

第5章 贪心法

|| 5.1 贪心法概述

苏格拉底、柏拉图和亚里士多德是古希腊著名思想家、哲学家，合称“古希腊三贤”。苏格拉底是柏拉图的老师，柏拉图是亚里士多德的老师。有一天，柏拉图问他的老师苏格拉底：“老师，什么是爱情？”苏格拉底说：“你去麦田捡麦穗，记住只能捡一次，不能回头。”柏拉图就去了，不一会，柏拉图回来了，但是什么也没有带回来，苏格拉底就问他，“你怎么什么也没捡到啊？”柏拉图：“我在地里看到几个特别大的麦穗，可是我不确定这个是不是地里最大的，于是就没有捡。但是，之后也没有发现更好的麦穗，所以我最后什么都没捡到。”苏格拉底说：“这就是爱情。”

有一天，柏拉图又问苏格拉底：“老师，什么是婚姻？”苏格拉底说：“你去那片树林砍一棵最高大的树，但不能走回头路，而且只能砍一次。”这次，柏拉图带了一棵并不算最高大但是看着还不错的树回来了。苏格拉底就问他：“你怎么砍了这样一棵树回来？”柏拉图说：“我穿过森林时，看到了几棵非常好的树，这次我吸取了上次的教训，就砍了下来。我很怕如果不选它，就又会空手而归。”苏格拉底说：“这就是婚姻。”

这个故事和贪心法有什么关系呢？其实，这个故事体现了寻优问题求解的两个思路，一个是寻找全局最优，另一个是寻找局部最优。在捡麦穗的故事里，柏拉图想找到问题的全局最优解，结果发现很困难，因为他没办法遍历整个麦田。在砍树的故事里，柏拉图发现寻找全局最优非常困难，就转而在他周围选了一棵比较高大的树，这个就是局部最优解。

在算法设计策略中，贪心法是求解寻优问题的一种通用策略，其核心思想是把一个大问题通过一步操作转换为小问题，这个操作基于贪心选择准则，保证这一步获得最大收益，即寻找局部最优解，至于这个局部最优解是不是能保证原问题达到全局最优，贪心法是无法保证的。不同的贪心选择准则可以得到不同的结果，找到正确的贪心选择准则是设计贪心算法的关键。贪心法适用于求解 CPU 任务调度、最小生成树、最短路径、旅行商问题、分数背包问题、装箱等寻优问题。

举一个找零钱的例子，给定无限多的 m 种面额的硬币，面额大小排序为 $d_1 > \dots > d_m$ ，现在要找一个总额为 n 的零钱，问题是如何使用最少的硬币数目。例如， $d_1 = 25c$ ， $d_2 = 10c$ ， $d_3 = 5c$ ， $d_4 = c$ ， $n = 48c$ 。对于这个问题，如果采用贪心法求解，首先要考虑问题怎么变小？一个很简单的想法是先找一个硬币，这样 n 就变小了，问题也就小了。问题就变为：这一个硬币怎么选？一般会优先选 $25c$ ，因为它能让问题变得最小，这个选择就是局部最优，因为只考虑了这一步的获益。当这个问题变成 $48c - 25c = 23c$ 时，这个小问题求解采用了

和原问题同样的策略,继续选择两个 $10c$,最后选择 3 个 c 的零钱,这样就得到贪婪解 $\langle 1, 2, 0, 3 \rangle$ 。从这个例子可以看出,贪心法的贪心选择准则(贪心策略)是求解问题的关键,即在当前状态得到局部最优,仅考虑目前状态,不考虑长远结果。这样做是有风险的,例如,当 $n=30c$ 时,如果按照前面的贪心准则,就会得到贪婪解 $\langle 1, 0, 5 \rangle$,而事实上,最优解是 $\langle 0, 3, 0 \rangle$ 。这说明贪心法不能保证得到最优解。但是,贪心法很简单,实现效率高。

