Report. Longest Substring

班级：建 41
学号：2014010026
姓名：罗辰

一. 所使用数据结构与算法的构思、原理和实现要点。

1.所使用的数据结构：**散列(hash)**和**后缀数组(suffix array)**

2.算法的**构思、原理和实现要点**：

思路 1：题目要求概括来说就是在字符串中查找最长频繁子串，如果对所有可能的子串长度都做一次查找，则会退化成为蛮力算法笃定超时。但是只要知道最长的出现 m 次子串，注意到如果存在长度为 L 的出现 m 次子串，那么也一定存在长度为 L-1 的出现 m 次的子串(即长度为 L 的出现 m 次的子串去掉串的最末尾的字符)。因此只需要对长度进行二分查找即可。对于每一个待判断的长度 x，需要对字符串中所有长度为 X 的子串进行统计，考察其中是否至少有 m 个相同。字符串 s 中有 len(s)-x+1 个长度为 x 的子串，希望通过计算这些子串的 hash 值在 O(1)的时间内得到该子串出现的次数。因此得到 hash 值的效率显得至关重要，注意到字符串中的小写字母 a~z 可视为 1~26 这 26 个正整数，字符串的字符排列可以视为一个 p(p>=27)进制的数，将这个 p 进制的数字转换为 10 进制并对 M(一般为素数)取余作为当前长度为 x 的字符串的 hash 值，则最左边第一个长度为 x 的子串得到 hash 值的时间为 O(x)，若之后的各个子串均采用从高位到低位逐个计算的方法，每一次都会耗费 O(x)的时间，在时间效率上是低的。根据模余满足加法的交换率和乘法的分配率，可以在原有 hash 值基础上减去原有首字符对应数字乘以 p 的 x-1 次方后对 M 模余的值，再加上新的尾字符对应的数字模余 M 的值，这样在 O(1)的时间内完成了 hash 值的更新。但是这样一旦出现散列冲突就需要做串匹配，最坏的情况是每次都需要做串匹配，每次匹配耗时为 O(x)，也会超时。为了避免模式串的匹配，可以取两个不同的 hash 函数，只需要修改进制和 M 的值即可，只有两个 hash 函数的值都相同的情况下才判定这两个字符串相同，否则就是作为两个不同的字符串分别加入到散列表中。虽然这样判断是有风险的，不过两个 hash 相等而串不相同的情况概率是非常低的，期望意义下这种判断是可行的。

思路 2：对一个字符串用倍增算法求出后缀数组 sa，rank 数组可由 sa 数组得到，最后根据 rank 数组求出 height 数组。height 数组中存放着排名相邻的两个后缀的最长公共前缀的长度，实质上对于某个长度为 x 的所有子串进行重复个数计算的时候，只需要对 height 数组做一次线性的遍历，第一次遇上 height[i]>=k 的时候说明排名为 i 和 i-1 的两个后缀的最长公共前缀的前 k 个字符满足条件，即满足条件的子串首次出现，height[i+1]如果也满足这个性质则说明该子串出现的次数加一；一旦性质不再满足即 height[i]<k 时说明之前满足条件的子串已经全部出现过了，再之后碰上满足条件的 height[i]对应的就是新的满足条件的子串了。扫描的过程中如果有若干个子串均满足至少出现 k 次的条件，则需要在扫描过程中记录并根据后缀数组中保存的首字符的秩来保证一直保存的都是最靠右的满足条件的子串。这种算法区别于散列法可能存在的风险，它能保证百分百的正确性。

实现要点：

散列法：1.hash 函数的快速更新。2.需要取至少两个 hash 函数来对字符串是否重复进行判断。3.对于待判断的长度 x,要预处理好 p 的 x-1 方模余 M 的值，便于在后续 hash 值更新时使用。

后缀数组法：1.倍增算法对字符串排序的过程是采用基数排序。2.对于长度为 1 的情况需要单独处理，属于边界情况单独处理。

散列法问题：最开始只使用了一个 hash 函数，对于冲突的处理是进行字符串匹配，结果超

时；若是将冲突直接当作字符串重复出现来处理又会 wrong answer。上完最后一次习题课后采用了额外计算一个 hash 函数来辅助判定的方法，虽然有风险但是出错概率极低。

后缀数组法问题：没有单独处理 1 的情况，会导致部分测试点 wrong answer。后面注意到当求字符串中至少出现一次的最长频繁子串的时候答案一定是 0，因为就是串本身满足条件。但是若采用对 height 数组遍历一遍的方法，在判断子串长度 x=len(s)的时候，由于 height 数组中最大值也只能为 len(s)-1，因此会将 x=len(s)的时候误判成不满足条件，从而影响二分查找的结果。因此对于 k=1 的情况单独处理成输出 0 即可，其余还是按照二分查找+遍历 height 数组的方法进行。

最终实现的时候采用的是对 T 项任务，每一次计算的时候都随机地取这两种方法之一进行计算。

参考资料：

散列法：

1.https://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/string_pm_kr/pm_kr.h.htm

