

NCUSCC选拔考核-C语言项目实验报告

严李阳

考核要求

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。

实验报告正式部分

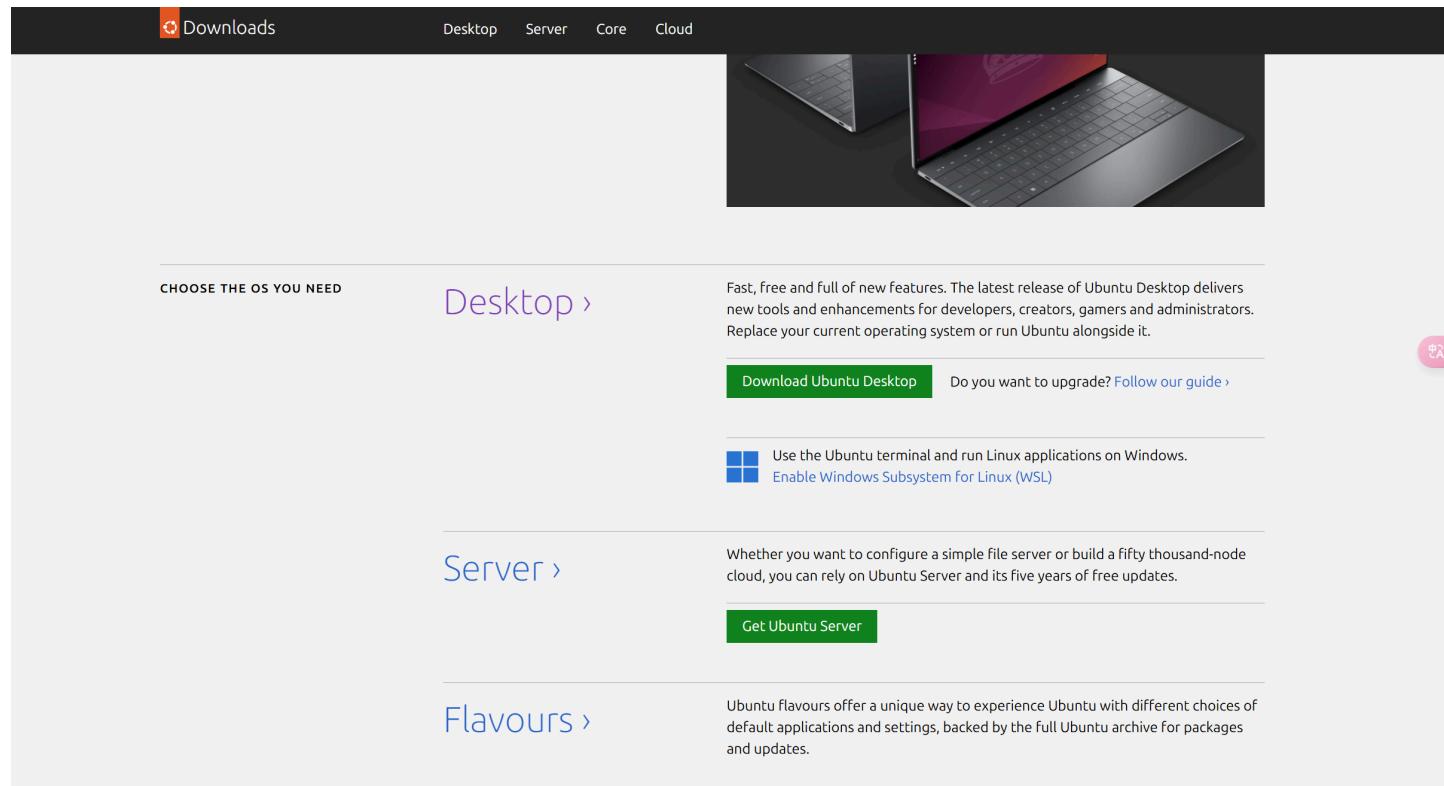
一、实验环境的搭建

实验目的：熟悉虚拟机的安装流程和Linux系统的基础操作，提高自身在不同编程环境下的适应力，并学会在不同操作系统中创造适合自己的编程环境，同时试错一些不必要的操作失误问题。

