**作業一 排序演算法比較**

1. **測試環境**

作業系統：Microsoft Windows 10 版本20H2(OS組建19042.928)

測試環境：visual studio code

處理器：intel core i5-8265U

RAM：8GB

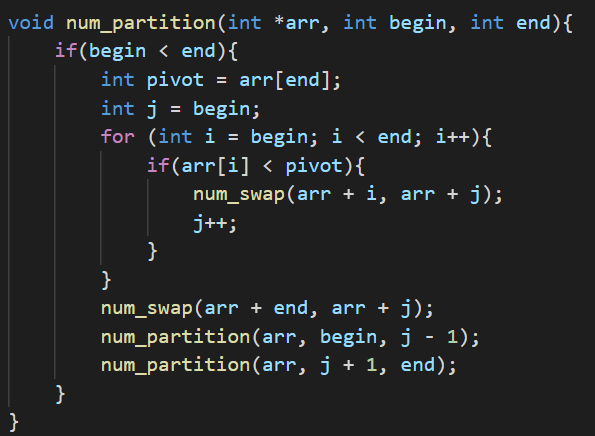
Git連結：kingslayer929/409410017-sort-functions-compare (github.com)

1. **解釋各種排序演算法 (quick sort, merge sort, heap sort)**

* **Quick Sort**

1. **先給定一個陣列(arr)，並給予欲排序的起點(begin)與終點(end)**
2. **將end設為pivot**
3. **此時設定一變數j位於begin，接著依序遊走從begin到end，若找到比pivot小的就和j這個位置的數字交換，若較大則繼續遊走，直到結束。**
4. **最後將end(即為pivot)和j這個位置的數字交換，就可得到比pivot小的數字都在pivot前面，比pivot大的數字都在pivot後面。**
5. **以pivot進行分割，將arr拆成兩部分，重複1~4直到不可再分割，即排序完成**

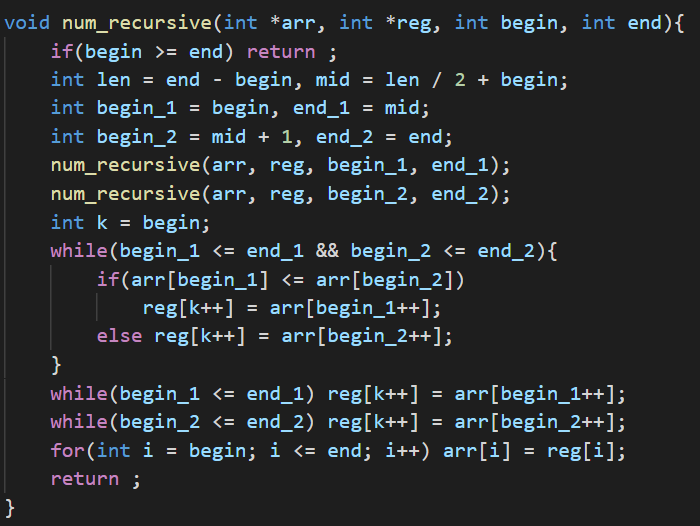
**下為範例程式碼：**

****

* **Merge Sort**

1. **給定陣列(arr)、起點(begin)、終點(end)與merge sort中最重要的佔存空間(reg)**
2. **將陣列從中間拆成兩部分，並再呼叫兩次自己，一次呼叫前半段，一次呼叫後半段，因為拆成兩部分，所以排序區間減少了一半**
3. **一直進行1~2直到不可再分割**
4. **此時拿取兩個陣列將數字穿插塞進reg陣列內，使得此區間內的數皆由小到大，最後將reg內的數複製回arr內**
5. **重複4直到組合完先前分割的陣列，此時即排序完成**

**以下為範例程式碼：**

****

* **Heap Sort**

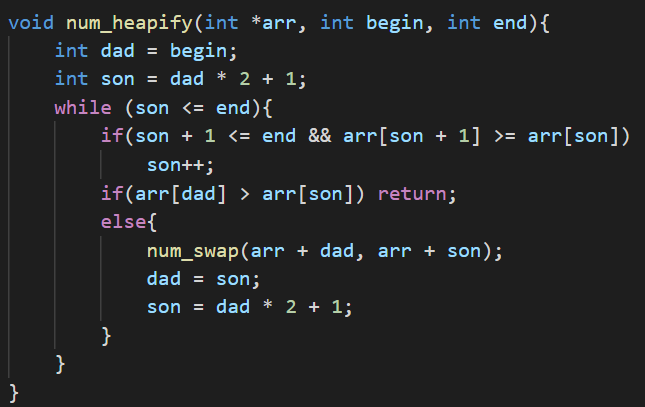
**此排序分成兩的部分，第一部分為建構priority queue，第二部分為排序，兩個部分皆會使用到heapify這個函式，因此從heapify開始說起。**

