字符串相关算法

0.1 相关API 字母表

`class Alphabet` 对字符串的字母表存在限制，基于字母表的基数（即字母表字母的数量）R，以及很重要的toChar和toIndex方法，优点：方便对字符进行索引访问，提高效率。

1. 字符串排序

字符串的排序我们会学习两种完全不同的排序算法，LSD和MSD，即低位优先和高位优先，（从右到左 vs 从左到右）。

1.1 键索引计数法 （适用于键为较小的整数）

关键：是先计算每种键的频率，从而得到每种键的起始位置，进而遍历就能得到排序结果

实现：

```
// a是一个数组，包含值和键，依据键对其进行排序
int N = a.length;

String[] aux = new String[N];
int[] count = new int[R+1];

// 计算出现频率
for (int i = 0; i < N; i++) {
    count[a[i].key() + 1]++;
}
// 转化索引
for (int i = 0; i < N; i++) {
    count[i+1] += count[i];
}
// 将元素分类
for (int i = 0; i < N; i++) {
    aux[count[a[i].key()]++] = a[i];
}
// 写回
for (int i = 0; i < N; i++) {
    a[i] = aux[i];
}
```

分析：只要N和R合适，那么键索引计数法是一个线性时间级别的方法，他不需要比较。

1.2 LSD（低位优先）

处理定长字符串，利用键索引优先方法，从右到左以此对字符串进行排序，遍历数据W次就可以对整个数组实现排序

算法实现

```
public class LSD {
    public static void sort (String[] a,  int W) {
        int N = a.length;
```

```
        int R = 256;
        String[] aux = new String[N];

        for (int d = N-1; d >= 0; d--) {
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++) {
                count[a[i].charAt(d) + 1]++;
            }
            for (int i = 0; i < R; i++) {
                count[i+1] += count[i];
            }
            for (int i = 0; i < N; i++) {
                aux[count[a[i].charAt(d)]++] = a[i];
            }
            for (int i = 0; i < N; i++) {
                a[i] = aux[i];
            }
        }

    }
}
```

运行时间与WN成正比。

1.3 MSD（高位优先）

对于变长字符串的排序问题，从左到右对字符串就行排序，从首字母开始，然后递归的对子数组进行键索引排序，注意字符串结束的处理。

算法实现

```
// 考虑到要使用递归，因此在小规模子数组时使用插入排序
public class MSD {

}
```

1.4 三切分字符串排序

考虑到等值键对于高位优先排序性能的影响，因此提出三切分字符串排序，类似于三切分快速排序的思路。

1.5 总结

表 5.1.2　各种字符串排序算法的性能特点

算　法	是否稳定	原地排序	在将基于大小为 R 的字母表的 N 个字符串排序的过程中调用 charAt() 方法次数的增长数量级（平均长度为 w，最大长度为 W）		优势领域
			运行时间	额外空间	
字符串的插入排序	是	是	N 到 N^2 之间	1	小数组或是已经有序的数组
快速排序	否	是	$N\log^2 N$	$\log N$	通用排序算法，特别适合用于空间不足的情况
归并排序	是	否	$N\log^2 N$	N	稳定的通用排序算法
三向快速排序	否	是	N 到 $N\log N$ 之间	$\log N$	大量重复键
低位优先的字符串排序	是	否	NW	N	较短的定长字符串
高位优先的字符串排序	是	否	N 到 Nw 之间	$N+WR$	随机字符串
三向字符串快速排序	否	是	N 到 Nw 之间	$W+\log N$	通用排序算法，特别适合用于含有较长公共前缀的字符串

2. 单词查找树

单词查找树所需API：符号表

在单词查找树中，键是从根节点到含有非空值的节点的路径所隐式表示的。同样使用递归的方法来构造我们的单词查找树。

```
public class TrieST<Value>
{
    private static int R = 256;
    private Node root;

    private static class Node {
        private Object val;
        private Node[] next = new Node[R];
    }

    public Value get (String key) {
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val;
    }

    private Node get(Node x, String key, int d) {
        if (x == null) return null;
        if (d == key.length()) return x;
        char c = key.charAt(d);     // 找到第d个字符对应的单词查找树
        return get(x.next[c], key, d+1);
    }

    public void put(String key, Value val) {
        root = put(root, key, val, 0);
    }

    private Node put (Node x, String key, Value val, int d) {
```

```
        // 如果key存在以x为根节点的单词查找树中，则更新它
        if (x == null) return null;
        if (d == key.length()) {x.val = val; return x}
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }
}
```

