行一个整数每一个表示己方攻击系统的攻击力是多少。

输出描述：输出仅有一行，表示可以造成的最大伤害。

样例输入：

3 5
1000
2000
1200
2100
1200
2000
1000
1000

样例输出：

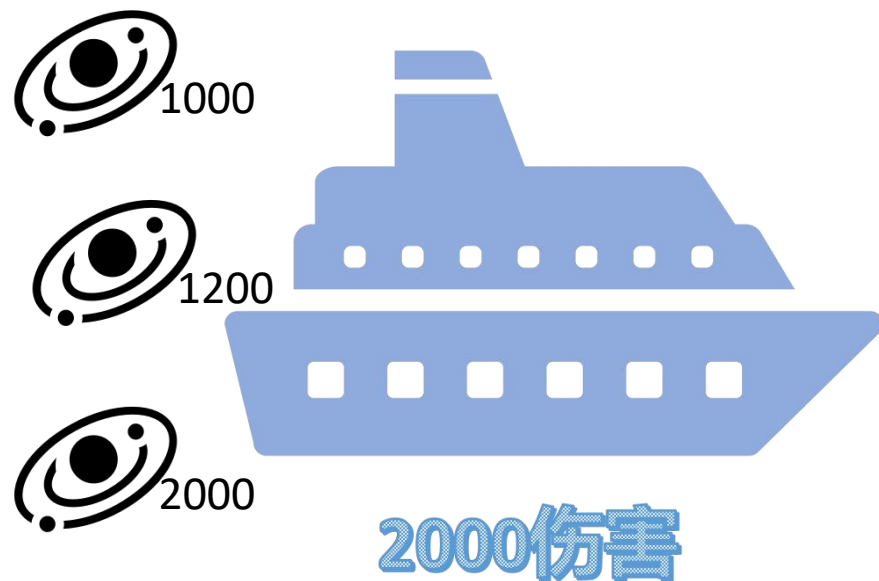
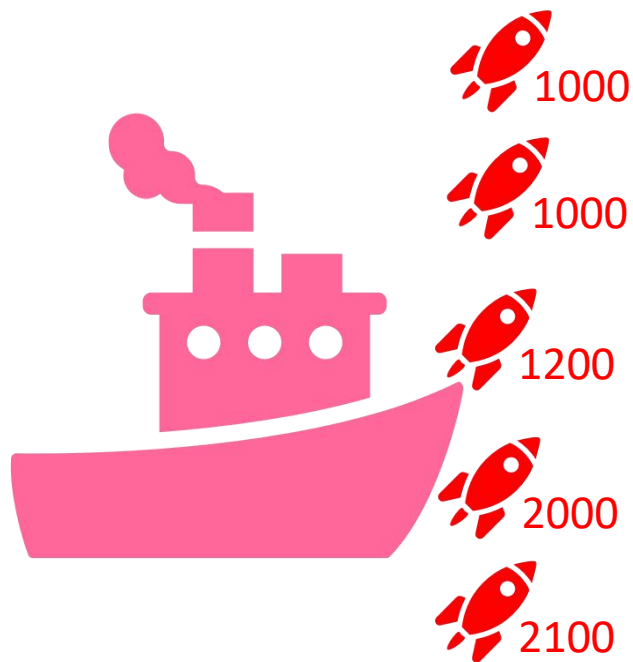
2000



题目分析

它是由若干个攻击系统和若干个防御系统组成的。

两艘母舰对决时，一艘母舰会选择用不同的攻击系统去攻击对面母舰的防御系统。当这个攻击系统的攻击力大于防御系统的防御力时，那个防御系统会被破坏掉。当一艘母舰的防御系统全部被破坏掉之后，所有的攻击都会攻击到敌方母舰本身上去造成伤害。



分析

目标是让伤害最大化

→ 每个攻击要选最合适的防御系统

→ 刚好能消灭防御系统（攻击=防御+k） $k \geq 1$, k 越小，最有选
择

→ 肯定要对攻击与防御排序

→ 是从大到小排

or 从小到大排



分析

方案1---先消灭大的防御系统

攻击从大到小排序

防御从大到小排序

第1次

从开始找---攻击中找出最合适的攻击系统

第2次

从开始找--从攻击中找出最合适的攻击系统

.....

