前缀和 差分 简单贪心

By — Nil

I. 前缀和(prefix-sum)

前缀和： 可以理解为「数列的前N项和」，是一种重要的预处理方式。

一维前缀和：对于一个长度为n的序列$\{a_i\}_{i=1}^{n}$，如果需要多次查询数组里$[l,r]$位置中序列数字的和（即$\sum_{i=l}^{r} a_i$），那么就可以考虑使用前缀和。
我们在高中学习数列的时候，都学过数列的前N项和，即

$$S_k = \sum_{i=1}^{k} a_i$$

这个时候我们知道：

$$S_0 = 0, S_i = S_{i-1} + a_i, i \in [1,n]$$

于是我们可以得到另一个式子:

$$\sum_{i=l}^{r} a_i = S_r - S_{l-1}$$

然后我们来考虑一下时间复杂度，看看前缀和究竟带给我们什么了～
让我们先来看看下面这个问题：

对于一个长度为N的数列A，我们有Q次询问
对于每一次询问，我们给出一个区间$[L,R]$，你需要给出

$$\sum_{i=L}^{R} a_i$$

solution 1: *Step1*: 读入数列 A $\Longrightarrow$ *Step2*: Q次循环，每次读入L,R
$\Longrightarrow$ *Step3:* $R-L$次循环，求和并输出
很显然的是，读入是$O(N)$的，查询的外层循环Q次，内层循环$R-L$次，
那么最坏情况就是每一次都枚举$1\sim N$所有的数字，时间复杂度是
$O(N+QN)$的.
solution 2: *Step1*: 读入数列 A $\Longrightarrow$ *Step2*: N次循环预处理出$S_{i\in[1,N]}$前缀和
$\Longrightarrow$ *Step3:* Q次循环，每次读入L,R $\Longrightarrow$ *Step4* : 直接输出S_R-S_{L-1}
读入是$O(N)$的，预处理循环Q次，查询的循环Q次，对于每一次查询，直接$O(1)$输出，所以复杂度是$O(N+N+Q)$的.

让我们来看看代码吧

```C++
int main() {
  int n,q,i; scanf("%d%d",&n,&q);
  int a[n+1]; for(i=1;i<=n;i++) scanf("%d",a+i);
  for(;q--;){
    int l,r,ans=0; scanf("%d%d",&l,&r);
    for(i=l;i<=r;i++) ans+=a[i];
    printf("%d",ans);
  }
}
```

```C++
 int main() {
   int n,q,i; scanf("%d%d",&n,&q);
   int a[n+1]; for(i=1;i<=n;i++) scanf("%d",a+i);
   int pre[n+1]; pre[0]=0;
   for(i=1;i<=n;i++) pre[i]=pre[i-1]+a[i];
   for(;q--;){
     int l,r; scanf("%d%d",&l,&r);
     printf("%d",pre[r]-pre[l-1]);
   }
 }
```

二维前缀和：其实和一维前缀和类似，但有一定区别， 实际上，对于一个大小为$M \times N$的二维数组A，其前缀和的式子应该是

$$S_{i,j} = \sum_{k=1}^{i}\sum_{h=1}^{j} A_{k,h}$$

那么我们要考虑其递推公式对吧，就应该是这样的：

$$S_{i,j} = A_{i,j} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1}$$

就如下图所示，

全部 = 蓝色 + 红绿 + 红黄 - 红

同理可以推导出 $(i,j) \Rightarrow (k,h)$的矩阵和为：

$$S_{k,h} - S_{i-1,h} - S_{k,j-1} + S_{i-1,j-1}$$

那么让我们来做一道题吧：

洛谷 P1387 最大正方形

https://www.luogu.com.cn/problem/P1387

solution : 最大不包含 0 的正方形，换句话说就是找到最大的正方形区域，使得这个区域内所有数字的和刚好为区域大小，形式化的讲:

$$\sum_{i=x}^{x+l}\sum_{j=y}^{y+l}A_{i,j}=l^2$$

前缀和计算需要$O(nm)$，枚举边长l需要$O(\min(n,m))$，枚举横纵需要$O(n),O(m)$，即$O(nm+\min(n,m)nm)$ 由于n,m同阶，所以复杂度就是$O\left(n^3\right)$的.

如果不使用前缀和，则计算过程需要多一个$O\left(l^2\right)$，最终就是$O\left(n^5\right)$的.