参考了代码示例列表中 Karp-Rabin 算法的模板，包括预处理和 hash 值的更新

2.散列时基数选取 131 和 1000000007 是在最后一次习题课上参考了丁铭助教的建议

3.使用两个 hash 函数分别计算 hash 值来尽可能避免散列冲突时逐个比较字符串字符的操作，这一步也是参考最后一次习题课上的提示

后缀数组法

1.https://www.cnblogs.com/zinthos/p/3899725.html

阅读了这篇博客，学习了后缀数组的相关知识

2.https://wenku.baidu.com/view/ed1be61e10a6f524ccbf85fd.html

阅读了《后缀数组—处理字符串的有力工具》这篇 2009 年的国家集训队论文，学习了后缀数组的应用

使用了上述论文中的第 6 页和 17 页的倍增算法和计算 height 数组的代码

三. 时间和空间复杂度的估算。

散列法

1.时间复杂度估算，这里语义上约定 n=strlen(s)，即 n 为字符串 s 的长度

定义变量、读入字符串和给变量赋值的时间视为 **O(1)**;

```
char temp[40002];
scanf("%d %s", &m, temp);
int lengthT = (int)strlen(temp);
int lo = 1;
int hi = lengthT + 1;
```

对字符串长度进行二分查找，查找的总次数为 **O(logn)**;

```
int finallen = binSearch1(temp, lo, hi, lengthT);
int binSearch1(char* T, int lo, int hi, int len) {
    while (lo < hi) {
        int mi = (lo + hi) >> 1;
        workable1(T, mi, len) == true ? lo = mi + 1 : hi = mi;
    }
    return --lo;
```

```
}
```
对具体的某个长度 x 进行判断的时候，先对散列表初始化，由于散列表的长度为 O(n),因此初始化散列表的时间复杂度为 **O(n)**;
```
pos = 0;
rankofright = -1;
for (int i = 0; i < M; i++) {//散列表初始化，独立链的桶均置为空
    hashnode[i].cnt = 0;
    hashnode[i].next = NULL;
    sup[i].cnt = 0;
    sup[i].next = NULL;
}
```
预处理 29 的 x-1 次方后模余 M 的值以及 131 的 x-1 次方后模余 M 的值，时间复杂度为 **O(x)**;
```
HashCode Dm, hashT = 0, checkhashT = 0;
Dm = prepareDm(x);
```
计算第一个长度为 x 的子串的 hash 值，时间复杂度为 O(x)，但是对于后面的 n-x 个长度为 x 的子串，hash 值的更新均只需要 O(1)的时间，**分摊**到每一步的求得 hash 实质上仍然是 O(1)的时间。
```
for (size_t i = 0; i < x; i++) {//第一个长度为 x 的子串的 hash 值
    hashT = (hashT*R + DIGIT(T, i)) % M;
    checkhashT = (checkhashT*checkR + DIGIT(T, i)) % checkM;
}
```
接下来是将字符串插入到散列表的过程。若 hash 值对应的散列表的桶为空，则直接插入并且更新出现次数等信息。若桶非空，则需要和桶的每一个槽位中的用来检验的 checkhash 值进行比较，只有 hash 值和 checkhash 值均相等的时候才算字符串重复出现。若整个比较过程中都没有出现相等的，则需要额外添加一个槽位。期望意义下散列若是均匀的，则单次插入到散列表的时间复杂度为 O(1)，由于共有 n-x+1 个子串需要插入到散列表，所以整个插入散列表的过程时间复杂度为 O(n-x+1);综上对于单独某个长度 x 进行判断的时间复杂度为 O(n)+O(x)+O(x)+O(n-x)+O(n-x+1)=**O(n)**;
```
hashnode[hashT].cnt++;
hashnode[hashT].checknum = checkhashT;
if (hashnode[hashT].cnt >= m)
    rankofright = 0;
for (size_t k = 1; k < len - x + 1; k++) {
    updateHash(hashT, checkhashT, T, x, k, Dm);//更新 hash 值
    if (hashnode[hashT].cnt == 0) {//遇上空桶直接插入
        hashnode[hashT].cnt++;
        hashnode[hashT].checknum = checkhashT;
        if (hashnode[hashT].cnt >= m)
            rankofright = k;
    }
    else {//遇上桶非空的情况则需要挂载新节点
        bool exist = false;
        Node* p = hashnode + hashT;
        for (p; p != NULL; p = p->next) {
```

```
                    if (p->checknum == checkhashT) {//如果用于校验的hash值也相等则认为字符串完全匹配，这一步实际上是有风险的
                        p->cnt++;
                        rankofright = p->cnt >= m ? k : rankofright;
                        exist = true;
                        break;//一旦匹配可以跳出循环
                    }
                }
                if (exist == false) {//若没有可匹配的则需要挂载新的节点
                    sup[pos].cnt++;
                    rankofright = sup[pos].cnt >= m ? k : rankofright;
                    sup[pos].checknum = checkhashT;
                    (sup + pos)->next = (hashnode + hashT)->next;
                    (hashnode + hashT)->next = sup + pos;
                    pos++;
                }
            }
        }
```

