Introduction to Algorithms HW3

109511207 蔡宗儒

Pseudo Code

```
LEFT(i)
1.  return 2*i

RIGHT(i)
1.  return 2*i+1

MAX-HEAPIFY(A,i)
1.  l = LEFT(i)
2.  r = RIGHT(i)
3.  if l≦A.heap-size and A[l] > A[i]
4.          largest = l
5.  else largest = i
6.  if r≦A.heap-size and A[r] > A[largest]
7.          largest = r
8.  if largest≠i
9.          exchange A[i] with A[largest]
10.         MAX-HEAPIFY(A,largest)

BUILD-MAX-HEAP(A)
1.  A.heap-size = A.length
2.  for i = floor(A.length/2) downto 1
3.          MAX-HEAPIFY(A,i)

HEAPSORT(A)
1.  BUILD-MAX-HEAP(A)
2.  for i = A.length downto 2
3.          exchange A[1] with A[i]
4.          A.heap-size = A.heap-size-1
5.          MAX-HEAPIFY(A,1)
```

Introduction of Code

maxHeapify

```
void maxHeapify(Heap &h,int i)
{
    int largest;

    // left node index
    int l = 2*i+1;

    // right node index
    int r = 2*i+2;

    // decide the largest node is i, left or right
    if(l <= h.size-1 && h.v[l] > h.v[i])
        largest = l;
    else
        largest = i;
    if(r <= h.size-1 && h.v[r] > h.v[largest])
        largest = r;

    // if the largest node is not i, then swap and do maxHeapify for previous largest node
    if (largest != i )
    {
        int t = h.v[i];
        h.v[i] = h.v[largest];
        h.v[largest] = t;
        maxHeapify(h,largest);
    }
}
```

因為講義給的 pseudo code 是以 index 為 1 開始的陣列，而我是以 index 從 0 開始，所以 left child = 2*i+1、right child = 2*i+2，接著做法和課本給的 pseudo code 一樣。首先找出 index = i(parent)、l(left child)、r(right child)，這三個 node 的最大值，如果 subtree 的 parent 不是最大值的話就跟是最大值的 child swap，這樣目前的 subtree 便能符合 max heap(parent 的值需大於兩個 child 的值)，接著 recursively do maxHeapify()往下遍歷，檢查新的 subtree 是否符合 max heap，最後只要有被 maxHeapify()檢查的 subtree，都會符合 max heap。

buildMaxHeap

```
void buildMaxHeap(Heap &h)
{
    h.size = h.v.size();
    for(int i=h.v.size()/2-1;i>=0;i--)
        maxHeapify(h,i);
}
```

因為沒有 child 的 node 一定不會違反 max heap，所以只要對所有有 child 的 node 做 maxHeapify()，便可以將一陣列變為 max heap，也就是對 index = ⌊N/2⌋-1 ~ 0 的 node 由下往上做 maxHeapify()。

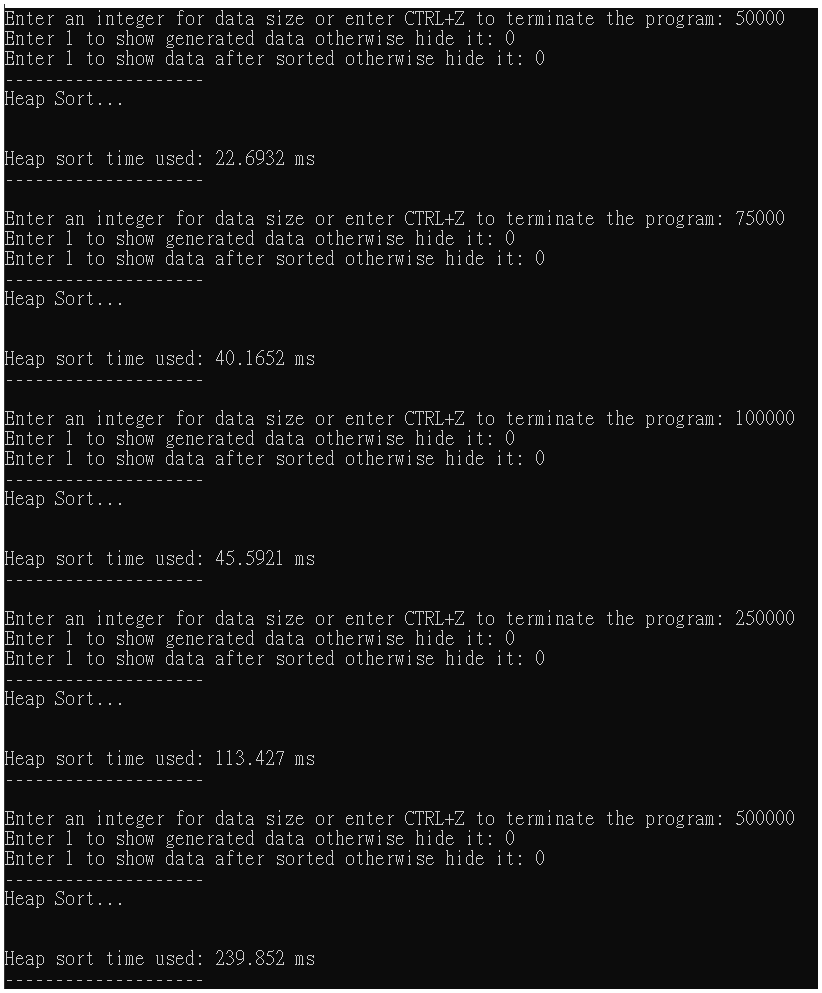
heapSort

```
void heapSort(Heap &h)
{
    buildMaxHeap(h);
    for(int i=h.v.size()-1;i>=1;i--)
    {
        int t = h.v[0];
        h.v[0] = h.v[i];
        h.v[i] = t;
        h.size--;
        maxHeapify(h,0);
    }
}
```

