Homework 3

0811324 胡政煒

```
Enter the input size: 100
Insertion Sort with size 100 took 0 microseconds.
Merge Sort with size 100 took 0 microseconds.
Heap Sort with size 100 took 0 microseconds.

Enter the input size: 1000
Insertion Sort with size 1000 took 1006 microseconds.
Merge Sort with size 1000 took 1015 microseconds.
Heap Sort with size 1000 took 0 microseconds.

Enter the input size: 10000
Insertion Sort with size 10000 took 27942 microseconds.
Merge Sort with size 10000 took 1014 microseconds.
Heap Sort with size 10000 took 1007 microseconds.

Enter the input size: 100000
Insertion Sort with size 100000 took 2542421 microseconds.
Merge Sort with size 100000 took 17944 microseconds.
Heap Sort with size 100000 took 17894 microseconds.

Enter the input size: 200000
Insertion Sort with size 200000 took 10158660 microseconds.
Merge Sort with size 200000 took 36948 microseconds.
Heap Sort with size 200000 took 38701 microseconds.
```

Heap Sort 和 Merge Sort 的時間複雜度均為 O(nlogn)，比 Insertion Sort 的時間複雜度 O(n^2)快得多。因此在處理大量數據排序時，Heap Sort 和 Merge Sort 都比 Insertion Sort 更快。根據程序結果，當數組大小達到 10000 時，Merge Sort 和 Heap Sort 比 Insertion Sort 快一位數字。當數組大小達到 100000 時，Merge Sort 和 Heap Sort 比 Insertion Sort 快兩位數字。換句話說，Heap Sort 和 Merge Sort 的速度比 Insertion Sort 快得多。

Heapsort：
時間複雜度：O(n log n)
最壞情況：O(n log n)
最佳情況：O(n log n)
優點：在大多數情況下，Heapsort 的效率與 Merge Sort 相當，但它使用的空間更少，因為它是一種「原址排序」。
缺點：由於其較高的時間複雜度，Heapsort 在排序小數組時可能比 Insertion Sort 還要慢。
適用時機：當數組很大而且空間有限時，Heapsort 可能是一個更好的選擇。

Merge Sort：
時間複雜度：O(n log n)
最壞情況：O(n log n)
最佳情況：O(n log n)
優點：穩定、效率高、不受輸入數據分布的影響。
缺點：使用額外的空間，空間複雜度較高。
適用時機：當需要對大量數據進行排序時，且允許使用額外的空間時，Merge Sort 是一個很好的選擇。

Insertion Sort：
時間複雜度：O(n^2)
最壞情況：O(n^2)
最佳情況：O(n)
優點：對於小數組，Insertion Sort 的效率比 Heapsort 和 Merge Sort 高。
缺點：時間複雜度高，當數據量很大時效率很低。
適用時機：當數據量較小時，Insertion Sort 可以是一個很好的選擇。

總體來說，Heapsort 在時間複雜度和空間複雜度上都優於 Merge Sort，但當數據量很小時，Insertion Sort 是最好的選擇。由於 Merge Sort 不受輸入數據分布的影響，因此它對於大數據集合和較為平均的數據分佈最有效。Insertion Sort 則對於已經部分有序的數據集合最有效。

至於 Heapsort 的最好和最壞情況，它們都是 O(n log n)。在最壞情況下，即當數組中的元素已經按照降序排列時，每個元素都需要下沉到數組的底部，這需要花費 O(log n) 的時間。因此，在最壞情況下，Heapsort 的時間複雜度為 O(n log n)。

在最好情況下，即當數組中的元素已經按照升序排列時，每個元素都不需要下沉。因此，在最好情況下，Heapsort 的時間複雜度仍然為 O(n log n)。這也說明了為什麼 Heapsort 的最好和最壞情況下的時間複雜度都相同，因為堆排序的時間複雜度只受數據集合中元素的總數影響，而不受輸入數據分布的影響。

Heap Sort & Merge Sort 比較:

一般情況下，Heap Sort 和 Merge Sort 的時間複雜度都是 O(n log n)，且在最壞情況下時間複雜度相同。但在實際應用中，哪一種排序算法更快取決於多種因素，例如數據規模、數據分佈、緩存大小等等。

當數據規模較小且數據分佈比較隨機時，一般情況下 Merge Sort 比 Heap Sort 更快，因為 Merge Sort 通常更好地利用 CPU 緩存，並且在數據較少的情況下，Merge Sort 的常數項比 Heap Sort 小。

然而，當數據規模較大時，Heap Sort 的常數項比 Merge Sort 小，因為 Heap Sort 在進行排序時只需要保持局部性，即只需將數據的一部分加載到緩存中進行操作。而 Merge Sort 在進行合併操作時需要額外的空間來存儲子序列，並且在進行合併操作時需要將數據從緩存中載入，這可能會導致 CPU 緩存不命中的情況。

```
149
150 void maxHeapify(int* arr, int n, int i)
151 {
152     int largest = i;// 初始化最大值的位置爲 i
153     int left = 2*i + 1;// 左子節點的位置
154     int right = 2*i + 2; // 右子節點的位置
155
156     // 找出三個節點中最大值的位置
157     if (left < n && arr[left] > arr[largest])// 如果左子節點比當前節點更大
158         largest = left; // 則更新最大值的索引
159
160     if (right < n && arr[right] > arr[largest])// 如果右子節點比當前節點更大
161         largest = right;// 則更新最大值的索引
162     // 如果最大值不是當前位置 i，交換兩者的值，然後遞迴進行最大堆化
163     if (largest != i) {
164         swap(arr[i], arr[largest]);// 則交換兩個元素，將最大值移到當前節點
165         maxHeapify(arr, n, largest);
166     }
167 }
168
169 void buildMaxHeap(int* arr, int n) // 建立最大堆，n 爲數組大小
170 {
171     for (int i = n / 2 - 1; i >= 0; i--)// 從最後一個非葉子節點開始進行最大堆化，直到根節點位置
172         maxHeapify(arr, n, i);
173 }
174
175 void heap_sort(int* arr, int n) // 堆排序，n 爲數組大小
176 {
177     buildMaxHeap(arr, n);// 先建立最大堆
178
179     for (int i = n-1; i >= 0; i--)  // 從最後一個元素開始，將最大元素移到數組末尾
180     {
181         swap(arr[0], arr[i]);// 將堆頂元素與當前未排序部分的最後一個元素交換
182         maxHeapify(arr, i, 0);// 維護最大堆的性質
183     }
184 }
```

void maxHeapify(int* arr, int n, int i)

第一段程式碼 maxHeapify 是用來維護最大堆的性質。最大堆的性質是指每個父節點的值都比它的子節點的值大，這樣就可以保證根節點為最大值。這個函數接收三個參數：一個數組 arr、數組的大小 n、以及需要維護最大堆性質的節點的索引 i。函數會找出節點 i 的子節點中最大的那個，將其索引存儲在變量 largest 中。然後判斷 largest 是否等於 i，如果不等於，就交換節點 i 和 largest 的值，然後遞迴調用 maxHeapify，對交換後的 largest 節點進行維護最大堆的性質。

void buildMaxHeap(int* arr, int n)

第二段程式碼 buildMaxHeap 用來建立最大堆。從數組的中間開始向前遍歷，調用 maxHeapify 函數，對每個節點進行維護最大堆的性質，直到遍歷完整個數組。由於只需要對數組中的一半元素進行維護最大堆的操作，所以時間複雜度為 O(n)。

void heap_sort(int* arr, int n)

第三段程式碼 heapSort 是堆排序算法的實現。首先調用 buildMaxHeap 建立最大堆，然後從最後一個元素開始遍歷，將其與根節點交換，然後對剩餘的元素重新構建最大堆。重複執行這個操作，直到整個數組排好序為止。在每次操作中，需要對交換後的根節點進行維護最大堆的操作，這是通過調用 maxHeapify 函數實現的。由於需要執行 n 次 maxHeapify 函數，所以時間複雜度為 O(n log n)。