实验流程：

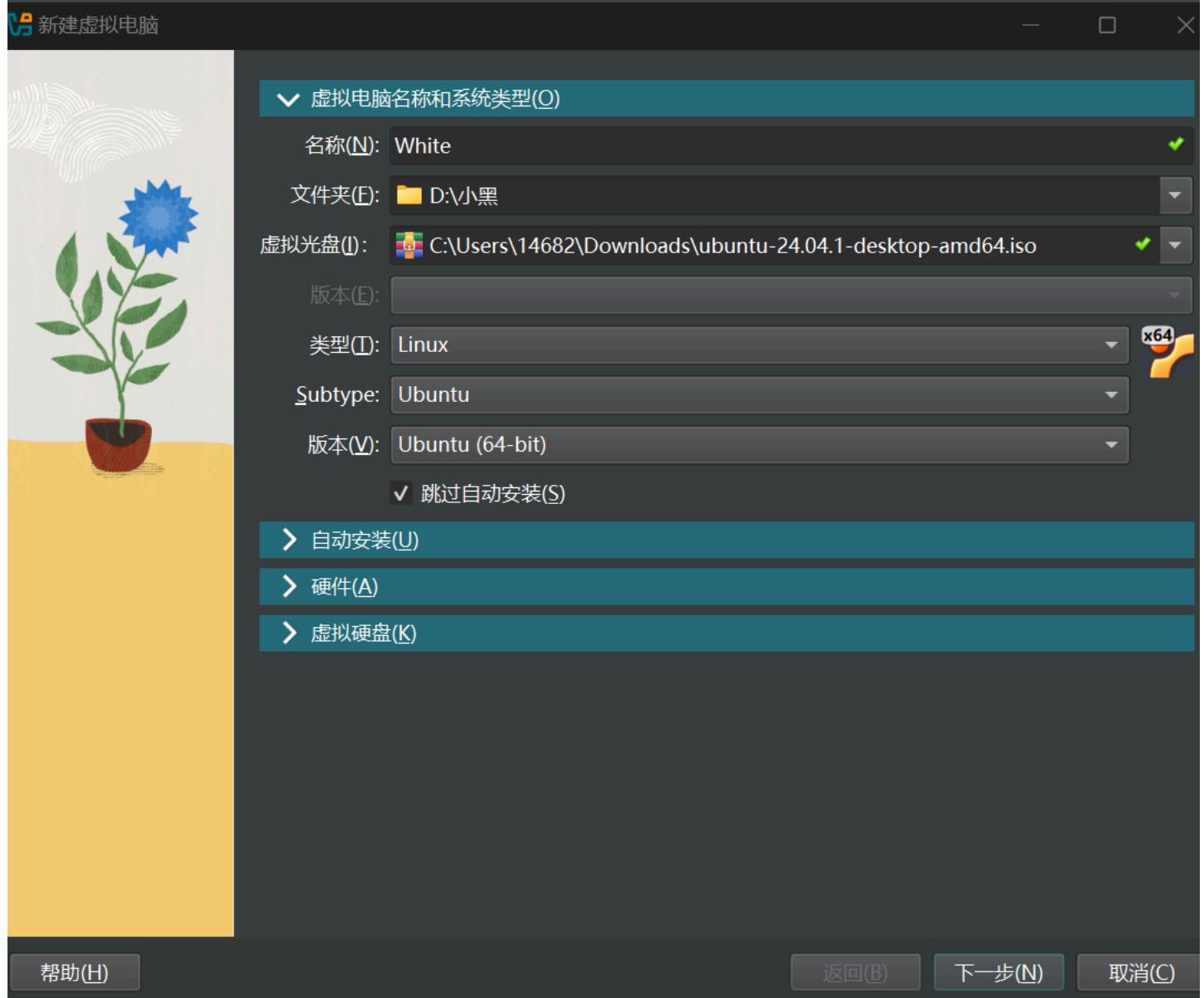
1. 使用Oracle VirtualBox创建虚拟机，并在虚拟机上安装Ubuntu 22.04 LTS操作系统

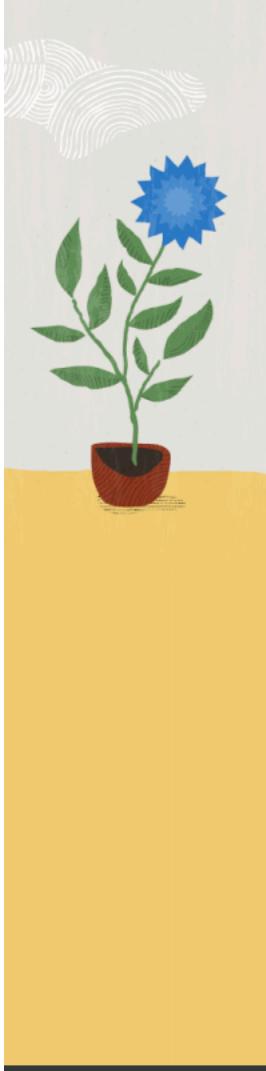
(1)前往[Ubuntu官方网站](#)获取Ubuntu-22.04-desktop的ISO文件，桌面板具有一定的图形界面，适合新手操作



