

排序算法编程实践

23090032047 计算机类1班 于景一

写在前面：

本文所有插图通过jsDelivr分发的GitHub Repo存储，未免有可用性問題。如无法正确显示，请临时查看我的“备用_Test6_23090032047_于景一.md”文件，该文件的图片均使用 base64 存储。谢谢老师！

1. 实践思路

本次课程总体来说有两个要点，一是“使用Markdown”编写实验报告，二是“冒泡排序和快速排序的算法实现”。由于在第四课中我们已经研究了“如何配置Python环境”，故在本次实验中不加赘述。

1.1. Markdown 应用

1.1.1. 基本语法

介绍Markdown中几个基础的语法：

```
标题：
# Heading 1
## Heading 2
### Heading 3
#### Heading 4 （以此类推）

> Quote
- List

文本类型
`Code` *Italic* **Bold** ***Italic & Bold***
[Text](https://www.example.com)
[Text][1]
![Pic](link)

表格：
| 表头 | 表头 |
| ---- | ---- |
| 单元格 | 单元格 |
| 单元格 | 单元格 |

特别的，在分割符之间通过添加“:”来设置内容和标题栏的对齐方式。
    -: 居右对齐      :- 居左对齐      :-: 居中对齐

转义：
\* \[ \#
```

HTML 标签：

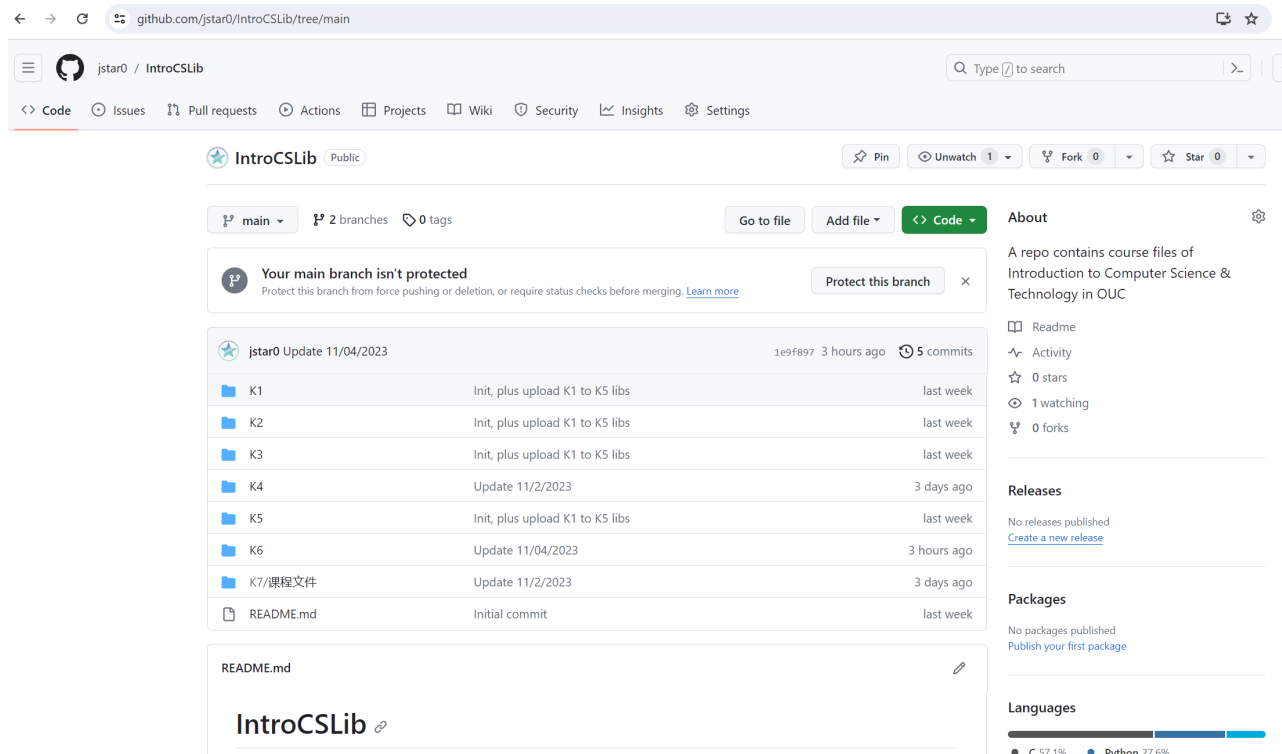
<kbd> <i> <sup> <sub>

[1]: <https://www.example.com>

1.1.2. 版本控制

1. 在GitHub上新建一个Repo

没有可以过多描述的，我新建的是面向Public的私人Repo，命名为IntroCSLib，您可以[从这里](#)查看。



图为我创建的Repo

2. 通过Git管理

Windows环境下我们通过[Git for Windows](#)进行配置，在此不加赘述。

Git有许多基础指令。通过在本地初始化一个Repo并提交First Commit来熟悉基本指令。

首先我们设置好面向commit的信息。

```
git config --global user.name "King Star"
git config --global user.email mcxin.y@gmail.com
```

注意：**--global**意味着在git全局配置此账号。如需对本项目单独配置，应将其移除。

接着，我们克隆远程仓库，使用

```
git clone https://github.com/USER_NAME/REPO_NAME.git
```

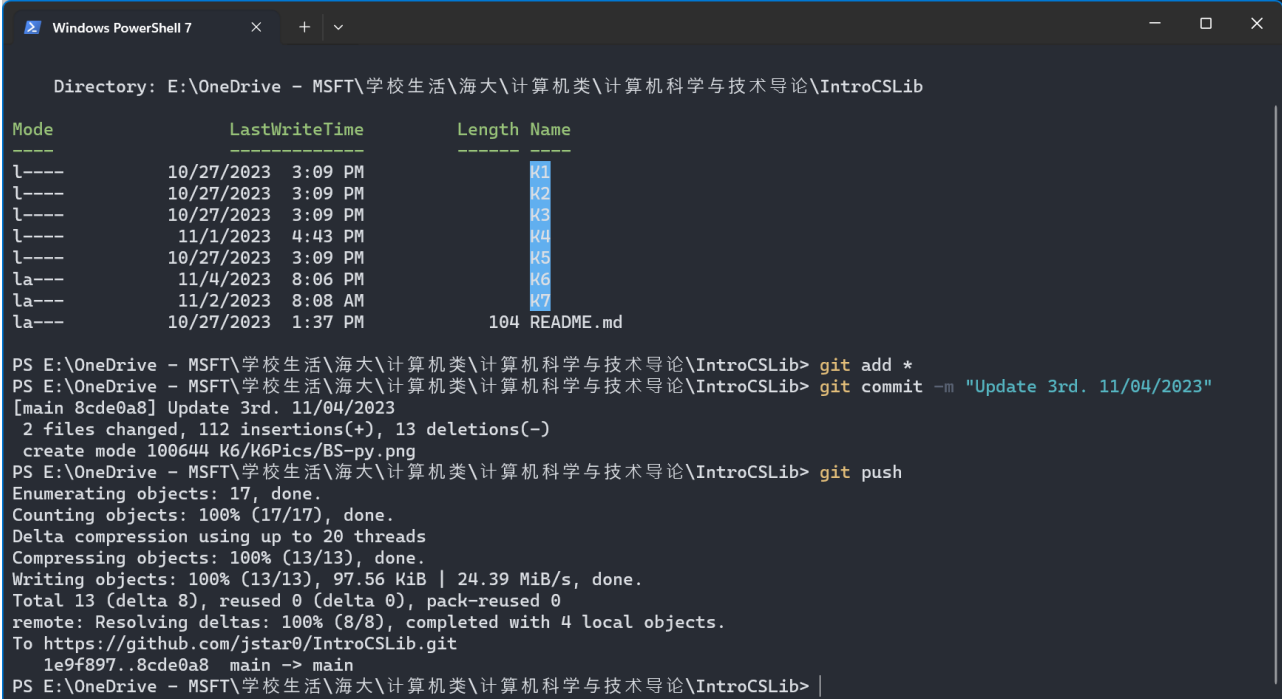
此后，在本地目录即可进行项目迭代。对每一次需要提交上传代码的情况，都可以使用

```
git add FILE_NAME //上传指定更改的文件，或使用*通配符以上传所有更改
git commit -m "PUT COMMIT MESSAGE HERE" //标记commit
git push //提交更改
```

对于已更改的Repo，如果要与远端同步，拉取命令为

```
git pull
```

以上是基础的利用Git对远程Repo的管理命令。



The screenshot shows a Windows PowerShell 7 terminal window with the following content:

```
Directory: E:\OneDrive - MSFT\学校生活\海大\计算机类\计算机科学与技术导论\IntroCSLib
```

