Kmp:

KMP 算法的工作原理

- 该算法会在模式串中寻找被称为 LPS（Longest Prefix which is also Suffix，最长前缀后缀）的重复子串，并将这些 LPS 信息存储在一个数组中。
- 算法从左到右逐个比较字符。
- 当发生不匹配时，算法使用一个预处理好的表（称为"前缀表"）来跳过字符比较。
- 算法预先计算一个前缀函数，帮助确定每次发生不匹配时可以在模式串中跳过多少字符。
- 相比暴力搜索方法，KMP 算法利用先前比较的信息，避免了不必要的字符比较，从而提高了效率。

KMP 算法的优势

- KMP 算法可以高效地帮助你在大量文本中找到特定的模式串。
- KMP 算法可以使你的文本编辑任务更快、更高效。
- KMP 算法保证了100% **的可靠性**。

"""""

compute_lps 函数用于计算模式字符串的 LPS 表。LPS 表是一个数组，
其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。
该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。
"""

```
def compute_lps(pattern):
    """
    计算 pattern 字符串的最长前缀后缀（Longest Proper Prefix which is also Suffix）表
    :param pattern: 模式字符串
    :return: lps 表
    """

    m = len(pattern)
    lps = [0] * m  # 初始化 lps 数组
    length = 0  # 当前最长前后缀长度
    for i in range(1, m):  # 注意 i 从 1 开始，lps[0]永远是 0
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]  # 回退到上一个有效前后缀长度
        if pattern[i] == pattern[length]:
            length += 1
        lps[i] = length

    return lps

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return 0
    lps = compute_lps(pattern)
    matches = []
```

```
    # 在 text 中查找 pattern
    j = 0  # 模式串指针
    for i in range(n):  # 主串指针
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]  # 模式串回退
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)  # 匹配成功
            j = lps[j - 1]  # 查找下一个匹配

    return matches


text = "ABABABABCABABABABCABABABABC"
pattern = "ABABCABAB"
index = kmp_search(text, pattern)
print("pos matched：", index)
# pos matched：  [4, 13]



'''
这是一个字符串匹配问题，通常使用 KMP 算法（Knuth-Morris-Pratt 算法）来解决。
使用了 Knuth-Morris-Pratt 算法来寻找字符串的所有前缀，并检查它们是否由重复的子串组成，
如果是的话，就打印出前缀的长度和最大重复次数。
'''

# 得到字符串 s 的前缀值列表
def kmp_next(s):
    # kmp 算法计算最长相等前后缀
    next = [0] * len(s)
    j = 0
    for i in range(1, len(s)):
        while s[i] != s[j] and j > 0:
            j = next[j - 1]
        if s[i] == s[j]:
            j += 1
        next[i] = j
    return next
```

```
def main():
    case = 0
    while True:
        n = int(input().strip())
        if n == 0:
            break
        s = input().strip()
        case += 1
        print("Test case #{}".format(case))
        next = kmp_next(s)
        for i in range(2, len(s) + 1):
            k = i - next[i - 1]          # 可能的重复子串的长度
            if (i % k == 0) and i // k > 1:
                print(i, i // k)
        print()


if __name__ == "__main__":
    main()
```

强联通单元：

以下是计算强连通单元的算法。
(1) 对图G调用dfs，以计算每一个顶点的结束时间。
(2) 计算图G^T。
(3) 对图G^T调用dfs，但是在主循环中，按照结束时间的递减顺序访问顶点。
(4) 第3步得到的深度优先森林中的每一棵树都是一个强连通单元。输出每一棵树中的顶点的id。

Kosaraju算法是一种用于在有向图中寻找强连通分量（Strongly Connected Components，SCC）的算法。它基于深度优先搜索（DFS）和图的转置操作。

Kosaraju算法的核心思想就是两次深度优先搜索（DFS）。

1. **第一次DFS**：在第一次DFS中，我们对图进行标准的深度优先搜索，但是在此过程中，我们记录下顶点完成搜索的顺序。这一步的目的是为了找出每个顶点的完成时间（即结束时间）。
2. **反向图**：接下来，我们对原图取反，即将所有的边方向反转，得到反向图。
3. **第二次DFS**：在第二次DFS中，我们按照第一步中记录的顶点完成时间的逆序，对反向图进行DFS。这样，我们将找出反向图中的强连通分量。

Kosaraju算法的关键在于第二次DFS的顺序，它保证了在DFS的过程中，我们能够优先访问到整个图中的强连通分量。因此，Kosaraju算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。

以下是Kosaraju算法的Python实现，使用stack模拟按照结束时间的递减顺序访问顶点。

```
def dfs1(graph, node, visited, stack):
```