所以，显然可以看出前缀和优化了时间复杂度，

关于前缀和我们就先讲到这里！

II. 差分

差分：对于序列$\{a_i\}_{i=1}^n$，他的差分序列$\{d_i\}_{i=1}^n$是：

$$d_i = a_i - a_{i-1}, \text{特别的 } a_0 = 0$$

其实不难发现，前缀和的差分是原序列，差分序列的前缀和是原序列！

也就是说，「前缀和」和「差分」是逆运算！

$$a_i = \sum_{j=1}^{i} d_i$$

$$S_i = \sum_{j=1}^{i}\sum_{k=1}^{j} d_k = \sum_{j=1}^{i}(i-j+1)\cdot d_j$$

但是，差分能做什么呢？
前缀和一般是用于处理离线的多次查询对吧！
差分就是用来处理多次修改的！

如果说我们要将序列$\{a_i\}$在区间$[l,r]$中的每个数都加上一个v，那么你就可以在他的差分序列$\{d_i\}$上做如下操作:

$$d_l \leftarrow d_l + v, d_{r+1} \leftarrow d_{r+1} - v$$

在对差分序列做完操作之后，$O(n)$进行一次前缀和还原成原序列即可！

你会发现，我们就实现了多次$O(1)$的区间修改，然后$O(n)$的前缀和还原即可

相比与之前朴素的$O(n)$的每次修改，如果是q次修改的话，那么朴素的复杂度就是$O(nq)$，而如果使用差分的方法修改，就是$O(q \cdot 1 + n)$的！

以上，便是一维差分！ 接下来，来一道例题吧！

对于一个长度为N的数列A，我们有Q次修改， 每次修改提供三个数字L, R, X，表示对于区间$[L, R]$的每个元素$+X$
求出修改后的数列A

```C++
#include<ios>
int main() {
  int n, q, i;
  scanf("%d%d",&n,&q);
  int a[n + 1],d[n + 1];
  for(i = 1; i <= n; i ++) scanf("%d",a+i);
  for(; q --;){
    int l,r,x; scanf("%d%d%d",&l,&r,&x);
    d[l] += x, d[r + 1] -= x;
  }
  for(i=1;i<=n;i++) a[i] = a[i - 1] + d[i];
}
```

这便是使用差分后的写法！ 差分我们就先介绍到这里，其实他的多维实现与前缀和并无差异，连公式都一样!!!!

III. 贪心算法入门

在生活中，我们会遇到许许多多类似的问题。它们往往具有一个共同的规律：整个问题的最优解，可以由其中各个小问题的最优解拼凑而成。

$$\Longrightarrow$$

换句话说，局部问题最优解的集合，就是整个问题的最终解！

这正是应用「贪心算法」的关键要素。

贪心算法（英语：**Greedy Algorithm**），是一种用计算机模拟“贪心”思维的决策过程。这个“人”非常贪婪 —— 每一步都按照某种指标选择当前看来最优的操作。而且他目光短浅，只关注眼前利益，不考虑未来可能的影响。

可想而知，贪心法并不总能得到全局最优解。因此，在使用贪心算法时，我们通常需要证明其正确性，才能放心地使用。

关于贪心算法的证明，这里提供两个方法：

1. 反证法：如果改变任意某个值会使答案变得更不好，那么就能证明当前状态是最优的了 $\Longrightarrow$ 当改变一个值不会使得情况变得差，那么改变这个值就是优的.
2. 归纳法：先计算出边界情况，或者说前几种状态的最优解F_1，然后再证明，对于后续的F_{n+1}，总可以有一个规律使得F_{n+1}可以由F_n推导而出.

其实贪心也有很多后续的变种，如反悔贪心等等.

接下来，来一道例题吧！

商店里有N堆糖果，每堆糖果有一个数目X，和一个价值Y
你需要选择一共M个糖果，使得你选择的所有糖果的总价值最大化
形式化的讲，你要最大化:

$$\text{ans} = \sum_{i=1}^{M} Y_i$$

这道题各位应该很容易就能想到，因为你要选择M个糖果，所以显然的选择价格最大的M个即可.
而这种思想，就是「贪心」！ 代码如下:

```C++
#include<ios>
#include<algorithm>
struct node{
  int y, x;
};
bool cmp(node a, node b){
  return a.y > b.y;
}
int main() {
  int n, m; scanf("%d%d",&n,&m);
  node a[n];
  for(int i = 0; i < n; i ++)
    scanf("%d%d",&a[i].x,&a[i].y);
  std::sort(a,a+n,cmp);
  long long ans = 0;
  int cnt = m;
  for(int i = 0; i < n && cnt > 0; i ++) {
    int take = std::min(a[i].x, cnt);
    ans += take * a[i].y;
    cnt -= take;
  }
  printf("%lld",ans);
}
```

