

算法分析与设计  
课程设计

学 院 两江人工智能学院

专 业 软件工程

班 级 121230202

学生姓名 王鹏

学号 12109990907

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 题组一题目一：最长回文子序列 1](#_Toc154665386)

[1.1 题目描述 1](#_Toc154665387)

[1.2 算法描述 1](#_Toc154665388)

[1.3 程序运行及其结果 2](#_Toc154665389)

[1.4 时间复杂度分析 2](#_Toc154665390)

[2 题组一题目二：最长递增子序列 3](#_Toc154665391)

[2.1 题目描述 3](#_Toc154665392)

[2.2 算法描述 3](#_Toc154665393)

[2.3 程序运行及其结果 4](#_Toc154665394)

[2.4 时间复杂度分析 4](#_Toc154665395)

[3 题组一题目三：最长公共子序列 5](#_Toc154665396)

[3.1 题目描述 5](#_Toc154665397)

[3.2 算法描述 5](#_Toc154665398)

[3.3 程序运行及其结果 5](#_Toc154665399)

[3.4 时间复杂度分析 6](#_Toc154665400)

[4 题组一题目三：编辑距离 7](#_Toc154665401)

[4.1 题目描述 7](#_Toc154665402)

[4.2 算法描述 7](#_Toc154665403)

[4.3 程序运行及其结果 8](#_Toc154665404)

[4.4 时间复杂度分析 8](#_Toc154665405)

[5 题组一题目五：最长等差子序列 9](#_Toc154665406)

[5.1 题目描述 9](#_Toc154665407)

[5.2 算法描述 9](#_Toc154665408)

[5.3 程序运行及其结果 10](#_Toc154665409)

[5.4 时间复杂度分析 11](#_Toc154665410)

[6 题组一题目六：雀魂启动 12](#_Toc154665411)

[6.1 题目描述 12](#_Toc154665412)

[6.2 算法描述 12](#_Toc154665413)

[6.3 程序运行及其结果 13](#_Toc154665414)

[6.4 时间复杂度分析 13](#_Toc154665415)

[7 题组一题目七：通配符匹配 14](#_Toc154665416)

[7.1 题目描述 14](#_Toc154665417)

[7.2 算法描述 14](#_Toc154665418)

[7.3 程序运行及其结果 14](#_Toc154665419)

[7.4 时间复杂度分析 15](#_Toc154665420)

[8 题组二题目一：并查集实现最小生成树的Kruskal算法 16](#_Toc154665421)

[8.1 题目描述 16](#_Toc154665422)

[8.2 算法描述 16](#_Toc154665423)

[8.3 程序运行及其结果 17](#_Toc154665424)

[8.4 时间复杂度分析 18](#_Toc154665425)

[9 题组二题目二：湖北跨省 19](#_Toc154665426)

[9.1 题目描述 19](#_Toc154665427)

[9.2 算法描述 19](#_Toc154665428)

[9.3 程序运行及其结果 19](#_Toc154665429)

[9.4 时间复杂度分析 20](#_Toc154665430)

[10 题组二题目三：二分查找和三分查找比较 21](#_Toc154665431)

[10.1 题目描述 21](#_Toc154665432)

[10.2 算法描述 21](#_Toc154665433)

[10.3 程序运行及其结果 22](#_Toc154665434)

[10.4 时间复杂度分析 23](#_Toc154665435)

[11 题组二题目四：蛮力算法和分治算法效率比较 24](#_Toc154665436)

[11.1 题目描述 24](#_Toc154665437)

[11.2 算法描述 24](#_Toc154665438)

[11.3 程序运行及其结果 26](#_Toc154665439)

[11.4 时间复杂度分析 28](#_Toc154665440)

[12 题组二题目五：任务分配问题 30](#_Toc154665441)

[12.1 题目描述 30](#_Toc154665442)

[12.2 算法描述 30](#_Toc154665443)

[12.3 程序运行及其结果 33](#_Toc154665444)

[12.4 时间复杂度分析 33](#_Toc154665445)

[13 题组三题目一：电话号码的字母组合 34](#_Toc154665446)

[13.1 题目描述 34](#_Toc154665447)

[13.2 算法描述 34](#_Toc154665448)

[13.3 程序运行及其结果 35](#_Toc154665449)

[13.4 时间复杂度分析 35](#_Toc154665450)

[14 题组三题目二：括号生成 36](#_Toc154665451)

[14.1 题目描述 36](#_Toc154665452)

[14.2 算法描述 36](#_Toc154665453)

[14.3 程序运行及其结果 37](#_Toc154665454)

[14.4 时间复杂度分析 37](#_Toc154665455)

[15 题组三题目三：单词搜索 38](#_Toc154665456)

[15.1 题目描述 38](#_Toc154665457)

[15.2 算法描述 38](#_Toc154665458)

[15.3 程序运行及其结果 39](#_Toc154665459)

[15.4 时间复杂度分析 40](#_Toc154665460)

[16 选做题目三：超市选址 41](#_Toc154665461)

[16.1 题目描述 41](#_Toc154665462)

[16.2 算法描述 41](#_Toc154665463)

[16.3 程序运行及其结果 42](#_Toc154665464)

[16.4 时间复杂度分析 43](#_Toc154665465)

[17 小结 44](#_Toc154665466)

# 题组一题目一：最长回文子序列

## 题目描述

给你一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

## 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）

本题属于动态规划问题，可以使用递归+二维数组的方式进行计算。设置一个二维数组用来存储下标范围内最长的回文子序列。同时此二维数组可以用来避免计算相同子问题的，优化了递归方法。

若此字符串的结尾相同，则缩小为计算除去头和尾的的最长回文子序列，然后再在答案中加2。若不相同，则从左或右开始进行寻找。

此递归方程如下：

f(s,i+1,j-1)+2 s[i]=s[j]

f(s,i,j)=

max(f(s,i+1,j),f(s,i,j-1)) // 比较从左和从右

1. 伪代码（以及必要的注释）

Function palindrome(s,i,j)

//用于返回最长文字子序列

// 输入：字符串，字符串头部下标，字符串尾部下标

// 输出：最长回文子序列

if i==j then

return 1 // 每个字符本身是回文

if i>j then

return 0 // 此时已经越界

if memo[i][j]!=0 then

return memo[i][j] //此时说明以j和i为起始点的回文已经计算过

if s.charAt(i).equals(s.charAt(j)) then // 头尾相同，化为更小问题

res palidrome(s, i+1, j-1) + 2 // 当前字符可以加入回文序列

else

res = maximum of palidrome(s, i+1, j) and palidrome(s, i, j-1)

return res //返回最长回文子序列

end

## 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 | 输出 |
| bbbab | 4 |
| cbbd | 3 |

1. 运行结果截图

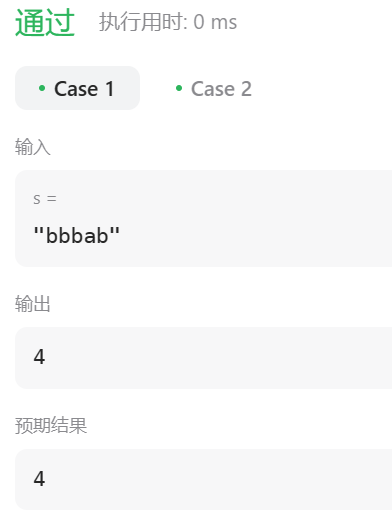


图1 最长回文子序列运行结果

## 时间复杂度分析

备忘录是一个二维数组，用来存储已经计算过的子问题的结果，避免重复计算。备忘录的大小是 n\*n，而每个子问题只会被计算一次，所以总的计算量是 O (n^2)。

# 题组一题目二：最长递增子序列

## 题目描述

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列

## 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）

本算法使用贪心算法和二分查找的思想。

首先定义一个数组d，用来记录递增子序列，并且d[i]表示长度为i的递增子序列的最后一个元素为最小值。最开始，使得d[i]=nums[0]，然后从前往后遍历数组nums，如果num[i]>d[len],则将此元素加入到d数组中，d数组长度len加一；如果num[i]<d[len],则使用二分查找的算法，找到比元素小的d[k],将此元素加入到d[k]后面,d[k+1]=nums[i]，直到数组遍历完。

