Homework 4

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.
Ramdom Quick Sort with size 100 took 0 microseconds.

Enter the input size: 1000
Insertion Sort with size 1000 took 1051 microseconds.
Merge Sort with size 1000 took 0 microseconds.
Heap Sort with size 1000 took 0 microseconds.
Ramdom Quick Sort with size 1000 took 0 microseconds.

Enter the input size: 10000
Insertion Sort with size 10000 took 27418 microseconds.
Merge Sort with size 10000 took 1999 microseconds.
Heap Sort with size 10000 took 1993 microseconds.
Ramdom Quick Sort with size 10000 took 997 microseconds.

Enter the input size: 100000
Insertion Sort with size 100000 took 2560599 microseconds.
Merge Sort with size 100000 took 18027 microseconds.
Heap Sort with size 100000 took 18721 microseconds.
Ramdom Quick Sort with size 100000 took 12127 microseconds.

Enter the input size: 200000
Insertion Sort with size 200000 took 10113365 microseconds.
Merge Sort with size 200000 took 35821 microseconds.
Heap Sort with size 200000 took 39610 microseconds.
Ramdom Quick Sort with size 200000 took 24675 microseconds.
```

在上份作業中已經報告過 Heapsort 與 Merge Sort 的比較了。因此以下都是針對 Randomized quicksort 與 Merge sort 做比較。

從數據能夠看出 Randomized quicksort 有比 Merge sort 快一些。一般來說，Randomized Quicksort 的時間複雜度為 O(n log n)，而 Merge Sort 的時間複雜度也為 O(n log n)。因此，在最壞情況下，兩者的效率應該相當。

不過，實際上 Randomized Quicksort 的效率會受到選擇基準點的方式和數據分佈的影響。當數據分佈較為亂時，Randomized Quicksort 可能比 Merge Sort 更快；但是，當數據分佈較為有序時，Merge Sort 可能會更快。

因此，兩者的效率優劣取決於具體情況，需要根據實際的應用場景進行選擇。

```cpp
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];  // 將最後一個元素作為基準點
    int i = low - 1;        // i 為小於基準點的元素的最後一個索引
    for (int j = low; j < high; j++) {
        // 如果當前元素小於等於基準點，則將 i 往後移動一位，並交換 i 和 j 所指的元素
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    // 最後將 i 往後移動一位，並交換 i 和 high 所指的元素
    swap(arr[i+1], arr[high]);
    return i+1;  // 回傳基準點所在的索引
}

int RandomizedPartition(int arr[], int low, int high) // 隨機切分，將基準點選擇隨機位置的元素
{
    srand(time(nullptr));// 使用 srand 和 time 函數來設置隨機種子
    int randomIndex = rand() % (high - low + 1) + low;// 隨機選擇一個位置，作為基準點
    swap(arr[randomIndex], arr[high]);// 將選擇的基準點放到區間的最右邊（也就是 high 的位置）
    return partition(arr, low, high);// 調用 partition 函數進行切分
}