**在此之前，必須了解二元樹的概念，當一個二元樹為平衡樹時，可用陣列表示之，即父節點的兩個子節點分別為父節點\*2加上1和父節點\*2加上2。**

**Priority queue是一個平衡的二元樹(義即不會有樹的一邊特別長、特別短的情況發生)，且所有父節點大於其子節點**

1. **給定陣列(arr)、起點(begin)、終點(end)**
2. **起始父節點(dad)為begin，起始子節點(son)為dad\*2+1**
3. **當子節點未超出end的範圍時，做以下事情**
4. **若son+1的數字比son還大，son變成son+1**
5. **如果dad的數字大於son的，因符合dad大於son的要求，因此直接跳出迴圈**
6. **將dad與son內的數字交換，並使son成為下一個dad，即dad=son，此時son=dad\*2+1**

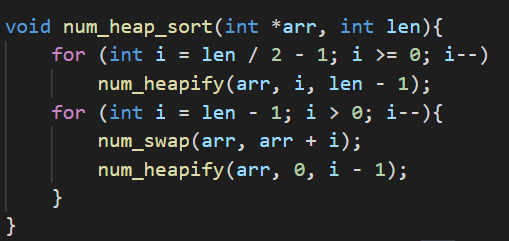
**以下為範例程式碼：**

****

**接下來為建立priority queue與排序**

1. **給定陣列(arr)宇欲排序長度(len)**
2. **從最後一個父節點開始往回做heapify的動作，使得整個陣列中的每個父節點都大於子節點**
3. **把陣列分為已排序區與未排序區，未排序區為priority queue，已排序區的初始大小為0。**
4. **將priority queue的首項(root)與末項互換，使得最後面變成最大的數，此時已排序區大小增加1，而未排序區大小減少1，重新調整未排序區成為priority queue**
5. **重複4直到未排序區大小為0即排序完成**

**以下為範例程式碼：**

****

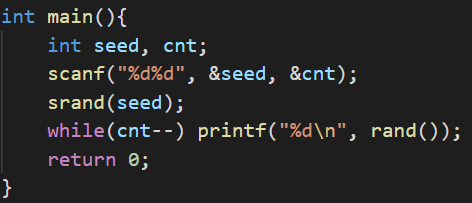
* **利用以上三種排序法對字串進行排序**

將字串存於指標陣列中，此指標陣列中的指標皆指向各個字串，排序時對指標陣列進行排序即可，不必動到字串。

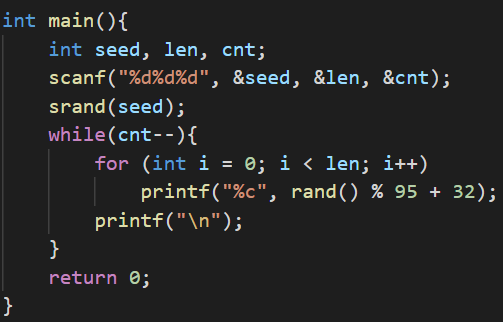
1. **建立 Data 方法及數量**

* **建立方式**

**輸入亂數種子碼與資料筆數，並將資料輸出至指定文件中**

****

**對於生成字串，則要求多輸入一個數字代表欲生成字串之長度**

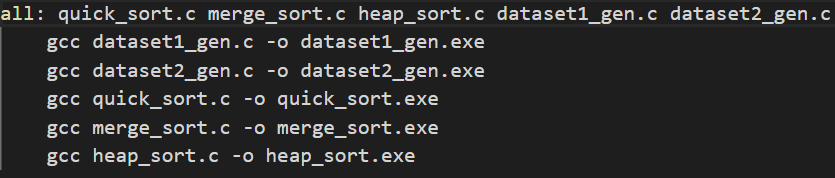
****

* **Data數量**

**1,000,000筆以rand()產生之資料集**

1. **makefile**

**利用makefile快速編譯所有程式碼**

****

1. **測量排序時間方式**
2. **執行**

**以下圖為例，執行quick\_sort.exe，利用dataset1.txt的資料集，並將輸出到result.txt**

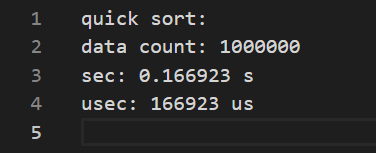
****

**若要對字串進行排序，則要在”./quick\_sort.exe”後加上”-s”**

****

1. **結果顯示**

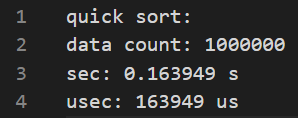
**下圖為輸出到result.txt的結果**

****

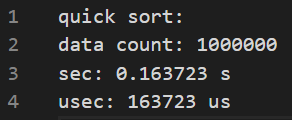
1. **實驗結果**
2. **Quick sort**
   * **數字測試**