下面再举一个活动选择的问题。假设你在迪士尼主题公园玩,公园里有很多活动,每个活动开始和结束的时间都不同。为了简化问题,忽略活动之间的行走时间。现在的问题是:怎么选择游玩的活动顺序,才能使你玩的活动项目最多?先给出这个问题的数学描述,假设有 n 个活动,记 $S = \{a_1, a_2, \dots, a_n\}$, a_i 是第 i 个活动,该活动的持续时间是 $[s_i, f_i)$, s_i 是开始时间, f_i 是结束时间,找出 S 的最大子集 A ,满足:①活动之间的时间不重叠;②活动的个数 $|A|$ 达到最大。

不失一般性,假设 $f_1 \leq f_2 \leq \dots \leq f_n$,例如,4 个活动的开始时间和结束时间如表 5-1 所示,图 5-1 给出了 4 个活动之间的前后时间关系。

表 5-1 4 个活动的开始时间和结束时间

活动 i	1	2	3	4
s_i	0	1	2	3
f_i	2	4	6	7

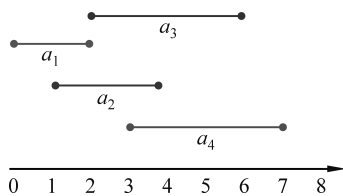


图 5-1 4 个活动的前后时间关系

如果选用最早结束时间优先的贪心准则,也就是哪个活动结束得越早,就优先选择哪个活动。在上面的例子中, a_1 最早结束,就先选择 a_1 ,由于 a_2 与 a_1 时间重叠,所以在剩下的活动中不考虑 a_2 ,只剩下活动 a_3 和 a_4 。原问题是 4 个活动的选择问题,而现在变成了 2 个活动的选择问题,问题规模变小了,这就是前面所说的贪心法的基本思想:利用一个操作(选取一个活动)将问题规模变小,这个操作基于局部最优的原则。在活动选择问题中选用了最早结束时间优先,基于这个贪心准则,选取下一个活动,问题的规模可以变得更小,在小问题中继续使用该贪心准则,直至问题解出。这时得到的一系列操作序列,就是问题的解。

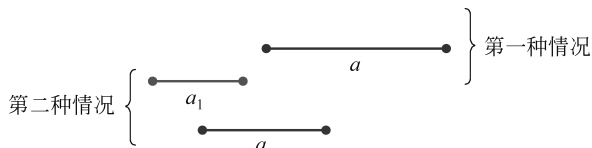
现在,活动选择问题已经利用贪心法解出了,得到的解是最优的吗?下面证明一下。证明的思路分两步:第一步先证明如果有最优解,则最优解中一定包含 a_1 ,也就是最早结束的那个活动;第二步再证明最早结束时间优先的贪心准则一定得到最优解。

定理 1: 如果活动 a_1 在所有活动中具有最早结束时间,则最优解中一定包含 a_1 。

证明: 假设集合 A 是活动选择问题的最优解, a_1 是贪心法选择的最早结束时间的活动,如果 $a_1 \in A$,则定理得证。

如果 $a_1 \notin A$,可以在集合 A 中找到最早结束的活动 a ,因为 $a_1 \notin A$,必然 $a_1 \neq a$ 。那么, a_1 和 a 的关系可能有图 5-2 中的两种情况,第一种情况是 a_1 和 a 时间上不相互重叠,第二种情况是 a_1 和 a 重叠。第一种情况可以排除,因为如果 a_1 和 a 没有时间重叠,那么 a_1 可以直接加入集合 A 中,使集合 A 的元素个数增加,这就与集合 A 是最优解的这个前提互相矛盾。那么只剩下第二种情况了,即 a_1 和 a 在时间上是重叠的,这时,只需要利用 a_1 替换 a 即可,这时集合 A 的元素个数不变,但是包含了最早结束的活动 a_1 ,从而定理得证。

