**tags: 資料結構**

# HW27

本篇作業需要比較四種不同 sort 在各自最壞複雜度的情況下所需的時間，首先會先介紹幾種排序方法各自的理論複雜度，最後在進行比較，這份作業我測試時間的方法都是使用 python 腳本測量，分別在開始即結束時使用 python 的 `time.time()` 獲得時間在相減來得到程序運行花費的時間。

## Analyze

### Insertion sort

- 資料量少時較高效，且比其他的 $O(n^2)$ 的排序法還要高效
- 穩定排序，相同鍵值的元素排序後的相對位置不變
- 可在線處理，且不需要花費額外空間來排序

| Status | Complexity |
| --- | --- |
| worst | $O(n^2)$ |
| best | $O(n)$ |
| average | $O(n^2)$ |
| space | $O(1)$ |

```
void InsertSort(int* arr, int len)
{
    for (int i = 2, tmp, j; i <= len; i++) {
        j = i - 1, tmp = arr[i];
        while (j && arr[j] > tmp)
            arr[j + 1] = arr[j], j--;
        arr[j + 1] = tmp;
    }
}
```

## Quick Sort

- 不穩定排序，相同鍵值的元素排序後的相對位置可能改變
- 需要花費額外空間來排序
- 分治算法

| Status | Complexity |
| --- | --- |
| worst | $O(n^2)$ |
| best | $O(n \log n)$ |
| average | $O(n \log n)$ |
| space | $O(\log n)$ |

這邊跟題序有點不一樣的部分是題目要求在 pivot 的選擇上要使用 median-of-three，但這做法這做法的常數均攤下來基本上會跟 mid 差不多，因此這邊直接使用 mid。

```
void QuickSort(int* arr, int L, int R)
{
    if (L < R) {
        int i = L, j = R, pivot = arr[(L + R) >> 1];
        while (i < j) {
            while (arr[i] < pivot)
                i++;
            while (arr[j] > pivot)
                j--;
            if (i < j)
                swap(arr[i], arr[j]), i++, j--;
        }
        QuickSort(arr, L, pivot - 1);
        QuickSort(arr, pivot + 1, R);
    }
}
```

## Merge Sort

- 最好、最壞、平均時間複雜度都是 $O(n \log n)$
- 穩定排序，相同鍵值的元素排序後的相對位置不變
- 分治算法

| Status | Complexity |
| --- | --- |
| worst | $O(n \log n)$ |
| best | $O(n \log n)$ |
| average | $O(n \log n)$ |
| space | $O(n)$ |

這邊提供了 recursive 以及 iteration 的 code。

```
void _merge(int* arr, int L, int R, int mid)
{
    int* tmp = (int*)malloc(sizeof(int) * (R - L + 1));
    int idx = 0, p1 = L, p2 = mid + 1;
    while (p1 <= mid && p2 <= R)
        tmp[idx++] = (arr[p1] < arr[p2] ? arr[p1++] : arr[p2++]);
    while (p1 <= mid)
        tmp[idx++] = arr[p1++];
    while (p2 <= R)
        tmp[idx++] = arr[p2++];
    for (int i = 0; i < idx; i++)
        arr[L + i] = tmp[i];
}

void RMergeSort(int* arr, int L, int R)
{
    if (L < R) {
        int mid = (L + R) >> 1;
        RMergeSort(arr, L, mid);
        RMergeSort(arr, mid + 1, R);
        _merge(arr, L, R, mid);
    }
}

void IMergeSort(int* arr, int L, int R)
{
    for (int i = 1; i <= R - L; i <<= 1)
        for (int j = L; j <= R; j += i << 1)
            _merge(arr, j, min(j + (i << 1) - 1, R), j + i - 1);
}
```

## Heap Sort

- 穩定排序，相同鍵值的元素排序後的相對位置不變
- 不需要花費額外空間來排序

| Status | Complexity |
| --- | --- |
| worst | $O(n \log n)$ |
| best | $O(n \log n)$ |
| average | $O(n \log n)$ |
| space | $O(1)$ |

```
1  void Heapfy(int* arr, int idx, int len)
2  {
3      if (idx << 1 <= len) {
4          while ((idx <<= 1) <= len) {
5              if (idx + 1 <= len && arr[idx + 1] > arr[idx])
6                  idx += 1;
7              if (arr[idx >> 1] < arr[idx])
8                  swap(arr[idx], arr[idx >> 1]);
9              else
10                 break;
11         }
12     }
13 }
14
15 void HeapSort(int* arr, int len)
16 {
17     for (int i = len >> 1; i >= 1; i--)
18         Heapfy(arr, i, len);
19     for (int i = len; i > 1; i--)
20         swap(arr[1], arr[i]), Heapfy(arr, 1, i - 1);
21 }
```

# Testcase

這邊針對各種 sort 的 worst case 講解不同的測資產生方法，因為測資及測試腳本皆是使用 python 故此處皆是 python 的 pseudo code。

