

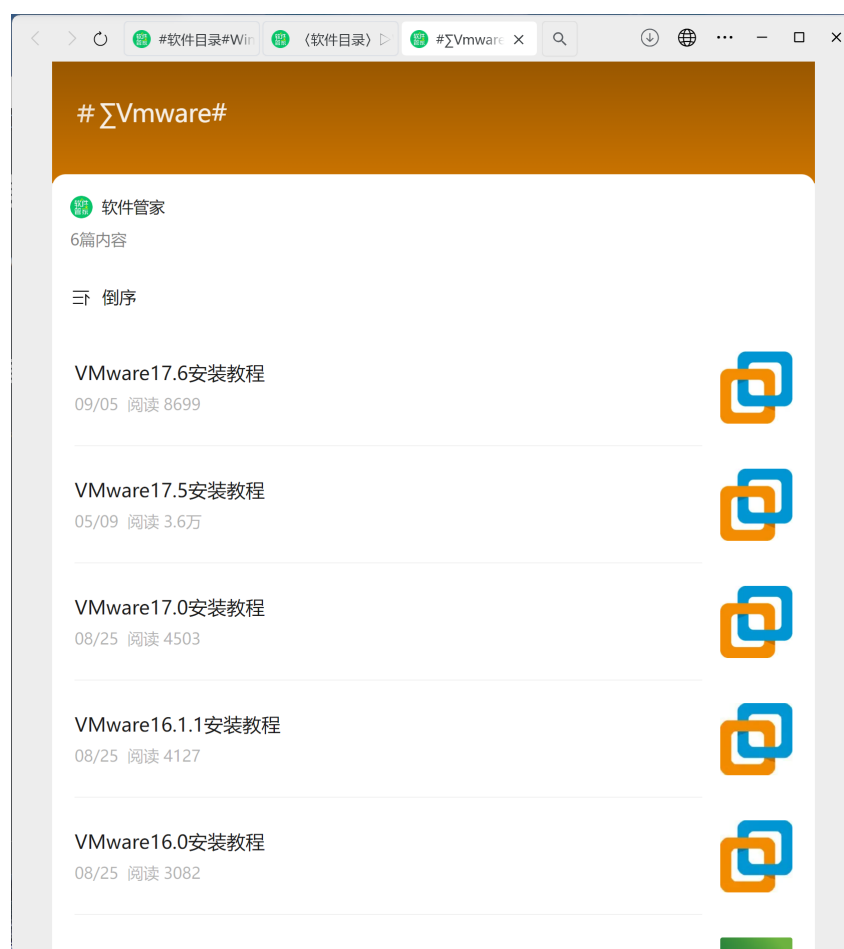
# 选题：C 语言试题

系统及硬件配置：Windows 11，12th Gen Intel(R) Core(TM) i5-12450H，RAM 16.0 GB

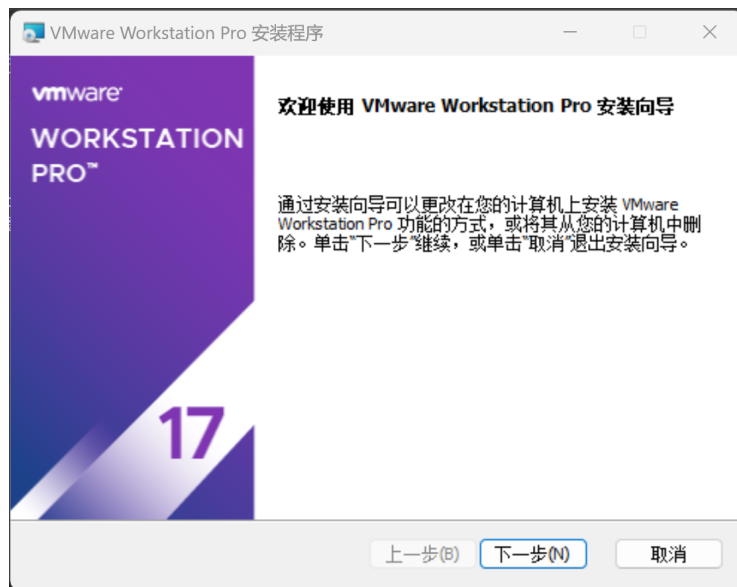
## 1. 虚拟机搭建过程

### 下载VMware

打开微信公众号“软件管家”，点击软件目录 → Windows 软件目录，在目录“电脑系统”栏找到 VMware 选项，进入后可看到如下界面：



选择 VMware17.6 安装教程，进入链接下载安装文件（这里使用更快的迅雷下载），待下载完成后按照提示完成解压安装：



## 安装Ubuntu映像

直接去Ubuntu中文官网上下载（地址<https://cn.ubuntu.com/download/desktop>），此处下载的版本为Ubuntu 22.04 LTS，与题目要求的一致。

下载好后导入VMware进行简易安装：



根据安装步骤安装好后即可进入系统：



检查一下联网配置发现可以正常使用

## 2. 安装C语言编译器

在网站上搜索 Ubuntu 清华源，更改系统目录下的 `\etc\apt\source.list`，将清华源的代码输入。接着在终端上输入

```
sudo apt-get update
sudo apt install gcc
```

等待安装完成后即可。编写一个简单的c语言源代码，在终端编译，检查是否能正常运行

```
xinxin@xinxin-VMware-Virtual-Platform:~/文档$ gcc -o new new.c
xinxin@xinxin-VMware-Virtual-Platform:~/文档$ ./new
hello world!
xinxin@xinxin-VMware-Virtual-Platform:~/文档$ S
```

## 3.使用C语言实现排序算法

现要求分别使用冒泡排序，基础堆排序和斐波那契堆排序对数据规模至少为  $10^5$  的数据进行排序，为了方便假定要排序的数据均为正整数，且大小不超过  $10^9$ （实数或其他数据范围类似，不影响算法实现）。

## 冒泡排序

```
int n; scanf("%d",&n);
for(int i=1;i<=n;i++)scanf("%d",&a[i]);