定理 2: 最早结束时间优先的贪心准则一定得到最优解。

图 5-2 a_1 和 a 的关系

证明： a_1 是贪心准则选择的最早结束的活动，令 S^* 是不与 a_1 重叠的活动子集，则

$$S^* = \{a_i \mid i = 2, \dots, n, \text{ 且 } s_i \geq f(a_1)\}$$

令 B 为 S^* 的最优解，从 S^* 的定义可知，集合 $A^* = \{a_1\} \cup B$ 是可行的，并且是原问题的解。下面利用反证法证明 A^* 是最优解。假设 A^* 不是最优解，而 A 是最优解，则 $|A^*| < |A|$ 。由前面的定理 1 可知， A 中必然包含 a_1 ，且 $|A - \{a_1\}| > |A^* - \{a_1\}| = |B|$ 。但是， $A - \{a_1\}$ 也是 S^* 的解，与 B 是 S^* 的最优解矛盾。这意味着 A^* 一定是原问题的最优解。

根据贪心准则就可以正确选择第一个活动，那么，这个问题就缩小为在 S^* 中求解活动选择问题，可以继续重复使用这个贪心准则。

|| 5.2 贪心法中的计算思维

贪心法是算法设计策略中一种常用的方法，体现了计算思维在求解问题时的分解、局部最优选择、抽象与建模等特点。

(1) 问题分解。

贪心法通过贪心准则将大问题转换为小问题，这个问题转换的特点是计算思维的一个重要特征，就是强调将复杂的问题分解为更小、更易求解的小问题。

(2) 局部最优选择。

寻优是计算思维的一个特征。在贪心法中，问题的解决通常是通过一步步的局部最优选择来解决的，每一步都基于前一步的解决结果，再选择当前状态下的最优解，进而推动整个寻优过程。

(3) 抽象与建模。

计算思维侧重于识别问题中的模式和规律，即将问题的实际细节简化为更高层次的抽象表示，便于计算机处理。贪心法把整个问题求解转换为一系列局部决策，这个过程本身就是对问题的一个抽象过程，贪心法需要识别出每一步寻找局部最优解的抽象模式，以及问题中的重复结构或规律，这种模式识别能力可以把问题求解抽象为数据结构和算法的形式，是计算思维抽象与建模特点的体现。

贪心法在实际问题中体现了计算思维的一些关键思考方式，包括问题分解、寻优、抽象化和建模等。这些思维方式有助于更高效、简单地解决特定类型的问题。

|| 5.3 贪心法的实践案例

5.3.1 最少站台问题

1. 问题描述

假设 n 辆火车的到达时间 arr 和离开时间 dep 已知，问题是最少需要多少个站台，才能

保证各个时间段的火车都能进站。

$n=6$;

到达时间 $arr[] = \{9:00, 9:40, 9:50, 11:00, 15:00, 18:00\}$;

离开时间 $dep[] = \{9:10, 12:00, 11:20, 11:30, 19:00, 20:00\}$;

由于 $[9:40, 12:00]$ 这个时间段的火车最多,为 3 辆,所以需要的最少站台数为 3。

2. 解决方法: 贪心法

(1) 算法原理。

本节要求的是最少站台数,其实就是找到每个到达-离开的时间段相重叠的最大个数。对于每个时间段,可以利用贪心法求重叠最多的时间段个数。而最少站台数就是在所有时间段对应的重叠个数中取最大值。

为了快速得到重叠时间段的个数,需要对到达时间进行排序,如图 5-3 所示,然后针对每一趟火车的停留时间段,找与之重叠的时间段。在找重叠时间段的过程中,采用了贪心法的思路,就是从当前火车到达时间开始向后搜索所有火车,只要停留时间与当前火车停留时间重叠,就计入重叠数,直至找到全部重叠时间的火车。

