

算法分析与设计  
课程设计

学 院 重庆理工大学

专 业 两江人工智能学院

班 级 121230202

学生姓名 宋东升

学号 12123020201

时间 2023年12月

|  |  |  |  |
| --- | --- | --- | --- |
| **检查项目** | **检查方式** | **比例** | **得分** |
| 能够建立时间复杂度评估模型进行算法的时间复杂度分析 | 文档内容检查 | 15% |  |
| 能够基于恰当的算法设计思想实现程序、解决问题 | 程序运行检查 | 40% |  |
| 能清晰阐述表述算法设计思想 | 学生表述情况评估 | 30% |  |
| 能采用文字表述和伪代码形式清晰准确的表达算法思想 | 文档内容检查 | 15% |  |
| 教师签名 |  | 总分 |  |

**目 录**

[1 题目一：一句话描述题目 1](#_Toc1186492439)

[1.1 题目描述 1](#_Toc922346101)

[1.2 算法描述 1](#_Toc570706221)

[1.3 程序运行及其结果 1](#_Toc809105881)

[1.4 时间复杂度分析 1](#_Toc165744788)

[2 题目二：一句话描述题目 2](#_Toc430865695)

[2.1 题目描述 2](#_Toc1396933858)

[2.2 算法描述 2](#_Toc1048067651)

[2.3 程序运行及其结果 2](#_Toc1961616640)

[2.4 时间复杂度分析 2](#_Toc637235290)

[3 小结 3](#_Toc1322587116)

[4 源代码（纸质文档不包含源代码） 4](#_Toc1873373245)

[4.1 题目一：一句话描述题目 4](#_Toc1249186250)

[4.1.1 题目描述 4](#_Toc89520327)

[4.1.2 源代码 4](#_Toc89949258)

[4.2 题目二：一句话描述题目 4](#_Toc1591975966)

[4.2.1 题目描述 4](#_Toc884678102)

[4.2.2 源代码 4](#_Toc338050691)

# 题目一：最长回文子序列

## 题目描述

一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。

示例 ：

输入：s = "bbbab"

输出：4

解释：一个可能的最长回文子序列为 "bbbb" 。

**1.2算法描述**

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

用动态规划方法，使用二维数组dp存储中间结果。

矩阵dp[i][j]表示字符串s中从索引i到索引j（包括两者）的最长回文子序列的长度。算法从初始化矩阵dp的对角线元素开始，因为单个字符本身总是一个回文。然后，以自底向上的方式遍历字符串，考虑所有可能的子串，从较短到较长。对于由索引i和j表示的每个子串，算法检查这些位置的字符是否相等（c1 == c2）。如果相等，通过将长度增加2到不包括当前字符的子串的长度来更新回文子序列的长度（dp[i + 1][j - 1] + 2）。如果字符不相等，则长度由排除第一个字符或最后一个字符后得到的子串的回文子序列长度的最大值确定。

最终结果存储在dp[0][n - 1]中，其中n是输入字符串的长度。这表示整个字符串中最长回文子序列的长度

| - 1, i == j

dp[i][j] = | dp[i + 1][j - 1] + 2, s[i] == s[j]

| - max(dp[i + 1][j], dp[i][j - 1]), otherwise

1. 伪代码（以及必要的注释）

function longestPalindromeSubseq(s):

Input:输入一个字符串s

Output:输出最长回文子序列的长度

n = s.length()

int[][] dp = new int[n][n]

for i from n - 1 to 0:

dp[i][i] = 1

c1 = character at index i in s //得到第i位的字符

for j from i + 1 to n - 1:

c2 = character at index j in s //得到第j位的字符

if c1 is equal to c2://如果相等，长度为第i+1位到第j-1位之间的最长递增子序列长度+2

dp[i][j] = dp[i + 1][j - 1] + 2

else://如果不相等，长度为第i+1位到第j位之间的最长递增子序列长度与第i+1位到第j-1位之间的最长递增子序列长度之间的最大值

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

return dp[0][n - 1]

## 1.3程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）

输入的字符串为abbbbab

输出的最长回文子序列长度 6

1. 运行结果截图



## 1.4时间复杂度分析

总的时间复杂度为 O(n^2)。

# 题目二：最长递增子序列

## 题目描述

给一个整数数组 nums ，找到其中最长严格递增子序列的长度。

示例 1：

输入：nums = [10,9,2,5,3,7,101,18]

输出：4

解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树

创建一个与输入数组相同长度的数组 lIs，用于存储以每个元素为结尾的最长递增子序列的长度。将 lIs 数组初始化为1，因为每个单独的元素都构成长度为1的递增子序列。从数组末尾开始，对于每个元素 nums[i]，遍历其后的元素 nums[j]。 如果 nums[i] 小于 nums[j]，说明可以将 nums[j] 加入以 nums[i] 为结尾的递增子序列，更新 lIs[i] 为当前最大长度。

