NCUSCC考核c语言选拔

author:杨锦松
时间：2024.10.21

超算选拔考核题

在本考核中，您需要在虚拟机环境中完成以下任务，包括安装操作系统、配置网络、编译与运行C语言代码，并撰写实验报告。

任务要求

1. 安装虚拟机：

- 在虚拟机中安装 Ubuntu 22.04 LTS 操作系统。
- 配置虚拟机的网络连接，确保可以正常联网。

2. 安装 C 语言编译器：

- 安装最新版本的 gcc（可通过 PPA 安装最新稳定版）。
- 验证编译器安装成功，并确保其正常工作。

3. 实现排序算法：

- 使用 C 语言手动实现以下排序算法：冒泡排序、基础堆排序以及斐波那契堆排序，不调用任何库函数。
- 运行测试代码，确认各排序算法的正确性。

4. 生成测试数据：

- 编写代码或脚本自动生成测试数据（随机生成浮点数或整数）。
- 测试数据应覆盖不同规模的数据集，其中必须包含至少 100 000 条数据的排序任务。

5. 编译与性能测试：

- 使用不同等级的 gcc 编译优化选项（如 -O0, -O1, -O2, -O3, -Ofast 等）对冒泡排序和堆排序代码进行编译。
- 记录各优化等级下的排序算法性能表现（如执行时间和资源占用）。

6. 数据记录与可视化：

- 收集每个编译等级的运行结果和性能数据。
- 分析算法的时间复杂度，并将其与实验数据进行对比。
- 将数据记录在 CSV 或其他格式文件中。

- 使用 Python、MATLAB 等工具绘制矢量图，展示实验结论。

7. 撰写实验报告：

- 撰写一份详细的实验报告，内容应包括：
- 实验环境的搭建过程（虚拟机安装、网络配置、gcc 安装等）。
- 冒泡排序、基础堆排序和斐波那契堆排序的实现细节。
- 测试数据的生成方法。
- 不同编译优化等级下的性能对比结果。
- 数据可视化部分（附图表）。
- 实验过程中遇到的问题及解决方案。
- 报告必须采用 LaTeX 或 Markdown 格式撰写。

