

算法与数据结构设计报告

（ 2024 / 2025学年 第 一 学期）

题 目： A2：TOP-N问题

B2: 游览校园

|  |  |
| --- | --- |
| **专 业** | **计算机科学与技术** |
| **学 生 姓 名** | **施淳博** |
| **班 级 学 号** | **B22040220** |
| **指 导 教 师** | **柯昌博** |
| **指 导 单 位** | **计算机科学与技术系** |
| **日 期** | **2024.10.28-2024.11.8** |

|  |  |  |
| --- | --- | --- |
| **支撑指标点** | **评价准则** | **计分（每项10分）** |
| **课程目标1**：文献调研与资料收集能力，问题发现、研究、分析与解决能力（20分） | 1、能够掌握算法与数据结构设计的相关基础知识，并能够针对求解的工程问题，收集资料进行合理的分析与设计 |  |
| 2、通过调研，能够选择合适的程序设计语言与编程开发平台，对求解的工程问题进行编程实现 |  |
| **课程目标2**：通过课程设计，培养学生综合应用算法和数据结构等知识解决工程问题的实践能力（20分） | 3、能够给出数据结构和算法的设计描述，给出关键算法的流程图或伪代码，并给出各算法之间的结构关系描述 |  |
| 4、具备一定的人机交互设计意识，人机交互设计合理、友好，操作简便 |  |
| **课程目标3**：培养解决工程问题的开发工具运用能力，能够利用程序设计软件或系统对问题求解进行模拟和实现，能够设计测试数据验证问题解决方法的正确性，并能够对问题解决方法的性能和效率进行分析（40分） | 5、具备一定的算法与数据结构设计分析能力，能够完成课题要求的各项任务和指标 |  |
| 6、能够结合计算机软硬件资源，合理选用算法、数据结构、数据存储方式等技术手段，对求解的工程问题进行有效建模和求解 |  |
| 7、具备一定自学能力与探索创新意识，能够充分利用教科书及其资源（如网络等）自学新知识与新技能 |  |
| 8、掌握调试方法与工具，对程序开发过程中出现的问题进行分析、跟踪与调试，并能够进行充分测试 |  |
| **课程目标4**：选择同类课题的学生能够通过讨论和交流解决课程设计中的难题，能在实验报告中准确阐述课程设计的内容，能够清晰陈述观点和回答问题（20分） | 9、能够正确、完整地回答指导教师关于课题的问询，反映其对课题内容，以及相关的工程基础知识具有较好的理解和掌握 |  |
| 10、具备一定的语言表达能力与文字处理能力，能够结合复杂工程问题撰写报告，报告内容和实验数据详实，格式规范 |  |
| 算法与数据结构设计能力测评总分 | |  |
| **指导教师： 年 月 日** | | |
| **备注：** | | |

**TOP-N问题和游览校园**

**一、课题内容和要求**

A TOP-N问题

假设全国有10000人参加物理竞赛，只录取成绩优异的前10名，并将他们从

高分到低分输出。而对落选的其他考生，无需排出名次，问此种情况下，试编写

尽可能快的算法解决Top-10问题？

扩展：如何在10亿数中找出前1000大的数？

B 游览校园

作为一名南京邮电大学的新生，你想游览一遍校园，其中有门（北门，南门，西门，东门）和景点（体育场，计算机学科楼，三食堂，二食堂，一食堂，图书馆，宿舍楼，教学楼，办公楼），你不想重复经过同一处景点，且必须从一个门进从另一个门出（路程中可 以经过门，但不能重复经过同一个门），你已经将游览路径抽象为无向图，请给出可行路径。

