greedy1

#! <https://zhuanlan.zhihu.com/p/547386904>

* 贪心Part1
  + 区间问题
    - 905. 区间选点
      * 问题定义
      * 逻辑推导
      * 代码实现
    - 908. 最大不相交区间数量
      * 问题定义
      * 逻辑推导
    - AcWing 906. 区间分组
      * 问题定义
      * 逻辑推导
      * 代码实现
    - 907. 区间覆盖
      * 问题定义
      * 逻辑推导
      * 代码实现
  + Huffman树
    - AcWing 148. 合并果子
      * 逻辑推导
      * 代码实现

# 贪心Part1

贪心预计讲两节，所有算法问题中，贪心和DP是最难的，甚至贪心比DP还要难。一个贪心算法的正确性的证明通常是很难的。贪心也没有一个常规的套路，更没有代码模板。DP虽然没有代码模板，但它至少有常用的套路。

贪心这一章的几道例题，代码都非常短，讲课主要是以证明为主。

本章涉及五种贪心问题:

1. 区间问题
2. Huffman树
3. 排序不等书
4. 绝对值不等式
5. 推公式

## 区间问题

### [905. 区间选点](https://www.acwing.com/problem/content/907/)

#### 问题定义

给定 N 个闭区间 [ai,bi]，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。

#### 逻辑推导

1. 将每个区间按右端点从小到大排序.
2. 从前往后依次枚举每个区间,如果当前区间已经包含点直接pass，否则选择当前区间的右端点.

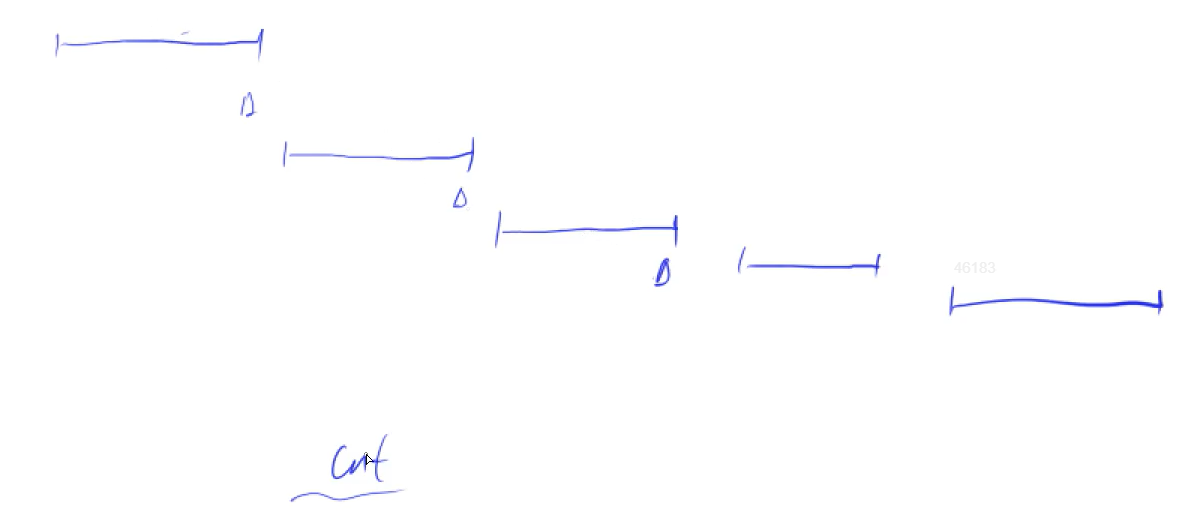
下面证明一下上面贪心策略的正确性：

设按照上面的策略，选出的点数为cnt，问题的答案为ans。那么我们就是要证明cnt = ans。

在数学上有一个思路，若要证A = B，则可以先证A >= B，再证 A <= B，以此得出 A = B。即，用不等式来推导出等式。

首先，按照上面的策略选点完毕后，能保证每个区间都至少有一个点。因为我们会依次枚举每个区间，若当前区间包含点，就跳过，若不包含，就选一个点。所以最终每个区间都至少有一个点。也就是说，通过这个策略得到的，一定是一个合法的选点方案（每个区间内都至少包含一个点即为合法）。而问题的答案，就是全部合法方案中的最小值。所以我们能得出：ans <= cnt