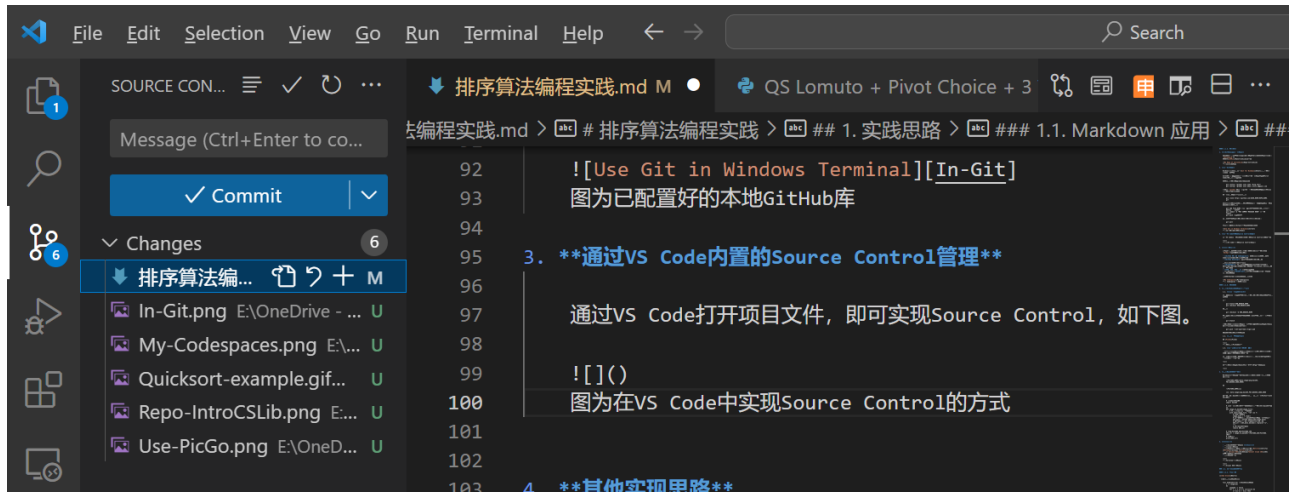
Mode	LastWriteTime	Length	Name
l----	10/27/2023 3:09 PM		x1
l----	10/27/2023 3:09 PM		x2
l----	10/27/2023 3:09 PM		x3
l----	11/1/2023 4:43 PM		x4
l----	10/27/2023 3:09 PM		x5
la---	11/4/2023 8:06 PM		x6
la---	11/2/2023 8:08 AM		x7
la---	10/27/2023 1:37 PM	104	README.md

```
PS E:\OneDrive - MSFT\学校生活\海大\计算机类\计算机科学与技术导论\IntroCSLib> git add *
PS E:\OneDrive - MSFT\学校生活\海大\计算机类\计算机科学与技术导论\IntroCSLib> git commit -m "Update 3rd. 11/04/2023"
[main 8cde0a8] Update 3rd. 11/04/2023
 2 files changed, 112 insertions(+), 13 deletions(-)
 create mode 100644 K6/K6Pics/BS-py.png
PS E:\OneDrive - MSFT\学校生活\海大\计算机类\计算机科学与技术导论\IntroCSLib> git push
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 20 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (13/13), 97.56 KiB | 24.39 MiB/s, done.
Total 13 (delta 8), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (8/8), completed with 4 local objects.
To https://github.com/jstar0/IntroCSLib.git
 1e9f897..8cde0a8  main -> main
PS E:\OneDrive - MSFT\学校生活\海大\计算机类\计算机科学与技术导论\IntroCSLib>
```

图为已配置好的本地GitHub库

3. 通过VS Code内置的Source Control管理

通过VS Code打开项目文件，即可实现Source Control，如下图。



图为在VS Code中实现Source Control的方式

4. 其他实现思路

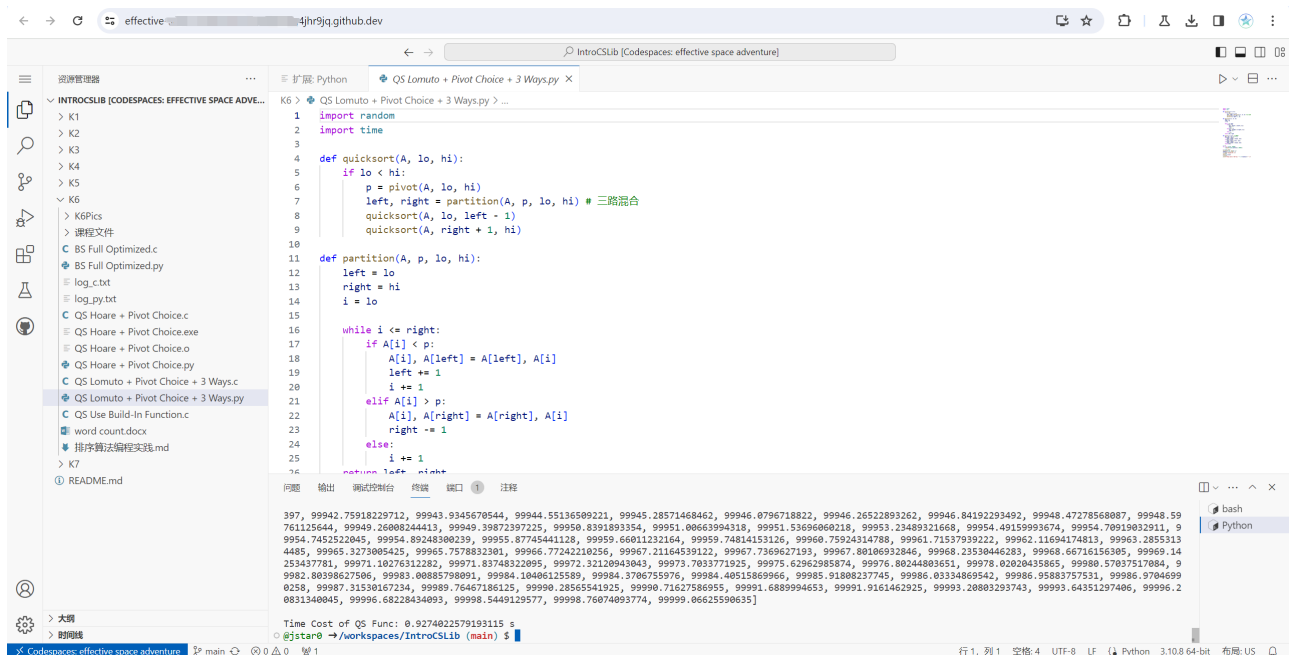
在PaaS流行的时代，有更多途径实现版本控制，形成完整的DevOps，提高生产力。譬如：

- **GitHub 推出的 Codespaces**: 允许创建在云端运行的VS Code环境，解除本地环境的限制。
- **GitHub Desktop**: 提供带GUI的版本控制面板。

也有类似Git的版本控制程序：

- **BitKeeper**: 早期Linux团队管理Linux内核程序码的软件，它的授权制度导致了Git的出现。不过，在Git发行后，它也成为开源软件。
- **SVN, CVS, RCS...** 在此不做深究。
- 类似的，其实**OneDrive**也自带版本管理功能，不过并不适用于项目迭代。

总体来说Git仍是当下情况的不二选择。



图为 Codespaces 实际运用展示

1.1.3. 档案存取

1. 使用GitHub托管的Repo作为图床

1.1. 创建一个新的branch

我们需要创建一个新的分支，从而实现轻量部署，支持CDN的分发和展现。

通过

```
git branch NEW_BRANCH_NAME  
git checkout NEW_BRANCH_NAME
```

或使用

```
git checkout -b NEW_BRANCH_NAME
```

这些指令用来创建并切换分支。欲列出已有分支和目前所在分支，使用

```
git branch
```

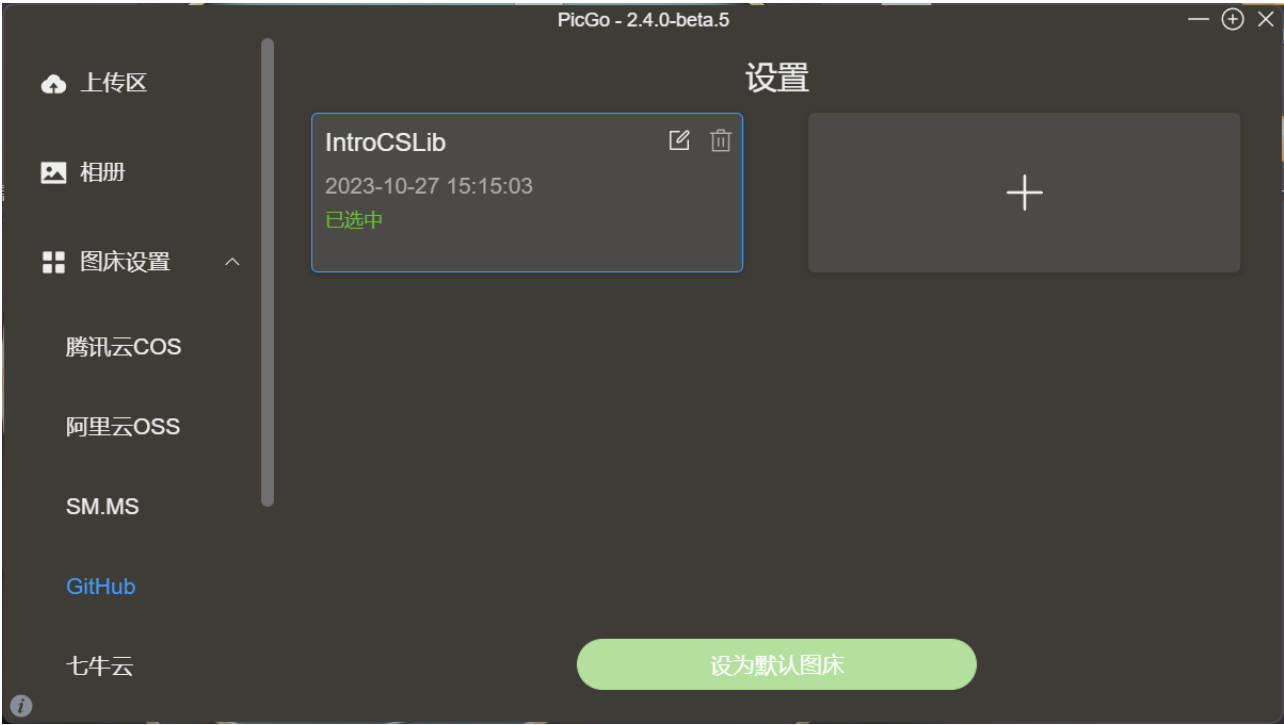
值得注意的是，若从本地新建一个分支，需要正确对接到GitHub上的远程仓库。完整的命令应该是

```
git push --set-upstream origin pub
```

如此可以正确地关联并推送。

1.2. 使用开源管理工具

如 [PicGo](#)