在更新 lIs[i] 的过程中，保持全局最大长度 max 的值。

最终返回 max，即最长递增子序列的长度

lIs[i]=max(lIs[i],lIs[j]+1) ,j为大于i,小于n的数

1. 伪代码（以及必要的注释）

function lengthOfLIS(nums):

input：一个整数数组

Output:输出最长递增子序列的长度

n = length(nums)

lIs = array of size n, initialized with 1//用于存储以每个元素为结尾的最长递增子序列的长度，初始化为1

max = 1//全局最大长度，初始化为1

for i from 0 to n-1:

for j from i+1 to n-1:

if nums[i] < nums[j]://如果 nums[i] 小于 nums[j]，说明可以将 nums[j] 加入以 nums[i] 为结尾的递增子序列

lIs[i] = max(lIs[i], lIs[j] + 1)

max = max(max, lIs[i])

return max

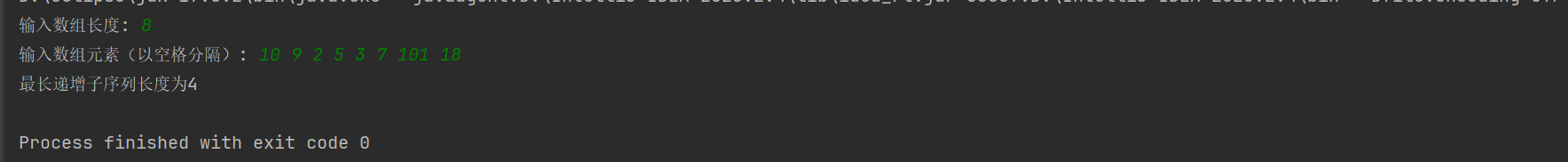
## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）

输入数组为：[10,9,2,5,3,7,101,18]

对应输出应该为4

1. 运行结果截图



## 时间复杂度分析

外层循环：O(n)

内层循环：O(n)（最坏情况下）

每个内层循环中的操作：O(1)

因此，总体时间复杂度为 O(n^2)。

# 题目三：最长公共子序列

## 题目描述

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

示例 1：

输入：text1 = "abcde", text2 = "ace"

输出：3

解释：最长公共子序列是 "ace" ，它的长度为 3 。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

定义两个字符串text1和text2，分别长度为m和n。

创建一个二维数组dp，其中dp[i][j]表示text1的前i个字符和text2的前j个字符的最长公共子序列的长度。

初始化动态规划数组，使得dp[0][j]和dp[i][0]（i和j为任意值）都为0，因为空字符串和任何字符串的最长公共子序列长度都为0。

使用双重循环遍历text1和text2的所有可能组合，填充动态规划数组。对于每一对字符text1.charAt(i - 1)和text2.charAt(j - 1)，如果它们相等，说明找到了新的匹配字符，可以在之前的最长公共子序列长度上加1，即dp[i][j] = dp[i - 1][j - 1] + 1；如果它们不相等，说明当前字符不匹配，那么取之前计算的最长公共子序列长度的较大值，即dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1])。

最终返回dp[m][n]，即text1和text2的最长公共子序列的长度

|-- 0 ,i=0/j=0

dp[i][j]= | dp[i-1][j-1]+1 , text1.charAt(i)=text1.charAt(j)

|-- max(dp[i][j-1],dp[i-1][j]), text1,text2对应的字符不相等

1. 伪代码（以及必要的注释）

function longestCommonSubsequence(text1, text2):

Input:输入两个字符串 text1,text2

Output:输出最长公共子串长度

m = length of text1

n = length of text2

// 初始化动态规划数组

int[][] dp =new int [m + 1] [n + 1]

// 填充动态规划数组

for i from 1 to m:

for j from 1 to n:

if text1[i - 1] == text2[j - 1]://如果两个字符相等

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

else:

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

// 返回最终结果

return dp[m][n]

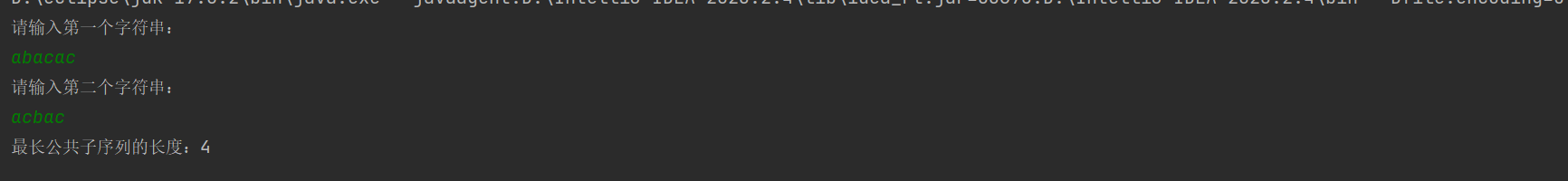
## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）

输入两个字符串分别为abacac acbac

输出长度为4

1. 运行结果截图



## 时间复杂度分析

对于每个数组元素 dp[i][j]，我们都进行了常数时间的计算。在填充整个数组的过程中，我们需要遍历两个字符串的所有可能组合，因此嵌套了两个循环。每个循环的迭代次数是字符串的长度（m 和 n）。

因此，总的时间复杂度为 O(m \* n)。

# 题目四：编辑距离

## 题目描述

给出两个单词 word1 和 word2， 返回将 word1 转换成 word2 所使用的最少操作数  。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