1. 伪代码（以及必要的注释）

Function LIS(nums[])

// 计算出最长递增子序列的长度

// 输入：目标数组

// 输出：子序列长度

if nums.length==0

return

for i1 to nums.length do

// 数组第i个元素大递增子序列的最大一个元素

if nums[i]>sub[len]

sub[++len]nums[i]

else

// 使用二分查找，减少时间

while l<= len then

mid(l+len)>>1 // 取子序列数组的中间值

if sub[mid]<nums[i] // 如果此元素处于数组右侧

posmid

lmid+1

else

len=mid-1

end

sub[len+1]nums[i] // 将此元素加入到子序列数组

return len // 返回最长子序列的长度

## 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： | 输出： |
| [10,9,2,5,3,7,101,18] | 4 |
| [0,1,0,3,2,3] | 4 |
| [7,7,7,7,7,7,7] | 1 |

1. 运行结果截图

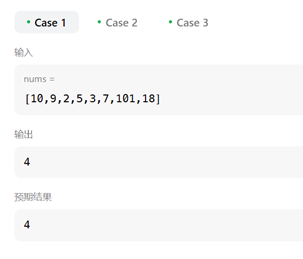


图2 最长递增子序列运行结果

## 时间复杂度分析

数组 nums 的长度为 n，我们依次用数组中的元素去更新 d 数组，而更新 d 数组时需要进行 O(logn) 的二分搜索

综上所诉，总时间复杂度为 O(nlogn)

# 题组一题目三：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 。

## 算法描述

1. 描述思路

此问题是动态规划问题，可以用备忘录和递归的方式实现。定义一个二维数组，用来存储访问过的每个结点，防止重复访问，用来优化递归算法。

从每个字符串的结尾开始向前寻找，刚开始数组所有内容全部定为-1，表示没有访问过。若当前两个字符相等，则规模各-1，同时在最长公共子序列的长度+1。若当前两个字符不相等，则是从分别以当前的字符作为最后一个字符，分别向前寻找，直到寻找到字符串边界，返回值。最后取两个返回值的最大值，作为最长公共子序列。

递归方程如下：

0 i<0,j<0

cache[i][j] cache[i][j]!=-1

f(i,j)= f(i-1,j-1)+1 t1[i]=t2[j]

max(d(i-1,j),f(i,j-1)) others

1. 伪代码

Function commonSubsequence(i,j)

// 运用二维数组和递归的方式，输出最长公共子序列

// 输入：两个字符串的最后一个字符的下标

// 输出：最长公共子序列

if i<0 or j<0

return 0

if cache[i][j]!=-1 // 若此节点访问过

return cache[i][j]

if text1[i]==text2[j]

// 同时缩小规模，长度+1

return cache[i][j] commonSubsequence(i-1,j-1)+1

else

return cache[i][j] max(commonSubsequence(i-1,j), commonSubsequence(i,j-1)) //取两边的最大值

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入 | 输出 |
| abcde ace | 3 |
| abc abc | 3 |
| abc def | 0 |

1. 运行结果截图



图3 最长公共子序列运行结果

## 时间复杂度分析

在递归函数中最坏情况下需要计算所有不同的i和j的组合，每次递归调用都在i和j上递减，并且每个位置的状态只计算一次，所以总的递归调用次数是O(n\*m)，n是第一个字符串的长度，m是第二个字符串的长度

# 题组一题目三：编辑距离

## 题目描述

给你两个单词 word1 和 word2， 请返回将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符

删除一个字符

替换一个字符

## 算法描述

1. 描述思路

可以化为规模更小的问题，则可以使用动态规划的思想，通过使用递归和备忘录的方法，来实现动态规划算法。

备忘录就和动态规划中的dp数组是一个道理，用来记录计算结果，避免重复计算来消耗时间。

每一次的插入，删除，修改都是一次操作，需要+1，删除和插入都只涉及一个字符串，所有一个字符串的指针向后移动一个不动即可，修改则是涉及到指针在同一个位置进行向后移动，然后对操作过的元素加入到备忘录中，然后调用递归方程，持续操作，直到一边的字符串查找完毕。

递归方程如下：

f(i-1,j-1) s1[i]==s2[j]

f(i,j)

min(f(i-1,j),f(i,j-1),f(i-1,j-1))+1

1. 伪代码

Function dp(word1,word2,i,j)

// 返回最小操作次数

// 输入：两个字符串，和各自的下标（从后往前）

// 输出：最小操作次数

if i==-1 return j+1

if j==-1 return i+1

if memo[i][j]!=-1

return memo[i][j] // 此时元素以及计算过，直接取用

if word1.charAt(i)==word2.charAt(j)

memo[i][j]dp(word1,word2,i-1,j-1) //元素相同，同时后移

else

memo[i][j]min(dp(word1,word2,i-1,j), dp(word1,word2,i,j-1),

dp(word1,word2,i-1,j-1))+1 // 进行三种操作取最小值，因为最后一位不同，需要加+1

return

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入 | 输出 |
| horse ros | 3 |
| intention execution | 5 |

1. 运行结果截图



图 4 编辑距离运行结果

## 时间复杂度分析

因为用到递归和备忘录，备忘录是二维，长宽分别为两个字符串的长度n和m，而递归的时间复杂度和递归深度有关系。

综上所述，时间复杂度是O(n\*m)

# 题组一题目五：最长等差子序列

## 题目描述

给你一个整数数组 nums，返回 nums 中最长等差子序列的长度。

示例 1：

输入：nums = [3,6,9,12]

输出：4

示例 2：

输入：nums = [9,4,7,2,10]

输出：3

## 算法描述

1. 描述思路

方法：

满足条件的解，必是以某两项作为该等差数列的最后两项。

dp[i][j] = n 表示以 A[i] A[j]作为最后两项的等差数列的最大长度为 n。

思路：

如果输入的数组长度为null或者小于3，则返回0或本身长度。若大于三，则用双层遍历，对于每个j，调用longLength方法，传入数组A，下标i和j，哈希表map，和二维数组dp2，得到以A[i]和A[j]为末尾两项的等差数列的长度，用num表示。比较num和ans，取较大值，并将当前元素放入哈希表，方便查找，遍历结束后，ans为最终结果。

在longLength方法中，判断判断二维数组dp2中dp2[i][j]是否不为0，如果不为0，说明已经计算过以A[i]和A[j]为末尾两项的等差数列的长度，直接返回dp2[i][j]。若i下标为0，则i是第一个元素，将2赋给dp2[i][j]，并返回2。计算A[j]和A[i]的差值，用diff表示，为公差。判断哈希表map中是否存在一个键，使得A[i]-A[k] = diff，其中k是该键对应的值，即数组中的下标。如果存在，说明存在一个元素A[k]，使得A[k]，A[i]，A[j]构成一个等差数列。此后运用递归，得到等差数列的长用preAns表示。将preAns + 1赋给dp2[i][j]，并返回preAns + 1，这是以A[i]和A[j]为末尾两项的等差数列的长度。

如果哈希表map中不存在这样的键，说明没有找到A[k]，那么以A[i]和A[j]为末尾两项的等差数列的长度只能是2，将2赋给dp2[i][j]，并返回2。

递归公式：

f(A, i, j, map, dp2) = dp2[i][j] // dp2[i][j]!=0

f(A,i,j,map,dp2) f(A, i, j, map, dp2) = 2 //i=0 map不存在元素

f(A, i, j, map, dp2) = f(A, map.get(A[i]-diff), i, map, dp2) + 1 // map中存在A[i]-diff的键