**平均時間：0.164168666 sec**

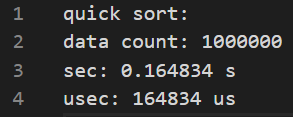
第一次測試：0.163949 sec

****

第二次測試：0.163723 sec

****

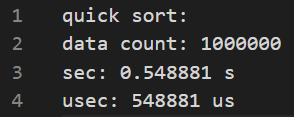
第三次測試：0.164834 sec

****

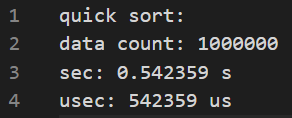
* + **字串測試**

**平均時間：0.543937666 sec**

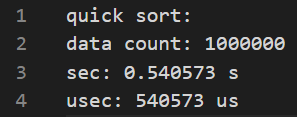
第一次測試：0.548881 sec

****

第二次測試：0.542359 sec

****

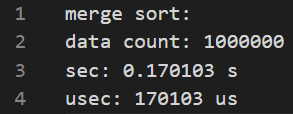
第三次測試：0.540573 sec

****

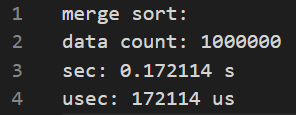
1. **Merge sort**
   * **數字測試**

**平均時間：0.170419333 sec**

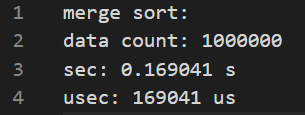
第一次測試：0.170103 sec



第二次測試：0.172114 sec



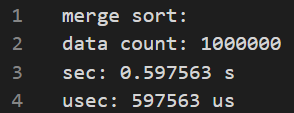
第三次測試：0.169041 sec



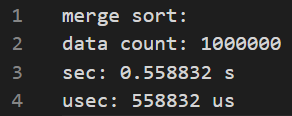
* + **字串測試**

**平均時間：0.570026666 sec**

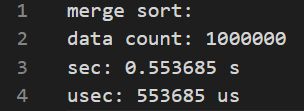
第一次測試：0.597563 sec



第二次測試：0.558832 sec



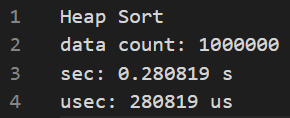
第三次測試：0.553685 sec



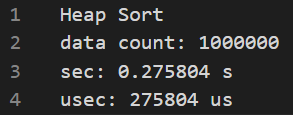
1. **Heap sort**
   * **數字測試**

**平均時間：0.285228333 sec**

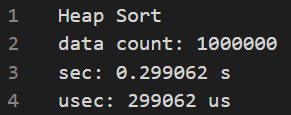
第一次測試：0.280819 sec



第二次測試：0.275804 sec



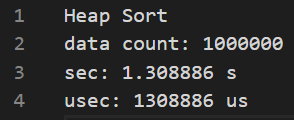
第三次測試：0.299062 sec



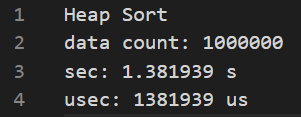
* + **字串測試**

**平均時間：1.3517547**

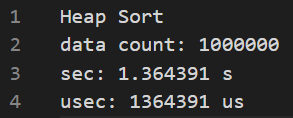
第一次測試：1.308886 sec



第二次測試：1.381939 sec



第三次測試：1.3641391 sec



1. **總結**
2. **時間複雜度**

**三者皆為O(nlogn)，quick sort與merge sort的速度差不多，heap sort則慢了很多，因為雖然heap sort排序時是O(nlogn)，但在建立heap的時候就要O(nlogn)了，所以才會慢了大約一倍左右。**

1. **空間複雜度**

**Quick sort是O(logn)~O(n)，因為呼叫函式進行堆疊需額外空間，且有遞迴的深度差異。**

**heap sort是O(1)，只使用原本的陣列，進行原地置換。**

**Merge sort是 O(n)，須多花一倍的空間進行排序。**

1. **參考資料**

合併排序-維基百科

https://zh.wikipedia.org/zh-tw/%E5%BD%92%E5%B9%B6%E6%8E%92%E5%BA%8F

堆積排序-維基百科

https://zh.wikipedia.org/zh-tw/%E5%A0%86%E6%8E%92%E5%BA%8F