****

算法分析与设计实验报告

|  |  |
| --- | --- |
| 题 目： | 算法分析与设计 |
| 学生姓名： | 曹家赫 |
| 学 号： | 8210222526 |
| 指导教师： | 阚世超 |
| 学 院： | 计算机学院 |
| 专业班级： | 大数据2204班 |
| 完成时间： | 2024/5/10 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指导老师评定： |  |  | 签名： |  |

目录

[实验一](#_Toc12191)

[section1 1](#_Toc12191)

[1. 题目内容 1](#_Toc17844)

[2. 概要设计 1](#_Toc17448)

[3. 详细设计 2](#_Toc24866)

[4. 测试程序的运行结果 4](#_Toc29157)

[5. 心得体会 4](#_Toc13939)

[section2 5](#_Toc25382)

[1. 需求分析 5](#_Toc29758)

[2. 概要设计 5](#_Toc20507)

[3. 详细设计 6](#_Toc24009)

[4. 测试程序的运行结果 8](#_Toc9442)

[5. 心得体会 9](#_Toc17246)

[section3 1](#_Toc26682)0

[1. 需求分析 1](#_Toc21538)0

[2. 概要设计 1](#_Toc19342)1

[3. 详细设计 1](#_Toc24719)2

[4. 测试程序的运行结果 1](#_Toc27051)4

[实验二 1](#_Toc26682)5

[1. 需求分析 1](#_Toc21538)5

[2. 概要设计 1](#_Toc19342)6

[3. 详细设计 16](#_Toc24719)

[4. 测试程序的运行结果 2](#_Toc27051)0

[实验三 2](#_Toc26682)1

[1. 需求分析 2](#_Toc21538)1

[2. 概要设计 21](#_Toc19342)

[3. 详细设计 2](#_Toc24719)2

[4. 测试程序的运行结果 25](#_Toc27051)

实验一

**section1**

1. 题目内容

在无序数组中查找第k大的数，要求最差情况下满足O(n)的时间复杂度，分析每一步算法的复杂度。

输入：

第一行输入m, k，表示输入的数组有m个数，查找第k大的数

第二行输入m个整数

输出:

第k大的数

测试数据:

* 输入（input）：
* 3，2
* 4，1，3
* 输出（output）：
* 3

1. 概要设计

(1)最差情况下需要满足O(n)的时间复杂度，根据常用且稳定的排序算法时间复杂度：归并排序-O(n log n);快速排序：O(n^2);堆排序：O(n log n);计数排序：O(n+k)……

(2)本程序使用计数排序完成任务。

(3)计数排序算法思想：计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。用来计数的数组C的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），然后进行分配、收集处理：

① 分配。扫描一遍原始数组，以当前值minvalue作为下标，将该下标的计数器增加1。  
 ② 收集。扫描一遍计数器数组，按顺序把值收集起来。

1. 详细设计

(1)计数排序基本步骤：

①找出待排序的数组中最大和最小的元素：这一步是为了确定计数数组的大小。

②统计数组中每个值为i的元素出现的次数：创建一个计数数组，数组的长度为最大值和最小值之差加一。遍历原始数组，以元素值减去最小值作为新数组的下标，对应下标的计数加一。

③对所有的计数累加：从计数数组的第一个元素开始，每一项和前一项相加。这一步是为了确定每个元素在排序后数组中的位置。

④反向填充目标数组：根据计数数组，从右至左遍历原始数组，将元素放在输出数组的正确位置上，并更新计数数组。

(2)主要操作的实现。（伪代码）

function findKthLargest(nums, k):

找出数组中的最大值和最小值

初始化计数数组

计算每个元素的出现次数

从后向前遍历计数数组，以找到第k大的元素

累加计数数组中的元素直到达到或超过k

返回第k大的元素

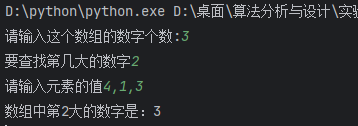
(3) python代码实现

def find\_num(nums , k):  
 max\_num = max(nums)  
 min\_num = min(nums)  
 # 初始化数组  
 count = [0] \* (max\_num - min\_num + 1)  
 for num in nums:  
 count[num - min\_num] += 1  
 idx = len(count) - 1  
 total = 0  
 while(total < k):  
 total +=count[idx]  
 idx -=1  
 return idx + min\_num + 1

m = int(input(f'请输入这个数组的数字个数:'))  
k = int(input(f'要查找第几大的数字'))  
array = list(map(int,input('请输入元素的值').split()))  
if k > m :  
 print("输入有误")  
else:  
 find\_num(array,k)  
 print(f'数组中第{k}大的数字是：{find\_num(array,k)}')