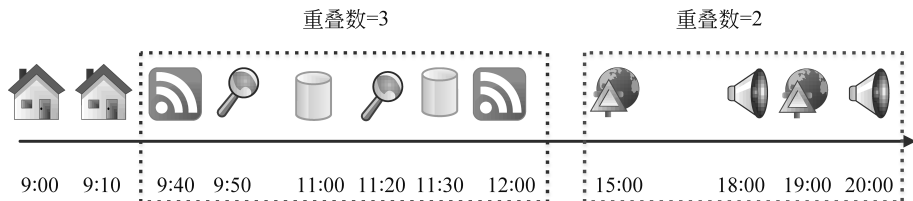


图 5-3 重叠时间段一计数(同一标识代表一辆火车的到达和离开时间)

上面的例子可以拆解为以下情况。

- 第 1 辆火车的到达时间为 9:00,离开时间为 9:10,没有和这个时间段重叠的其他火车,即这段时间需要最少站台数为 1。
- 第 2 辆火车的到达时间为 9:40,离开时间为 12:00,这个时间段内有第 2、3、4 辆火车停靠,需要最少站台数为 3。
- 第 3 辆火车的到达时间为 9:50,离开时间为 11:20,这个时间段内有第 3、4 辆火车停靠,即这段时间需要最少站台数为 2。
- 第 4 辆火车的到达时间为 11:00,离开时间为 11:30,这个时间段内有第 3、4 辆火车停靠,即这段时间需要最少站台数为 2。
- 第 5 辆火车的到达时间为 15:00,离开时间为 19:00,这个时间段内有第 5、6 辆火车停靠,即这段时间需要最少站台数为 2。
- 第 6 辆火车的到达时间为 18:00,离开时间为 20:00,这个时间段内有第 5、6 辆火车停靠,即这段时间需要最少站台数为 2。

目前得到的相互重叠的最大时间段为 3 段,即最少需要 3 个站台。

(2) 伪代码。

Greedy Programming

Input. 达到时间 arr , 离开时间 dep

Output. 最少站台个数



```

Method.
min_Platform(arr,dep,n)
1   res = 1
2   for i = 1 to n
3       min_plat = 1
4       for j = 1 to n
5           If i != j
6               If arr[i]>= arr[j]&&dep[j]>= arr[i] ||
dep[i]>= arr[j]&&dep[i]<= dep[j]
7                   min_plat ++;
8       res = max (res, min_plat)
9   Return res

```

(3) 复杂度分析。

时间复杂度：由于使用了两层循环完成，时间复杂度是 $O(n^2)$ 。

空间复杂度： $O(1)$ 。

(4) 堆优化。

根据到达时间对火车进行排序，然后检查下一班火车的离开时间是否小于前一班火车的离开时间，如果小于则说明出现重叠，增加所需的站台数量，否则不增加。

对于该思路，可以利用一个小顶堆来高效解决，堆顶始终存储当前的最早离开时间。

到达时间记为 $arr[] = \{9:00, 9:40, 9:50, 11:00, 15:00, 18:00\}$ ；

离开时间记为 $dep[] = \{9:10, 12:00, 11:20, 11:30, 19:00, 20:00\}$ 。

假设对所有火车的到达时间从小到大排序，把 9:00 简写为 900，则排序后的火车顺序是 $\{(900, 910), (940, 1200), (950, 1120), (1100, 1130), (1500, 1900), (1800, 2000)\}$ 。

- 小顶堆初始时存储第 1 辆到达的火车的离开时间 910；
- 第 2 辆火车的到达时间 940 大于存储在小顶堆的离开时间 910，说明它们没有相交，对重叠数量没有贡献，可以把堆顶元素 910 弹出，再把离开时间 1200 加入堆，堆顶变为 1200；
- 第 3 辆火车的到达时间 950 小于存储在小顶堆的离开时间 1200，说明出现重叠，重叠数量加 1，此时把新的离开时间 1120 加入堆，对小顶堆进行调整，堆顶元素变为 1120；
- 第 4 辆火车的到达时间 1100 小于存储在小顶堆的离开时间 1120，说明出现重叠，重叠数量加 1，此时把新的离开时间 1130 加入堆，对小顶堆进行调整，堆顶元素仍然是 1120，堆里目前有三个元素；
- 第 5 辆火车的到达时间 1500 大于存储在小顶堆的离开时间 1120，说明它们没有相交，对重叠数量没有贡献，可以把堆顶元素 1120 弹出，再把离开时间 1900 加入堆，重新调整堆后，堆顶变为 1130；
- 第 6 辆火车的到达时间 1800 大于存储在小顶堆的离开时间 1130，说明它们没有相交，对重叠数量没有贡献，可以把堆顶元素 1130 弹出，再把离开时间 2000 加入堆，堆顶变为 1200。