遍历单词查找树的所有键的方法就是树的深度遍历方法，使用递归的思路。

2.1 单词查找树的性质

时间：logR N

空间：　所有键较短：连接总数接近R*N

　　所有键均较长：链接总数接近于R*N*w(w为键的平均长度)

因此缩小R可以节省大量空间

- 在单词查找树中查找或插入一个键时，访问数组的次数是最多的键长度加1.
-

2.2 三向单词查找树

考虑到单词查找树会出现长的单向分支，每个节点含有一个字符，三个连接和一个值。左连接小，中间连接相等，右连接大。因此查找的正确结果仍然是从中间的分支执行得到。

2.3 三项单词查找树的性质

- 查找成本：对应的单词查找树的查找成本乘以每个节点的二叉查找树所需的成本。

2.4 总结

表 5.2.3　各种字符串查找算法的性能特点

算法（数据结构）	处理由大小为 R 的字母表构造的 N 个字符串（平均长度为 w）的增长数量级		优　点
	未命中查找检查的字符数量	内存使用	
二叉树查找 (BST)	$c_1(\lg N)^2$	$64N$	适用于随机排列的键
2-3 树查找（红黑树）	$c_2(\lg N)^2$	$64N$	有性能保证
线性探测法（并行数组）	w	$32N$~$128N$	内置类型 缓存散列值
字典树查找（R 向单词查找树）	$\log_R N$	$(8R+56)N$~$(8R+56)Nw$	适用于较短的键和较小的字母表
字典树查找（三向单词查找树）	$1.39\lg N$	$64N$~$64Nw$	适用于非随机的键

3. 子字符串查找

3.1 暴力匹配算法

逻辑：即逐一对文本和模式进行比对，若不匹配则回退i指针和j指针，直到匹配成功或者匹配失败。

优点：实现简单

缺点：性能不一定好，没有充分利用已知信息，导致损失

3.2 KMP算法

利用已经匹配好的文本和模式的关系，来决定下一步匹配的位置

逻辑：不回退文本指针，只借助dfa数组回退模式指针。

3.2.1 DFA数组的构成

理解：**对于有限状态机的理解**，dfa[c][i]数组是对于第i个状态，遇到字符c时要跳转到的状态。其中一个重要的点是存储X来记录每个状态在匹配失败时跳转回的状态。重点是X一定是小于i的一个状态，所以初始化X为除了第一个字符所在的位置为1（表示匹配成功，前进一个），其余为0.