如：北门-体育场-计算机学科楼-三食堂-二食堂-图书馆-宿舍楼-一食堂-教学楼-办公楼-南门。

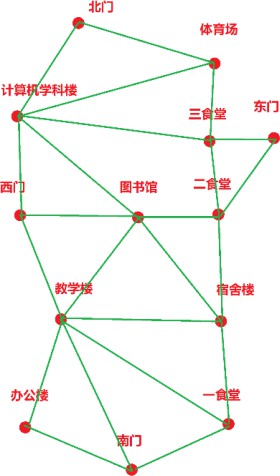


图1 南京邮电大学地图及无向图

基本要求：

（1）使用邻接表或邻接矩阵表示图

（2）设计算法得到至少一种可行方案

扩展要求：

若希望两次游览校园，且两次路径经过的景点和门不能重复，请给出可行方案

如：西门-图书馆-计算机学科楼-三食堂-体育场-北门；东门-二食堂-宿舍楼-一食堂-教学楼-办公楼-南门

**二、数据结构说明**

A TOP-N问题

PriorityQueue<Integer> minHeap = new PriorityQueue<>(); //优先权队列建立最小堆

B 游览校园

graph = { //邻接表

'北门': ['体育场', '计算机学科楼'],

'体育场': ['北门', '计算机学科楼', '三食堂'],

'计算机学科楼': ['北门', '西门', '体育场', '三食堂', '图书馆'],

'南门': ['一食堂', '教学楼', '办公楼'],

'一食堂': ['南门', '宿舍楼', '教学楼'],

'教学楼': ['南门', '西门', '图书馆', '办公楼', '宿舍楼', '一食堂'],

'办公楼': ['南门', '教学楼'],

'西门': ['计算机学科楼', '图书馆', '教学楼'],

'图书馆': ['西门', '计算机学科楼', '二食堂', '宿舍楼', '教学楼'],

'东门': ['二食堂', '三食堂'],

'二食堂': ['东门', '三食堂', '图书馆', '宿舍楼'],

'三食堂': ['东门', '体育场', '计算机学科楼', '二食堂'],

'宿舍楼': ['二食堂', '一食堂', '图书馆', '教学楼'],

}

if visited is None: //已访问节点的集合

visited = set()

if path is None: //当前路径节点列表

path = []

if all\_paths is None: //有效路径节点列表

all\_paths = []

**三、算法设计**

A TOP-N问题

（1）最小堆

使用最小堆来解决TOP-N问题，最小堆是一种数据结构，它是一棵完全二叉树并且它所有父节点的值小于或等于两个子节点的值。

最小堆的存储结构实际上是一个数组。如下图：

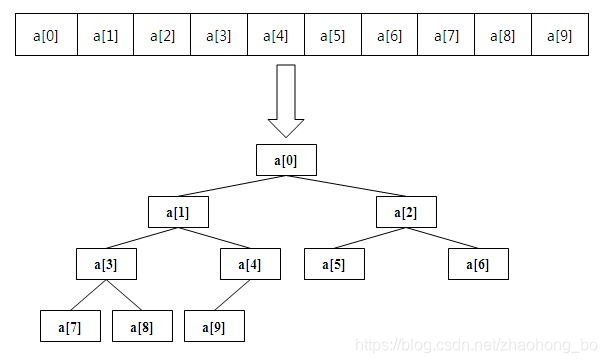


图2 最小堆

在第一问中，前10元素直接添加入最小堆，且最小元素上浮至堆顶，之后的元素如果大于堆顶元素则替换堆顶元素。在解决数据量较大的问题时能提高效率。

主要逻辑：

读取分数：readScoresFromTextFile方法使用BufferedReader从文件中逐行读取分数。每行的分数通过空格分隔，并被解析为整数后添加到ArrayList中。最后，ArrayList被转换为int数组并返回。找出前10个最高分数

findTop10方法使用一个PriorityQueue（最小堆）来存储分数。遍历所有分数，如果堆的大小小于10，直接将分数加入堆中。如果堆的大小已经达到10，并且当前分数大于堆顶元素，那么移除堆顶元素，并将当前分数加入堆中。这样，堆中始终存储着当前找到的最高的10个分数。最后，从堆中取出元素并逆序存储到数组中，因为堆是最小堆，所以需要逆序以保证分数从高到低排列。

（2）分治法

第二问中要在10亿数中选择前1000大的数，由于数据量非常大，所以采用分治法并行计算，提高效率。将10亿数分为100个子集，对每个子集使用最小堆找出前1000大的数，再将每个子集的结果合并再找出前1000。

主要逻辑：

读取数字：main方法使用BufferedReader从文件中逐行读取数字。每行的数字通过空格分隔，并被解析为整数后添加到ArrayList中。找出前1000个最大的数字：

findTop1000方法将数字列表分割成100个子集，每个子集包含大约1000个数字。对每个子集调用findTop1000InSubset方法找出前1000个最大的数字。然后调用mergeTop1000方法合并所有子集的结果。在子集中找出前1000个最大的数字：

findTop1000InSubset方法使用一个PriorityQueue（最小堆）来存储前1000个最大的数字。如果子集大小小于等于1000，直接对子集进行排序并取前1000个数字。如果子集大小大于1000，使用最小堆来找出前1000个最大的数字。

合并子集结果：mergeTop1000方法同样使用一个PriorityQueue来合并所有子集的结果。遍历每个子集的结果，如果堆的大小小于1000，直接将数字加入堆中；如果堆的大小已经达到1000，并且当前数字大于堆顶元素，则替换堆顶元素。

B 游览校园

（1）邻接表

邻接表是图的一种最主要存储结构,用来描述图上的每一个点。使用邻接表表示该无向图便于后面解决问题

（2）回溯法（深度优先搜索）

问题要求经过所有的景点，相当于要找出该图的哈密顿路径。为了提高效率，在当前节点的相邻节点都被访问且该节点不是终点时使用回溯法对图进行深度优先搜索来找出答案。

（3）算法复杂度分析

时间复杂度：

由于算法使用深度优先搜索（DFS）策略，时间复杂度取决于图的大小和深度。在最坏的情况下，算法需要遍历图中的所有节点和边，因此时间复杂度为O(V+E)，其中V是节点数，E是边数。