1. 测试程序的运行结果

以下为实例调试：



1. 心得体会

计数排序需要两个额外的数组用来对元素进行计数和保存排序的输出结果，所以空间复杂度为O(n+k)。

计数排序的一个重要性质是它是稳定的：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的相对次序是相同的。也就是说，对两个相同的数来说，在输入数组中先出现的数，在输出数组中也位于前面。

**section2**

1. 题目内容

给定两个序列X[1..n]和Y[1..m], 求解X和Y的最长公共子序列，输出所有可能的公共子序列

输入：

第一行输入m, k，表示输入的数组有m个数，查找第k大的数

第二行输入m个整数

输出:

第k大的数

测试数据:

* 输入（input）：
* ABCD
* BAD
* 输出（output）：
* 2
* BD
* AD

1. 概要设计

题目要求：寻找所有可能的最长公共子序列

算法思想：动态规划

算法概述（具体）：

* 1. 根据两个字符串序列确定字符串长度，初始化二维数组存放行列对应的字符。
  2. 根据在附近2\*2的二维数组中的元素和对应位置字符是否相同求解右下角元素（对应行列元素相同，为{max[2\*2-1]+1}；否则为max[2\*2-1]）。
  3. 通过回溯算法打印相同的字符

1. 详细设计
2. 设计步骤
3. 初始化长度变量：计算两个输入序列X和Y的长度。

2.创建动态规划表格：构建一个二维数组dp，其中dp[i][j]表示 序列X[1…i] 和Y[1…j]的最长公共子序列的长度。

3.初始化动态规划表格：通过比较X和Y的元素来更新dp数组。 如果元素相同， 则当前dp值为左上角的dp值加一；如果不同， 则当前dp值为左边或上边 的dp值中的较大者。

4.寻找最长公共子序列的长度：dp数组的最后一个元素dp[n][m] 表示整个序列X和Y的最长公共子序列的长度。

回溯函数：从dp数组的右下角开始回溯，找出所有可能的最长公 共子序列。

5.获取所有最长公共子序列：使用回溯函数收集所有可能的最长 公共子序列。

6.输出结果：首先输出最长公共子序列的长度，然后输出所有可 能的最长公共子序列。

1. 代码实现（python）

def lcs\_all\_sequences(X, Y):

n = len(X)

m = len(Y)

dp = [[0] \* (m + 1) for \_ in range(n + 1)]

for i in range(1, n + 1):

for j in range(1, m + 1):

if X[i - 1] == Y[j - 1]:

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

else:

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

lcs\_length = dp[n][m]

def backtrack(i, j):

if i == 0 or j == 0:

return set([""])

elif X[i - 1] == Y[j - 1]:

return {sub + X[i - 1] for sub in backtrack(i - 1, j - 1)}

else:

results = set()

if dp[i - 1][j] >= dp[i][j - 1]:

results.update(backtrack(i - 1, j))

if dp[i][j - 1] >= dp[i - 1][j]:

results.update(backtrack(i, j - 1))

return results

all\_lcs = list(backtrack(n, m))

print(lcs\_length)

for seq in sorted(all\_lcs):

print(seq)

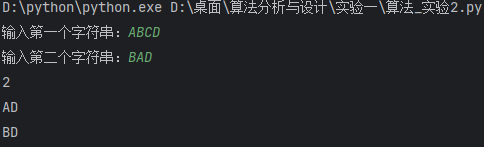
input\_X = input('输入第一个字符串：')

input\_Y = input('输入第二个字符串：')

lcs\_all\_sequences(input\_X, input\_Y)

1. 调试分析

以下为实例调试结果：



1. 心得体会

通过最大公共子序列练习了动态规划，[将复杂问题划分为更简单的子问题，递归地求解并存储子问题的解，从而有效地解决广泛的问题](https://zhuanlan.zhihu.com/p/679646682" \t "https://cn.bing.com/_blank)。其中也训练了回溯算法，通过回溯可以通过同一个“动作”完成函数的多次调用。

**section3**

1. 题目内容

有n项工作，工作j的开始时间是sj，结束时间是fj，完成工作j获得的报酬是wj； 如果两项工作的时间没有重叠，则同一个人可以完成两项工作；目标：在同一个人可以完成的工作中，找出所获报酬最大的工作集合；

