C 語言的函式庫---qsort()

在使用 **qsort()**以前必須先了解一種排序演算法，也就是 **quicksort**，又稱為分割區交換排（**partition-exchange sort**），由東尼·霍爾所發明的 **quicksort**，幾乎是現在排序演算法中最快的存在，在大多數的情況下，他的平均時間複雜度是 **O(nlog n)**，除非有特定的情況發生**(**例如 **pivot** 總是選到最大或最小值**)**，否則這個排序的方式非常方便且快速。

簡單介紹 **quicksort** 的原理:

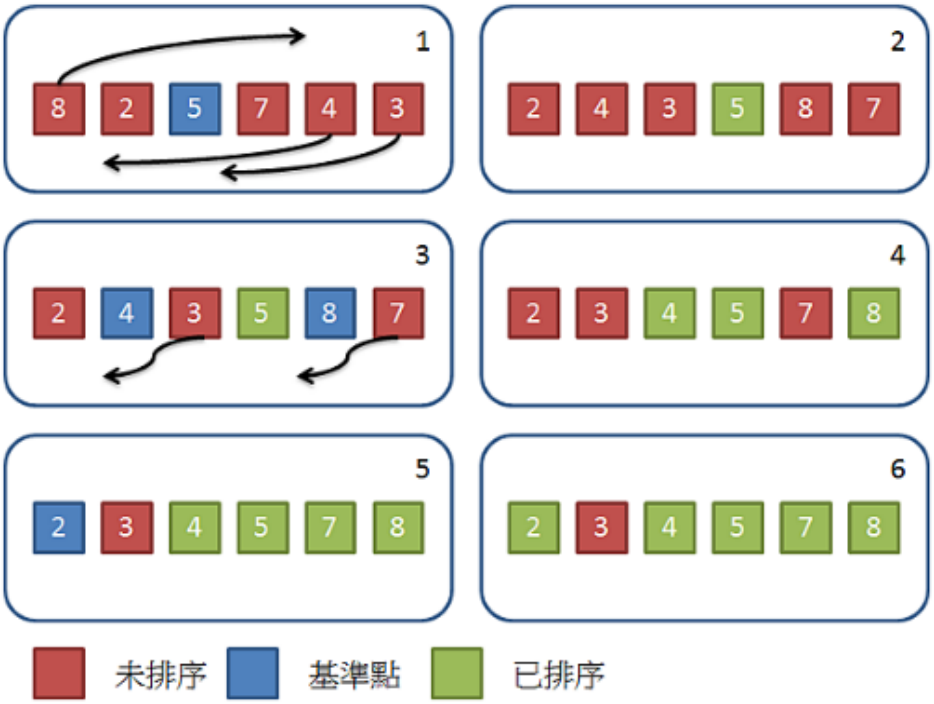
Quicksort 是一個分治演算法（**divide-and-conquer**），不斷遞迴下列三個步驟**=>**

1. 選擇 **Pivot**：在序列中任意選擇一個元素，稱為 **Pivot**。

2.分割序列：將序列重新排序，分為兩部分，比 **pivot** 小 的元素置換到 **pivot** 之前，比 **pivot** 大的元素置換到 **pivot** 之後，而 **pivot** 本身會座落在它最終的正確位置。

3.遞迴：分別將「比 **pivot** 小」及「比 **pivot** 大」

兩部分重複上述步驟，直到新序列的長度小於等於 **1**，無法繼續分割為止，此時排序完成。



以上這張圖大概講述了第一到第三步驟，圖**(1)**就是在選擇 **pivot**，圖**(2)**在講第二步驟把 **5** 大的放在右邊，比 **5** 小的放到左邊，剩下的圖就是分別執行遞迴直到排序完成。

用虛擬碼的方式呈現:

```
1   function sort(list)
2       if list.length < 2
3           return list
4       end if
5       pivot = 從list取出一基準點
6       var less, greater, result
7       for i = 0;i < list.length;++i
8           if list[i] > pivot
9               greater.add(list[i])
10          else
11              less.add(list[i])
12          end if
13      end for
14      result.add(sort(less))
15      result.add(pivot)
16      result.add(sort(greater))
17      return result;
18  end function
```

