# 算法分析与设计基础 第十一周作业

徐浩博 软件02 2020010108

**Problem 1**

**a.**

```
FIB_HEAP_CHANGE_KEY(H, x, k):
    if k == x.key:
        return
    if k < y.key:
        FIP-HEAP-DECREASE-KEY(H, x, k)
        return
    FIP-HEAP-DELETE(H, x)
    FIP-HEAP-INSERT(H, k)
```

若修改的值比原值小，则直接调用FIP-HEAP-DECREASE-KEY，由之前讨论，删除节点的摊还时间为O(lgn). 若修改的值等于原值，则不修改，复杂度O(1). 若修改的值大于原值，则将该节点删去，然后再插入要修改的新值；删除的摊还时间为O(lgn)，插入摊还时间为O(1)，总共摊还时间为O(lgn)+O(1)=O(lgn).

**b.**

```
FIB_HEAP_PRUNE(H, r):
    if r > H.n:
        r = H.n
    for i from 1 to r:
        x = leaf_list->head
        if x == H.min && H.n != 1:
            x = another element of leaf_list
        y = x.p
        if y != NUL:
            remove x from the leaf list
            remove x from the child list of y
            decrement y.degree
            if y.degree == 0:
                insert y into the leaf list
            CASCADING-CUT-2.0(H, y)
        else:
            remove x from the root list
```

```
CASCADING-CUT-2.0(H, y):
    z = y.p
    if z!=NULL:
        if y.mark == FALSE:
            y.mark = TRUE
        else:
            CUT(H, y, z)
            if z.degree == 0:
                insert z into the leaf list
            CASCADING-CUT(H, z)
```

我们建立一个叶子结点的链表，该链表中只含H的叶子结点，每次删点时从该链表中删q个，并且执行CUT操作保证每个节点mark的正确性，从而不改变H的其他性质. 注意的是，这里的CUT进行了改进：如果儿子节点被删空，那么也要将该点插入叶子结点链表.

下面我们来分析均摊复杂度，我们修改势函数为$\Phi(H) = t(H) + 2m(H) + 3H.n$，即添加一项H的size，可以发现，在以上所有已实现操作中，改变size的数量最多1个，因此均摊复杂度的改变是O(1)的，并不会影响之前的所有操作的均摊时间. 我们来分析PRUNE的均摊时间. 对于删除节点，我们假设级联CUT操作使得根的数量增加c个，则被标记的最多有m(H)-c+q(清除c个标记，有可能q次调用时均增加了1个标记)，而H.n又减小了q，因此势能函数最多变化了：

$$((t(H) + c) + 2(m(H) - c + q) + 3(H.n - q)) - (t(H) + 2m(H) + 3H.n) = -c - q$$

下面考虑操作的时间开销：首先级联CUT操作c次的时间开销为O(c)，其次，从叶子节点找寻节点并删去q次的时间开销为O(q)，综上，时间开销为O(c+q). 均摊时间为O(c+q)-c-q=O(1)，因为能将势能函数的常数提高到到O(c)和O(q)的常数水平.

**Problem 2**

```
COMPUTE-TRANSITION-FUNCTION(p, Σ):
    m = p.length
    let elements in π[] is equal to 0
    π[0] = −1;
    k = −1;
    for i = 1 to m − 1:
        while k >= 0 and p[i] != p[k + 1]:
            k = π[k]
        if p[i] == p[k + 1]:
            k = k + 1
            π[i] = k
```

```
12      for each character a in Σ:
13          δ(0, a) = 0
14      for i = 0 to m:
15          if i > 0:
16              for each character a in Σ:
17                  δ(i, a) = δ(π[i − 1] + 1)
18          if i < m:
19              δ(i, p[i]) = i + 1
20              // no matter δ(i, p[i]) exists or not, let δ(i, p[i]) = i + 1
```

首先类似于KMP先求出$\pi$数组，可以发现对于$a \neq p[i]$有$\delta$(i, a) = $\delta(\pi$[i - 1] + 1)，而对于$a = p[i]$有$\delta$(i, a) = i+1，结合这两点即可快速算出$\delta$. 6-11行是KMP算法的预处理，已知其复杂度为O(m)，14-19行外循环(m+1)次，内嵌套对$\Sigma$的循环，所以复杂度为$O(m|\Sigma|)$. 整体复杂度为$O(m|\Sigma|)$.

# 各种字符串匹配时间开销比较实验报告

**摘要**

字符串匹配算法是在文本串中找寻某个模式串的算法，该算法能够极大提高编辑文本程序时搜索某一段字符串的响应效率. 一般来讲，朴素的算法复杂度是O(nm)，而KMP、BM等算法能够降低复杂度至线性. 本实验中我们将实现这几种算法，同时对其运行效率进行比较.