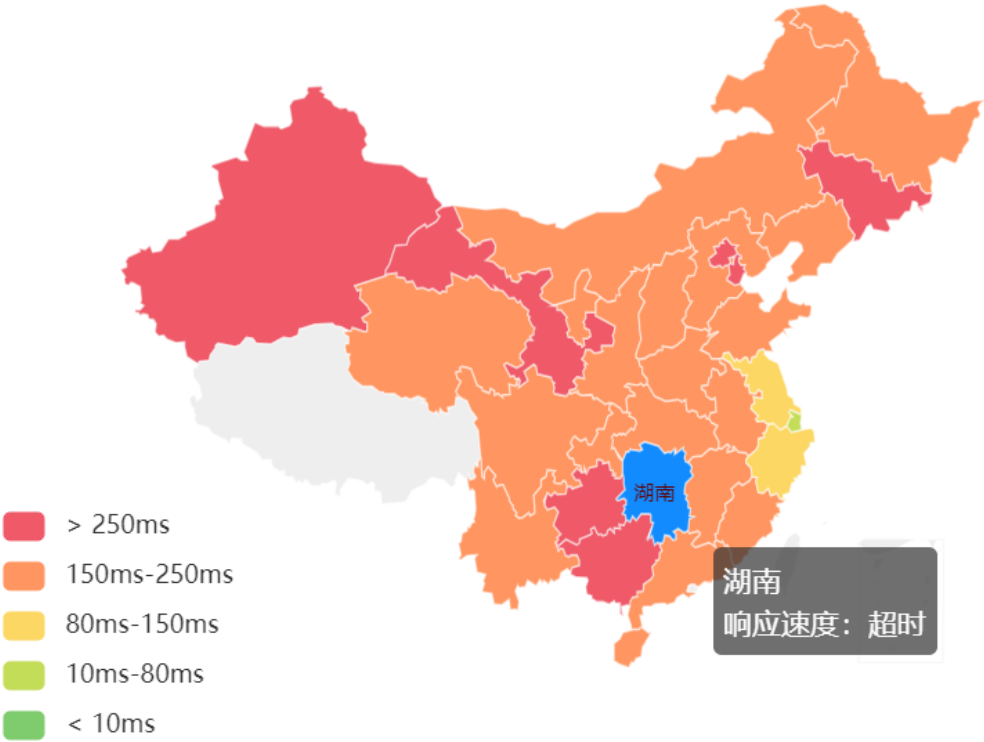
图为正确使用PicGo的截图

1.3. 通过jsDelivr等实现CDN加速

一个事实是，jsDelivr默认镜像并不能在中国很好地工作，有很多区域是无法访问成功的，如下图。

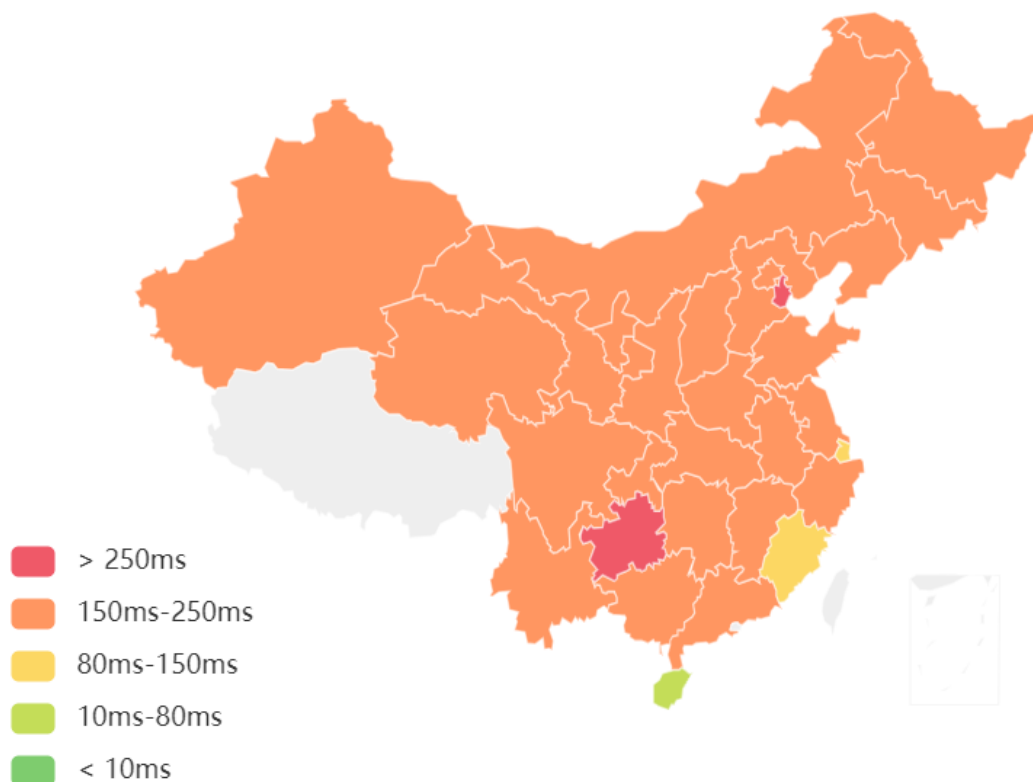
由于jsDelivr有很多子节点，在此选用基于Gcore的以提高可用性，其可用性见下图。

PING检测结果 cdn.jsdelivr.net 2023-10-27 17:13:29



图为jsDelivr的默认节点，有很多超时情况

PING检测结果 gcore.jsdelivr.net 2023-10-27 17:14:27



图为jsDelivr的Gcore节点，基本无超时情况

****特别注意：**这个方法并非永久有效，所以需要使用如状态探针等手段监控其可用性。****不过，**jsDelivr作为全球性的CDN提供商，可用性有一定的保障。

通过 https://gcore.jsdelivr.net/gh/GITHUB_USER_ID/REPO/FILE_PATH/FILE 这种格式的链接，可以正确连接到jsDelivr加速的GitHub Repo并通过HTTP GET到图片。

2. 使用Base64存储图片

Markdown本身支持将图片以base64格式存储在文件中，使用这样的格式：

```
![PICTURE_NAME](data:image/png;base64,PUT_BASE64_CODE_HERE)
```

或

```
![PICTURE_NAME][1]
```

```
[1]: data:image/png;base64,PUT_BASE64_CODE_HERE
```

将图片转换为base64格式的问题，我们简单使用一个Python程序解决，如：

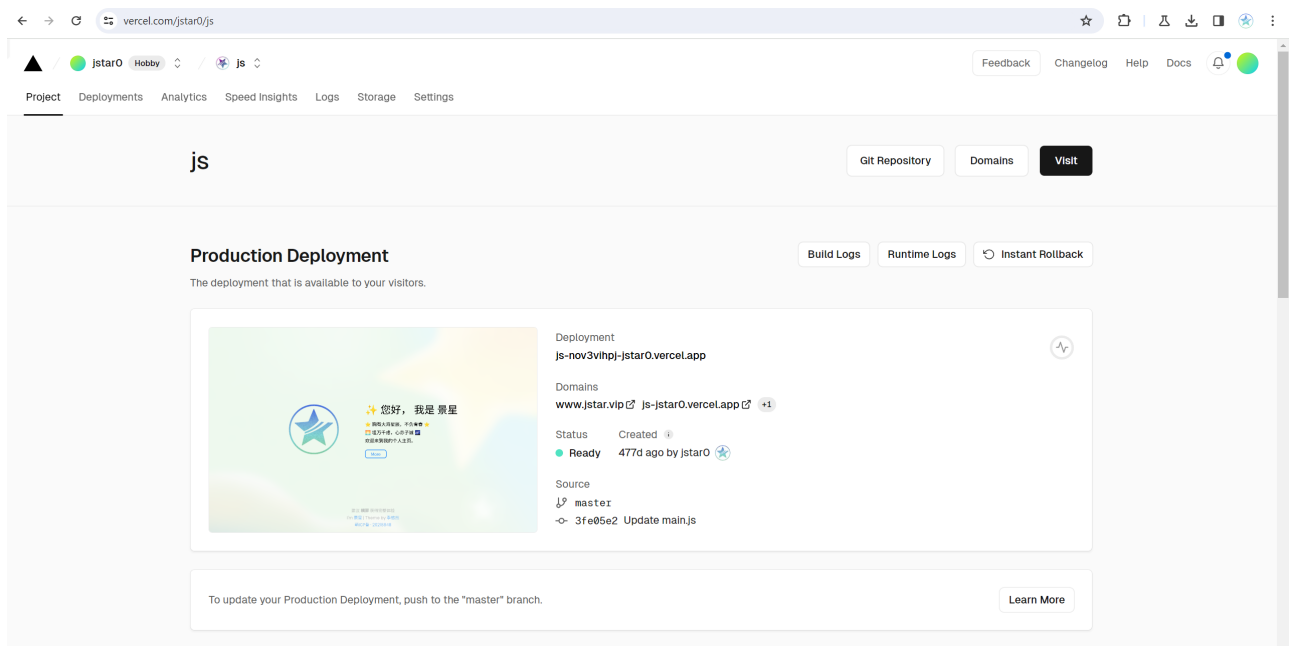
```
# 导入base64模块
import base64

# 定义一个函数，接受图片文件名作为参数，返回base64字符串
def image_to_base64(image_file):
    # 以二进制模式打开图片文件
    with open(image_file, "rb") as f:
        # 读取图片数据
        image_data = f.read()
        # 对图片数据进行base64编码，得到一个字节对象
        b64_data = base64.b64encode(image_data)
        # 将字节对象转换为字符串，去掉换行符
        b64_str = b64_data.decode().replace("\n", "")
        # 返回base64字符串
        return b64_str

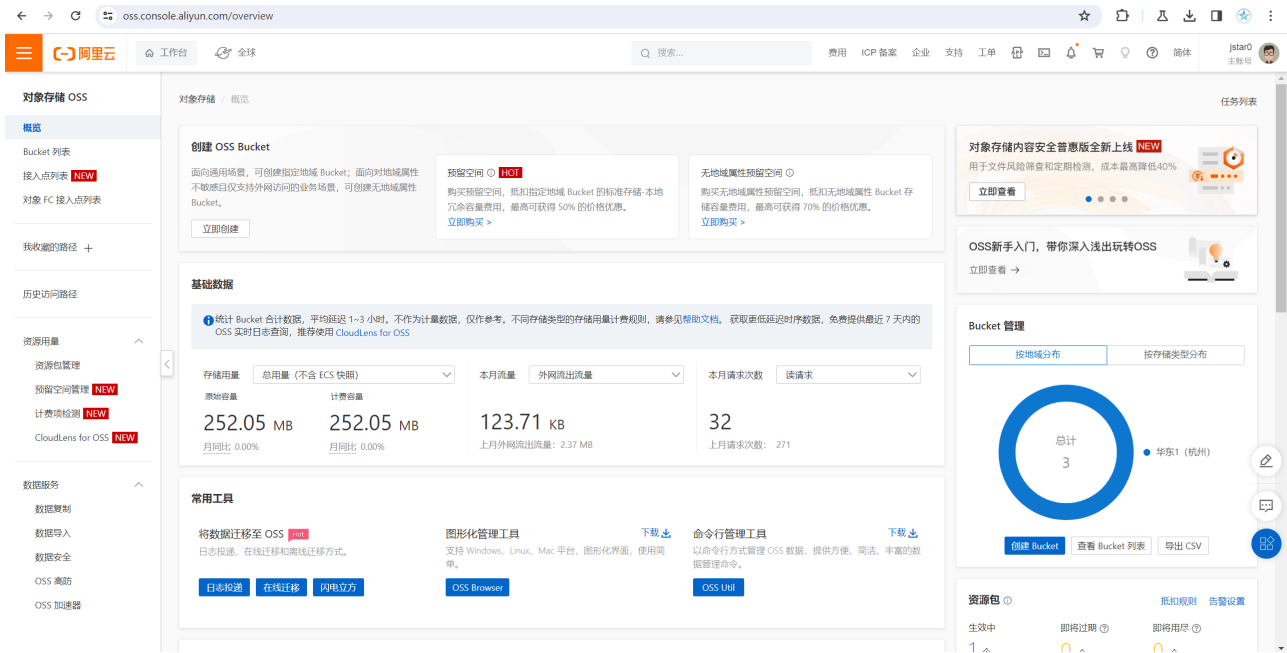
# 替换/PICTURE_DIR/PICTURE.JPG
b64_str = image_to_base64("/PICTURE_DIR/PICTURE.JPG")
# 打印结果
print(b64_str)
```