定义两个字符串 word1 和 word2，分别长度为 m 和 n。创建一个二维数组 dp，其中 dp[i][j] 表示将 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最小编辑距离。初始化第一行和第一列，即将一个空字符串转换为另一个字符串的编辑距离。dp[i][0] 表示将 word1 的前 i 个字符转换为空字符串的最小编辑距离，等于 i；dp[0][j] 同理。使用双重循环遍历 word1 和 word2 的所有可能组合，填充动态规划数组。对于每一对字符 word1.charAt(i - 1) 和 word2.charAt(j - 1)，如果它们相等，说明不需要进行编辑操作，因此 dp[i][j] = dp[i - 1][j - 1]；如果它们不相等，选择插入、删除或替换操作中最小的一种，即 dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])。

最终返回 dp[m][n]，即将整个 word1 转换为 word2 所需的最小编辑距离。

|- dp[i - 1][j - 1] ,两个单词最后一个字母相同

dp[i][j]=|

|- 1+min(dp[i][j−1],dp[i−1][j],dp[i−1][j−1]),两个单词最后一个字母不相同

1. 伪代码（以及必要的注释）

function minDistance(word1, word2):

Input:输入两个字符串word1,word2

Output:输出word1 转换成 word2 所使用的最少操作数  。

m = length of word1

n = length of word2

// 初始化动态规划数组

int[][] dp =new int [m + 1] [n + 1]

// 初始化第一行和第一列

for i from 0 to m:

dp[i][0] = i

for j from 0 to n:

dp[0][j] = j

// 填充动态规划数组

for i from 1 to m:

for j from 1 to n:

if word1[i - 1] == word2[j - 1]:

// 字符相等，无需编辑操作

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

else:

// 字符不相等，选择插入、删除或替换操作中最小的一种

dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])

// 返回最终结果，即整个word1转换为word2的最小编辑距离

return dp[m][n]

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）

输入两个单词 kitten ， sitting

输出应该为 4

1. 运行结果截图



## 时间复杂度分析

填充动态规划数组：双重循环，每个循环的迭代次数为字符串的长度（m 和 n）。在每个数组元素的计算中，涉及常数时间的比较和计算操作。因此，填充的时间复杂度为 O(m \* n)。

# 题目五：并查集实现Kruskal算法

## 题目描述

利用并查集实现Kruskal算法生成最小生成树

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

UnionFind 类：实现了并查集数据结构，用于在 Kruskal 算法中维护各个节点的集合信息。\_\_init\_\_（）方法初始化并查集，每个节点的父节点初始为节点本身，秩（rank）初始为 0。Find() 方法用于查找节点所属的集合的根节点，并在查找路径上进行路径压缩，优化后续查找操作。

Union() 方法用于合并两个集合，通过秩进行优化。

kruskal 函数：

实现 Kruskal 算法，对图的边按权重进行排序。

遍历排序后的边，按权重从小到大加入最小生成树，并使用并查集判断是否形成环。

1. 伪代码（以及必要的注释）

class UnionFind:

function \_\_init\_\_(vertices):

// 初始化并查集，每个节点的父节点初始为节点本身，秩（rank）初始为 0

for each vertex in vertices:

parent[vertex] = vertex

rank[vertex] = 0

function find(vertex):

// 查找节点所属的集合的根节点，路径压缩

if parent[vertex] != vertex:

parent[vertex] = find(parent[vertex])

return parent[vertex]

function union(root1, root2):

// 合并两个集合，按秩进行优化

if rank[root1] > rank[root2]:

parent[root2] = root1

else if rank[root1] < rank[root2]:

parent[root1] = root2

else:

parent[root2] = root1

rank[root1] += 1

function kruskal(graph):

// 实现 Kruskal 算法，对图的边按权重进行排序

edges = list of edges in graph sorted by weight

minimum\_spanning\_tree = empty graph

union\_find = new UnionFind(graph.nodes)

for each edge (v1, v2, weight) in edges:

root1 = union\_find.find(v1)

root2 = union\_find.find(v2)

if root1 != root2:

add edge (v1, v2) with weight to minimum\_spanning\_tree

union\_find.union(root1, root2)

return minimum\_spanning\_tree

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）

输入各边情况为('A', 'B', 4), ('A', 'C', 2), ('B', 'C', 5), ('B', 'D', 10), ('C', 'D', 1), ('C', 'E', 7), ('D', 'E', 3)

输出为Kruskal算法生成的最小生成树:

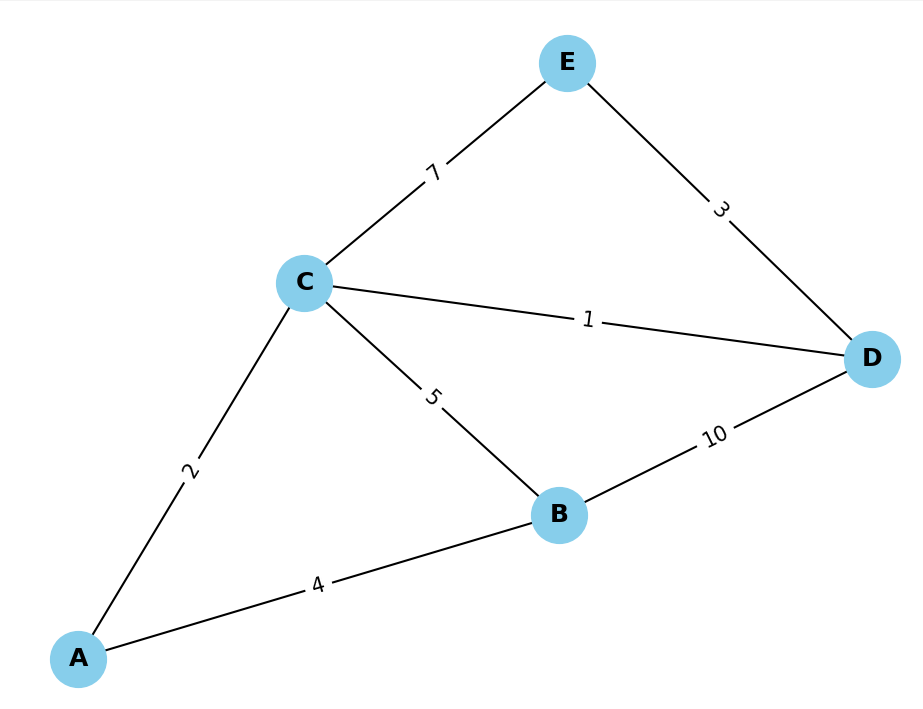
('C', 'D', {'weight': 1})

