作業一 排序演算法比較

一、 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)

二、 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：

```
void num_partition(int *arr, int begin, int end){
    if(begin < end){
        int pivot = arr[end];
        int j = begin;
        for (int i = begin; i < end; i++){
            if(arr[i] < pivot){
                num_swap(arr + i, arr + j);
                j++;
            }
        }
        num_swap(arr + end, arr + j);
        num_partition(arr, begin, j - 1);
        num_partition(arr, j + 1, end);
    }
}
```

- **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：

```
void num_recursive(int *arr, int *reg, int begin, int end){
    if(begin >= end) return ;
    int len = end - begin, mid = len / 2 + begin;
    int begin_1 = begin, end_1 = mid;
    int begin_2 = mid + 1, end_2 = end;
    num_recursive(arr, reg, begin_1, end_1);
    num_recursive(arr, reg, begin_2, end_2);
    int k = begin;
    while(begin_1 <= end_1 && begin_2 <= end_2){
        if(arr[begin_1] <= arr[begin_2])
            reg[k++] = arr[begin_1++];
        else reg[k++] = arr[begin_2++];
    }
    while(begin_1 <= end_1) reg[k++] = arr[begin_1++];
    while(begin_2 <= end_2) reg[k++] = arr[begin_2++];
    for(int i = begin; i <= end; i++) arr[i] = reg[i];
    return ;
}
```

- **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
   a. 若 son+1 的數字比 son 還大，son 變成 son+1
   b. 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
   c. 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：

```
void num_heapify(int *arr, int begin, int end){
    int dad = begin;
    int son = dad * 2 + 1;
    while (son <= end){
        if(son + 1 <= end && arr[son + 1] >= arr[son])
            son++;
        if(arr[dad] > arr[son]) return;
        else{
            num_swap(arr + dad, arr + son);
            dad = son;
            son = dad * 2 + 1;
        }
    }
}
```

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：

```
void num_heap_sort(int *arr, int len){
    for (int i = len / 2 - 1; i >= 0; i--)
        num_heapify(arr, i, len - 1);
    for (int i = len - 1; i > 0; i--){
        num_swap(arr, arr + i);
        num_heapify(arr, 0, i - 1);
    }
}
```

- **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。

三、 **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

```
int main(){
    int seed, cnt;
    scanf("%d%d", &seed, &cnt);
    srand(seed);
    while(cnt--) printf("%d\n", rand());
    return 0;
}
```

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

```
int main(){
    int seed, len, cnt;
    scanf("%d%d%d", &seed, &len, &cnt);
    srand(seed);
    while(cnt--){
        for (int i = 0; i < len; i++)
            printf("%c", rand() % 95 + 32);
        printf("\n");
    }
    return 0;
}
```

- **Data 數量**
  1,000,000 筆以 rand()產生之資料集

四、 **makefile**

利用 **makefile** 快速編譯所有程式碼

```
all: quick_sort.c merge_sort.c heap_sort.c dataset1_gen.c dataset2_gen.c
    gcc dataset1_gen.c -o dataset1_gen.exe
    gcc dataset2_gen.c -o dataset2_gen.exe
    gcc quick_sort.c -o quick_sort.exe
    gcc merge_sort.c -o merge_sort.exe
    gcc heap_sort.c -o heap_sort.exe
```

五、 測量排序時間方式

1. 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**

```
$ ./quick_sort.exe < dataset1.txt > result.txt
```

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

```
$ ./quick_sort.exe -s < dataset2.txt > result.txt
```

2. 結果顯示

下圖為輸出到 **result.txt** 的結果

```
1   quick sort:
2   data count: 1000000
3   sec: 0.166923 s
4   usec: 166923 us
5
```

六、 實驗結果

1. **Quick sort**

- **數字測試**

  平均時間：**0.164168666 sec**

  第一次測試：0.163949 sec

```
quick sort:
data count: 1000000
sec: 0.163949 s
usec: 163949 us
```

  第二次測試：0.163723 sec

```
quick sort:
data count: 1000000
sec: 0.163723 s
usec: 163723 us
```

  第三次測試：0.164834 sec

```
quick sort:
data count: 1000000
sec: 0.164834 s
usec: 164834 us
```

- **字串測試**

  平均時間：**0.543937666 sec**

  第一次測試：0.548881 sec

```
quick sort:
data count: 1000000
sec: 0.548881 s
usec: 548881 us
```

  第二次測試：0.542359 sec

```
quick sort:
data count: 1000000
sec: 0.542359 s
usec: 542359 us
```

第三次測試：0.540573 sec

```
quick sort:
data count: 1000000
sec: 0.540573 s
usec: 540573 us
```

2. **Merge sort**
   - **數字測試**

     平均時間：0.170419333 sec

     第一次測試：0.170103 sec

```
merge sort:
data count: 1000000
sec: 0.170103 s
usec: 170103 us
```

     第二次測試：0.172114 sec

```
merge sort:
data count: 1000000
sec: 0.172114 s
usec: 172114 us
```

     第三次測試：0.169041 sec

```
merge sort:
data count: 1000000
sec: 0.169041 s
usec: 169041 us
```

   - **字串測試**

     平均時間：0.570026666 sec

     第一次測試：0.597563 sec

```
merge sort:
data count: 1000000
sec: 0.597563 s
usec: 597563 us
```

第二次測試：0.558832 sec

```
merge sort:
data count: 1000000
sec: 0.558832 s
usec: 558832 us
```

第三次測試：0.553685 sec

```
merge sort:
data count: 1000000
sec: 0.553685 s
usec: 553685 us
```

3. **Heap sort**
   - **數字測試**

     平均時間：0.285228333 sec

     第一次測試：0.280819 sec

```
Heap Sort
data count: 1000000
sec: 0.280819 s
usec: 280819 us
```

     第二次測試：0.275804 sec

```
Heap Sort
data count: 1000000
sec: 0.275804 s
usec: 275804 us
```

     第三次測試：0.299062 sec

```
Heap Sort
data count: 1000000
sec: 0.299062 s
usec: 299062 us
```

- 字串測試

  平均時間：**1.3517547**

  第一次測試：1.308886 sec

```
1    Heap Sort
2    data count: 1000000
3    sec: 1.308886 s
4    usec: 1308886 us
```

  第二次測試：1.381939 sec

```
1    Heap Sort
2    data count: 1000000
3    sec: 1.381939 s
4    usec: 1381939 us
```

  第三次測試：1.3641391 sec

```
1    Heap Sort
2    data count: 1000000
3    sec: 1.364391 s
4    usec: 1364391 us
```

七、 總結

1. 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。

2. 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。

八、 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