第 5 行 = 第一步驟

7 到 13 行 = 第二步驟

14 行開始 = 第三步驟

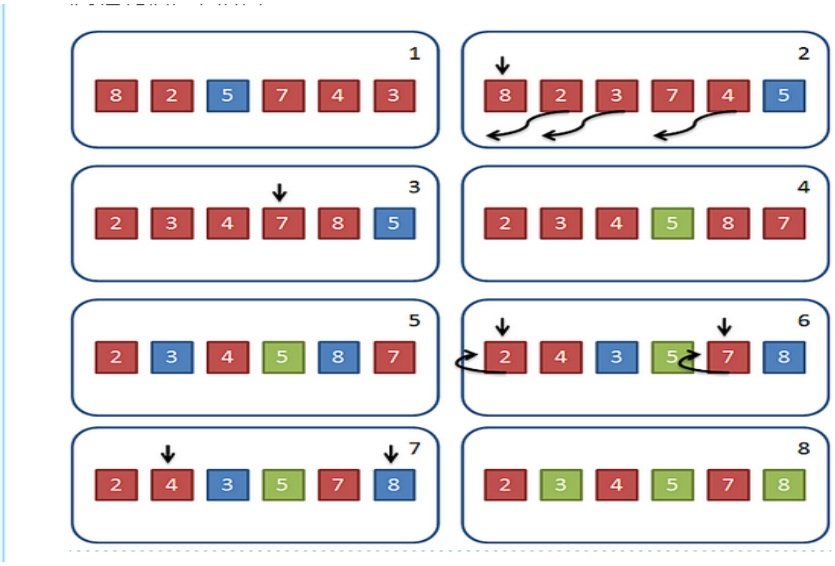
	Complexity
Worst	$O(n^2)$
Best	$O(n \log n)$
Average	$O(n \log n)$
Worst space	$O(\log n)$ or $O(n)$ auxiliary

而這種方式的排法複雜度如上圖。

除了最基礎的分法之外，還有一種使用較少空間的原地(In-place)方式。

In-place 的方式主要概念是將基準點暫時移到最右邊，小於基準的移至數列一端並記錄遞增索引，最後將基準點換回索引位置，過程依照以下步驟進行(遞增為例)：

1. 數列中選擇一元素作為基準點(pivot)，並與最右邊的元素交換位置。
2. 建立一索引指向最左邊元素。
3. 小於基準的元素與索引位置的元素交換位置，每次交換後遞增索引。
4. 完成後將基準點與索引位置的元素交換位置。
5. 基準點左邊和右邊視為兩個數列，並重複做以上動作直到數列剩下一個或零個元素。



從圖(2)開始給一個標記每次遇到小於 5 的數字時，標記和數字換位子，同時標記++，到圖(5)時分成 left 和 Right 兩大部分，在分別對其部分重複上述動作。

用虛擬碼的方式呈現:

```
1   function sort(list, left, right)
2       if right <= left
3           return
4       end
5       pivotIndex = 從list取出一基準點
6       pivot = list[pivotIndex]
7       swap(list[pivotIndex], list[right])
8       swapIndex = left
9       for i = left;i < right;++i
10          if list[i] <= pivot
11              swap(list[i], list[swapIndex])
12              ++swapIndex
13          end if
14      end for
15      swap(list[swapIndex], list[right])
16      sort(list, left, swapIndex - 1);
17      sort(list, swapIndex + 1, right);
18  end function
```

從程式碼 9-14 行可以更清楚的了解上面文字描寫的過程，在分完兩大區域後執行函式遞迴，直到全部分完。

使用(In-place)方式可以更快完成排序，而 C 函式庫的 qsort 也是用此改良的方法做基礎來做排序，而它的空

間複雜度更是降到了 **O(log(n))**。

使用 **qsort():**

在使用之前先看看他的函式定義。

```
void qsort(void* base,size_t num,size_t width,int(__cdecl*compare)(const void*,const void*));
```

