

NCUSCC 2024秋超算考核试题 C语言试题报告

Author: Orchid

试题要求

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Orchid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

一、实验环境搭建

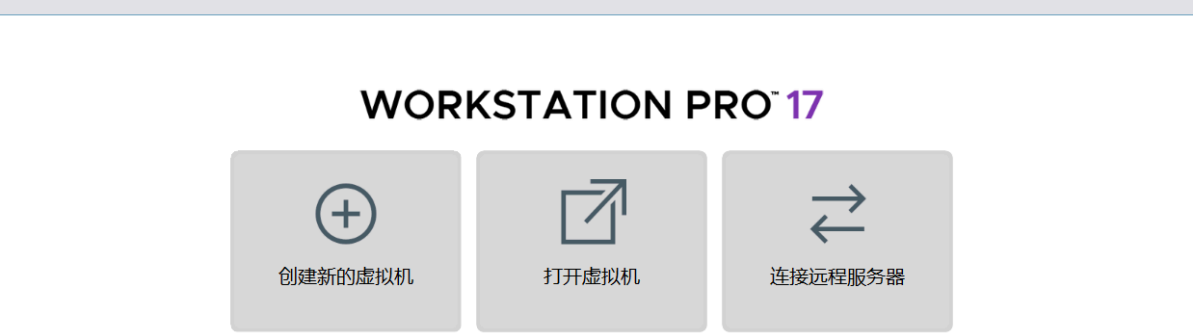
1.VMWare虚拟安装Ubuntu 22.04 LTS操作系统

(1) 前往浙江大学开源镜像站下载Ubuntu 22.04 LTS amd64镜像:



Index of /ubuntu-releases/24.04.1/

../	29-Aug-2024 16:07	-
netboot/	29-Aug-2024 16:08	810
FOOTER.html	29-Aug-2024 16:08	4582
HEADER.html	29-Aug-2024 16:08	202
SHA256SUMS	29-Aug-2024 16:08	833
SHA256SUMS.gpg	27-Aug-2024 16:25	6203355136
ubuntu-24.04.1-desktop-amd64.iso	29-Aug-2024 16:03	473638
ubuntu-24.04.1-desktop-amd64.iso.torrent	29-Aug-2024 16:02	13630656
ubuntu-24.04.1-desktop-amd64.iso.zsync	27-Aug-2024 16:25	26830
ubuntu-24.04.1-desktop-amd64.list	27-Aug-2024 16:18	61313
ubuntu-24.04.1-desktop-amd64.manifest	27-Aug-2024 15:40	2773874688
ubuntu-24.04.1-live-server-amd64.iso	29-Aug-2024 16:08	212002
ubuntu-24.04.1-live-server-amd64.iso.torrent	29-Aug-2024 16:08	5417973
ubuntu-24.04.1-live-server-amd64.iso.zsync	27-Aug-2024 15:40	19120
ubuntu-24.04.1-live-server-amd64.list	27-Aug-2024 14:30	20960
ubuntu-24.04.1-live-server-amd64.manifest	29-Aug-2024 16:07	84853185

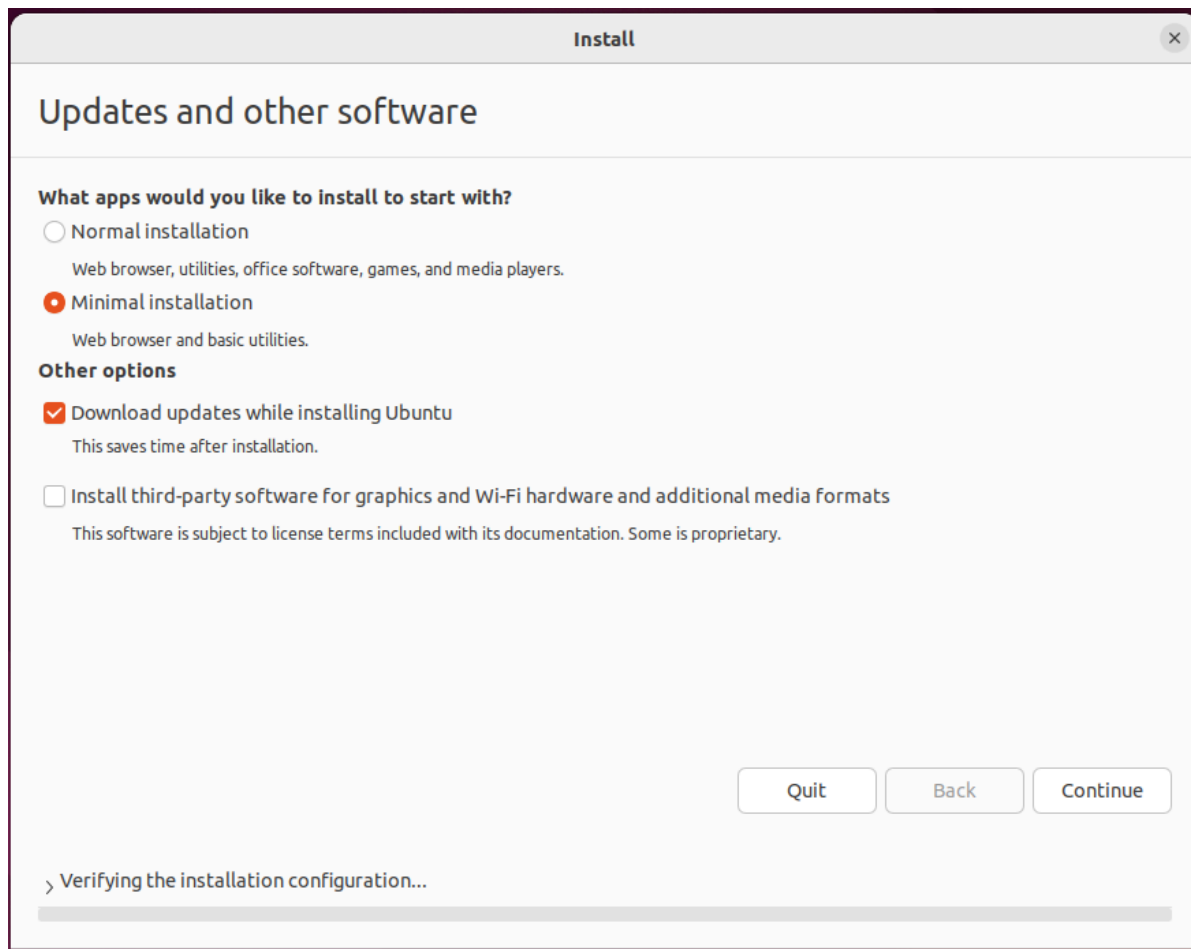
(2) 打开VMWare虚拟机,选择创建新的虚拟机



(3) 选择合适的CPU数量、核心、运行内存:

设备	摘要
 内存	8 GB
 处理器	16
 硬盘 (SCSI)	80 GB

(4)启动虚拟机,选择最小安装设置,取消更新勾选:



2.安装C语言编译器

(1)安装gcc编译器工具,使用 `sudo apt install build-essential` 命令进行安装:

```
orchid@orchid-virtual-machine:~$ sudo apt install build-essential
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
build-essential 已经是最新版 (12.9ubuntu3)。
```

使用 `gcc -v` 检测gcc是否正常:

```
orchid@orchid-virtual-machine:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/11/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none;andgcn-andhsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 11.4.0-1ubuntu22.04' --with-bugurl=file:///usr/share/doc/gcc-11/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,n2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-11 --program-prefix=x86_64-linux-gnu --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdc++-debug --enable-libstdc++-timeyes --with-default-libstdcxx-abi=new --enable-gnu-unIQUE-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --enable-llvmpobos-checking=release --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multitarch --disable-werror --enable-cet --with-arch=32+1686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none;/build/gcc-11-xe791V/gcc-11-11.4.0/debian/tmp-nvptx/usr, andgcn-andhsa;/build/gcc-11-xe791V/gcc-11-11.4.0/debian/tmp-gcn/usr --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu --with-build-config=bootstrap-lto-lean --enable-link-serialization=2
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu22.04)
```

二、算法的实现细节 测试数据生成方法

3.4.实现算法,算法测试和大量数据测试

(1)算法实现

算法实现细节如下:

冒泡排序:

冒泡排序通过重复遍历要排序的数列，一次比较两个元素，若顺序错误则将其调转过来，直至所有元素均位于正确的位置上，只需两个嵌套的循环即可实现

```
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        // 最后i个元素已经是排好序的了，不需要再比较
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // 相邻元素两两比较
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

简单堆排序:

首先将无序数组构建成为一个小顶锥（大顶锥），后将堆顶元素与数组末尾元素互换，然后对堆进行调整，使其满足堆的性质，重复该过程，最终数组就会按照大小排列

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int arr[], int n, int i) {
    int largest = i; // 初始化最大元素的索引为根索引
    int left = 2 * i + 1; // 左子节点的索引
    int right = 2 * i + 2; // 右子节点的索引

    // 如果左子节点大于根节点，则更新最大元素的索引
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // 如果右子节点大于当前最大元素，则更新最大元素的索引
    if (right < n && arr[right] > arr[largest])
        largest = right;
```

```

// 如果最大元素的索引不是根索引，则交换它们，并继续堆化子树
if (largest != i) {
    swap(&arr[i], &arr[largest]);
    heapify(arr, n, largest);
}
}

void heapSort(int arr[], int n) {
    // 构建堆（从最后一个非叶子节点开始，到根节点）
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // 从堆中提取元素并排序
    for (int i = n - 1; i >= 0; i--) {
        // 将当前根（最大值）与最后一个元素交换
        swap(&arr[0], &arr[i]);
        // 堆化从根到新的末尾元素的子树
        heapify(arr, i, 0);
    }
}

void swapFloat(float *a, float *b) {
    float temp = *a;
    *a = *b;
    *b = temp;
}

void heapifyFloat(float arr[], int n, int i) {
    int largest = i; // 初始化最大元素的索引为根索引
    int left = 2 * i + 1; // 左子节点的索引
    int right = 2 * i + 2; // 右子节点的索引

    // 如果左子节点大于根节点，则更新最大元素的索引
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // 如果右子节点大于当前最大元素，则更新最大元素的索引
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // 如果最大元素的索引不是根索引，则交换它们，并继续堆化子树
    if (largest != i) {
        swapFloat(&arr[i], &arr[largest]);
        heapifyFloat(arr, n, largest);
    }
}

void heapSortFloat(float arr[], int n) {
    // 构建堆（从最后一个非叶子节点开始，到根节点）
    for (int i = n / 2 - 1; i >= 0; i--)
        heapifyFloat(arr, n, i);

    // 从堆中提取元素并排序
    for (int i = n - 1; i >= 0; i--) {
        // 将当前根（最大值）与最后一个元素交换
        swapFloat(&arr[0], &arr[i]);
    }
}

```

```

        // 堆化从根到新的末尾元素的子树
        heapifyFloat(arr, i, 0);
    }
}

```

斐波纳契堆排序:

斐波那契堆是一种用于实现优先队列的数据结构，此处以数组元素构建斐波那契堆，并通过操作生成的斐波那契堆进行排序。由于斐波那契堆的复杂性以及本人能力的局限性，无法对其进行更加详细的阐述

```

typedef struct Node {
    int key;
    struct Node *parent, *child, *left, *right;
    int degree; // 子节点的数量
    int mark; // 是否有子节点被删除过
} Node;

typedef struct FibonacciHeap {
    Node *min;
    Node *root;
    int size;
} FibonacciHeap;

// 创建一个新的节点
Node* createNode(int key) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->key = key;
    node->parent = node->child = node->left = node->right = NULL;
    node->degree = 0;
    node->mark = 0;
    return node;
}

// 创建一个新的斐波那契堆
FibonacciHeap* createFibonacciHeap() {
    FibonacciHeap *heap = (FibonacciHeap *)malloc(sizeof(FibonacciHeap));
    heap->min = NULL;
    heap->root = NULL;
    heap->size = 0;
    return heap;
}

// 插入节点到斐波那契堆
void insert(FibonacciHeap *heap, Node *node) {
    if (heap->min == NULL) {
        heap->min = node;
        node->left = node->right = node;
    } else {
        node->left = heap->min;
        node->right = heap->min->right;
        heap->min->right->left = node;
        heap->min->right = node;
    }
    heap->size++;
}

