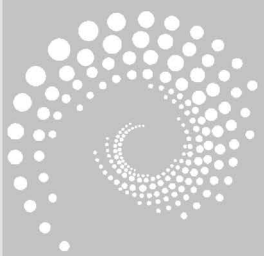
第3章

贪心算法

CHAPTER 3



3.1　贪心算法的思想

贪心算法依循一个简单的原则：在算法的每一个步骤中，总能做出在当前情况下看似最佳的选择，并希望这种局部的最优决策能够促成全局的最优解。这种策略虽然看起来过分简单，甚至有些幼稚，但实际上，在众多场合下，它都能带来意想不到的效果。贪心算法效率高且实现简单。与那些需要深入挖掘所有潜在解决方案的算法不同，贪心算法在大多数情况下仅需线性时间就可以完成任务，尤其是在某些特定问题上，如任务调度、构建最小生成树或货币找零问题，它不但能迅速找到最优解，而且执行效率极高。然而，这并不意味着贪心算法总能找到最优解。在一些场合下，单纯追求局部最优选择可能导致忽视更佳的全局解决方案。因此，在应用贪心策略之前，判断其是否适用于当前问题是至关重要的。

从数学的角度来看，贪心算法是一种基于局部最优选择来寻找全局最优解的策略。在每一步决策中，算法都会选择当前状态下的最优解，而不考虑这一选择可能对未来决策的影响。这可以被视为一个连续的最大化或最小化过程，其中每一步都是基于当前可用信息的最优决策。假设有一个目标函数，函数目标是希望最大化或最小化某个结果。在贪心策略下，不直接求解整体的问题，而是将其分解为一系列子问题，每个子问题都有其对应的局部目标函数。通过每一步最大化或最小化，从而逼近整体的最优解。

贪心算法思想的一个经典实例是硬币找零问题。假设有面额为 1 分、5 分、10 分、25 分的足够数量的硬币，需要给顾客找零，目标是满足顾客的找零金额，并且找零的硬币数量最少。可以将找零过程分解为每次选择一个硬币的多次选择过程。选择策略是每次都选择面额最大的硬币，直到找零的总额达到需要找零的金额。

在第 1 次选择时，有以下 4 种情况。

(1) 如果需要找零金额＞25 分，那么选择一枚面额为 25 分的硬币，记为选择 1 个 25 分面额的硬币。

(2) 如果需要找零金额＞10 分，那么选择一枚面额为 10 分的硬币，记为选择 1 个 10 分面额的硬币。

(3) 如果需要找零金额＞5 分，那么选择一枚面额为 5 分的硬币，记为选择 1 个 5 分面额的硬币。

(4) 如果需要找零金额＞1 分，那么选择一枚面额为 1 分的硬币，记为选择 1 个 1 分面额的硬币。

第 1 次选择之后如果还未达到找零的总额，则更新剩余的找零金额，令剩余的找零金额＝找零总额－已经选择的硬币金额。然后开始第 2 次选择，同样面临 4 种情况，继续选择直到剩余的找零金额等于 0，此时找零完成。例如，要找零金额为 36 分，贪心算法的步骤如下。

第 1 次，选择一枚面额为 25 分的硬币，剩余需要找零金额为 11 分。

第 2 次，选择一枚面额为 10 分的硬币，剩余需要找零金额为 1 分。

第 3 次，选择一枚面额为 1 分的硬币，剩余需要找零金额为 0。

最后用一枚面额为 25 分的硬币、一枚面额为 10 分的硬币和一枚面额为 1 分的硬币，共三枚硬币找零金额为 36 分。这个过程可以总结为以下 3 个步骤，并且将其推广到其他求解问题。

(1) 建立数学模型来描述问题。找零问题的目标函数是找零的硬币数量最少,并且约束条件的硬币金额加起来等于要求金额。

(2) 把求解的问题分成若干子问题,并对每个子问题求解。在找零问题中,最终目标是找出若干硬币,如果每次只选择一枚硬币,并且每次都是尽可能选择面额大的硬币,则可以使找零的硬币数最少。

(3) 把子问题的局部最优解合成原来问题的一个解。在找零问题中,选择的次数就是最终硬币的数量。

但是并非所有的问题利用贪心算法总能得到全局最优解,适用贪心算法的问题通常需要满足一些条件,即贪心选择性质和最优子结构性质。

3.2 贪心算法的要素

3.2.1 贪心选择性质

贪心选择性质指全局最优解可以通过一系列局部最优解得到。在求解过程中,只需要考虑做出一个看似当前最好的选择,也就是局部最优解,而不需要考虑子问题的解,这样可以简化决策过程。这一点与动态规划算法是有区别的。动态规划算法在每一步都可能会重新考虑以前的决策,如果发现更优的解决方案,则会舍弃以前的选择。贪心选择性质使得每一步都可以做出最优的选择,并且得到的全局解也是最优的。

用数学语言来表述即是考虑一个集合 A,从中选择一个子集 B 来最大化目标函数 $f(B)$。如果存在一个元素 $a \in A$ 满足对于所有包含 a 的子集 B',都有 $f(B') > f(B)$,其中 B 是不包含 a 的任何子集,那么称这个问题具有贪心选择性质。贪心选择性质为贪心算法提供了数学上的正当性,它确保了局部最优的决策可以逐步构建出全局最优解,从而使得贪心算法在某些特定的问题上能够成功地找到最优解。

3.2.2 最优子结构性质

在算法设计与数学优化领域,最优子结构性质是贪心算法与动态规划策略的核心。简而言之,最优子结构性质表明,一个问题的最优解包含了它各个子问题的最优解。动态规划算法利用了最优子结构性质通过存储子问题的解来避免重复工作,这通常称为“记忆化”。这种方法在解决斐波那契数列、最短路径问题等具有重叠子问题的情况时非常有效。而贪心算法则在每一步都做出在当前看来最优的选择,并期望这些局部最优能够导致全局的最优解。在哈夫曼编码、最小生成树等问题中,贪心策略能够确保达到全局最优。

假设有一个优化问题,目标函数为 f,其定义域为解空间 S。如果对于任意 $s \in S$,存在一个子集 $T \subseteq S$(其中 T 是与 s 相关的子问题的解空间),使得 $f(s)$可以通过 T 中的元素来计算,那么称函数 f 具有最优子结构性质。这一性质的核心思想在于可以通过解决小规模的子问题来解决大规模的原始问题。这种自下而上的方法允许逐步构建解决方案,并且每一步都基于前一步的结果。

贪心选择性质和最优子结构性质是贪心算法能够成功应用的关键,如果一个问题不能

满足这两个条件，那么贪心算法可能无法得到最优解。例如，在背包问题和旅行商问题中，贪心策略并不能保证得到最优解。

3.3　活动选择问题

3.3.1　问题概述

活动选择问题是优化问题的经典示例，它涉及一系列活动。每个活动都有一个开始时间和结束时间，这两个时间是活动是否能够进行的约束条件。优化目标是选择最大数量的活动，并且选择的活动之间没有时间上的冲突。换句话说，没有两个活动是重叠的。

数学上，假设有一组活动 $S=\{a_1,a_2,\cdots,a_n\}$，每个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i。目标是找到一个活动子集 A，使得对于所有在 A 中的活动 a_i 和 a_j，都有 $f_i \leqslant s_j$ 或 $f_j \leqslant s_i$，并且 A 中的元素数量最多。

