**暨南大学本科实验报告专用纸**

课程名称 算法分析与设计实验 成绩评定

实验项目名称 指导教师 李展 实验地点 N116

实验项目编号 5-13 实验项目类型 综合型

学生姓名 张瑞鹏 学号 2020101124

学院 信息科学技术学院 系 计算机 专业 计算机科学与技术

实验时间 2022 年 月 日 午～ 月 日 下 午

**（一）问题描述**

**（二）算法思路**（用文字简单说明）

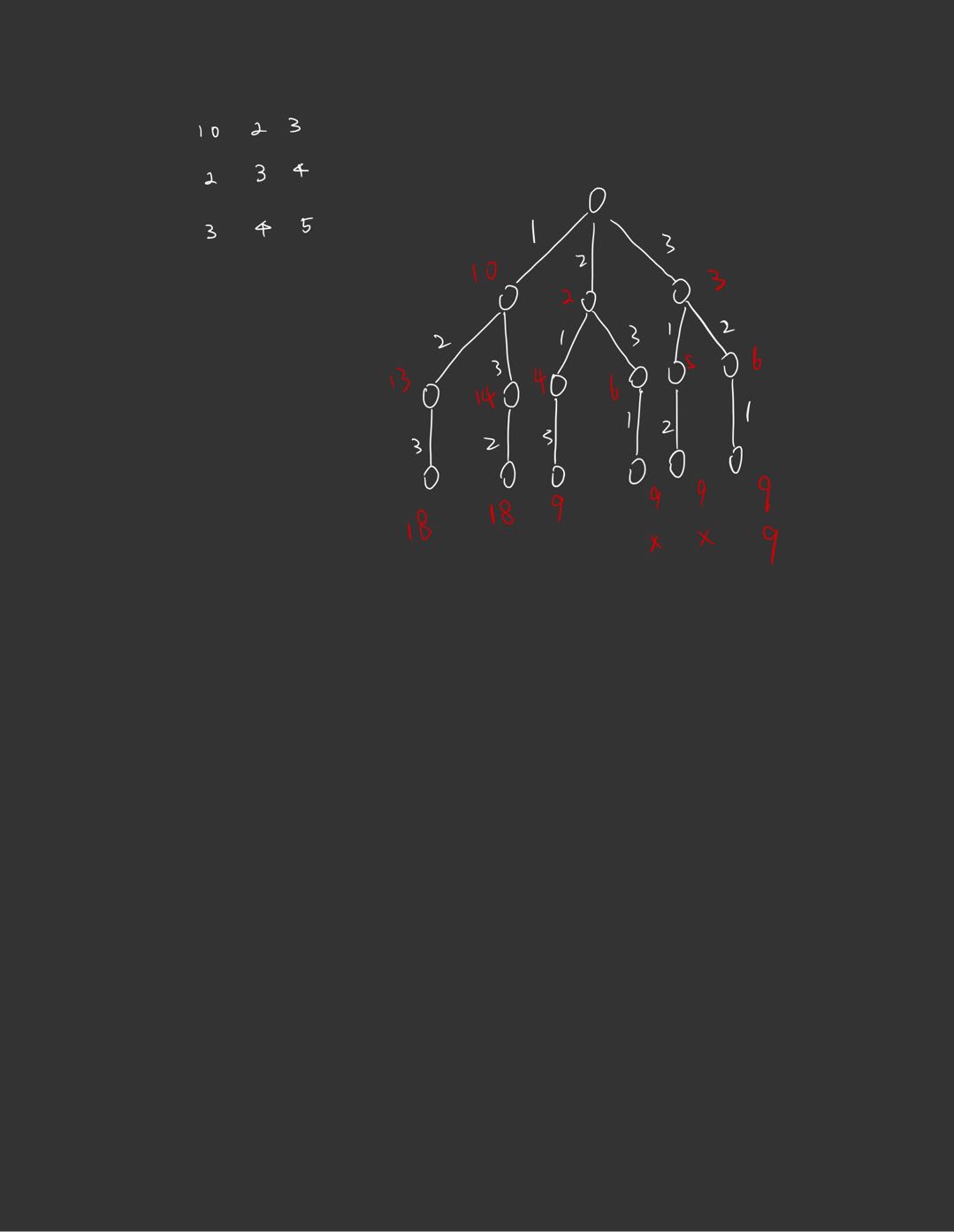
**这是一个很明显的排列问题，n个物品，n个工人，很明显就是一个工人匹配一个任务，于是采用排列树更为合适，当然子集树也是可以做的，因此本次还是采用排列树的方式，毕竟排列树没有那么常用，借此机会熟悉一下。**

**递归方式：**

**首先判断是否已经到达终点，如果已经到达终点的话直接存储结果就可以了，没有到达终点的话就通过减枝函数，判断是否符合递归的条件再进入下一层递归**

**非递归方式：**

**开始的时候t==1，深度优先，每次先做减枝，如果不满足的话顺这方向往下一个该交换的元素移动（产生排列数的核心是用两个swap函数），也就是按顺序swap下一个的意思，直到满足条件或者已经跳出去了，如果全都不满足，回退一个节点，累加变量恢复。如果是满足了条件，若还没遍历到最后一层，直接t++下一层。否则的话需要记录结果，并且退两层，因为是排列数，因此需要直接推两层。因此排列树最后两层比较特别，因此设计的时候比较复杂，换成非递归就更加复杂了，于是我采用相对比较容易的方式。**