1. 伪代码

Function getLength(a)

// 用于返回最长等差序列长度

// 输入：数组

// 输出：最长等差序列长度

if a==null

return 0

else a.length<3

return a.length

else

for i  0 to a.length do

for j i+1 to a.length do

// 调用递归目标函数 输入数组，下标，哈希表，dp2数组

num  longLength(a,i,j,map,dp2)

ans  max(ans,num) //依次选择长度最大值

end

map.put(a[i],i) // 哈希表存入当前元素，方便查找

return ans；

end

Function longLength(a,i,j,map,dp2)

// 计算最长等差数列长度

// 输入：输入数组，下标，哈希表，dp2数组

// 输出：当前i结点为中间元素的等差序列的长度

if dp2[i][j]!=0

return dp2[i][j]

else i==0

return 0

else

// diff为公差，判断map中是否存在元素满足此公差

if map.containsKey(A[i]-diff)

// 递归调用函数，一直寻找是否有满足公差的元素

preAns  longLength(A, map.get(A[i]-diff), i, map, dp2)

dp2[i][j]=preAns+1 // 此时的元素所满足的最长序列产长度

return preAns+1

else

dp2[i][j]=2

return 2

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入 | 输出 |
| [3,6,9,12] | 4 |
| [9,4,7,2,10] | 3 |
| [20,1,15,3,10,5,8] | 4 |

1. 运行结果截图

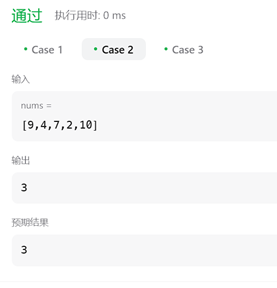


图 5 最长等差子序列运行结果

## 时间复杂度分析

对于getLength函数，它有一个双重循环，因此，这部分的时间复杂度是O (n^2)。其次，对于longLength函数，它有一个if-else语句，判断dp2 [i] [j]是否为0，如果不为0，直接返回dp2 [i] [j]，这部分的时间复杂度是O (1)；如果为0，再判断是否存在k，使得A [i]-A [k] = diff，如果存在，递归调用longLength函数，并更新dp2 [i] [j]，这部分的时间复杂度是O (n)；如果不存在，直接将dp2 [i] [j]设为2，这部分的时间复杂度是O (1)。

综上，这段代码的总的时间复杂度是O (n^2) \* O (n) = O (n^3)。

# 题组一题目六：雀魂启动

## 题目描述

总共有36张牌，每张牌是1~9。每个数字4张牌。手里有其中的14张牌，如果这14张牌满足如下条件，即算作和牌

14张牌中有2张相同数字的牌，称为雀头。

除去上述2张牌，剩下12张牌可以组成4个顺子或刻子。

顺子的意思是递增的连续3个数字牌（例如234,567等），刻子的意思是相同数字的3个数字牌（例如111,777）

## 算法描述

1. 描述思路

这段算法基于递归的思想，同时也有回溯的思想，作为一个回溯的变形，不断试错，尝试了所有可能的方法，直到一个可行的解或者遍历完了所有情况。

1. 伪代码

Function judgeHu(arr,total,hasHead)

// 判断输入的牌组是否可以和牌

// 输入：牌数组，牌的数量，是否有雀牌

// 输出：是否能够和牌true或false

if total==0 return true

if !hasHead //没有雀牌

for i0 to 9 do

if arr[i]>=2 //判断同一数字的牌面数量是否大于2

arr[i]-=2

if(judgeHu(arr,total-2,true))return true //大于2则有雀牌

arr[i]+=2 //没有雀牌，返回原数组

end

return false

else

for i0 to 9 do

if arr[i]>0

// 寻找刻子

if arr[i]>=3

arr[i]-=3

if(judgeHu(arr,total-3,true))return true //有刻子，继续寻找

arr[i]+=3 //没有顺子，返回原数组

end

// 寻找顺子

If i+2<9 and arr[i+1]>0 and arr[i+2]>0

arr[i]--

arr[i+1]—

arr[i+2]—

if(judgeHu(arr,total-3,true))return true //有顺子，继续寻找

arr[i]++

arr[i + 1]++

arr[i + 2]++

end

end

return false

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

输入：1 1 1 2 2 2 3 3 3 5 7 7 9

输出：不能和牌

1. 运行结果截图

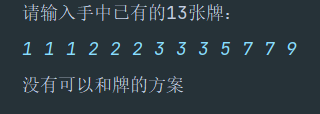


图 6 雀神启动运行结果

## 时间复杂度分析

算法中使用了两个for循环，在内部循环中，递归和加入的操作的时间复杂度可以看作O（1）。

综上所诉，该算法的时间复杂度为O(n^2)

# 题组一题目七：通配符匹配

## 题目描述

给你一个输入字符串 (s) 和一个字符模式 (p) ，请你实现一个支持 '?' 和 '\*' 匹配规则的通配符匹配：

'?' 可以匹配任何单个字符。

'\*' 可以匹配任意字符序列（包括空字符序列）。

判定匹配成功的充要条件是：字符模式必须能够 完全匹配 输入字符串（而不是部分匹配）。

## 算法描述

1. 描述思路

这道题使用了贪心算法的思想，若成功，则继续向后寻找，若失败，则退回到失败之前的位置继续进行寻找。

遍历字符串s，依次比较s和p中对应位置的字符，如果相等或者p中的字符是'?'，则两个指针同时后移。如果p中的字符是'\*'，则记录当前s和p的位置为sStar和pStar，并将pPos向后移动。如果sPos不为-1，即被\*匹配过，向后移动，pPos指针回到此位置，这段即为回溯。

如果以上情况都不满足，则匹配失败。在遍历结束后，还需要检查模式字符串p剩余部分是否都是'\*'，如果是则匹配成功，否则匹配失败。

1. 伪代码

Function isMatch(s,p)

// 用于匹配目标字符串和字符模式

// 输入：目标字符串s，字符模式p

// 输出：是否匹配 true or false

While sPos<s.length then // sPos为此时s指针，初值为0

If pPos<p.length and (s.charAt(sPos)==p.charAt(pPos) or p.charAt(pPos)=='?') //s和p标记字符相等 或 p字符是'?'

sPos++

pPos++

else pPos<p.length and p.charAt(pPos)=='\*'

sStarsPos//记录遇到\*的位置 初始值为-1

pStarpPos

else sStar!= -1

pPospStar //指针重新回溯到最近的\*的位置

sPos++sStar

else

return false

end

while pPos<p.length and p.charAt(pPos)=='\*'

pPos++

return pPos==n

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入 | 输出 |
| aa a | false |
| aa \* | true |
| cb ?a | false |

1. 运行结果截图

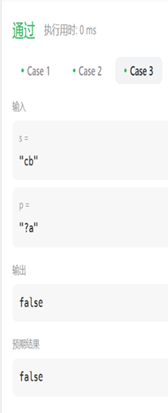
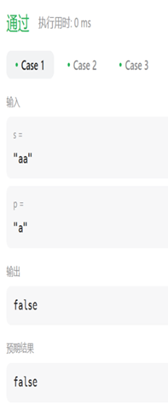


图 7 通配符匹配运行结果

## 时间复杂度分析

有两个while，最坏的情况分别是两个字符串都跑完，分别是O(m)和O（n）,分别为两个字符串的长度，所以这段代码的时间复杂度是O(n\*m)