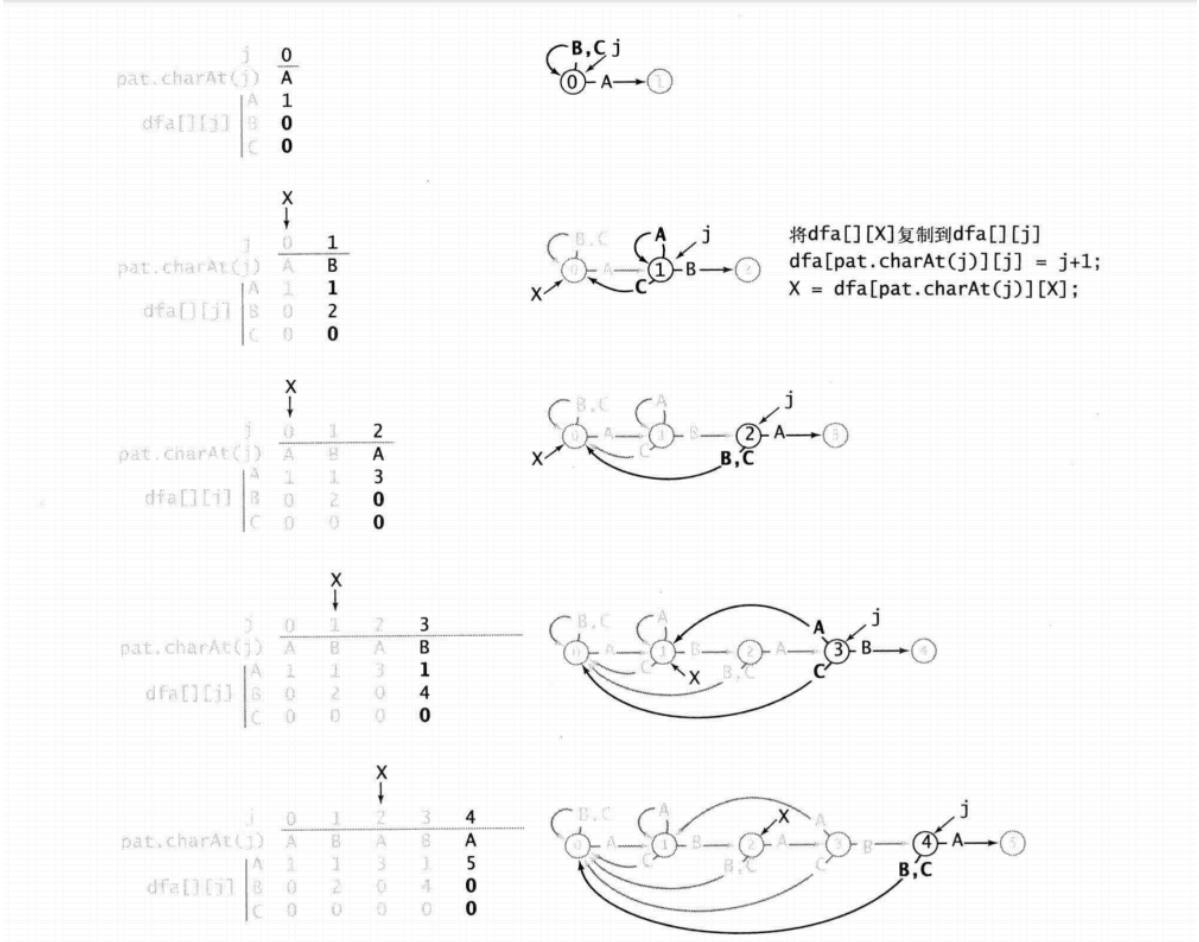
步骤:

1. 对于匹配失败的情况，跳转回X记录的状态
2. 对于匹配成功的情况，更新dfa[charAt(i)][i]为i+1
3. 更新X，为当前状态要跳转回的状态，即X = dfa[pat.charAt(i)][X]

算法实现

```
// R表示字母表基数
public void dfa_generate (int[][] dfa, String pat) {
    M = pat.length();
    dfa = new int[R][M];
    dfa[pat.charAt(0)][0] = 1;
    // 遍历整个模式串，实现对整个模式串的dfa数组构造，第一个状态已经初始化成功，因此从第二个状态开始
    for (int j = 1; j < M; j++) {
        // 匹配失败，跳转回X记录的位置
        for (int c = 0; c < R; c++) {
            dfa[c][j] = dfa[c][X];
        }
        // 匹配成功，对状态进行更新
        dfa[pat.charAt(j)][j] = j+1;
        // 更新X
        X = dfa[pat.charAt(j)][X];
    }
}
```

可视化展示



3.3 BM(Boyer-Moore)

与KMP算法不同，BM算法选择使用从右往左的字符串匹配顺序，因为如果匹配不成功，存在字符并没有大量重复的情况下，BM算法可以尽可能地跳过更多地距离，比如，跳过整个模式串地长度。

不同：BM算法除了移动模式串指针之外，还会移动文本指针。

算法实现：

```
public class BM {
    private String pat;
    private int[] right;
    private int R = 256;
    private int M;

    public BM (String pat) {
        this.pat = pat;
        right = new int[R];
        this.M = pat.length();
        for (int i = 0; i < R; i++) {
            right[i] = -1;
        }
        for (int i = 0; i < M; i++) {
            right[pat.charAt(i)] = i;
        }
    }

    public static int search (String txt) {
        int N = txt.length();
        int skip = 0;
```

```
        for (int i = 0; i < N - M; i += skip) {
            for (int j = M-1; j >= 0; j--) {
                if (pat.charAt(j) != txt.charAt(i+j)) {
                    skip = j - right[txt.charAt[j]];  // 应该使用txt的字符来确定右跳
数
                    if (skip < 1) skip = 1;
                    break;
                }
            }
            if (skip == 0) return i; // 找到匹配
        }
        return N;     // 未找到匹配
    }

    public static void main (String[] args) {

    }
}
```