接着，我们换一种角度，按照上面的策略，什么时候会增加一个点呢？那就是从前往后枚举每个区间时，遇到了当前区间没有点这个分支条件时，才会实际上增加一个点。那我们通过上面的策略最终选出了cnt个点，也就是有cnt个区间走到了当前区间没有点这个分支上。而由于区间是按照右端点从小到大排序的，那么我们能从全部的区间中，抽取出cnt个区间，这cnt个区间从左到右依次排列，且两两不相交。



由于合法的方案，需要保证每个区间内都至少有一个点，所以，所有的合法方案，都必须要覆盖掉这cnt个两两不相交的区间，而覆盖掉这cnt个区间，至少需要cnt个点。所以，所有的合法方案的点数，都一定要大于等于cnt。而问题的解也是合法方案中的一种，所以它也要满足大于等于cnt。于是就有了：ans >= cnt

根据ans <= cnt 和 ans >= cnt，我们就能得出 ans = cnt，该策略的正确性证毕。

#### 代码实现

#include <iostream>  
#include <algorithm>  
using namespace std;  
  
const int N = 1e5 + 10;  
  
int n;  
  
// 定义一个表示区间的结构体  
struct Range  
{  
 int l, r;  
 // 重载运算符, 按照区间右端点排序  
 bool operator< (const Range &w) {  
 return r < w.r;  
 }  
} range[N];  
  
int main() {  
 scanf("%d", &n);  
 for(int i = 0; i <n; i++) {  
 int l, r;  
 scanf("%d%d", &l, &r);  
 range[i] = {l, r};  
 }  
  
 // 按照右端点对区间进行排序  
 sort(range, range + n);  
  
 // 第一个区间一定会选  
 int cnt = 1, ed = range[0].r;  
 // 从第二个区间开始枚举  
 for(int i = 1; i < n; i++) {  
 if(range[i].l > ed) {  
 cnt++;  
 ed = range[i].r;  
 }  
 }  
 printf("%d\n", cnt);  
 return 0;  
}  
  
复制

其实也可以按照左端点排序，思路和合并区间有一些些类似，但维护的信息有些许不同

#include <iostream>  
#include <algorithm>  
using namespace std;  
typedef pair<int, int> PII;  
const int N = 1e5 + 10;  
const int INF = 0x3f3f3f3f;  
PII segments[N];  
  
int main() {  
 int n, a, b;  
 cin >> n;  
 for (int i = 0; i < n; i++) {  
 cin >> a >> b;  
 segments[i] = {a, b};  
 }  
  
 sort(segments, segments + n);  
 int r = INF, cnt = 1;  
 for (int i = 0; i < n; i++) {  
 PII p = segments[i];  
 if (p.first <= r) {  
 r = min(r, p.second);  
 } else {  
 r = p.second;  
 cnt++;  
 }  
 }  
 printf("%d\n", cnt);  
 return 0;  
}  
  
复制

区间选点问题，有个对应的实际问题：[112. 雷达设备](https://www.acwing.com/problem/content/114/)

### [908. 最大不相交区间数量](https://www.acwing.com/problem/content/910/)

#### 问题定义

给定 N 个闭区间 [ai,bi]，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。

#### 逻辑推导

这个题目可以对应一些生活的实际场景。比如在某一天内，我们有很多个活动可以去参加，每个活动有一个开始时间和结束时间。问如何选择，能够使我们当天参加的活动数量最大，并且活动之间没有时间冲突。

这个问题的做法，和上一个问题一样，代码也一样。

重点还是来证明一下这种贪心策略的正确性。同样的，按照上面的策略，我们能选出cnt个区间，这些区间之间两两不相交。

那么这是一种合法的方案（选出的所有区间之间不能相交，即为合法）。而问题的答案ans，是所有合法方案中，区间数量最大的那种方案。所以ans >= cnt。

对于ans <= cnt的证明，可以考虑使用反证法。先假设ans > cnt，看有没有什么矛盾。

假设ans > cnt，则说明可以选择出ans个互不相交的区间，那么要覆盖掉全部的区间，则至少需要ans个点。而根据我们上面的策略，能够得知，只需要cnt个点，就能够把全部的区间覆盖完毕。

也就是说，如果存在ans > cnt，则至少需要ans（大于cnt）个点，才能覆盖掉全部的区间，这与只需要cnt个点就能覆盖掉全部的区间矛盾了。所以ans > cnt不成立，即ans <= cnt成立。