1. Insertion sort
   這是最簡單的，只需要將測資倒序排列就會是 worst case。

```
def _universal(self, type):
        test_case = "{}\n".format(type)
        for i in range(self.amount, 0, -1):
            test_case += "{} ".format(i)
        test_case += "\n"
        return test_case
```

2. Quick sort
   影響 Quick sort 複雜度的除了測資外，pivot 的選擇也會有很大的影響，像是選擇中間的元素相比選擇頭尾的元素就會相差很大，一種通用的 worst case 就是所有的元素都相通，這樣每層遞迴就會需要通通跑完。

```
def _quick(self):
        test_case = "1\n"
        for _ in range(self.amount):
            test_case += "{} ".format(1)
        test_case += "\n"
        return test_case
```

3. Merge sort
   理論上並沒有所謂的 worst case 因為 merge sort 每次都需要遞迴到底部，但如果要考慮 swap 帶來的時間差的話，那倒序會是最壞的情況，但也僅僅會是常數的誤差。

```
def _universal(self, type):
        test_case = "{}\n".format(type)
        for i in range(self.amount, 0, -1):
            test_case += "{} ".format(i)
        test_case += "\n"
        return test_case
```

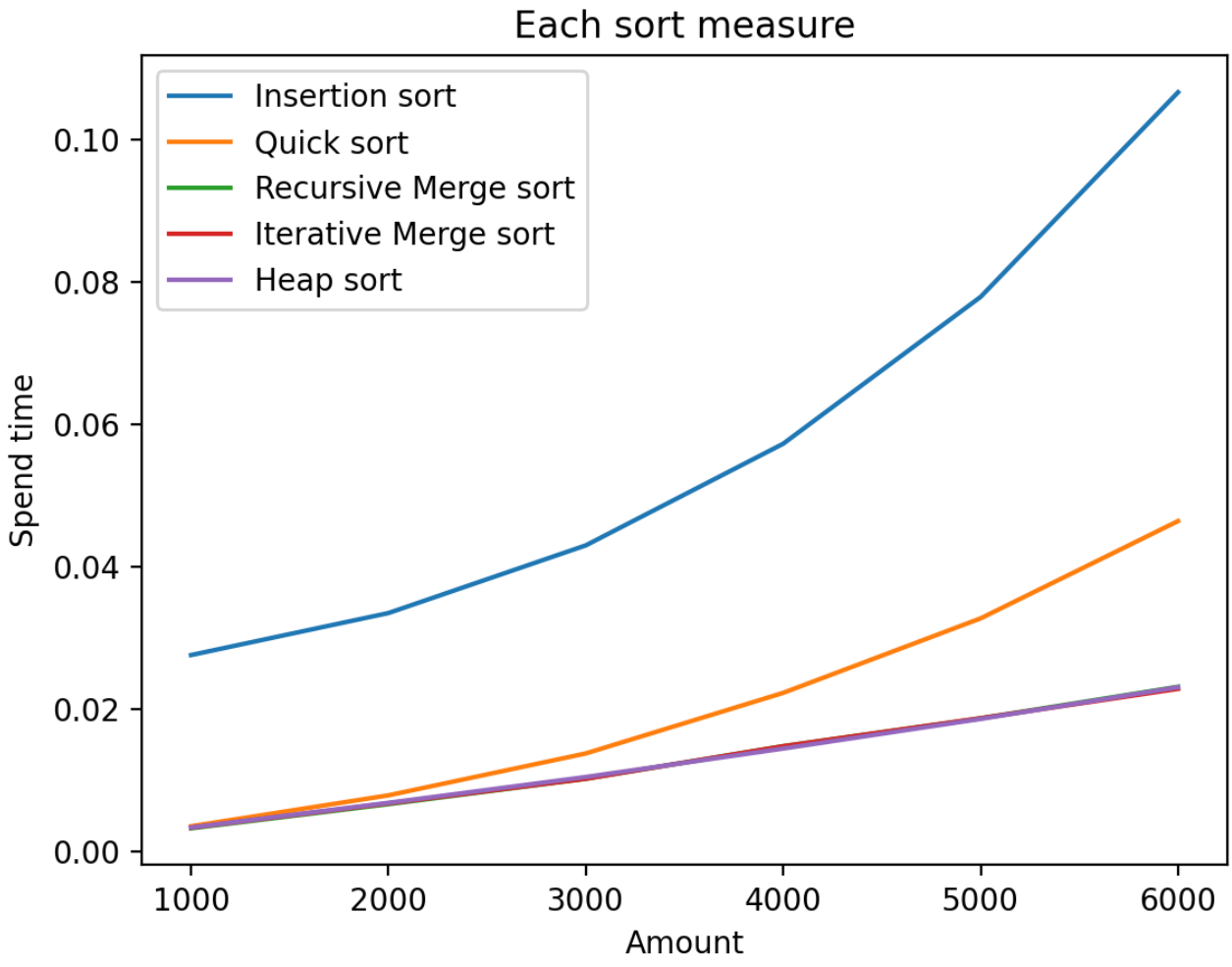
4. Heap sort
   對於 Heao sort 的測資選擇會較為麻煩，因此這邊採用 random 然後多跑幾次取平均的方法來測試。

```
def _heap(self):
        test_case = "3\n"
        elements = [i for i in range(self.amount)]
        random.shuffle(elements)
        for i in elements:
            test_case += "{} ".format(i)
        test_case += "\n"
        return test_case
```