The screenshot shows the Ubuntu website's main navigation bar with 'Downloads' selected. Below it, there are three main categories: Desktop, Server, and Core. The 'Desktop' section is highlighted with a large image of two laptops. A sub-section titled 'CHOOSE THE OS YOU NEED' offers the 'Desktop' edition, described as fast, free, and full of new features, suitable for developers, creators, gamers, and administrators. It includes a 'Download Ubuntu Desktop' button and a link to upgrade via WSL. The 'Server' section is also visible, mentioning its use for file servers and cloud configurations. The 'Flavours' section highlights the variety of Ubuntu distributions available.

(2)新建虚拟机，适当增加内存、处理器、硬盘分配，以便匹配后续性能和存储需求，勾选跳过自动安装





> 虚拟电脑名称和系统类型(O)

> 自动安装(U)

▼ 硬件(A)

内存大小(M): 4096 MB

4 MB

16384 MB

处理器(P): 4

1 CPU

32 CPU

 启用 EFI (只针对某些操作系统)

> 虚拟硬盘(K)

帮助(H)

返回(B)

下一步(N)

取消(C)



(3)系统安装，手动重启

安装完成



Ubuntu 24.04.1 LTS 已经安装并准备好使用了

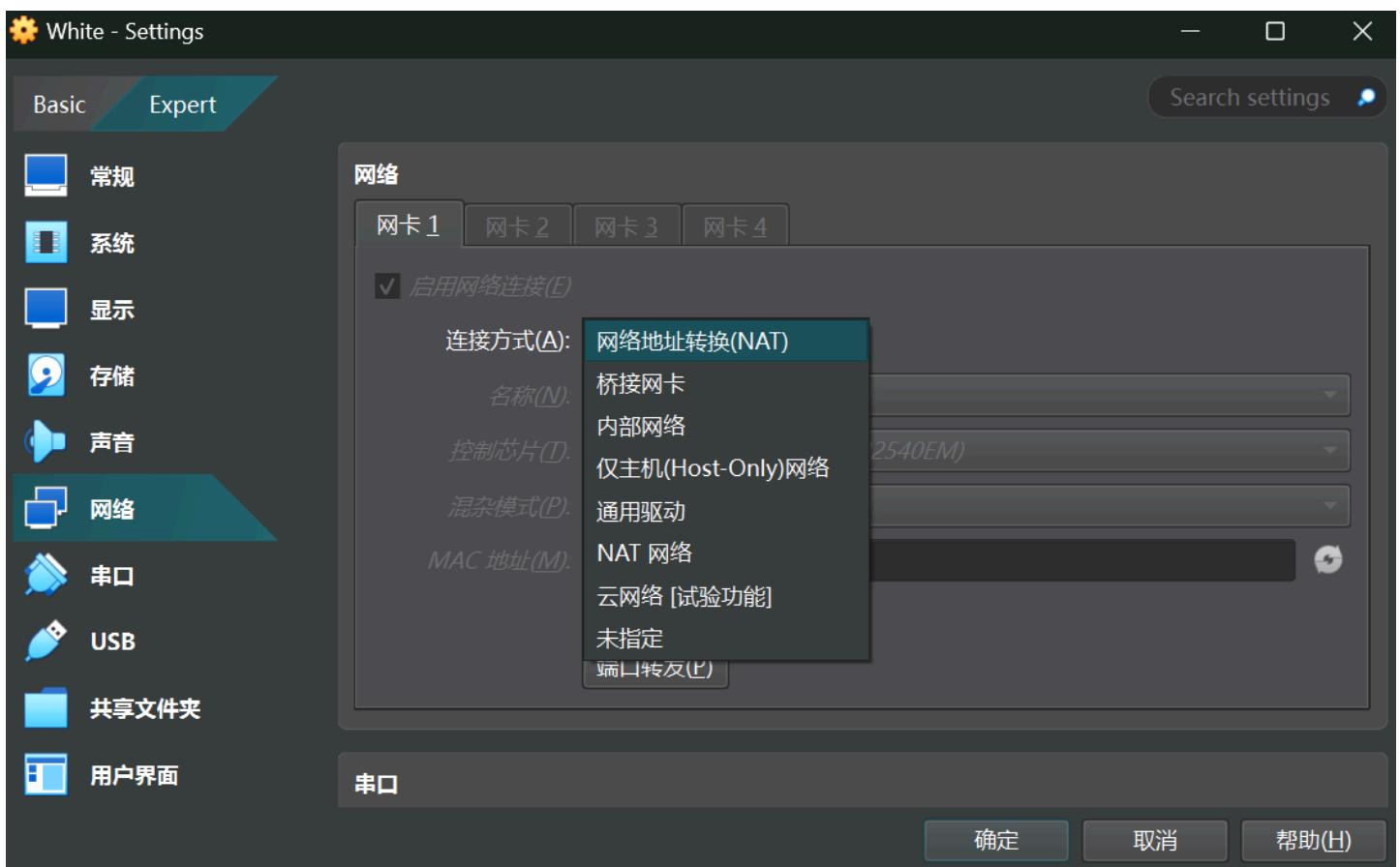
重新启动以完成安装，也可以继续测试。
您做出的任何更改都不会被保存。

继续试用

立即重启

2. 网络配置

(1) 连接到互联网安装提示界面选择我现在不想连接到互联网，方便后续调整网络连接方式



3.gcc安装

(1)右键鼠标打开终端



(2)输入指令更新并下载

```
sudo apt update  
sudo apt install build-essential
```

(3)检查版本，确认是否安装

```
gcc --version
```

发现问题：

- 1.同时安装多个相同系统的虚拟机，删除文件时会删除共有文件，导致原机映像文件受损
- 2.第一次安装系统后可能会陷入黑屏无响应情况

解决方案：

- 1.如果处理时间较短可以考虑用Data Numen大海捞针；如果你已经像我一样使用了重启大法，导致硬盘数据碎片丢失，那就只能选择重装系统，调用备份数据了
- 2.第一次装好系统黑屏的话，重启虚拟机一般就能解决问题，如果重启了三四次仍然黑屏或进入选择系

