位运算、双指针、排序

2351871 郎若谷

位运算

运算

- 高程学到的C++运算都是基于十进制的
- 接下来认识几个基于二进制的运算

位运算

- 左移 <<
- 右移 >>
- 按位与 &
- 按位或 |
- 按位异或 ^
- 按位取反 ~

- 注意对于负数的左移右移有需要注意的地方：
- 右移补充的数位是1，左移是 ub

优先级

- 位运算的运算优先级极低，所以建议常加括号
- 除了单目运算符按位取反 ~

- 此外在 cin 和 cout 也会用到 >> 和 << 运算符，但这俩是被重载过的，和现在讨论的位运算不是一个东西

为什么要位运算

- 位运算就是基于整数的二进制进行的运算，因为计算机底层存储数字就用的二进制，所以不需要对数据进行任何处理就可以直接进行运算操作，因此位运算的运算速度极快

- 位运算可以通过一些奇技淫巧达成很多类似于十进制的操作或者用十进制很难实现的操作。不管这些操作是怎么样的，只要用位运算实现，就会跑得很快

- 下面来看一些例子

最基本的几个

- 整除2或者乘2可以用左移和右移来加速实现
- 进一步地，整除或者乘 2 的整数次幂都可以用左移和右移来加速实现

- 判断一个数是奇数还是偶数也可以用按位与来加速实现
- 进一步地，对 2 的整数次幂取模都可以用位运算加速实现

- 这些强度还是不够大，来点更有意思的

常见的二进制数操作

- 获取一个数的二进制某位的值

```
int getBit(int a, int b) { return (a >> b) & 1; }
```

- 将某一位设置成0

```
int unsetBit(int a, int b) { return a & ~(1 << b); }
```

- 将某一位设置成1

```
int setBit(int a, int b) { return a | (1 << b); }
```

- 将某一位取反

```
int flapBit(int a, int b) { return a ^ (1 << b); }
```

奇技淫巧

- 判断两个数符号是否一致:

```
bool checkSign(int a, int b) {
    return (a ^ b) >= 0;
}
```

- 取绝对值:

```
int Abs(int n) {
  return (n ^ (n >> 31)) - (n >> 31);
```

奇技淫巧

- 交换两个数

```
void swap(int &a, int &b) {
    a ^= b ^= a ^= b;
}
```

- 把一个整数乘10

```
int b = (a << 1) + (a << 3);
```

- 容易注意到这个方法可以拓展到10以外的其他数

快速查看一个数的二进制

- 使用一个新的 STL 容器，名叫 bitset，需要引用 bitset 库
- 使用方法： bitset <位数> 变量名;

- bitset 简单地说就是一个固定位数的、只能存 0/1 的数组
- 可以对整个数组当成一个大01串进行位运算操作
- 如果对一个一般的数组进行运算是 $O(n)$ 的
- 但是 bitset 运算可以除以一个常数 $O(\frac{n}{w})$，w 一般是 32

```
int s = 0423;
bitset <10> b;
b = s;
cout << b << '\n';
```

```
0100010011
```

二进制的集合操作

- 把集合中的元素顺序排放
- 一个位置是 1 代表这个位置对应的元素存在，是 0 代表对应位置的元素不存在
- 按位与等价于求交
- 按位或等价于求并
- 按位取反等价于取补集

降序枚举子集

```cpp
int s = 023;
for(int x = s; x; x = (x - 1) & s) {
    cout << bitset <8> (x) << '\n';
}
```

```
00010011
00010010
00010001
00010000
00000011
00000010
00000001
```

- 真子集对应的整数显然小于全集，因此有种很简单的想法是从全
- 集开始不断 −1，然后判断当前这个数的 1 是否都是全集里的 1
- 这么做程序复杂度很高，原因是枚举了很多多余的状态，考虑如何直接跳过这么
- 没用的状态，即可以直接对全集进行按位与

例题

- 一行一个**奇数** n，接下来 n 个正整数，保证只有一个数值出现过奇数次，其他的数值都出现过偶数次，请你找出那个出现过奇数次的数值
- $n \leq 10^7, 1 \leq a_i \leq 10^{18}$
- 要求程序中只能使用至多 5 个整型变量
- 将所有的数全部异或起来即可
- P1469找筷子