输入：

第一行输入工作的数量

第二行开始，每行输入一个工作编号和对应的报酬、开始时间以及结束时间，以空格隔开，时间按XX:YY:ZZ的格式

输出：

第一行输出所获报酬最大的工作编号集合

第二行输出对应的最大报酬

测试数据：

输入：

3

1 10 08:00:00 09:00:00

2 8 08:30:00 9:30:00

3 6 09:10:00 10:10:00

输出：

{1,3}

16

1. 概要设计
2. 步骤设计：

输入处理：读取工作数量和每项工作的详细信息。

排序：按照工作的结束时间对工作进行排序。

选择工作：从排序后的工作列表中选择结束时间最早的工作，然后选择与 已选择工作时间不重叠的下一个工作。

计算报酬：累加所选工作的报酬。

输出结果：输出所选工作的编号集合和总报酬。

1. 伪代码

function selectJobsWithMaxReward(jobs):

按结束时间对工作进行排序

sorted\_jobs = sort(jobs, key=lambda x: x.end\_time)

初始化工作的内容

selected\_jobs = []

max\_reward = 0

current\_end\_time = 0

遍历排序后的工作列表

for job in sorted\_jobs:

if job.start\_time >= current\_end\_time:

# 选择工作

selected\_jobs.append(job.id)

max\_reward += job.reward

current\_end\_time = job.end\_time

输出结果

读取输入

调用函数

输出所获报酬最大的工作编号集合和对应的最大报酬

1. 代码算法思想：采用贪心算法思想
2. 详细设计

**代码实现（python）：**

def parse\_time(time\_str):

h, m, s = map(int, time\_str.split(':'))

return h \* 3600 + m \* 60 + s

def max\_reward\_jobs(jobs):

jobs.sort(key=lambda x: x[3])

n = len(jobs)

dp = [0] \* n

dp[0] = jobs[0][1]

job\_selection = [[] for \_ in range(n)]

job\_selection[0].append(jobs[0][0])

for i in range(1, n):

include\_reward = jobs[i][1]

for j in range(i - 1, -1, -1):

if jobs[j][3] <= jobs[i][2]:

include\_reward += dp[j]

break

if include\_reward > dp[i - 1]:

dp[i] = include\_reward

if j >= 0 and jobs[j][3] <= jobs[i][2]:

job\_selection[i] = job\_selection[j] + [jobs[i][0]]

else:

job\_selection[i] = [jobs[i][0]]

else:

dp[i] = dp[i - 1]

job\_selection[i] = job\_selection[i - 1]

max\_reward = dp[-1]

max\_jobs = set(job\_selection[-1])

return max\_jobs, max\_reward

n = int(input("工作的数量："))

jobs = []

for \_ in range(n):

job\_id, reward, start\_time, end\_time = input().split()

jobs.append((int(job\_id), int(reward), parse\_time(start\_time), parse\_time(end\_time)))

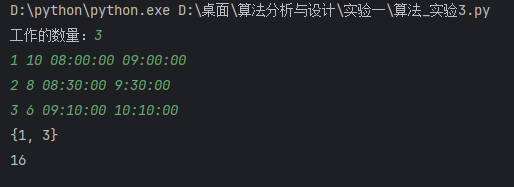
job\_set, total\_reward = max\_reward\_jobs(jobs)

print(f"{job\_set}")

print(total\_reward)

1. 测试程序的运行结果

以下为实例测试：



实验二

1. 题目内容

给定一个无向图，自选一种数据存储结构，实现最小生成树计算的Prim算法和Kruskal算法，分析基于所选数据存储结构得到的算法复杂度。

输入：

第一行输入两个数m, n，表图示有m个顶点（所有顶点的字母各不相同），n条边；

接下来n行每行输入两个顶点，一个值w，表示这两个顶点之间有边连接，且边的权重为w；

输出:

分别输出Prim算法和Kruskal算法得到的最小生成树（若某个节点存在最小生成树的多个选择，则按照字母表顺序进行优先选择，即输出只有一种结果）

测试数据:

* 输入（input）:
* 3,3
* A,B,2
* A,C,5
* B,C,4
* 输出（output）:
* Prim:
* A,B,2
* B,C,4
* Kruskal:
* A,B,2
* B,C,4

1. 概要分析

（1）最小生成树（个人理解）：任意两点可达，并且图中所有边的权重最小

（2）prim算法思想：通过贪心算法的理念，每次在原有连通图中寻找不在树中的结点连接成权重最小的边，直到所有顶点都被连接，即生成最小生成树。

算法设计思想：在初始化的u，v，w中任意找一个初始点放在{u}，{v}中删除这个点；再找与图中点相连的边权值最小的顶点，重复添加和删除的工作；直到{v}空.