# 题组二题目一：并查集实现最小生成树的Kruskal算法

## 题目描述

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

## 算法描述

1. 描述思路

Kruskal算法，是生成最小生成树的一种贪心算法，同时运用了并查集，提升了Kruskal算法的时间效率；

定义了一个边的结构类，包括起点，终点和权值。定义一个并查集，包括了一个父节点的数组、一个树的层级数组和一个未连通结点的个数count。初始定义每个结点的父类为本身，则父节点数组相对应的数为本身下标，层级为一。并查集内，有find（int Index）函数，运用递归算法，调用本身最终找到根结点，isConnected（int aIndex,int bIndex）函数，通过调用find函数寻找根节点是否相同来判断两个元素是否连通，若未连通，则通过union（int aIndex，int bIndex）函数来连通两个元素，通过比较树的层级，从而确定层级树大的元素为根节点，通过路径压缩来避免形成链表样式。每连通一个元素，count- -，直到减为1，表示所有元素连通。

主要的Kruskal算法，先将边数组通过权重进行升序排序。通过遍历边数组，对于每一条边，判断它的起点和终点是否已经连通，如果不连通，则表示不会形成环，就将它加入结果列表，并将它的起点和终点进行合并，同时更新权重之和。如果已经连通，就跳过这条边。当结果并查集的count为一时，最小生成树完成。

1. 伪代码

Function find(index)

// 寻找元素的根节点

// 输入：元素下标

// 输出：根节点下标

if parentIndex==index

return index

else

parents[index]<-find(parents[index]) // 递归寻找根节点

return parents[index]

end

Function isConnected(from,to)

// 判断两个元素是否连结

// 输入：一条边的起始结点和终点

// 输出：true或false

return find（from）==find（to）// 判断根节点是否相同

end

Function union(aIndex,bIndex)

// 连接两个元素

// 输入：两个元素的下标

// 输出：未连通元素个数减一

if rank[find(aIndex)]>rank[find(bIndex)] // 比较层级

// 以层级大的为根节点

parents[find(bIndex)] <- find(aIndex)

else

parents[find(aIndex)] <- find(bIndex)

count--

end

Function Kruskal(graph)

// 该算法用于生成一个最小生成树

// 输入：一个带权无向图

// 输出：最小生成树（边列表形式）

Graph.sort();//根据权值升序排序

for i <- 0 to graph.length do

if !isConnected(graph(i).from,graph(i).to) //判断边是否连通

union(graph(i).from,graph(i).to) //连接两个元素

result.add(graph(i))

if count==1

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

输入示例：一个较为复杂的带权无向图

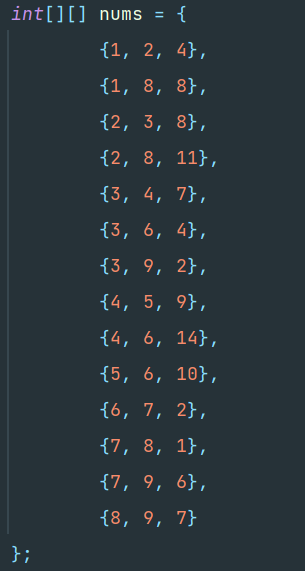


图8 带权无向图输入示例

1. 运行结果截图

运行结果：红色路径是最小生成树

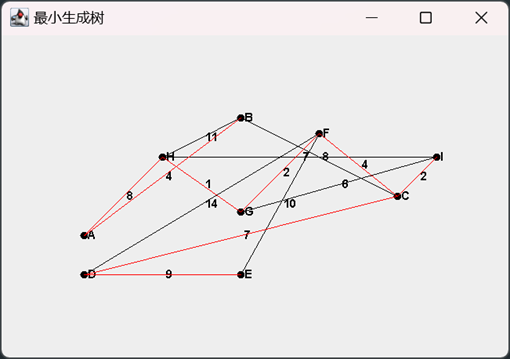


图 9 最小生成树运行情况

## 时间复杂度分析

Kruskal算法主要有两个步骤，第一个是排序，时间复杂度是O(E log E)，其中 E 是边的个数。第二个步骤需要用到并查集这种数据结构，它可以在 O(α(V)) 的时间内完成查找和合并两个顶点所属的集合的操作，其中 α(V) 是Kruskal算法的反函数，它的增长速度非常慢，可以近似认为是一个常数。因为最多需要进行 V - 1 次查找和合并操作，其中 V 是顶点的个数，所以这个步骤的时间复杂度是 O(V α(V))。所以，Kruskal算法的总的时间复杂度是 O(E log E + V α(V))，由于 E 最多可以达到 V^2，所以这个时间复杂度也可以写成 O(E log V + V α(V))。在实际应用中，由于 α(V) 的值非常小，所以可以忽略不计，因此Kruskal算法的时间复杂度可以简化为 O(E log E) 或者 O(E log V)。

# 题组二题目二：湖北跨省

## 题目描述

据说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。请设计实验对此进行验证。

## 算法描述

1. 描述思路

算法思路：

此问题可以将题目化为以湖北省为源点，将中国地图转化为无权无向图，实验验证即是源点到图的各个点的最短路径长度是否都小于3.

因为需要访问最短距离小于3的节点，不需要访问更多的节点，所以这道题选择广度优先遍历是最符合的题解算法

算法描述：

首先创建一个队列queue和一个用于记录到源点距离的数组distance，其初值都为无穷大，将起始节点即湖北放入队列中。

当队列不为空时，执行以下步骤：

从队列中取出一个节点，记为当前节点currentNode。

遍历currentNode的所有相邻节点neibor，若是此点的距离值为最大，即未被访问，则更新neighbor的距离为distance[currentNode] + 1，currentNode的距离加一，如果neighbor的路径path[neighbor]是空的，说明它还没有被记录过复制currentNode的路径path[currentNode]到path[neighbor]，将currentNode加入path[neighbor]的末尾，表示currentNode是neighbor的前驱节点。将neighbor加入队列，等待下一轮访问

1. 伪代码

Function BFS(source)

// 广度优先遍历，输出距离小于3的地图链表

// 输入：源点（湖北）

// 输出：地图链表

While queue is not empty then

currentNodequeue.poll

//遍历此节点的子节点

for neighbor in adjacencyList[currentNode] do

if distance[neighbor]==Integer.MAX //此节点未被访问

// neighbor的距离为distance[currentNode] + 1，currentNode的距离加一

distance[neighbor]distance[currentNode]+1

if path[neighbor]==null

path[neighbor]path[currentNode]//复制路径

path[neighbor].add(currentNode);//curentNode为前驱节点

queue.add(neighbor);

## 程序运行及其结果

1. 给出程序接收的输入和输出

无输入

1. 运行结果截图



图 10 湖北跨省运行情况

## 时间复杂度分析

此算法是广度优先遍历，节点个数是v，边的个数是e，最坏的情况下，访问所有节点和边，并且存储每个节点的距离和路径，则时间复杂度为O(V+E)，但是在无向图中，边数与节点数成正比，e=O（v），最终时间复杂度为O(V)

# 题组二题目三：二分查找和三分查找比较

## 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

二分查找：

与分治法相似，每次将问题的规模都缩小为原来的一半。查找时，每次将目标数与数组中间值进行比较，若是目标值比中间值大，则在右侧继续递归，进行二分查找，直到找到目标值。

递归函数如下：

mid array[mid]=target

f(array,left,right)= f(array,left,mid-1) array[mid]>target

f(array,mid+1,right) array[mid]<target

三分查找：

算法相似，将问题规模依次缩小为原来的三分之一，正因为三分之一的划分，增加了查找的次数。每次将目标与三分之一处和三分之二处的值进行比较，然后再在相对因的区间进行规模更小的查找，直到找到目标值。