空间复杂度：

空间复杂度主要由递归栈和存储路径的列表决定。在最坏的情况下，递归栈的深度可以达到图的深度，而路径列表的长度可以达到节点数，因此空间复杂度为O(V)。

主要逻辑：

初始化visited集合、path列表和all\_paths列表。从起始节点开始，递归地探索图中的所有路径。在每一步，检查当前节点是否为结束节点，并且是否已访问所有必需的节点。如果是，则保存当前路径。对于当前节点的每个未访问的邻居节点，递归地调用find\_all\_paths函数。在递归返回之前，从visited集合和path列表中移除当前节点，以便进行下一次递归调用在所有递归调用完成后，all\_paths列表中包含了从起始节点到结束节点的所有路径，这些路径都访问了所有必需的节点。将所有路径打印出来，并写入到文件travel.txt中。

**四、详细设计**

A TOP-N问题

问题1：

package org.example;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.PriorityQueue;

/\*\*

\* 这个类用于找出分数最高的前10个数。

\*/

public class TopTenScores {

public TopTenScores() {

}

/\*\*

\* 程序的主入口点。

\*/

public static void main(String[] args) {

// 从文件中读取分数并存入数组

int[] scores = readScoresFromTextFile("data2.txt");

// 找出分数最高的前10个数

int[] top10Scores = findTop10(scores);

// 打印结果

for (int score : top10Scores) {

System.out.println(score);

}

}

/\*\*

\* 从文本文件中读取分数。

\*/

private static int[] readScoresFromTextFile(String filename) {

List<Integer> scoreList = new ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader(filename))) {

String line;

while ((line = br.readLine()) != null) {

String[] scoreStrings = line.trim().split("\\s+");//数字以空格分隔

for (String scoreString : scoreStrings) {

scoreList.add(Integer.parseInt(scoreString)); // 将字符串转换为整数并添加到列表

}

}

} catch (IOException e) {

e.printStackTrace();

}

// 将列表转换为数组并返回

return scoreList.stream().mapToInt(Integer::intValue).toArray(); // 使用stream将ArrayList转换为int数组

}

/\*\*

\* 找出数组中分数最高的前10个数。

\* @param scores 包含所有分数的整型数组。

\* @return 包含前10个最高分数的整型数组。

\*/

public static int[] findTop10(int[] scores) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

int i = scores.length;

// 使用优先队列（最小堆）来找出前10个最高分数

for (int var4 = 0; var4 < i; ++var4) {

int score = scores[var4];

if (minHeap.size() < 10) {

minHeap.add(score); //堆未满直接加入

} else if (score > minHeap.peek()) {

minHeap.poll(); //堆满如果当前元素大于堆顶元素则替换

minHeap.add(score);

}

}

int[] top10 = new int[10];

// 从优先队列中取出元素并逆序存储到数组中

for (i = 9; i >= 0; --i) {

top10[i] = minHeap.poll();

}

return top10;

}

}

问题2：

package Top1000Numbers;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.\*;

import java.util.stream.Collectors;

public class Top1000Numbers {

public static void main(String[] args) {

// 读取文件并将所有数字存储在列表中

List<Integer> numbers = new ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader("data2.txt"))) {

String line;

while ((line = br.readLine()) != null) {

String[] nums = line.trim().split("\\s+"); // 以空格分隔数字

for (String num : nums) {

numbers.add(Integer.parseInt(num)); // 将字符串转换为整数并添加到列表

}

}

} catch (IOException e) {

e.printStackTrace();

}

// 找出前1000个最大的数字

List<Integer> top1000Numbers = findTop1000(numbers);

// 打印结果

System.out.println(top1000Numbers);

}

// 主要方法：找出前1000个最大的数字

public static List<Integer> findTop1000(List<Integer> numbers) {

int numSubsets = 100; // 子集的数量

int subsetSize = numbers.size() / numSubsets; // 每个子集的大小

List<List<Integer>> subsets = new ArrayList<>(); // 存储所有子集

// 将数据分割成多个子集

for (int i = 0; i < numSubsets; i++) {

int start = i \* subsetSize;

int end = (i + 1) \* subsetSize;

if (i == numSubsets - 1) {

end = numbers.size(); // 最后一个子集包括所有剩余元素

}

subsets.add(new ArrayList<>(numbers.subList(start, end))); // 创建子集并添加到列表

}

List<List<Integer>> top1000InSubsets = new ArrayList<>(); // 存储每个子集的前1000个最大数字

for (List<Integer> subset : subsets) {

top1000InSubsets.add(findTop1000InSubset(subset)); // 找出每个子集的前1000个最大数字

}

return mergeTop1000(top1000InSubsets); // 合并所有子集的结果

}

// 在单个子集中找出前1000个最大的数字

private static List<Integer> findTop1000InSubset(List<Integer> subset) {

if (subset.size() <= 1000) {

return subset.stream().sorted(Collections.reverseOrder()).limit(1000).collect(Collectors.toList()); // 如果子集大小小于等于1000，直接排序