void Random_quick_Sort(int arr[], int low, int high)
{
    if (low < high) {
        int pi = RandomizedPartition(arr, low, high);  // 分割 arr[]
        Random_quick_Sort(arr, low, pi-1);  // 對分割後的左半部分排序
        Random_quick_Sort(arr, pi+1, high);  // 對分割後的右半部分排序
    }
}
```

int partition(int arr[], int low, int high)

這個函數用於將數列分為兩部分，一部分比一個選定的基準值（pivot）小，另一部分比它大。具體的步驟如下：

- 將第一個元素當作基準值（pivot）
- 從數列的右端（high）向左查找，找到一個小於等於 pivot 的元素
- 從數列的左端（low）向右查找，找到一個大於 pivot 的元素
- 如果這兩個元素的位置沒有相遇，則將它們交換位置
- 重複上述步驟，直到左右指針重合，此時所有小於等於 pivot 的元素都在它的左側，所有大於 pivot 的元素都在它的右側

int randomizedPartition(int arr[], int low, int high)

這個函數與 partition 函數類似，只是在選定 pivot 值時是從數列中隨機選擇一個元素作為基準值。這樣可以避免遇到最壞情況（即選定的 pivot 值為最大或最小值），從而提高快速排序的效率。

void quicksort(int arr[], int low, int high)
這個函數是快速排序算法的主體，它通過遞迴實現分治策略，將數列不斷分為小的子數列進行排序。具體的步驟如下：

- 選定一個 pivot 值
- 將數列分為兩部分，一部分比 pivot 值小，另一部分比它大
- 對兩個子數列分別遞迴調用 quicksort 函數，直到排序完成

在 randomizedQuicksort 中，快速排序的過程與 Quicksort 類似，但是選擇 pivot 的方式是隨機的，可以在最壞情況下提高排序的效率。因此，randomizedPartition 和 Quicksort 函數的作用與普通快速排序算法中的 partition 和 quicksort 函數相同，只是 pivot 的選擇方式不同。

Randomized Quicksort

WORST CASE

最壞情況發生在每次切分都選擇了當前區間的最大值或最小值作為基準點。這樣，每次切分後的兩個子區間分別為 [low, pivotIndex-1] 和 [pivotIndex+1, high]，其大小分別為 n-1 和 0。

在這種情況下，QuickSort 的遞迴深度為 n-1，因為每次切分後的子區間大小都只減少了 1。而且，由於基準點的選擇方式是隨機的，因此最壞情況的機率很小，通常情況下 Randomized Quicksort 的表現都比普通的 Quicksort 要好。

因此，Randomized Quicksort 的最壞時間複雜度為 O(n^2)，但是在實際應用中發生的機率很小。平均情況下，Randomized Quicksort 的時間複雜度為 O(n log n)。

BEST CASE

最佳情況發生在每次切分都選擇了當前區間的中位數作為基準點。這樣，每次切分後的兩個子區間大小都大致相等，使得 QuickSort 的效率達到最優。

當然，在實際應用中，這種情況發生的機率很小，因為基準點的選擇方式是隨機的。不過，通過使用隨機選擇基準點的方式，可以在很大程度上減少最壞情況的發生機率，從而提高 Randomized Quicksort 的效率。

因此，Randomized Quicksort 的最佳情況時間複雜度為 O(n log n)，與平均情況下的時間複雜度相同。

AVERAGE CASE

平均情況下，時間複雜度為 O(n log n)。這是因為在每次切分中，基準點都是隨機選擇的，並且每次選擇的位置都是等概率的。

在這種情況下，每次切分的期望子區間大小都大致相等，並且總共需要切分 log n 次，因此時間複雜度為 O(n log n)。

需要注意的是，隨機選擇基準點的方式可以大大減少最壞情況的發生機率，但是不能完全排除最壞情況的出現。因此，儘管平均情況下 Randomized Quicksort 的效率很高，但在某些極端情況下，仍然可能達到最壞情況的時間複雜度 O(n^2)。

我們也可以透過計算期望值來獲得時間複雜度:

定義 indicator random variable Xij 為 I{Zi is compared to Zj} for each pair (Zi, Zj)。 因為 A 當中的每對元素都最多被比較一次，可以得到：

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

接著對兩邊取期望值:

$$E[x] = \sum_{1 \le i < j \le n} E[X_{ij}]$$

其中:

$$E[X_{ij}] = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

所以:

$$\begin{aligned} E[x] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(logn) \\ &= O(nlogn) \end{aligned}$$

我們知道 T(n) $\in \Omega(nlogn)$，因此 RM_Quicksort 的 expected running time 即為 Θ(nlogn)。

Merge sort

在 Merge Sort 中，不論數組中的數據已經排序好還是亂序，它的時間複雜度都是 O(n log n)。因此 Merge Sort 沒有最佳情況或最壞情況，它的時間複雜度在任何情況下都是 O(n log n)。

這是因為 Merge Sort 是一種基於分治法的排序算法，它將原數組分成兩個子數組，分別對子數組進行排序，最後將兩個已排序的子數組合併成一個有序的數組。在這個過程中，Merge Sort 總是將兩個有序的子數組合併成一個更大的有序數組，因此無論數據怎麼樣分割，最終都能得到排序好的結果，時間複雜度均為 O(n log n)。

需要注意的是，Merge Sort 在實現時需要額外的空間來存儲中間過程的結果，因此空間複雜度比較高。此外，在遇到非常大的數組時，可能會因為需要大量的額外空間而不適合使用 Merge Sort。

Randomized quick sort & Merge sort 比較

Randomized quicksort 在大多數情況下，運行時間比 Merge sort 要優秀，尤其是在處理大量數據時。這是因為 Randomized quicksort 具有快速的分割能力，且平均時間複雜度為 O(nlogn)。此外，Randomized quicksort 對於數據的分佈狀況沒有要求，因此適用於數據分佈較為隨機的場景。

而 Merge sort 具有穩定的時間複雜度，且最壞情況下的時間複雜度為 O(nlogn)，不受數據分佈的影響。此外，Merge sort 是一種穩定排序算法，不會改變相同元素之間的相對位置，因此在需要保持元素相對位置的場景中，Merge sort 更為適合。

總之，如果需要對大量數據進行排序，而且數據分佈比較隨機，那麼 Randomized quicksort 可能是一個較好的選擇。而在需要保持元素相對位置的場景中，或者數據分佈較為特殊的場景中，Merge sort 則更為適合。

Conclusion of randomized quick sort

其主要的優點是具有隨機性，可以避免最壞情況的發生，也就是說，在大多數情況下，其運行時間是比較優秀的。當然，由於隨機性的存在，其最壞情況的時間複雜度仍然是 O(n^2)，但發生機率很小。

選用 Randomized quicksort 的時機，可以考慮以下幾個因素：

1. 數據量大：在排序大量數據的時候，QuickSort 的效率要比其他排序算法高，因為其平均時間複雜度為 O(nlogn)。
2. 數據雜亂無章：當數據序列雜亂無章時，選擇 QuickSort 可以避免因為數據規律性引起的效率問題。
3. 時間要求高：如果對算法執行時間有較高的要求，QuickSort 是一個不錯的選擇，因為它的實現比較簡單，代碼也較少，所以執行效率比較高。

總之，如果需要對大量雜亂無章的數據進行排序，並且需要高效率，那麼 Randomized quicksort 是一個非常不錯的選擇。當然，在特定的情況下，還需要根據具體情況進行選擇。