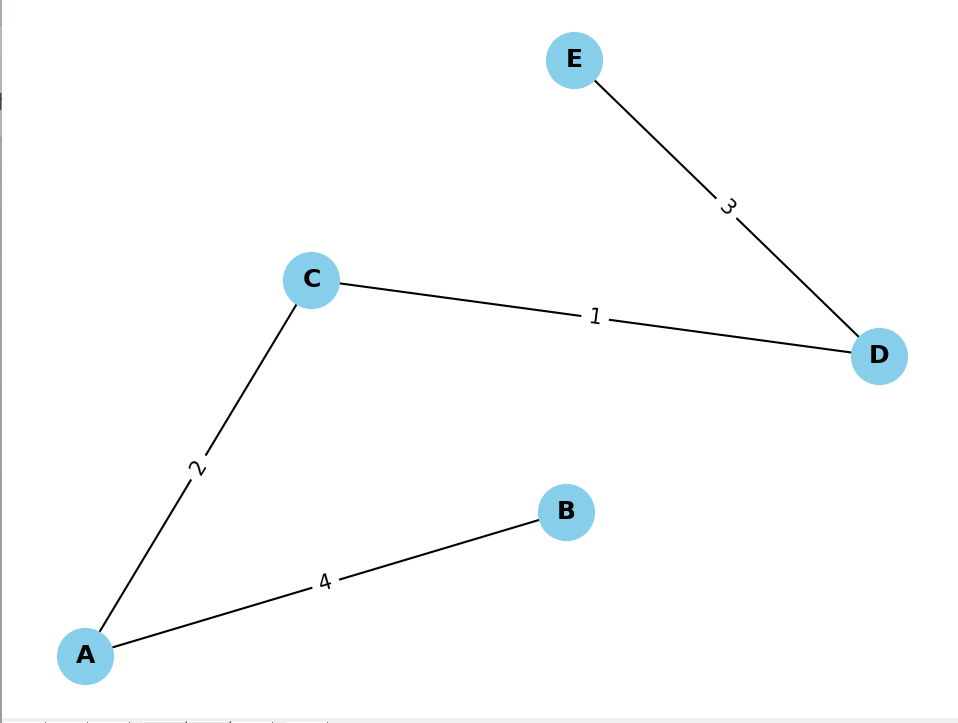
('C', 'A', {'weight': 2})

('D', 'E', {'weight': 3})

('A', 'B', {'weight': 4})

1. 运行结果截图





## 时间复杂度分析

排序边的时间复杂度：

将图的所有边按照权重进行排序。排序的时间复杂度通常为 O(E log E)，其中 E 是边的数量。在这里，E 表示图的边数。

遍历边构建最小生成树的时间复杂度：

遍历排序后的边，对于每条边执行 Union-Find 操作。对于包含 N 个节点的并查集，Union 和 Find 操作的时间复杂度通常为 O(log N)。

对于排序后的 E 条边，总体的时间复杂度为 O(E log N)。

因此，总体的时间复杂度为 O(E log E + E log N)。在这里，E 表示边数，N 表示节点数

# 题目六：验证从湖北出发最多只需要跨越两个省就可以到达中国任何一个省。

## 题目描述

现要验证以下假设：从湖北省出发，经过若干次道路转换，最多只需要跨越两个省就可以到达中国的任何一个省份。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