藉由 buildMaxHeap()，可以將任一陣列變成 max heap，若要將此陣列由小到大 sort 的話只需要把 root 跟 heap 的最後一個 node swap，這樣可以讓 heap 的最後一個 node 變為最大值，並假裝最後一個 node 被去除了(即 h.size--)，並對 root 做 maxHeapify()，調整為新的 max heap，重複這些步驟就能將陣列排好序。

Heap Sort 執行結果

```
Enter an integer for data size or enter CTRL+Z to terminate the program: 10
Enter 1 to show generated data otherwise hide it: 0
Enter 1 to show data after sorted otherwise hide it: 0
--------------------
Heap Sort...


Heap sort time used: 0.0022 ms
--------------------

Enter an integer for data size or enter CTRL+Z to terminate the program: 100
Enter 1 to show generated data otherwise hide it: 0
Enter 1 to show data after sorted otherwise hide it: 0
--------------------
Heap Sort...


Heap sort time used: 0.025 ms
--------------------

Enter an integer for data size or enter CTRL+Z to terminate the program: 1000
Enter 1 to show generated data otherwise hide it: 0
Enter 1 to show data after sorted otherwise hide it: 0
--------------------
Heap Sort...


Heap sort time used: 0.3631 ms
--------------------

Enter an integer for data size or enter CTRL+Z to terminate the program: 10000
Enter 1 to show generated data otherwise hide it: 0
Enter 1 to show data after sorted otherwise hide it: 0
--------------------
Heap Sort...


Heap sort time used: 4.7702 ms
--------------------

Enter an integer for data size or enter CTRL+Z to terminate the program: 25000
Enter 1 to show generated data otherwise hide it: 0
Enter 1 to show data after sorted otherwise hide it: 0
--------------------
Heap Sort...


Heap sort time used: 12.3864 ms
--------------------
```



上二圖為 Insertion Sort 在 input size 分別為 10, 100, 1000, 10000, 25000, 50000, 75000, 100000, 250000, 500000 所花的時間。

Run Time Comparison

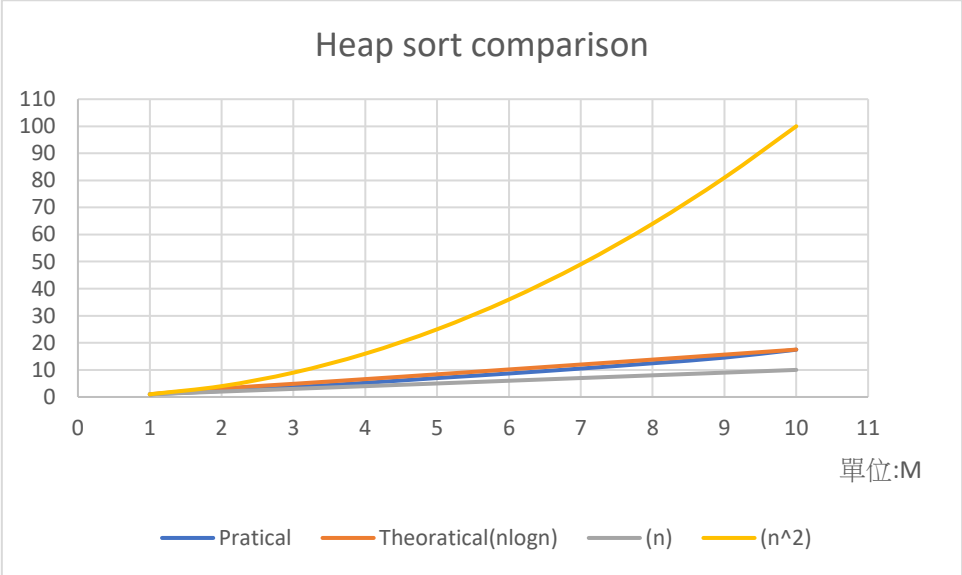
Input Size	Heapsort	Insertion Sort	Merge Sort
10	0.0022ms	0.0008ms	0.0585ms
100	0.025ms	0.0224ms	0.1942ms
1000	0.3631ms	1.8397ms	2.0226ms
10000	4.7702ms	154.844ms	16.856ms
25000	12.3864ms	909.187ms	41.2024ms
50000	22.6932ms	3651.05ms	78.3804ms
75000	40.1652ms	8113.59ms	113.191ms
100000	45.5921ms	14399.6ms	151.412ms
250000	113.427ms	110209ms	401.183ms
500000	239.852ms	529504ms	816.087ms