for(int i=n;i>=1;i--)
    for(int j=1;j<i;j++)
        if(a[j]>a[j+1])swap(a[j],a[j+1]);
for(int i=1;i<=n;i++)printf("%d ",a[i]);
```

## 基础堆排序

基础二叉堆采用顺序数组的形式存储，下标 1 为根节点，下标  $i$  对应第  $i$  个节点，其左右儿子分别为  $2i$  和  $2i + 1$  号节点，这样的二叉树表示形式可以很方便地进行储存与访问，剩下便是实现二叉堆的两个基本操作：插入数据和弹出堆顶

```
int a[N],b[M],n,cnt;
void insert(int x){ // 插入数据
    b[++cnt]=x; int i=cnt;
    while(i!=1&& b[i/2]>b[i]){
        swap(b[i],b[i/2]);
        i/=2;
    }
}
void mesh(){ // 弹出堆顶的数据并调整堆结构
    printf("%d ",b[1]);
    b[1]=b[cnt]; b[cnt]=inf; cnt--;
    int i=1;
    while(b[i]>b[i*2]||b[i]>b[i*2+1]){
        int x=b[i*2]<b[i*2+1]?i*2:i*2+1;
        swap(b[i],b[x]); i=x;
    }
}
```

主函数实现部分：

```
scanf("%d",&n);
for(int i=1;i<=n;i++)scanf("%d",&a[i]);
memset(b,0x3f,sizeof(b));
for(int i=1;i<=n;i++)insert(a[i]);
for(int i=1;i<=n;i++)mesh();
```

## 斐波那契堆排序

斐波那契堆功能多样，完整的斐波那契堆实现需要用到双向链表、二叉树的孩子兄弟表示法以及延迟删除标记等，但本题所需的斐波那契堆只需要插入和弹出两个操作，故可以在实现上进行一定简化。

斐波那契堆的核心思想是维护一个根节点度数互不相同的森林，每棵树均满足堆性质，且两个度数相等的堆恰可以合并为一个度数+1的新堆，这提示我们可以为每一种度数的堆开一个指针记录它的位置，便可以把整个森林记录下来。