**关键词：**

## 1 实验环境

操作系统：Windows 10
编译器：g++
处理器：Intel Core i7-10750H 六核CPU @ 2.60GHz
编程语言：C++11

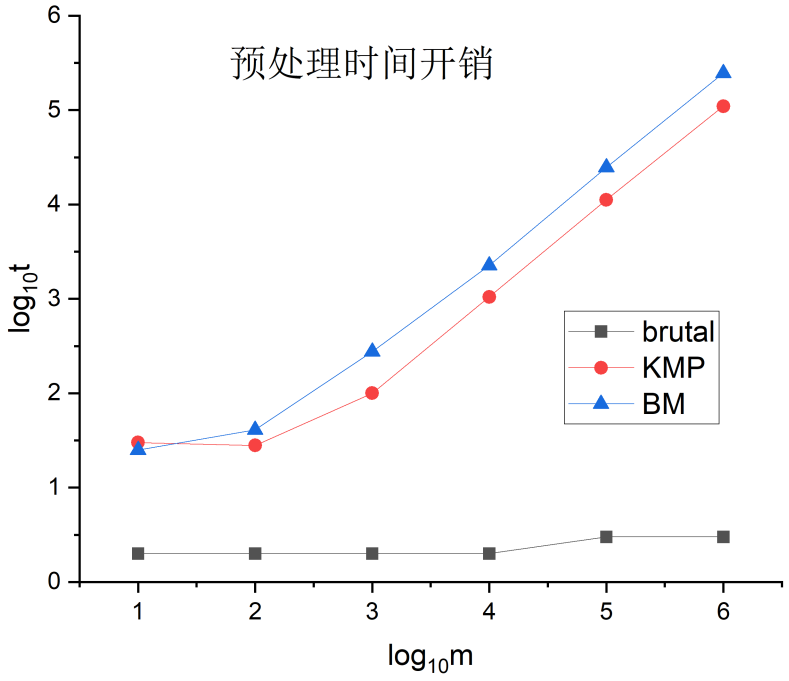
## 2 实验设计思路

我们分别测试预处理时间和匹配时间，于是我们先固定文本串，改变模式串的长度进行测量；之后我们固定模式串，改变文本串的长度进行测量.

## 3 结果分析

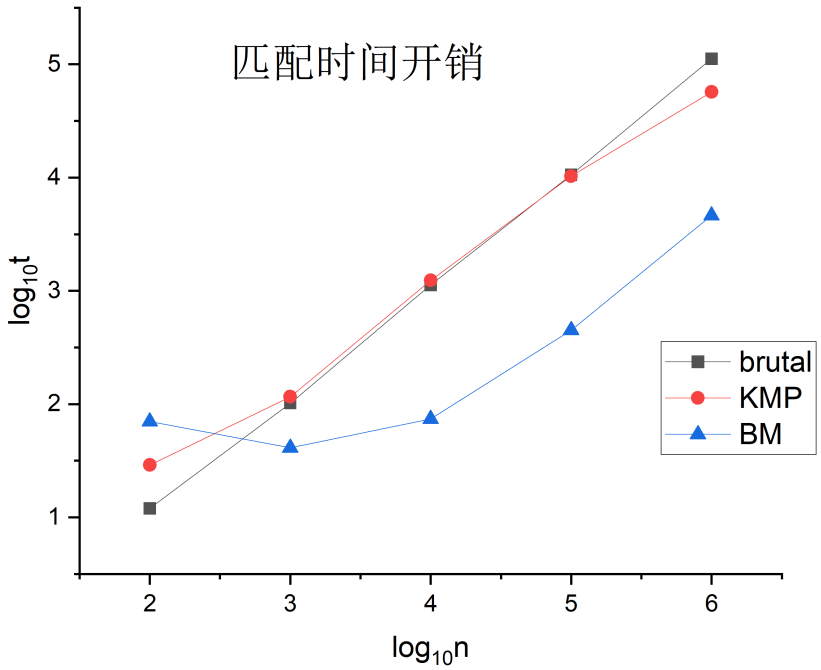
固定文本串，改变模式串长度m时的结果（单位：微秒）：

| m | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|
| brutal | 2 | 2 | 2 | 2 | 3 | 3 |
| KMP | 30 | 28 | 110 | 1049 | 11214 | 109684 |
| BM | 25 | 41 | 275 | 2265 | 24792 | 245831 |




通过图表我们能够看出brutal的预处理是O(1)的，其余两种预处理均是线性O(m)的. 固定模式串m=100，改变文本串n时的结果（单位：微秒）：

| m | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|
| brutal | 12 | 102 | 1125 | 10570 | 111743 |
| KMP | 29 | 116 | 1236 | 10297 | 57012 |
| BM | 70 | 41 | 74 | 450 | 4621 |




我们能够看到BM显著快于KMP，而KMP比brutal略快，在模式串随机的情况下，KMP时间开销近乎与brutal是一个量级的. 然而我们看到，几种算法的运行时间大致都是线性增长的.

首先，KMP算法比brutal快是很好理解的，但运行效率并没有获得很大提升，这大概与我们的模板串是随机生成的有关：模板串随机生成，内部的自相似性很差，因此$\pi$数组并没有发挥很大的skip优势，加之预处理还需要一定的时间，导致KMP并没有比brutal显示出很大优势.

其次，BM显著快于这两种算法，原因是我们的模板串相似性差，因此BM跳过时几乎可以跳过整个模板串，复杂度金思维O(n/m)，所以会显著快于前两种算法.

模板串内部相似性提高时，我们也许能够看到KMP效率获得显著的提升.

## 4 总结

我们对比了三种字符串算法，能够看到BM算法获得了最高的运行效率，而随机模板串下，KMP相比于原始方法并没有获得显著的效率上的提升，这可能与模板串内部相似性不高，$\pi$数组没有发挥优势有关. 然而在实际应用场景下，如文本搜索，模板串自相似性想必也不会很大，因此在KMP效率没有显著改进的情况下，我们可以采用BM或者改进版的BM算法以获得效率优势.