以上為 heapsort 和 HW1 做的 insertion sort 跟 merge sort 的 run time 做比較，可以發現在 size 較小的 case 下(約 n = 100 以下)，insertion sort 依然是有著最快的

run time；而在 size 較大的 case 下，heap sort 有較佳的 run time。

而因為 size 較小時可能會有比較大的誤差，於是我又每隔 1000000 跑了 n=1000000~10000000 的情況，根據執行後的結果，並以 n=1000000 做為 unit size，可以做出以下圖表。

Size	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	9000000	10000000
Practical Run Time	487.646	1089.91	1772.08	2549.74	3406.25	4234.28	5123.85	6075.2	7093.57	8514.33
Pratical	1	2.23504	3.63395	5.22867	6.98509	8.6831	10.5073	12.4582	14.5466	17.46006
Theoratical(nlogn)	1	3.15051	4.85784	6.60206	8.37371	10.1672	11.9789	13.8062	15.647	17.5
(n)	1	2	3	4	5	6	7	8	9	10
(n^2)	1	4	9	16	25	36	49	64	81	100



可以發現執行的實際結果也接近理論值 O(logn)。

Analysis heapsort

因為 maxHeapify()從上至下遍歷 tree，而大小為 n 的 nearly complete binary tree 的高度為$\lfloor \log_2 n \rfloor + 1$，所以所需的最多步驟數為$\lfloor \log_2 n \rfloor + 1$，maxHeapify 相對應的時間複雜度為 O(logn)。

而 buildMaxHeap()的時間複雜度並不是 O(nlogn)。因為建立一個 heap 時，除了 root 以外的 node 都不會往下移動 logn 次，最底層的 node(共 n/2 個)不會往下 swap，倒數第二層的 node(共 n/2 個)最多只會往下 swap 1 次，依此類推。所以可以得到以下式子：

$$\frac{n}{2} * 0 + \frac{n}{4} * 1 + \frac{n}{8} * 2 + \cdots + \log_2 n$$

所以 buildMaxHeap()的時間複雜度實際上是 O(n)

最後在 heapSort()中，使用了 buildMaxHeap()，然後用 for loop call maxHeapify() maintain max heap property。因為每次 heap 的 node 都會少 1，所以第一次

maxHeapify()需要 log(n-1)的時間、第二次需要 log(n-2)的時間，依此類推，可以得到以下式子：

$$\log(n-1) + \log(n-2) + \cdots + \log(1) = \log\big((n-1)!\big)$$

進一步簡化可以得到時間複雜度為 O(nlogn)，用更直覺更簡單的方法分析的話是因為 call 了 maxHeapify() n-1 次，所以時間複雜度為 O(nlogn)。

而因為不論 best、average 還是 worst case，只會影響 buildMaxHeap()的時間，但 heap sort 主要被 for loop 內 call maxHeapify()的次數 dominant 住，所以不論 best、average 還是 worst case，時間複雜度皆為 O(nlogn)。

Conclusion

Heap sort

Heap sort 的時間複雜度為 O(nlogn)，空間複雜度為 O(1)。優點是它在 worst case 下的 performance 仍然很好，且屬於 in-place，不需要花額外的空間來進行排序，更重要的是它很好實現。但缺點是它需要頻繁地 swap，所以實際 performance 可能會較差，且 heap sort 不是一種穩定的排序法。

Insertion sort

Insertion sort 時間複雜度為 $O(n^2)$。優點是它很好實現，適合用在 size 較小或是已幾乎排序好的序列，且空間複雜度小，屬於 in-place，不需要花額外的空間來進行排序。缺點則是在 average 和 worst case 下的時間複雜度較其他兩者差。

Merge sort

Merge Sort 的時間複雜度為 O(nlogn)，空間複雜度為 O(n)。它的優點是較適合處理 size 較大的序列，它也是穩定的排序法，可以確保排序後相同 data 的相對位置不會改變。但缺點是需要額外的空間來存中間過程，在空間有限的情況下可能較不適用。

總結來說，heap sort 比較適合用於空間有限的情況下；insertion sort 比較適合用於 size 較小或是已大致排好序的序列；merge sort 比較適合用於 size 較大的序列。

Reference

https://reurl.cc/4QlW82
https://reurl.cc/klpaQb