正确性得证。

可见，贪心的题目，更多是逻辑上的推理和证明，所以它非常难。

### [AcWing 906. 区间分组](https://www.acwing.com/problem/content/908/)

#### 问题定义

给定 N 个闭区间 [ai,bi]，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。

#### 逻辑推导

1. 将所有区间按照左端点从小到大排序
2. 从前往后处理每个区间

判断能否将当前区间放到某个现有的组当中（判断现有组中的最后一个区间的右端点，是否小于当前区间的左端点）

如果不存在这样的组，就意味着当前区间，与所有的组都有交集，就必须要开一个新的组，把这个区间放进去  
如果存在这样的组，就将当前区间放到这个组中，并更新当前组的右端点(Max\_r)  
正确性证明：（ans表示最终答案，cnt表示按照上述算法得到的分组数量），仍然从两方面来证明  
ans <= cnt  
ans >= cnt  
首先，按照上面的算法步骤，得到的方案一定是一个合法方案，因为任意两个组之间都不会有交集，然后ans是所有合法方案中的最小值，故有ans <= cnt。

然后，观察一下最后一个新开的组的情况，什么情况需要新开一个组呢？当某个区间和现有的所有分组都有交集时，则需要新开一个组。当新开第cnt个组时，则当前这个区间和其余的cnt-1个组都有交集，而区间的左端点是从小到大排列的。设当前这个区间的左端点为L，则全部的cnt个分组，都有一个公共的点L。也就是说，至少有cnt个区间，在L这个点上是相交的。故要把这些区间分开来，则至少要分cnt个组。即，ans >= cnt。由此得ans = cnt，得证。

#### 代码实现

代码题解如下，其中判断能够将某个区间放到现有的某个组中，如果所有组右端点的最小值都大于等于L[i]，说明不存在这样的组需要新开一个，动态维护最小值可以用小根堆来进行优化，堆中存放的是每个分组中的最右边的端点。

#include <iostream>  
#include <algorithm>  
#include <vector>  
#include <queue>  
  
using namespace std;  
  
const int N = 1e5 + 10;  
  
int n;  
  
struct Range {  
 int l, r;  
 bool operator < (const Range &w) {  
 return l < w.l;  
 }  
} range[N];  
  
int main() {  
 scanf("%d", &n);  
 for(int i = 0; i < n; i++) {  
 int l, r;  
 scanf("%d%d", &l, &r);  
 range[i] = {l, r};  
 }  
  
 sort(range, range + n);  
  
 // 用小根堆来维护每个分组的最右端点  
 priority\_queue<int, vector<int>, greater<int> > heap;  
  
 for(int i = 0; i < n; i++) {  
 auto r = range[i];  
 // 若堆为空, 或堆顶(所有组的右端点的最小值)大于等于当前区间的左端点, 则需要新开一个组  
 if(heap.empty() || heap.top() >= r.l) heap.push(r.r);  
 else {  
 // 否则, 可以插入到堆顶这个组, 则更新堆顶这个组的右端点  
 heap.pop();  
 heap.push(r.r);  
 }  
 }  
  
 printf("%d\n", heap.size());  
  
 return 0;  
}  
复制

本题实际问题:[111. 畜栏预定](https://www.acwing.com/problem/content/113/)