小顶堆的调整过程如图 5-4 所示。从上述过程可以看出，小顶堆的目的就是快速找出目前最早的离开时间，方便后面到达的火车进行比较，由于堆调整的复杂度是 $O(\log n)$ ，可以保证调整效率比较高。

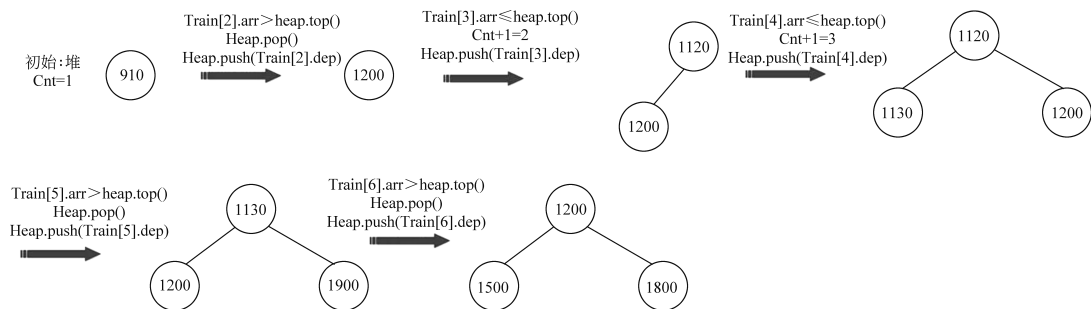


图 5-4 小顶堆的调整过程

(5) 堆优化伪代码。

```
Greedy Programming
Input. 达到时间 arr, 离开时间 dep
Output. 最少站台个数
Method.
min_Platform(arr, dep, n)
1   for i = 1 to n
2       train[i] = { arr[i], dep[i] }
3   sort (train.arr)
4   heap.push (train[0].dep)
5   cnt = 1
6   for i = 2 to n
7       if heap.top() >= train[i].arr
8           cnt ++
9       else heap.pop()
10      heap.push (train[i].dep)
11  return cnt
```

(6) 堆优化复杂度分析。

时间复杂度：由于采用堆调整的策略,时间复杂度是 $O(n \log(n))$ 。

空间复杂度：存储堆的代价是 $O(n)$ 。

(7) 排序优化。

这个优化思路按所有时间排序,包括到达时间和离开时间,计数当前已到达但未离开的火车数量,它的最大值即为所需最少站台数。图 5-5 列出了这个求解过程。

(8) 排序优化伪代码。

```
Greedy Programming
Input. 达到时间 arr, 离开时间 dep
Output. 最少站台个数
Method.
min_Platform(arr, dep, n)
1   sort(arr, arr + n)
2   sort(dep, dep + n)
3   i = 2, j = 1, count = 1
4   while i < n && j < n
```

```
5         if arr[i] <= dep[j]
6             count ++, i++
7         else count --, j ++
8         res = max (res, count)
9     return res
```

排序完时间：
arr[]={900,940,950,1100,1500,1800}
dep={910,1120,1130,1200,1900,2000}

时间	状态	计数
900	到达	1
910	离开	0
940	到达	1
950	到达	2
1100	到达	3
1120	离开	2
1130	离开	1
1200	离开	0
1500	到达	1
1800	到达	2
1900	离开	1
2000	离开	0
min_count		3