统界面，那大概率需要重装系统（以我现在的技术力确实无法对点解决缺失文件问题）

GNU GRUB 2.12 版

```
*Ubuntu
Advanced options for Ubuntu
Memory test (memtest86+x64.bin)
Memory test (memtest86+x64.bin, serial console)
```

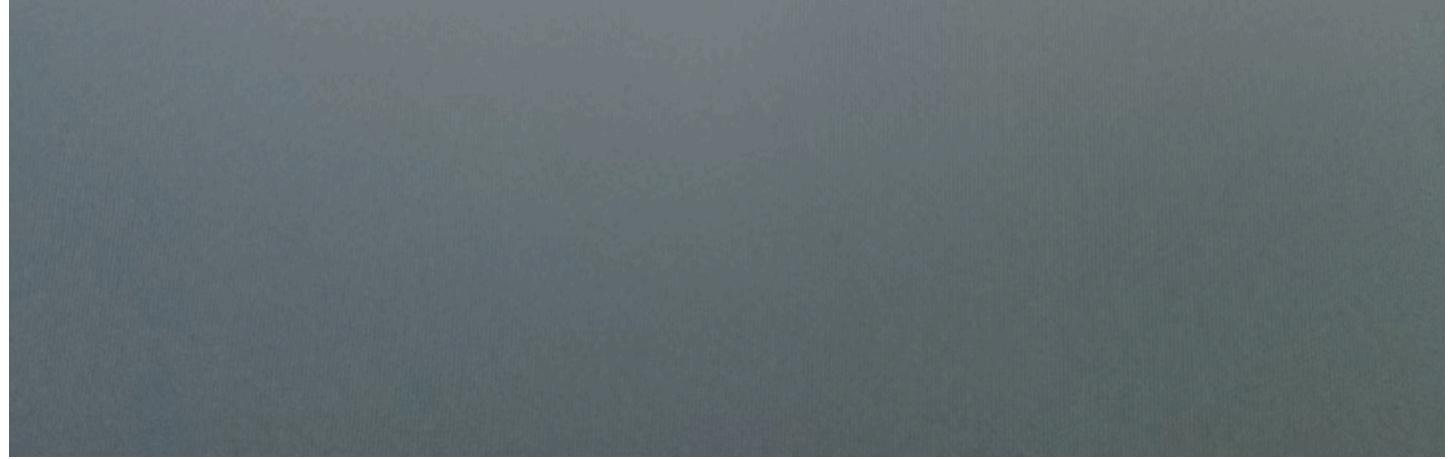
使用↑键和↓键选择要高亮的启动项。
按enter引导选定的操作系统，按e在引导之前编辑命令，按c进入命令行。
高亮显示的启动项将在20秒后自动执行。



```
- Missing modules (cat /proc/modules; ls /dev)
ALERT!  UUID=791fb645-b65d-42d3-8a4c-f3585025feae does not exist.  Dropping to a
shell!

BusyBox v1.36.1 (Ubuntu 1:1.36.1-6ubuntu3.1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

(initramfs) help
Built-in commands:
-----
. : [ alias break cd chdir command continue echo eval exec exit
export false getopt hash help history let local printf pwd read
readonly return set shift sleep test times trap true type ulimit
umask unalias unset wait [ [[ acpid arch ascii ash awk base32
basename blockdev busybox cat chmod chroot chvt clear cmp cp
crc32 cut date deallocvt deluser devmem df dirname du dumpkmap
echo egrep env expr false fbset fgrep find fold fstrim grep gunzip
gzip hostname hwclock i2ctransfer ifconfig ip kill ln loadfont
loadkmap ls lzop mkdir mkfifo mknod mkswap mktemp modinfo more
mount my nuke openvt pidof printf ps pwd readlink reboot reset
rm rmdir run-init sed seq setkeycodes sh sleep sort stat static-sh
stty switch_root sync tail tee test touch tr true ts tty umount
uname uniq wc wget which yes
(initramfs) _
```