引數代表=>

1 待排序陣列，排序之後的結果仍放在這個陣列中

2 陣列中待排序元素數量

3 各元素的佔用空間大小（單位為位元組）

4 指向函式的指標，用於確定排序的順序（需要使用者自定義一個比較函式）

其中自訂義的函式回傳值式 **int**，回傳大於 **0** 的話表示要用升序排列。

使用的範例**(int** 排序**):**

```
int cmp_int(const void* _a , const void* _b)
{
    int* a = (int*)_a;    //強制型別轉換
    int* b = (int*)_b;
    return *a - *b;
}
```

這邊要求回傳值要是 **int**，也就是要指標代表的值，所以要先強制轉型後放到等式左邊在相減。

現在來實際測驗 **qsort()**時間是否有比較快速，測式的方法為之前 **HW5** 助教提供的測量時間的程式碼加上使用 **qsort()**排序。

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define TEST_DATA_CNT 50000
int cmp_int(const void* _a , const void* _b)
{
    int* a = (int*)_a;
    int* b = (int*)_b;
    return *a - *b;
}
int main()
{
    int i;
    int test_data[TEST_DATA_CNT + 5];
    srand(1);
    for (i = 0; i < TEST_DATA_CNT; i++)
    {
        test_data[i] = rand();
    }
    struct timeval start;
    struct timeval end;
    unsigned long diff;

    gettimeofday(&start, NULL);
    qsort(test_data,TEST_DATA_CNT,sizeof(test_data[0]),cmp_int);
    gettimeofday(&end, NULL);

    diff = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec; // 實際的時間差
    printf("Using qsort Sorting performance %ld us (equal %f sec)\n", diff, diff / 1000000.0);

    return 0;
}
```

這裡結合上一次 **HW** 的內容，給 **50000** 筆測資來檢測結果如下:

```
u07510062@csie2:~ % gcc ./BubbleSort.c
u07510062@csie2:~ % ./a.out
Sorting performance 11940627 us (equal 11.940627 sec)
u07510062@csie2:~ % gcc ./SelectionSort.c
u07510062@csie2:~ % ./a.out
Sorting performance 5179708 us (equal 5.179708 sec)
u07510062@csie2:~ % gcc ./InsertionSort.c
u07510062@csie2:~ % ./a.out
Sorting performance 3125063 us (equal 3.125063 sec)
u07510062@csie2:~ %
```

```
u07510062@csie2:~/HW % ./a.out
Using qsort Sorting performance 11048 us (equal 0.011048 sec)
```

在之前使用到的排序最快的是插入排序的 **3.12** 秒，不過使用 **qsort** 後時間，時間僅 **0.011** 秒，可以說是非常快的，雖然考慮到 **qsort** 是內建的函式庫而之前的作業是自己寫一個函式，不過這個時間差還是可以充分證明 **qsort** 的強大所在。

以下影片比較了各排序的時間視覺化，也可以出份展示 **qsort** 為何被廣泛應用。

https://www.youtube.com/watch?v=BeoCbJPuvSE

結論:

在之前看過的某一支影片曾經說到，現代演算法的應用不式追求 **100%**的正確率，而式趨近於 **100%**的正確率，人們不太關注 **90** 多%和完全正確有其區別，因為這樣可以有更高的效率去完成一件事情，**qsort** 就是一個很好的例子，雖然它並不是一定都這麼快速，但還是保證了它的通用及快速性，**qsort** 其他的競爭對手也是如此，其他的高效排序例如 **Heapsort**（堆積排序）或 **Mergesort(**合併排序**)**，也都有其優缺點，但他們都能快速地完成他們的任務，所以能夠被廣泛地運用，我們可以很開心地使用 **qsort()**，在大多數我們想要做

各種排序的時候正是如上述報告所展示的它的便利所在。

參考資料

https://rust-algo.club/sorting/mergesort/index.html

https://emn178.pixnet.net/blog/post/88613503

https://www.itread01.com/content/1549814064.html

https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/460267/

https://zh.wikipedia.org/wiki/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F

https://www.itread01.com/content/1547261714.html