提交要求

- 将完整的实验报告和源代码上传至个人 GitHub 仓库。
- 提交报告的 PDF 文件及仓库链接。

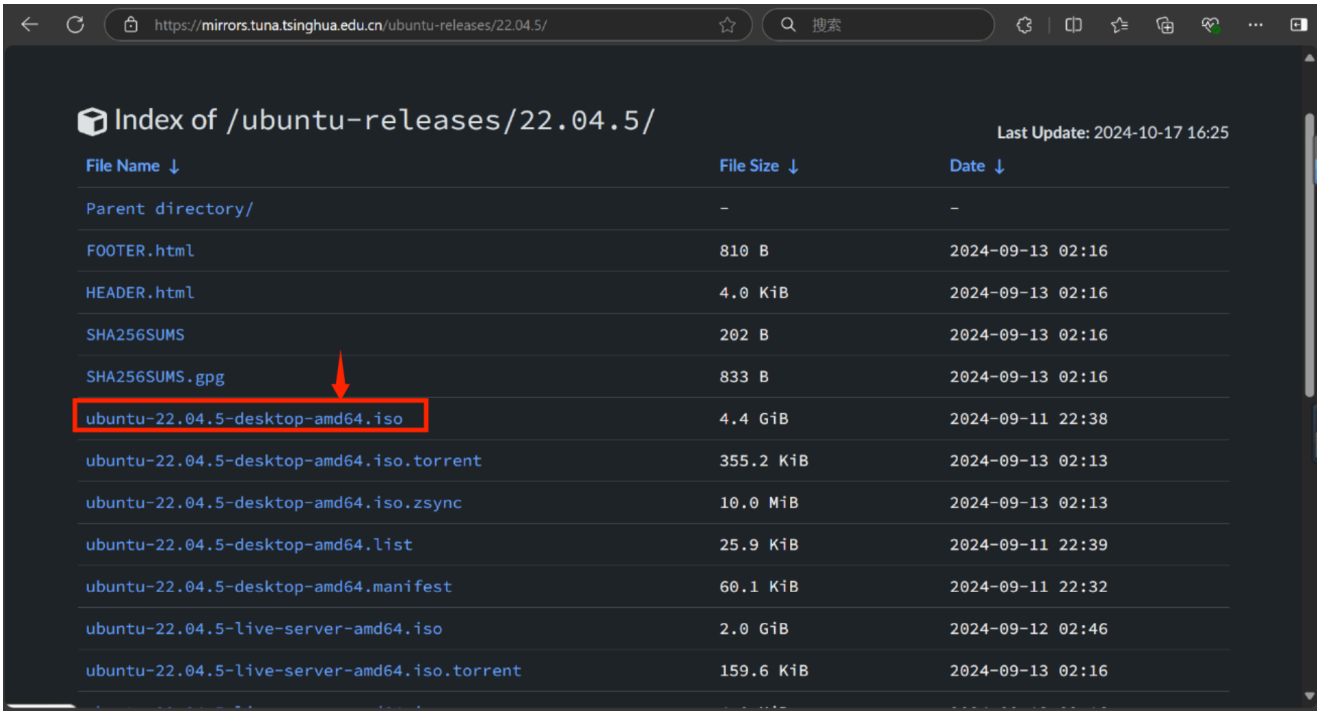
实验完成部分可以使用 AI 工具，但是报告书写部分请亲自完成！

一、实验环境的搭建

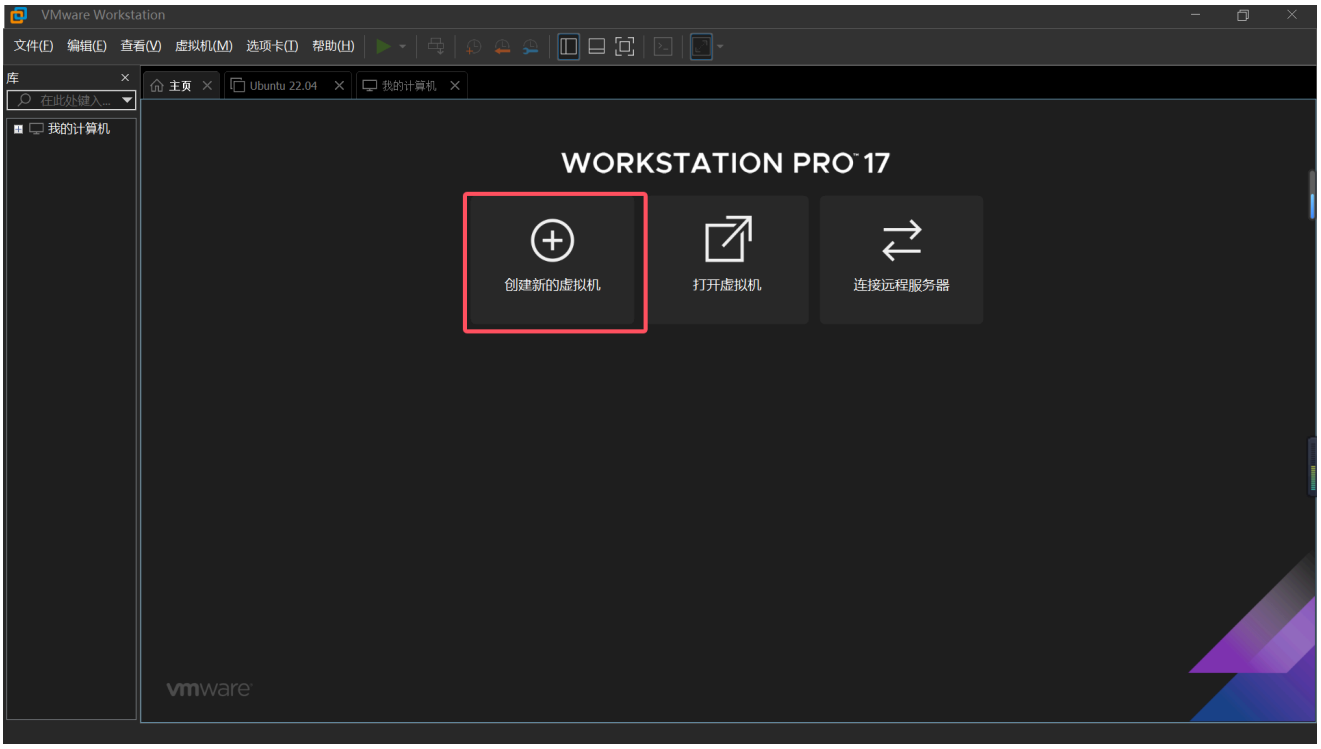
1.VMware虚拟机安装Ubuntu22.04 LTS