活动选择问题具有贪心选择性质和最优子结构性质，因此可以使用贪心算法进行求解。在贪心算法中，选择结束时间最早的活动，然后从剩余的活动中选择出与已选择的活动不冲突的活动，重复这个过程，直到没有更多的活动可以选择。值得注意的是，活动选择问题的贪心策略并不总是选择开始时间最早的活动，因为开始时间最早的活动可能会持续很长时间，从而排除了其他许多活动。相反，选择结束时间最早的活动可以为后续活动提供更多的机会。

3.3.2　算法步骤

首先按照活动的结束时间进行排序，然后从前到后选择每个活动，如果一个活动的开始时间大于或等于前一个活动的结束时间，就选择这个活动。

具体的算法步骤及步骤说明如下。

(1) 建立活动表，添加进所有的活动，将所有的活动按照结束时间进行排序。

(2) 建立活动子集，从活动表中选择结束时间最早的活动，将其添加到活动子集中，并从原来的活动表中删除。

(3) 按顺序从活动表剩下的活动中选择开始时间大于或等于上一个被选择活动的结束时间的活动，将其添加到活动子集中。

(4) 重复步骤(3)，直到所有的活动都被考虑过。

这个算法的时间复杂度是 $O(n\log n)$，主要是需要对所有的活动进行排序。如果活动已经按结束时间排序，则选择活动的过程是线性的，即 $O(n)$。

3.3.3　案例讲解

【例 3.1】 广东某所学校正在筹备一系列的庆祝活动来庆祝建国 75 周年，由于排练表演节目的需要，学校活动中心收到了 11 个班级的周末使用申请。因为排练场地有限，所以同一时间段内只能有 1 个排练活动。活动中心的负责人必须安排尽可能多的排练活动，确保尽可能多的表演节目得到充分排练。每个排练活动都有一个开始时间和结束时间，表示

为($s[i]$,$f[i]$)。通过安排不同排练活动的开始时间和结束时间,使得这些活动不会在时间上冲突。所有排练活动的开始时间和结束时间如下:(1,4),(3,5),(0,6),(5,7),(3,9),(5,9),(6,10),(8,11),(8,12),(2,14),(12,16)。

算法分析:首先定义一个活动列表 activities 作为输入,将所有排练活动的开始时间和结束时间都存储进去,每个活动是一个元组,包含开始时间和结束时间。如果采用 C 语言编程,可以定义结构体,代码如下所示。

```
//定义活动结构体,包含开始时间和结束时间
typedef struct {
  int start;
  int finish;
} Activity;
```

所有活动的开始时间和结束时间可以用甘特图来直观表示,如图 3.1 所示。

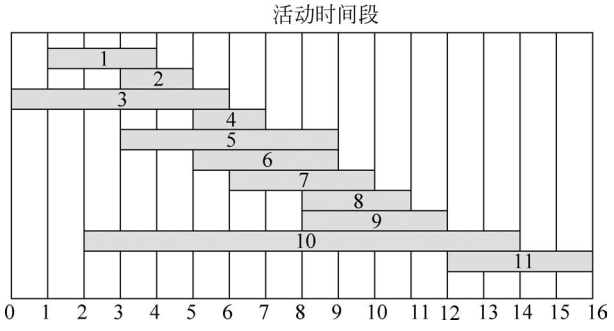


图 3.1 活动排序甘特图

从图 3.1 中可以看出,有些活动的时间段有重叠,意味着这些活动之间是互相冲突的,只能选其中一个。

接着,将活动列表按照活动的结束时间对活动进行排序,C 语言可以使用 qsort()函数,其他编程语言,如 Python 可以使用 sorted()函数。初始化一个空的列表 selected_activities,将结束时间最早的活动添加到这个列表中。结束时间最早的活动是活动 1,开始时间是 1,结束时间是 4,因此首先添加到列表 selected_activities 中。

接下来遍历取出活动 1 后的活动列表 activities,对于每个活动,如果它的开始时间大于或等于活动 1 的结束时间,就将它添加到 selected_activities 列表中。剩余活动中,开始时间大于或等于 4 的有活动 4 和活动 6。但因为活动 4 的结束时间更早,在一开始的排序中排在前面,因此首先被选中并添加进列表 selected_activities。此时已选所有活动的结束时间为 7,继续遍历活动列表 activities,找到活动开始时间大于或等于 7 的活动。由于活动列表已经排过序,因此可以从上一次找到活动的位置开始往后遍历,而不需要从头开始。最终找到活动 8,开始时间为 8,大于活动 4 的结束时间。活动 8 的结束时间为 11,开始时间大于这个结束时间的只有活动 11。最终得到的活动顺序列表 selected_activities 是[(1,4),(5,7),(8,11),(12,16)]。需要注意的是,问题的最优解不止一个时,贪心算法只能找到其中一个解。

活动选择问题的伪代码如图 3.2 所示。

```
输入：activities，活动数组，包含开始时间和结束时间的活动结构体数组
                n，活动的数量
输出：最大的相互兼容的活动集合
void greedyActivitySelector(Activity activities[], int n) {
  // 使用C语言中自带的qsort()函数对活动按结束时间进行排序
  int i = 0;
  printf("(%d, %d), ", activities[i].start, activities[i].finish); // 输出最早结束的一个活动
  for (int j = 1; j < n; j++) {
    // 如果活动的开始时间大于或等于前一个活动的结束时间，则选择该活动
    if (activities[j].start >= activities[i].finish) {
      printf("(%d, %d), ", activities[j].start, activities[j].finish);
      i = j; // 更新 i 为当前选择的活动
    }
  }
}
```

图 3.2　活动选择问题的伪代码

【例 3.2】 某城市的社区中心计划在周末举办一系列文化活动来庆祝当地的传统节日。由于资源有限，同一时间社区中心的大厅只能容纳一项活动。因此，社区中心的组织者需要仔细安排活动，以确保尽可能多的活动能够进行，让社区居民能享受到丰富多彩的节日氛围。每个活动都有一个固定的开始时间和结束时间。

社区中心收到了 10 个不同小组的活动申请，每个活动都有明确的开始时间和结束时间。组织者的目标是选择最大数量的活动进行，同时确保没有时间上的冲突。所有申请活动的开始时间和结束时间如下。

活动 1：(2,3)。

活动 2：(1,4)。

活动 3：(5,8)。

活动 4：(6,10)。

活动 5：(8,9)。

活动 6：(9,11)。

活动 7：(11,14)。

活动 8：(13,15)。

活动 9：(14,16)。

活动 10：(15,17)。

请根据以上信息设计一个活动时间表，使得在不发生时间冲突的情况下，能够安排尽可能多的活动。

算法分析：首先，将所有活动按照结束时间升序排序。这是因为选择结束时间最早的活动将留给后面尽可能多的时间来安排其他活动。按结束时间排序的活动顺序是：活动 1、活动 2、活动 3、活动 5、活动 4、活动 6、活动 7、活动 8、活动 9、活动 10。接着选择活动。