二、冒泡排序、基础堆排序和斐波那契堆排序的实现细节

实验目的：考验代码书写硬实力，理解排序结构和实现原理，并学会导入数据基本测试查错和实际运用

1. 冒泡排序

```
void bubbleSortInt(int arr[], int n) {  
    bool swapped;  
    for (int i = 0; i < n - 1; i++) {  
        swapped = false;  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
                swapped = true;  
            }  
        }  
        if (!swapped) {  
            break;  
        }  
    }  
}
```

代码实现：冒泡排序真正的代码实现实际上只有两个嵌套for循环结构和一个if判断语句的使用，用于遍历元素进行像冒泡泡一样的对比筛选机制使较大的“泡泡”逐渐上浮排序。外循环利用bool值设定用于表示这次遍历开始时没有发生元素交换，内循环进行元素交换和排序的工作，每次循环后最后i个数是已经排序好的，所以只需要遍历到n-i-1即可。内循环中的三行代码是执行对比交换工作的核心，排好序后bool值表示经过交换，则循环继续进行。如果数据一开始就排好序，则系统直接执行外部if语句，跳出循环避免不必要的空耗运行。读取浮点数数据只需要修改数据类型即可

```

#include <stdio.h>
#include <stdbool.h>

void bubbleSortInt(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) {
            break;
        }
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);

    bubbleSortInt(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}

```

运行测试：

先以上述简单数组为例进行测试，后续再导入复杂数据
键入

```
gcc -o bubble_test bubble_test.c  
./bubble_test
```

在gcc编译器中编译并成功执行

```
muili@muili-VirtualBox:~/桌面/NCUSCC-Cproject/src$ gcc -o bubble_test bubble_te  
t.c  
muili@muili-VirtualBox:~/桌面/NCUSCC-Cproject/src$ ./bubble_test  
Original array:  
64 34 25 12 22 11 90  
Sorted array:  
11 12 22 25 34 64 90  
muili@muili-VirtualBox:~/桌面/NCUSCC-Cproject/src$ S
```

