第二周直播课（下午）
基础算法思想 I

洛谷网校
基础-提高衔接计划
2024-07
disangan233

课前声明

- 欢迎各位开始洛谷基础-提高衔接计划的学习
- 本周为第三周。考虑到由易到难的课程特征，本周的作业内容难度中等
- 本周作业讲解将会分析每道必做作业题目的思路，并进行部分作业题目的代码讲解
- 如果有不懂的其他内容，可以 QQ 群进行提问

目录

前缀和

- 前缀和可以简单理解为数列前 i 项的和
- 它是一种重要的预处理方式，能大大降低没有修改、多次查询的时间复杂度
- 一般来讲，我们会预处理一个数组。对数组中每个元素，我们记录从起始到该元素对应下标/状态所有数字的总和
- 具体来讲，前缀和一般分为以下几种
- 一维前缀和、二维前缀和、多维前缀和

一维前缀和

- 给定一个长度为 n 的数组 a，预处理数组 f 作为前缀和

$$f_i = \sum_{j=1}^{i} a_j = a_1 + a_2 + \ldots + a_i$$

- 可以直接递推求解

$$f_i = f_{i-1} + a_i$$

二维前缀和

- 给定一个 $n \times m$ 的数组 a，预处理二维数组 f 作为前缀和

$$f_{i,j} = \sum_{k=1}^{i} \sum_{l=1}^{j} a_{k,l}$$

- 考虑如何对 $f_{i,j}$ 进行递推，有

$$f_{i,j} = f_{i-1,j} + f_{i,j-1} - f_{i-1,j-1} + a_{i,j}$$

- 预处理时间复杂度为 $O(nm)$

多维前缀和

- 可以类推其定义
- 具体如何维护多维前缀和，涉及到容斥原理的知识，超出本课程范围，可以自行学习
- 二维前缀和的维护就是容斥原理的应用

前缀和 例题

- B3612 【深进 1. 例 1】求区间和
- 给定数列 a_n 和 m 个区间 $[l_i, r_i]$，分别这 m 个区间的区间和
- $n, m \leq 10^5$， $a_i \leq 10^5$

前缀和 例题

- B3612 【深进 1. 例 1】求区间和
- 给定数列 a_n 和 m 个区间 $[l_i, r_i]$，分别这 m 个区间的区间和
- $n, m \leq 10^5$， $a_i \leq 10^5$
- 维护前缀和 b，那么 $b_r - b_{l-1} = \sum_{i=l}^{r} a_i$

```
1 for(int i=1;i<=n;i++)
2     b[i]=b[i-1]+a[i];
3 for(int i=1;i<=m;i++)
4     printf("%d\n",b[r[i]]-b[l[i]-1]);
```

前缀和 例题

- P2280 [HNOI2003] 激光炸弹
- 有 n 个目标，第 i 个目标位于 (x_i, y_i)，价值为 v_i，求 $m \times m$ 的矩形内最大的目标价值和
- $n \leq 10^4$， $x_i, y_i, m \leq 5 \times 10^3$

前缀和 例题

- P2280 [HNOI2003] 激光炸弹
- 有 n 个目标，第 i 个目标位于 (x_i, y_i)，价值为 v_i，求 $m \times m$ 的矩形内最大的目标价值和
- $n \leq 10^4$，$x_i, y_i, m \leq 5 \times 10^3$
- 没有修改，可以直接用前缀和来快速求区间和
- 合法的矩形只有 $(M-m)^2$ 个，M 为坐标上界
- 类似于求递推式的过程，利用前缀和 b 求区间和，有

$$\sum_{i=x_1}^{x_2}\sum_{j=y_1}^{y_2} a_{i,j} = b_{x_2,y_2} - b_{x_1-1,y_2} - b_{x_2,y_1-1} + b_{x_1-1,y_1-1}$$