(1) 选择活动 1(结束于 3)。

(2) 下一个可选的是活动 3(结束于 8),因为它在活动 1 结束后开始。

(3) 下一个可选的是活动 5(结束于 9),因为它在活动 3 结束后开始。

(4) 选择活动 6(结束于 11)。

(5) 选择活动 7(结束于 14)。

(6) 选择活动 9(结束于 16)。

因此,根据贪心算法,最多可以安排 6 个活动,分别是活动 1、活动 3、活动 5、活动 6、活动 7 和活动 9。问题的求解代码如下所示。

```
#include <stdio.h>
#include <stdlib.h>
//定义活动结构体
typedef struct {
  int start;
  int end;
} Activity;
//比较函数,用于排序
int compareActivities(const void *a, const void *b) {
  Activity *activityA = (Activity *)a;
  Activity *activityB = (Activity *)b;
  return activityA->end - activityB->end;
}
//选择活动的函数
void selectActivities(Activity activities[], int n, Activity selected[]) {
  //基于完成时间排序活动
  qsort(activities, n, sizeof(Activity), compareActivities);
  //最后选择的活动的结束时间
  int last_end_time = -1;
  //被选中活动的数量
  int count = 0;
  //迭代
  for (int i = 0; i < n; i++) {
    if (activities[i].start >= last_end_time) {
       selected[count++] = activities[i];
       last_end_time = activities[i].end;
    }
  }
  selected[count].start = -1;     //标记数组结束
}
int main() {
  Activity activities[] = {{2, 3}, {1, 4}, {5, 8}, {6, 10}, {8, 9}, {9, 11}, {11, 14}, {13,
15}, {14, 16}, {15, 17}};
  int n = sizeof(activities) / sizeof(activities[0]);
  Activity selectedActivities[n];
  selectActivities(activities, n, selectedActivities);
  printf("Selected activities (start, end):\n");
  for (int i = 0; selectedActivities[i].start != -1; i++) {
     printf("(%d, %d)\n", selectedActivities[i].start, selectedActivities[i].end);
  }
  return 0;
}
```

3.4　任务调度问题

3.4.1　问题概述

任务调度问题是计算机科学中的一类经典问题，它涉及如何有效地安排和分配任务，以达到某种优化目标，如最小化完成时间、最大化利润等。在现实生活中，任务调度问题广泛存在，如工厂生产调度、项目管理、CPU 任务调度等。活动选择问题是任务调度问题的一种特殊情况，它关注的是如何在一系列的活动中选择出最多的互不冲突的活动。而任务调度问题考虑的是如何在最短时间内完成所有任务，并且任务调度问题通常需要考虑更多的约束条件或更多的优化目标，如任务的利润或任务的截止时间。

任务调度问题通常是这样的：给定一组任务和多个处理单元，每个处理单元在任何给定的时间段内只执行一个任务。每个任务都有一个开始时间、结束时间和执行时间，每个任务在任何指定的时间段内都只能由一个处理单元执行，即任务不可分割。目标是在最短的时间内完成所有任务。

这个问题可以表示为图 $G=(V,E)$，其中 V 是任务的集合，E 是项目之间的依赖关系，例如只有先完成某些任务后才能完成对应的任务。这是一个 NP-hard 问题，在问题规模比较小时贪心算法往往可以找到最优解。当问题规模较大时，贪心算法虽然可能不是全局最优的，但是在实际应用中往往非常接近全局最优解，而且计算效率高。

3.4.2　算法步骤

使用贪心算法来解决任务调度问题的基本思想是：总是选择下一个可以在当前处理器上执行的结束时间最早的任务。具体的算法步骤及步骤说明如下。

(1) 将所有任务按照结束时间进行升序排序。

(2) 初始化一个空的调度表，用来记录每个处理单元执行的任务。

(3) 对于每个处理单元，执行以下步骤。

① 从排序后的任务列表中选择结束时间最早的任务。

② 检查是否有处理单元可以完成这个任务，如果有，就将这个任务分配给这个处理单元。

③ 将该任务添加到处理单元的调度表中。

④ 从任务列表中移除该任务。

(4) 重复步骤(3)，直到所有任务都被调度或没有合适的处理单元可用。

贪心算法在任务调度问题上的算法复杂度与活动选择问题是一样的，主要受排序的影响。如果活动已经按结束时间排序，则选择活动的过程是线性的，即 $O(n)$。如果活动没有排序，则需要首先进行排序，这会使时间复杂度变为 $O(n\log n)$。

3.4.3　案例讲解

【例 3.3】 一家中国软件开发公司为了响应国家关于开发国产工业软件的号召，正在加紧完成研发软件任务。当前有 5 个任务需要完成，每个任务在执行过程中都需要完整占

用一个任务小组所有的资源，因此每个任务小组在任何给定的时间内只能执行一个任务。每个任务都有一个任务完成时间和任务截止时间，具体如表 3.1 所示。共有 3 个任务小组，问题的目标是找到所有任务的调度安排，使得所有任务都能在其任务截止时间之前完成，并且尽可能地减少任务完成时间。

表 3.1 任务时间表

任 务 编 号	任务完成时间/周	任务截止时间
任务 1	2	第 4 周
任务 2	5	第 8 周
任务 3	4	第 6 周
任务 4	1	第 3 周
任务 5	3	第 5 周

算法分析：首先定义一个任务列表用于存放任务完成时间和任务截止时间，同时定义任务的数量和任务小组的数量。同样，若是采用 C 语言定义结构体，代码如下所示。

```
//定义任务结构体
typedef struct {
  int execute_time;              //任务完成时间
  int deadline;                  //任务截止时间
  int id;                        //任务编号
} Task;
```

按照任务截止时间对任务进行排序，经过排序后，任务的排列如表 3.2 所示。

表 3.2 任务排序表

任 务 编 号	任务完成时间/周	任务截止时间
任务 4	1	第 3 周
任务 1	2	第 4 周
任务 5	3	第 5 周
任务 3	4	第 6 周
任务 2	5	第 8 周

然后尝试将每个任务分配给一个任务小组。先遍历每个任务，如果对于当前任务，遍历任务小组发现当前任务小组可以在该任务的截止时间之前完成，则更新任务小组的当前时间并标记任务已分配。每个任务小组的当前时间均为 0，表示为 $t[i]$。每个任务均未被标记。首先遍历任务，对于任务 4，任务小组 1 可以完成，将任务 4 分配给任务小组 1，并更新任务小组 1 的当前时间 $t[1]=1$。接着对于任务 1，任务完成时间等于 2，同样可以分配给任务小组 1，而不会超过任务截止时间。更新任务小组 1 的当前时间 $t[1]=3$。对于任务 5，若指派给任务小组 1 则超过任务截止时间，因此分配给任务小组 2，更新任务小组 2 的当前时间 $t[2]=3$。同样，任务 3 分配给任务小组 3，更新任务小组 3 的当前时间 $t[3]=4$。最后一个任务 2 分配给任务小组 1，更新任务小组 1 的当前时间 $t[1]=8$。实现的伪代码如图 3.3 所示。

```
输入：任务数组; 任务数量; 任务小组数量
输出：每个任务分配给哪个任务小组
int main(tasks, NUM_TASKS, NUM_GROUPS) {
  int currentTime[NUM_GROUPS] = {0}; // 初始化每个任务小组的当前时间
  bool assigned[NUM_TASKS] = {false}; // 标记任务是否已被分配
  按照任务截止时间对任务进行排序;
  for (int i = 0; i < NUM_TASKS; i++) {
    for (int j = 0; j < NUM_GROUPS; j++) {
      if (currentTime[j] + tasks[i].execute_time <= tasks[i].deadline) {
        printf("Task %d assigned to Group %d\n", tasks[i].id, j + 1);
        currentTime[j] += tasks[i].execute_time; // 更新任务小组的当前时间
        assigned[i] = true; // 标记任务已被分配
        break;
      }
    }
  }
```