（3）kruskal算法：对初始化的图中边上的权重排序，依次选择“最小权重”的边（选择的规则为不能形成环）

1. 详细设计

（1）主要操作的实现。（伪代码）

Prim（G）：

# 输入：加权连通图G=<V,E>

# 输出：Et，组成G的最小生成树的边集合

Vt = {V0}

Et = ∅

for i=1--|V|-1:

所有边中找到最小边（u，v）

把Vi加入到Vt，u加入到V-Vt

return Et

kruskal算法：

kruskal（G）:

Et = 空集合

count = 0 （集合规模）

已处理边 = 0

while count<|V|-1:

k += 1

if 无环路：

Et更新

count += 1

return Et

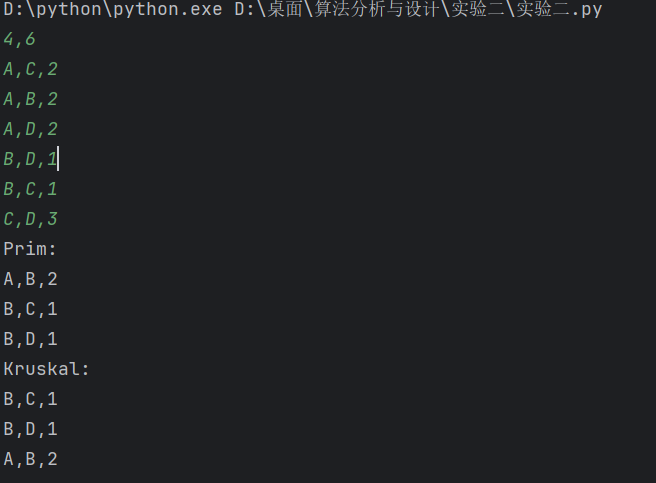
1. **python代码实现**

import heapq  
class Graph:  
 def \_\_init\_\_(self, vertices):  
 self.V = vertices  
 self.graph = [[] for \_ in range(vertices)]  
  
 def add\_edge(self, u, v, w):  
 self.graph[u].append((v, w))  
 self.graph[v].append((u, w))  
  
 def prim(self):  
 key = [float('inf')] \* self.V  
 parent = [-1] \* self.V  
 key[0] = 0  
 min\_heap = [(0, 0)]  
 in\_heap = [True] \* self.V # 添加一个标记数组  
  
 while min\_heap:  
 weight, u = heapq.heappop(min\_heap)# pop最小元素  
 in\_heap[u] = False # 标记为已从堆中移除  
  
 for v, w in self.graph[u]:  
 # 检查是否在堆中并且是否可以更新键值  
 if in\_heap[v] and w < key[v]:  
 key[v] = w  
 parent[v] = u  
 heapq.heappush(min\_heap, (w, v)) # 重新将更新后的顶点加入堆  
  
 return parent  
  
 def kruskal(self):  
 edges = []  
 for u in range(self.V):  
 for v, w in self.graph[u]:  
 if u < v: # 确保每条边只添加一次  
 edges.append((w, u, v))  
 edges.sort()  
  
 parent = [i for i in range(self.V)]  
 rank = [0] \* self.V  
  
 def find(u):  
 if u != parent[u]:  
 parent[u] = find(parent[u])  
 return parent[u]  
  
 def union(u, v):  
 root\_u = find(u)  
 root\_v = find(v)  
 if rank[root\_u] < rank[root\_v]:  
 parent[root\_u] = root\_v  
 elif rank[root\_u] > rank[root\_v]:  
 parent[root\_v] = root\_u  
 else:  
 parent[root\_v] = root\_u  
 rank[root\_u] += 1  
  
 mst = []  
 for w, u, v in edges:  
 if find(u) != find(v):  
 union(u, v)  
 mst.append((u, v, w))  
  
 return mst  
  
m, n = map(int, input().split(','))  
g = Graph(m)  
  
for \_ in range(n):  
 u, v, w = input().split(',')  
 g.add\_edge(ord(u) - ord('A'), ord(v) - ord('A'), int(w))  
  
parent = g.prim()  
print("Prim:")  
for i in range(1, m):  
 print(chr(ord('A') + parent[i]), chr(ord('A') + i), sep=',', end=',')  
 for v, w in g.graph[parent[i]]:  
 if v == i:  
 print(w)  
 break  
  
mst = g.kruskal()  
print("Kruskal:")  
for u, v, w in mst:  
 print(chr(ord('A') + u), chr(ord('A') + v), w, sep=',')

