字符串常用技巧黑盒专题

BKDR hash

- 哈希函数定义

$$\sum magic^i * s_i$$

- 一般magic取一个大素数如1e9+7
- 然后使用unsigned long long自然溢出
- 无符号自然溢出的情况下，加减乘只要本来数值是一样的，最后得到的值都是一样的。

BKDR hash实现

```
const unsigned long long magic = 1000000007;
unsigned long long power[N];
void gen(unsigned long long hash[], char s[]) {
    int n = strlen(s);
    hash[0] = s[0];
    power[0] = 1;
    for (int i = 1; i < n; i++) {
        hash[i] = hash[i - 1] * magic + s[i];
        power[i] = power[i - 1] * magic;
    }
}
unsigned long long get(int l, int r) {
    return hash[r] - (l == 0 ? 0 : hash[l - 1] * power[r - l + 1]);
}
```

BKDR hash实现

```
const unsigned long long magic = 1000000007;
unsigned long long power[N];
void gen(unsigned long long hash[], char s[]) {
    int n = strlen(s);
    hash[0] = s[0];
    power[0] = 1;
    for (int i = 1; i < n; i++) {
        hash[i] = hash[i - 1] * magic + s[i];
        power[i] = power[i - 1] * magic;
    }
}
unsigned long long get(int l, int r) {
    return hash[r] - (l == 0 ? 0 : hash[l - 1] * power[r - l + 1]);
}
```

123456789
1234567
1234000
567

hash的作用

- 判断两个串是否相等

hash的作用

- 判断两个串是否相等
- 有时候可以直接放进set里，判断有没有一样的元素

KMP

- KMP一般用于解决形如在：“串B中找串A出现的位置”一类的字符串匹配问题。
- 比如：找“abc”出现在“dabcdabcd”的那些位置

KMP

- KMP一般用于解决形如在：“串B中找串A出现的位置”一类的字符串匹配问题。
- 比如：找“abc”出现在“dabcdabcd”的那些位置
- 但是这么裸的题基本已经绝种了……

KMP

- KMP的过程大致如下：
- 先对串A[1..n]求一个next数组，这个next数组的定义为:

$$next_i = \max\{len\} \ (A_{1..len} = A_{i-len+1..i})$$

KMP

- KMP的过程大致如下：
- 先对串A[1..n]求一个next数组，这个next数组的定义为:

$$next_i = \max\{len\}\ \ (A_{1..len} = A_{i-len+1..i})$$

- 然后对串B的每个位置i，求：

$$res_i = \max\{len\}\ \ (A_{1..len} = B_{i-len+1..i})$$

KMP

- KMP的过程大致如下：
- 先对串A[1..n]求一个next数组，这个next数组的定义为:

$$next_i = \max\{len\} \ (A_{1..len} = A_{i-len+1..i})$$

- 然后对串B的每个位置i，求：

$$res_i = \max\{len\} \ (A_{1..len} = B_{i-len+1..i})$$

- 显然，当这个值等于|A|时，就是匹配上

KMP

- 求解过程
- 假装已经算好了next，在做第二步。
- 那么假设res_i(记为x)已经算好，现在要算res_{i+1}(记为y)。
  - 如果$A_{x+1} = B_{i+1}$，那么$y = x + 1$。
  - 否则不匹配，令$x = next_x$，继续尝试匹配。
  - 关键在于，如果不匹配，当前知道的信息是

$$A_{1..len} = B_{i-len+1..i}$$

- next的计算同理

KMP一些常见的考点

- 对next数组以及求解过程的理解(内功)

KMP一些常见的考点

- 对next数组以及求解过程的理解(内功)
- 对非字符串的匹配使用

Problem1 hdu4749

- 抽象模型以后，题目变成这样：
  - 定义两个数字序列A和B相等为：
    - |A|=|B|
    - 定义函数f(A)返回一个与A等长的新的序列C
      - C[i] = count(j) A[j] < A[i]   1 <= i, j <= |A|
    - 则f(A)=f(B)
  - 寻找A[]在B[]的连续子序列中出现的位置