图 3.3　贪心算法实现任务分配的伪代码

最终任务小组的任务调度如图 3.4 所示。

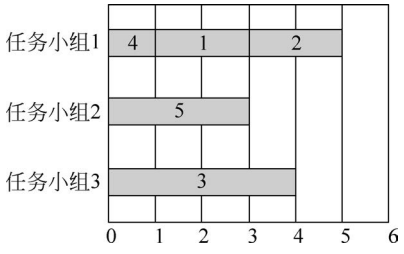


图 3.4　任务调度图

3.5　最小生成树问题

3.5.1　问题概述

最小生成树问题是图论中的一个经典问题，主要出现在网络设计、电路设计、数据通信等领域。这个问题的目标是在一个连通的无向图中找到一棵包含所有顶点的树，使得树中所有边的权重之和最小。以国家为例，一个国家想要连接其所有的城市，使得任何两个城市之间都有一条道路。为了节省成本，国家希望总的建设长度最短。在这种情况下，城市可以被视为图中的顶点，而道路可以被视为连接这些顶点的边。每条道路的建设成本可以被视为边的权重。因此，该问题可以转换为在一个带权的图中找到一棵最小生成树。

解决最小生成树问题的算法有很多，包括 Prim 算法和 Kruskal 算法等。这些算法都是

基于贪心策略的，即在每一步中都选择当前最小的边，以此来保证最后得到的生成树的总权重最小。Kruskal算法的工作原理是按照边的权重对所有的边进行排序，并逐个添加到生成树中，直到生成树包含所有的顶点。而Prim算法则从某个顶点开始，并逐步增加权重最小的边，直到所有的顶点都被包含在生成树中。

3.5.2 算法步骤

如果采用Prim算法，具体的算法步骤及步骤说明如下。

(1) 初始化：选择任意一个顶点作为起始点，将其加入最小生成树的集合中。

(2) 边的选择：在图中找到一条连接已在最小生成树集合中的顶点和不在集合中的顶点的权重最小的边。

(3) 顶点的添加：将这条边的另一端点(不在最小生成树集合中的顶点)加入最小生成树的集合中。

(4) 重复步骤：重复步骤(2)和(3)，直到所有的顶点都被加入最小生成树的集合。

(5) 完成：当所有的顶点都被加入最小生成树的集合时，算法结束，得到的集合即为图的最小生成树。

Prim算法的时间复杂度取决于它的实现方式。①邻接矩阵：如果使用邻接矩阵表示图并使用简单的线性搜索来查找权重最小的边，则Prim算法的时间复杂度为$O(V^2)$，其中V是顶点的数量。②优先队列：如果使用优先队列，如二叉堆来加速权重最小的边的查找过程，并使用邻接表表示图，则Prim算法的时间复杂度可以降低到$O(E\log V)$，其中E是边的数量。③斐波那契堆：使用斐波那契堆作为优先队列可以进一步降低时间复杂度到$O(E+V\log V)$。

总体来说，Prim算法的时间复杂度与图的表示方式和所使用的数据结构有关。在实际应用中，为了获得更好的性能，通常会选择使用优先队列和邻接表的组合来实现Prim算法。

如果采用Kruskal算法，具体的算法步骤及步骤说明如下。

(1) 排序：将图中的所有边按权重从小到大排序。

(2) 初始化：创建一个空集合，用于存放最小生成树中的边。同时，为每个顶点创建一个单独的集合，表示它们所属的连通分量。

(3) 边的选择：按权重从小到大的顺序考虑每条边。对于每条边，检查它连接的两个顶点是否属于同一个连通分量。

(4) 合并：如果两个顶点不在同一个连通分量中，将这条边加入最小生成树的集合，并合并两个顶点所在的连通分量。

(5) 完成：当最小生成树中的边数达到$(V-1)$(其中V是顶点的数量)或所有的边都被考虑过时，算法结束。

Kruskal算法的时间复杂度主要取决于两个步骤：边的排序和连通分量的合并。排序所有的边需要$O(E\log E)$的时间，其中E是边的数量。使用并查集数据结构可以在近乎$O(1)$的时间内判断两个顶点是否在同一个连通分量中，并在$O(\log V)$的时间内合并两个连通分量，其中V是顶点的数量。

综上，Kruskal 算法的总时间复杂度为 $O(E\log E)$。由于 E 可以达到 V^2 的数量级，所以在稠密图中，这可以简化为 $O(E\log V)$，其中 V 是顶点的数量。但在实际应用中，由于图通常不会太稠密，所以 $O(E\log E)$是更常用的表示。

3.5.3　案例讲解

【例 3.4】 每当夏季来临，广东就会迎来用电高峰。为了保证每个地区都能用上电，并且电力部门的维护费用最低，需要设计好城市的电网系统。假设广东某个地方要设计电网系统，已知该地可以分为 6 个区域，每个区域可以看作一个电网节点。连接两个区域的电线可以看作边，电线的长度可以看作边的权值。具体的区域连接示意如图 3.5 所示。在实际的电网设计中，电线的长度反映了电网的建造成本和维护成本。问题的目标是使得所有的区域都能接通电，并且总的电线长度最短。

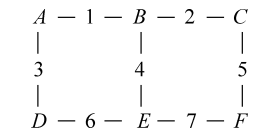


图 3.5　城市区域连接示意

算法分析：首先构建一个邻接矩阵，用于存放各个城市区域之间的距离。采用 Prim 算法来求解，执行以下步骤。

(1) 选择任意一个顶点作为起始点，将其加入最小生成树的集合中。假设选择顶点 A 加入最小生成树的集合。

(2) 在图中找到一条连接已在最小生成树集合中的顶点和不在集合中的顶点的权重最小的边。由于最小生成树集合中只有顶点 A，因此找到权重最小的边 AB，将边 AB 的另一端点 B 加入最小生成树集合中。这部分的实现代码如下所示。

```
//从集合中选择最小值的顶点的函数
   int selectMinVertex(int weight[], bool inMST[]) {
     int min = INF;                    //初始化最小值为无穷大
     int vertex;                       //用于存储权重最小的顶点的索引
     for(int i = 0; i < V; i++) {      //遍历所有顶点
       if(inMST[i] == false && weight[i]< min) { //如果顶点不在最小生成树的集合(MST)中且其
//权重小于当前的最小值
         min = weight[i];              //更新最小值
         vertex = i;                   //更新最小权重的顶点索引
       }
     }
     return vertex;                    //返回权重最小的顶点的索引
   }
```

(3) 继续寻找边，此时最小生成树集合中有顶点 A 和 B，因此找到权重最小的边 BC，将边 BC 的另一端点 C 加入最小生成树集合中。

(4) 重复步骤，最小生成树集合中有顶点 A、B 和 C，找到权重最小的边 AD，将边 AD 的另一端点 D 加入最小生成树集合中。

(5) 重复步骤，最小生成树集合中有顶点 A、B、C 和 D，找到权重最小的边 BE，将边 BE 的另一端点 E 加入最小生成树集合中。

(6) 重复步骤，最小生成树集合中有顶点 A、B、C、D 和 E，找到权重最小的边 CF，将边 CF 的另一端点 F 加入最小生成树集合中。

此时，所有的顶点都已经加入最小生成树中，算法结束，最小生成树的连接图如图 3.6 所示。

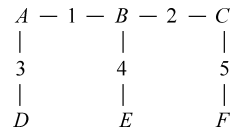


图 3.6　最小生成树的连接图

最终最小生成树集合为$\{AB, BC, AD, BE, CF\}$，总的权值是 1＋2＋3＋4＋5＝15。Prim 算法的具体实现代码如下。

```
//  Prim算法的主要实现
   void  prim(int graph[V][V]) {
     int parent[V];                          //用于存储每个顶点的父顶点
     int weight[V];                          //用于存储从 MST 到每个顶点的最小权重
     bool inMST[V];                          //布尔数组,用于标记顶点是否已经在 MST 中
     for(int i = 0; i < V; i++) {            //初始化所有顶点的权重为无穷大,MST 集合为空
        weight[i] = INF;
        inMST[i] = false;
        parent[i] = -1;                      //初始化父节点为 -1,表示没有父节点
     }
     weight[0] = 0;                          //将第一个顶点的权重设置为 0,以便从它开始
     for(int i = 0; i < V - 1; i++) {        //遍历除一个顶点之外的所有顶点
        int U = selectMinVertex(weight, inMST);                    //选择当前权重最小的顶点
        inMST[U] = true;                     //将该顶点标记为已加入 MST
        //更新权重和父节点
        for(int j = 0; j < V; j++) {         //遍历所有顶点
           //如果 U 和 j 之间有边,j 不在 MST 中,且 U 到 j 的权重小于 j 的当前权重
           if(graph[U][j] != 0 && inMST[j] == false && graph[U][j] < weight[j]) {
              weight[j] = graph[U][j];       //更新 j 的权重
              parent[j] = U;                 //设置 U 为 j 的父节点
           }
        }
     }
```