1. 测试程序的运行结果

以下为实例调试：





实验三

1. 题目内容

以邻接表的存储方式，实现图的BFS和DFS遍历，并分析复杂度。（100分）

输入：

第一行输入两个数m, n，表示图有m个顶点（所有顶点的字母各不相同），n条边；接下来n行每行输入两个顶点，表示这两个顶点之间有边相连；最后一行输入遍历开始的顶点

输出:

从遍历开始的顶点出发，分别输出图的BFS和DFS遍历的结果（若某个节点存在多种遍历方式，则按照字母表顺序来进行遍历，即输出只有一种结果）

测试数据:

* 输入（input）:
* 3,3
* A,B
* A,C
* B,C
* A
* 输出（output）:
* A,B,C
* A,B,C

1. 概要分析
2. BFS：广度优先搜索。由根节点向叶子结点遍历，叶子结点在同一级的优先遍历。利用队列先进先出的特点完成BFS。搜索核心是从头结点开始，寻找一步到达的合法可行点，并加入队列，然后弹出头结点，由依次对队列中的结点执行寻找操作，直至队列为空。
3. DFS：深度优先搜索。由根节点向左下完全遍历，直到最后遍历到的结点没有叶子节点，在从根节点的另外一个叶子节点按照上一步的做法向下遍历。

通过邻接矩阵的形式表示两点之间有无边的连接。

DFS伪代码：

A[][]//初始化邻接矩阵

visited={false,false...}//初始化为未访问

DFS(u)://从u点开始DFS访问

visted[u]=true

for i=1 to n:

if !visited[i] and A[u][i]

DFS(i)

1. 详细设计

（1）主要操作的实现。（伪代码）

Prim（G）：

# 输入：加权连通图G=<V,E>

# 输出：Et，组成G的最小生成树的边集合

Vt = {V0}

Et = ∅

for i=1--|V|-1:

所有边中找到最小边（u，v）

把Vi加入到Vt，u加入到V-Vt

return Et

kruskal算法：

kruskal（G）:

Et = 空集合

count = 0 （集合规模）

已处理边 = 0

while count<|V|-1:

k += 1

if 无环路：

Et更新

count += 1

return Et

（2）**python代码实现**  
from collections import defaultdict, deque  
  
class Graph:  
 def \_\_init\_\_(self, vertices):  
 self.graph = defaultdict(list)  
 self.V = vertices  
  
 def add\_edge(self, u, v):  
 self.graph[u].append(v)  
  
 def BFS(self, s):  
 visited = [False] \* self.V  
 queue = deque([s])  
 visited[s] = True  
 bfs\_order = []  
  
 while queue:  
 s = queue.popleft()  
 bfs\_order.append(s)  
  
 for i in sorted(self.graph[s]):  
 if not visited[i]:  
 queue.append(i)  
 visited[i] = True  
 return bfs\_order  
  
 def DFSUtil(self, v, visited, dfs\_order):  
 visited[v] = True  
 dfs\_order.append(v)  
  
 for i in sorted(self.graph[v]):  
 if not visited[i]:  
 self.DFSUtil(i, visited, dfs\_order)  
  
 def DFS(self, v):  
 visited = [False] \* self.V  
 dfs\_order = []  
 self.DFSUtil(v, visited, dfs\_order)  
 return dfs\_order  
  
# 示例输入  
m, n = map(int, input().split(','))  
g = Graph(m)  
edges = []  
  
for \_ in range(n):  
 u, v = input().split(',')  
 edges.append((u, v))  
  
# 添加边  
for u, v in edges:  
 g.add\_edge(ord(u) - ord('A'), ord(v) - ord('A'))  
  
# 遍历开始的顶点  
start\_vertex = ord(input().strip()) - ord('A')  
  
# 输出BFS和DFS遍历结果  
print("BFS:", ','.join([chr(ord('A') + i) for i in g.BFS(start\_vertex)]))  
print("DFS:", ','.join([chr(ord('A') + i) for i in g.DFS(start\_vertex)]))

1. 测试程序的运行结果

以下为实例调试：