}

PriorityQueue<Integer> minHeap = new PriorityQueue<>(1000); // 创建最小堆，存储前1000个最大数字

for (Integer number : subset) {

if (minHeap.size() < 1000) {

minHeap.offer(number); // 堆中元素少于1000，直接添加

} else if (number > minHeap.peek()) {

minHeap.poll(); // 当前数字大于堆顶元素，替换堆顶元素

minHeap.offer(number);

}

}

return new ArrayList<>(minHeap); // 返回堆中的元素

}

// 合并所有子集的结果，找出前1000个最大的数字

private static List<Integer> mergeTop1000(List<List<Integer>> results) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>(1000); // 创建最小堆，存储前1000个最大数字

for (List<Integer> result : results) {

for (Integer number : result) {

if (minHeap.size() < 1000) {

minHeap.offer(number); // 堆中元素少于1000，直接添加

} else if (number > minHeap.peek()) {

minHeap.poll(); // 当前数字大于堆顶元素，替换堆顶元素

minHeap.offer(number);

}

}

}

return new ArrayList<>(minHeap); // 返回堆中的元素

}

}

B 游览校园

问题1：

def find\_all\_paths(graph, current, end, required\_nodes, visited=None, path=None, all\_paths=None):

"""

在图中找到所有从当前节点到结束节点的路径，同时确保访问了所有必需的节点。

:param graph: 图的邻接表表示

:param current: 当前节点

:param end: 结束节点

:param required\_nodes: 必须访问的节点集合

:param visited: 已访问的节点集合

:param path: 当前路径

:param all\_paths: 所有路径的集合

:return: 包含所有路径的列表

"""

if visited is None:

visited = set()

if path is None:

path = []

if all\_paths is None:

all\_paths = []

# 标记当前节点为已访问，并将其添加到路径中

visited.add(current)

path.append(current)

# 检查是否到达结束节点，并且已访问所有必需的节点

if current == end and required\_nodes.issubset(visited):

all\_paths.append(path[:])

else:

# 探索邻居节点

for neighbor in graph[current]:

if neighbor not in visited: # 只访问未访问过的节点

find\_all\_paths(graph, neighbor, end, required\_nodes, visited, path, all\_paths)

# 回溯：从已访问的节点和路径中移除当前节点

visited.remove(current)

path.pop()

return all\_paths

# 定义图的邻接表

graph = {

'北门': ['体育场', '计算机学科楼'],

'体育场': ['北门', '计算机学科楼', '三食堂'],

'计算机学科楼': ['北门', '西门', '体育场', '三食堂', '图书馆'],

'南门': ['一食堂', '教学楼', '办公楼'],

'一食堂': ['南门', '宿舍楼', '教学楼'],

'教学楼': ['南门', '西门', '图书馆', '办公楼', '宿舍楼', '一食堂'],

'办公楼': ['南门', '教学楼'],

'西门': ['计算机学科楼', '图书馆', '教学楼'],

'图书馆': ['西门', '计算机学科楼', '二食堂', '宿舍楼', '教学楼'],

'东门': ['二食堂', '三食堂'],

'二食堂': ['东门', '三食堂', '图书馆', '宿舍楼'],

'三食堂': ['东门', '体育场', '计算机学科楼', '二食堂'],

'宿舍楼': ['二食堂', '一食堂', '图书馆', '教学楼'],

}

# 定义必须访问的非门节点

required\_nodes = {'体育场', '计算机学科楼', '三食堂', '二食堂', '一食堂', '图书馆', '宿舍楼', '教学楼', '办公楼'}

# 设置起始和结束的门

gates = ['北门', '南门', '西门', '东门']

# 为所有起始和结束门的组合找到所有路径

all\_results = []

for start\_gate in gates:

for end\_gate in gates:

if start\_gate != end\_gate:

paths = find\_all\_paths(graph, start\_gate, end\_gate, required\_nodes)

for path in paths:

all\_results.append((start\_gate, end\_gate, path))

# 打印所有结果

for start\_gate, end\_gate, path in all\_results:

print(f"从{start\_gate}到{end\_gate}的路径：{path}")

with open('travel.txt', 'w', encoding='utf-8') as file:

for start\_gate, end\_gate, path in all\_results:

file.write(f"从{start\_gate}到{end\_gate}的路径：{' -> '.join(path)}\n")

print("所有路径已写入travel.txt")

input("程序已运行完成，按Enter键退出...")

问题2：

from itertools import permutations

def find\_disjoint\_paths(graph, start1, end1, start2, end2, required\_nodes):

"""

在图中找到两条不相交的路径，每条路径都从指定的起点到终点，并且两条路径一起覆盖所有必需的节点。

:param graph: 图的邻接表表示

:param start1: 第一条路径的起点

:param end1: 第一条路径的终点

:param start2: 第二条路径的起点

:param end2: 第二条路径的终点

:param required\_nodes: 必须访问的节点集合

:return: 生成所有不相交的路径对

"""

def find\_all\_paths(graph, start, end, visited=None, path=None):

"""