时间分析

攻击最大有10000,防御最大有10000

$O(10000 \times 10000)$ 时间可能超限

分析

方案2---先消灭小的防御系统

攻击从小到大排序

防御从小到大排序

第1次

从开始找---攻击中找出最合适的攻击系统

第2次

从上次位置接着找 从攻击中找出最合适的攻击系统

.....

时间分析

攻击最大有10000,防御最大有10000

$O(10000)$ 时间肯定不会超限

代码实现---保存攻击与防御

```
4 int main(){
5     int attack[N];
6     int defend[N]; // #define N 100005
7     int a_cnt, d_cnt;
8     cin >> d_cnt >> a_cnt; // d 防御力 a 攻击力
9     for(int i=1; i<=d_cnt; i++) cin >> defend[i]; //
10    for(int i=1; i<=a_cnt; i++) cin >> attack[i]; //
11    // 注意顺序不要错
12    sort(attack+1, attack+a_cnt+1);
13    sort(defend+1, defend+d_cnt+1);
14    bool flag;
15    int last=0; // 记录当前打掉防御系统的攻击系统位置
16    for(int i=1; i<=d_cnt; i++){
34
35    long long attack_sum=0;
36    for(int i=1; i<=a_cnt; i++){
37        attack_sum+=attack[i];
38    }
39    cout << attack_sum;
40    return 0;
```



代码实现---排序攻击与防御

```
15      int last=0; // 记录当前打掉防御系统的攻击系统位置
16      for(int i=1;i<=d_cnt;i++){
17          flag=false;
18          for(int j=last+1;j<=a_cnt;j++){
19              last=j; // 不能放在if语句里面
20              // 放在if里面last不能每次都更新, 会超时
21              if(attack[j]>defend[i]){
22                  attack[j]=0;
23                  defend[i]=0;
24                  flag=true;
25                  break;
26              }
27          }
28          if (!flag)
29          { // 一旦为false, 表明这个防御没打掉, 失败
30              cout << 0 << endl;
31              return 0;
32          }
33      }
```



代码实现---消耗防御导弹

```
int fangs[10000 + 1];
int gongs[10000 + 1];
int main()
{
    //消耗防御导弹
    bool flag; //记录是否找到
    int last = 0; //记录上次查找攻击位置
    for (int i = 1; i <= fang_cnt; i++) //依次消耗防御
    {
        //找出最合适的攻击

    }
    //累计未发射攻击值
    return 0;
}
```



代码实现---找出最合适的攻击

```
//找出最合适的攻击
flag = false; //记录是否找到
for (int j = last+1; j <= gong_cnt; j++) //从上个位置开始找
{
    last = j; //更新下次查找的起始位置
    if (gongs[j] > fangs[i]) //摧毁当前防御
    {
        gongs[j] = 0;
        fangs[i] = 0;
        flag = true;
        break;
    }
}
if (!flag) //没有找到合适的攻击，无法摧毁所有防御
{
    cout << 0 << endl;
    return 0;
}
```



代码实现---累计未发射攻击值

```
//累计未发射攻击值
long long gong_sum = 0;
for (int i = 1; i <= gong_cnt; i++)
{
    gong_sum += gongs[i];
}
cout << gong_sum << endl;
```



代码实现

```
static int fangs[SIZE + 1];
static int gongs[SIZE + 1];
int main()
{
    int gong_cnt, fang_cnt; //攻击数量 与防御数量
    cin >> fang_cnt >> gong_cnt;
    for (int i = 1; i <= fang_cnt; i++)
    {
        cin >> fangs[i];
    }
    for (int i = 1; i <= gong_cnt; i++)
    {
        cin >> gongs[i];
    }
    sort(fangs + 1, fangs + 1 + fang_cnt);
    sort(gongs + 1, gongs + 1 + gong_cnt);
```

```
    bool flag;
    int last = 0;
    for (int i = 1; i <= fang_cnt; i++)
    {
        flag = false;
        for (int j = last + 1; j <= gong_cnt; j++)
        {
            last = j;
            if (gongs[j] > fangs[i])
            {
                gongs[j] = 0;
                fangs[i] = 0;
                flag = true;
                break;
            }
        }
    }
    long long gong_sum = 0;
    for (int i = 1; i <= gong_cnt; i++)
    {
        gong_sum += gongs[i];
    }
    cout << gong_sum << endl;
    return 0;
}
```