现在让我们一块试试这道题吧！

CodeForces Round 1043 (division 3)

Problem E : Arithmetics Competition

https://codeforces.com/contest/2132/problem/E

hint：这道题要运用我们今天学的所有内容哦

考虑贪心，如果没有x_i,y_i的限制，那么直接选取前z_i大的牌是最优的。
因为$z_i \leq x_i + y_i$，可以看出此时的解只有两种不合法情况，即a超出x_i，或b超出y_i，二者不会同时满足。
对于超出的部分，我们用另一种卡替代。
可以证明这是最优的情况，因为对于两组卡牌，我们都选取了尽可能多且大的牌。
我们可以通过排序前缀和统计出前i大a，b牌组的贡献和选了前i大的总牌数属于a，b的数量来求出答案。

```C++
int a[N],b[N],cnta[N<<1],cntb[N<<1],s[N<<1],sa[N],sb[N];
void solve(){
    int n,m,q;
    cin>>n>>m>>q;
    vector<pair<int,int>>c;
    c.push_back({0,0});
    for(int i=1;i<=n;i++) cin>>a[i],c.push_back({a[i],1});
    for(int i=1;i<=m;i++) cin>>b[i],c.push_back({b[i],2});
    sort(a+1,a+1+n,greater<int>()),sort(b+1,b+1+m,greater<int>());
    sort(c.begin()+1,c.end(),greater<pair<int,int>>());
    for(int i=1;i<=n;i++) sa[i]=sa[i-1]+a[i];
    for(int i=1;i<=m;i++) sb[i]=sb[i-1]+b[i];
```

```
13      for(int i=1;i<=n+m;i++){
14          cnta[i]=cnta[i-1]+(c[i].second==1);
15          cntb[i]=cntb[i-1]+(c[i].second==2);
16          s[i]=s[i-1]+c[i].first;
17      }
18      while(q--){
19          int x,y,z;
20          cin>>x>>y>>z;
21          if(cnta[z]<=x&&cntb[z]<=y){
22              cout<<s[z]<<"\n";
23              continue;
24          }
25          if(cnta[z]>x) cout<<sa[x]+sb[min(z-x,m)]<<"\n";
26          if(cntb[z]>y) cout<<sa[min(z-y,n)]+sb[y]<<"\n";
27      }
28      for(int i=1;i<=n;i++) a[i]=sa[i]=0;
29      for(int i=1;i<=m;i++) b[i]=sb[i]=0;
30      for(int i=1;i<=n+m;i++) cnta[i]=cntb[i]=s[i]=0;
31  }
```

让我们再来看一道题

洛谷 P2949 [USACO09OPEN] Work Scheduling G

https://www.luogu.com.cn/problem/P2949

```C++
using PII = pair<int,int>;
void solve(){
    int n, ans = 0;cin >> n;
    vector<PII>arr(n);
    for(auto &[x,y] : arr)
        cin >> x >> y;
    sort(arr.begin(),arr.end());
    priority_queue<int,vector<int>,greater<int>>pq;
    for(auto [x,y] : arr){
        if(pq.size() < x){
            pq.push(y);
            ans += y;
            continue;
        }
        if(pq.size() >= x){
            if(pq.top() < y){
                ans += y;
                ans -= pq.top();  pq.pop();
                pq.push(y);
            }
        }
    }
    cout << ans << endl;
}
```

那么，以上就是今天的培训内容了，但是我还是想带同学们一起做一道思维题！

yukicoder No.3287 Golden Ring （黄金之环）

https://yukicoder.me/problems/no/3287

对于排列在圆环上的长度大于等于 2 的数组，若“任意相邻元素之和都不存在相同的情况”，则将其定义为“黄金环”。

也就是说，满足以下所有条件的数组 A 即为“黄金环”。

1. $2 \leq N$
2. A 是长度为N的数组
3. $S_i = A_i + A_{i+1} \quad (1 \leq i \leq N-1)$
4. $S_N = A_N + A_1$
5. 如果$i \neq j$则$S_i \neq S_j$

给定一个数N，从1到N的数组$(1, 2, ..., N)$已经给出，请你判断这个数组是否可以通过自由重排该数组来形成“黄金环”。如果可以输出这个“黄金环”数组！