**（三）算法实施步骤和流程**（伪代码/流程图等方式描述）

**递归：**

void traceback(int t,int &best,int\* x, int help) {

if (t == n) {

help += matrix[t][x[t]];

if (help <= best) {

best = help;

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

bestpointer[i] = x[i];

}

}

return;

}

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

swap(x, t, i);

help += matrix[t][x[t]];

if(help<=best)traceback(t+1, best, x,help);

help -= matrix[t][x[t]];

swap(x, t, i);

}

}

**非递归：**

void traceback(int t, int& best, int\* x, int help) {

int remember[10] = {0};//记录跟谁换

while (t > 0) {

remember[t] += 1;

if (remember[t] <= n) {

swap(x, t, remember[t]);

help += matrix[t][x[t]];

}

while (help >= best && remember[t] <= n) {

help -= matrix[t][x[t]];

swap(x, t, remember[t]);

remember[t]+=1;

if (remember[t] <= n) {

swap(x, t, remember[t]);

help += matrix[t][x[t]];

}

}

if (remember[t] <= n) {

if (t == n) {

best = help;

for (int i = 1; i <= n; i++) { bestpointer[i] = x[i]; }

help -= matrix[t][x[t]];

t-=1;

help -= matrix[t][x[t]];

swap(x, t, remember[t]);

}

else {

t+=1;

remember[t] = t - 1;

}

}

else {

t--;

help -= matrix[t][x[t]];

swap(x, t, remember[t]);

}

}

}

**（四）源代码**（通过了编译运行的正确程序）

# include<iostream>

using namespace std;

int n = 3;

int matrix[5][5] = {

{0,0,0,0},

{0,10,2,3},

{0,2,3,4},

{0,3,4,5}

};

int x[5] = { 0,1,2,3,4 };

int bestpointer[10] = { 0 };

void swap(int\* x, int from, int to) {

int temp = x[from];

x[from] = x[to];

x[to] = temp;

return;

}

void traceback(int t, int& best, int\* x, int help) {

int remember[10] = {0};//记录跟谁换

while (t > 0) {

remember[t] += 1;

if (remember[t] <= n) {

swap(x, t, remember[t]);

help += matrix[t][x[t]];

}

while (help >= best && remember[t] <= n) {

help -= matrix[t][x[t]];

swap(x, t, remember[t]);

remember[t]+=1;

if (remember[t] <= n) {

swap(x, t, remember[t]);

help += matrix[t][x[t]];

}

}

if (remember[t] <= n) {

if (t == n) {

best = help;

for (int i = 1; i <= n; i++) { bestpointer[i] = x[i]; }

help -= matrix[t][x[t]];

t-=1;

help -= matrix[t][x[t]];

swap(x, t, remember[t]);

}

else {

t+=1;

remember[t] = t - 1;

}

}

else {

t--;

help -= matrix[t][x[t]];

swap(x, t, remember[t]);

}

}

}

int main(void) {

int best = 100;

int help = 0;

cout << "test example:" << endl;

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

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

printf("%-3d", matrix[i][j]);

}

cout << endl;

}

traceback(1, best, x, help);

cout << "the result is:" << best << endl;

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

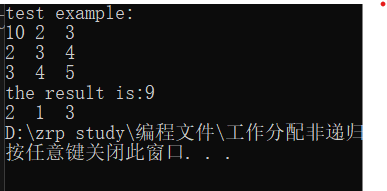
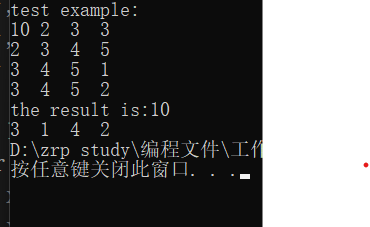
printf("%-3d", bestpointer[i]);

}

return 0;

}

**（五）测试结果**（至少有两个以上算例及程序运行结果，截图贴进实验报告）



**（六）实验总结**（至少三句话，可以写复杂度分析、遇到问题、可能的改进措施、心得体会等）

1.回溯算法的技术在于确定剪枝函数的设计。回溯法大致可以分成两种，一种是排列树，一种是子集树，而且子集树又包括了排列树，总体而言，子集树更加的一般。排列树相当于子集树又增加了一项限制，主要体现在最后一步更为复杂。

2.尝试了非递归写法，非递归的难度更加大，因为要更加熟悉模拟计算机压栈的过程，定义的t其实相当于一个栈，经常会出现++，--的操作，意思代表的就是栈的移动，整个设计思路都不太一样，难点在于需要分情况来恢复。

3.时间复杂度方面是O（n！）是一个相当大的时间复杂度。这是回溯的特点，复杂度感觉都会非常大，但是一般使用回溯的算法感觉都是比较复杂的算法，不具有最优子结构，甚至很多已经被证明是非线性时间能够解决的问题都可以解决，所以相对而言回溯其实实用性还是挺广的。

4还要多尝试非递归的算法，这种算法对整个理解更为透彻，感觉更加没有模板，很多东西都需要随机改变。

**暨南大学本科实验报告专用纸(附页)**