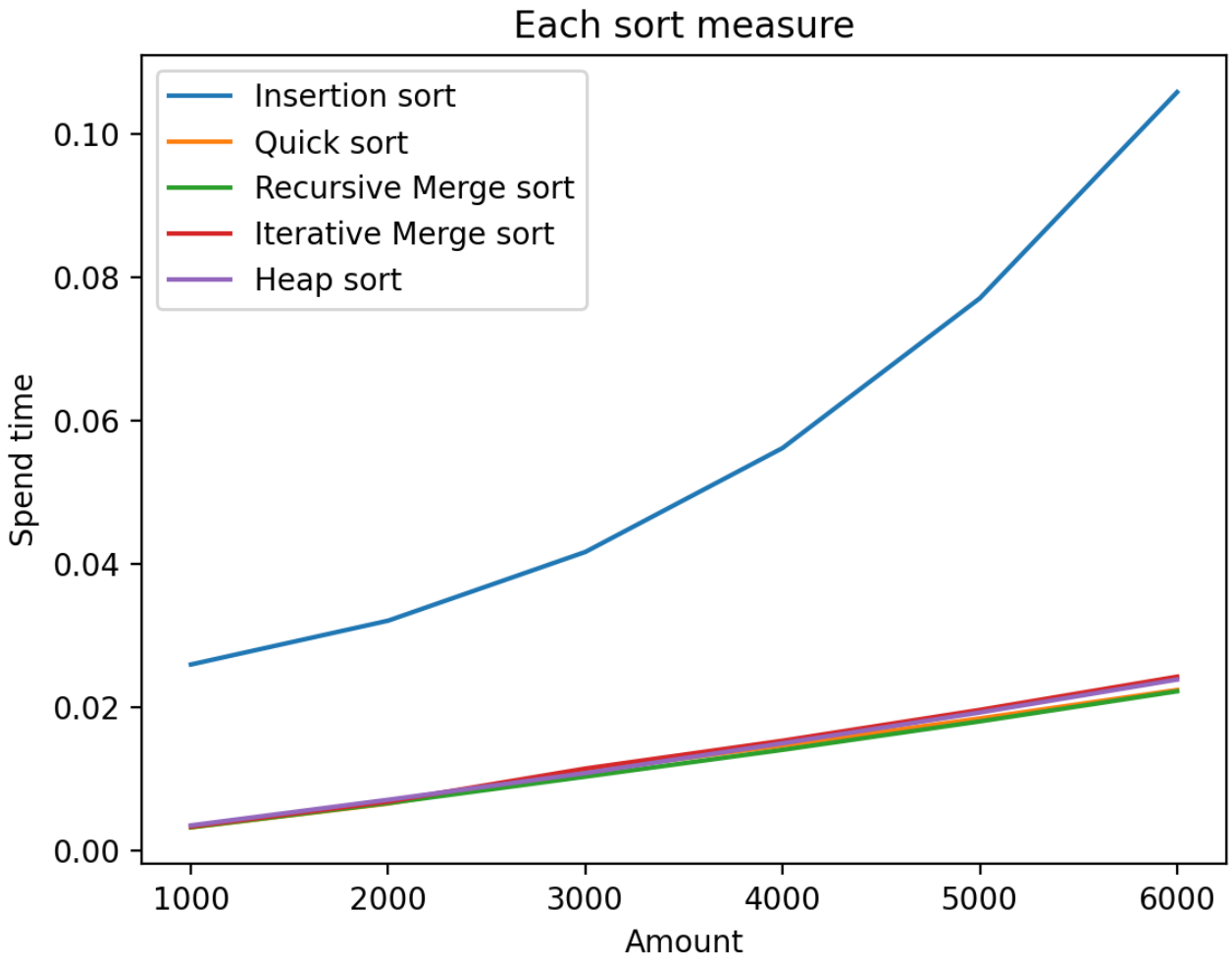
## Measure

在這邊我們多做了幾種不同的改動，除了新增遞迴版本的 merge sort 以外，也新增了幾種不同選擇 pivot 方式的 quick sort 對比，見下圖。

1. First pivot of quick sort




2. Median-of-Three pivot of quick sort


Each sort measure
Insertion sort
Quick sort
Recursive Merge sort
Iterative Merge sort
Heap sort
Spend time
0.00
0.02
0.04
0.06
0.08
0.10
1000
2000
3000
4000
5000
6000
Amount

最後再額外補上一張 random 的測試圖，可以看到的確符合上面給出的複雜度，其實在百萬數據集內基本上都不會相差太大。