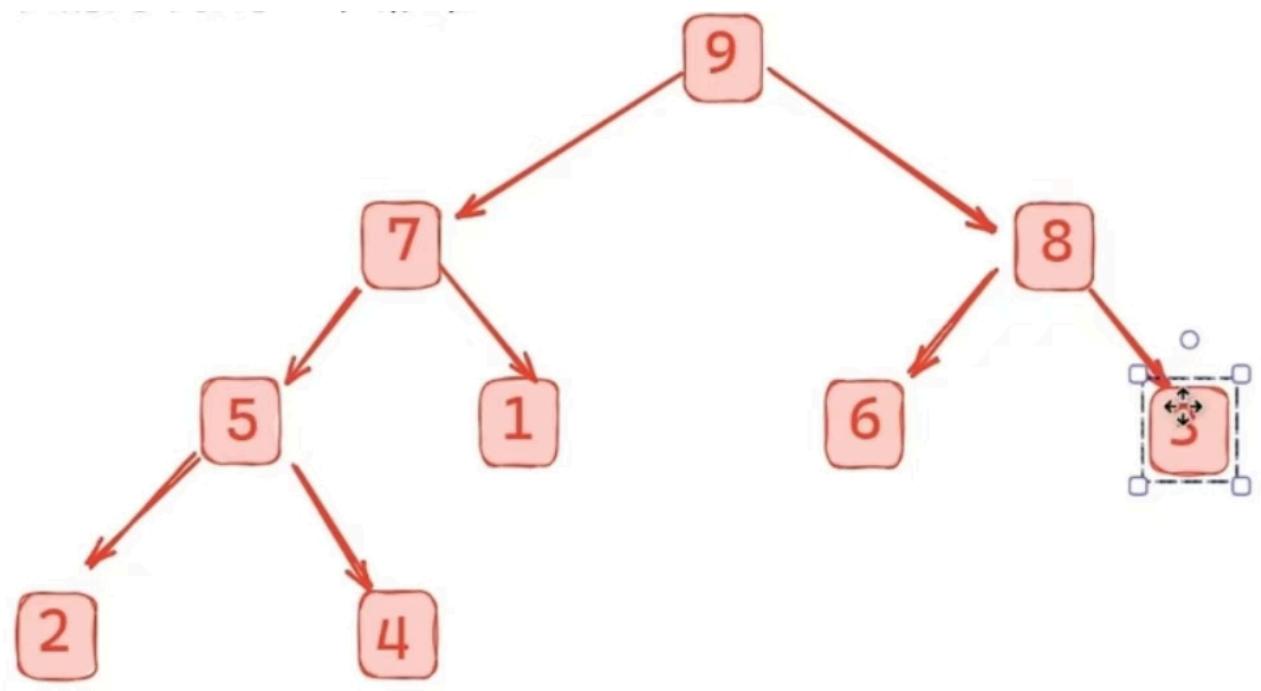
数据读取的实现

```
// Read and sort integer data  
for (int i = 0; i < 3; i++) {  
    sizes[i] = readIntDataFromFile(filenames[i], intArrays[i]);  
    bubbleSortInt(intArrays[i], sizes[i]);  
    printf("Sorted %s: ", filenames[i]);  
    printIntArray(intArrays[i], sizes[i]);  
}  
  
// Read and sort float data  
for (int i = 3; i < 6; i++) {  
    sizes[i] = readFloatDataFromFile(filenames[i], floatArrays[i - 3]);  
    bubbleSortFloat(floatArrays[i - 3], sizes[i]);  
    printf("Sorted %s: ", filenames[i]);  
    printFloatArray(floatArrays[i - 3], sizes[i]);  
}
```

**对于我所生成的浮点数

2.基础堆排序

**代码实现：基础堆排序主要以一种特殊完全二叉树的数据结构为其基准，实现更为高效的排序；首先进行的是大根堆的数据排序，其次是每次将大根堆堆顶的最大数字与最后存储位的数据交换再排出最大数据，然后继续大根堆排序循环直至排出顺序



```

// 交换两个整数元素
void swapInt(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// 堆化子树（整数版本）
void heapifyInt(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swapInt(&arr[i], &arr[largest]);
        heapifyInt(arr, n, largest);
    }
}

// 堆排序（整数版本）
void heapSortInt(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapifyInt(arr, n, i);
    }

    for (int i = n - 1; i >= 0; i--) {
        swapInt(&arr[0], &arr[i]);
        heapifyInt(arr, i, 0);
    }
}

```

发现问题：

1. 数据调用
 2. 斐波那契堆排序的代码实现
- (1) 斐波那契堆初始化

```
// 定义节点结构体
typedef struct Node {
    int key;                      // 节点的键值
    struct Node* parent;           // 指向父节点的指针
    struct Node* child;            // 指向子节点的指针
    struct Node* left;             // 左兄弟节点
    struct Node* right;            // 右兄弟节点
    int degree;                   // 节点的度数，即子节点的数量
    int mark;                     // 用于标记节点是否被删除
} Node;
```

```
// 定义斐波那契堆结构体
typedef struct {
    Node* min;                   // 指向最小节点的指针
    int total_nodes;              // 堆中节点的总数
    int total_degree;             // 堆中所有节点的度数之和
} FibonacciHeap;
```

```
// 创建一个新的节点
Node* createNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    if (!node) {
        perror("Failed to allocate memory for node");
        exit(EXIT_FAILURE);
    }
    node->key = key;
    node->parent = NULL;
    node->child = NULL;
    node->left = node;           // 初始时节点是自己的左兄弟
    node->right = node;          // 初始时节点是自己的右兄弟
    node->degree = 0;
    node->mark = 0;
    return node;
}
```

```
// 初始化斐波那契堆
void initFibonacciHeap(FibonacciHeap* heap) {
    heap->min = NULL;
    heap->total_nodes = 0;
    heap->total_degree = 0;
}
```

```

// 将节点添加到斐波那契堆中
void insertNode(FibonacciHeap* heap, Node* node) {
    node->left = node;
    node->right = node;
    if (heap->min == NULL) {
        heap->min = node;
    } else {
        node->right = heap->min;
        node->left = heap->min->left;
        heap->min->left->right = node;
        heap->min->left = node;
    }
    heap->total_nodes++;
}