代码补充

```
int fangs[10000 + 1];
int gongs[10000 + 1];
int main()
{
    int gong_cnt, fang_cnt; //攻击数量 与防御数量
    cin >> fang_cnt >> gong_cnt;
    //输入攻击防御

    //排序攻击与防御

    //消耗防御导弹
    bool flag; //记录是否找到
    int last = 0; //记录上次查找攻击位置
    for (int i = 1; i <= fang_cnt; i++) //依次消耗防御
    {
        //找出最合适的攻击

    }
    //累计未发射攻击值

    return 0;
}
```

[1164] 田忌赛马

这是中国历史上的一个著名故事。

大约 2300 年前，田忌是齐国的一位将军，他喜欢与国王等人赛马。

田忌和国王都有三匹不同等级的马——下马、中马、上马。规则是一场比赛要进行三个回合，每匹马进行一回合的较量，单回合的获胜者可以从失败者那里得到 200 银元。

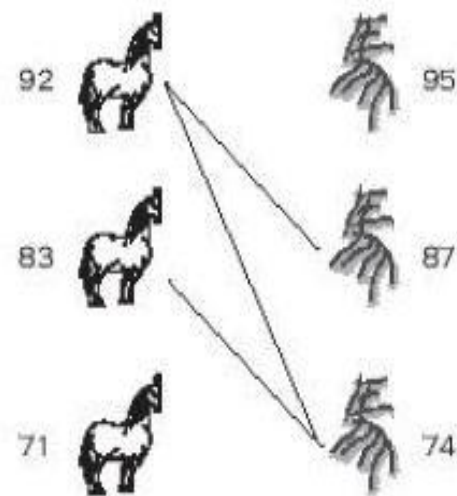
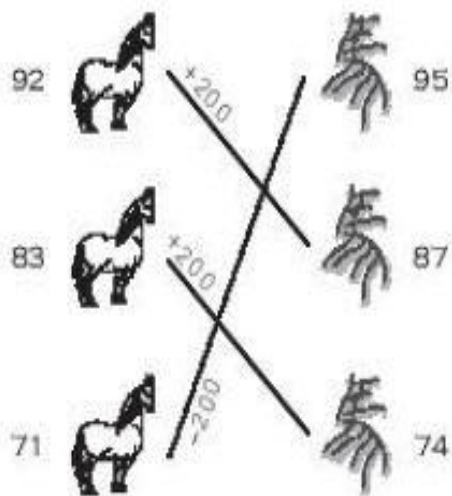
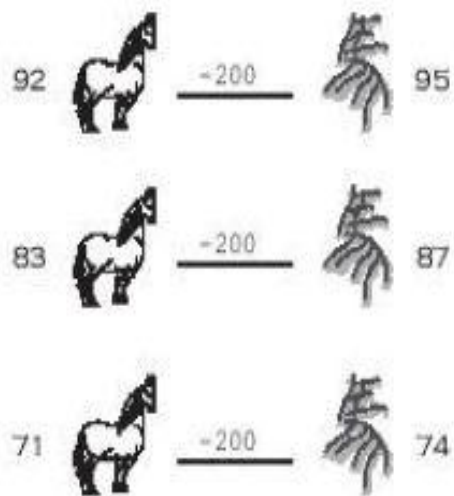
比赛的时候，国王总是用自己的上马对田忌的上马，中马对中马，下马对下马。