```

```
}
```

```
// 提取最小元素
```

```
Node* extractMin(FibonacciHeap *heap) {
    Node *z = heap->min;
    if (z != NULL) {
        if (z->child != NULL) {
            Node *child = z->child;
            while (child != NULL) {
                Node *brother = child->right;
                child->parent = NULL;
                insert(heap, child);
                child = brother;
            }
        }
        if (z->left == z) {
            heap->min = NULL;
        } else {
            heap->min = z->right;
            z->left->right = z->right;
            z->right->left = z->left;
        }
        if (heap->root != NULL) {
            Node *y = heap->root;
            while (y != NULL) {
                if (y != z && y->mark == 1) {
                    Node *x = y->child;
                    while (x != NULL) {
                        x->parent = NULL;
                        insert(heap, x);
                        x = x->right;
                    }
                    y->child = NULL;
                    y->mark = 0;
                }
                Node *brother = y->right;
                if (y->degree < heap->min->degree) {
                    heap->min = y;
                }
                y = brother;
            }
        }
        heap->root = NULL;
        heap->size--;
    }
    return z;
}
```

```
// 斐波那契堆的合并
```

```
void merge(FibonacciHeap *heap1, FibonacciHeap *heap2) {
    if (heap1->min == NULL) {
        *heap1 = *heap2;
        free(heap2);
    } else if (heap2->min != NULL) {
        insert(heap1, heap2->min);
        if (heap2->min->key < heap1->min->key) {
```

```

        heap1->min = heap2->min;
    }
    free(heap2);
}
}

// 释放斐波那契堆
void freeFibonacciHeap(FibonacciHeap *heap) {
    Node *x, *y;
    if (heap->min != NULL) {
        x = heap->min;
        while (x != NULL) {
            y = x->right;
            free(x);
            x = y;
        }
    }
    free(heap);
}

```

(2)算法测试

首先编写一份随机数生成器以用于生成实验数据

```

int randomArray[ARRAY_SIZE];
int i;

// 使用114514作为随机数生成的种子
srand(114514);

// 生成随机数组
for (i = 0; i < ARRAY_SIZE; i++) {
    randomArray[i] = MIN_VALUE + rand() % (MAX_VALUE - MIN_VALUE + 1);
}

```

分别使用三种算法对生成的数组进行排序，发现算法完全正常

不同编译优化等级下的性能对比结果 数据可视化部分

5.6.编译,性能测试与数据收集可视化

使用GNU项目自带的 `gprof` 命令进行性能测试

(1)增加编译选项

使用终端为算法添加编译选项，并保证其能输出 `gmon.out` 文件，获取算法的资源占用情况

(2)收集实验数据

将收集的 `gmon.out` 文件转化为 `.prof` 文件，收集其中数据，并生成 `csv` 格式文件、

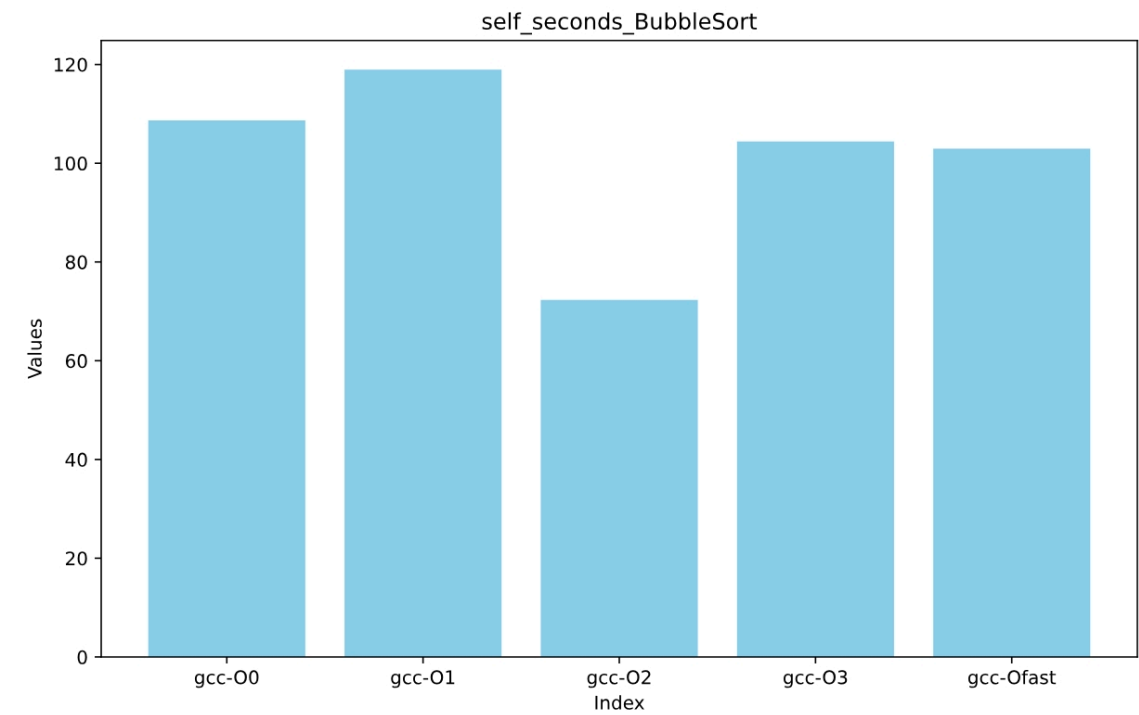
Time (%)	Cumulative Seconds	Self Seconds	Calls	Self s/call	Total s/call	Function Name
100	108.66	108.66	1	108.66	108.66	bubbleSort
0.00	115.47	0.00	1	0.00	0.00	createNode
21.97	115.47	25.37	0.00	0.00	0.00	createFibHeapSort
78.03	90.10					insert
50	0.02	0.02	300000	0.00	0.00	heapify
25	0.03	0.01	300000	0.00	0.00	heapifyFloat
12.5	0.04	0.01	3350234	0.00	0.00	swapFloat
12.5	0.04	0.01	1	5.00	25.00	heapsort
0.00	0.04	0.00	3350039	0.00	0.00	swap
0.00	0.04	0.00	1	0.00	15.00	heapSortFloat

(3)实验数据处理与可视化

先行安装 `matplotlib` 库

```
pip install matplotlib
```

使用 `matplotlib` 库绘制svg矢量图.如下:



参与测试的指标为:

容量为200000的int型数组样本

3次重复的 `gprof` 运行采样

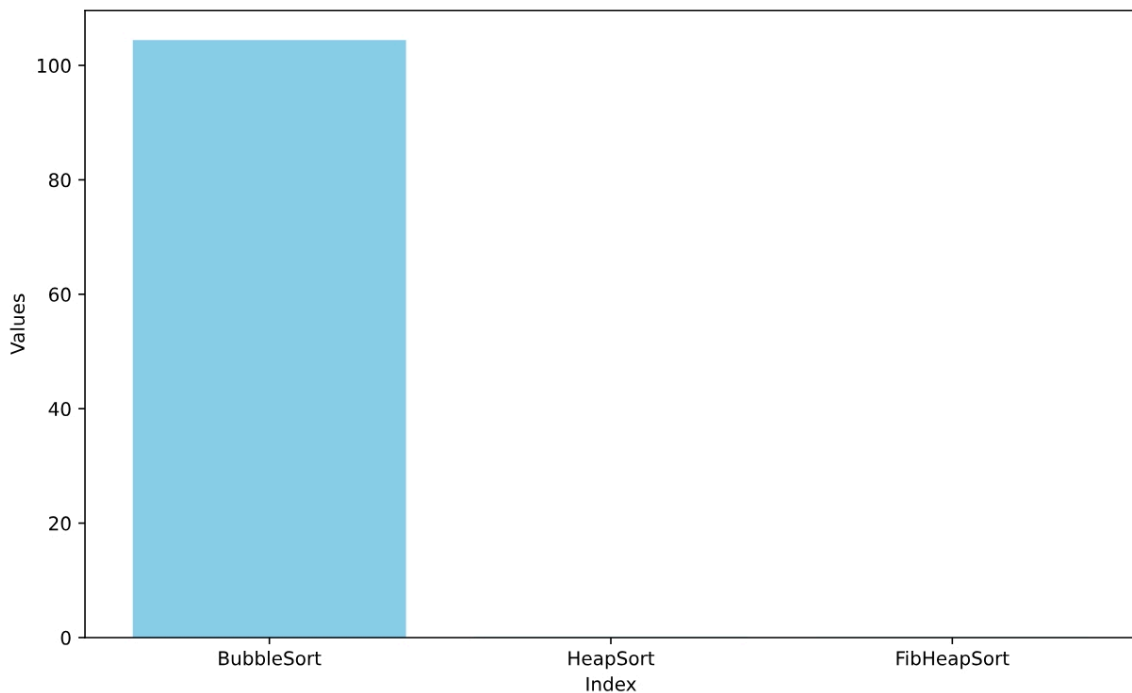
算法的时间复杂度分析

对于冒泡排序：最佳情况：当输入的数列已经是有序的，冒泡排序只需要遍历一次数列，即可完成排序。因此，最佳情况下的时间复杂度是 $O(n)$ ，其中 n 是数列的长度。

平均情况和最坏情况：对于平均情况和最坏情况，冒泡排序需要进行 $n-1$ 轮比较，每轮比较都要遍历整个数列。在每轮中，如果当前元素比它后面的元素大，则需要交换，这意味着下一轮可以少比较一次。因此，总的比较次数大约是 $n/2 + (n-2)/2 + \dots + 1/2$ ，这个和可以用等差数列求和公式计算，结果是 $(n-1)n/2$ 。所以平均情况和最坏情况下的时间复杂度是 $O(n^2)$ 。