3. 其他思路

- 保存图片在本地，使用相对路径引用
- 使用PaaS应用实现持续部署，典型如 [Netlify](#) [Railway](#) [Vercel](#)
- 使用 [Aliyun OSS](#)、[Tencent Cloud COS](#) 等服务商的对象存储管理
- 使用可信的图床



图为Vercel持续部署的[我的个人主页](#)展示



图为Aliyun OSS实战展示

1.2. 冒泡排序的核心算法

6 5 3 1 8 7 2 4

图：Bubble Sort的实现示意图（来自Wikipedia）

1.2.1. 算法实现

(See Also)

一般的，使用伪代码表示：

```
func bubbleSort(A) //引入待排序数组A
  n := length(A)
  do
    swapped := false
    for i := 1 to n-1 inclusive do
      if A[i-1] > A[i] then
        swap(A[i-1], A[i])
        swapped := true
    while swapped is True
  return
```

观察发现，第 n 次排序会找到第 n 个最大的元素，并将其放到最后的位置。因此，在运行第 n 次时，内循环可以避免查看最后 $n - 1$ 个项目。修改后即：

```
func bubbleSort(A)
  n := length(A)
  do
    swapped := false
    for i := 1 to n-1 inclusive do
      if A[i-1] > A[i] then
        swap(A[i-1], A[i])
        swapped := true
    n := n - 1
  while swapped is True
return
```

更普遍的情况是，在一次交换中，可能会有不止一个元素被放到它们的最终位置。特别是，每次通过后，最后一次交换后的所有元素都已排序，无需再次检查。这样，我们就可以跳过许多元素，在最坏的情况下，比较次数大约能减少50%。

```
func bubbleSort(A)
  n := length(A)
  do
    newn := 0
    for i := 1 to n - 1 inclusive do
      if A[i - 1] > A[i] then
        swap(A[i - 1], A[i])
        newn := i
    n := newn
  while n > 1
return
```

1.2.2 时间复杂度

冒泡排序是一种稳定的算法，其平均时间复杂度为 $O(n^2)$ 。

在数列完全有序时，只需遍历一遍数组而不作交换操作，时间复杂度为 $O(n)$ 。最坏情况下要进行 $(n-1)n/2$ 次交换，时间复杂度为 $O(n^2)$ 。

1.3. 快速排序的核心算法

6 5 3 1 8 7 2 4

图：Quick Sort的实现示意图，使用Hoare的版本（来自[Wikipedia](#)）

1.3.1. 算法组成

1. 算法实现

由于快速排序算法具有多种实现方式，为便于研究，我们将算法分为分区方案 (*Partition Scheme*) 和实现方式 (*Implementation Issues*)，对于后者我们仅讨论其中相当重要的部分：基准的 *选择* (*Choice of pivot*)、重复元素情况 (*Repeated elements*) 及序列较短时用插入排序。

([See Also](#))

快速排序的基本思路是分治。不管是何种具体方法，其基本思路是一致的，即：

1. 判断：如果目标数组元素少于两个，则立即返回，无事可做。

注：对于其他较短的情况，或许可以使用到其他排序方法，因为快速排序的优势主要体现在较大的范围中。

2. 选取基准 (pivot)：取决于具体的分区方案和实现方式。

3. 重新排序：大于基准的在其右，小于在其左，等于者可移入左右任一位置。

4. 递归处理：递归地 (recursively) 处理子数组。

2. 时间复杂度

快速排序是一种不稳定的排序算法。一般的，它的平均时间复杂度是 $O(n \cdot \log n)$ ，最坏时间复杂度是 $O(n^2)$ 。通过优化，我们可以适当的使它得到改进。具体的数学证明略。（其中一种证明见 [OI-Wiki](#)）

1.3.2. 分区方案 (Partition Scheme)

1. Nico Lomuto 的方案

通过*sorting*, *deviding*, *recurring*三个过程，实现这个算法。该方案通常选择数组中的最后一个元素作为基准(pivot)。

排序一个从 *lo* 到 *hi* 的数组 *A* 的伪代码如下：

```
// 给一个 ( 或其中一部分 ) 数组排序 · 分区 · 再排序 。  
func quicksort(A, lo, hi)
```

```

// 确定下标指向正确的位置
if lo >= hi || lo < 0 then
    return

// 给数组分区，得到p作为pivot的项的下标
p := partition(A, lo, hi)

// 给得到的两个分区排序
quicksort(A, lo, p - 1) // Left side of pivot
quicksort(A, p + 1, hi) // Right side of pivot

// 给数组分区
func partition(A, lo, hi)
    pivot := A[hi] // 选择数组中的最后一个元素作为基准(pivot)

    // 定义 Temporary pivot index
    i := lo - 1 // 一般的，如果lo是0，则i的初始值是-1

    for j := lo to hi-1 do // hi-1是避开最后一项即pivot
        // 如果元素小于等于pivot
        if A[j] <= pivot then
            // 移动 Temporary pivot index 向后一位
            i := i + 1
            // 将j所在的项与 Temporary pivot index 所在的项交换
            swap A[i] with A[j]

    // 最后将pivot项放在正确的位置（大子列和小子列之间）
    i := i + 1 // 指向该位置
    swap A[i] with A[hi] // 并不会破坏大小子列的相对有序性
    return i // 返回pivot所在项的下标

```

注意：这种方法效率通常低于 Hoare 的原始方案。

2. Tony Hoare 的方案

Hoare 是快速排序算法的原发表者。由于其排序方式并不如 Lomuto 一般显然。

依然通过 *sorting*, *deviding*, *recursing* 三个过程，实现这个算法。不过，在这里基准(pivot)通常选数组最中间的那一项。

```

// 给一个（或其中一部分）数组排序，分区，再排序。
func quicksort(A, lo, hi)
    if lo >= 0 && hi >= 0 && lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p) // 注意：此时的p已包含在内。
        quicksort(A, p + 1, hi)

// 给数组分区
func partition(A, lo, hi)
    // pivot是数组的中间那一项

```

```

pivot := A[ floor((hi - lo)/2) + lo ]
// 通过这种方式，能避免 hi + lo 过大导致数据溢出的问题。

// 左下标
i := lo - 1

// 右下标
j := hi + 1

loop forever
    // 如果左侧小于基准值，左下标右移至少一位。
    do i := i + 1 while A[i] < pivot

    // 如果右侧大于基准值，右下标左移至少一位。
    do j := j - 1 while A[j] > pivot

    // 如果两个下标交叉或错位，返回pivot所在项
    if i >= j then return j

    // 左右互换
    swap A[i] with A[j]

```

1.3.3. 实现方式 (Implementation Issues)

1. 基准的选择 (Choice of pivot)

我们通过选择随机的基准，或*从前、中、后三个位置的基准取中值（三数取中）*的方案来避免最坏情况的产生。

In the very early versions of quicksort, the **leftmost element of the partition** would often be chosen as the pivot element. Unfortunately, this causes **worst-case behavior on already sorted arrays**, which is a rather common use-case. The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot. (as recommended by Sedgewick)

对 Lomuto 方案的基础上进行三数取中

```

mid := L(lo + hi) / 2
// 或可使用 floor((hi - lo)/2) + lo 避免数值溢出
if A[mid] < A[lo]
    swap A[lo] with A[mid]
if A[hi] < A[lo]
    swap A[lo] with A[hi]
if A[mid] < A[hi]
    swap A[mid] with A[hi]
pivot := A[hi]

```

2. 重复元素情况 (Repeated elements)

即使选择了很好的pivot值，在 Lomuto 的方法中，遇见有许多重复元素时（尤其当所有元素都相同时）快排算法也不会有太优秀的表现。

Hoare 的算法有较好的表现，原因如下：

However, with a partitioning algorithm such as the Hoare partition scheme, repeated elements generally results in better partitioning, and although needless swaps of elements equal to the pivot may occur, the running time generally decreases as the number of repeated elements increases (with memory cache reducing the swap overhead).