验证从湖北省出发最多只需要跨越两个省就可以到达中国任何一个省，可以考虑使用图论中的遍历算法，广度优先搜索（BFS），来遍历道路连接关系。

构建图： 将直接的道路连接关系表示为一个有向图或无向图，其中省份为图的节点，道路连接为图的边。

遍历验证： 从湖北省出发，通过深度优先搜索或广度优先搜索遍历图，尽可能多地覆盖其他省份。在遍历的过程中，记录每个省份的到达顺序。

验证结果： 分析遍历的结果，检查是否存在一种方案，使得从湖北省出发，最多只需要跨越两个省就能到达任何一个省份。

1. 伪代码（以及必要的注释）

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）
2. 运行结果截图

## 时间复杂度分析

# 题目七：比较三分查找算法和二份查找算法平均情形时间复杂度。

## 题目描述

设计实验比较三分查找算法和二份查找算法平均情形时间复杂度

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

实现算法：

实现二分查找算法和三分查找算法。确保它们能够正确地在有序数组中查找给定的元素。

生成测试数据：

创建具有不同规模（大小）的有序数组。可以考虑使用随机生成的数据，确保数组已按升序排序。

测量运行时间：

使用合适的工具或编程语言，在各自的查找算法中嵌入时间测量代码。

对于每个算法，运行多次并计算平均运行时间，以获得更准确的结果。确保在每次运行之前都重新生成有序数组，以避免缓存效应。

分析结果：

将不同规模的测试数据应用于两种查找算法，并记录它们的平均运行时间。

1. 伪代码（以及必要的注释）

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）
2. 运行结果截图

## 时间复杂度分析

# 题目八：用蛮力算法和分治算法找到平面的最接近点对

## 题目描述

随机产生平面若干点，利用蛮力算法和分治算法找到平面的最接近点对，并考查随 n 变大时，两者的效率差异、实验效率和理论效率的一致性。平面点集能直观的进行观察。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）
2. 伪代码（以及必要的注释）

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）
2. 运行结果截图

## 时间复杂度分析

# 题目九：顶点个数，出现有向边的概率与出现环的关系

## 题目描述

随机产生个顶点的有向图，并预设两点之间出现有向边的概率，然后探索前两个参数和图中出现环的关系。需要有图形展示。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）
2. 伪代码（以及必要的注释）

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）
2. 运行结果截图

## 时间复杂度分析

# 题目十：电话号码数字组合

## 题目描述

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 任意顺序 返回

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

建数字到字母的映射关系，用一个 Map 存储每个数字对应的字母集合。

定义一个递归的回溯函数，该函数接受当前的数字串、数字到字母的映射、当前正在构建的字母组合、以及当前的索引作为参数。在回溯函数中，首先判断是否已经构建完成了一个字母组合，如果是，则将其加入结果列表。否则，取出当前数字对应的字母集合，对每个字母进行递归调用，构建下一个数字对应的字母组合。不断递归，直到构建完成所有的可能组合。

这个思路通过递归实现了对所有可能组合的穷举，最终得到了答案。在代码中，通过回溯函数 backtrack 不断尝试不同的选择，构建字母组合，直到遍历完所有数字。

1. 伪代码（以及必要的注释）

# 定义生成数字串的所有字母组合的函数

function letterCombinations(digits, phoneMap):

Input:输入一个数字字符串digits

Output:输出可能的所有组合

# 初始化结果列表

combinations = new List<String>()

# 如果数字串为空，则直接返回空列表

if length of digits is 0:

return combinations

# 调用回溯方法生成字母组合

backtrack(combinations, phoneMap, digits, 0, new StringBuffer())

return combinations

# 定义回溯方法

function backtrack(combinations, phoneMap, digits, index, combination):

# 如果当前索引达到数字串的长度，说明一个组合生成完成，将其加入结果列表

if index equals length of digits:

add combination.toString() to combinations

else:

# 取出当前数字对应的字母集合

digit = character at index in digits

letters = phoneMap.get(digit)

lettersCount = length of letters

# 遍历当前数字对应的字母集合，递归调用回溯方法

for i from 0 to lettersCount - 1:

Combination+=letters.charAt(i)

backtrack(combinations, phoneMap, digits, index + 1, combination)

delete character at index from combination//删除刚才所选的那个字母

## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）



输入字符23

对应输出ad, ae, af, bd, be, bf, cd, ce, cf

1. 运行结果截图



## 时间复杂度分析

letterCombinations 函数的时间复杂度：

由于是回溯算法，需要遍历所有可能的组合。

最坏情况下，每个数字对应 4 个字母（'7' 和 '9'）。

所以最坏情况下，生成的组合数量为 4^n，其中 n 是输入数字串的长度。

时间复杂度：O(4^n)

backtrack 函数的时间复杂度：

由于每个数字对应的字母数量不同，实际生成组合的时间取决于具体的输入数字串。

但在最坏情况下，需要遍历 4 个字母。

因此，在最坏情况下，回溯方法的时间复杂度为 O(4^n)。

总体而言，整个程序的时间复杂度主要由生成的组合数量决定，因此是 O(4^n)

# 题目十一：生成括号

## 题目描述

正整数 n 代表生成括号的对数，请设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

## 算法描述

1. 描述思路（含必要的文字说明、递归方程或者状态空间树）

创建一个空的列表 result，用于存储最终的括号组合。

调用递归函数 backtrack，初始时传入空字符串 ""、左括号数量 0、右括号数量 0 和目标括号对数 n。

在递归函数内部：

如果当前字符串长度达到了目标括号对数的两倍，将当前字符串添加到结果列表中，并返回。

如果左括号数量小于目标括号对数，可以尝试添加左括号，递归调用 backtrack。

如果右括号数量小于左括号数量，可以尝试添加右括号，递归调用 backtrack。

最终，返回结果列表。

1. 伪代码（以及必要的注释）

function backtrack(current, open, close, max, result):

if length of current is 2 \* max:

add current to result

return

if open < max://左括号的数量小于括号对数

// try adding an open parenthesis

backtrack(current + "(", open + 1, close, max, result)

if close < open://左括号的数量大于右括号数量

// try adding a close parenthesis

backtrack(current + ")", open, close + 1, max, result)