图 5-5 排序-计数求解过程

(9) 排序优化复杂度分析。

时间复杂度：由于采用了排序算法,排序效率是 $O(n\log(n))$,而遍历排序结果的效率是 $O(n)$,所以总的时间复杂度是 $O(n\log(n))$ 。

空间复杂度：不需要辅助空间,所以空间复杂度是 $O(1)$ 。

5.3.2 最短超级字符串

1. 问题描述

给定一个字符串列表,在所有字符串中没有一个字符串是另一个字符串的子字符串。现在的问题是：找到一个最短的超级字符串,使它包含列表中的所有字符串作为其子字符串。

例如,输入一个字符串列表：[CATGC,CTAAGT,GCTA,TTCA,ATGCATC],输出超级字符串为：GCTAAGTTCATGCATC。该超级字符串包含了字符串列表中的每个子字符串,同时也是最短的。

2. 解决方法：贪心法

(1) 算法原理。

这个问题需要构造一个包含所有子字符串的最短字符串,根据贪心算法的思想,需要先

找到局部最优解,然后利用局部最优解构建全局最优解。这里的贪心策略就是利用两个字符串合并成一个超级字符串,然后通过超级字符串替换原来的两个字符串,这样字符串列表就少了一个字符串。重复这个过程,就可以在最终的字符串列表中得到全局超级字符串。

以上面的字符串列表为例,利用局部最优解构造全局最优解的过程如图 5-6 所示。

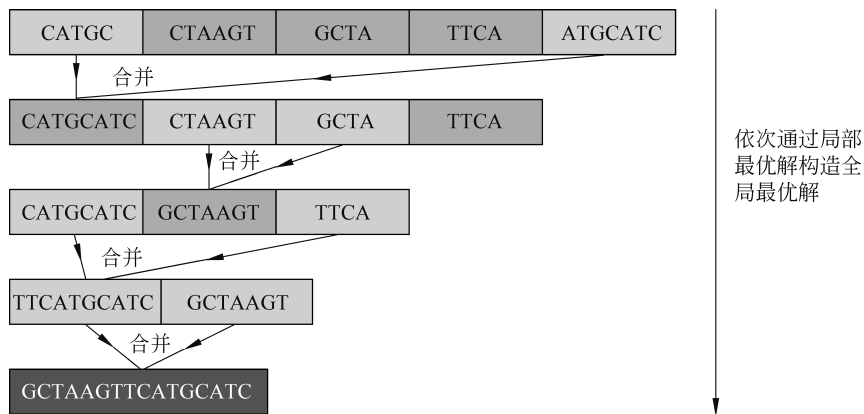


图 5-6 通过局部最优解构造全局最优解

下面讨论如何求出局部最优解,即如何将两个字符串合并,得到它们的超级字符串。假设两个字符串为 a 、 b , a 的字符串长度为 n , b 的字符串长度为 m 。若 a 、 b 中有相同的子字符串,那么超级字符串的结构就是:“其中一个字符串的前半段+两个字符串相同部分+另一个字符串的后半段”。注意,这里要找的相同子字符串需要同时出现在 a 的前半段和 b 的后半段,或者 b 的前半段和 a 的后半段,不能随意在 a 和 b 中找相同子字符串。例如, a = “CATGC”, b = “ATGCATC”, a 和 b 都包含的字符串是 “ATGC”,这个子串是 a 的后半段,同时是 b 的前半段,但是如果找的公共子串是 “TGC”,这个子串在 b 的中间,那么合并时就会截断 b ,无法得到超级字符串。

假设两个字符串相同部分的长度为 i ,针对下面两种情况分别分析。

① 若 a 的前半段与 b 的后半段相同,此时公共子串 c 就是 a 字符串的下标 0 到 $i-1$ 的子字符串,也是 b 字符串的下标 $m-i$ 到 $m-1$ 的子字符串,合并后的字符串包括 b 字符串以及 a 下标从 i 开始之后的子字符串,如图 5-7 所示。