在每个堆内部的树结构储存上，作者放弃实现冗杂的双向链表结构，采用更加简单的**链式前向星**进行存储，在不涉及数据减值等复杂操作的情况下，此方式在常数和实现难度上有更好的效果。

核心操作：

```
int hd[N],pre[M],to[M],num; // 链式前向星
int d[N],point; // d[i]指向度数为i的根节点
int a[N],n;
int stack[N],cnt;
int get(){
    return cnt?stack[cnt--]:++num; // 因为操作过程中涉及到边的加删，故使用栈回收利用空间
}
void adde(int x,int y){
    int nm=get();pre[nm]=hd[x];hd[x]=nm;to[nm]=y;
}
void update(int i){ // 检查d[i]是否为最小的根节点
    if(a[d[i]]<a[d[point]])point=i;
}
int top,deg[N];
int max(int x,int y){
    return x>y?x:y;
}
void insert(int x,int i){
    while(d[i]){ // 尝试在i处插入，若度数为i的根已存在，则不断合并直到不存在度数相同的根为止
        if(a[d[i]]>a[x])swap(d[i],x);
        adde(d[i],x); deg[d[i]]++;
        x=d[i]; d[i]=0; i++;
    }
    d[i]=x; update(i);
    top=max(top,i);
}
void mesh(){ // 弹出最小值
```

```

printf("%d ",a[d[point]]);
int x=d[point]; d[point]=0; // point记录最小的根节点位置
if(point==top)top--;
for(int i=hd[x];i;i=pre[i]){
    int u=to[i];
    insert(u,deg[u]); // 将最小的根弹出后，把它的每一棵子树都插入到森林中
}
for(int i=hd[x];i;i=pre[i])stack[++cnt]=i; // 空间回收
for(int i=0;i<=top;i++)update(i); // 访问每一个根节点更新point
}

```

主函数部分：

```

scanf("%d",&n);
for(int i=1;i<=n;i++)scanf("%d",&a[i]);
a[0]=inf;
for(int i=1;i<=n;i++)insert(i,0);
for(int i=1;i<=n;i++)mesh();
printf("\n");

```

## 测试算法正确性

本地测试：

```

data.in:
10
9 8 2 7 6 1 5 4 3 10
bubble.out:
1 2 3 4 5 6 7 8 9 10
bin.out:
1 2 3 4 5 6 7 8 9 10
fib.out:
1 2 3 4 5 6 7 8 9 10

```

可以看到在该测试数据下三个算法的输出结果均正确，我们再将三种排序算法的代码提交到 <https://www.luogu.com.cn/problem/P1177> 进行评测，其中冒泡排序通过了  $10^3$  规模的数据（较大数据超时），二叉堆排序与斐波那契堆排序通过了全部数据，可以认为三种排序算法在实现上正确性可以保证。

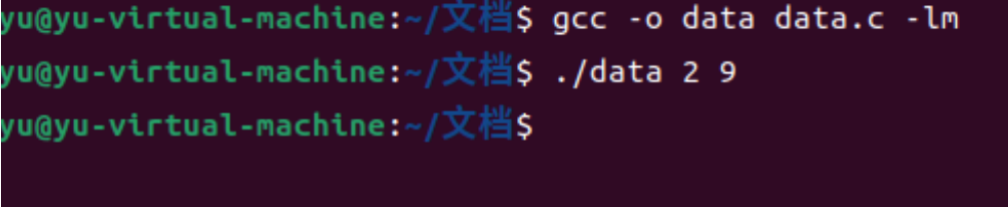
## 4.生成数据与性能测试

### 数据生成

编写 data.c, 使用 c 自带随机函数 rand() 对测试数据进行生成, 根据运行时的参数将对应规模的随机数据生成到 data.in 中

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"
int rd(int x,int y){
    return rand()%(y-x+1)+x;
}
int main(int argc, char *argv[])
{
    srand((unsigned)time(NULL));
    int n=pow(10,(argv[1][0]-'0')),w=pow(10,(argv[2][0]-'0'));
    freopen("data.in","w",stdout);
    printf("%d\n",n);
    for(int i=1;i<=n;i++)printf("%d ",rd(1,w));
    return 0;
}
```

如下图编译运行, 运行时传入的两个参数分别代表数据规模以及要排序的数字大小。例如下图中 data 将会在把一组包含  $10^2$  条数据, 每个数字在  $[1, 1e9]$  范围内的测试数据输出到 data.in 中。