（1）Ubuntu镜像下载

进入官网（官网下载速度较慢，所以选择清华大学镜像站[清华大学开源软件镜像站 | Tsinghua Open Source Mirror](#)下载），找到Ubuntu-releases选择22.04.05版本。



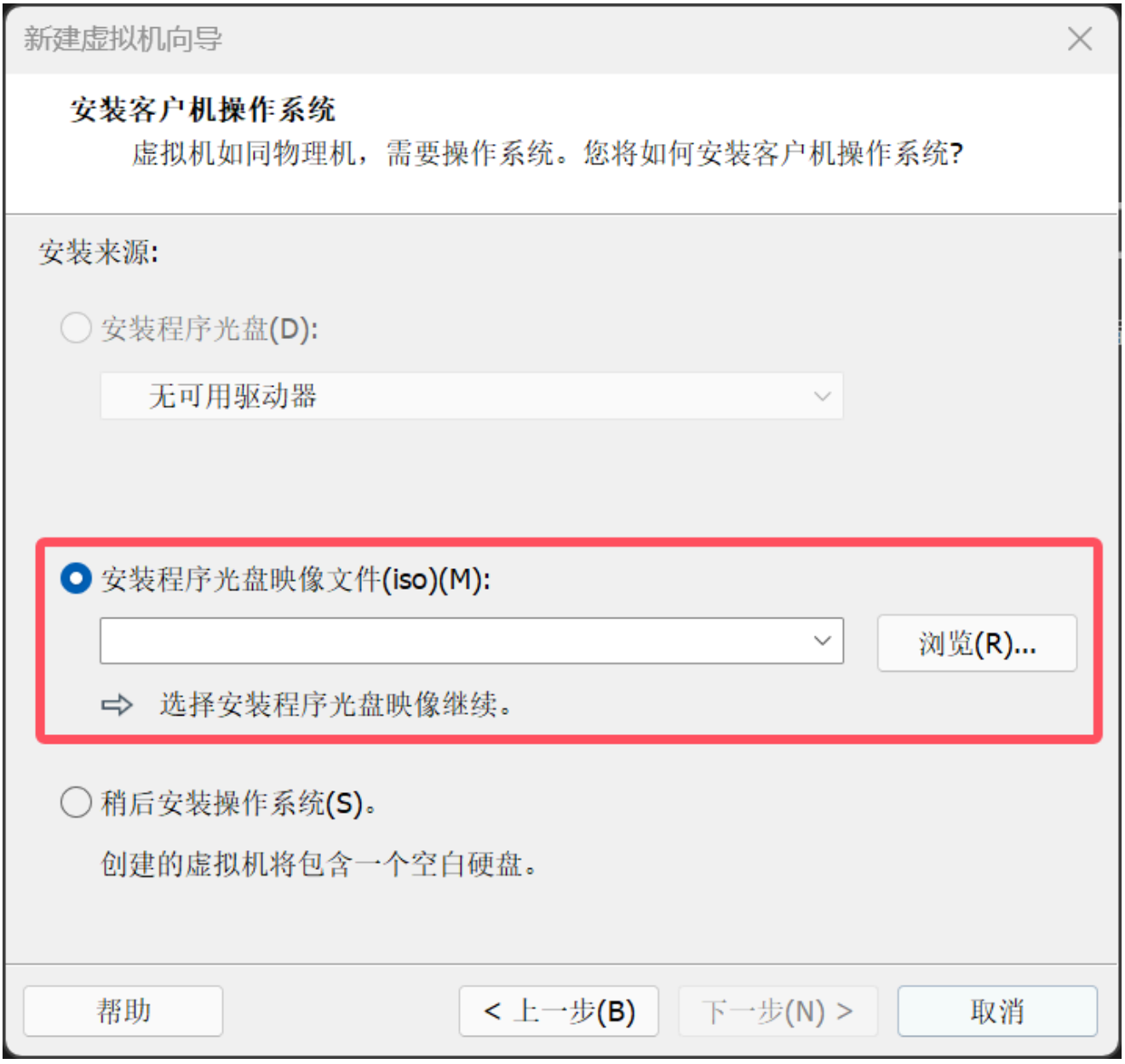
（2）虚拟机中安装Ubuntu22.04.05（由于安装时未记录过程，所以只有部分图片）


VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟机(M) 选项卡(T) 帮助(H)
库
在此处键入...
我的计算机
主页
Ubuntu 22.04
我的计算机
WORKSTATION PRO™ 17
创建新的虚拟机
打开虚拟机
连接远程服务器
vmware

-->


新建虚拟机向导
vmware
WORKSTATION
PRO™
17
欢迎使用新建虚拟机向导
您希望使用什么类型的配置?
典型(推荐)(T)
通过几个简单的步骤创建 Workstation 17.5 or later 虚拟机。
自定义(高级)(C)
创建带有 SCSI 控制器类型、虚拟磁盘类型以及与旧版 VMware 产品兼容性等高级选项的虚拟机。
帮助
< 上一步(B)
下一步(N) >
取消

-->



由于这里我已经建立了一个虚拟机了所以这里映象文件没有显示，如果正常安装，下面会显示已检测到Ubuntu 64位 22.04.05，接下来就是用户名密码等的设定。处理器为2，处理器内核为2，共4。内存选用4GB.

网络设置选用默认的NAT，可正常使用。启动虚拟机,选择`Try Or Install Ubuntu`选项,进行安装，选择最小安装设置,取消更新勾选:

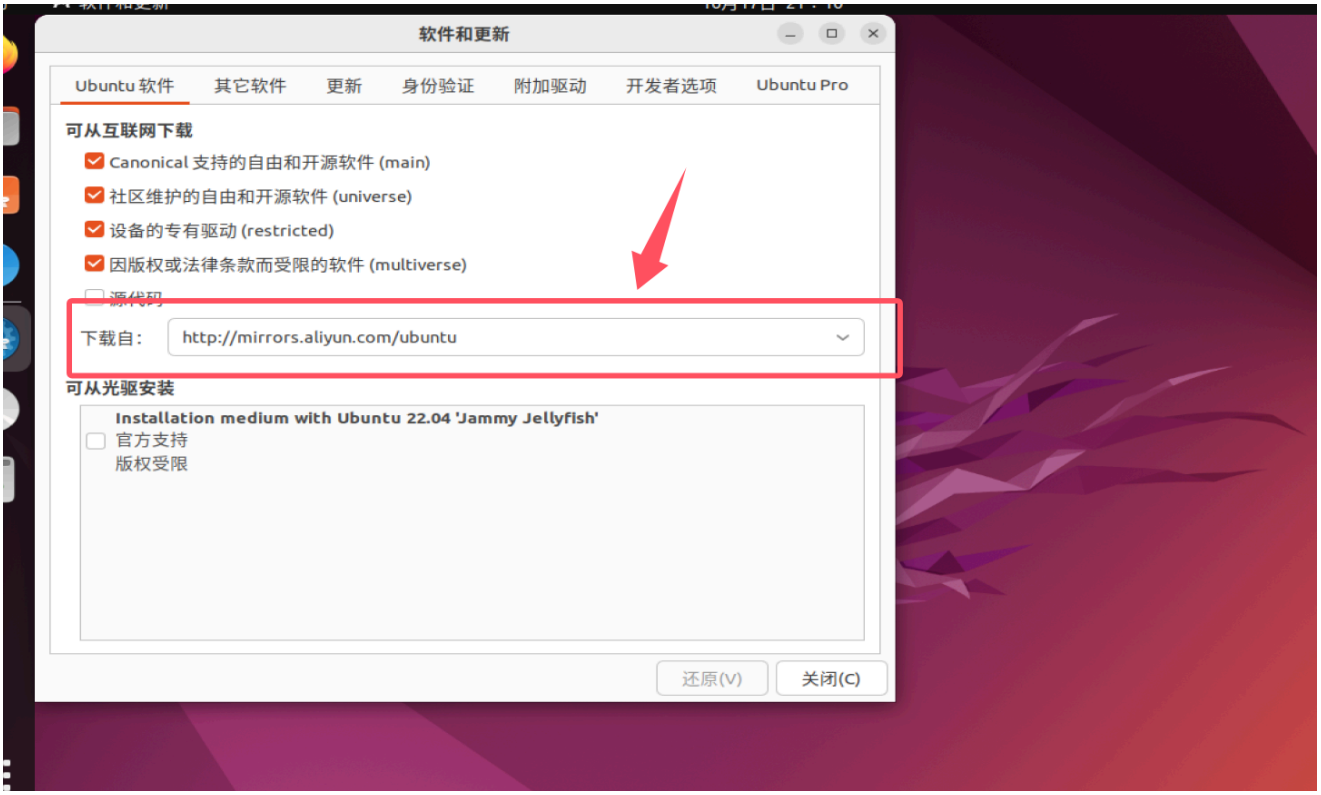
（2）镜像源改变

进入Ubuntu后需要进一步跟着配置，重启即可完成。（我个人跟着csdn进行了镜像源的改变，更换为了国内的镜像源

`sudo cp /etc/apt/sources.list/etc/apt/==sources==.list.bak`）进行下一步

-->

-->



改用阿里云的镜像源

(3)gcc的安装

终端中使用 `sudo apt install build-essential` 进行安装

输入 `gcc -v` 检测是否正常安装，出现下图说明已经完成

```
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
```