- 单次可以 $O(1)$ 求区间和，总复杂度 $O(m^2)$

前缀和 例题

```
for(int i=1;i<=N;i++)
for(int j=1;j<=N;j++)
    s[i][j]+=s[i-1][j]+s[i][j-1]-s[i-1][j-1];
int ans = 0;
for(int i=m;i<=N;i++)
for(int j=m;j<=N;j++) {
    int num = s[i][j] - s[i - m][j] - s[i][j - m]
        + s[i - m][j - m];
    // num 为以 (i, j) 为右下角的边长为 m 的正方形
        区域中的目标价值之和
    ans = max(ans, num);
    // 用 num 更新答案
}
```

差分

- 差分可以简单地理解为前缀和的逆运算
- 它也是一种重要的处理方式，能大大降低多次修改、最终只有一次查询的时间复杂度
- 一般来讲，我们会处理一个数组。对数组中每个元素，记录该元素对应下标/状态代表的值与其之前一个的值的差值
- 形式化的讲，对一个长度为 n 的数组 a，建立差分数组 b

$$b_i = \begin{cases} a_1, & i = 1 \\ a_i - a_{i-1}, & 2 \le i \le n \end{cases}$$

- 对差分数组做一次前缀和，即可得到原数组

差分

- 主要作用：多次修改但最终只有一次查询
- 给定 n 个整数，m 次修改，每次将区间 $[l_i, r_i]$ 中数字加上 x_i，求 m 次修改之后的序列
- 如果使用暴力，需要对每个修改从 l_i 枚举到 r_i，时间复杂度会很大
- 考虑对差分数组进行操作，每次修改相当于

$$b_{l_i} \leftarrow b_{l_i} + x_i, \quad b_{r_i+1} \leftarrow b_{r_i+1} - x_i$$

- 最后查询时做一遍前缀和即可

差分

```
scanf("%d%d",&n,&m);
for(int i=1;i<=n;i++) {
    scanf("%d",&a[i]);
    b[i]=a[i]-a[i-1];
}
for(int i=1;i<=m;i++) {
    scanf("%d%d%d",&l,&r,&x);
    b[l]+=x;
    b[r+1]-=x;
}
for(int i=1;i<=n;i++)
    b[i]+=b[i-1];
for(int i=1;i<=n;i++)
    printf("%d ",b[i]);
```

差分 例题

- P3397 地毯
- 在 $n \times n$ 的格子上有 m 个地毯，给出每块地毯的左上角 (x_1, y_1) 和右下角 (x_2, y_2)，求每个点被多少地毯覆盖
- $n, m \leq 10^3$

差分 例题

- P3397 地毯
- 在 $n \times n$ 的格子上有 m 个地毯，给出每块地毯的左上角 (x_1, y_1) 和右下角 (x_2, y_2)，求每个点被多少地毯覆盖
- $n, m \leq 10^3$
- m 次二维的区间加 1，最后询问一次
- 将二维前缀和的思想转换一下，即可得到二维差分

$$a_{x_1,y_1} \leftarrow a_{x_1,y_1} + 1, \quad a_{x_1,y_2+1} \leftarrow a_{x_1,y_2+1} - 1$$
$$a_{x_2+1,y_1} \leftarrow a_{x_2+1,y_1} - 1, \quad a_{x_2+1,y_2+1} \leftarrow a_{x_2+1,y_2+1} + 1$$

- 时间复杂度 $O(nm)$

差分 例题

```
scanf("%d%d",&n,&m);
for(int i=1;i<=m;i++) {
    int x1,y1,x2,y2;
    scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
    a[x1][y1]++;
    a[x1][y2+1]--;
    a[x2+1][y1]--;
    a[x2+1][y2+1]++;
}
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++) {
    a[i][j]+=a[i-1][j]+a[i][j-1]-a[i-1][j-1];
    printf("%d",a[i][j]);
    putchar(" \n"[j==n]);
}
```

