在散列表（哈希表）中，解决冲突（即两个不同的键映射到同一个哈希值）的主要方法有以下几种：

链地址法（Separate Chaining）：

基本思想：每个哈希表的槽（bucket）存储一个链表（或其他数据结构，如平衡树），所有映射到同一个哈希值的元素都保存在这个链表中。

优点：容易实现，处理冲突时只需在链表上插入或删除元素。

缺点：如果哈希表的负载因子（装填因子）过高，链表长度可能变长，导致查找效率降低为

O(n)。示例：

hash\_table = [[] for \_ in range(size)]

def insert(hash\_table, key, value):

index = hash\_function(key) % size

hash\_table[index].append((key, value))

def search(hash\_table, key):

index = hash\_function(key) % size

for k, v in hash\_table[index]:

if k == key:

return v

return None

开放地址法（Open Addressing）：

基本思想：当发生冲突时，使用探测序列在哈希表内找到下一个空槽。

常见策略：

线性探测（Linear Probing）：冲突后依次检查下一个槽，直到找到空槽。

公式：index=(hash(key)+i)%size，其中 i 是探测次数。

优点：实现简单，连续存储有利于缓存性能。

缺点：容易发生聚集（clustering），即冲突元素聚在一起，导致探测次数增加。

二次探测（Quadratic Probing）：冲突后按照二次方序列探测。

公式：index=(hash(key)+i^2 )%size。

优点：减少了聚集现象。

缺点：仍然可能产生二次聚集，探测序列必须处理好避免无限循环。

双重散列（Double Hashing）：使用两个哈希函数，发生冲突时根据第二个哈希函数计算探测步长。

公式：index=(hash1(key)+i×hash2(key))%size。

优点：分布更均匀，减少聚集现象。

缺点：实现稍复杂，需要设计两个合适的哈希函数。

示例（线性探测）：

def linear\_probing\_insert(hash\_table, key, value):

index = hash\_function(key) % size

while hash\_table[index] is not None:

index = (index + 1) % size

hash\_table[index] = (key, value)

def linear\_probing\_search(hash\_table, key):

index = hash\_function(key) % size

while hash\_table[index] is not None:

if hash\_table[index][0] == key:

return hash\_table[index][1]

index = (index + 1) % size

return None

再哈希法（Rehashing）：

基本思想：当哈希表负载因子达到一定阈值时，重新调整哈希表大小，并重新计算所有元素的哈希值，将它们放入新的哈希表中。

优点：可以显著减少冲突，保持高效的查找性能。

缺点：再哈希过程较耗时，涉及所有元素的重新插入。

示例：

def rehash(hash\_table):

new\_size = size \* 2

new\_table = [None] \* new\_size

for item in hash\_table:

if item is not None:

new\_index = hash\_function(item[0]) % new\_size

new\_table[new\_index] = item

return new\_table

KMP算法（Knuth-Morris-Pratt算法）是一种经典的字符串匹配算法，用于在一个文本串中查找一个模式串的出现位置。它的时间复杂度为 O(n+m)，其中 n 是文本串的长度，m 是模式串的长度。KMP算法通过预处理模式串，避免了在匹配过程中重复扫描已经匹配的部分，从而提高了效率。

KMP算法的基本思想

KMP算法的核心思想是通过构建一个部分匹配表（也称为前缀函数或失配函数），记录模式串前缀的匹配信息，以便在发生不匹配时能够跳过不必要的字符比较。

1. 部分匹配表（前缀函数）

部分匹配表用于存储模式串的前缀和后缀的最长公共部分的长度。部分匹配表定义为一个数组 π，其中 π[i] 表示模式串中从开头到位置 i 的子串的最长前缀的长度，这个前缀同时也是这个子串的后缀。

构建部分匹配表的步骤如下：

初始化：π[0] = 0。

对于模式串中的每个字符，逐个计算 π[i] 的值。

如果模式串中的字符 pattern[i] 和 pattern[k] 匹配，则 π[i] = k + 1，否则根据 π[k-1] 的值继续比较。

2. 匹配过程

在匹配过程中，KMP算法利用部分匹配表的信息来跳过已经匹配的部分，从而避免重复比较：

初始化文本串和模式串的指针 i 和 j，均指向第一个字符。

如果 pattern[j] 与 text[i] 匹配，则两者同时向前移动一位（i++，j++）。

如果 pattern[j] 与 text[i] 不匹配且 j > 0，则根据部分匹配表更新 j 的值为 π[j-1]，而 i 的值不变。

如果 pattern[j] 与 text[i] 不匹配且 j == 0，则仅 i++。

如果 j 达到模式串的末尾，则表示匹配成功，可以记录匹配的位置，并更新 j 的值为 π[j-1] 以继续查找其他可能的匹配位置。

示例

假设文本串为 text = "ababcabcabababd"，模式串为 pattern = "ababd"。

构建部分匹配表：

对于 pattern = "ababd"，其部分匹配表 π 为：

复制

π = [0, 0, 1, 2, 0]