递归函数如下：

mid1 array[mid1]=target

mid2 array[mid2]=target

f(array,left,right)= f(array,left,mid1-1) array[mid1]>target

f(array,mid1+1,mid2-1)array[mid1]<target<array[mid2]

f(array,mid2+1,right) array[mid2]<target

1. 伪代码

function binarySearch(arr, target, left, right)

// 二分查找 输出找到目标值的次数

//输入：有序数组arr，目标数target，左端left，右端right

//输出：比较次数count

while left <= right then

count++

mid  left + (right - left) / 2

if arr[mid] == target

return count

if target < arr[mid]// 目标值在左侧区间

binarySearch(arr, target, left, mid-1);

else:// 目标值在右侧区间

binarySearch(arr, target, mid2+1, right);

return count

function ternarySearch(arr, target，left，right)

// 三分查找 寻找目标函数并且返回寻找次数

//输入：有序数组arr，目标数target，左端left，右端right

//输出：比较次数count

while left <= right then

count++

// 分割查找范围

mid1  left + (right - left) / 3

mid2 right - (right - left) / 3

if arr[mid1] == target or arr[mid2] == target

return count

if target < arr[mid1] // 目标值在左侧区间

ternarySearch(arr, target，left，mid1-1)

elif target > arr[mid2] // 目标值在右侧区间

count++

ternarySearch(arr, target，mid2+1，right)

else// 目标值在中间区间

count++

ternarySearch(arr, target，mid1+1，mid2-1)

return count

## 程序运行及其结果

1. 给出程序接收的输入和输出

无输入

1. 运行结果截图

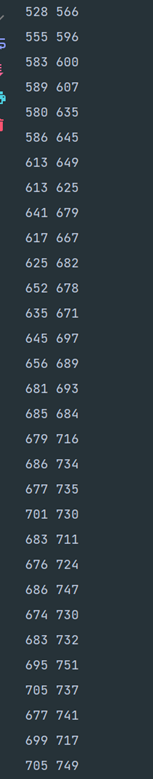


图 11 二分查找和三分查找的比较结果图

## 时间复杂度分析

二分查找：每次比较次数为1，同时规模为上一次的一半：

T(n)=T(n/2)+O(1) T(1)=1

最终时间复杂度为O(log₂n)

三分查找：平均比较次数为3/2，同时规模为上次的三分之一：

T(n)=T(n/3)+O(3/2) T(1)=1

最终时间复杂度为O(3/2log₃n)

# 题组二题目四：蛮力算法和分治算法效率比较

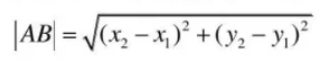
## 题目描述

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

暴力算法：

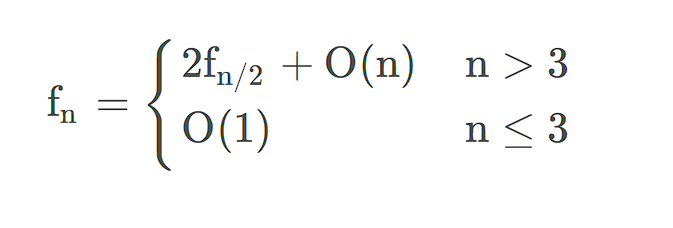


根据以上公式，通过两个for循环，依次比较每个点与除自己以外的点的公式，依次替换取最小值，从而得到答案。

分治法：

分治法会先将所有点按照x坐标来进行升序排序，然后将结点从x轴中间划分为两部分，两部分分别对结点之间的距离进行计算，同样递归求最小值，这是递归的求解步骤，分别对左右两个子集调用递归函数，得到两个子问题的最近点对和最小距离，然后取较小值作为当前的最小距离。同样，有递归的合并步骤，因为有可能会有两个之间距离更小的结点分别处于两边，在中间区域内，找出所有 x 坐标在 [midX-d3,midX+d3] 范围内的点，其中midx为中间的x坐标，d3为两边递归求出的最小距离。最后，进行递归的优化，将列表中的点按照 y 坐标排序，然后对每个点，只需与它在 y 坐标上相邻的 6 个点（或者 4 个点）比较距离，如果小于当前的最小距离，就更新最小距离。 最终得出最小距离。

递归算法：



本算法递归方程：

// points为结点数组 from ，midx为起始坐标 ，n为此时一侧的结点个数

func(Arrays.copyOfRange(points, from midx), n);

1. 伪代码

暴力算法：

Function bruteForce(pointNum,pointList)

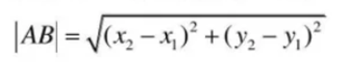
// 得出两点间的最小距离

// 输入：结点个数，结点列表

// 输出：最小距离

for i🡨0 to pointList.size-1 do

for j🡨i+1 to pointList.size do

distance🡨 //距离公式计算

if distance<double.MAX\_VALUE //距离小于double的枚举

mid🡨distance

end

end

return distance

end

分治法：

Function closetPair(points,pointNum)

// 计算两点之间最短距离

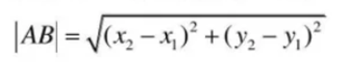
// 输入：结点数组，结点个数

// 输出：最短距离

If pointNum<=3

for i🡨0 to points.length-1 do

for j🡨i+1 to points.length do

distance🡨 //距离公式计算

if distance<double.MAX\_VALUE //距离小于double的枚举

mid🡨distance

end

end

return distance

else

Arrays.sort(points, new CompareX());// 将点按照x坐标升序排序

mid 🡨 pointNum/2

midX🡨points[mid].getX

//递归地对左右两个子集调用分治法，得到两个子问题的最近点对和最小距离

d1🡨 closestPair(Arrays.copyOfRange(points, 0, mid), mid)

d2🡨 closestPair(Arrays.copyOfRange(points, mid, pointNum), pointNum - mid)

d3🡨min(d1,d2)

