hw12

> 32.1-2，32.1-4；32.2-3；32.3-5；32.4-1，32.4-5；

32.1-2

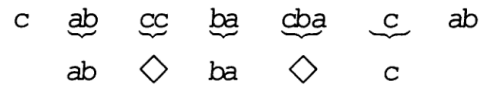
32.1-2 假设在模式 P 中所有字符都不相同。试说明如何对一段 n 个字符的文本 T 加速过程 NAIVE-STRING-MATCHER 的执行速度，使其运行时间达到 $O(n)$。

我们知道，P在T中的一个出现不能与另一个重叠，所以我们不需要像朴素算法那样进行重复检查。如果我们在文本中找到P的前k个字符与之匹配后跟随一个不匹配的字符，我们可以将s增加k而不是1。可以通过以下方式进行修改：

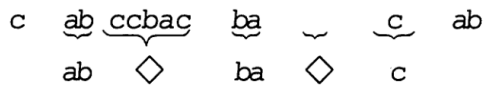
```
n = T.length
m = P.length
k = 0
s = 0

while s ≤ n − m do
    i = 1
    if T[s] == P[1] then
        k = s
        i = 0
        while T[k + i] == P[i] and i < m do
            i = i + 1
        end while
        if i == m then
            Print “Pattern occurs with shift” k
        end if
    end if
    s = s + i
end while
```

32.1-4

32.1-4 假设允许模式 P 中包含一个**间隔符**◇，它可以和任意字符串匹配(甚至可以和长度为 0 的字符串匹配)。例如，模式 ab◇ba◇c 在文本 cabccbacbacab 中的出现为



和



注意，间隔符可以在模式中出现任意次，但是不能在文本中出现。给出一个多项式时间算法，以确定这样的模式 P 是否在给定的文本 T 中出现，并分析算法的运行时间。

我们可以将包含 $g-1$ 个间隔字符的模式分解成形式 $a_1 \diamond a_2 \diamond \cdots \diamond a_g$。由于我们只关心模式是否出现在某个位置，因此只需寻找 a_1 的第一个出现，然后是紧跟着 a_1 之后任何位置的 a_2 的第一个出现，依此类推。如果模式 P 的长度为 m，文本的长度为 n，那么朴素策略的运行时间是 $O(nm)$。

在这种策略中，对于模式中的每个部分 a_i，都需要在整个文本中进行搜索，以找到其出现的位置。由于这种搜索是连续进行的，并且每次搜索可能需要检查整个文本，所以整个过程的时间复杂度是文本长度与模式长度的乘积，即 $O(nm)$.

32.2-3

32.2-3 试说明如何扩展 Rabin-Karp 算法用于处理以下问题：在一个 $n \times n$ 的二维字符数组中搜索一个给定的 $m \times m$ 的模式。(该模式可以在水平方向和垂直方向移动，但是不可以旋转。)

在这种方法中，我们使用维护模式的哈希值并计算文本的运行哈希值的相同思想。然而，每一步更新哈希的时间可能长达 $\Theta(m)$，因为进入和离开哈希窗口的条目数量是 $2m$，并且在它们进入和离开时至少需要检查所有这些条目。这将导致总的预期运行时间（不发生太多错误命中）是 $(n-m+1)^2 \cdot m$，在最坏情况下与朴素算法相同，即 $(n-m+1)^2 \cdot m^2$。

为了计算这个哈希，我们将为窗口中的每个条目赋予一个与其位置唯一对应的 d 的幂次。位于第 i 行第 j 列的条目将被乘以 d^{m^2-mi+j}。然后，在向右移动时，我们将哈希值乘以 d，减去左列中的缩放条目，并加入右列中的条目，这些条目也根据它们所在的行进行适当的缩放。向上或向下移动窗口也是类似的处理。同样，所有这些运算都是在某个大质数下取模进行的。

32.3-5

32.3-5 给定一个包括间隔字符(参见练习 32.1-4)的模式 P，说明如何构造一个有限自动机，使其在 $O(n)$的时间内找出 P 在文本 T 中的一次出现位置，其中 $n=|T|$。

为了创建能够处理间隔字符的确定性有限自动机，可以构建一个具有 $|P|+1$ 个状态的DFA。设 m 为间隔字符的数量。假设在模式 P 中所有间隔字符的位置由 g_i 给出，并且设 $g_0=0, g_m=|P|+1$。称间隔字符 i 之后但在间隔字符 $i+1$ 之前出现的模式段为 P_i。然后，可以想象按顺序尝试匹配这些模式，但如果在匹配某个特定模式时遇到困难，那么不能撤销在匹配早期模式时取得的成功。

更具体地，假设 $(Q_i, q_{i,0}, A_i, \Sigma_i, \delta_i)$ 是与模式 P_i 对应的DFA。那么，可以构造DFA，使得 $Q=\bigsqcup_i Q_i$，$q_0=q_{0,0}$，$A=A_{m+1}$，$\Sigma=\bigcup_i \Sigma_i$，并且 δ 如下描述。如果处于状态 $q\in Q_i$ 并看到字符 a，如果 $q\notin A_i$，就按照 $\delta_i(q,a)$ 规定的状态进行转换。然而，如果 $q\in A_i$，那么 $\delta(q,a)=\delta_{i+1}(q_{i+1,0},a)$。这种构造实现了上述英文描述的功能。

通过这种方式，DFA能够顺序地检查每个 P_i，并在成功匹配完一个段后移动到下一个段的匹配过程。这允许DFA处理间隔字符，并在匹配模式时保持灵活性。

32.4-1

32.4-1 计算对应于模式 ababbabbabbababbabb 的前缀函数 π。

i	π(i)
1	0
2	0
3	1
4	2
5	0
6	1
7	2
8	0

i	π(i)
9	1
10	2
11	0
12	1
13	2
14	3
15	4
16	5
17	6
18	7
19	8

32.4-5

32.4-5 用势函数证明 KMP-MATCHER 的运行时间是 $\Theta(n)$。

每当我们执行第7行时，就会至少将q减少1。由于唯一增加q值的地方是在第9行，而且只增加1，因此这些运行第5行的操作都是通过运行第9行来支付的。

以此为动机，我们让势能函数与q的值成正比。这意味着当我们执行第9行时，我们支付一个恒定的金额来提升势能函数。而当我们运行第7行时，我们减少势能函数，这将减少第6行while循环迭代的平摊成本，使其达到零平摊成本。我们改变q值的唯一其他时间是在第12行，而这只在执行外层for循环时每次运行一次，而且在那里的平摊也对我们有利。

由于在这个势能函数下，外层for循环每次迭代的平摊成本是恒定的，而该循环运行n次，算法的总成本是$\Theta(n)$。