因此总体时间复杂度T(n)=T*(O(1)+O(n)*O(logn))=T*O(nlogn)=**O(nlogn)**;其中**n为字符串s的长度**

2.空间复杂度的估算

散列表的长度为O(n)，散列表中每个桶中存放了对应字符串出现的首字母的秩以及用来校验是否相等的checkhash值，每个桶的空间占用可视为常数，因此整个散列表的空间复杂度为O(n);

其余的辅助用的临时变量和全局变量视为O(1);

因此整体的空间复杂度=O(n)+O(1)=**O(n)**;

后缀数组法

1.时间复杂度估算，这里语义上约定n=strlen(s)，即n为字符串s的长度

定义变量和读入字符串的时间复杂度为O(1);

```
char str[40050];
int k;
scanf("%d %s", &k, str);
```

对字符串进行预处理，把每个小写字母转为对应的正整数(a~z分别对应1~26)，时间复杂度为**O(n)**;

```
int length = strlen(str);
preprocess(str, length);
```

使用倍增算法对后缀进行排序，由于字符集小，使用基数排序的方法，时间复杂度为**O(n)**;

```
da(length + 1, 27);
```

计算height数组，时间复杂度为**O(n)**;

```
calheight(length);
```

也是对长度进行二分查找，查找次数为**O(logn)**;

单独地对某一长度 x 进行判断的时候,其实就是对 height 数组做一次遍历，遍历过程中更新所需要的最长子串最靠右出现的信息。时间复杂度为 O(n);

```
bool workable2(int lengthT, int k, int x) {//对于子串要求至少出现 k 次的条件，校验子串长度为 x 时候是否符合要求
        bool exist = false;
        int cnt = 1;
        mostRightRank = -1;//初始化为-1，即校验失败的情况
        RightRank = sa[1];
        for (int i = 2; i <= lengthT; i++) {
            if (height[i] >= x) {
                cnt++;//sa[i]和 s[i-1]的最长公共前缀长度大于等于 x 时候，满足条件的子串数量加一
                RightRank = RightRank >= sa[i] ? RightRank : sa[i];//更新此时的指针位置，取较大值
                if (cnt >= k) {
                    exist = true;//一旦总数满足了要求则校验成功
                    mostRightRank = mostRightRank >= RightRank ? mostRightRank : RightRank;//更新最靠右的满足条件的指针位置
                }
            }
            else {
                cnt = 1;//lcp 的长度不满足要求时候说明前缀不满足要求，需要重新计数
                RightRank = sa[i];
            }
        }
        return exist;
    }
```

因此总体时间复杂度 T(n)=T*(O(1)+O(n)+O(n)+O(n)+O(n)*O(logn))=T*O(nlogn)=**O(nlogn)**; **其中 n 为字符串 s 的长度**

2.空间复杂度的估算

倍增算法计算后缀数组过程中的空间复杂度为 O(n),具体的是 **6*O(n)**,分别是 r[maxn],sa[maxn],wa[maxn],wb[maxn],wv[maxn],wt[maxn]；

计算 height 数组的空间复杂度为 O(n),具体为 **1*O(n)**;

其余的辅助用的临时变量和全局变量视为 O(1);

因此整体的空间复杂度=6*O(n)+O(n)+O(1)=**O(n)**;

四.理论分析与实测效果的吻合程度

采用后缀数组的方式时间复杂度相较于使用两个散列函数的方法在时间上会稍快一些。在代码中我采用的是对 T 项任务中的每一项都随机使用这两种方法之一，耗费的时间大致介于单独使用这两种方法之间。

Case No.	Result	Time(ms)	Memory(KB)
1	Correct	11	17008
2	Correct	10	17008
3	Correct	12	17008
4	Correct	9	17008
5	Correct	8	17008
6	Correct	9	17008
7	Correct	14	17008
8	Correct	15	17008
9	Correct	69	17008
10	Correct	72	17008
11	Correct	9	17008
12	Correct	9	17008
13	Correct	15	17008
14	Correct	11	17008
15	Correct	17	17008
16	Correct	12	17008
17	Correct	120	17008
18	Correct	121	17008

随机选择两种方法之一

Case No.	Result	Time(ms)	Memory(KB)
1	Correct	22	15756
2	Correct	18	15756
3	Correct	15	15756
4	Correct	15	15756
5	Correct	15	15756
6	Correct	20	15756
7	Correct	34	15756
8	Correct	34	15756
9	Correct	151	15756
10	Correct	154	15756
11	Correct	16	15756
12	Correct	14	15756
13	Correct	15	15756
14	Correct	15	15756
15	Correct	22	15756
16	Correct	21	15756
17	Correct	166	15756
18	Correct	254	15756

只使用散列

Case No.	Result	Time(ms)	Memory(KB)
1	Correct	10	15728
2	Correct	8	15728
3	Correct	7	15728
4	Correct	9	15728
5	Correct	10	15728
6	Correct	9	15728
7	Correct	11	15728
8	Correct	11	15728
9	Correct	45	15728
10	Correct	50	15728
11	Correct	8	15728
12	Correct	8	15728
13	Correct	8	15728
14	Correct	10	15728
15	Correct	7	15728
16	Correct	8	15728
17	Correct	135	15728
18	Correct	91	15728

只使用后缀数组