```

(2)斐波那契堆的合并和排序

```

// 合并两个斐波那契堆
void mergeHeaps(FibonacciHeap* heap1, FibonacciHeap* heap2) {
    if (heap2->min == NULL) {
        free(heap2);
        return;
    }
    if (heap1->min == NULL) {
        *heap1 = *heap2;
        free(heap2);
        return;
    }
    if (heap1->min->key > heap2->min->key) {
        Node* temp = heap1->min;
        heap1->min = heap2->min;
        heap2->min = temp;
    }
    Node* temp_left = heap1->min->left;
    Node* temp_right = heap1->min->right;
    temp_left->right = heap2->min;
    heap2->min->left = temp_left;
    temp_right->left = heap2->min->right;
    heap2->min->right = temp_right;
    heap1->total_nodes += heap2->total_nodes;
    heap1->total_degree += heap2->total_degree;
    free(heap2);
}

```

```

// 提取斐波那契堆中的最小元素
Node* extractMin(FibonacciHeap* heap) {
    Node* minNode = heap->min;
    if (minNode != NULL) {
        if (minNode->child != NULL) {
            Node* child = minNode->child;
            while (child != minNode) {
                child->parent = NULL;
                child->mark = 0;
                insertNode(heap, child);
                child = child->right;
            }
        }
        if (minNode->left == minNode) {
            heap->min = NULL;
        } else {
            heap->min = minNode->right;
            minNode->left->right = minNode->right;
            minNode->right->left = minNode->left;
        }
        heap->total_nodes--;
    }
    return minNode;
}

```

```

// 使用斐波那契堆对数组进行排序
void fibonacciHeapSort(int arr[], int n) {
    FibonacciHeap heap;
    initFibonacciHeap(&heap);

    for (int i = 0; i < n; i++) {
        insertNode(&heap, createNode(arr[i]));
    }

    for (int i = 0; i < n; i++) {
        Node* minNode = extractMin(&heap);
        if (minNode != NULL) {
            arr[i] = minNode->key;
            free(minNode);
        }
    }
}

```

(3)简单数组进行测试

```
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    fibonacciHeapSort(arr, n);

    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

解决方案：

1.

```
const char* filenames[] = {"small_data.txt", "medium_data.txt",
    "large_data.txt", "small_data_float.txt", "medium_data_float.txt", "large_data_float.txt"};
直接使用了const char* 类型的指针，指向每一个字符串字面量，进而达到调用文件的目的
```

2.

三、测试数据的生成方法

实验目的：学会使用脚本生成不同规模的数据，用于程序的性能和效率测试

1. 合成随机数代码脚本细节

(1) 不同规模数据的生成

```

void generate_random_data(const char* filename, int num_elements, int is_float) {
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        perror("Unable to open file");
        return;
    }

    srand((unsigned int)time(NULL));

    for (int i = 0; i < num_elements; ++i) {
        if (is_float) {
            double random_float = (double)rand() / RAND_MAX * 10000;
            fprintf(file, "%f\n", random_float);
        } else {
            int random_int = rand() % 10000;
            fprintf(file, "%d\n", random_int);
        }
    }
    fclose(file);
}

```

该C语言随机数生成器主要以当前系统时间 (time(NULL)) 作为种子，以确保生成数据的随机性；如果is_float值为真，则生成随机浮点数，通过rand()转化浮点数在进行增位减位获得10000以内浮点数，随机整数的生成逻辑类似

(2)将100000条数据导入csv文件，这里我选择了shell脚本直接实现实验数据的获取和导入，处理脚本时发现了容易忽视的小问题，文件名中间一定要记得打空格，不然问题真的很难检索 😱