Problem1 hdu4749

- 抽象模型以后，题目变成这样：
  - 定义两个数字序列A和B相等为：
    - |A|=|B|
    - 定义函数f(A)返回一个与A等长的新的序列C
      - C[i] = count(j) A[j] < A[i]   1 <= i, j <= |A|
    - 则f(A)=f(B)
  - 寻找A[]在B[]的连续子序列中出现的位置
- A = [1, 2, 3],   B = [2, 2, 3, 7, 5, 6, 9]
- [2, 3, 7]和A匹配，[5, 6, 9]也和A匹配。
- f([1, 2, 3]) = 0, 1, 2   f([2, 3, 7]) = 0, 1, 2

Problem1 hdu4749

- 先观察这个模型具有的性质
  - 传递性。A~B，B~C，则A~C(显然，因为f(A)=f(B)=f(C))
  - 前缀匹配性质。
    - 若 $A_{1..n} \approx B_{1..n}$
    - 则 $A_{1..i} \approx B_{1..i} \;\; for \;\; i \leq n$

Problem1 hdu4749

- 回到KMP，每次询问的就是已知

$$A_{1..i} \approx B_{1..i}$$

- 想要知道是否有

$$A_{1..i+1} \approx B_{1..i+1}$$

- 在这里，实际上是偏序集合的匹配。想要判断加了一个数字以后两个序列是否仍旧匹配，只需要新加的字符在自身字符串中，排的位置是多少。于是统计小于与等于自己的数字数目即可。

Problem1 hdu4749

- 这个统计用树状数组就可以完成。于是KMP的每次比较是lg k的，总体复杂度是O(n lg k)的。
- 实现：http://edward-mj.com/archives/1632

扩展KMP

- KMP求解的目标

$$res_i = \max\{len\}\ \ (A_{1..len} = B_{i-len+1..i})$$

- 扩展KMP求解的目标

$$res_i = \max\{len\}\ \ (A_{1..len} = B_{i..i+len-1})$$

- 求解过程在此略过，但求字符串循环最小表示的算法与扩展KMP的求解过程类似，也是有可能用到的。

Probleam2 BNU34990

- 在B里找A出现的位置，匹配时允许至多两个字符不同。
- $1 \le |A|, |B| \le 10^5$

Probleam2 BNU34990

- 枚举匹配上的位置，正反求lcp，中间再看是否相同。

- 因为这个lcp都是和A串的首尾求的，所以只要正反做一次扩展KMP，就能求得lcp，中间部分hash就可以了。

Problem3 ZOJ3199

- 给定一个串A，假定它匹配上*ss*，s的最长长度是多少？
- dabcabce

Problem3 ZOJ3199

- 分治+扩展KMP
- 大致思路是每次把串分成两半，然后处理跨越中间的答案。然后递归下去。
- 跨越中间的答案最后会化归为以中间这个位置为开头求正反的lcp，同样的道理，每次处理使用扩展KMP。
- 那么T(n) = 2T(n/2) + O(n)
- 所以复杂度是O(n lg n)的
- http://edward-mj.com/archives/1907