【例 3.5】 在环保和节能的大背景下，一个新开发的生态小镇计划建设一个高效且成本最低的水资源管理系统。小镇被分为 5 个主要区域，每个区域都需要被纳入水资源管理网络。每两个区域之间可以通过管道连接，管道的长度代表建设和维护成本(权值)。为了保证水资源的有效管理，需要将所有区域通过管道网络连接起来，同时保证总的管道长度(即成本)最小。

小镇的区域和管道可能的布局如下所示。

(1) 区域 A 到区域 B 的管道长度为 4 单位。

(2) 区域 A 到区域 C 的管道长度为 1 单位。

(3) 区域 B 到区域 C 的管道长度为 2 单位。

(4) 区域 B 到区域 D 的管道长度为 5 单位。

(5) 区域 C 到区域 D 的管道长度为 8 单位。

(6) 区域 C 到区域 E 的管道长度为 10 单位。

(7) 区域 D 到区域 E 的管道长度为 3 单位。

请设计一个管道网络，使得所有区域都通过管道连接起来，同时总的管道长度最小。这就需要找到覆盖这个小镇的最小生成树。

算法分析：这个问题是一个经典的最小生成树问题，其中节点代表小镇的区域，边代表

区域间的管道，边的权值代表管道的长度。使用 Kruskal 算法来解决最小生成树问题的过程涉及对图中的边进行排序，并依次选择不形成环路的边，直到连接了所有的节点。可以按照以下步骤来求解提出的管道网络问题。

(1) 按照长度排序，边有(*AC*,1),(*BC*,2),(*DE*,3),(*AB*,4),(*BD*,5),(*CD*,8),(*CE*,10)。

(2) 初始时，每个区域(*A*,*B*,*C*,*D*,*E*)是一棵独立的树(集合)。按照边的权重顺序，从最小的开始选择边。对于每条边，检查它的两个端点是否属于同一棵树：如果属于不同的树，则选择这条边，并将这两棵树合并为一棵树；如果属于同一棵树，则跳过这条边(选择它会形成环路)。根据这个要求，选择边 *AC*，合并 *A* 和 *C*。接着选择边 *BC*，合并 *B* 到 *AC* 树中。选择边 *DE*，合并 *D* 和 *E*。选择边 *AB* 被跳过，因为 *A* 和 *B* 已经在同一棵树中。选择边 *BD*，合并 *DE* 树到 *ABC* 树中，此时形成了覆盖所有区域的树。

(3) 算法截止，最小生成树包括边 *AC*,*BC*,*DE*,*BD*。总管道长度＝1＋2＋3＋5＝11 单位。

Kruskal 算法的主体求解代码如下。

```
//Kruskal 算法实现
void kruskal(Edge edges[], int edgesCount, Edge mst[], int *mstSize) {
  //初始化并查集
  for (int i = 0; i < 1000; i++) {
     parent[i] = i;
  }
  //将边按权重升序排序
  qsort(edges, edgesCount, sizeof(Edge), compareEdges);
  *mstSize = 0;
  for (int i = 0; i < edgesCount; i++) {
     int vertex1 = edges[i].vertex1;
     int vertex2 = edges[i].vertex2;
         //选择边构造最小生成树
     if (find(vertex1) != find(vertex2)) {
        unionSet(vertex1, vertex2);
        mst[(*mstSize)++] = edges[i];
     }
  }
}
```

上述实现的代码使用了并查集，C 语言通过使用数组和结构体来实现，对应的方法代码如下。

```
//定义边的结构体
typedef struct {
  int vertex1;
  int vertex2;
  int weight;
} Edge;
//并查集的父节点数组
int parent[1000]; //假设顶点数量不超过 1000
```

```
//查找并查集中的根节点
int find(int i) {
   while (parent[i] != i) {
     i = parent[i];
   }
   return i;
}
//合并并查集中的两个集合
void unionSet(int i, int j) {
  int ri = find(i);
  int rj = find(j);
  if (ri != rj) {
     parent[ri] = rj;
  }
}
//边的比较函数,用于排序
int compareEdges(const void * a, const void * b) {
   Edge * edgeA = (Edge * )a;
   Edge * edgeB = (Edge * )b;
   return edgeA -> weight - edgeB -> weight;
}
```

3.6 单源最短路径问题

3.6.1 问题概述

单源最短路径问题同样也是图论中的一个经典问题,它的目标是在一个带权重的有向图或无向图中,找出从一个指定的源顶点到其他所有顶点的最短路径。在这个问题中,图的顶点可以表示各种实体,如城市、交叉路口等。边可以表示实体之间的连接,如道路、航线等,边的权重可以表示连接的成本,如距离、时间、费用等。

贪心算法在解决这个问题上起到了关键的作用。在单源最短路径问题中,贪心算法在每一步都选择当前距离起始顶点最近的那个顶点,并更新它的邻居的距离。Dijkstra 算法是解决单源最短路径问题的最著名的贪心算法。它从起始顶点开始,逐步选择距离最短的顶点,并更新其邻居的距离。这个过程持续到所有的顶点都被访问为止。

3.6.2 算法步骤

Dijkstra 算法具体的算法步骤及步骤说明如下。

(1) 初始化:将所有顶点标记为未访问。创建两个集合,一个是已经找到最短路径的顶点集合 SPT,另一个是尚未找到最短路径的顶点集合。设置起始顶点的最短路径值为 0,其他所有顶点的最短路径值为无穷大,表示还不知道从起始顶点到其他顶点的距离,这可以通过一个数组或优先队列来实现。将起始顶点添加到 SPT 集合。

(2) 选择最小距离的顶点:从尚未处理的顶点集合中选择一个距离最小的顶点。

(3) 更新距离：对于顶点 u 的每一个未访问的邻居 v，如果起点通过 u 到 v 的距离小于已知的起点到 v 的距离，则更新起点到 v 的距离。

(4) 标记已访问：将当前顶点标记为已访问的顶点，添加到 SPT 集合。

(5) 重复：返回步骤(2)，直到所有的顶点都被访问。

Dijkstra 算法是一种有效解决单源最短路径问题的方法，然而，Dijkstra 算法也有其局限性。当图的边权重有负值时，Dijkstra 算法可能无法找到正确的最短路径，此时应该使用 Bellman-Ford 算法。

Dijkstra 算法的时间复杂度取决于它的实现。若使用邻接矩阵和线性数组来表示图和最短路径集合，时间复杂度为 $O(V^2)$，其中 V 是顶点的数量。若使用优先队列，如二叉堆可以提高算法的效率。当使用优先队列存储未访问的顶点时，每次选择最小距离的顶点的时间复杂度为 $O(\log V)$，并且更新一个顶点的距离也是 $O(\log V)$。因此，对于每条边和每个顶点，算法的总时间复杂度为 $O(E+V\log V)$，其中 E 是边的数量。