```
yu@yu-virtual-machine:~/文档$ gcc -o data data.c -lm
yu@yu-virtual-machine:~/文档$ ./data 2 9
yu@yu-virtual-machine:~/文档$
```

### 在不同优化选项下运行

使用 -O0, -O1, -O2, -Ofast 分别编译以上三个程序, 在 data.c 生成的一组  $10^5$  级别的数据下使用 time ./ 命令获取它们的运行时间, 得到如下结果:

冒泡排序运行时间

优化级别	-O0	-O1	-O2	-Ofast
运行时间	20.327s	11.906s	12.118s	12.122s

基础堆排序运行时间

优化级别	-O0	-O1	-O2	-Ofast
运行时间	0.031s	0.021s	0.025s	0.025s

### 斐波那契堆排序运行时间

优化级别	-O0	-O1	-O2	-Ofast
运行时间	0.080s	0.052s	0.050s	0.045s

可以看到除了-O0 优化选项运行时间相对较长外，其他三个选项的运行用时接近，可认为在波动范围之内，故可知编译优化在 -O1 级别时就已接近优化临界值，在上述三个程序中进一步优化并不会带来显著的效率提升。

根据算法的复杂度分析可知冒泡排序的时间复杂度为  $O(n^2)$ ，而基础堆排序和斐波那契堆排序的复杂度均为  $O(n \log n)$ ，从上述表格中得到，在最高的编译优化选项下，冒泡排序的运行时间在两种堆排序的 200 ~ 500 倍之间，符合上述的复杂度分析。

## 使用脚本收集更多的性能数据

为了避免重复的测试操作，作者编写了如下脚本，可以一次性将某一数据范围内一个程序在各优化选项下的性能数据（时间、空间占用）写入到一个 .csv 文件中

```
#!/bin/bash

# 定义文件名和优化级别
FILE="$1.c"
OPTIMIZATION_LEVELS=("O0" "O1" "O2" "Ofast")

# 数据规模
./data $2 9

# 创建一个CSV文件来存储结果
OUTPUT_FILE="$1$2_results.csv"
echo "优化级别,运行时间(s),内存占用(KB)" > $OUTPUT_FILE

# 遍历每个优化级别
for OPT_LEVEL in "${OPTIMIZATION_LEVELS[@]"; do
    # 编译文件
    gcc -$OPT_LEVEL -o new_$OPT_LEVEL $FILE

    # 运行并统计时间和内存使用
    echo "Running with optimization level -$OPT_LEVEL..."
```



```

TIME_OUTPUT=$(/usr/bin/time -f "%e,%M" ./new_${OPT_LEVEL} 2>&1)