双指针

- 双指针（Two-Pointer）是同时使用两个指针，指向序列、链表结构上的位置（或树、图结构中的节点），通过或同向/相向移动来维护、统计信息
- 具体使用方法有
  1. 利用序列有序性
  2. 维护区间信息
  3. 快慢指针-在单向链表中找环
- 时间复杂度一般为 $O(n)$

利用序列有序性

- P1102 A-B 数对
- 给定 n 个正整数和正整数 C，计算出所有满足 $A - B = C$ 的数对个数
- $n \leq 2 \times 10^5$

利用序列有序性

- P1102 A-B 数对
- 给定 n 个正整数和正整数 C，计算出所有满足 $A - B = C$ 的数对个数
- $n \leq 2 \times 10^5$
- 将数组排序后使用双指针，前指针枚举 A，后指针尝试寻找满足条件的 B。
- 对于某个 A，当后指针指向的元素小于 $A - C$ 时，不断向数组增大方向跳跃，直至找到 $\geq A - C$ 的元素

维护区间信息

- P1147 连续自然数和
- 给定正整数 M，求出所有的连续正整数段 $(l,r)(l<r)$，使得 $l+(l+1)+\ldots+(r-1)+r=M$
- $m \leq 2 \times 10^6$

维护区间信息

- P1147 连续自然数和
- 给定正整数 M，求出所有的连续正整数段 $(l, r)(l < r)$，使得 $l + (l+1) + \ldots + (r-1) + r = M$
- $m \leq 2 \times 10^6$
- 用 l, r 两个指针维护区间和 s
- 在 $s < M$ 的情况下，每次让 r 右移一位，$s \leftarrow s + r$
- 当 $s > M$ 时，不断右移 l，同时 $s \leftarrow s - l$
- 如果移动 l 后 $s = M$，则输出此时的 l, r

维护区间信息

```
int l = 1, r = 1, cur = 1;
while (r < m) {
    while (cur > m)
        cur -= (l++);
    if (cur == m)
        cout << l << " " << r << endl;
    cur += (++r);
}
```

在单向链表中找环

- 快慢指针可以用于在单向链表中找环
- 首先两个指针都指向链表的头部，令一个指针一次走一步，另一个指针一次走两步，如果它们相遇了，证明有环，否则无环
- 总时间复杂度 $O(n)$
- 找到环的起点：在两个指针相遇后，将其中一个指针移到表头，让两者都一步一步走，再度相遇的位置即为环的起点
- 证明：设二者第一次相遇时慢指针一共走 k 步，快指针走了 $2k$ 步。设单指针在环上走了 l 步，环长为 C，有

$$2k = nC + l + (k - l) \implies k = nC$$

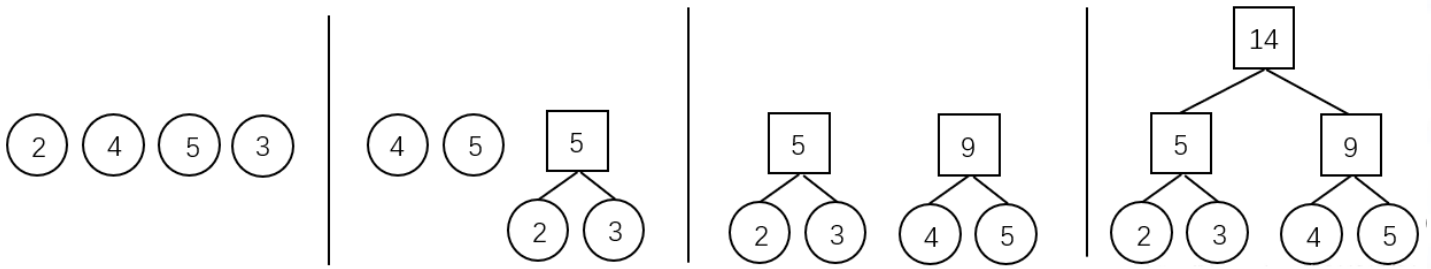
- 第一次相遇时 $n = 1$，即 $k = C$

Huffman 树

- 树的带权路径长度（WPL）：设二叉树有 n 个带权叶节点，从根节点到各叶节点的路径长度与相应叶节点权值的乘积之和
- Huffman 树：对于给定一组具有确定权值的叶节点，可以构造出不同的二叉树，其中 WPL 最小的二叉树称为哈夫曼树（Huffman Tree）