由于国王每个等级的马都比田忌的马强一些，所以比赛了几次，田忌都失败了，每次国王都能从田忌那里拿走 600 银元。田忌对此深感不满，直到他遇到了著名的军事家孙臆，利用孙臆给他出的主意，田忌终于在与国王的赛马中取得了胜利，拿走了国王的 200 银元。



[1164] 田忌赛马

其实胜利的方法非常简单，他用下马对战国王的上马，上马对战国王的中马，中马对战国王的下马，这样虽然第一回合输了，但是后两回合都胜了，最终两胜一负，取得了比赛胜利。



[1164] 田忌赛马

话说齐王和田忌又要赛马了，他们各派出N匹马，每场比赛，输的一方将要给赢的一方200两黄金，如果是平局的话，双方都不必拿出钱。现在每匹马的速度值是固定而且已知的，而齐王出马也不管田忌的出马顺序。请问田忌该如何安排自己的马去对抗齐王的马，才能赢取最多的钱？

输入格式

第一行为一个正整数 n ($n \leq 1000$)，表示双方马的数量。第二行有 N 个整数表示田忌的马的速度。第三行的 N 个整数为齐王的马的速度。

输出格式

仅有一行，为田忌赛马可能赢得的最多的钱，结果有可能为负。



分析

对田忌和国王的马都从小到大排序

1. 如果田忌最快的马 $>$ 国王最快的马，则让它们比赛(因为这种情况下，田忌最快的马对国王所有的马都能赢，肯定是耗掉国王最快的马对田忌剩余的马最有优势)
2. 如果田忌最快的马 $<$ 国王最快的马，则让田忌最慢马与国王最快的马比赛(因为这种情况下，田忌所有的马对国王最快的这匹马都会输，反正要输一场，用田忌最慢的马去输，肯定对田忌剩余的马最有优势)
3. 如果田忌最快的马 $=$ 国王最快的马，
则 比较 最慢的马。

分析

3. 如果田忌最快的马 == 国王最快的马, 则 比较 最慢的马:
- a. 如果田忌的更慢, 肯定还是用最慢的马 去耗国王最快马
 - b. 如果田忌的更快, 说明田忌所有的马都能赢国王最慢的马, 让田忌最慢的马去赢, 保留更快的马, 对之后比赛更有优势。
 - c. 如果一样快, 让 最慢去耗 国王最快马

排序 $O(n\log n)$, 贪心比较过程 $O(n)$
时间复杂度 $O(n\log n)$



```
1  #include<stdio.h>
2  #include<algorithm>
3  using namespace std;
4  int tj[2010]; //田忌的马的速度
5  int king[2010]; //齐王的马的速度
6  int ans;
7  int main()
8  {
9      int n,m;
10     scanf("%d",&n);
11     for(int i=1;i<=n;++i)
12         scanf("%d",&tj[i]);
13     for(int i=1;i<=n;++i)
14         scanf("%d",&king[i]);
15
16     sort(tj+1,tj+1+n); //从小到大排序
17     sort(king+1,king+1+n);
```

```

19  int ra=n,rb=n; //ra 田忌最强的马, rb齐王最强的,
20  int la=1,lb=1; //la田忌最弱的, lb齐王最弱的
21  while(lb<=rb)
22  { //扫描没一匹马
23
24      if(tj[ra]>king[rb])
25      { //1如果田忌最强的赢过齐王最强的, 那就直接比赛
26          rb--,ra--;
27          ans++;
28      }
29      else if(tj[ra]<king[rb])
30      { //如果田忌最强的弱于齐王最强的, 那就用最弱的对掉齐王最强的
31          rb--,ans--;
32          la++;
33      }

```

```

34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
}

else if(tj[ra]==king[rb])
{ //如果田忌最强的和齐王最强的一样
    if(tj[la]>king[lb]) //如果田忌最弱的强国齐王最弱的，直接比赛
    {
        ans++;
        la++, lb++;
    }
    else
    {
        if(tj[la]<king[rb]) ans--; //否则，田忌用最弱的对掉齐王最强的
        la++, rb--;
    }
}
}

printf("%d\n", ans*200);

return 0;
}

```

今天的课程结束啦.....



下课了...
同学们再见!