3.4 Rabin-Karp指纹字符串查找算法（未完成）

使用散列表，哈希出的余数，对比哈希出的余数，余数相等则找到对应的字符串，其中分为两种具体算法，一种是拉斯维加斯算法，这种算法需要知道模式字符串，而改进之后的算法是蒙特卡洛算法，并不需要知道模式串，可以使用RM来去除第一个数的时的计算。

具体原理（回去认真了解）！！！！

3.5 总结

（续）

表 5.3.2　各种字符串查找算法的实现的成本总结

算　法	版　本	操作次数		在文本中回退	正确性	额外的空间需求
		最坏情况	一般情况			
暴力算法	—	MN	$1.1N$	是	是	1
Knuth-Morris-Pratt 算法	完整的 DFA（算法 5.6）	$2N$	$1.1N$	否	是	MR
	仅构造不匹配的状态转换	$3N$	$1.1N$	否	是	M
	完整版本	$3N$	N/M	是	是	R
Boyer-Moore 算法	启发式的查找不匹配的字符（算法 5.7）	MN	N/M	是	是	R
Rabin-Karp 算法*	蒙特卡洛算法（算法 5.8）	$7N$	$7N$	否	是*	1
	拉斯维加斯算法	$7N$*	$7N$	是	是	1

* 概率保证，需要使用均匀和独立的散列函数。

9

4. 正则表达式

4.1 使用正则表达式表示模式

模式的描述由3种基本操作和作为操作数的字符组成。

- 连接操作
- 或操作（|）
- 闭包操作（*）：表示多个（包括0个）该字符组成

4.2 NFA与DFA

正则表达式因为模式串的不确定，所以使用NFA即不确定有限状态机。

4.3 NFA的使用

4.4 NFA的构造

算法实现:

```
public class NFA {
    private char[] re;       // 匹配转换
    private Diagraph G;      // epsilon转换
    private int M;           //状态数量

    public NFA (String regexp) {
        // 根据正则表达式构造其非确定状态有限机
        Stack <Integer> ops = new Stack<Integer>();
        re = regexp.toCharArray();
        M = re.length();

        for (int i = 0; i < M; i++) {
            int lp = i;
            if (re[i] == '(' || re[i] == '|')
                ops.push(i);
            else if (re[i] == ')') {
                int or = ops.pop();
                if (re[or] == '|') {
                    lp = ops.pop();
                    G.addEdge(lp, or+1);
                    G.addEdge(or, i);
                }
                else {
                    lp = or;
                }
            }
            if (i < M - 1 && re[i+1] == '*') {
                G.addEdge(lp, i+1);
                G.addEdge(i+1, lp);
            }
            if (re[i] == '(' || re[i] == '*' || re[i] == ')') {
                G.addEdge(i, i+1);
            }
            }
        }
    }

    public boolean recognizes (String txt) {
        // NFA是否可以识别文本txt
        Bag<Integer> pc = new Bag<Integer>();
        DirectedDFS dfs = new DirectDFS(G, 0);
        for (int v = 0; v < G.v(); v++) {
            if (dfs.marked(v)) pc.add(v);
        }
```

```
        for (int i = 0; i < txt.length(); i++) {
            // 计算txt[i+1]可能到达的所有NFA状态
            Bag<Integer> match = new Bag<Integer>();
            for (int v:pc) {
                if (v < M) {
                    if (re[v] == txt.charAt[i]  || re[v] == '.') {
                        match.add(v+1);
                    }
                }
            }
            pc = new Bag<Integer>();
            dfs = new DirectedDFS(G, match);
            for (int v = 0; v < G.v(); v++) {
                if (dfs.marked(v)) pc.add(v);
            }
            for (int v:pc) if (v == M) return true;
            return false;
        }
    }
}
```

5. 数据压缩

比特流或者字节流进行压缩以及展开。

5.1 比特流的输入输出

- BinaryStdIn
- BinaryStdOut