for i🡨0 to pointNum do

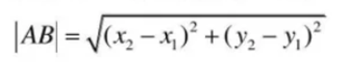
//abs求绝对值，此表示是否在[midX-d3,midX+d3]范围内

if Math.abs(points[i].getX() - midX) < d3

strip.add(point[i])//新的结点列表添加元素

for i🡨0 to strip.size do

for j🡨i+1 to strip.size and (纵坐标距离小于d3) do

d4🡨 //距离公式计算

if d4<d3

d3🡨d4

end

end

return d3

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

输入：结点数量分别为10和50

输出：最小点对

1. 运行结果截图

输入示例：输入节点数为10

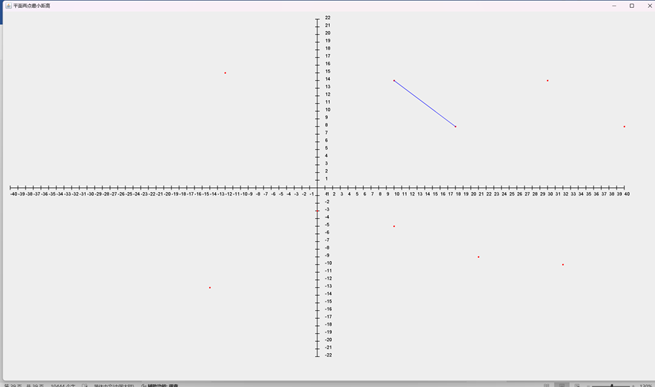


图 12 最小点对平面展示图

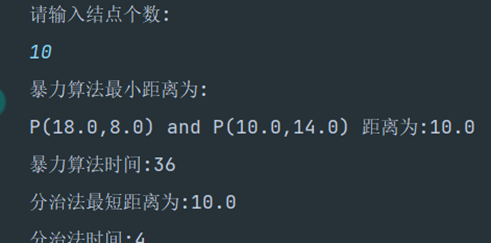


图 13 最小点对算法结果比较

输入示例：输入节点数为50

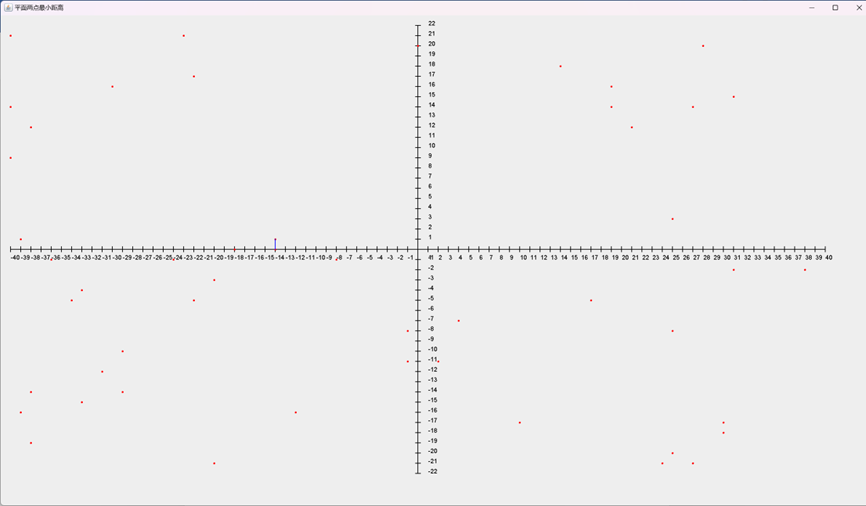


图 14 最小点对展示图

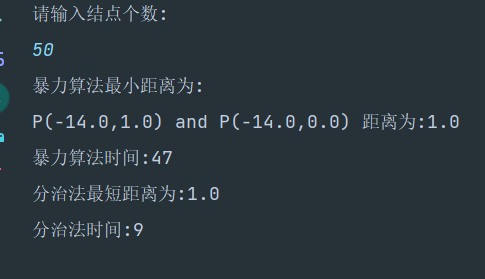


图 15 最小点对算法比较

测试数据：



图 16 测试数据

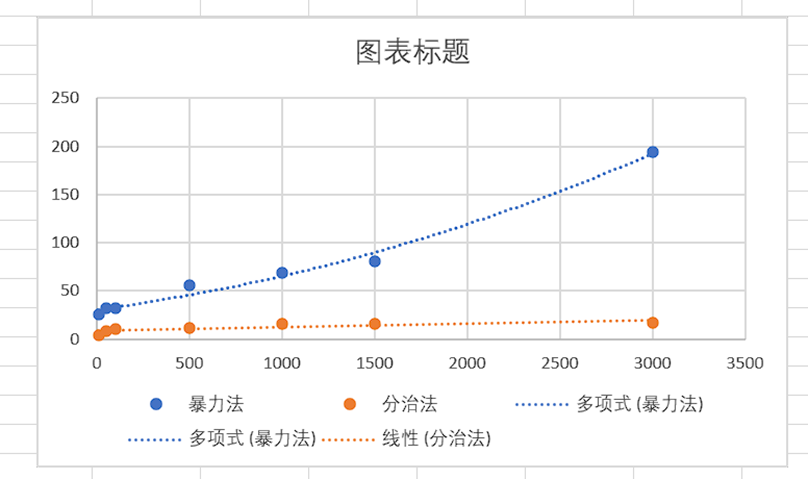


图 17 所得分析图

## 时间复杂度分析

暴力算法：

如果平面上有 n 个点，那么要考虑的点对数就为 n (n-1)/2。每一对点的距离计算需要常数时间，所以总的时间复杂度是 O (n^2)。

分治法：

分治法求解平面最近点对问题的时间复杂度可以通过递归式来分析。设 T (n) 表示求解 n 个点的最近点对的时间，那么有：

T (n) = 2T (n/2) + O (n log n)

其中，2T (n/2) 表示递归求解两个子集的时间，O (n log n) 表示在边界区域内寻找最近点对的时间。

根据主定理，可以得到 T (n) = O (n log^2 n)

# 题组二题目五：任务分配问题

## 题目描述

实现任务分配问题（随机产生不同的代价矩阵）的蛮力算法、迭代算法（匈牙利算法）、分支限界算法，并对后两者的效率进行比较和分析。

## 算法描述

1. 描述思路

蛮力算法：

蛮力算法通过暴力穷举所有可能的方案来寻找最优解。简单直观，能直接找到最优解，但是当问题规模比较大时，蛮力算法在寻找最优解时，会消耗大量的时间和资源来计算，效率非常低。

匈牙利算法：

匈牙利算法基于贪心算法，在某些情况下，可以在较短的时间内找到最优解。

先将n阶任务分配矩阵的行减去行中最小元素，列也同样操作后得到一个新的矩阵；对于该矩阵，按行圈出唯一的一个0元素后将该列的0元素划掉，如果每行每列都不存在独立0，则找到行列数0元素最少的圈起，行列的其他0元素划掉。该操作表示每个人承担不同的任务、每个任务是不同的人做。同样对于列也执行相同操作

如果统计圈出0元素的个数如果小于n阶，说明该方案不是最优解就需要调整矩阵，调整方式如下：无被圈起0元素的行打勾，重复操作。未打勾行和打勾列画线，未被线覆盖的数中找到最小值min，横竖线的交点处数值加上min、未被线覆盖的数值减去min。得到圈起0元素数=阶数时则为最优方案，找到原矩阵对应输出最优方案

分支界限法：

分支界限发基于分治思想，将问题划分为若干个子问题，通过求解子问题来接近求出原问题的解。分支界限法逐步剪枝不能达到上界，直到最后的根节点作为最小值。

1. 伪代码

蛮力算法：

Function allocate（n）

// 返回蛮力算法求得的最佳安排

// 输入：阶数n，全排列二维数组

// 输出：一维数组

// 部分排列的每个元素后面都插入i，表示第几种分配方案

For i🡨2 to n do

For j🡨0 to s.size do

s.add(j,i)

ps.add(s)// 组成全排列

// 进行分配

For i🡨0 to ps.size do

Cost🡨0

For j🡨0 to n do

// j对应第几个人第几种任务

cost += c[j][ps.get(i).get(j) - 1]

return result

匈牙利算法：

Function code（n）

// 匈牙利算法计算最优解

// 输入：n阶，n阶矩阵

// 输出：最优解

// 行列规约

For i🡨0 to n-1 do

For j🡨0 to n-1 do

a[i][j]-min

circle()or zero() // 进行查找并画线操作

if 有未被划线操作

zero（）//无行列中单独的0元素，找出最小元素并化掉

else

judge() // 判断是否符合算法条件

// 若不满足则继续变换

While num<n then

计算打勾行列和未打勾但划线

Circle（） or zero（）

最优解

End

分支界限法：

// 串联结点形成遍历方案

Function FeasibleChildren（node）

// 返回结点数组

// 输入：单个结点

// 输出：结点数组

If node.length+1>4 return

For i🡨1 to 5 do

// 扩展结点遍历数组

If feasible(node.i)

Cild🡨new Node()

Child.partialSolution.putAll(node.partialSolution)

Child.partialSolution.put(node.length.i)

Child.depth🡨node.length+1

Child.lowerBound🡨getLowerBound(chile) //下界估计值更新

nodeList.add(child)

end

end

return nodelist

function result

// 剪枝遍历得到最优方案

// 输出：最优方案

Node.lowerBound🡨getLowerBound(node) //下界值更新

Queue.add(node) // 将结点加入到队列

While queue is not empty then

Node🡨s.remove()

If node.lowerBound>=best continue //进行剪枝

For Node node : FeasibleChild(node)