# 解析时间和内存使用
RUN_TIME=$(echo $TIME_OUTPUT | cut -d',' -f1)
MEM_USAGE=$(echo $TIME_OUTPUT | cut -d',' -f2)

# 将结果写入CSV文件
echo "-${OPT_LEVEL},${RUN_TIME},${MEM_USAGE}" >> $OUTPUT_FILE

# 删除生成的可执行文件
rm new_${OPT_LEVEL}

done

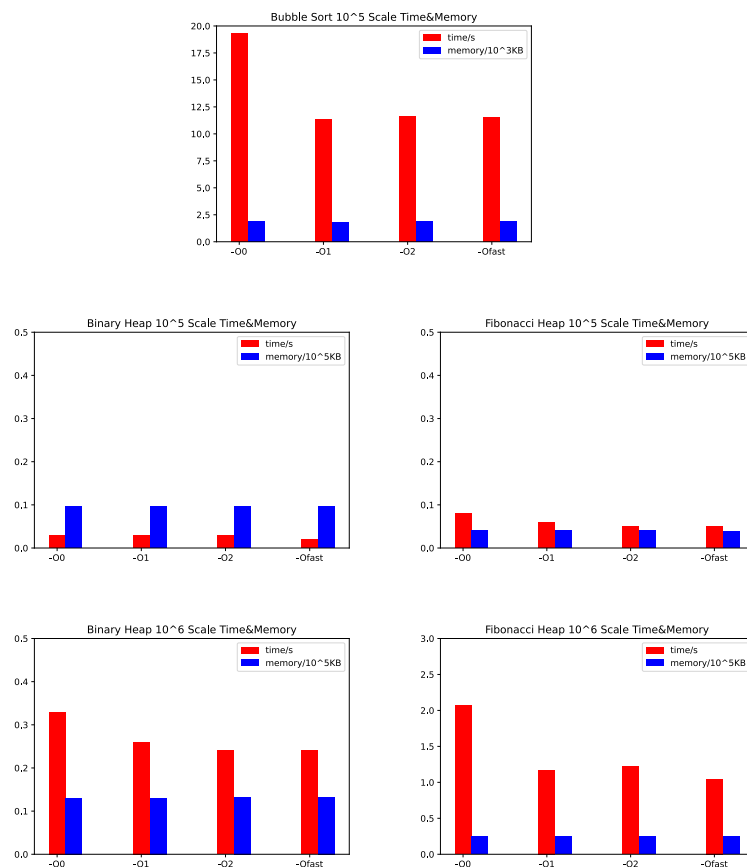
echo "Results have been saved to $OUTPUT_FILE"

```

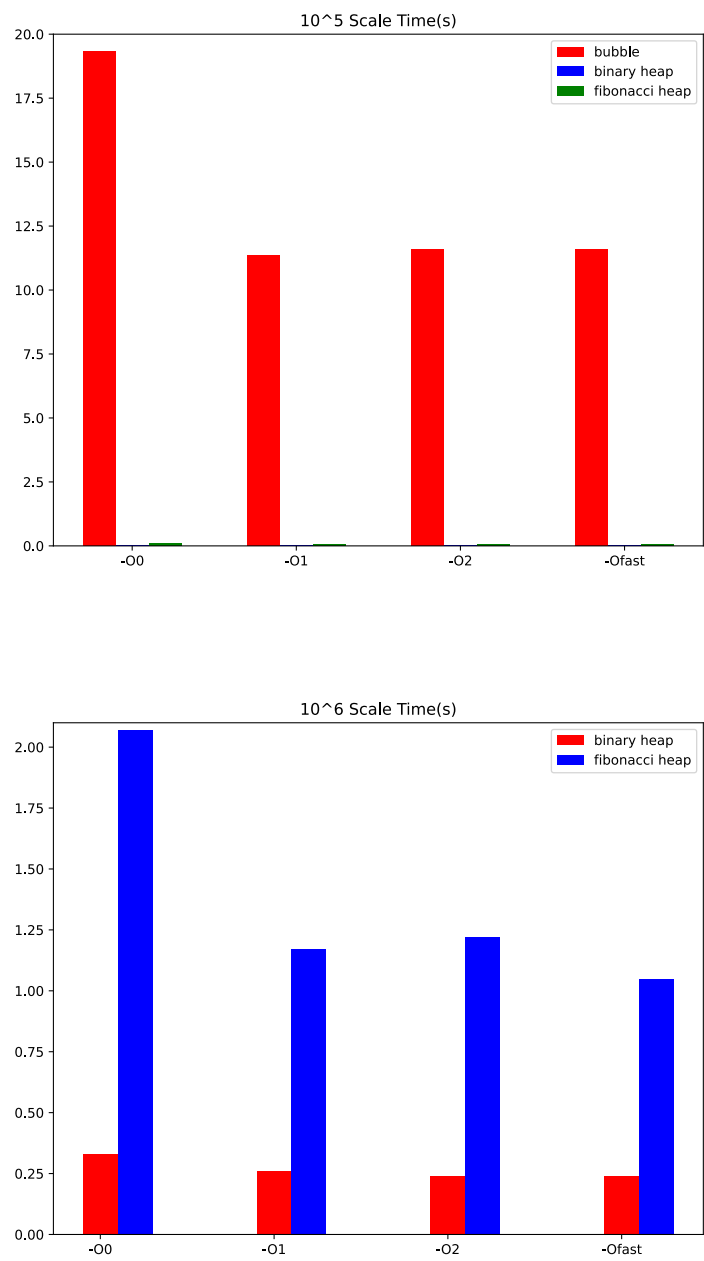
分别在  $10^5$   $10^6$  数据规模下对三个程序运行测试，性能测试数据存储在 bubble5\_results.csv bin5\_results.csv bin6\_results.csv fib5\_results.csv fib6\_results.csv 中（其中  $10^6$  数据规模下冒泡排序的性能测试因运行时间过长且无参考价值而跳过）

## 测试数据的可视化

使用 python 的 matplotlib.pyplot 库对上述收集到的数据进行可视化，作者分别绘制了不同数据规模下三种排序在各优化选项下时间、空间性能的对比



此外，还绘制了  $10^5$  与  $10^6$  规模下各算法在各优化选项下的运行效率对比图：



可以看到在  $10^5$  规模数据下冒泡排序的运行效率显著低于另外两种排序方式，这是由算法复杂度决定的；而  $10^6$  规模下二叉堆排序与斐波那契堆排序的运行效率相差一个较大的常数，且这个常数在加入编译优化后有较大程度的优化。推测这可能是因为斐波那契堆本身实现上的复杂度比二叉堆高很多，并且程序运行过程中发生了许多函数调用，所以其运行效率对受优化选项的影响较大；相比之下二叉堆运行效率受优化选项的影响有限，因其实现较为简单，可优化空间小。

二叉堆

优化级别	运行时间(s)	内存占用(KB)