### [907. 区间覆盖](https://www.acwing.com/problem/content/909/)

#### 问题定义

给定 N 个闭区间 [ai,bi] 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。

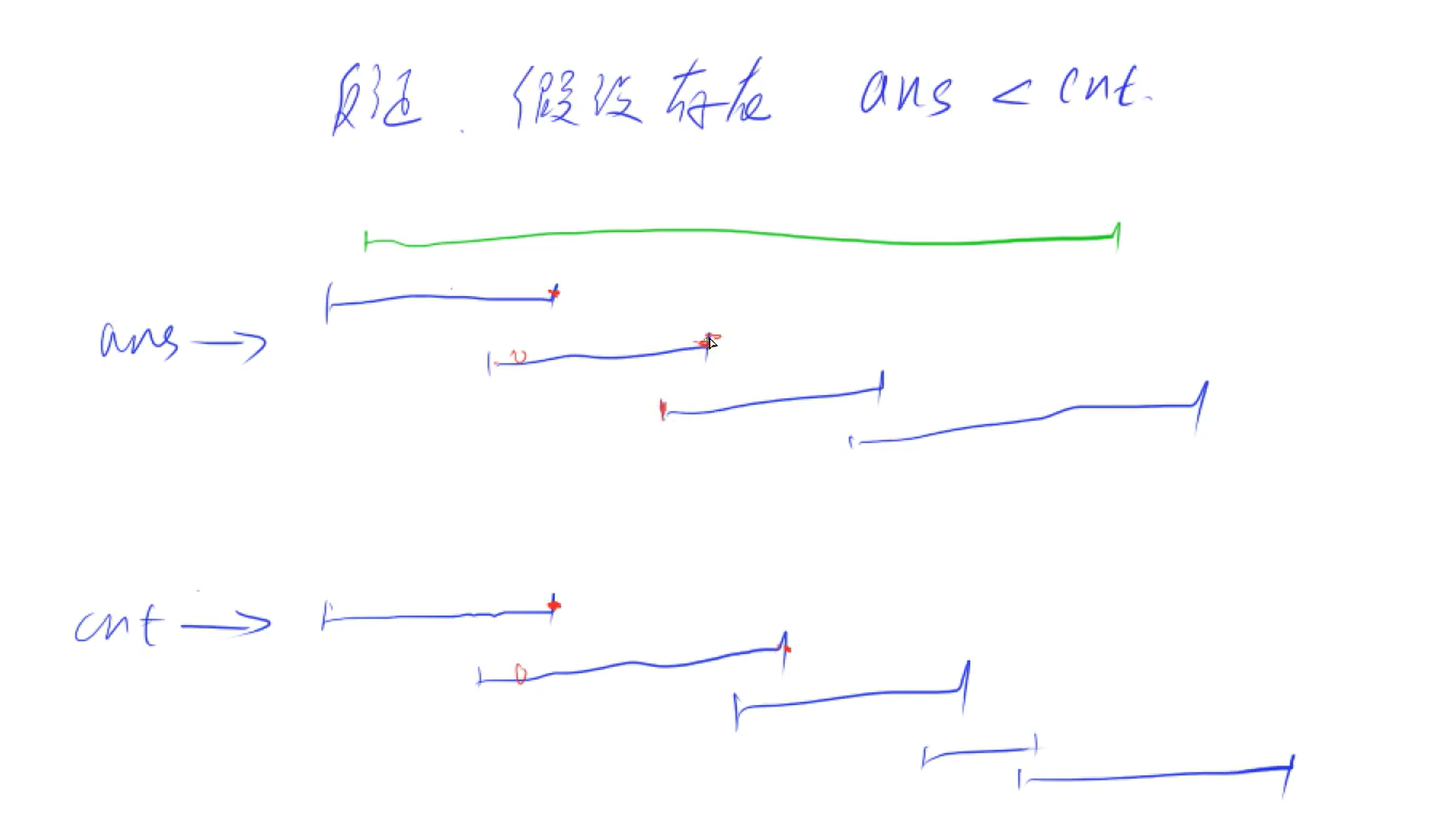
输出最少区间数，如果无法完全覆盖则输出 −1。

#### 逻辑推导

设线段的左端点为start，右端点为end

将所有区间按照左端点从小到大排序  
从前往后依次枚举每个区间，在所有能覆盖start的区间中，选择一个右端点最大的区间，随后，将start更新为选中区间的右端点  
当start >= end，结束  
区间覆盖  
正确性证明，同样从两个方面

ans <= cnt  
ans >= cnt  
首先，（在有解的前提下）上面的策略可以找出一个可行的合法方案，将这个方案需要用到的区间数量记为cnt，而ans表示的是所有合法方案中的最少区间数量，所以有ans <= cnt。

接着，假设ans < cnt，则在ans选择区间时，一定从某个区间开始，和cnt的选择不一样。而cnt每次是选择能覆盖当前start，且右端点最大的区间，则可以将ans该次的选择，用cnt的选择替换掉，且不会增加所选区间的个数。依次往后推，可以得出ans一定是等于cnt的。（其实可以直接证出ans = cnt，而不需要前面的ans <= cnt的证明了）  