接下来安装我在乌班图中代码编译软件vim

`sudo apt-get install vim`

二、算法的实现细节 测试数据生成方法

1.冒泡排序

终端中使用 `vim maopao.c` 建立冒泡排序的项目

冒泡排序

```c
#include<stdio.h>
#include <stdlib.h>
#include <time.h>
```

```c
int main(){
    int n = 100000;
    int *arr = malloc(n * sizeof(int));
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand();//随机数设置
    }
    for(int i=1;i<n-1;i++){
            for(int j=0;j<n-i;j++){
                    if(arr[j]>arr[j+1]){
                            int temp=arr[j];
                            arr[j]=arr[j+1];
                            arr[j+1]=temp;//冒泡排序
                        }
            }
    }for(int i=0;i<n;i++){
            printf("%d ",arr[i]);
    }
    return 0;
}
```

（1）实现细节：通过上一个数对和下一个数的比较，大的数换到后面小的数换到前面，完成逐步替换，最后使得最大的数到达端点，第一个for循环表示进行次数，第二个for循环表示数组内比较的次数，由于每次比较都会使一个最大数到达端点，所以每次组内比较次数应该为总数减次数。

（2）通过 `gcc maopao.c -o maopao` 生成可执行程序

再通过 `./maopao` 进行运行测试

2.基础堆排序

基础堆排序

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// 交换两个整数的值
#define EXCHANGE(a, b) { int temp = (a); (a) = (b); (b) = temp; }

void maxHeapify( int num[], int start, int end )
{
    //建立父节点指标和子节点指标
    int dad = start;
    int son = dad * 2 + 1;
    while( son <= end )
//若子节点指标在范围内才做比较
    {
        if( son + 1 <= end && num[son] < num[son + 1] )
        //先比较两个子节点大小，选择最大的
        {
```

```
            son++;
        }
        if( num[dad] > num[son] )   //如果父节点大於子节点代表调整完毕，直接跳出函数
        {
            return;
        }
        else                        //否则交换父子内容再继续子节点和孙节点比较
        {
            EXCHANGE( num[dad], num[son] );
            dad = son;
            son = dad * 2 + 1;
        }
    }
}
void heapSort( int num[], int count )
{
    int i;

    for( i = count / 2 - 1; i >= 0; i-- )
    //1.最大堆调整 初始化，i从最後一个父节点开始调整
    {
        maxHeapify( num, i, count - 1 );
    }
    for( i = count - 1; i > 0; i-- )
    {
        EXCHANGE( num[0], num[i] );  //2.堆排序 将第一个元素和已排好元素前一位做交换
        maxHeapify( num, 0, i - 1 );     //3.创建最大堆，重新调整父节点和子节点
    }
}
    int main() {
    int n = 10000; // 数组大小
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    srand((unsigned int)time(NULL)
    for (int i = 0; i < n; i++) {
        arr[i] = rand() ; //生成随机数
    }
    // 执行堆排序
    heapSort(arr, n);
    // 打印排序后的数组
    printf("\nSorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
        if ((i + 1) % 100 == 0) printf("\n");
    }
```

```
65  }
```

（1）基础堆排序类似二叉树的结构，先将数据从上到下按顺序排序，转换为二叉树，紧接着使其变为最大堆（子节点小于父节点），取出堆顶元素放在最末尾，重新调整剩余数据，再重复上述过程。最终按大小排序。
（2）同冒泡排序的方式`gcc jichudui.c -o jichudui`及`./jichudui`。
3.斐波那契数列

斐波那契堆

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXDEGREE 20


// 函数原型声明
Node* createNode(int key);
FibHeap* initFibHeap();
void insertNode(FibHeap *heap, Node *x);
Node* extractMin(FibHeap *heap);
void consolidate(FibHeap *heap);
void link(FibHeap *heap, Node *y, Node *x);
void decreaseKey(FibHeap *heap, Node *x, int k);
void cut(FibHeap *heap, Node *x, Node *y);
void cascadingCut(FibHeap *heap, Node *y);
void destroyFibHeap(FibHeap *heap);
void fibHeapSort(FibHeap *heap);

typedef struct Node {
    int key;
    struct Node *parent, *child, *left, *right;
    int degree, mark;
} Node;

typedef struct {
    Node *min;
    int n;
} FibHeap;

Node* createNode(int key) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->key = key;
    node->parent = node->child = NULL;
    node->left = node->right = node;
    node->degree = 0;
    node->mark = 0;
```

```
    return node;
}

FibHeap* initFibHeap() {
    FibHeap *heap = (FibHeap *)malloc(sizeof(FibHeap));
    heap->min = NULL;
    heap->n = 0;
    return heap;
}

void link(FibHeap *heap, Node *y, Node *x) {
    if (y->right != y) {
        y->right->left = y->left;
        y->left->right = y->right;
    }
    y->left = y->right = y;
    y->parent = x;
    x->degree++;
    y->mark = 0;
}

void insertNode(FibHeap *heap, Node *x) {
    x->left = x->right = x;
    if (heap->min == NULL) {
        heap->min = x;
    } else {
        x->right = heap->min->right;
        x->left = heap->min;
        heap->min->right->left = x;
        heap->min->right = x;
    }
    heap->n++;
}

Node* extractMin(FibHeap *heap) {
    Node *z = heap->min;
    if (z != NULL) {
        if (z->child != NULL) {
            Node *child = z->child;
            while (child != z) {
                child->parent = NULL;
                child = child->right;
            }
            insertNode(heap, child);
        }
        z->left->right = z->right;
        z->right->left = z->left;
        if (z == z->right) {
            heap->min = NULL;
        } else {
```

```
            heap->min = z->right;
            consolidate(heap);
        }
        heap->n--;
    }
    return z;
}

void mergeHeaps(FibHeap *heap1, FibHeap *heap2) {
    if (heap1->min != NULL && heap2->min != NULL) {
        if (heap1->min->key > heap2->min->key) {
            Node *temp = heap1->min;
            heap1->min = heap2->min;
            heap2->min = temp;
        }
    }
    heap1->min->right->left = heap2->min->left;
    heap2->min->left->right = heap1->min->right;
    heap1->min->right = heap2->min;
    heap2->min->left = heap1->min;
    heap1->n += heap2->n;
    free(heap2);
}

void consolidate(FibHeap *heap) {
    int A[MAXDEGREE], degree, i;
    Node *x, *y, *w, *z;
    Node *a[2 * MAXDEGREE + 1];

    for (i = 0; i <= MAXDEGREE; i++) {
        a[i] = NULL;
    }

    x = heap->min;
    while (x != x->right) {
        w = x->right;
        degree = x->degree;
        while (a[degree] != NULL) {
            y = a[degree];
            if (x->key > y->key) {
                z = x;
                x = y;
                y = z;
            }
            link(heap, y, x);
            a[degree] = NULL;
            degree++;
        }
        a[degree] = x;
        x = w;
```

```
    }
    heap->min = NULL;
    for (i = 0; i <= MAXDEGREE; i++) {
        if (a[i] != NULL) {
            x = a[i];
            x->left = x->right = x;
            if (heap->min == NULL) {
                heap->min = x;
            } else {
                x->right = heap->min->right;
                x->left = heap->min;
                heap->min->right->left = x;
                heap->min->right = x;
            }
        }
    }
}

void decreaseKey(FibHeap *heap, Node *x, int k) {
    if (k > x->key) {
        return;
    }
    x->key = k;
    Node *p = x->parent;
    if (p != NULL && x->key < p->key) {
        cut(heap, x, p);
        cascadingCut(heap, p);
    }
    if (x->key < heap->min->key) {
        heap->min = x;
    }
}

void cut(FibHeap *heap, Node *x, Node *y) {
    if (y->child == x) {
        y->child = x->right;
    }
    x->left->right = x->right;
    x->right->left = x->left;
    x->left = x->right = x;
    x->parent = NULL;
    x->mark = 0;
    insertNode(heap, x);
}

void cascadingCut(FibHeap *heap, Node *y) {
    Node *z;
    while (y != NULL && y->parent != NULL && y->mark == 0) {
        z = y->parent;
        if (y->degree == z->degree - 1 && z->child != NULL) {
```

```
            y->mark = 1;
        } else {
            cut(heap, y, z);
            cascadingCut(heap, z);
        }
        y = z;
    }
}

void destroyFibHeap(FibHeap *heap) {
    Node *x, *y;
    x = heap->min;
    while (x != NULL) {
        y = x->right;
        free(x);
        x = y;
    }
    free(heap);
}

void fibHeapSort(FibHeap *heap) {
    Node *z;
    while ((z = extractMin(heap)) != NULL) {
        printf("%d ", z->key);
        free(z);
    }
}

int main() {
    FibHeap *heap = initFibHeap();
    int n = 10000;
    int *arr = (int *)malloc(n * sizeof(int));

    srand((unsigned)time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;
        Node *node = createNode(arr[i]);
        insertNode(heap, node);
    }

    printf("Sorted elements: ");
    fibHeapSort(heap);
    printf("\n");

    free(arr);
    destroyFibHeap(heap);
    return 0;
}
```