3.6.3　案例讲解

【例 3.6】 近期，旅游业开始恢复，旅游景区开始有游客涌入。大多数游客都希望一次游玩可以参观更多的景点，尽量减少在路上的时间。一个景区往往有多个景点，每个景点之间的路线距离不同，有些景点之间还没有直达的路线。因此游客在出行时往往需要考虑选择最短距离的出行路线。假设粤东某旅游景区中有 6 个旅游景点，分别是 A、B、C、D、E 和 F。每个旅游景点之间的距离如图 3.7 所示，其中没有直接相连的两个景点表示对应的两个景点之间没有直达路线。有个外地游客想从景点 A 出发，去往不同景点游玩，请计算他到每个景点的最短路径。

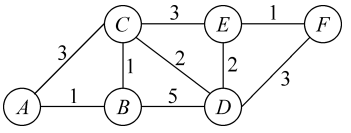


图 3.7　景点距离图

以旅游景点 A 为起点应用 Dijkstra 算法，执行以下步骤。

(1) 将所有顶点标记为未访问，设置起始顶点 A 的最短路径长度为 0，设置其他所有顶点的最短路径长度为无穷大。

(2) 选择最小距离的顶点，初始时只有起点 A 被选择，添加进 SPT 集合。

(3) 更新从起点 A 到其他未访问的邻居顶点的距离。此时起点 A 未访问的邻居有顶点 B 和 C，通过顶点 A 到顶点 B 和 C 的距离 AB、AC 分别为 1 和 3，均小于无穷大。因此更新从顶点 A 到顶点 B、C 的距离 dist[B]=1，dist[C]=3。

(4) 选择最小距离的顶点，dist[B]最小，将顶点 B 添加进 SPT 集合。更新距离，顶点 B 未访问的顶点为 C 和 D，通过 B 到达 C 的距离 dist[C]=2<3，更新数据，同理，dist[D]=1+5=6。

(5) 继续从尚未处理的顶点集合中选择最小距离的顶点，dist[C]最小，将顶点 C 添加进 SPT 集合。顶点 C 未访问的顶点为 D 和 E，通过 C 到达 D 的距离 dist[D]=2+2=4<6，更新距离。同理，dist[E]=2+3=5。

(6) 继续从尚未处理的顶点集合中选择最小距离的顶点，dist[D]最小，将顶点 D 添加进 SPT 集合。顶点 D 未访问的顶点为 E 和 F，通过 D 到达 E 的距离 dist[E]=4+2=6>5，因此不更新。通过 D 到达 F 的距离 dist[F]=4+3=7，更新距离。

(7) 继续从尚未处理的顶点集合中选择最小距离的顶点，dist[E]最小，将顶点 E 添加进 SPT 集合。顶点 E 未访问的顶点为 F，通过 E 到达 F 的距离 dist[F]=5+1=6<7，更新距离。

(8) 将剩下的唯一顶点 F 添加进 SPT 集合。所有的站点都被访问，算法结束。最终通过 Dijkstra 算法找到了从 A 点到所有其他景点的最短路径和最短路径长度，如表 3.3 所示。实现的伪代码见图 3.8。

表 3.3 A 点到其他景点路径表

景　　点	最短路径长度	最 短 路 径
B	1	A-B
C	2	A-B-C
D	4	A-B-C-D
E	5	A-B-C-E
F	6	A-B-C-E-F

```
输入：图 G, 起始顶点 src
输出：起始顶点 src 到所有顶点的距离
void  dijkstra(int graph[V][V], int src) {
    初始化距离数组 dist，使每个顶点到起始顶点的距离为无穷大
    初始化一个集合 sptSet 表示已经处理的顶点
    距离起始顶点到自己的距离设置为 0
    for (int count = 0; count < V - 1;  count++) { // 对于每个顶点
      从尚未处理的顶点中选择距离最短的顶点 u
      sptSet[u] = 1; // 标记选择的顶点为已处理
      for (int v = 0; v < V; v++) // 遍历所有顶点 v
        // 如果 v 未被处理，且 u 到 v 有边，且通过 u 到 v 的距离比当前已知的距离更短
        if (!sptSet[v] &&  graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v]  < dist[v])
          dist[v] = dist[u] +  graph[u][v]; // 更新距离 dist[v]
    }
    printSolution(dist); // 打印结果数组 dist
}
```

图 3.8 伪代码实现 Dijkstra 算法

【例 3.7】 近年来，一个新开发的生态公园吸引了众多游客。这个公园内设有 5 个独特的景点，分别为 X、Y、Z、W 和 V，它们通过不同的路径连接。由于生态保护的原因，一些景点之间没有直接连接的路径。假设一位游客计划从景点 X 开始游览，请计算出这位游客从 X 出发访问所有景点的最短路径及其距离。

已知有相连路径的景点之间的距离如下。

X 到 Y 的距离：6 单位。

X 到 Z 的距离：3 单位。

Y 到 Z 的距离：2 单位。

Y 到 W 的距离：5 单位。

Z 到 Y 的距离：1 单位。

Z 到 W 的距离：3 单位。

Z 到 V 的距离：4 单位。

W 到 V 的距离：2 单位。

V 到 W 的距离：1 单位。

算法分析：如果将每个景点视为一个节点，则路径视为连接这些节点的边。因此整个路径网络可以视为一幅加权图(边的权重是距离)，并且还是非完全图(不是每两个节点都有边连接)。目标是找到从景点 X 出发，到达所有其他景点的最短路径，并计算最短路径的距离。

采用 Dijkstra 算法，首先将除节点 X 外的其他节点的最短距离设为无穷大，并使用一个优先队列(或其他结构)来跟踪待处理的节点，以及它们当前的最短距离。最初，队列只包含起点 X。当优先队列非空时，从队列中取出当前距离最短的节点作为当前节点。这部分实现的代码如下所示：

```
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index; //初始化最小值为最大整数,min_index 为最小距离顶点的索引

    for (int v = 0; v < V; v++)    //遍历所有顶点
         if (sptSet[v] == 0 && dist[v] <= min) //如果顶点 v 未处理且 v 到源的距离小于或等
                                               //于当前最小值
              min = dist[v], min_index = v;    //更新最小值和最小值顶点的索引

    return min_index;              //返回最小距离顶点的索引
}
```

对于当前节点的每一个邻接节点，如果通过当前节点到达邻接节点的路径比已知的最短路径更短，则更新该路径的距离，并将邻接节点加入优先队列，重复这个过程直到优先队列为空。完成后，每个节点的最短距离将被计算出来。相关的代码如下所示：

```
void dijkstra(int graph[V][V], int src) {
    int dist[V];            //dist[i]将保存从 src 到 i 的最短距离
    int sptSet[V];          //sptSet[i]为真如果顶点 i 包含在最短路径树中或最短距离确定
    //初始化所有距离为无穷大,sptSet[]为假
    for (int i = 0; i < V; i++)
         dist[i] = INT_MAX, sptSet[i] = 0;
    //起点到自身的距离总是 0
    dist[src] = 0;
    //找到所有顶点的最短路径
    for (int count = 0; count < V - 1; count++) {
         //从未处理的顶点集合中选择最小距离顶点
         int u = minDistance(dist, sptSet);
         //标记该顶点已经被处理
         sptSet[u] = 1;
         //更新相邻顶点的距离值
         for (int v = 0; v < V; v++)
              if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                   && dist[u] + graph[u][v] < dist[v])
                   dist[v] = dist[u] + graph[u][v];
    }
    //打印构建的距离数组
    printSolution(dist);
}
```