-O0	0.33	13056

优化级别	运行时间(s)	内存占用(KB)
-O1	0.26	13056
-O2	0.24	13312
-Ofast	0.24	13312

斐波那契堆

优化级别	运行时间(s)	内存占用(KB)
-O0	2.07	24960
-O1	1.17	24960
-O2	1.22	24960
-Ofast	1.05	24960

## 5. 实验过程中遇到的问题及解决方案

1. 实验过程中的 Ubuntu 是在虚拟机中运行的，这个过程中需要与主机进行某些文件文本的传输，但是刚安装好的 linux 虚拟机不支持与主机之间的文本复制粘贴、文件拖拽等操作。解决方案：按照<https://blog.csdn.net/davidhzq/article/details/101621482>中所写的进行操作，安装 vmware tools 后问题解决。
2. `/etc/apt/source.list` 被系统锁定无法修改，解决方案为在终端中进入对应的目录，使用 `sudo` 语句将其打开，即可获得修改权限
3. 一开始试图使用 `ps` 命令得到程序运行的内存大小，但是在程序运行时间很短的情况下，`ps` 命令无法捕获到程序相关的信息，解决方案为使用 `/usr/bin/time -p ./` 命令，可以同时监视一个程序运行过程中的时间和内存使用使用情况。
4. 在python绘图程序编写过程中，遇到了如下语句：`map(eval,s)`，其中s为一个type为str的一维 `numpy.ndarray` 对象，该语句试图将一个由字符串组成的数组转换为数值数组，但该语句只会返回一个map对象，并不会得到一个与 s 相同结构的数值数组。解决方案：将map对象先转换为 list 对象，再进一步转为 `numpy.ndarray` 对象，即改为 `numpy.array(list(map(eval,s)))`即可

## 6.鸣谢

感谢北京深度求索人工智能基础技术研究有限公司开发的 deepseek 大语言模型，在实验过程中提供了很多问题的解答，并提供了宝贵的学习资料。

感谢本次考核出题组精心准备的试题，在完成题目的过程中本人学习到很多未曾涉猎的新知识，掌握了不少相关的技术能力。