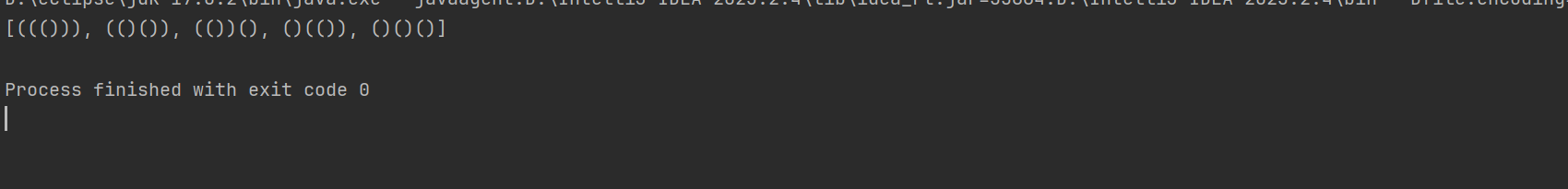
## 程序运行及其结果

1. 给出程序接受的输入和对应的输出（即使截图包含输入和输出，也需要用文字单独说明）

输入括号对数为3

输出为((())), (()()), (())(), ()(()), ()()()

1. 运行结果截图



## 时间复杂度分析

假设 n 表示括号对数，最终结果中的有效组合总数是卡特兰数，即 O(4^n / n^(3/2))。这是因为对于每个有效组合，我们都有 2n 个位置可以放置左括号或右括号，其中选择 n 个位置放置左括号，其余自然是右括号。

因此，时间复杂度可以表示为 O(4^n / sqrt(n))。

# 小结

在编写代码的过程中，我获得了对经典算法问题的深入理解，并且通过实践加深了对这些概念的认识。以下是我对这段经历的一些总结和反馈。

动态规划和字符串处理方面，我体会到了动态规划的强大之处。在处理诸如最长公共子序列和编辑距离等问题时，通过建立状态转移方程，动态规划能够高效地找到解决方案。

对于图算法和最小生成树，学习和实现克鲁斯卡尔算法让我对图的表示方法和相关算法有了更深入的了解。这些知识不仅限于计算机科学领域，还可以应用于其他各种问题。

在深度优先搜索和广度优先搜索方面，我更加明白了递归和队列在解决问题中的重要性。这些搜索策略在处理诸如省份连接和路径规划等问题时非常有效。

贪心算法在解决最小生成树问题时展示了其优势。通过按照权重排序并选择最小的边来构建生成树，克鲁斯卡尔算法得以高效运行。

回溯法在解决排列组合问题时大放异彩。通过递归和剪枝技术，我能够处理诸如有效括号组合和电话号码字母组合等复杂问题。

在编写代码的过程中，我也意识到了良好的代码组织和注释的重要性。清晰的代码结构和注释不仅可以提高代码的可读性，还有助于后期的维护和优化。

总的来说，这段编码经历让我对算法和数据结构有了更深刻的理解，并且让我掌握了许多实用的编程技巧。这不仅帮助我解决了具体问题，还提升了我对计算机科学基础的认识。我期待着将这些知识应用于更广泛的领域和挑战中。

# 源代码（纸质文档不包含源代码）

## 题目一：最长回文子序列

### 题目描述

一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。

### 源代码

import java.util.Arrays;

import java.util.Scanner;

public class exercise1 {

//自底向上

public static int longestPalindromeSubseq(String s) {

int n=s.length();

int[][] max=new int[n][n];

for(int i=n-1;i>=0;i--){

max[i][i]=1;

for(int j=i+1;j<n;j++) {

if(s.charAt(i)==s.charAt(j)){

max[i][j]=max[i+1][j-1]+2;

}else{

max[i][j]=Math.max(max[i][j-1],max[i+1][j]);

}

}

}

return max[0][n-1];

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("输入一个字符串 :");

String s=scanner.next();

int i=longestPalindromeSubseq(s );

System.out.println("该字符串最长回文子序列长度为"+i);

}

}

## 题目二：最长递增子序列

### 题目描述

给一个整数数组 nums ，找到其中最长严格递增子序列的长度

### 源代码

public static int lengthOfLIS(int[] nums) {

int n=nums.length;

int[] lIs=new int[n];

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

lIs[i]=1;

}

for(int i=n-1;i>=0;i--){

for(int j=i+1;j<n;j++){

if(nums[i]<nums[j]) {

lIs[i]=Math.max(lIs[i],lIs[j]+1);

}

}

}

return Arrays.stream(lIs).max().orElse(0);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// 输入数组长度

System.out.print("输入数组长度: ");

int n = scanner.nextInt();

// 输入数组元素

System.out.print("输入数组元素（以空格分隔）: ");

int[] nums = new int[n];

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

nums[i] = scanner.nextInt();

}

System.out.println("最长递增子序列长度为"+lengthOfLIS(nums));

}

## 题目三：最长公共子序列

### 题目描述

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

### 源代码

package group1;

import java.util.Scanner;

public class exercise3 {

public static void main(String[] args) {

// 从标准输入读取两个字符串

Scanner scanner = new Scanner(System.in);

System.out.println("请输入第一个字符串：");

String text1 = scanner.nextLine();