在图中找到从起点到终点的所有路径，避免重复访问节点。

:param graph: 图的邻接表表示

:param start: 起点

:param end: 终点

:param visited: 已访问的节点集合

:param path: 当前路径

:yield: 生成所有路径

"""

if visited is None:

visited = set()

if path is None:

path = []

visited.add(start)

path.append(start)

if start == end:

yield path[:] # 如果到达终点，生成当前路径

else:

for neighbor in graph.get(start, []): # 遍历邻居节点

if neighbor not in visited:

yield from find\_all\_paths(graph, neighbor, end, visited, path) # 递归查找路径

visited.remove(start) # 回溯，移除当前节点

path.pop()

# 为第一条路线找到所有可能的路径

all\_first\_paths = list(find\_all\_paths(graph, start1, end1))

for first\_path in all\_first\_paths:

# 计算剩余需要访问的节点，并构建排除第一条路径节点的子图

first\_path\_set = set(first\_path)

remaining\_required\_nodes = required\_nodes - first\_path\_set

remaining\_graph = {node: [neighbor for neighbor in neighbors if neighbor not in first\_path\_set]

for node, neighbors in graph.items() if node not in first\_path\_set}

# 在剩余图中为第二条路线找到所有可能的路径

all\_second\_paths = list(find\_all\_paths(remaining\_graph, start2, end2))

for second\_path in all\_second\_paths:

# 检查第二条路径是否覆盖了所有剩余需要访问的节点

if remaining\_required\_nodes.issubset(second\_path):

yield first\_path, second\_path

# 定义图的邻接表

graph = {

'北门': ['体育场', '计算机学科楼'],

'体育场': ['北门', '计算机学科楼', '三食堂'],

'计算机学科楼': ['北门', '西门', '体育场', '三食堂', '图书馆'],

'南门': ['一食堂', '教学楼', '办公楼'],

'一食堂': ['南门', '宿舍楼', '教学楼'],

'教学楼': ['南门', '西门', '图书馆', '办公楼', '宿舍楼', '一食堂'],

'办公楼': ['南门', '教学楼'],

'西门': ['计算机学科楼', '图书馆', '教学楼'],

'图书馆': ['西门', '计算机学科楼', '二食堂', '宿舍楼', '教学楼'],

'东门': ['二食堂', '三食堂'],

'二食堂': ['东门', '三食堂', '图书馆', '宿舍楼'],

'三食堂': ['东门', '体育场', '计算机学科楼', '二食堂'],

'宿舍楼': ['二食堂', '一食堂', '图书馆', '教学楼'],

}

# 定义必须访问的非门节点

required\_nodes = {'体育场', '计算机学科楼', '三食堂', '二食堂', '一食堂', '图书馆', '宿舍楼', '教学楼', '办公楼'}

# 定义所有门

gates = ['北门', '南门', '西门', '东门']

# 存储所有有效的路径组合

all\_solutions = []

# 遍历所有门的排列组合

for (start1, end1, start2, end2) in permutations(gates, 4):

# 为每个门组合找到不相交的路径

for first\_path, second\_path in find\_disjoint\_paths(graph, start1, end1, start2, end2, required\_nodes):

all\_solutions.append({

'First Path': first\_path,

'Second Path': second\_path,

'Gate Combination': f"{start1}-{end1} 和 {start2}-{end2}"

})

# 输出所有解决方案

for solution in all\_solutions:

print("门的组合:", solution['Gate Combination'])

print("第一条路径:", solution['First Path'])

print("第二条路径:", solution['Second Path'])

print("-" \* 50)

# 定义文件路径

file\_path = "travel2.txt"

# 将结果保存到文件

with open(file\_path, "w", encoding="utf-8") as file:

for solution in all\_solutions:

file.write(f"门的组合: {solution['Gate Combination']}\n")

for node in solution['First Path']:

file.write(f"-> {node}")

file.write("\n")

for node in solution['Second Path']:

file.write(f"-> {node}")

file.write("\n\n")

file.write("-" \* 50 + "\n")

print(f"结果已保存到 {file\_path}")

input("程序已运行完成，按Enter键退出...")

