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$\{a_i\}$，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：

$$S_0 = 0, S_i = S_{i-1} + a_i$$

于是我们可以得到另一个式子:

$$\sum a_{[l,r]} = S_r - S_{l-1}$$

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

```cpp
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);
  }
}
```

```cpp
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]);
  }
}
```

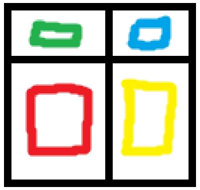
二维前缀和：其实和一维前缀和类似，但有一定区别， 实际上，对于一个大小为$M*N$的二维数组A，其前缀和的式子应该是

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

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：

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

洛谷 P1387 最大正方形

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:

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

前缀和计算需要$O(n\cdot m)$，枚举边长l需要$O(\min(n,m))$，枚举横纵需要$O(n),O(m)$，即$O(n^3)$.

如果不使用前缀和，则计算过程需要多一个$O(l*l)$，最终就是$O(N^5)$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$\{a_i\}$，他的差分序列$\{d_i\}$是：

$$d_i = a_i - a_{i-1}, a_0 = 0$$

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$O(n \cdot q)$，而如果使用差分的方法修改，就是$O(q+n)$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$[L, R]$均添加 X
求出修改后的数列A

```cpp
#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非常非常多的这样的问题，它们都有一个规律，整个问题的答案（或者说最优解），是由其中的小问题的最优解拼凑而成的

$$\Longrightarrow$$

换言之，局部问题的最优解的集合 即 最终问题的解！

上面这句话也是应用「贪心」算法的「充分必要条件」！！

贪心算法（英语：**greedy algorithm**），是用计算机来模拟一个「贪心」的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

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:

$$\sum_{i\in[1,M]} Y_i$$