```

# 定义测试数据文件数组
test_data_files=("small_data.txt" "medium_data.txt" "large_data.txt" "small_data_float.txt" "me

```

(3)估测时间复杂度与实验数据的比较

在非最佳情况下的时间复杂度理应是 $O(n^2)$ ，后续实验数据也能佐证这一点，实验数据吻合较好；基础堆排序

发现问题：

1. 数据文件名称和文件引用名称不同

解决方案：

1. 编写代码时一定要查清文件名的大小写、分隔符、文件后缀，以防后续问题

四、不同编译优化等级下的性能对比结果

实验目的：了解表现程序算法优化性能的等级，体会算法优化对不同排序的不同程度，对比冒泡排序和基础堆排序的优化效果，分析二者特点

1. 编译优化等级种类

- O0：没有优化。编译器不会进行任何优化，这通常会导致代码运行得更慢，但编译速度更快，并且生成的二进制文件更容易调试。
- O1：启用基本优化。这是默认的优化等级，它会启用一些不会显著增加编译时间的优化。
- O2：进一步优化。这个等级会启用更多的优化选项，旨在提高程序的运行效率，同时保持合理的编译时间。
- O3：启用更多的优化，包括那些可能会增加编译时间的优化。这个等级包括-O2的所有优化，并添加了更多的优化选项。
- Os：空间优化。这个等级的优化旨在生成更小的二进制文件，可能会牺牲一些运行时性能。
- Ofast：这是一个不标准的优化等级，它启用了所有-O3的优化，并包括一些不安全的优化，比如放宽浮点数精度以换取性能提升。

2. 性能对比结果

堆排序的运行时间在不同优化等级下变化幅度小且运行时间短，排序效率高，而相较之下冒泡排序时间变化幅度明显，运行时间较长，排序效率较低较为原始

详细数据可见文件[performance_data.csv](#)

发现问题：

1. 因为内存不足或其他原因，没有去掉csv文件中的出数字、小数点外的英文字母，导致图像绘制出现阻碍
2. 没有理解时间复杂度的表示含义

解决方法：

1. 可以使用进程异常结束前的数据暂代全程数据使用处理字母作为数据使用

```
def clean_and_convert_time(time_str):
```

2. 大致以数学中的函数增长速率快慢来间接理解

五、数据可视化

实验目的：直观观察感受数据变化和排序算法性能，并掌握搭建虚拟环境使用Python类脚本图库进行可视化的能力

1. Python虚拟环境搭建

搭建Python脚本虚拟环境所需的虚拟软件

```
sudo apt install python3.12-venv
```

创建并命名一个新的虚拟环境

```
| python3 -m venv myenv
```

激活虚拟环境

```
| source myenv/bin/activate
```

以及退出虚拟环境的指令

```
| deactivate
```

另外，为了以图像的形式呈现，在虚拟环境中安装了pandas库

```
| pip3 install pandas
```

2.脚本实现

对比效果：

由折线图可以看出冒泡排序的排序时间明显长于堆排序，且波动性大，在不同的数据环境中所用时间

具有更大的不确定性，但二者内存占用没有太大差别

经由折线图的直观展示更能展现两种排序的效率差异

图库见[visualization](#)

发现问题：

1.在虚拟环境中运行脚本总是报同样的错，修改后也仍然报错，现仍然不清楚原因，看来只能等深度学习后后续解决了

总结：这次考核我真的学到了很多东西，虽然我的计算机基础真的很差，是个在小县城里十几年没碰过电脑的小白，最多的专业知识也就是在暑假听的80多节C语言网课，但是我也真的很感谢当时的自己，虽然这些基础知识不能让我短时间内成为代码大牛，但好歹是为我的计算机之路垫上了一块敲门砖。实际上我的实验做得很坎坷，就算是到了最后即将上交的节点时也仍然有很多问题没有搞透或者没有解决，但是只是顺着自己的路往前走，就算是三不摔两步也终究还是收获了很多知识，像虚拟机的使用，逐渐分清计算机的文件管理逻辑，还有Markdown居然是种语法而不是什么占有软件名词等等，还拓宽了自己视野和对计算机的认识。这种学习的状态是我在高中三年所没有体会过的，我会为了项目的失败而难受，但不会为了一两分抓耳

挠腮，打开任务要求全是盲区确实让人头皮发麻，但是只要静下心来慢慢学习，慢慢思考，最终我也是勉强搭出了个毛坯房 😊。最后再次感谢牢e的精彩宣讲，让我真正有机会找到一条曲折但最终能通向罗马的道路，挑战自己。就算这次失败了，我相信自己也会沿着这条路继续出发，慢慢走也能走到终点。毕竟我还没看过"金子做"的大机子呢 😢。

附：

仍待解决的问题：

1. 独立完成斐波那契堆排序
2. 数据收集的异常中断问题
3. 虚拟环境下的卡文本问题
4. 学会使用Python写出自己的脚本
5. 继续深度学习数据结构，对各种结构有更清晰的认识

(以下为本人对自己后续需要修正问题的总结)

特别鸣谢：

kimi^[1]

howxu

(借鉴了浩哥的一些文件命名方式，还有文件存放管理方法

1. kimi, 一款有时人性的人工智能 ↵