Huffman 树

- Huffman 算法用于构造一棵 Huffman 树，算法步骤如下：
  1. 初始化：由给定的 n 个权值构造 n 棵只有一个根节点的二叉树，得到一个二叉树集合 F
  2. 选取与合并：从二叉树集合 F 中选取根节点权值最小的两棵二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根节点的权值为其左、右子树根结点的权值和
  3. 删除与加入：从 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中
  4. 重复 2、3 步，当集合中只剩下一棵二叉树时，这棵二叉树就是 Huffman 树

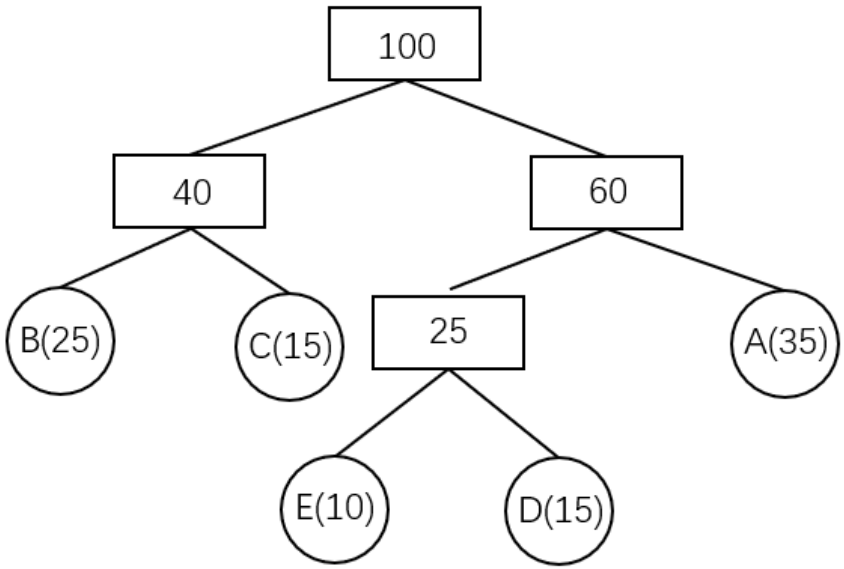


Huffman 编码

- 在进行程序设计时，通常给每一个字符标记一个单独的代码来表示一组字符，即编码
- 在进行二进制编码时，假设所有的代码都等长，那么表示 n 个不同的字符需要 $\lceil \log_2 n \rceil$ 位，称为等长编码
- 如果每个字符的使用频率相等，那么等长编码无疑是空间效率最高的编码方法
- 如果字符出现的频率不同，则可以让频率高的字符采用尽可能短的编码，频率低的字符采用尽可能长的编码，来构造出一种不等长编码，从而获得更好的空间效率
- 前缀编码：任一编码都不是其他任何一个编码的前缀

Huffman 编码

- Huffman 编码：将字符集及出现频率作为叶节点插入构造 Huffman 树。规定 Huffman 编码树的左分支代表 0，右分支代表 1，则从根结点到每个叶结点所经过的路径组成的 0, 1 序列即为该叶结点对应字符的编码



字符	频率	编码
A	35	11
B	25	00
C	15	01
D	15	101
E	10	100

Huffman 编码 例题

- CSP-J 2022 初赛，第 7 题
- 假设字母表 {a，b，c，d，e} 在字符串中出现的频率分别为 $10\%, 15\%, 30\%, 16\%, 29\%$。若采用哈夫曼编码的方式对字母进行不定长的二进制编码，字母 d 的编码长度为（ ）位
- A. 1　　B. 2　　C. 2 或 3　　D. 3