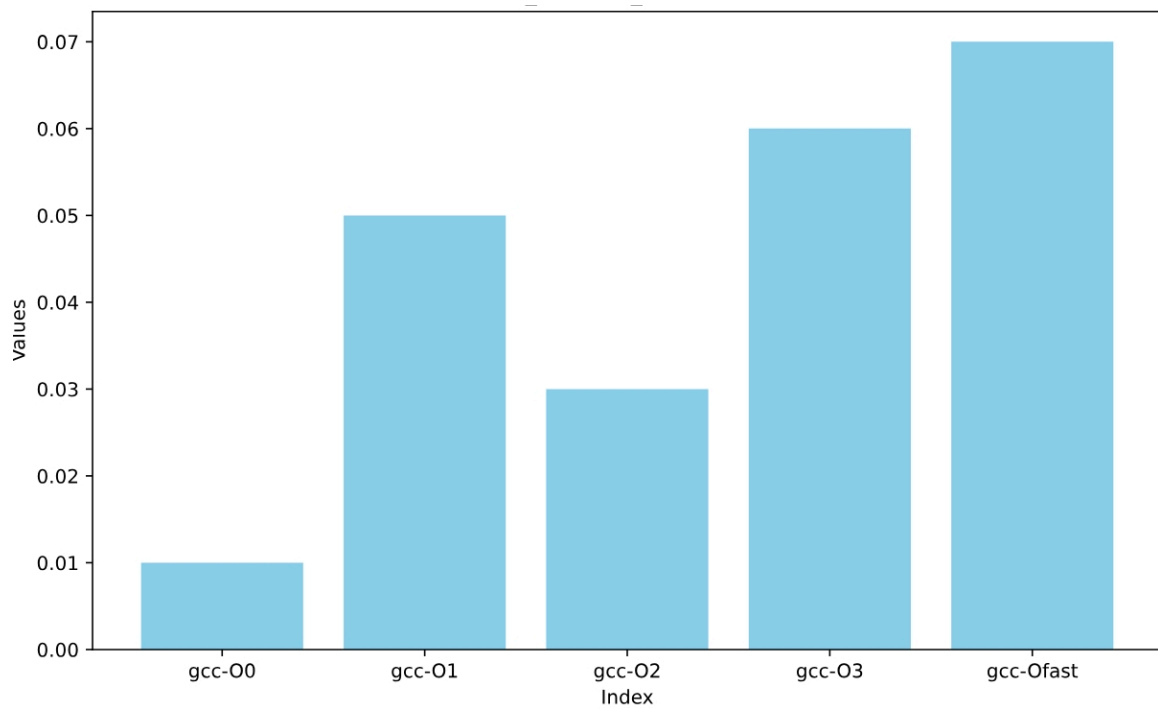
空间复杂度：

在 $O(3)$ 级别的横向对比中可以看到冒泡排序在自身运行时间上远远超过另外两个算法：



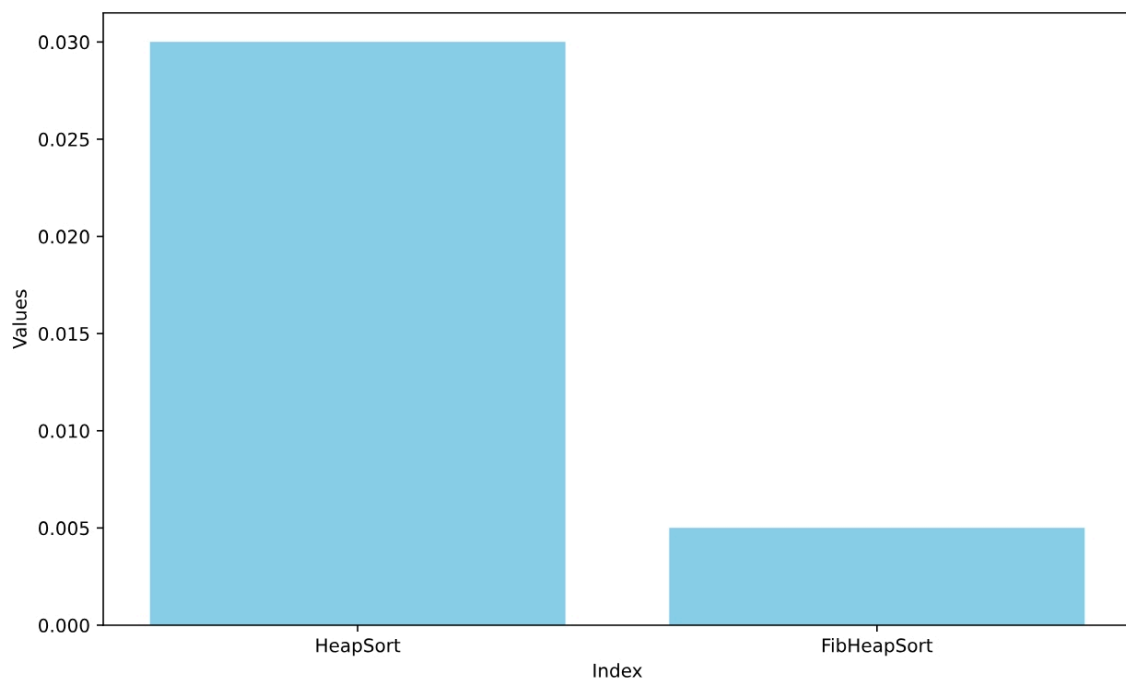
对于简单堆排序，首先需要完成构建堆的过程，对于一个有 n 个元素的数组，构建堆的时间复杂度是 $O(n)$ 。在完成堆的构建后通过反复移除堆顶元素，然后重新调整堆来完成排序，每次该操作的时间复杂度为 $O(\log n)$ ，共计完成 n 次操作，故时间复杂度为 $O(n \log n)$ ，

从多个级别的编译优化中可看出其高效性：



对于斐波那契堆排序，插入操作、合并操作、减小键值复杂度都为 $O(1)$ ，而提取最小元素和删除节点的时间复杂度为 $O(\log n)$ ，

参看O2级别的算法对比：

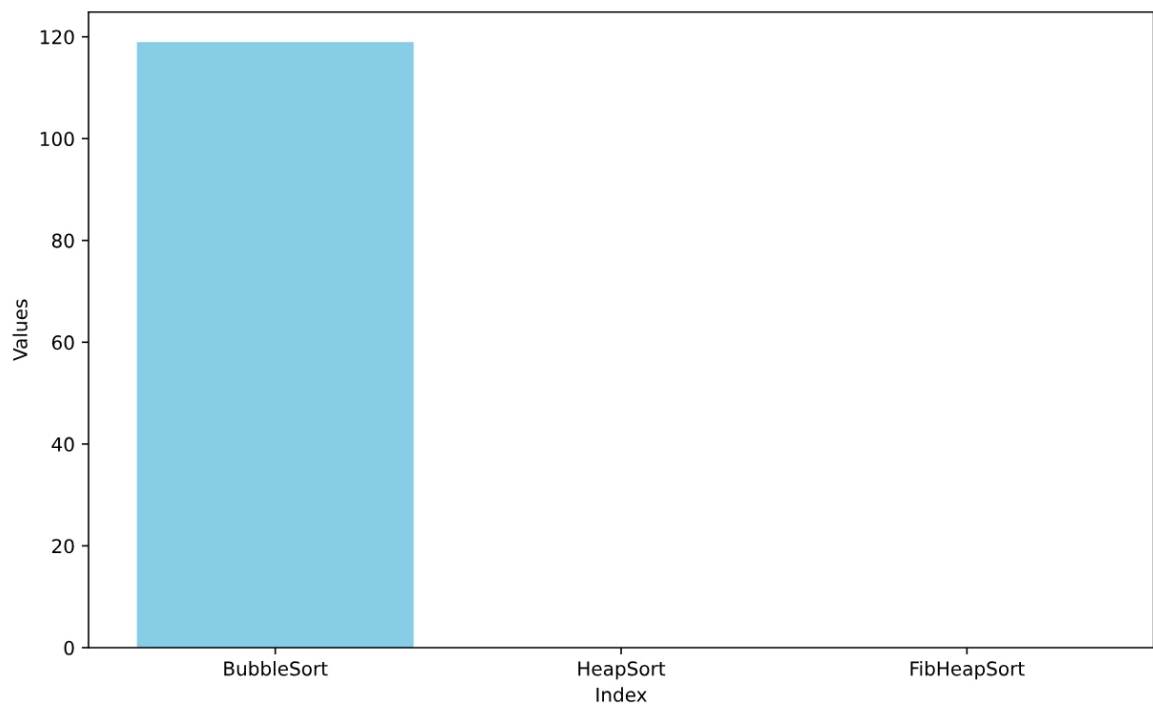


(由于冒泡排序执行时间过长故不在此比较)