System.out.println("请输入第二个字符串：");

String text2 = scanner.nextLine();

// 调用最长公共子序列方法

int result = longestCommonSubsequence(text1, text2);

// 输出结果

System.out.println("最长公共子序列的长度：" + result);

}

public static int longestCommonSubsequence(String text1, String text2) {

// 上述给出的最长公共子序列方法的实现

int m = text1.length();

int n = text2.length();

int[][] dp = new int[m + 1][n + 1];

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (text1.charAt(i - 1) == text2.charAt(j - 1)) {

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

} else {

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

}

}

}

return dp[m][n];

}

}

## 题目四：编辑距离

### 题目描述

给出两个单词 word1 和 word2， 返回将 word1 转换成 word2 所使用的最少操作数

### 源代码

package group1;

import java.util.Scanner;

public class exercise4 {

public static void main(String[] args) {

// 从标准输入读取两个字符串

Scanner scanner = new Scanner(System.in);

System.out.println("请输入第一个字符串：");

String word1 = scanner.nextLine();

System.out.println("请输入第二个字符串：");

String word2 = scanner.nextLine();

// 调用最小编辑距离方法

int result = minDistance(word1, word2);

// 输出结果

System.out.println("两个字符串的最小编辑距离为：" + result);

}

public static int minDistance(String word1, String word2) {

int m = word1.length();

int n = word2.length();

// 初始化动态规划数组

int[][] dp = new int[m + 1][n + 1];

// 初始化第一行和第一列

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

dp[i][0] = i;

}

for (int j = 0; j <= n; j++) {

dp[0][j] = j;

}

// 填充动态规划数组

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (word1.charAt(i - 1) == word2.charAt(j - 1)) {

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

} else {

dp[i][j] = 1 + Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);

}

}

}

// 返回最终结果

return dp[m][n];

}

}

## 题目五：实现最小生成树的 Kruskal 算法

### 题目描述

利用并查集实现最小生成树的 Kruskal 算法。需要有最小生成树的直观界面展示。

### 源代码

import networkx as nx

import matplotlib.pyplot as plt

class UnionFind:

def \_\_init\_\_(self, vertices):

self.parent = {v: v for v in vertices}

self.rank = {v: 0 for v in vertices}

def find(self, vertex):

if self.parent[vertex] != vertex:

self.parent[vertex] = self.find(self.parent[vertex])

return self.parent[vertex]

def union(self, root1, root2):

if self.rank[root1] > self.rank[root2]:

self.parent[root2] = root1

elif self.rank[root1] < self.rank[root2]:

self.parent[root1] = root2

else:

self.parent[root2] = root1

self.rank[root1] += 1

def kruskal(graph):

edges = []

for v1, v2, data in graph.edges(data=True):

edges.append((v1, v2, data['weight']))

edges.sort(key=lambda x: x[2]) # Sort edges by weight

minimum\_spanning\_tree = nx.Graph()

union\_find = UnionFind(graph.nodes)

for v1, v2, weight in edges:

root1 = union\_find.find(v1)

root2 = union\_find.find(v2)

if root1 != root2:

minimum\_spanning\_tree.add\_edge(v1, v2, weight=weight)

union\_find.union(root1, root2)

return minimum\_spanning\_tree

def visualize\_graph\_fixed\_positions(graph, original\_positions, title):

labels = nx.get\_edge\_attributes(graph, 'weight')

nx.draw(graph, pos=original\_positions, with\_labels=True, font\_weight='bold', node\_size=700, node\_color='skyblue')

nx.draw\_networkx\_edge\_labels(graph, pos=original\_positions, edge\_labels=labels)

plt.title(title)

plt.show()

# 示例用法

manual\_graph = nx.Graph()

# 添加节点

nodes = ['A', 'B', 'C', 'D', 'E']

manual\_graph.add\_nodes\_from(nodes)

# 添加边及权重

edges\_with\_weights = [('A', 'B', 4), ('A', 'C', 2), ('B', 'C', 5), ('B', 'D', 10), ('C', 'D', 1), ('C', 'E', 7), ('D', 'E', 3)]

manual\_graph.add\_weighted\_edges\_from(edges\_with\_weights)

print("随机生成的图:")

original\_positions = nx.spring\_layout(manual\_graph) # 记录原始节点位置

visualize\_graph\_fixed\_positions(manual\_graph, original\_positions, "随机图")

minimum\_spanning\_tree = kruskal(manual\_graph)

print("\nKruskal算法生成的最小生成树:")

for edge in minimum\_spanning\_tree.edges(data=True):

print(edge)

visualize\_graph\_fixed\_positions(minimum\_spanning\_tree, original\_positions, "最小生成树")

## 题目六：验证从湖北出发最多只需要跨越两个省就可以到达中国任何一个省。

### 题目描述

现要验证以下假设：从湖北省出发，经过若干次道路转换，最多只需要跨越两个省就可以到达中国的任何一个省份。

### 源代码

import random

def calculate\_area(f, xrange, yrange):

inside = 0

attempts = 100000

for i in range(attempts):

x = random.uniform(xrange[0], xrange[1])

y = random.uniform(yrange[0], yrange[1])

if f(x,y) >= 0:

inside += 1

area = inside / attempts \* (xrange[1] - xrange[0]) \* (yrange[1] - yrange[0])

return area

def circle(x, y):

return 1 - (x\*\*2 + y\*\*2)\*\*0.5

if \_\_name\_\_ == "\_\_main\_\_":

area = calculate\_area(circle, [-1, 1], [-1, 1])

print(f"Estimated area: {area:.5f}")