根据 Dijkstra 算法的计算结果，从起点 X 到其他节点的最短路径及其距离如下。

X 到 X 的最短距离：0 单位(起点)。

X 到 Y 的最短距离：4 单位。

X 到 Z 的最短距离：3 单位。

X 到 W 的最短距离：6 单位。

X 到 V 的最短距离：7 单位。

3.7 哈夫曼编码问题

3.7.1 问题概述

在信息论和数据压缩中，哈夫曼编码是一种广泛使用的熵编码技术。它的核心思想是基于字符出现的频率为每个字符分配一个独特的变长编码。频繁出现的字符被分配较短的编码，而较少出现的字符被分配较长的编码。这种方法确保了编码后的数据的平均长度最小，从而实现了数据的有效压缩。哈夫曼编码的优势在于它能够为数据集提供一种高效的编码方式，从而实现数据的有效压缩。此外，由于它是无损的，因此原始数据可以完全恢复，而不会丢失任何信息。这使得哈夫曼编码在许多应用中，如文件压缩、图像压缩和音频压缩等，都得到了广泛的应用。

哈夫曼编码问题可以用一个集合 S 来表示所有的元素，其中每个元素 i 都有一个对应的频率 f_i。需要为每个元素 i 分配一个唯一的编码 c_i，编码的长度为 l_i。目标是最小化整个文件的编码长度，即最小化 $\sum f_i l_i$，其中求和是对所有 $i \in S$ 进行的。

3.7.2 算法步骤

哈夫曼编码的基础是哈夫曼树，也称为最优二叉树。它是一种特殊类型的二叉树，其中每个叶节点都与输入数据中的字符相关联，并且这些字符的频率与其在树中的权重相对应。构建哈夫曼树的过程是一个迭代过程，从创建一个节点列表开始，每个节点代表一个字符及其频率。在每次迭代中，选择两个频率最低的节点并合并它们，直到只剩下一个节点，即树的根节点。一旦哈夫曼树被构建完成，就可以为数据集中的每个数据项分配一个唯一的编码。从根节点到每个叶节点的路径定义了该叶节点对应数据项的哈夫曼编码。通常哈夫曼树的左分支被编码为 0，右分支被编码为 1。构建哈夫曼树的步骤如下。

(1) 为数据集中的每个项创建一个节点，并将这些节点按照它们的频率排序。

(2) 选择两个频率最低的节点。

(3) 创建一个新的父节点，并将这两个节点作为子节点。新节点的频率是其两个子节点的频率之和。

(4) 从列表中删除这两个节点，并添加新的父节点。

重复步骤(2)～(4)，直到只剩下一个节点，即哈夫曼树的根节点。

哈夫曼编码问题的具体算法步骤如下。

(1) 初始化：为数据集中的每个项创建一个节点，并将这些节点按照它们的频率排序。

这些节点被放在一个优先队列中。

(2) 构建哈夫曼树：当优先队列中的节点数量大于 1 时，构建哈夫曼树。

(3) 编码：从哈夫曼树的根节点开始，为左分支分配编码 0，为右分支分配编码 1。从根到每个叶节点的路径定义了该叶节点对应项的哈夫曼编码。

(4) 解码：从编码的位流开始，从哈夫曼树的根节点开始遍历，根据位流选择左或右分支，直到达到叶节点，从而得到原始数据项。

构建哈夫曼树的时间复杂度是 $O(\log n)$，其中 n 是数据项的数量。这是因为每次从优先队列中取出两个节点并插入一个新节点的操作都需要 $O(\log n)$时间，而这个操作需要执行 $n-1$ 次。编码和解码的时间复杂度与输出的编码长度成正比，即 $O(L)$，其中 L 是输出编码的总长度。

3.7.3　案例讲解

【例 3.8】 在当今的数字化世界中，数据量正在以惊人的速度增长。每天都有无数的文本文件被创建并在网络上分享。在一个文本文件中，每个字符都可以被看作一个元素，每个元素都有一个对应的 ASCII 码。然而，不同的字符在文件中出现的频率是不同的。例如，英文中的 e 字符出现的频率最高，而 z 字符出现的频率最低。因此，如果为每个字符分配一个固定长度的编码，那么这将导致一些频繁出现的字符占用了大量的存储空间。为了解决这个问题，需要为每个字符分配一个唯一的编码，使得整个文件的编码长度最小。这样就可以有效地压缩文件，节省存储空间。为了解决这个问题，几位创业的大学生决定开发一个基于哈夫曼编码算法的压缩软件，该软件首先要解决的是文本文件的压缩。假设有一个文本文件包含了以下字符串"ABBRACADABRA"，要为其字符创建哈夫曼编码。

算法分析：首先，需要计算每个字符的频率：{A:5,B:3,R:2,C:1,D:1}。将所有字符按照它们的频率由高到低进行排序，并将它们视为单独的节点，得到{A(5),B(3),R(2),C(1),D(1)}。由于节点的数量大于 1，开始构建哈夫曼树。

选择两个频率最低的节点，创建一个新的父节点，并将这两个节点作为子节点。父节点的权重为这两个节点的权重之和，即 C(1)＋D(1)＝CD(2)，并更新到队列中。现在的节点和它们的权重为{A(5),B(3),R(2),CD(2)}。对应的哈夫曼树如图 3.9 所示。

图 3.9　节点 C 和节点 D 合并

再次选择两个频率最低的节点，创建一个新的父节点，并将这两个节点作为子节点。新的父节点的权重为 R(2)＋CD(2)＝RCD(4)，并更新到队列中。现在的节点和它们的权重为{A(5),B(3),RCD(4)}。对应的哈夫曼树如图 3.10 所示。

再次选择两个频率最低的节点，创建一个新的父节点，并将这两个节点作为子节点。新的父节点的权重为 B(3)＋RCD(4)＝RCD(7)，并更新到队列中。现在的节点和它们的权重为{A(5),BRCD(7)}。对应的哈夫曼树如图 3.11 所示。

最后合并剩下的两个节点，得到根节点的权重为 A(5)＋BRCD(7)＝ABRCD(12)，并更新到队列中。对应的哈夫曼树如图 3.12 所示。

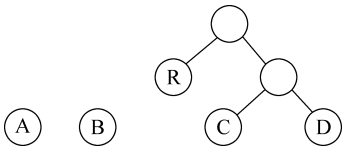


图 3.10 合并节点 R

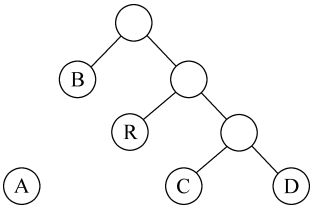


图 3.11 合并节点 B

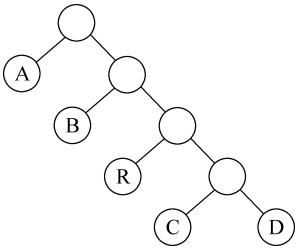


图 3.12 合并节点 A

为每个字符分配哈夫曼编码。从根节点到每个叶节点的路径定义了该叶节点对应字符的哈夫曼编码。例如，左分支为 0，右分支为 1，得到一种可能的编码如下：{A:0，B:10，R:110，C:1110，D:1111}。