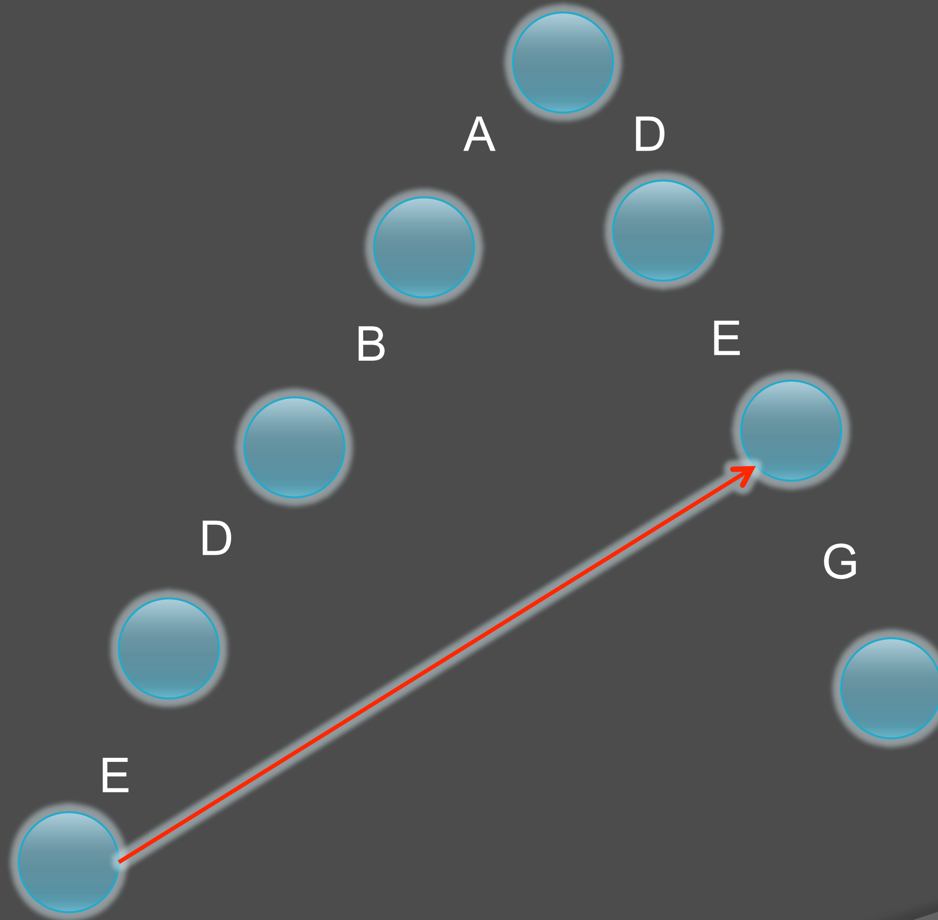
AC自动机

- KMP的多串版，这个太常见。
- 建立过程略过。
- 一个trie，上面多了一个next指针。
- KMP的next定义：

$$next_i = \max\{len\} \ (A_{1..len} = A_{i-len+1..i})$$

- 这里的next其实一样

AC自动机



AC自动机

- 随便搜一搜例题吧。

后缀树

- 给一个串$A_{1..n}$，定义$S_i=A_{i..n}$,把所有S_i插入到一棵trie树中，得到的就是后缀树。
- 这样中间有很多结点只会有一个儿子，显然可以压缩。
- 压缩以后结点数数O(n)的

后缀数组

- 给一个串$A_{1..n}$，定义$S_i=A_{i..n}$，那么对S_i按字典序排序，得到的就是后缀数组。
- abac
- {abac, bac, ac, c}
- {abac, ac, bac, c}记为B
- 每个串显然没有必要用string存，存它从原串哪个位置开始就可以了