通过 三路快速排序（快速排序和基数排序的混合），将数值分为三组：小于pivot的；等于pivot的；大于pivot的。等于者已然排序好，那么只需要排序小于的和大于的。

对 Lomuto 方案如：

```
func quicksort(A, lo, hi)
    if lo < hi then
        p := pivot(A, lo, hi)
        left, right := partition(A, p, lo, hi) // 注意：返回了多个值
        quicksort(A, lo, left - 1)
        quicksort(A, right + 1, hi)
```

3. 当序列较短时，使用 插入排序 的效率更高

由于本课不研究插入排序，与之相关的算法略。

1.4. 记录时间

利用 [第四课](#) 中已经研究好解决方案，我们直接利用其代码即可。具体如下：

1.4.1. 在C语言中

```
clock_t start_t, finish_t;
start_t = clock();
.....
finish_t = clock();
printf("Run Time: %fs\n", (double)(finish_t - start_t) / CLOCKS_PER_SEC);
```

1.4.2. 在Python中

```
import time
t = time.time()
.....
print(f'coast:{time.time() - t:.4f}s')
```

1.5. 随机数生成

课题要求：“随机生成100000个随机数”

1. C中实现：

我们知道，在C中可以调用 `stdlib.h` 库中的 `rand()` 函数获取伪随机数。

由于`rand()`的原理，是通过一个系统每次启动时生成的种子，对于一个表单进行与种子相关的正态分布规则的查询。如果只调用`rand()`函数，会导致程序每次运行时（系统未重启的条件下）结果都相等。

我们通过`srand()`对种子进行重新配置。

市面上通常的方法是利用`time()`函数格式化时间作为种子，但这种方法会导致随机数也有增大或减小的趋势，是因为时间总是线性增大的。在这里，由于CPU时钟极度依赖系统环境，且灵敏度高，我选用CPU时钟计时`clock()`可以保证每次运行的值都有所不同。

另外，`rand()`的上线取决于`RAND_MAX`的值，在编译器中是固定的，如在gcc中是32767。因此，`rand()`给出的值总是0~32767之间的整数值。为了产生一个浮点数，我们可以采用多个`rand()`组合形成。

具体的实现方法如：

```
srand((unsigned) clock());
double a = (double) rand() / 100000 + rand();
printf("%lf\n", a);
```

不过，这种方法会损耗一定的运行效率。

2. Python中实现：

类似的，Python中也可以调用内置的`random`库来生成随机数。由于Python的实现比较简单，在这里直接给出实现方法：

```
import random

print(random.uniform(0, 100000))
```

即可生成0~100000之间的浮点随机数。

1.6. 打印结果

1. C语言中

由于C中对数组的操作较死板，我们只能遍历整个数组来实现写入文件。

```
FILE *fp = NULL;
fp = fopen("./log_c.txt", "w+");

for(int i = 0; i < MAXN; i++)
{
    fprintf(fp, "%f, ", A[i]);
}
fclose(fp);
```

2. Python中

有鉴于`print()`函数即可实现文件写入，我们这样来完成：

```
f = open('log_py.txt', 'w')
print(A, file=f)
f.close()
```

1.7. 实践环境

利用 [第四课](#) 中已经配置好的环境，我们直接在VS Code中编写并调试C和Python程序即可。

3. 冒泡排序的编程运行

在这里，我们直接编写经过完全优化的冒泡法排序程序。

3.1. 使用C实现

见 [*BS Full Optimized.c*](#)


```

C: > Users > JStar > Desktop > Temps > C BS Full Optimized.c > main()

25     FILE *fp = NULL;
26
27     fp = fopen("./log_c.txt", "w+");
28
29     for(int i = 0; i < MAXN; i++)
30     {
31         fprintf(fp, "%f, ", A[i]);
32         printf("%f, ", A[i]);
33     }
34
35     fclose(fp);
36
37     printf("\nRun Time: %fs\n", (double)(finish_t - start_t) / CLOCKS_PER_SEC);
38
39 }
40
41 void bubblesort(float A[])
42 {
43     for(int i = 0; i < MAXN; i++)
44     {
45         for(int j = i + 1; j < MAXN; j++)
46         {
47             if(A[i] > A[j])
48             {
49                 float temp = A[i];
50                 A[i] = A[j];
51                 A[j] = temp;
52             }
53         }
54     }
55 }
56
57 int main()
58 {
59     float A[MAXN];
60     srand((int)time(0));
61     for(int i = 0; i < MAXN; i++)
62     {
63         A[i] = (float)rand() / RAND_MAX;
64     }
65     bubblesort(A);
66     return 0;
67 }

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS SQL CONSOLE

```

.166016, 32721.283203, 32721.322266, 32722.042969, 32722.195312, 32722.208984, 32722.308594,
.000000, 32724.099609, 32724.298828, 32724.298828, 32726.134766, 32727.033203, 32727.320312,
.072266, 32729.195312, 32729.222656, 32730.013672, 32730.076172, 32730.251953, 32730.253906,
.218750, 32731.244141, 32731.281250, 32732.019531, 32732.029297, 32733.076172, 32734.101562,
.039062, 32737.076172, 32737.087891, 32737.107422, 32737.152344, 32738.121094, 32739.166016,
.060547, 32741.140625, 32741.142578, 32741.218750, 32741.228516, 32741.289062, 32741.302734,
.048828, 32743.111328, 32743.117188, 32743.222656, 32743.322266, 32744.001953, 32744.134766,
.166016, 32747.199219, 32747.253906, 32748.062500, 32748.160156, 32748.166016, 32748.173828,
.195312, 32749.222656, 32749.292969, 32749.314453, 32750.093750, 32750.150391, 32750.177734,
.005859, 32752.019531, 32752.156250, 32752.181641, 32752.285156, 32753.095703, 32753.324219,
.273438, 32755.072266, 32755.113281, 32755.166016, 32755.173828, 32755.189453, 32755.201172,
.128906, 32757.167969, 32758.138672, 32758.306641, 32759.011719, 32759.306641, 32760.146484,
.228516, 32763.308594, 32764.021484, 32764.054688, 32766.060547, 32766.136719, 32766.162109,
.128906, 32767.326172,
Run Time: 20.038000s
PS C:\Users\JStar>

```

main* 0 0 0 0 0 Connect

图为正确运行C语言编写的 *BS Full Optimized*

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

#define MAXN 100001

float A[MAXN] = {};

void bubblesort(float A[]);

```

```
int main()
{
    srand((unsigned) clock());
    for(int i = 0; i < MAXN; i++)
    {
        A[i] = (float) rand() / 100000 + rand();
    }

    clock_t start_t, finish_t;
    start_t = clock();
    bubblesort(A);
    finish_t = clock();

    FILE *fp = NULL;

    fp = fopen("./log_c.txt", "w+");

    for(int i = 0; i < MAXN; i++)
    {
        fprintf(fp, "%f, ", A[i]);
        printf("%f, ", A[i]);
    }

    fclose(fp);

    printf("\nRun Time: %fs\n", (double)(finish_t - start_t) / CLOCKS_PER_SEC);
}

void bubblesort(float A[])
{
    int n = MAXN;
    int i, newn;
    float temp;
    do
    {
        int newn = 0;
        for(i = 1; i < n; i++)
        {
            if(A[i - 1] > A[i])
            {
                temp = A[i - 1];
                A[i - 1] = A[i];
                A[i] = temp;
                newn = i;
            }
        }
        n = newn;
    } while (n > 1);
}
```

3.2. 使用Python实现

见 [*BS Full Optimized.py*](#)

```
79.55321463059, 99879.65057019737, 99879.67496414632, 99881.40475812048, 99881.97141732278, 99882.5763308742, 99888.46415032486, 99892.335
92391665, 99893.42494574146, 99895.17311266765, 99895.21856713043, 99895.35505679554, 99895.52006114612, 99898.20568467992, 99899.79529572
22, 99901.42108781815, 99901.78382139647, 99902.82873950533, 99904.4119022224, 99905.12853262445, 99905.25748271009, 99905.37923478488, 99
906.64811438585, 99906.67650705006, 99910.21485371595, 99911.57570582008, 99911.88486747137, 99912.44989911257, 99913.66349122104, 99913.7
2309806148, 99916.33883223307, 99916.83818389243, 99918.21553417071, 99918.23789599338, 99919.2631434411, 99920.99847704786, 99922.8744731
0168, 99924.6339953904, 99925.95220905976, 99926.02281661834, 99926.35142025816, 99927.68487662767, 99928.5298866226, 99929.18568812456, 9
9929.65721362458, 99930.12048347997, 99931.0671328476, 99933.07903708103, 99934.79494848284, 99935.29985211165, 99935.63652383882, 99935.6
9623328035, 99938.41615215811, 99939.21083621576, 99939.50251789289, 99940.94824729116, 99941.38637906782, 99942.19860294383, 99942.442673
23625, 99942.65269933957, 99946.36926937364, 99947.96871477956, 99947.99943001848, 99948.65058053318, 99949.08193812729, 99949.0908079504,
99949.95278455708, 99950.17423605826, 99953.04981833181, 99953.79870178233, 99955.20511423434, 99957.79840125906, 99968.74147257928, 9996
9.38292041917, 99971.60074750353, 99972.5628050251, 99972.75069111565, 99973.47078925626, 99973.71840300233, 99974.82843169445, 99974.9051
4743066, 99974.99812905036, 99977.22421519457, 99981.26028078492, 99981.55897057122, 99983.51870484716, 99983.52542585008, 99984.033532189
16, 99987.41427673475, 99988.68846303268, 99991.92182957967, 99992.65007577713, 99992.94401251926, 99995.3720433267, 99997.38136166478, 99
997.96766488555, 99998.56961863933, 99999.04468377287, 99999.22778410625, 99999.9502798487]
Time Cost of BS Func: 287.8909664154053 s
```

图为正确运行Python语言编写的 *BS Full Optimized*

```
import random
import time

def bubblesort(A):
    n = len(A)
    while n > 1:
        newn = 0
        for i in range(1, n):
            if A[i - 1] > A[i]:
                A[i - 1], A[i] = A[i], A[i - 1]
                newn = i
        n = newn
    return A

A = []
for i in range(0, 100000):
    A.append(random.uniform(0, 100000))

t = time.time()
bubblesort(A)
timeCost = time.time() - t
f = open('log_py.txt', 'w')
print(A)
print(A, file=f)
f.close()
print("\nTime Cost of BS Func: " + str(timeCost) + " s")
```