Huffman 编码 例题

- CSP-J 2022 初赛，第 7 题
- 假设字母表 {a，b，c，d，e} 在字符串中出现的频率分别为 $10\%, 15\%, 30\%, 16\%, 29\%$。若采用哈夫曼编码的方式对字母进行不定长的二进制编码，字母 d 的编码长度为（ ）位
- A. 1　　B. 2　　C. 2 或 3　　D. 3

- B

字符串哈希

- 我们定义一个把字符串映射到整数的函数 f，这个 f 称为是 Hash 函数
- 我们希望函数 f 可以方便地帮我们判断两个字符串是否相等
- Hash 的核心思想在于，将输入映射到一个值域较小、可以方便比较的范围
- 具体来说，哈希函数最重要的性质可以概括为下面两条：
  1. 在 Hash 函数值不一样的时候，两个字符串一定不一样
  2. 在 Hash 函数值一样的时候，两个字符串不一定一样（但有大概率一样，且我们当然希望它们总是一样的）
     - Hash 函数值一样但原字符串不一样的现象称为哈希碰撞
- 一个例子：有 9 个 QQ 号，只要记末两位就可以知道谁是谁了

字符串哈希

- 我们需要关注的是时间复杂度和 Hash 的准确率
- 通常我们采用的是多项式 Hash 的方法，对于一个长度为 l 的字符串 s 来说，可以这样定义多项式 Hash 函数：

$$f(s) = \sum_{i=1}^{l} s_i \times b^{l-i} \pmod{M}$$

- 例如，对于字符串 xyz，其哈希函数值为 $xb^2 + yb + z$
- 特别要说明的是，也有很多人使用的是另一种定义，即

$$f(s) = \sum_{i=1}^{l} s_i \times b^{i-1} \pmod{M}$$

- 这种定义下，字符串 xyz 的哈希值变为 $x + yb + zb^2$

字符串哈希

- 两种哈希函数的定义函数都是可行的，但二者在之后会讲到的计算子串哈希值时所用的计算式是不同的，因此千万注意不要弄混了这两种不同的 Hash 方式
- 前者的 Hash 定义计算更简便、使用人数更多、且可以类比为一个 b 进制数来帮助理解，我们之后的讲解也使用前者
- M 需要选择一个素数（至少要比最大的字符要大），b 可以任意选择，常用 $131, 233, 13131$ 等

字符串哈希

- 参考代码：（效率低下的版本，实际使用时一般不会这么写）

```
const int M = 1e9 + 7;
const int B = 233;
typedef long long ll;
int get_hash(string s) {
    int res = 0;
    for (int i = 0; i < s.size(); ++i) {
        res = ((ll)res * B + s[i]) % M;
    }
    return res;
}
bool cmp(string s, string t) {
    return get_hash(s) == get_hash(t);
}
```

哈希的分析与改进

- 假定哈希函数将字符串随机地映射到大小为 M 的值域中，总共有 n 个不同的字符串，那么未出现碰撞的概率是 $\prod_{i=0}^{n-1} \frac{M-i}{M}$（第 i 次进行哈希时，有 $\frac{M-i}{M}$ 的概率不会发生碰撞）
- 在随机数据下，若 $M = 10^9 + 7$，$n = 10^6$，未出现碰撞的概率是极低的
- 所以，进行字符串哈希时，经常会对两个大质数分别取模，这样的话哈希函数的值域就能扩大到两者之积，错误率就非常小了
- 这就是我们所常用的双哈希（结合两个 Hash 值 $\{h_1, h_2\}$ 作为值域）