② 若 b 的前半段与 a 的后半段相同,此时 b 字符串的下标 0 到 $i-1$ 的子字符串与 a 字符串的下标 $n-i$ 到 $n-1$ 的子字符串相同,合并后的字符串包括 a 字符串以及 b 下标从 i 开始的子字符串,如图 5-8 所示。

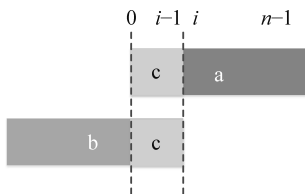


图 5-7 a 的前半段与 b 的后半段相同

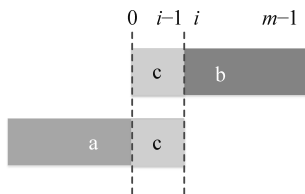


图 5-8 b 的前半段与 a 的后半段相同

下面可以从 i 的长度入手,根据上述两种情况考虑,依次增加 i 的长度,找到超级字符串最短的情况,即局部的最优解,也就是贪心法的思想。

以[CATGC,ATGCATC]为例,其求解局部最优解的过程如图 5-9 所示。

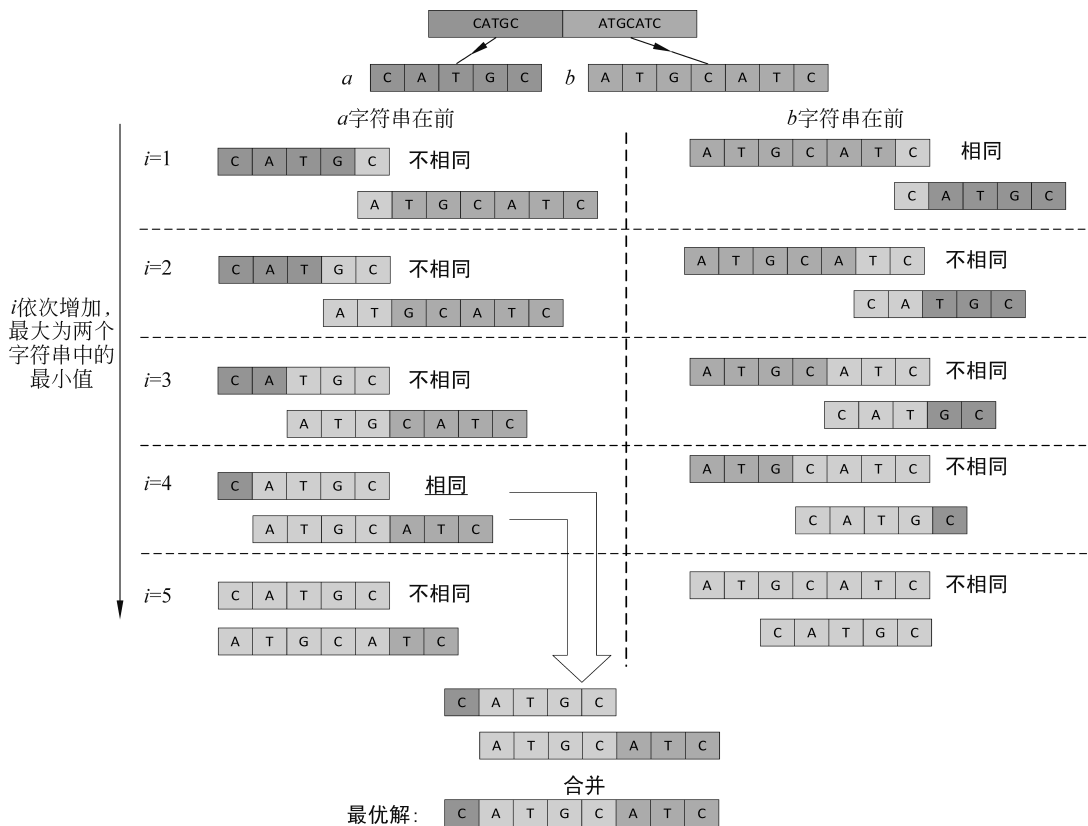


图 5-9 合并两个字符串的过程

注意,在该过程中,若出现多次子字符串匹配相同的情况,要取 i 最大的情况,因为要求的是最短超级字符串,最优解合并出来的字符串需要是最短的。

根据以上过程,可以得到两个字符串合并的超级字符串,也就是局部最优解,然后替换原字符串序列中的两个字符串,再重复上述过程,就可以求出全局最优解。

(2) 伪代码。

```

Greedy Algorithm
Input. 字符串集合 S
Output. 最短超级字符串
Method.
findShortestSuperstring(words)
1  n = words.size()           //记录字符串的个数
2  while (n != 1)             //若字符串列表长度大于 1,说明存在可以合并的字符串
3      max = INT_MIN           //存储两个字符串之间相同部分最长的长度
4      for (i = 0; i < n; i++) //依次比较两个字符串
5          for (j = i + 1; j < n; j++)
6              r = findOverlappingPair(words[i], words[j], str)
                  //寻找两个字符串 words[i]和 words[j]的公共子串 str,返回 str 的长度 r
    
```



```

7         if (max < r)           //找到相同部分长度最长的情况
8             max = r           //记录最长的相同部分长度,用于后续比较
9             res_str.assign(str) //res_str 记录合并后的字符串
10            p = i, q = j //记录比较的两个字符串在列表中的位置
11            n--                //减少列表长度,替换局部最优解