假设使用固定长度的编码来表示每个字符，由于有 5 个不同的字符{A，B，R，C，D}，因此至少需要 3 位来表示每个字符，因为 2 位只能表示 4 个不同的字符。原始编码可能如下：{A:000，B:001，R:010，C:011，D:100}。字符串"ABBRACADABRA"的长度为 12 个字符。因此，使用原始编码，总长度为 12×3＝36 位。而对应的哈夫曼编码为 A(0) B(10) B(10) R(110) A(0) C(1110) A(0) D(1111) A(0) B(10) R(110) A(0)＝0 10 10 110 0 1110 0 1111 0 10 110 0。总长度为 25 位，节省了 11 位。

在上面的例子中，如果每个字符的频率为{A:5，B:2，R:2，C:1，D:1}，则合并节点 C 和 D 之后会出现 3 个权重一样的节点，即 B(2)、R(2)和 CD(2)。此时采用不同的合并方式，如合并节点 B 和节点 R，所得到的哈夫曼编码也会不一样。但计算方法是一样的，同样可以有效压缩编码。并且，使用程序求哈夫曼编码时有两种方法：一种是从叶节点一直找到根节点，逆向记录途中经过的标记；另一种是从根节点出发，一直到叶节点，记录途中经过的标记。两种寻找方式带来的哈夫曼编码也可能不一样。实现的伪代码如图 3.13 所示。

```
输入：字符数组, 频率数组, 大小
输出：哈夫曼编码
void HuffmanCodes(char data[], int  freq[], int size) {
    Node* left, * right, * top;
    创建最小堆 minHeap；
    将每个字符和对应的频率插入最小堆中；
    构建最小堆；
    while (!isSizeOne(minHeap)) {  // 重复以下步骤，直到最小堆的大小为 1
      提取两个频率最低的节点 left 和 right；
      创建一个新节点 top，其频率为两个子节点的频率之和；
      设置左右子节点；
      插入新节点 top 到最小堆 minHeap 中
    }
    int arr[100], top2 = 0; // 创建数组存储哈夫曼编码
  }
```

图 3.13 哈夫曼编码伪代码实现

3.8　小结

在实际应用中，贪心算法已被广泛用于各种领域，如网络路由、数据压缩和任务调度等。尽管它不一定总是提供最优解，但由于其高效性，它仍然是许多实际问题的首选方法。在算法设计中，贪心策略的优势在于其简洁性和效率。由于它不需要评估所有可能的解决方案，因此通常比其他算法更快。然而，这种速度的提升往往是以牺牲最终解的完美性为代价的。但在很多情况下，它提供的解决方案是足够好的。在某些情况下，求解最优解的时间复杂度或求解困难度较高，因此快速获得一个较好的近似解也是可以接受的。

总体来说，贪心算法是算法设计中的一个强大工具，它提供了一种简单而高效的方法来解决许多优化问题。然而，使用贪心策略时，必须小心确保问题具有适当的性质，以确保得到的解决方案是有效的。在理解了其基本原理和局限性后，贪心算法可以成为每位算法设计师工具箱中的一个宝贵工具。

习题

1. 在下列(　　)情况下，贪心算法是最有效的？

A. 当问题的全局最优解可以由局部最优解组成时

B. 当问题的解空间非常大时

C. 当问题有多个解时

D. 当问题需要深度搜索时

2. 以下(　　)问题不适合使用贪心算法？

A. 分数背包　　B. 0-1 背包　　C. 最小生成树　　D. 最短路径

3. 哈夫曼编码的贪心算法所需的计算时间复杂度为(　　)。

A. $O(n2^n)$　　B. $O(n\log n)$　　C. $O(2^n)$　　D. $O(n)$

4. 给定一段文本中的 4 个字符(a，b，c，d)。设 a 和 b 具有最低的出现频率。下列(　　)组编码是这段文本可能的哈夫曼编码？

A. a:000,b:001,c:01,d:1　　B. a:000,b:001,c:01,d:11

C. a:000,b:001,c:10,d:1　　D. a:010,b:001,c:01,d:1

5. 用于求最小生成树的 Prim 算法和 Kruskal 算法都是基于(　　)思想设计的算法。

A. 分治法　　B. 穷举法　　C. 贪心算法　　D. 回溯算法

6. 采用贪心算法的最优装载问题的主要计算量在于将集装箱依其质量从小到大排序，故算法的时间复杂度为(　　)。

A. $O(n2^n)$　　B. $O(n\log n)$　　C. $O(2^n)$　　D. $O(n)$

7. 下列问题中不能使用贪心算法解决的是(　　)。

A. 单元最短路径　　B. N 皇后问题

C. 最小花费生成树　　D. 背包问题

8. 能用贪心算法求最优解的问题，一般具有的重要性质为(　　)。

A. 最优子结构　　B. 重叠子问题与贪心选择

C. 最优子结构与贪心选择　　D. 贪心选择

9. 贪心算法通常适用于(　　)类型的问题。(多选)

A. 当问题的局部最优解也是全局最优解时

B. 当问题可以通过选择当前最好的选项来解决时

C. 当问题的解决方案需要回溯时

D. 当问题的解决方案可以通过解决子问题来得到时

10. 在老电影《007之生死关头》中有一个情节,007被毒贩抓到一个鳄鱼池中心的小岛上,他通过直接踩着池子里一系列鳄鱼的大脑袋跳上岸逃脱。设鳄鱼池是长宽分别为100米的方形,中心坐标为(0,0),且东北角坐标为(50,50)。池心岛是以(0,0)为圆心、直径为15米的圆。给定池中分布的鳄鱼的坐标,以及007一次能跳跃的最大距离,需要告诉他是否有可能逃出生天。请设计一个判断算法。

11. 一位家长想要给孩子们一些小饼干,但是每个孩子最多只能给一块饼干。对每个孩 i,都有一个胃口值 $g[i]$,这是能让孩子们满足胃口的饼干的最小尺寸。并且每块饼干 j,都有一个尺寸 $s[j]$。如果 $s[j]\geqslant g[i]$,可以将这个饼干 j 分配给孩子 i,这个孩子就会得到满足。目标是尽可能满足尽量多数量的孩子,并输出这个最大数值。

12. 鼹鼠是整洁、勤劳的动物,喜欢把它们的地下住所安排得井井有条。为了实现这一点,鼹鼠用隧道将地下各个洞穴连接起来,这样就有了一种从一个洞穴到任何其他洞穴的独特方式。两个洞穴之间的距离是从一个洞穴到另一个洞穴途中经过的其他洞穴数量。问鼹鼠应该如何用隧道连接,才能使最远的两个洞穴之间的距离尽可能小,并且仍然可以从其他洞穴到达每个洞穴。

13. 给定一个长度为 n 的整数数组 height。有 n 条垂线,第 i 条线的两个端点是 $(i,0)$ 和 $(i,\text{height}[i])$。找出其中的两条线,使得它们与 x 轴共同构成的容器可以容纳最多的水,要求不能倾斜容器。最终返回容器可以存储的最大水量。

14. 学校的会议中心每天都会有许多活动,有时这些活动的计划时间会发生冲突,需要选择出一些活动进行举办。小刘的工作就是安排学校会议中心的活动,每个时间最多安排一个活动。现在小刘有一些活动计划的时间表,他想尽可能安排更多的活动,请问他该如何安排。

输入格式:

第一行是一个整型数 n,表示共有 n 个活动。

随后的 n 行,每行有两个正整数 B_i,E_i $(0\leqslant B_i, E_i < 10\,000)$,分别表示第 i 个活动的起始时间与结束时间 $(B_i \leqslant E_i)$。

输出格式:

输出最多能够安排的活动数量。

输入样例:

```
3
1 10
9 11
11 20
```

输出样例:

```
2
```