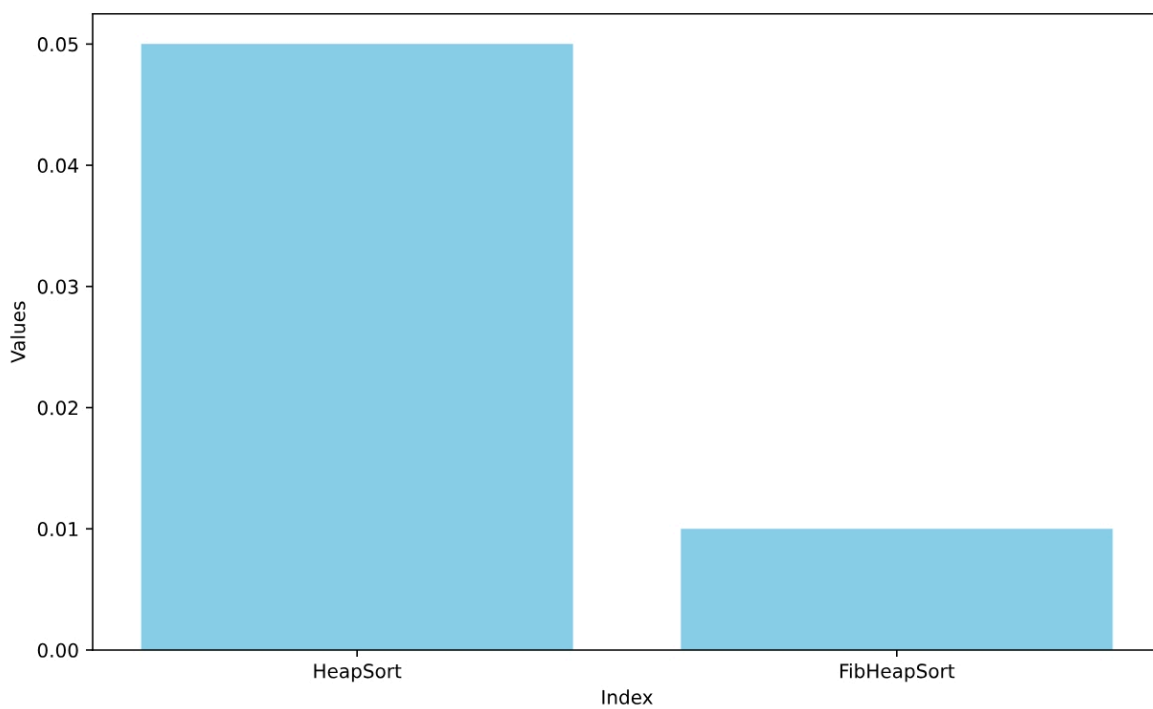
同一优化下不同算法对比

统一在O1的优化级别下进行对比.

运行时间对比,即算法速度(性能):

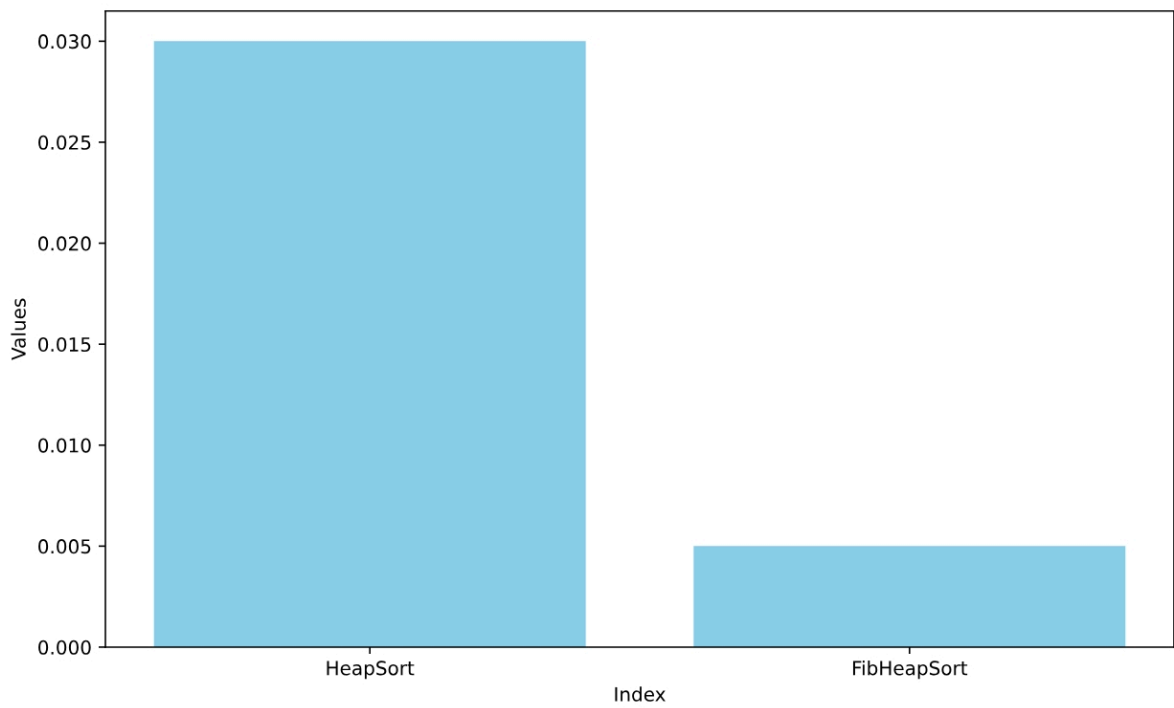
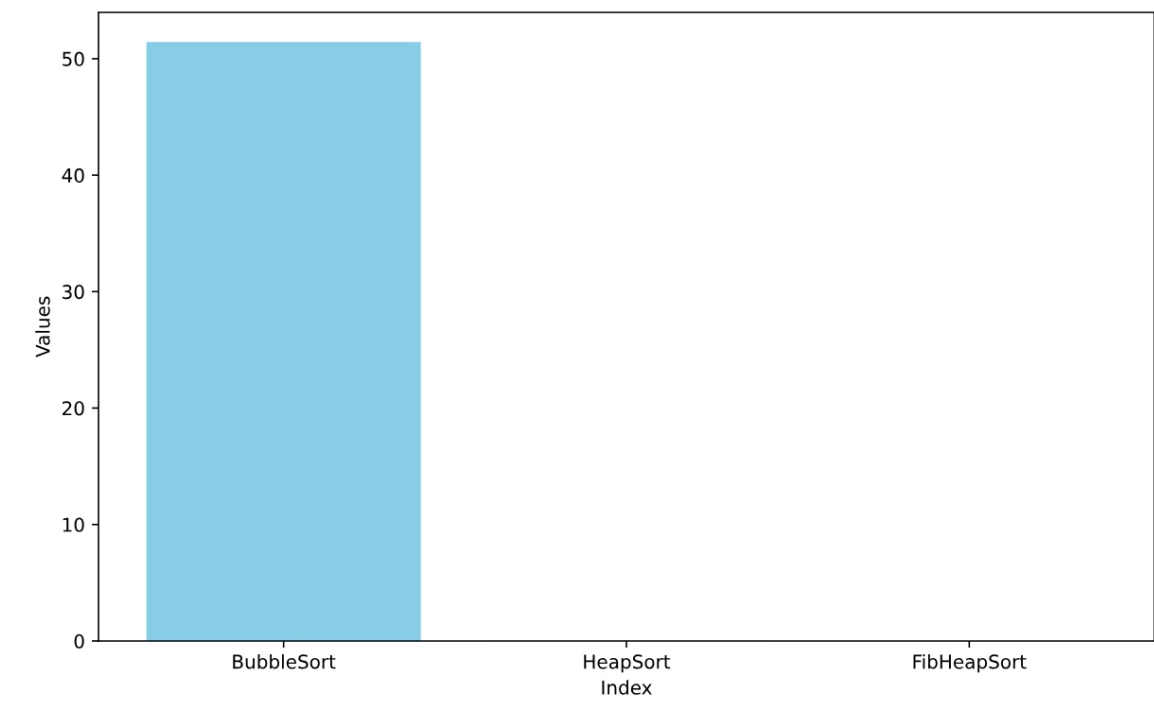