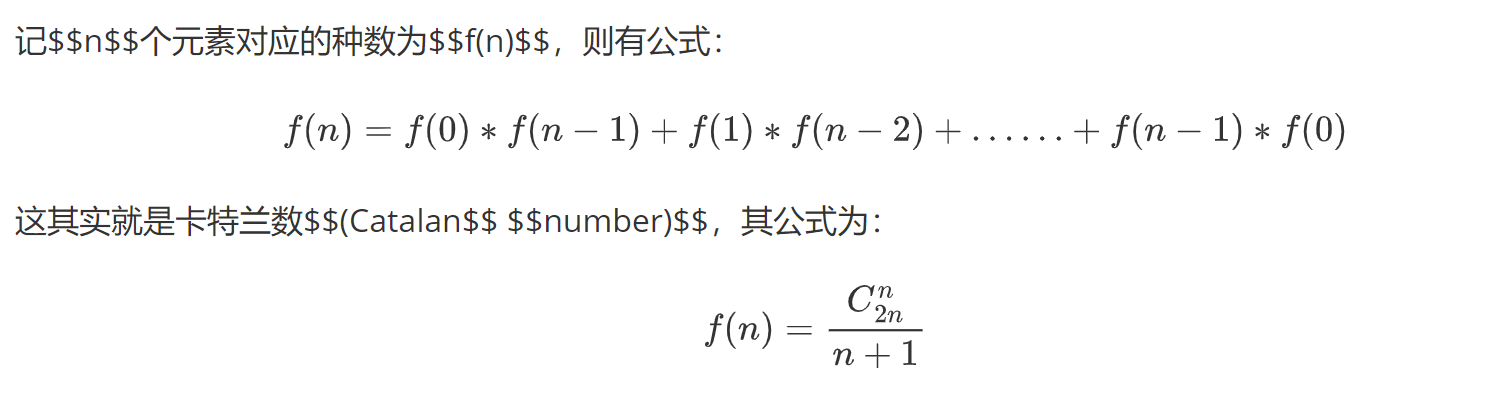
匹配过程：

开始时，i = 0，j = 0。

当 pattern[j] 与 text[i] 匹配时，i 和 j 同时向前移动。

当 pattern[j] 与 text[i] 不匹配时，根据部分匹配表调整 j 的值。

继续上述过程，直到找到匹配或遍历完整个文本串。

是的，双端队列（Deque，即Double-Ended Queue）可以使用两个栈（Stack）来实现。

双端队列是一种具有队列和栈性质的数据结构，允许在队列的两端进行插入和删除操作。栈是一种后进先出（LIFO）的数据结构，而队列是一种先进先出（FIFO）的数据结构。尽管栈和队列的性质不同，但是可以利用两个栈的特性来模拟实现双端队列。

具体实现方法如下：

1. 使用两个栈，分别称为 `stack1` 和 `stack2`。

2. 元素的插入操作（`push\_back`）时，将元素压入 `stack1`。

3. 元素的删除操作（`pop\_front`）时，首先检查 `stack2` 是否为空。如果不为空，直接从 `stack2` 弹出栈顶元素；如果为空，将 `stack1` 中的所有元素逐个弹出并压入 `stack2`，然后从 `stack2` 弹出栈顶元素。

4. 元素的插入操作（`push\_front`）时，将元素压入 `stack2`。

5. 元素的删除操作（`pop\_back`）时，首先检查 `stack1` 是否为空。如果不为空，直接从 `stack1` 弹出栈顶元素；如果为空，将 `stack2` 中的所有元素逐个弹出并压入 `stack1`，然后从 `stack1` 弹出栈顶元素。

通过以上操作，我们可以使用两个栈模拟实现双端队列的插入和删除操作。这是因为栈的特性使得元素在插入和删除时可以保持正确的顺序。

需要注意的是，使用两个栈实现的双端队列可能会涉及到元素的频繁移动，因此在某些情况下可能不如直接使用链表或动态数组实现的双端队列高效。但从概念上来说，使用两个栈来实现双端队列是可行的。

3. 请编写算法maxsub，返回两个字符串astr和bstr的1个最长公共子串，给定astr的长度不大于bstr。例如：maxsub("kabcd", "nbcyuu")返回"bc"，maxsub("123", "abcd")返回""。

def maxsub(astr, bstr):

len\_a = len(astr)

len\_b = len(bstr)

```python

# 创建一个 (len\_a+1) x (len\_b+1) 的二维数组，并初始化为 0

dp = [[0] \* (len\_b + 1) for \_ in range(len\_a + 1)]

# 变量来追踪最长公共子串的长度和结束位置

max\_length = 0

end\_index = 0

# 填充 dp 数组

for i in range(1, len\_a + 1):

for j in range(1, len\_b + 1):

if astr[i - 1] == bstr[j - 1]:

dp[i][j] = dp[i - 1][j - 1] + 1

if dp[i][j] > max\_length:

max\_length = dp[i][j]

end\_index = i

else:

dp[i][j] = 0

# 提取最长公共子串

if max\_length == 0:

return ""

else:

return astr[end\_index - max\_length:end\_index]

```

这涉及到图论中的路径问题，特别是使用相邻矩阵来表示图中的路径。

\*\*相邻矩阵 \(A\)\*\*

相邻矩阵 \(A\) 是一个 $n \times n$ 的矩阵（对于 n 个顶点的图），其中 $A\_{ij}$ 表示顶点 $V\_i$ 和顶点 $V\_j$ 之间是否有直接的边。如果有直接的边，则 $A\_{ij}$ 为 1，否则为 0。

\*\*路径矩阵\*\*

相邻矩阵的 \(m\) 次幂 $A^m$ 的第 i 行第 j 列的元素表示从顶点 $V\_i$ 到顶点 $V\_j$ 有长度为 m 的路径的数量。如果该元素不为零，说明存在长度为 \(m\) 的路径。

\*\*判定路径\*\*

为了判定任意两个顶点 $V\_i$ 和 $V\_j$ 之间是否有长度为 m 的路径相连，只要检查 $A^m$ 的第 i 行第 j 列的元素是否为零即可。