4. 快速排序的编程运行

4.1. 使用C实现

1. 直接调用C标准库中的 `qsort()` 函数

见 [*QS Use Build-In Function.c*](#)

比较有趣的是，在C语言标准库`stdlib.h`中，有`qsort()`函数。该函数是R. S. Scowen版本的快速排序的变体。

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
```

对一个数组`list[]`，`n`是需要排序的数量，通过`sizeof(list[0])`确定数组中每个元素占用空间的大小。

提供一个 `cmp()` 作为指针，确定排序的顺序：

```
int cmp (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b ); //从小到大的实现方法
    //return ( *(int*)b - *(int*)a ); //从大到小的实现方法
}
```

`qsort()` 函数的用法：

```
qsort(list, n, sizeof(list[0]), cmp);
```

```

5  #define MAXN 100001
6
7  int cmp (const void * a, const void * b)
8  {
9      return ( *(int*)a - *(int*)b ); //从小到大的实现方法
10     //return ( *(int*)b - *(int*)a ); //从大到小的实现方法
11 }
12
13 int main()
14 {
15     float A[MAXN] = {};
16     srand((unsigned) clock());
17     for(int i = 0; i < MAXN; i++)
18     {

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE

```

2712.197266, 32712.203125, 32713.166016, 32713.197266, 32714.099609, 32714.175781, 32714.18
2717.222656, 32718.255859, 32719.056641, 32719.064453, 32719.068359, 32719.214844, 32720.00
2722.046875, 32722.125000, 32722.181641, 32722.287109, 32723.269531, 32723.281250, 32723.28
2725.095703, 32725.253906, 32725.308594, 32726.044922, 32726.109375, 32727.021484, 32727.18
2730.279297, 32731.058594, 32731.318359, 32732.033203, 32732.101562, 32732.128906, 32732.14
2734.173828, 32734.212891, 32734.324219, 32735.046875, 32735.207031, 32735.273438, 32736.04
2737.300781, 32738.279297, 32739.025391, 32740.042969, 32741.255859, 32742.003906, 32742.05
2744.207031, 32744.212891, 32744.253906, 32744.263672, 32745.179688, 32745.181641, 32746.01
2749.033203, 32749.126953, 32749.158203, 32749.322666, 32750.113281, 32750.136719, 32750.16
2751.208984, 32751.250000, 32751.279297, 32752.091797, 32752.207031, 32753.076172, 32754.18
2756.058594, 32756.119141, 32756.203125, 32756.220703, 32757.021484, 32757.052734, 32757.10
2758.058594, 32758.140625, 32759.027344, 32759.037109, 32759.039062, 32759.216797, 32760.12
2761.242188, 32762.175781, 32762.281250, 32763.083984, 32763.230469, 32763.279297, 32763.30
2765.140625, 32766.035156, 32766.125000, 32766.167969, 32766.195312, 32766.324219, 32767.11
Run Time: 0.008000s
PS C:\Users\JStar>

```

图为正确运行C语言编写的 *QS Build-In Function.c*

写在具体程序里即：

```

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

#define MAXN 100001

int cmp (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b ); //从小到大的实现方法
    //return ( *(int*)b - *(int*)a ); //从大到小的实现方法
}

int main()
{
    float A[MAXN] = {};
    srand((unsigned) clock());
    for(int i = 0; i < MAXN; i++)
    {

```

```
        A[i] = (double) rand() / 100000 + rand();
    }

    clock_t start_t, finish_t;
    start_t = clock();

    qsort(A, MAXN, sizeof(A[0]), cmp);

    finish_t = clock();

    FILE *fp = NULL;

    fp = fopen("./log_c.txt", "w+");

    for(int i = 0; i < MAXN; i++)
    {
        fprintf(fp, "%f, ", A[i]);
        printf("%f, ", A[i]);
    }

    fclose(fp);

    printf("\nRun Time: %fs\n", (double)(finish_t - start_t) /
CLOCKS_PER_SEC);

}
```

2. 基于 *Hoare* 的 Partition Scheme + 三数取中的 Pivot Choice

见 [QS Hoare + Pivot Choice.c](#)

```

1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4  #include <math.h>
5
6  #define MAXN 100001
7
8  float A[MAXN] = {};
9  float temp;
10 int p = 0;
11
12 void quicksort(int lo, int hi);
13 int partition(int lo, int hi);
14 float pivot(int lo, int hi);
15
16 int main()
17 {
18     srand((unsigned) clock());

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE

```

12.310547, 32713.023438, 32713.111328, 32714.025391, 32714.148438, 32714.259766, 32714.29101
17.083984, 32717.302734, 32718.037109, 32718.162109, 32718.164062, 32719.007812, 32719.02343
22.087891, 32722.115234, 32722.128906, 32722.283203, 32723.283203, 32724.162109, 32724.28906
26.042969, 32726.072266, 32727.072266, 32727.111328, 32727.156250, 32727.203125, 32727.21484
29.244141, 32729.279297, 32729.283203, 32731.275391, 32732.029297, 32732.099609, 32732.11328
34.238281, 32735.035156, 32735.068359, 32735.080078, 32735.177734, 32735.273438, 32735.28515
37.042969, 32737.082031, 32737.083984, 32737.130859, 32737.203125, 32737.277344, 32737.28515
39.003906, 32739.042969, 32739.093750, 32740.234375, 32740.300781, 32741.115234, 32742.18945
44.195312, 32745.173828, 32746.035156, 32746.101562, 32746.107422, 32746.144531, 32747.03906
50.185547, 32751.046875, 32751.214844, 32751.279297, 32752.148438, 32752.289062, 32753.18945
55.021484, 32755.052734, 32755.058594, 32755.087891, 32756.167969, 32756.203125, 32757.03125
58.226562, 32758.251953, 32758.310547, 32759.000000, 32759.041016, 32760.078125, 32760.11523
62.078125, 32762.216797, 32762.271484, 32763.095703, 32763.115234, 32763.193359, 32764.24218
66.177734, 32766.236328, 32767.035156, 32767.212891, 32767.302734,
Run Time: 0.010000s
PS C:\Users\JStar>

```

图为正确运行C语言编写的 QS Hoare + Pivot Choice

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

#define MAXN 100001

float A[MAXN] = {};
float temp;
int p = 0;

void quicksort(int lo, int hi);
int partition(int lo, int hi);
float pivot(int lo, int hi);

int main()

```

```
{
    srand((unsigned) clock());
    for(int i = 0; i < MAXN; i++)
    {
        A[i] = (double) rand() / 100000 + rand();
    }

    clock_t start_t, finish_t;
    start_t = clock();
    quicksort(0, 100000);
    finish_t = clock();

    FILE *fp = NULL;

    fp = fopen("./log_c.txt", "w+");

    for(int i = 0; i < MAXN; i++)
    {
        fprintf(fp, "%f, ", A[i]);
        printf("%f, ", A[i]);
    }

    fclose(fp);

    printf("\nRun Time: %fs\n", (double)(finish_t - start_t) /
CLOCKS_PER_SEC);
}

void quicksort(int lo, int hi)
{
    if(lo >= 0 && hi >= 0 && lo < hi)
    {
        p = partition(lo, hi);
        quicksort(lo, p);
        quicksort(p + 1, hi);
    }
}

int partition(int lo, int hi)
{
    int i = lo - 1, j = hi + 1;
    float pivotN = pivot(lo, hi);
    while(1)
    {
        do i++; while(A[i] < pivotN);

        do j--; while(A[j] > pivotN);

        if(i >= j) return j;
    }
}
```



```
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
}

float pivot(int lo, int hi)
{
    int mid = ((hi - lo) / 2) + lo;
    if(A[mid] < A[lo]) {
        temp = A[lo];
        A[lo] = A[mid];
        A[mid] = temp;
    }
    if(A[hi] < A[lo]) {
        temp = A[lo];
        A[lo] = A[hi];
        A[hi] = temp;
    }
    if(A[mid] < A[hi]) {
        temp = A[hi];
        A[hi] = A[mid];
        A[mid] = temp;
    }
    return A[hi];
}
```

3. 基于 *Lomuto* 的 Partition Scheme + 三数取中的 Pivot Choice + 三路快排优化

见 [*QS Lomuto + Pivot Choice + 3 Ways.c*](#)

```

56
57 int partition(float p, int lo, int hi)
58 {
59     left = lo;
60     right = hi;
61     int i = lo;
62
63     while(i <= right)
64     {
65         if(A[i] < p)
66         {
67             temp = A[i];
68             A[i] = A[left];
69             A[left] = temp;
70             left++;
71             i += 1;
72         }
73         else if(A[i] > p)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE

```

5, 32716.308594, 32716.310547, 32717.042969, 32717.199219, 32717.257812, 32717.25976
9, 32719.083984, 32719.304688, 32720.005859, 32720.083984, 32720.119141, 32720.17382
1, 32723.193359, 32723.232422, 32723.255859, 32723.285156, 32724.201172, 32724.24414
2, 32726.132812, 32726.136719, 32726.195312, 32726.203125, 32726.208984, 32727.03515
0, 32729.173828, 32729.222656, 32729.271484, 32729.271484, 32730.021484, 32730.14648
9, 32733.058594, 32733.181641, 32733.250000, 32735.052734, 32735.074219, 32735.11718
5, 32737.273438, 32738.013672, 32738.099609, 32739.062500, 32739.103516, 32739.16015
3, 32744.017578, 32744.261719, 32744.275391, 32744.324219, 32745.287109, 32746.02734
9, 32747.046875, 32747.052734, 32747.167969, 32747.320312, 32748.130859, 32749.11914
5, 32751.294922, 32753.037109, 32753.134766, 32753.150391, 32753.234375, 32753.30468
1, 32755.156250, 32756.281250, 32757.148438, 32757.179688, 32757.195312, 32757.20507
1, 32761.005859, 32761.179688, 32761.224609, 32762.158203, 32762.224609, 32762.32617
5, 32765.001953, 32765.033203, 32765.050781, 32765.097656, 32765.296875, 32766.11328
1, 32767.179688, 32767.251953,
Run Time: 0.015000s
PS C:\Users\JStar>

```

main* 0 0 0 0 Connect

图为正确运行C语言编写的 QS Lomuto + Pivot Choice + 3 Ways

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

#define MAXN 100001

float A[MAXN] = {};
float temp;
int left, right;