后缀数组

- 重要工具：height

$$height_i = lcp(B_i, B_{i-1})$$

- 重要性质：

$$lcp(B_i, B_j) = \min_{i<k\le j} height_k$$

- 于是建立ST表可以O(1)求两个后缀的lcp

后缀数组

- 实际上，后缀数组可以看成对后缀树先序遍历依次拿出来的结果。
- lcp对应着后缀树中的lca

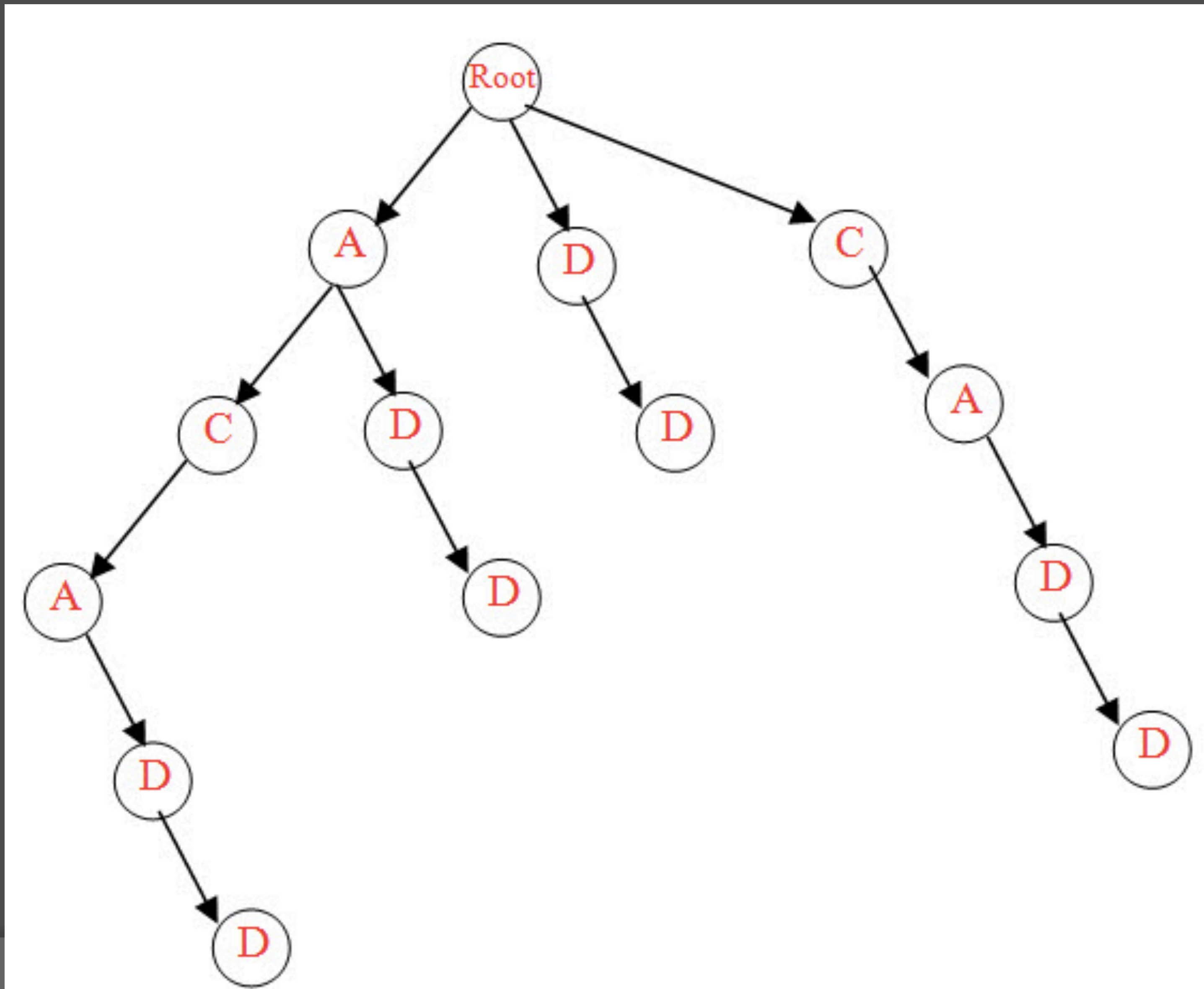
练习

- 还是ZOJ3199
- 给串A去匹配*ss*，问s最长能是多少。

后缀自动机

- 实际上，不压缩的后缀树就可以是一个后缀自动机，只需要把那个描述后缀的结点标为终结状态。

后缀自动机



后缀自动机

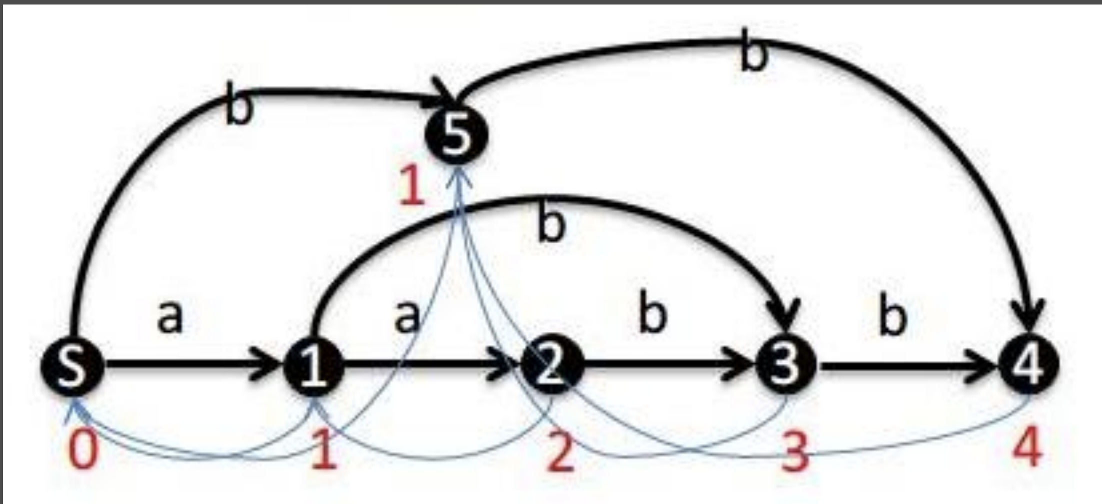
- 先来讲讲组成一个自动机的三个要素
  - 起始状态
  - 转移函数
  - 终结状态
- 后缀自动机
  - 从起始点一直跑，如果现在跑到终结状态，那么路上的字符拼成了一个后缀

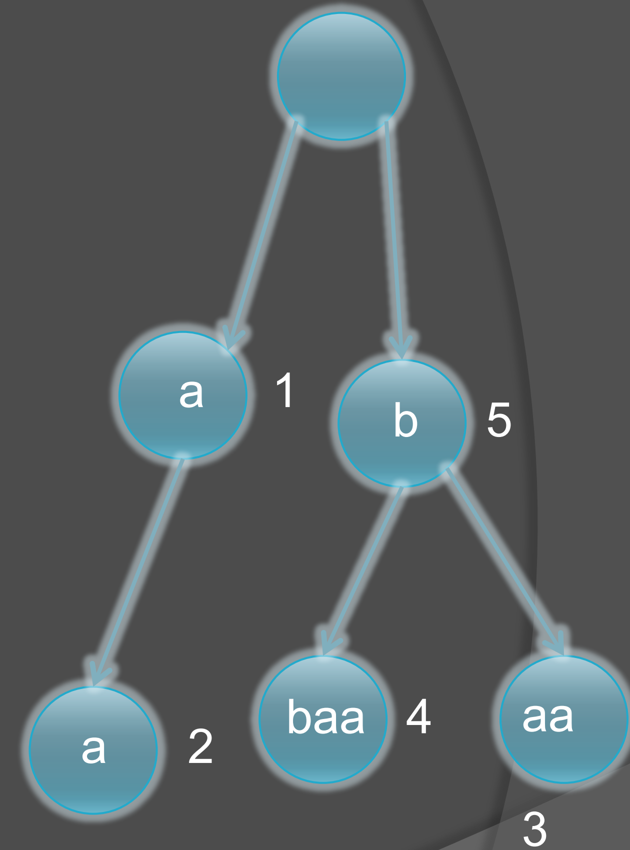
后缀自动机

- 重要参数
- root —— 起始状态
- val —— 能跑到这个状态的最长长度
- fa —— 逆串后缀树的father
- last —— 其中一个终结状态，其它终结状态由last一直取fa跳到根的路径组成

后缀自动机

- aabb reverse：bbaa





SRM621 900pt

- 给定一个串S，然后你统计它的magic substring的数目
  - magic substring定义为在串中不重叠地出现至少两次的串
- 比如”aaaabb”，符合的substring为{”a”, ”b”, ”aa”}，答案为3。

SRM621 900pt

- 先建立后缀自动机，这时候相当于有了一棵后缀树。
- 按照后缀树的思路去想，遍历一遍这棵树，那么就得到所有子串。
- 对于每个子串，只要判定它最早出现的位置和最晚出现的位置与它自己本身的长度比较哪个比较大，就可以知道是否符合答案。

SRM621 900pt

- 但是在后缀树中有些点是做了合并的。
- 能够利用的信息就是上面的val值。
- 于是这个结点对答案的影响就是

$$max(0, min(p.val, p.right - p.left) - p.fa.val)$$