**五、测试数据及其结果分析**

A TOP-N问题

问题1：

用随机数生成器生成了随机的一万个数，经过运算得到结果

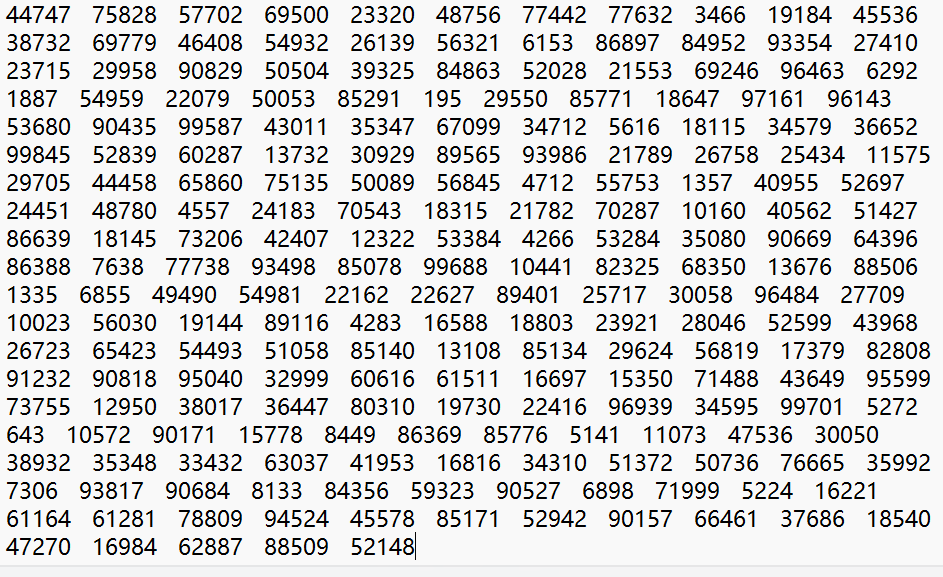


图5 TOP10输入

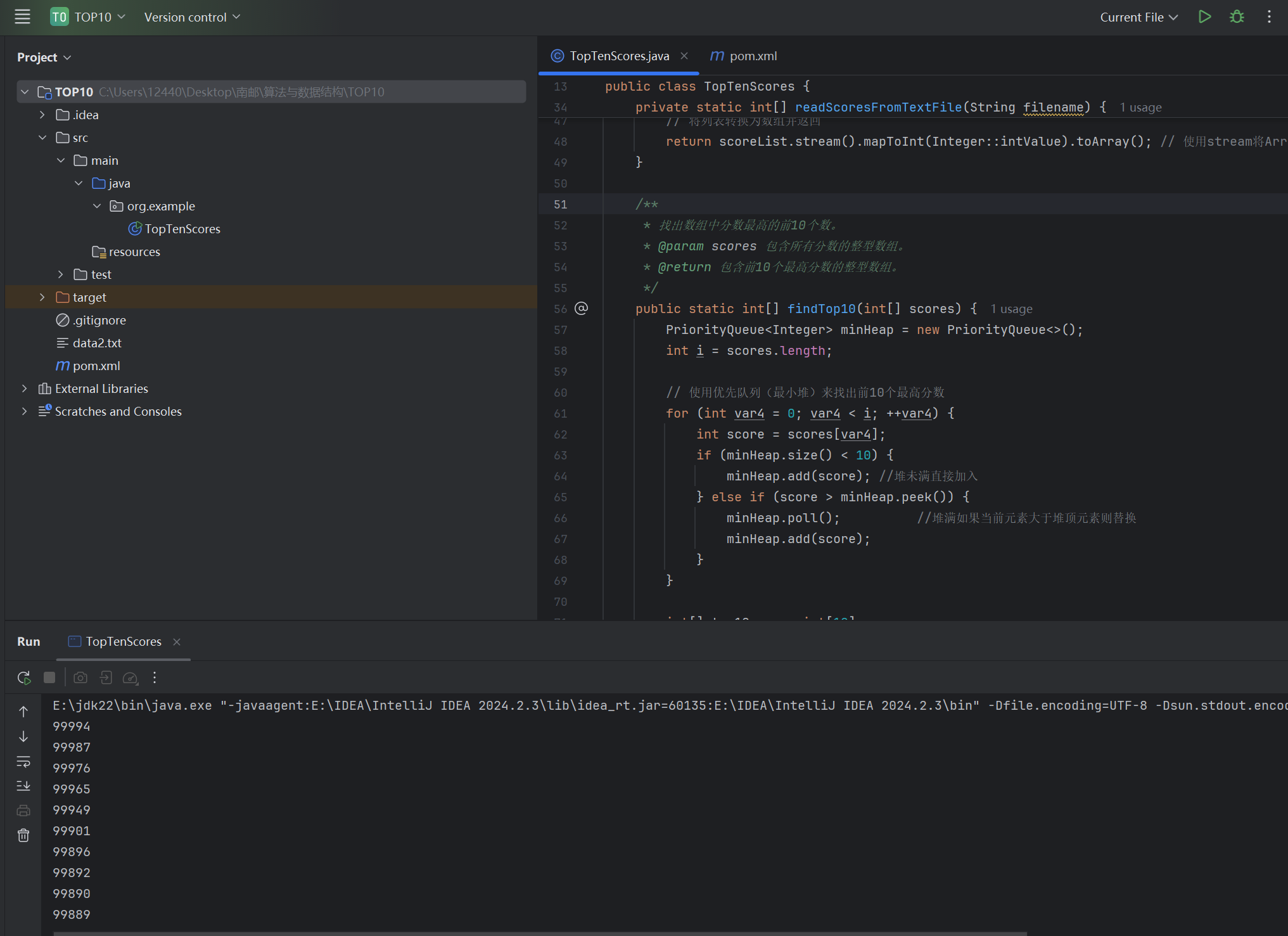


图4 TOP10结果

问题2：

步骤同1

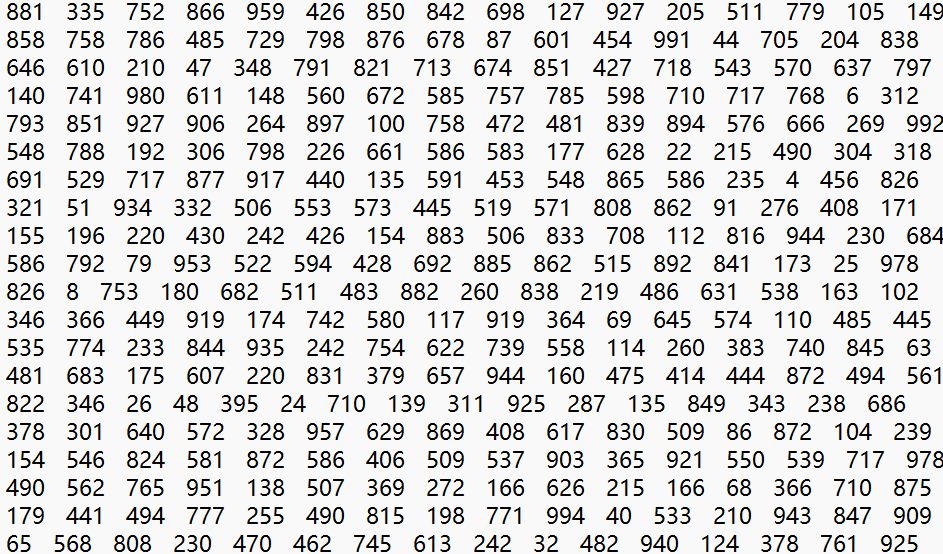


图5 TOP1000输入

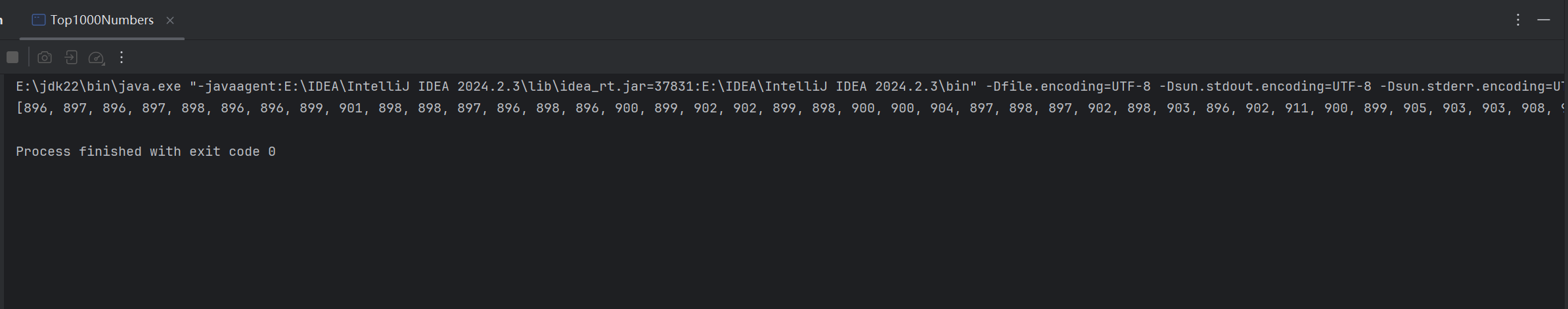


图6 TOP1000结果

B 游览校园

问题1：



图7 游览一次路径

问题2：

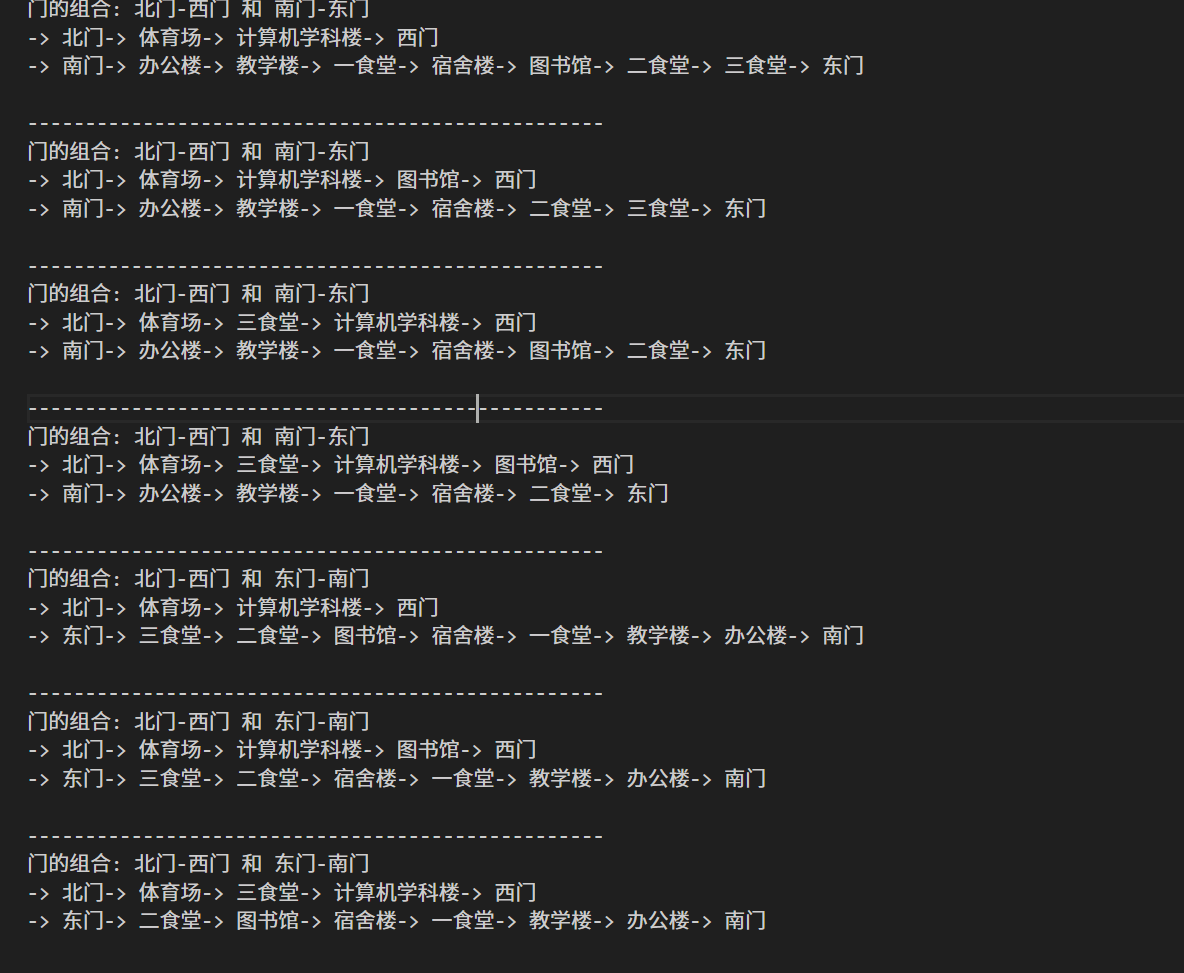


图8 游览两次的路径

**六、算法设计和程序调试过程中的问题**

算法设计和程序调试时遇到哪些问题？你是如何解决的？（至少写3点）例如：