双哈希

```
const int M1 = 1e9 + 7, M2 = 998244353;
const int B1 = 23333, B2 = 13131;
int h1(string s) {
    int res = 0;
    for (int i = 0; i < s.size(); ++i)
        res = ((ll)res * B1 + s[i]) % M1;
    return res;
}
int h2(string s) {
    int res = 0;
    for (int i = 0; i < s.size(); ++i)
        res = ((ll)res * B2 + s[i]) % M2;
    return res;
}
bool cmp(string s, string t) {
    return h1(s) == h1(t) && h2(s) == h2(t);
}
```

多次询问子串哈希

- 单次计算一个字符串的哈希值复杂度是 $O(n)$，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下
- 一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了：
- 令 $f_i(s)$ 表示 $f(s_{1..i})$，即原串长度为 i 的前缀的哈希值，那么按照定义有 $f_i(s) = s_1 \cdot b^{i-1} + s_2 \cdot b^{i-2} + \cdots + s_{i-1} \cdot b + s_i$
- 现在，我们想要用类似前缀和的方式快速求出 $f(s_{l..r})$，按照定义有字符串 $s_{l..r}$ 的哈希值为
  $f(s_{l..r}) = s_l \cdot b^{r-l} + s_{l+1} \cdot b^{r-l-1} + \cdots + s_{r+1} \cdot b + s_r$

多次询问子串哈希

- 对比观察上述两个式子，发现 $f(s_{l..r}) = f_r(s) - f_{l-1}(s) \times b^{r-l+1}$ 成立（可以手动代入验证一下），因此我们用这个式子就可以快速得到子串的哈希值
- 其中 b^{r-l+1} 可以 $O(n)$ 预处理，然后 $O(1)$ 的回答每次询问（当然也可以快速幂 $O(\log n)$ 的回答每次询问）

字符串哈希 例题

- P3370 【模板】字符串哈希
- 给定 n 个字符串 s_n，求有多少个不同的字符串
- $n \leq 10^4$，$|s_i| \leq 1500$

字符串哈希 例题

- P3370 【模板】字符串哈希
- 给定 n 个字符串 s_n，求有多少个不同的字符串
- $n \leq 10^4$，$|s_i| \leq 1500$
- 小质数（10^9 左右）的单哈希会被卡，使用双哈希可以通过
- 可以采用 `set<pair<int,int>>` 来存储双哈希

字符串哈希 例题

```
set<pair<int,int>>S;
for(int i=1;i<=n;i++) {
    scanf("%s",s);
    pair<int,int> h=make_pair(h1(s),h2(s));
    if(!S.count(h))
        S.insert(h);
}
cout<<S.size();
```

字符串哈希 例题

- P3879 [TJOI2010] 阅读理解
- 有 N 篇短文，每篇短文含 L 个单词
- 给定 M 个生词，求每个生词在哪些短文中出现过
- $N \leq 10^3$，$M \leq 10^4$，$L \leq 5 \times 10^3$，$|s| \leq 20$

字符串哈希 例题

- P3879 [TJOI2010] 阅读理解
- 有 N 篇短文，每篇短文含 L 个单词
- 给定 M 个生词，求每个生词在哪些短文中出现过
- $N \leq 10^3$，$M \leq 10^4$，$L \leq 5 \times 10^3$，$|s| \leq 20$
- 用 vector、map 等数据结构存储每个短文的 L 个 Hash 值，在其中查找生词的 Hash 值

字符串哈希 例题

```
set<pair<int,int>>S[1005];
for(int i=1;i<=n;i++) {
    scanf("%d",&l);
    for(int j=1;j<=l;j++) {
        scanf("%s",s);
        S[i].insert(make_pair(h1(s),h2(s)));
    }
}
while(m--) {
    scanf("%s",s);
    pair<int,int>h=make_pair(h1(s),h2(s));
    for(int i=1;i<=n;i++)
        if(S[i].count(h))
            printf("%d ",i);
    printf("\n");
}
```