(由于冒泡排序与另外两种算法时间差异过大，现将简单堆排序和斐波那契堆排序单独对比，下同)



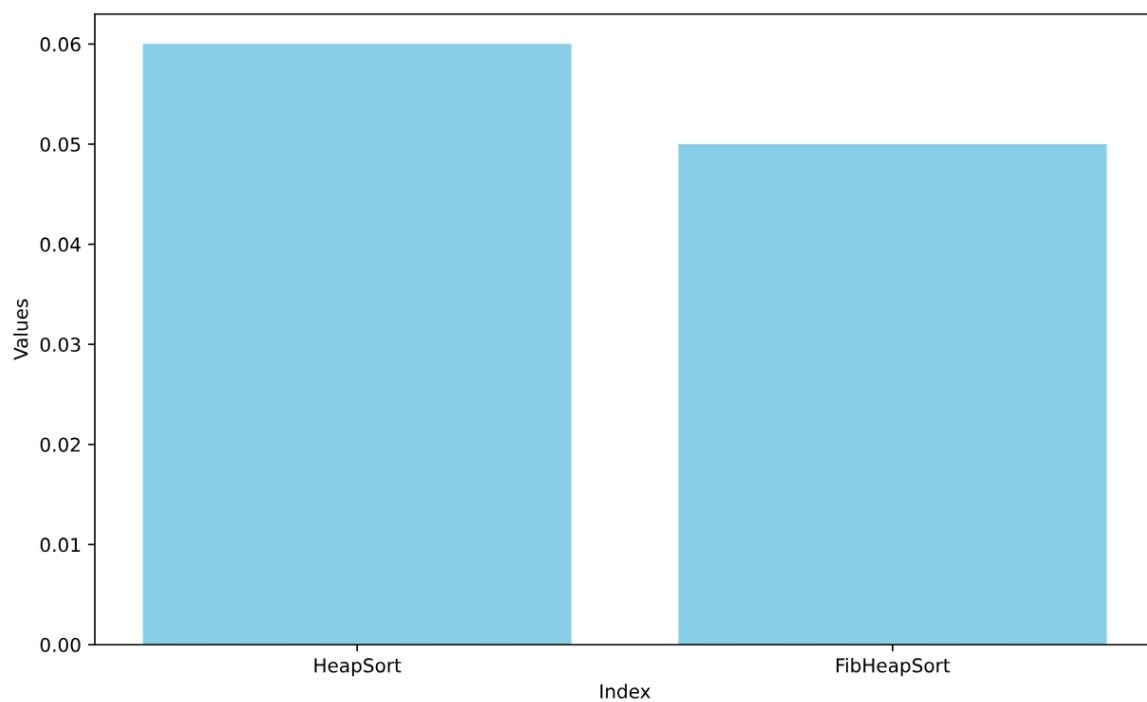
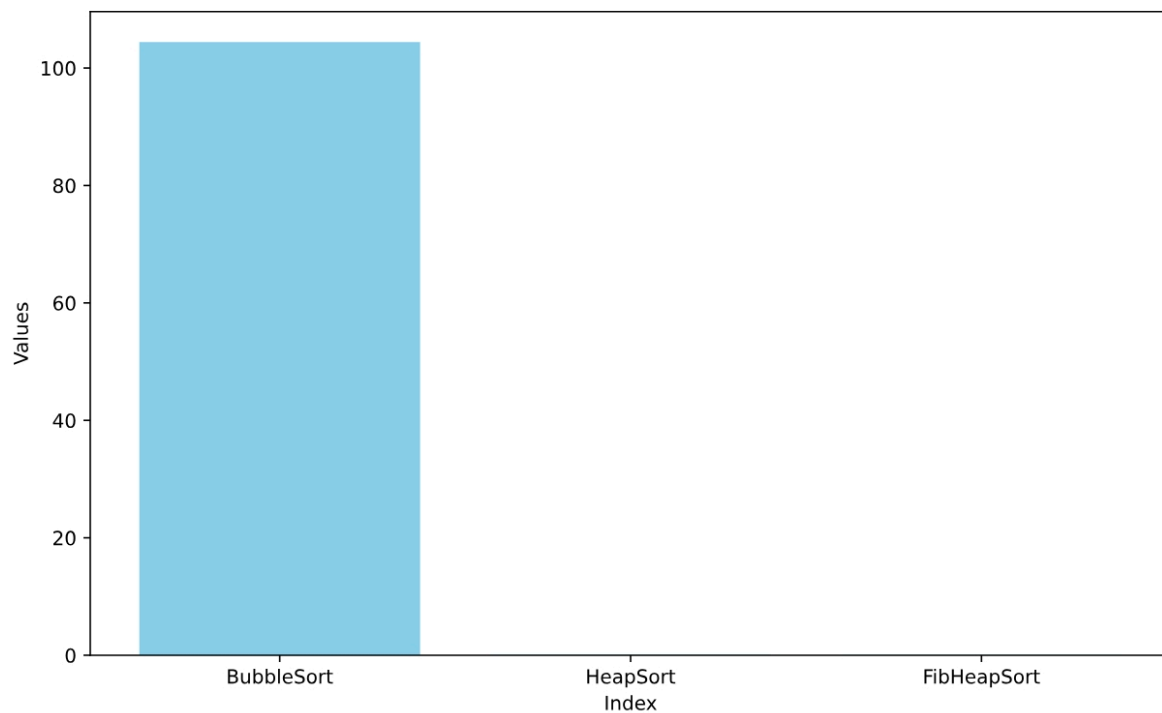
统一在O2的优化级别下进行对比.

运行时间对比,即算法速度(性能):



统一在O3的优化级别下进行对比.