（1）提高运行时间，更快找到最小元素，基于二进制堆但是较为松散，可以是任意树，每个节点都可以有更多子节点。删除最小元素前，先把最小元素的子节点移出，为了使节点数和树的数量少，合并树（将根数相同的合并 ）由于看了几遍视频也是粗略的懂了点，代码为ai生成

（2）同上述两种方式进行测试

三.数据记录与可视化

1.分析

（1）冒泡排序：最优的情况下是在第一次轮中没有交换，直接结束

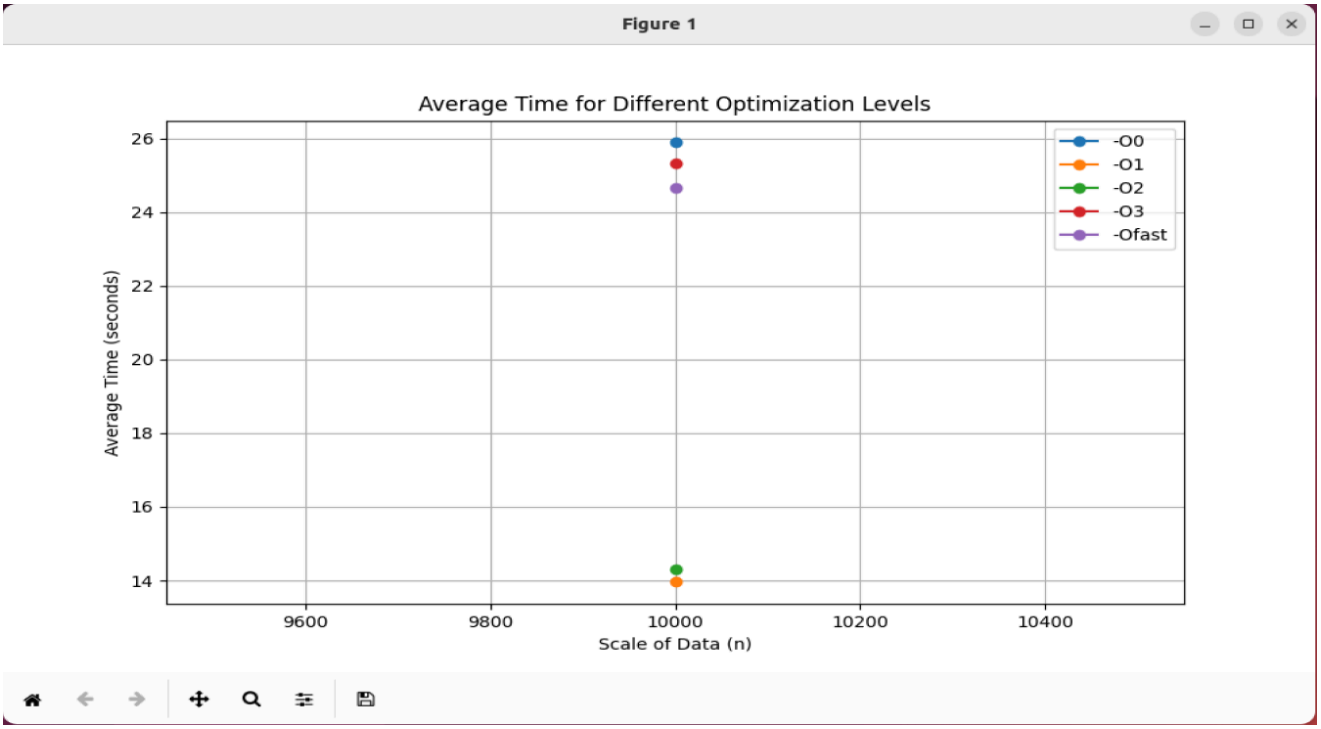
如果多轮，时间复杂度O（n^2），由于每轮都需要进行交换，所以会耗费大量时间

（2）基础堆排序：通过每次交换比较，将最大数提出，通过递归实现的时间复杂度通常为O（logn）

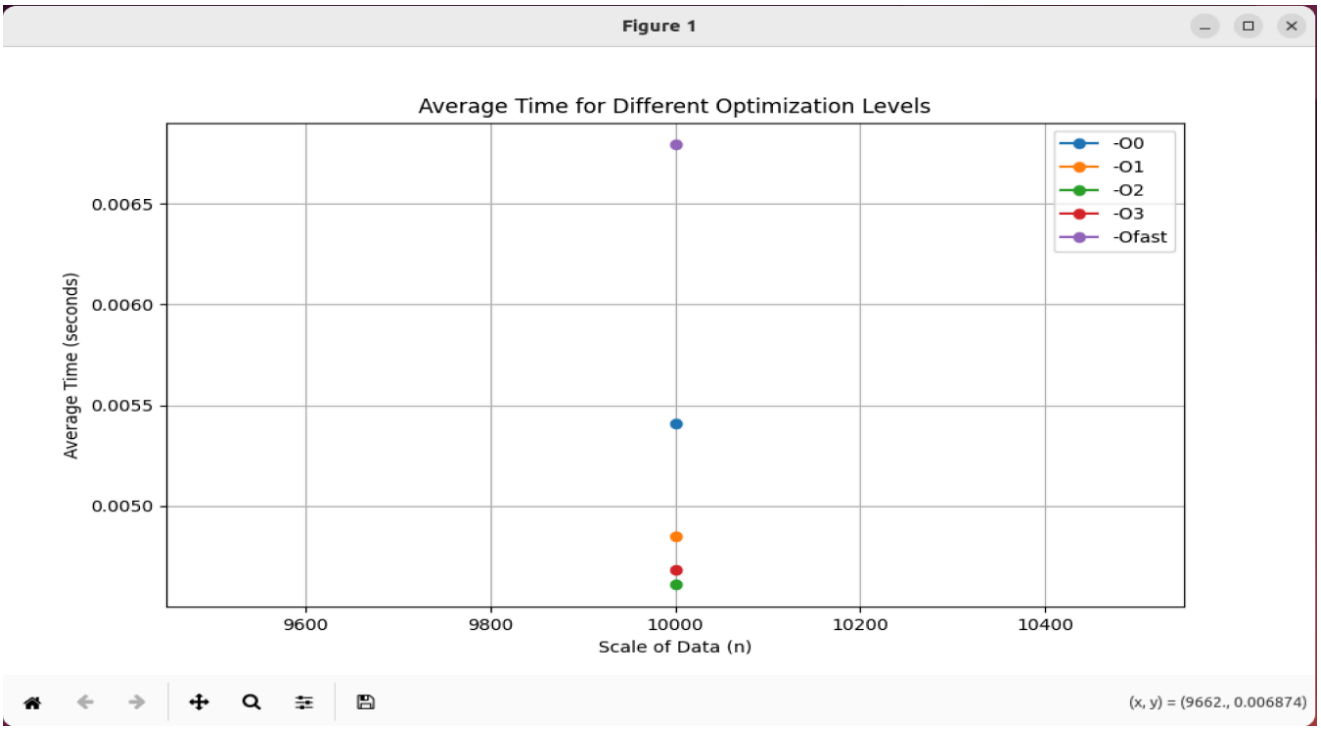
（3）斐波那契堆排序：找到最小元素，删除最小元素，合并子堆时间复杂度O（1）较短，插入、减小、删除等O（logn），总体平均为O（nlogn）