**问题1**：在TOP-N问题中，第一想法是想用最小堆来解决，所以使用了python编写程序，但在java中没有直接的最小堆。

**解决方法**：选择使用了优先权队列来实现最小堆的功能

**问题2**：在游览校园问题1中，因为门节点可以经过也可以不经过，所以如果将问题当作欧拉图问题来解决要考虑很多要素，不利于处理。

**解决方法**：将问题看作寻找哈密顿路径，直接使用回溯法解决，将除门以外的其余节点定义为必须经过的节点，进行回溯和剪枝。

**问题3：**在游览校园问题2中，一开始对两次游览的问题找不到办法解决，如果是要把第一问中的路径组合好保证覆盖全部节点会使问题很复杂

**解决方法：**进行两次寻找路径函数，不经过门节点找出路径后，在第二次寻找时把已经访问的节点去除，重新寻找路径会减少计算量

**七、课程设计总结**

通过这次算法与数据结构设计课程设计，我深刻体会到了理论与实践相结合的重要性。在课程设计的过程中，我不仅复习和巩固了课堂上学到的各种算法和数据结构知识，还有机会亲手实现它们，解决实际问题，这让我对这些抽象概念有了更加深刻的理解。

收获：

1.理论与实践的结合：我学会了如何将理论知识应用到具体的编程问题中，这种实践经验是课堂上无法获得的。