运行时间对比,即算法速度(性能):

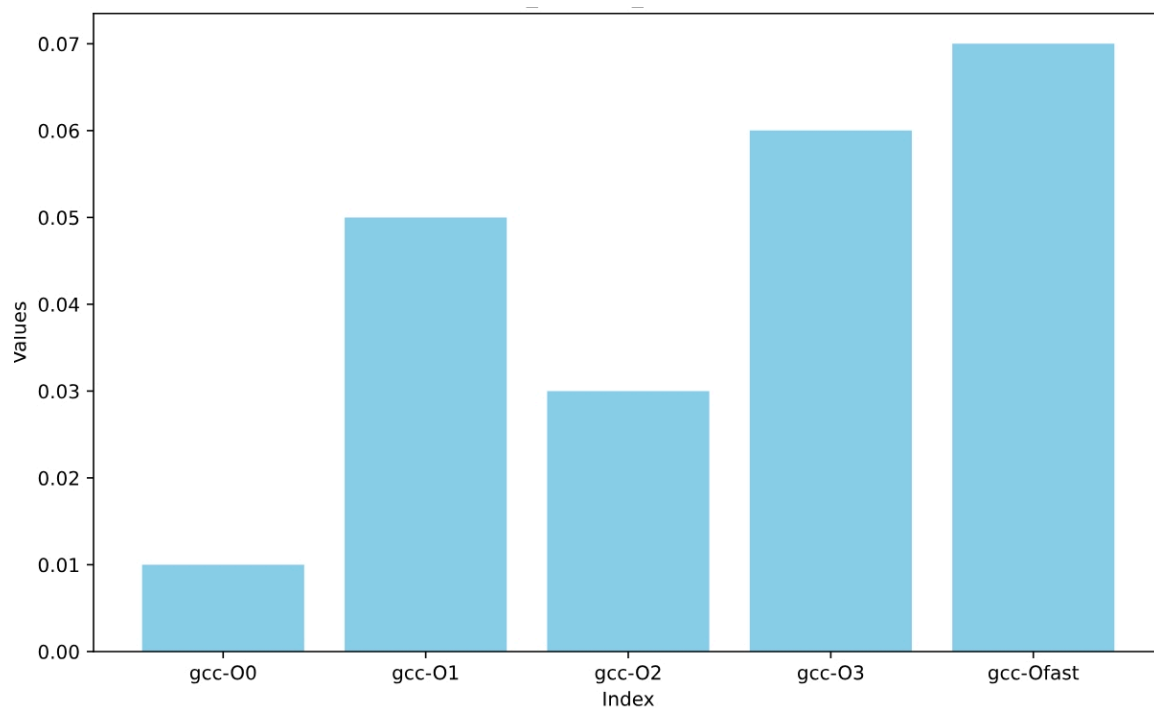


由上述对比可看出冒泡排序的低效性，无论在何种编译优化等级下均与其他两种算法的编译时长有指数级的差异，而另外两种算法的性能较为相近，在测试中可能由于生成数组的影响，使得斐波那契堆排序的性能优于简单堆排序

不同优化等级对同一算法性能的影响

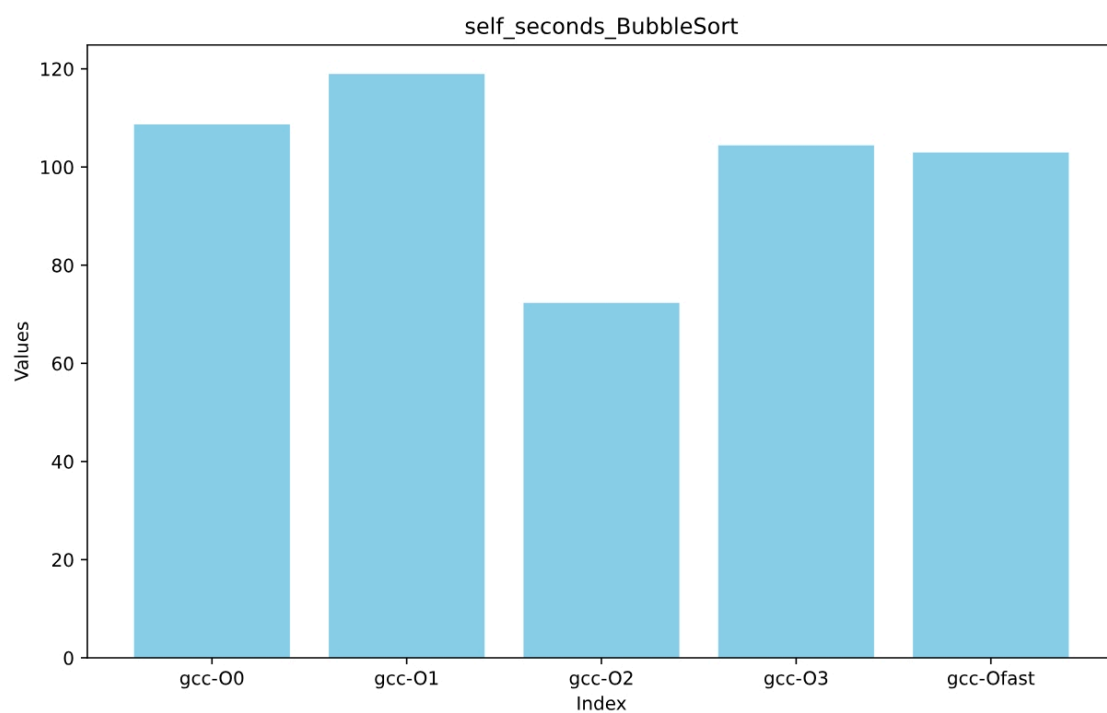
这里先对比简单堆排序.

来看运行时间对比:



可以看到简单堆排序的高效性，在不同编译优化选项下的差异

对于冒泡排序



无论何种优化等级下均占用较多资源，使用较长时间

同时对比斐波那契堆排序算法的优化差异,可以看到在极大量的数据下,不同的优化选项更多体现在不同算法对资源的占用上,而处理速度则近似相同.

实验过程中遇到的问题及解决方案

1.实验部分

（1）实验数据收集：本想模仿howthon使用脚本收集实验数据并进行处理，但在使用ai工具生成脚本后发现处理结果不如人意，在多次尝试修改无果后心力交瘁，最终放弃使用脚本，转而通过人力收集、处理实验数据。

（2）还是实验数据收集：可能是由于数据收集工具的精度不足而程序运行时间过短导致收集到的数据全显示0，反复执行程序后收集到实验数据，可能会产生误差

特别鸣谢



浩神[HowXu](#)