If child不是叶子结点

Best🡨min(best,getLowerBound(child)) //继续寻找最优解

得出结果

## 程序运行及其结果

1. 给出程序接收的输入和输出

输入:矩阵数目 3 矩阵 4 5 9 8 6 7 1 3 5

输出：分配安排

1. 运行结果截图

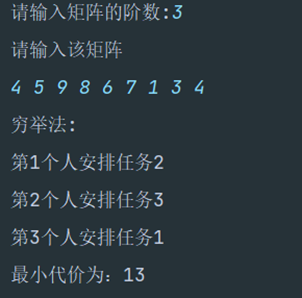


图 18 暴力算法运行情况

## 时间复杂度分析

分支界限法通过寻找所有可能方案，时间复杂度是O（n）。匈牙利算法需要进行遍历和匹配操作，遍历的时间复杂度是O(n^2),匹配操作的时间复杂度是O（n），所以，匈牙利算法的时间复杂是O（n^3）

# 题组三题目一：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。给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

## 算法描述

1. 描述思路

本道题使用回溯法，使用递归，运用了深度优先遍历的算法，依次访问每个节点，直到访问到叶子节点，然后回到根节点再此换条路经向下再次遍历

节点树如下：

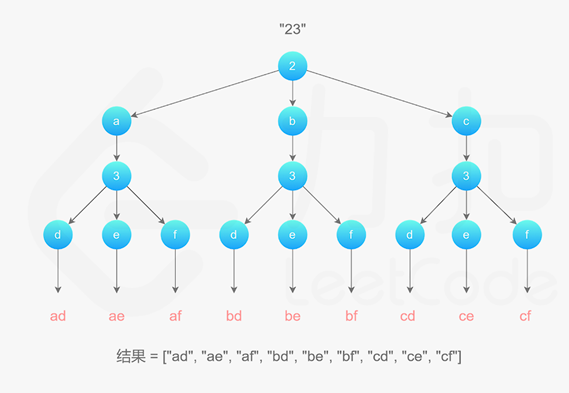


图 20 深度遍历所得生成树

递归方法如下：

f(str,letter.i+1) i!=str.length

f(str,letter,i)

end

1. 伪代码

function iterStr(str,letter,index)

// 返回字母组合

// 输入：数字构成的字符串，结果，遍历数字位置坐标

// 输出：每个叶子节点构成的结果集

if index==str.length

result.add(letter.toString)

return

strmap.get(str.charAt((index)-48)) // index下标对应的位置

for i0 to str.length do

letter.append(str.charAt(i))

iterStr(str,letter,index+1)//index下标加1

letter.deletecharAt(str.length-1)//依次删除每次组合，减少消耗

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入 | 输出 |
| 23 | ["ad","ae","af","bd","be","bf","cd","ce","cf"] |
|  | [] |
| 2 | ["a","b","c"] |

1. 运行结果截图

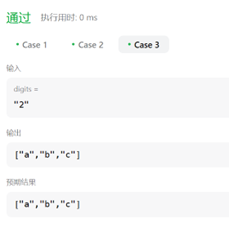


图 21电话号码的字母组合运行情况

## 时间复杂度分析

递归次数为一个满四叉树的节点个数，那么一共会递归 O(4^n)次（等比数列和），再算上加入答案时需要 O(n) 的时间，所以时间复杂度为 O(n\*4^n)

# 题组三题目二：括号生成

## 题目描述

数字 n 代表生成括号的对数，请你设计一个函数，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

本算法使用了深度优先遍历和递归来实现，采用回溯和剪枝的思想进行构建可能的括号组合。在递归过程中，如果出现右括号数量大于左括号的情况，意味着当前括号组合是非法的，因此可以进行剪枝，不再继续递归。递归结束的条件为0时结束，，产生左分支时只考虑左括号数是否大于0，产生右分支时需先判断左括号剩余数大于右括号剩余数才可以产生分支，因为不能产生“)(”这样的组合形式。

深度优先遍历如下：

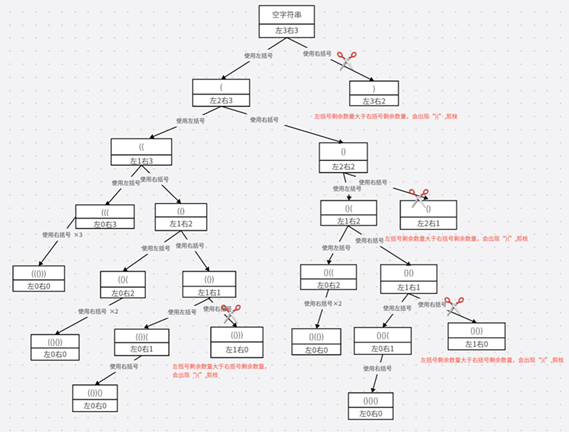


图 22 深度优先遍历和剪枝后的生成树

递归方程如下:

str i=0,j=0

f(s,i,j) f(str+“(”,i-1,j) i>0

f(str+“)”,i,j-1) j>0,i>j

1. 伪代码

Function getRound(str,i,j)

// 深度优先遍历 搜索出所有可能结果

// 输入：一种括号拼接，左括号数量，右括号数量

// 输出：结果集

If i==0 and j==0

Result.add(str)//将一种拼接加入结果集

Return

if i>j

return

if i>0

getRound(str+“（”，i-1，j）

if j>0

getRound(str+“)”，i,j-1)

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入 | 输出 |
| 3 | ["((()))","(()())","(())()","()(())","()()()"] |
| 1 | ["()"] |

1. 运行结果截图

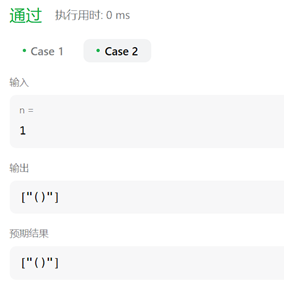
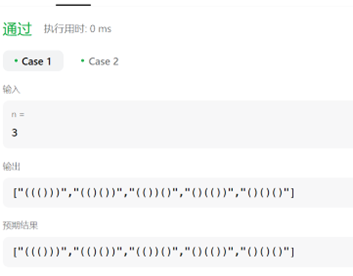


图 23 括号生成运行情况

## 时间复杂度分析

假设括号对数目为n，那么在最坏情况下，左括号和右括号都要被选择n次，因此递归的层数为2n。每一层递归中，都有两个分支，一个是选择左括号，一个是选择右括号。由于递归的层数为2n且每层有两个分支，因此总的递归调用次数为2^(2n)。而在每次递归调用时，我们只进行了常数级别的操作，所以可以将每次递归的时间复杂度视为O(1)。因此，整个算法的时间复杂度为O(2^(2n))。

总结起来，该算法的时间复杂度为O(2^(2n))，其中n表示括号对数目。

# 题组三题目三：单词搜索

## 题目描述

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

## 算法描述

1. 描述思路

这段代码的思路是基于回溯法的，它尝试了给定的二维数组中的所有可能的起点和路径，直到找到一个匹配的结果，或者确定不存在这样的结果。

主要是通过递归的方法，向元素的上下左右进行匹配，若能匹配到对应的字母，则将元素赋值为“.”表示此元素已经访问过，若没能匹配到相对应的值，则回溯元素为原来的值，表示此条路径不满足条件，跳出循环重新寻找源点和路径。

剪枝树：

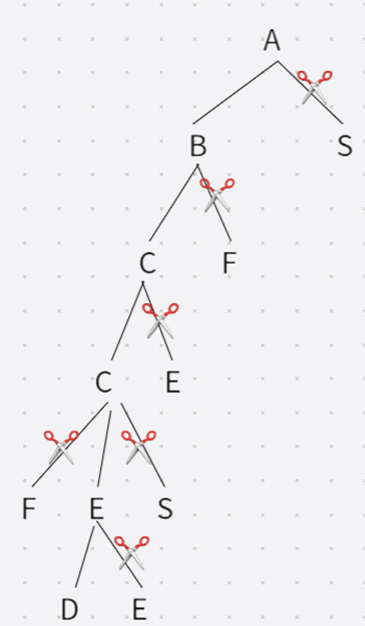


图 24 剪枝树

1. 伪代码

Function exist(board,word,i,j,index)