双指针

何为双指针

- 并非高程课上所指的语法上的指针（int *p = &a）
- 而是指用两个变量去记录数组（或其他什么容器）的两个位置，然后通过不断移动这两个指针指向的位置去解决问题
- 这种问题一般具有一定的单调性，用双指针算法往往可以把复杂度降一维

例题

- 给定一个长度为 n 的正整数数组和整数 k，找出该数组内乘积小于 k 的连续子区间的个数
- $n \leq 10^7, 1 \leq a_i \leq 1000, k \leq 10^{18}$

例题

- 建两个指针 l,r 去寻找所有合法的区间，并再开设一个变量 x 记录当前 $[l,r]$ 内数的乘积
- 考虑不断右移指针 r，直到乘积 x 大于等于 k，随后固定 r 指针，然后不断右移指针 l，直到乘积 x 小于 k，然后再固定 l 去移动 r，以此循环...
- 考虑该算法的复杂度。注意到两个指针都在从数组的头部移动到尾部，并且不会折返，因此复杂度是 $O(n)$

- Leetcode 713

例题

- 给定一个已按照升序排列的整数数组 $\{a_n\}$，请你从数组中找出两个数满足相加之和等于给定的数 k
- $n, a_i, k \leq 10^5$
- $n, a_i, k \leq 10^7$

例题

- 考虑两个指针 l,r，l 初始指向 1，即最小的数，r 初始指向 n，即最大的数
- 计算 $a_l + a_r$，如果这比 k 大，说明需要使这两个数的和变小，那么就让大的那个数 a_r 变小，即让 r 减一即可
- 同理如果这个数比 k 小，那么就需要使这两个数的和变大，那么就让小的那个数 a_l 变大，即让 l 加一即可
- 这样让 l 不断右移，r 不断左移，即可逼近得到所有答案
- 注意到指针 l,r 最多只会把数组遍历一遍，所以总复杂度为 $O(n)$

- Leetcode 167

排序算法原理

归并排序，快速排序

前置知识

- 冒泡排序（高程课上讲过） $O(n^2)$
- 库函数 sort （上次课讲过）$O(n \log n)$
- 如何用 sort 实现个性化排序?
- 一：如果想从大到小排序，C++ 已经为你提供了一个函数

```
sort(a + 1, a + 1 + n, greater<int>());
```

- 二：如果想使用更加个性化的排序方法，请写 cmp 函数

  比如我想对 n 个数按照个位数的大小排序，该怎么写?
- 三：重载 < 运算符，重载内容与 cmp 一致

排序算法

- 排序作为一个很基础的命题，已经有了很多算法
- 这些仅是在算法竞赛中可以得到应用的排序方法

- 但是我们并不需要全部掌握
- 我们只挑计数排序、归并排序、快速排序来讲讲

选择排序

冒泡排序

插入排序

计数排序

基数排序

快速排序

归并排序

堆排序

桶排序

希尔排序

锦标赛排序

tim排序

计数排序

- 对于值域不是很大的数组排序，可以开个值域大小的数组 b，$b[i]$ 存数值 i 出现的次数
- 进行一次 for 循环每次把 $b[a[i]]++$
- 结束以后再把 b 数组从小到大扫一遍即可

- 稳定排序算法
- 时间复杂度 $O(n+V)$，空间复杂度也是 $O(n+V)$
- 只适用于值域 V 不那么大的情况

归并排序

- 归并排序的算法流程是，先不断地从 $[l,r]$ 往下递归到 $[l,mid]$ 和$(mid,r]$
- 直到递归的区间长度变为 1 时为止，显然长度为 1 的数组一定是有序的
- 再向上回溯合并
- 此时我们保证了对于回溯到的区间 $[l,r]$，他的左右两个子区间 $[l,mid]$ 和 $(mid,r]$ 内部已经有序，我们需要将这两个有序的数组合并成一个大的有序的数组 $[l,r]$
- 可以使用双指针来合并
- 根据主定理可以证明，该算法复杂度是 $O(n\log n)$，稳定排序算法
- 以后有些其他算法也会用到归并排序的思想原理