12            if (max == INT_MIN) words[0] += words[n] //若没有找到可以匹配的两个字符串,
//将最后一个字符串与第一个字符串合并,作为此时的局部最优解,以便消去最后一个
//字符串,减少列表长度
13            else //若找到局部最优解,则将位置靠前的字符串的位置替换为合并后的字符串,
//靠后的字符串则用来存储被消去的最后一个字符串
14                words[p] = res_str
15                words[q] = words[n]
16            return words[0]      //当局部最优解全部求出,第一位的字符串即为全局最优解

```

```

findOverlappingPair(s1, s2, str) //函数 findOverlappingPair 来求取字符串 s1 和 s2
//的合并字符串 str
1  max = INT_MIN                //用于存储合并后字符串的长度,设为最小以便后续更新该值
2  m = s1.length()
3  n = s2.length()
4  for (i = 1; i <= min(m, n); i++) //相同的子字符串的长度 i
5      if (s1.compare(m - i, i, s2, 0, i) == 0) //若 s1 的后半段与 s2 的前半段相同
6          if (max < i) //若该相同子字符串的长度大于最大值
7              max = i //替换最大相同的子字符串的长度
8              str = s1 + s2.substr(i) //合并后的字符串
9  for (i = 1; i <= min(m, n); i++) //若 s2 的后半段与 s1 的前半段相同,处理方法同上
10     if (s1.compare(0, i, s2, n - i, i) == 0)
11         if (max < i)
12             max = i
13             str = s2 + s1.substr(i)
14  return max //此时合并后的字符串已经通过引用的 str 存下,返回相同部分最长的长度

```

5.3.3 重排字符串问题

1. 问题描述

给定一个字符串,通过重新排列字符,使得字符串中相邻位置的字符互不相同。举两个例子:

- ① 给定字符串 $s = \text{"aaabc"}$,输出:“abaca”可以保证相邻字符互不相同;
- ② 给定字符串 $s = \text{"aaba"}$,输出:不存在一个新的排列结果满足条件。

2. 解决方法:贪心法 1

(1) 算法原理。

从字符串中字符摆放的顺序来分析,出现在最前面的字符还有可能在后面出现,所以在所有字符中出现的概率最大,字符出现的概率会随着位置靠后越来越小,这也意味着相邻字符重复出现的概率会变小。按照这个思路,贪心法的核心思想是把出现频率高的字符放在