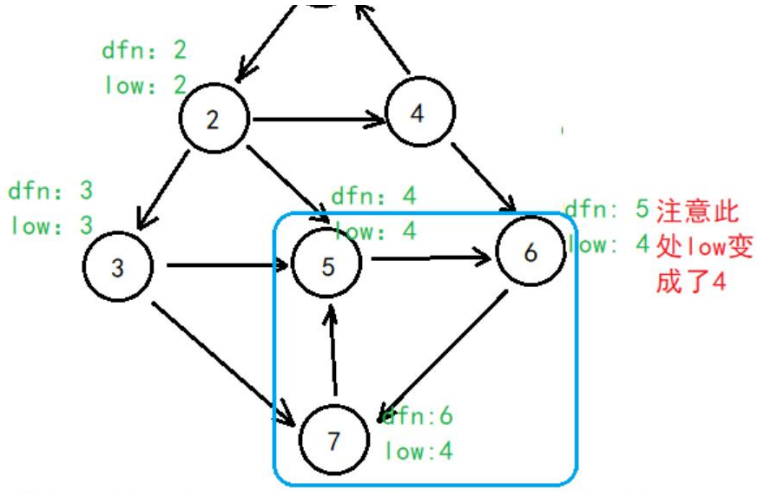


```
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[0, 3, 2, 1]
```

算法 2 ：

在没有遇到强连通分量前，按照正常dfs进行搜点

stack:1 2 3 5 6

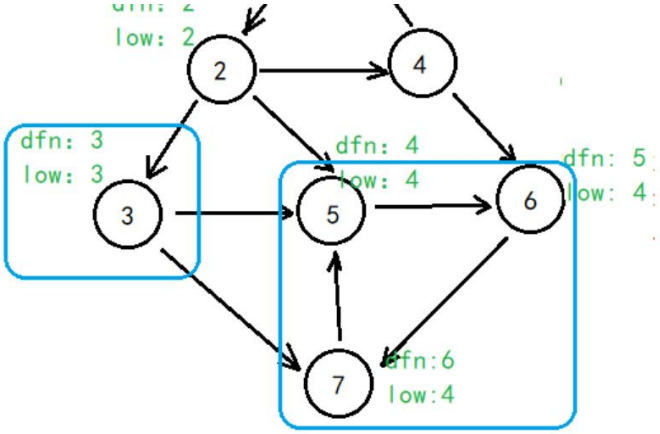


当搜索点7连接的点时，发现一个栈中的元素，接着更新它们的low，并把找到的第一个强连通分量除第一个元素依次弹出。当我们接着对点5搜索时，发现它已经没有其它指向的点，因此以5为起始点的强连通分量完全被找到了。把5也弹出。

stack：1 2 3 5 6 7

⇩

stack：1 2 3



依据dfs的性质，回溯到点3继续进行搜索，搜索到点7，此时点7已经不再栈中，但是点7已经被搜索过了，我们只比较两者的low，看看是否能把点3加入到这个强连通分量中。显然点3比点7所在的强连通分量更早被发现，无法加入其中。此时点3没有剩余结点可以搜索，所以点3构成一个孤立的分量。

stack：1 2

(1) 线性探测法

di = 1, 2, 3, ..., m-1

这种探测方法可以将散列表假想成一个循环表，发生冲突时，从冲突地址的下一单元顺序寻找空单元，如果到最后一个位置也没找到空单元，则回到表头开始继续查找，直到找到一个空位，就把此元素放入此空位中。如果找不到空位，则说明散列表已满，需要进行溢出处理。

(2) 二次探测法

$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \ldots, +k^2, -k^2\ (k \leq m/2)$

(3)伪随机探测法

di = 伪随机数序列

例如，散列表的长度为 11，散列函数 H(key)=key%11，假设表中已填有关键字分别为 17、60、29 的记录，如图2(a)所示。现有第四个记录，其关键字为38，由散列函数得到散列地址为 5，产生冲突。

若用线性探测法处理时，得到下一个地址6，仍冲突；再求下一个地址7，仍冲突；直到散列地址为8的位置为“空”时为止，处理冲突的过程结束，38填入散列表中序号为8的位置，如图2(b)所示。

若用二次探测法，散列地址5冲突后，得到下一个地址6，仍冲突；再求得下一个地址 4，无冲突，38填入序号为4的位置，如图 2(c)所示。

若用伪随机探测法，假设产生的伪随机数为9，则计算下一个散列地址为(5+9)%11=3，所以38 填入序号为3 的位置，如图 2(d)所示。