归并排序

```
void qsort(int l, int r) {
    if(l == r) return;
    int mid = (l + r) >> 1;
    qsort(l, mid);
    qsort(mid + 1, r);
    int p = l, q = mid + 1, now = l;
    while(p <= mid && q <= r) {
        if(a[p] < a[q]) b[now++] = a[p++];
        else b[now++] = a[q++];
    }
    while(p <= mid) b[now++] = a[p++];
    while(q <= r)   b[now++] = a[q++];
    for(int i = l; i <= r; i++) a[i] = b[i];
}
```

快速排序

- 设现在要对数组下标为 $[l,r]$ 的数进行排序，我们随机一个 a 数组中的数 $x=a_k$
- 对 $[l,r]$ 进行一次遍历以将小于 x 的数放进一个数组里（记为数组 u），等于 x 的数放进另一个数组里（记为数组 v），将大于 x 的放进第三个数组里（记为 w）
- 然后将三个数组按照 uvw 的顺序连在一起重新合成 $[l,r]$，此时 $[l,r]$ 区间分成三部分 $[l,p],[p,q],[q,r]$，形如这样
- 小于 x 的元素 | 等于 x 的元素 | 大于 x 的元素
- 在递归如此这样分别对 $[l,p],[q,r]$ 进行处理即可
- 不断处理直到区间大小为 1 后结束递归，回溯完后整个数组就被排好序了

快速排序

- 很显然可以发现，如果 x 取得不同，每次往下递归的区间 $[l,p],[q,r]$ 的大小就不同，每次排序所用的时间就会有所不同
- 根据主定理可以证明，当每一次递归时，$[l,p],[q,r]$ 的大小都接近于区间 $[l,r]$ 的一半时（即 x 将 $[l,r]$ 平分了时），快速排序的时间复杂度为 $O(n\log n)$
- 当每次递归时，这两个区间中一个区间的大小接近 $[l,r]$ 原本的大小，另一个区间很小时，时间复杂度会退化为 $O(n^2)$
- 所以快速排序是 不稳定排序算法
- 时间复杂度 $O(n\log n)\sim O(n^2)$
- 对于分界值等概率随机的平均情况，可以证明快速排序的复杂度是$O(n\log n)$，因为其常数比较小，所以随机情况下表现优于堆排序等其他 $O(n\log n)$ 的算法

快速排序

```
void qsort(int l, int r) {
    if(l >= r) return;
    int us = 0, vs = 0, ws = 0;
    int x = a[(l + r) >> 1];
    for(int i = l; i <= r; i++) {
        if(a[i] < x) u[++us] = a[i];
        else if(a[i] == x) v[++vs] = a[i];
        else w[++ws] = a[i];
    }
    for(int i = 1; i <= us; i++) a[l + i - 1] = u[i];
    for(int i = 1; i <= vs; i++) a[l + us + i - 1] = v[i];
    for(int i = 1; i <= ws; i++) a[l + us + vs + i - 1] = w[i];
    qsort(l, l + us - 1);
    qsort(l + us + vs, r);
}
```

快速排序

- 一种常数更小，占用空间更小，写起来也更短的写法：i 代表现在枚举的坐标，$[l, j-1]$ 存比 x 小的数，$[k+1, r]$ 存比 x 大的数

```
void qsort(int l, int r) {
    if(l >= r) return;
    int x = a[(l + r) >> 1];
    int i = l, j = l, k = r;
    while(i <= k) {
        if(a[i] < x) swap(a[i++], a[j++]);
        else if(a[i] == x) i++;
        else swap(a[i], a[k--]);
    }
    qsort(l, j - 1); qsort(k + 1, r);
}
```

快速排序

- 可以发现，纸面上的数据，快速排序是不如归并排序的，因为它可能会退化到 $O(n^2)$，但是总所周知 C++ 的库函数 sort 就是通过快速排序实现的
- 那为什么 C++ 的编写者会选择用快排实现 sort 而不使用其他排序方法？而且我们使用 sort 时会发现这个函数跑得飞快完全没有出现退化的现象
- 原因是 C++ 的编写者使用了各种奇技淫巧来优化快速排序算法，具体可以见这篇洛谷日报，总而言之，通过各种技巧优化后的 sort 可以处理各种会导致复杂度退化的情况，并且常数极小
- https://www.luogu.org/blog/Victory-Defeat/qian-xi-sort

THANKS