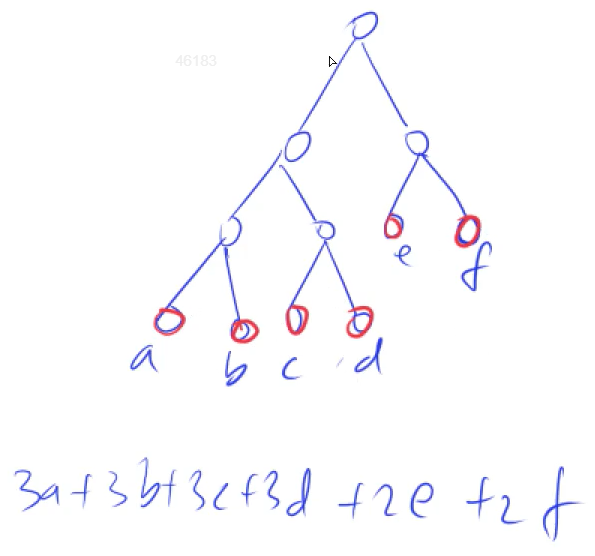
#### 代码实现

#include <iostream>  
#include <algorithm>  
using namespace std;  
  
const int N = 1e5 + 10;  
  
int n, s, t;  
  
struct Range  
{  
 int l, r;  
 bool operator < (const Range &w) {  
 return l < w.l;  
 }  
} range[N];  
  
int main() {  
 scanf("%d%d%d", &s, &t, &n);  
 for(int i = 0; i < n; i++) {  
 int l, r;  
 scanf("%d%d", &l, &r);  
 range[i] = {l, r};  
 }  
  
 sort(range, range + n);  
  
 int res = 0;  
  
 bool success = false;  
  
 for(int i = 0; i < n; i++) {  
 int j = i, r = -2e9;//双指针算法遍历所有左端点在start左边，右端点最大值是多少  
 while(j < n && range[j].l <= s) {  
 r = max(r, range[j].r);  
 j++;  
 }   
 // 当跳出循环后, j是第一个不满足上述条件的区间  
 // 由于区间按照左端点从小到大排序  
 // 则j是第一个左端点大于s的区间  
   
 // 右端点最大, 都没有覆盖掉s, 则无解  
 if(r < s) {  
 res = -1;  
 break;  
 }  
  
 res++;  
  
 // 已经覆盖完毕  
 if(r >= t) {  
 success = true;  
 break;  
 }  
  
 // 更新s  
 s = r;  
 // 更新下一轮的起点  
 // 由于j之前的所有区间的右端点都小于r  
 // 而下一轮要覆盖掉r, 所以枚举的区间要从j开始  
 i = j - 1;  
 }  
  
 if(!success) res = -1;  
  
 printf("%d\n", res);  
  
 return 0;  
}  
  
复制

## Huffman树

### [AcWing 148. 合并果子](https://www.acwing.com/problem/content/150/)

#### 逻辑推导

果子的合并过程，可以用一棵树来表示  
  
所有的叶子节点，是每一堆果子的重量，而每个非叶子节点，就表示了一次合并操作消耗的体力。则消耗的总体力，就是全部非叶子节点的总和。比如，对于a这个节点，我们可以看到，其需要参与3次合并，a会被累加3次，被累加的次数也是这个节点到根节点的路径长度。

所以，要使得消耗的总的体力最小，我们需要使权重大的节点（消耗体力大的节点），到根节点的路径尽可能的短（使得这个节点被计算的次数尽可能少）。

这就跟哈夫曼编码一个道理。对于出现频率最高的字符，编码时要使得其位数越少（即到根节点的路径越短），而出现频率低的字符，其到根节点的路径可以稍长，这样就使得采用此种编码进行信息传输时，占用的空间最少。

这道题的思路，就是每次合并时，挑当前重量最小的两堆，进行合并。即，每次都用贪心的策略进行选择，每次都选择一个局部最优解，最终能找到全局的最优解。

#### 代码实现

代码可以用一个小根堆来存储所有的果子重量，每次合并最小的2个，直到堆中只剩一个元素，说明合并完成。

#include <iostream>  
#include <algorithm>  
#include <queue>  
  
using namespace std;  
  
int main() {  
 int n;  
 scanf("%d", &n);  
 // 小根堆  
 priority\_queue<int, vector<int>, greater<int>> heap;  
  
 // 将全部的果子插入到堆  
 while(n--) {  
 int x;  
 scanf("%d", &x);  
 heap.push(x);  
 }  
  
 // 消耗的总体力  
 int res = 0;  
 // 当堆的大小大于1时, 进行合并  
 while(heap.size() > 1) {  
 // 取出最小的两个  
 int a = heap.top();  
 heap.pop();  
 int b = heap.top();  
 heap.pop();  
  
 res += a + b; // 记录此次合并消耗的体力  
 heap.push(a + b); // 将合并后的作为一堆新的果子, 插入到堆  
 }  
  
 printf("%d\n", res);  
  
 return 0;  
}  
  
复制