```

```
float p = 0;

void quicksort(int lo, int hi);
int partition(float p, int lo, int hi);
float pivot(int lo, int hi);

int main()
{
    srand((unsigned) clock());
    for(int i = 0; i < MAXN; i++)
    {
        A[i] = (double) rand() / 100000 + rand();
    }

    clock_t start_t, finish_t;
    start_t = clock();
    quicksort(0, 100000);
    finish_t = clock();

    FILE *fp = NULL;

    fp = fopen("./log_c.txt", "w+");

    for(int i = 0; i < MAXN; i++)
    {
        fprintf(fp, "%f, ", A[i]);
        printf("%f, ", A[i]);
    }

    fclose(fp);

    printf("\nRun Time: %fs\n", (double)(finish_t - start_t) /
CLOCKS_PER_SEC);
}

void quicksort(int lo, int hi)
{
    if(lo < hi)
    {
        p = pivot(lo, hi);
        partition(p, lo, hi);
        quicksort(lo, left - 1);
        quicksort(right + 1, hi);
    }
}

int partition(float p, int lo, int hi)
{
    left = lo;
    right = hi;
```

```
int i = lo;

while(i <= right)
{
    if(A[i] < p)
    {
        temp = A[i];
        A[i] = A[left];
        A[left] = temp;
        left++;
        i += 1;
    }
    else if(A[i] > p)
    {
        temp = A[i];
        A[i] = A[right];
        A[right] = temp;
        right--;
    }
    else {
        i++;
    }
}
return 0;
}

float pivot(int lo, int hi)
{
    int mid = ((hi - lo) / 2) + lo;
    if(A[mid] < A[lo]) {
        temp = A[lo];
        A[lo] = A[mid];
        A[mid] = temp;
    }
    if(A[hi] < A[lo]) {
        temp = A[lo];
        A[lo] = A[hi];
        A[hi] = temp;
    }
    if(A[mid] < A[hi]) {
        temp = A[hi];
        A[hi] = A[mid];
        A[mid] = temp;
    }
    return A[hi];
}
```

4.2. 使用Python实现

1. 基于 *Hoare* 的 Partition Scheme + 三数取中的 Pivot Choice

见 [QS Hoare + Pivot Choice.py](#)

```

6         p = partition(A, lo, hi)
7         quicksort(A, lo, p)
8         quicksort(A, p + 1, hi)
9
10    def partition(A, lo, hi):
11
12        i = lo
13        j = hi
14
15        mid = (hi - lo) // 2 + lo
16        if A[mid] < A[lo]:
17            A[lo], A[mid] = A[mid], A[lo]
18        if A[hi] < A[lo]:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE

99901.70030410726, 99902.59261056781, 99902.94131514635, 99905.09768066149, 99906.24273961426, 99908, 99909.07600252931, 99909.25748453679, 99909.53087771306, 99909.65904651128, 99912.10027005471, 3145, 99916.62031746581, 99917.25462908926, 99917.55946480618, 99917.89718841232, 99918.65818982314, 97309, 99921.78822163281, 99922.18844836197, 99923.23264505615, 99923.28875088987, 99923.9698820029, 15997779, 99926.3425946496, 99926.93661485071, 99928.58561573034, 99928.8035576084, 99928.972106364, 3947462, 99933.39418223905, 99934.35166563009, 99934.61370246591, 99935.95226275605, 99936.11684886, 3604367097, 99943.14108201096, 99943.35762202571, 99945.72957363068, 99947.35701134775, 99947.62111, 2.31155525685, 99952.51216231413, 99952.61882363059, 99953.25846746718, 99953.66330413078, 99954.07, 955.5728216341, 99955.75132202554, 99956.20825039127, 99956.64411538358, 99959.29899510769, 99959.6, 9965.31984440632, 99965.4290292877, 99965.76839097952, 99966.40745664155, 99967.99569553451, 99968, 9973.19003470149, 99973.4397074047, 99973.6484003915, 99973.9242488516, 99974.31794679615, 99975.33, 984.52375004972, 99984.94376794792, 99985.60160477253, 99985.88665377715, 99986.79292746038, 99987, 99998.04749885153, 99998.38107621095, 99998.5018630885, 99998.83624236638, 99999.41004926212, 99999

Time Cost of QS Func: 0.12379217147827148 s
PS C:\Users\JStar> █

图为正确运行Python语言编写的 *QS Hoare + Pivot Choice*

```

import random
import time

def quicksort(A, lo, hi):
    if lo >= 0 and hi >= 0 and lo < hi:
        p = partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

def partition(A, lo, hi):

    i = lo
    j = hi

    mid = (hi - lo) // 2 + lo
    if A[mid] < A[lo]:
        A[lo], A[mid] = A[mid], A[lo]
    if A[hi] < A[lo]:
        A[lo], A[hi] = A[hi], A[lo]
    if A[mid] < A[hi]:
        A[hi], A[mid] = A[mid], A[hi]

```

```
    pivot = A[hi]

    while True:
        while A[i] < pivot:
            i += 1
        while A[j] > pivot:
            j -= 1
        if i >= j:
            return j
        A[i], A[j] = A[j], A[i]

A = []
for i in range(0, 100000):
    A.append(random.uniform(0, 100000))

t = time.time()
quicksort(A, 0, len(A) - 1)
timeCost = time.time() - t
f = open('log_py.txt', 'w')
print(A)
print(A, file=f)
f.close()
print("\nTime Cost of QS Func: " + str(timeCost) + " s")
```

2. 基于 *Lomuto* 的 Partition Scheme + 三数取中的 Pivot Choice + 三路快排优化

见 [*QS Lomuto + Pivot Choice + 3 Ways.py*](#)

```

1  import random
2  import time
3
4  def quicksort(A, lo, hi):
5      if lo < hi:
6          p = pivot(A, lo, hi)
7          left, right = partition(A, p, lo, hi) # 三路混合
8          quicksort(A, lo, left - 1)
9          quicksort(A, right + 1, hi)
10
11 def partition(A, p, lo, hi):
12     left = lo
13     right = hi
14     i = lo
15
16     while i <= right:
17         if A[i] < p:
18             A[i], A[left] = A[left], A[i]

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE

```

.28830979914, 99896.94198307514, 99897.46050925758, 99898.28107969691, 99898.50516744115, 99898.919
3.50374256982, 99906.83082376841, 99906.98884039104, 99907.18831454031, 99907.65475621028, 99908.11
10.33515733764, 99913.05696415987, 99913.24085327271, 99914.26772300863, 99914.51159467241, 99914.7
99922.39417296847, 99922.97465796997, 99923.18785585437, 99924.03571542572, 99924.23890958131, 9992
, 99934.18952975351, 99934.22058856481, 99934.70402531652, 99935.91642659169, 99936.88706347594, 99
473, 99945.05663715475, 99945.29207917457, 99947.73383190572, 99949.15902663153, 99949.75188237154,
992133, 99951.22607668523, 99952.16212709743, 99952.58008233513, 99955.20286575652, 99956.863090846
125217875, 99959.189344746, 99959.87689855167, 99960.44733490367, 99961.32532870177, 99962.60443915
23478572, 99965.85347144512, 99966.09867244755, 99967.379003183, 99967.7577804323, 99969.7603819749
5181018, 99972.93129684603, 99973.23927867493, 99974.51767928695, 99979.93767101552, 99981.21905257
6957131785, 99987.40169433052, 99987.55257238733, 99987.75029914762, 99988.13224574248, 99988.42281
1.09673269668, 99992.24130351358, 99992.8085107002, 99994.27229193386, 99994.33389322877, 99994.513
7.67304080911, 99997.67700096779, 99998.75741936438]

```

Time Cost of QS Func: 0.15372896194458008 s
PS C:\Users\JStar>

图为正确运行Python语言编写的 *QS Lomuto + Pivot Choice + 3 Ways*

```

import random
import time

def quicksort(A, lo, hi):
    if lo < hi:
        p = pivot(A, lo, hi)
        left, right = partition(A, p, lo, hi) # 三路混合
        quicksort(A, lo, left - 1)
        quicksort(A, right + 1, hi)

def partition(A, p, lo, hi):
    left = lo
    right = hi
    i = lo

    while i <= right:

```

```
        if A[i] < p:
            A[i], A[left] = A[left], A[i]
            left += 1
            i += 1
        elif A[i] > p:
            A[i], A[right] = A[right], A[i]
            right -= 1
        else:
            i += 1
    return left, right

def pivot(A, lo, hi): # 三数取中
    mid = (hi - lo) // 2 + lo
    if A[mid] < A[lo]:
        A[lo], A[mid] = A[mid], A[lo]
    if A[hi] < A[lo]:
        A[lo], A[hi] = A[hi], A[lo]
    if A[mid] < A[hi]:
        A[hi], A[mid] = A[mid], A[hi]
    return A[hi]

A = []
for i in range(0, 100000):
    A.append(random.uniform(0, 100000))

t = time.time()
quicksort(A, 0, len(A) - 1)
timeCost = time.time() - t
f = open('log_py.txt', 'w')
print(A)
print(A, file=f)
f.close()
print("\nTime Cost of QS Func: " + str(timeCost) + " s")
```

5. 结果比较

5.1. 建立列表

我所说的时间测定是**只计算算法部分的运行时间**，不包含**导入随机数组、输出随机数组的时间**。

在我连接电源的笔记本下，控制计算机运行状态基本一致，进行五次测量取平均值，得到结果：

a. 冒泡排序：

	C	Python
1st	19.903s	287.891s
2nd	19.970s	284.689s
3rd	20.095s	281.532s

	C	Python
4th	19.988s	283.463s
5th	20.291s	285.917s
Avr	20.049s	284.698s

b. Hoare 快排：

	C	Python
1st	0.009s	0.120s
2nd	0.010s	0.129s
3rd	0.010s	0.125s
4th	0.010s	0.126s
5th	0.009s	0.124s
Avr	0.0096s	0.1248s

c. Lomuto 快排：

	C	Python
1st	0.014s	0.142s
2nd	0.012s	0.145s
3rd	0.011s	0.149s
4th	0.015s	0.148s
5th	0.011s	0.149s
Avr	0.0126s	0.1466s