## 题目七：比较三分查找算法和二份查找算法平均情形时间复杂度

### 题目描述

设计实验比较三分查找算法和二份查找算法平均情形时间复杂度

### 源代码

import random

def calculate\_area(f, xrange, yrange):

inside = 0

attempts = 100000

for i in range(attempts):

x = random.uniform(xrange[0], xrange[1])

y = random.uniform(yrange[0], yrange[1])

if f(x,y) >= 0:

inside += 1

area = inside / attempts \* (xrange[1] - xrange[0]) \* (yrange[1] - yrange[0])

return area

def circle(x, y):

return 1 - (x\*\*2 + y\*\*2)\*\*0.5

if \_\_name\_\_ == "\_\_main\_\_":

area = calculate\_area(circle, [-1, 1], [-1, 1])

print(f"Estimated area: {area:.5f}")

## 题目八：用蛮力算法和分治算法找到平面的最接近点对

### 题目描述

随机产生平面若干点，利用蛮力算法和分治算法找到平面的最接近点对，并考查随 n 变大时，两者的效率差异、实验效率和理论效率的一致性。平面点集能直观的进行观察。

### 源代码

import random

def calculate\_area(f, xrange, yrange):

inside = 0

attempts = 100000

for i in range(attempts):

x = random.uniform(xrange[0], xrange[1])

y = random.uniform(yrange[0], yrange[1])

if f(x,y) >= 0:

inside += 1

area = inside / attempts \* (xrange[1] - xrange[0]) \* (yrange[1] - yrange[0])

return area

def circle(x, y):

return 1 - (x\*\*2 + y\*\*2)\*\*0.5

if \_\_name\_\_ == "\_\_main\_\_":

area = calculate\_area(circle, [-1, 1], [-1, 1])

print(f"Estimated area: {area:.5f}")

## 题目九：顶点个数，出现有向边的概率与出现环的关系

### 题目描述

随机产生个顶点的有向图，并预设两点之间出现有向边的概率，然后探索前两个参数和图中出现环的关系。需要有图形展示。

### 源代码

import random

def calculate\_area(f, xrange, yrange):

inside = 0

attempts = 100000

for i in range(attempts):

x = random.uniform(xrange[0], xrange[1])

y = random.uniform(yrange[0], yrange[1])

if f(x,y) >= 0:

inside += 1

area = inside / attempts \* (xrange[1] - xrange[0]) \* (yrange[1] - yrange[0])

return area

def circle(x, y):

return 1 - (x\*\*2 + y\*\*2)\*\*0.5

if \_\_name\_\_ == "\_\_main\_\_":

area = calculate\_area(circle, [-1, 1], [-1, 1])

print(f"Estimated area: {area:.5f}")

## 题目十：电话号码的字母组合

### 题目描述

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 任意顺序 返回。

### 源代码

package group3;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

class exercises1 {

public static List<String> letterCombinations(String digits) {

List<String> combinations = new ArrayList<String>();

if (digits.length() == 0) {

return combinations;

}

Map<Character, String> phoneMap = new HashMap<Character, String>() {{

put('2', "abc");

put('3', "def");

put('4', "ghi");

put('5', "jkl");

put('6', "mno");

put('7', "pqrs");

put('8', "tuv");

put('9', "wxyz");

}};

backtrack(combinations, phoneMap, digits, 0, new StringBuffer());

return combinations;

}

public static void backtrack(List<String> combinations, Map<Character, String> phoneMap, String digits, int index, StringBuffer combination) {

if (index == digits.length()) {

combinations.add(combination.toString());

} else {

char digit = digits.charAt(index);

String letters = phoneMap.get(digit);

int lettersCount = letters.length();

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

combination.append(letters.charAt(i));

backtrack(combinations, phoneMap, digits, index + 1, combination);

combination.deleteCharAt(index);

}

}

}

public static void main(String[] args) {

List<String> result1 = letterCombinations("23");

System.out.println("Combinations for input digits \"23\": " + result1);

}

}

## 题目十一：生成括号

### 题目描述

正整数 n 代表生成括号的对数，请设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

### 源代码

import java.util.ArrayList;

import java.util.List;

public class GenerateParentheses {

public List<String> generateParenthesis(int n) {

List<String> result = new ArrayList<>();

backtrack(result, "", 0, 0, n);

return result;

}

private void backtrack(List<String> result, String current, int open, int close, int max) {

// 如果当前字符串的长度达到了括号对数的两倍，说明是一个有效的组合，加入结果列表

if (current.length() == max \* 2) {

result.add(current);

return;

}

// 如果左括号的数量小于最大限制，可以添加左括号

if (open < max) {

backtrack(result, current + "(", open + 1, close, max);

}

// 如果右括号的数量小于左括号的数量，可以添加右括号

if (close < open) {

backtrack(result, current + ")", open, close + 1, max);

}

}

public static void main(String[] args) {

GenerateParentheses solution = new GenerateParentheses();

List<String> result = solution.generateParenthesis(3);

System.out.println(result);

}

}