2.问题解决能力：在设计过程中，我遇到了许多挑战，比如算法选择、效率优化等，解决这些问题的过程锻炼了我的逻辑思维和问题解决能力。

3.编程技能的提升：在实现算法和数据结构的过程中，我的编程技能得到了显著提升，特别是在代码优化和调试方面。

4.对算法性能的理解：通过实际测试和比较不同算法的性能，我对算法的时间复杂度和空间复杂度有了更加直观的认识。

感受：

1.挑战与乐趣并存：设计过程中充满了挑战，但每当解决一个难题，实现一个新功能，我都感到无比的兴奋和满足。

2.耐心与细致：算法实现和调试需要极大的耐心和细致，这让我学会了在面对复杂问题时保持冷静，逐步分析和解决问题。

3.自我驱动的学习：在课程设计中，我发现自己需要学习课堂之外的知识，这激发了我自主学习的热情。

4.成就感：当最终的设计成果呈现出来时，我感到非常自豪。这是对我学习成果的最好证明，也是我努力工作的回报。

5.对未来的期待：这次课程设计让我对未来的学习和职业发展充满了期待，我相信这些经验将为我未来的工作打下坚实的基础。

总结来说，算法与数据结构设计课程设计是一次宝贵的学习经历，它不仅提升了我的专业技能，也让我学会了如何面对挑战和解决问题。我将这些收获视为我职业生涯中的重要资产。