// 判断在二维表格board中是否能够找到word

// 输入：二维表格，指定字符，行，列，当前找到字符的下标

// 输出：二维表中是否存在目标word，true或者false

if i,j 越界 or board[i][j]！=word[index]

Return false // 越界，单词不相等

if index==word.length-1 return true // 字符查找完毕

cboard[i][j] //存储此时元素，以便之后进行回溯

board“.” // 表示此元素已经访问过

// 递归进行上下左右寻找，并返回true或者false

resexist(board,word,i+1,j,index+1) or

exist(board,word,i-1,j,index+1) or

exist(board,word,i,j+1,index+1) or

exist(board,word,i,j-1,index+1)

board[i][j]c // 变为原值，以便进行回溯

return res

## 程序运行及其结果

1. 给出程序接收的输入和输出

|  |  |
| --- | --- |
| 输入： | 输出： |
| [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]  "ABCCED" | true |
| [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]  "SEE" | true |
| [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]  "ABCB" | false |

1. 运行结果截图





图 25 单词搜索运行情况

## 时间复杂度分析

这段算法首先有两个for循环，行和列的长度分别是n和m，所有时间复杂度为O（nm）,又因为要进行递归，需要像四个方向进行，每次递归的深度为n，所以时间复杂度为O（4^n）

综上所述，总的时间复杂度为O（nm\*4^n）

# 选做题目三：超市选址

## 题目描述

设计内容：对于某一学校超市，其他各单位到其的距离不同，同时各单位人员去超市的频度也不同。请为超市选址，要求实现总体最优。

设计要求：

（1）设计该问题的核心算法；

（2）设计可视化的界面，界面中能有效显示学校超市可设立的地点和各单位的位置以及它们之间的有效路径；

（3）程序能自动计算出最优设立点，并以图示化方式演示。

## 算法描述

1. 描述思路

此题目可以理解为，一个带权有向图，选择一个源点，使得各点到此源点的总权值最小，同时，因为题目要求距离不同外，还有频率也不相同，所以，此有向图的权值为距离\*频率。此题目可以使用弗洛伊德算法。

初始化最短路径graph.a为输入的图的邻接矩阵graph.arcs，并初始化路径矩阵graph.path为记录每条最短路径的前驱顶点的矩阵。对于每个顶点k，遍历所有的顶点对（i，j），如果通过k可以使得i到j的距离变短，更新graph.a[i][j]为graph.a[i][k]+graph.a[k][j]，并更新graph.path[i][j]为graph.path[k][j]。最终输出最短路径矩阵

1. 伪代码

Function Floyd(graph,n)

// 弗洛伊德算法 得出最小路径二维数组

// 输入：带权有向图，顶点个数

// 输出：最小路径二维数组

for i1 to n do

for j1 to n do

graph.a[i][j]graph.arcs[i][j] // 复制数组

if i!=j and a[i][j]<Ingeter.MAX\_VALUE // 顶点间的距离小于最大值

// 将最小路径的一个顶点存入路径数组

graph.path[i][j]i

else

graph.path[i][j]0;

end

//从i号顶点到j号顶点只经过前k号点的最短路程

for k1 to n do

for i1 to n do

for j1 to n do

// 有更小路径

if graph.a[i][k]+graph.a[k][j]<graph.a[i][j]

graph.a[i][j]graph.a[i][k]+graph.a[k][j] //更新数组

graph.path[i][j]graph.path[k][j] //更新路径

end

end

end

## 程序运行及其结果

1. 给出程序接收的输入和输出

输入：含有权的有向图



图 26 超市选址运行情况

1. 运行结果截图

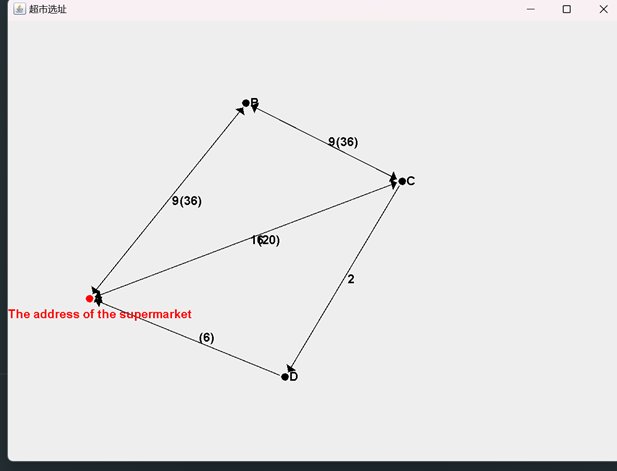


图 27 超市选址平面图

## 时间复杂度分析

弗洛伊德算法有三层for循环，有n个节点，所以此算法时间复杂度是O（n^3）。计算总权值有两个循环，所以时间复杂度是O（n^2）

综上所述，总的时间复杂度是0（n^3）

# 小结

在计算机软件专业中，算法分析与设计是一门非常重要的课程，很多问题的解决以及程序的编写都要依赖它，在软件还是面向过程的阶段，就有程序=数据结构＋算法这个公式。算法的学习对于培养一个人的逻辑思维能力是有极大帮助的，它可以培养我们养成思考分析问题以及解决问题的能力。一个算法的优劣可以用时间复杂度和空间复杂度来衡量，不同的算法可能需要不同的时间和空间，效率上也会有差别。算法可以使用自然语言、伪代码、流程图等多种方法描述，但我们一般使用伪代码描述一个算法。

本次算法课设，遇到了一些未见的题目，但是都是平时基础算法所改变的。本次题目中使用了动态规划，贪心算法，回溯，深度优先遍历。还有一些没有见过的算法，比如任务分配中的匈牙利算法，到现在为止理解的还是不深刻，还需要多熟悉多练习。算法是一个极度练习思想的一门课，虽然很难，但是很多基础算法我们必须要会，这样会让我们以后工作更加容易。

在学习过程中，我逐渐深入了解了各种算法的原理和应用场景。我开始明白，算法不仅仅是编程的工具，更是解决问题的思维模式。在解决实际问题的过程中，算法能为我提供一个清晰、高效的思考路径。但与此同时，我也体会到了算法学习的挑战性。尤其是在算法的复杂度分析部分，如何准确地评估各种算法的性能成为了我学习的重点和难点。

在学习过程中，我通过大量的实践和案例分析，逐渐掌握了如何分析算法的效率。我开始理解，算法的好坏不仅取决于其能否解决问题，更在于其解决问题的效率和可扩展性。每次当我成功地优化了一个算法的性能，我都会感受到一种由内而外的成就感。

当然，学习的过程并非一帆风顺。在学习过程中，我也遇到了很多困难和挫折。有时候我会对某个算法的原理感到困惑，有时候我会在实践中遇到难以解决的问题。但正是这些困难，促使我不断地深入思考、反复实践，从而不断进步。

对于这门课程，我有以下几点建议：首先，希望课程能更多地引入实际案例，通过案例分析来加深我们对算法原理和应用的理解；其次，希望能有更多的实践机会，通过实际操作来提升我们的算法设计和分析能力；最后，希望课程能涵盖更多种类的算法，帮助我们更全面地了解和掌握算法的世界。

经过这段时间的学习，我深感算法与分析设计的重要性和魅力。它不仅提升了我的编程技能，更培养了我解决问题的思维模式。我相信，在未来的学习和工作中，这门课程所学到的知识和技能将会是我宝贵的财富。