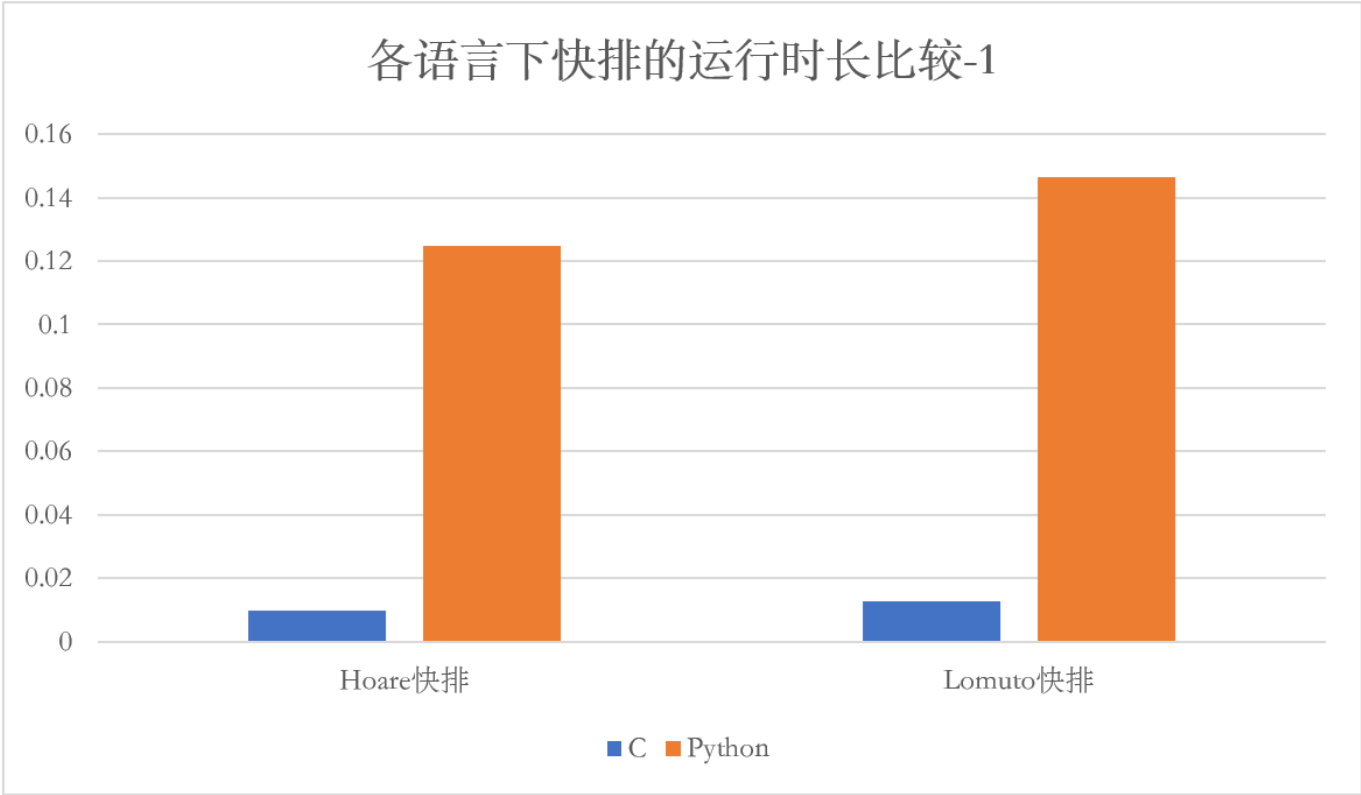
注：由于C内置的快排算法并不纯粹，在这里不做比较。

整理得下表。

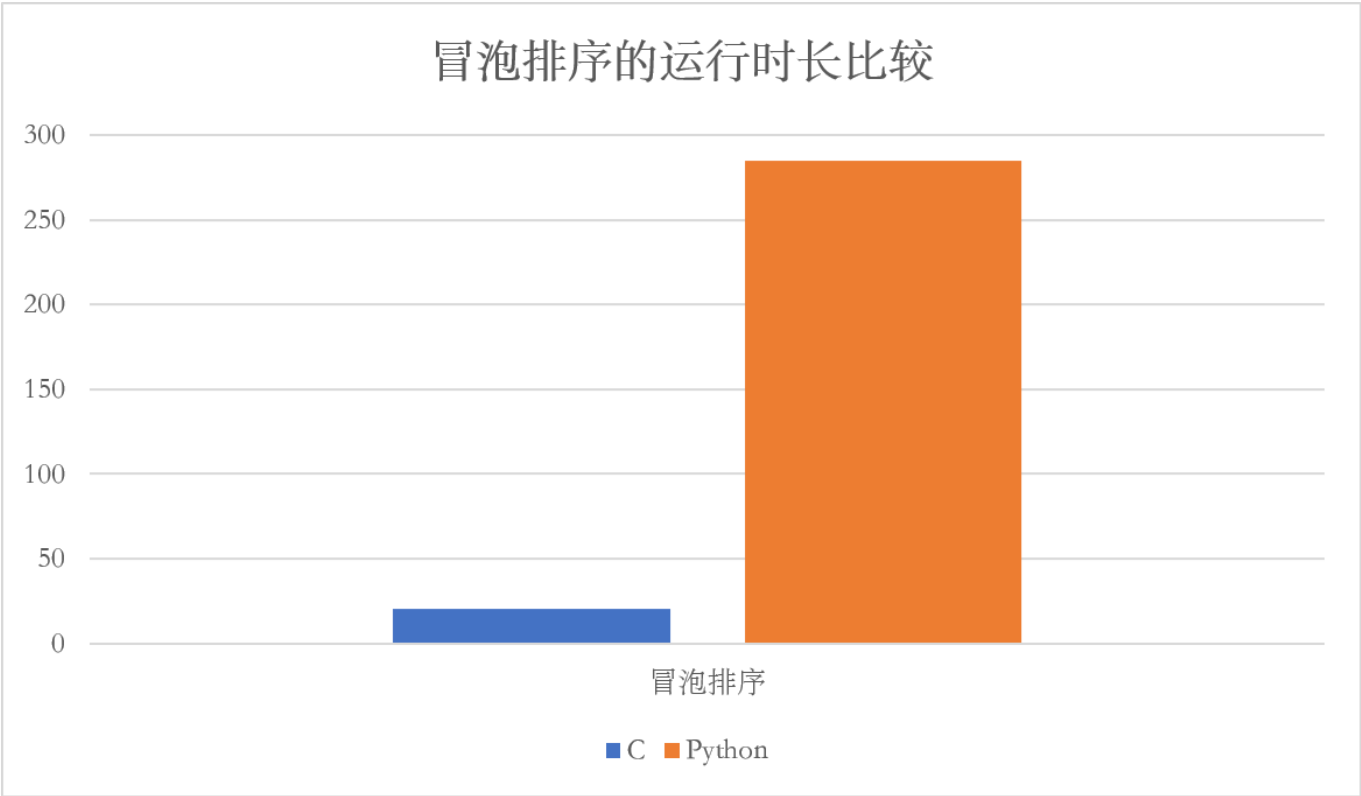
	C	Python
冒泡排序	20.049s	284.698s
Hoare 快排	0.0096s	0.1248s
Lomuto 快排	0.0126s	0.1466s

5.2. 数据分析

快排和冒泡排序两者时间量级相差过大，因此分别比较。



图为C与Python的快排算法比较



图为C与Python的冒泡排序算法比较

可以明显发现，每个算法的执行效率都体现了Python慢于C的情况。如何正确理解这个现象？

根据我查阅的资料，提供如下回答：

Python运行效率慢于C语言的原因主要有以下几点：

1. 解释执行 vs 编译执行：Python是一种解释型语言，它在运行时需要一个解释器逐行解释代码并执行，这个过程会增加额外的运行时间。而C语言是编译型语言，它的代码在运行前就已经被编译成机器语言，因此执行速度更快。
2. 动态类型：Python是动态类型语言，变量的类型是在运行时确定的，这给变量操作带来了便利，但也增加了类型检查的开销。C语言是静态类型语言，变量的类型在编译时就已确定，这减少了运行时的类型检查和转换。
3. 全局解释器锁（GIL）：Python的标准实现CPython中有一个全局解释器锁，它限制了多线程的并发执行，因为在任何时刻只有一个线程可以执行Python代码。这意味着即使在多核处理器上，Python程序也无法充分利用多核的优势。
4. 内存管理：Python使用自动垃圾收集来管理内存，这提供了编程的便利，但也意味着性能上的牺牲。C语言允许程序员更直接地控制内存分配和回收，这可以提高效率。
5. 内置的高级数据结构和库：Python提供了丰富的内置数据结构和库，这些都是为了方便开发者而设计的，但它们的通用性和便利性可能会牺牲一些性能。

尽管Python在运行效率上慢于C语言，但它在开发效率、代码可读性和跨平台性等方面具有优势。此外，有一些工具和项目旨在提高Python的性能，例如使用JIT编译器的PyPy，将Python代码转换为C++代码的Nuitka，以及为Python编写C扩展的Cython等。这些工具和项目通过不同的方式来优化Python代码的执行速度，使其更适合特定的应用场景。

6. 问题解决

我的解决方案写在文中了，在此一一列出，您可以单击各个条目以跳转到对应位置。

1. [整理Markdown的基本语法](#)
2. [版本控制的多种实现思路](#)
3. [jsDelivr的线路选择](#)
4. [使用何种程序辅助GitHub Repo作图床](#)
5. [如何在Markdown中用base64存储图片](#)
6. [Bubble Sort有什么优化方法](#)
7. [Quick Sort有什么优化方法](#)
8. [如何生成随机数](#)
9. [如何打印结果到文档](#)
10. [Python运行效率为什么低于C语言](#)

7. 原创声明

在这里特别列出非原创内容：

- [冒泡排序法的优化](#)
- [快速排序法的优化](#)
- [Python与C语言效率比较](#)

除特别指出的知识性内容参考了网络资料外，**内容均由我个人原创。**