2.时间测试

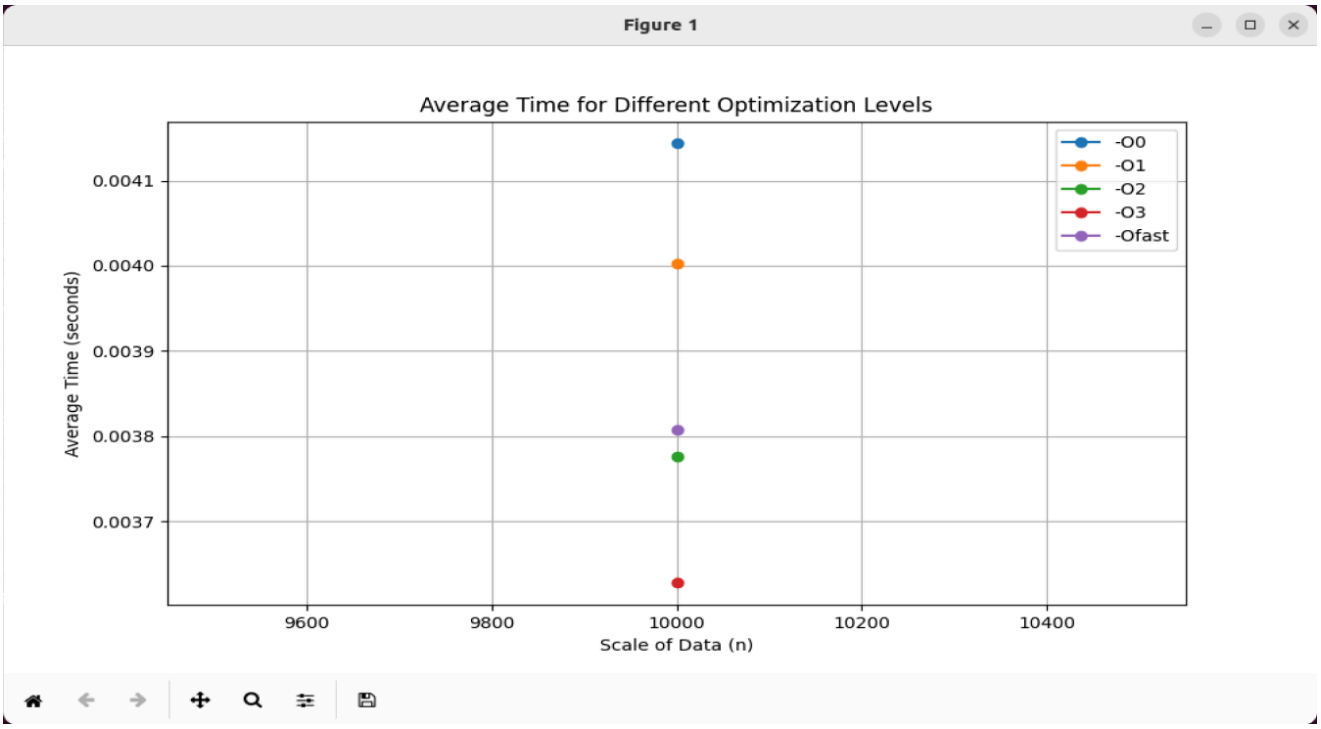
（1）冒泡排序



（2）基础堆排序



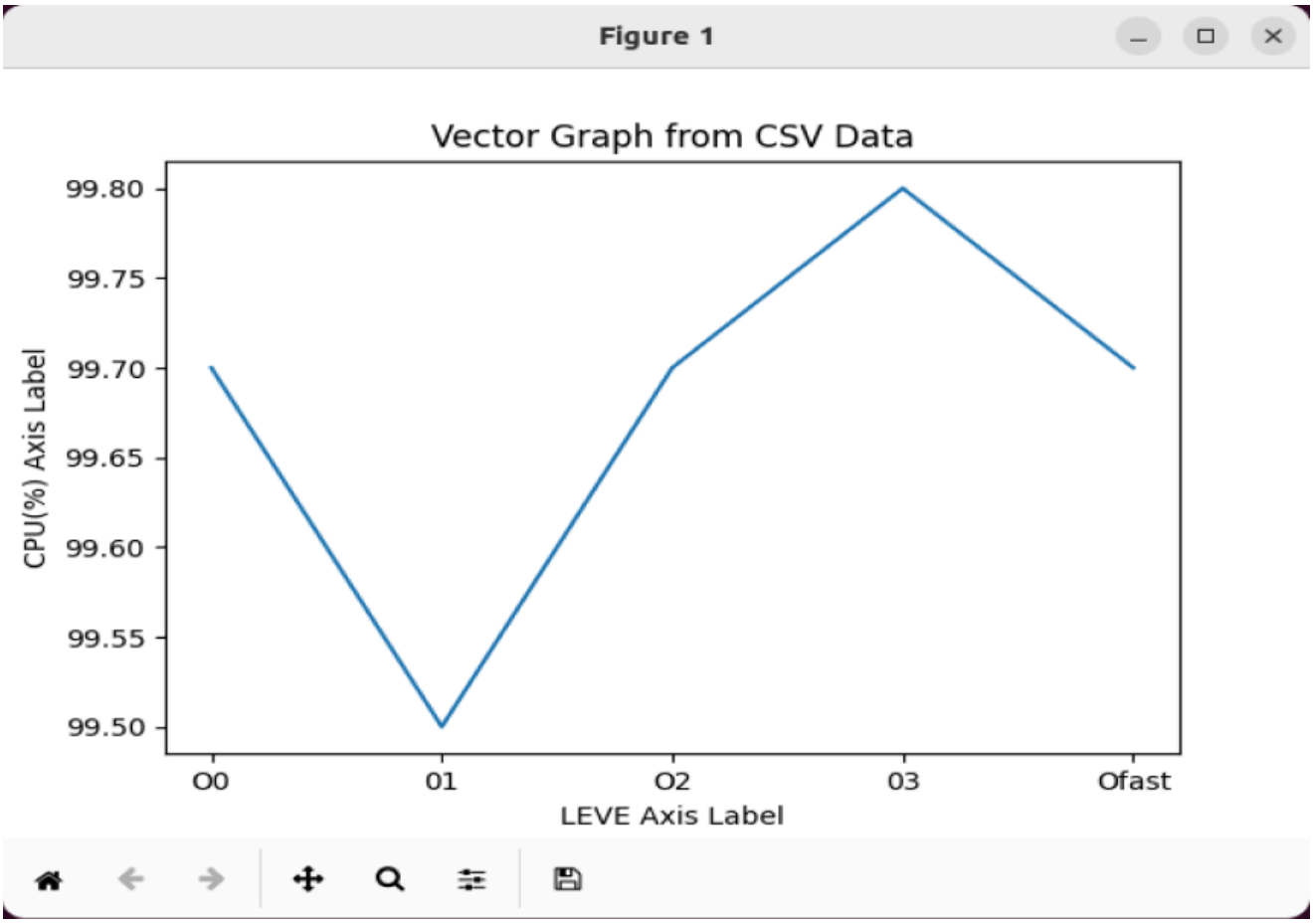
（3）斐波那契堆排序



通过对比可以发现，冒泡排序的时间是最长的（等结果等的时间也是最长的），其余两种排序耗费时间都较短

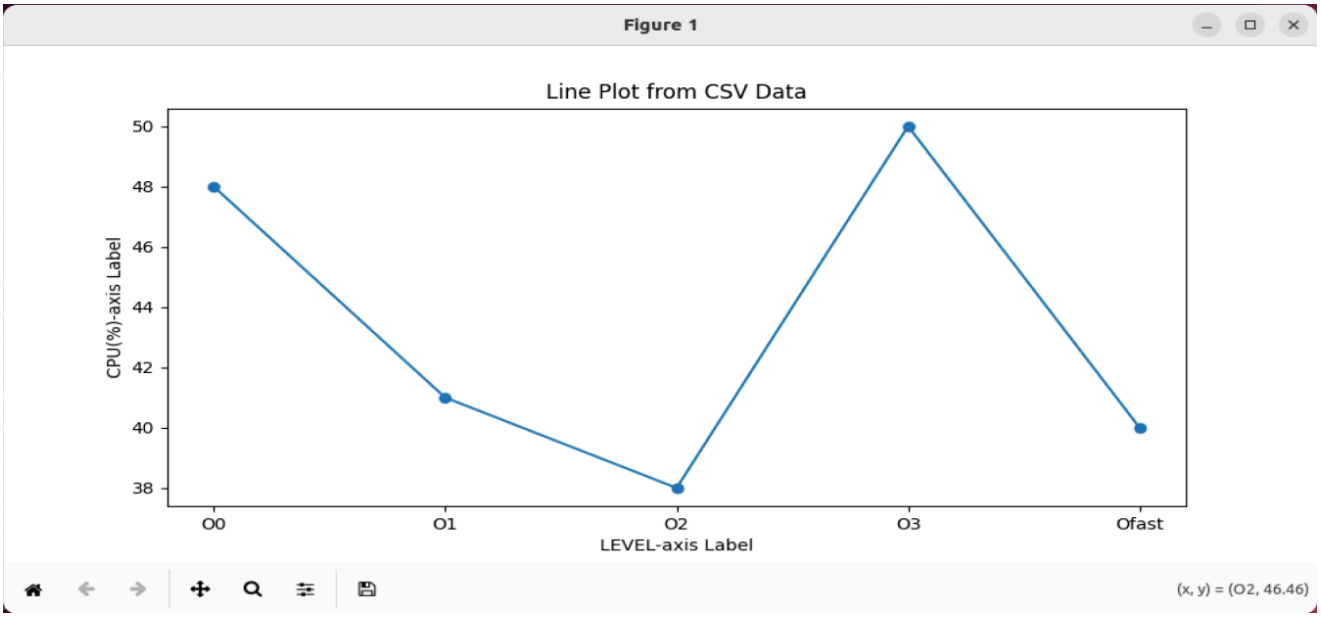
3.资源占用

（1）冒泡排序



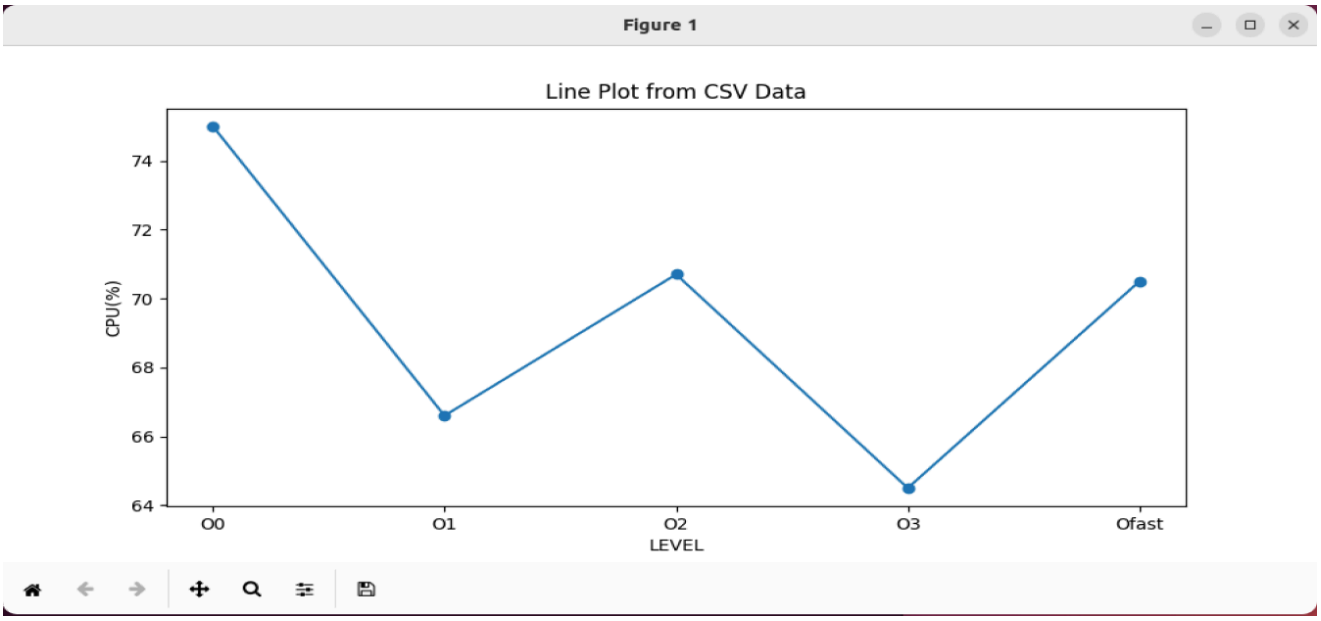
图中可见冒泡排序的不管是时间还是cpu使用率都很高，显示出冒泡排序是一种十分耗费的排序，占用许多资源，跟每次循环都需要一次次交换有关

（2）基础堆排序



相比冒泡排序，可以发现cpu占比小很多，通过图中可以看见，在不同等级下，由于优化的不同，cpu占比也不稳定

（3）斐波那契堆排序



根据图中可以得出基本趋于稳定，并并没有很大波动

四、实验中遇到的问题及解决方案

由于是第一次接触，难免会遇到许多问题
1.最开始遇到的问题无疑是虚拟机的配置，按照教程下载的乌班图一直无法成功运行，再多次重新配置无果后，将虚拟机vmware删除重新下载解决
2.接下来就是代码的编写，第一次听说斐波那契堆（刚开始还以为是斐波那契），找了许多资料发现都十分抽象感谢b站上的视频让我粗浅的了解
【Fibonacci heaps 斐波那契堆】 https://www.bilibili.com/video/BV1x54y1w7Y6/?share_source=copy_web&vd_source=2261c9576a4b4cf4d5bae89613c8ab5a
3.接下来本来是希望kimi能写个脚本帮我一次性测试所有不同编译等级下的时间分析和资源占用情况发现基本没几次运行成功的，就乖乖一个个记录了。

第一次写实验报告，也是第一次接触虚拟机和linux系统，过程中遇到许多问题，也感谢超算队提供的这个机会去促使自己学到更多东西

特别鸣谢
kimi 牢鸽 超算队