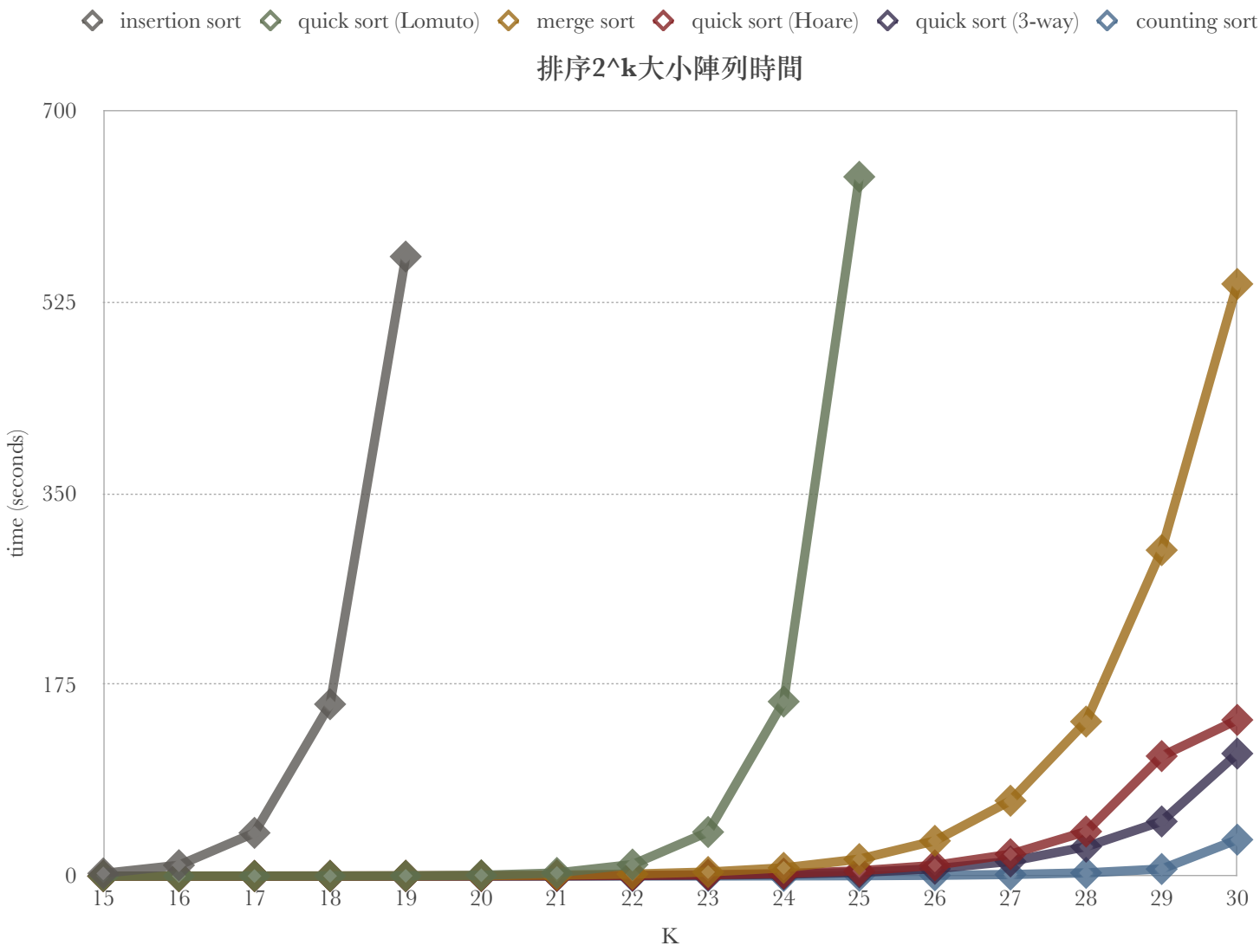
比較 insertion sort, merge sort, 三種 randomized quick sort (Lomuto Partition, Hoare Partition, 與 3-Way Partition) 與 counting sort

張芫禎
資科三109102027

折線圖：排序n=2^k大小陣列的時間



- counting sort 是最快完成排序的，而且差距高達 3 倍以上，應該是因為相較其他排序演算法少了 swap 的動作，且有明確規範亂數的範圍，才能有較好的表現。
- Hoare partition 相較 Lomuto partition 平均少了 3 次 swap，結果顯示採用 Hoare partition 的 quick sort 的確較有效率。

k	Counting sort	Merge sort	Quick sort (3-way)	Quick sort (Hoare)	Quick sort (Lomuto)	Insertion sort
15	0.0004893	0.0114777	0.0044423	0.0041131	0.0056493	0.616201
16	0.0008078	0.0207661	0.0069115	0.0081895	0.0088972	2.42304
17	0.0010725	0.0429415	0.0139581	0.0167666	0.0224537	9.91374
18	0.0021589	0.0907333	0.027403	0.0339494	0.0646025	39.5274
19	0.0046101	0.180428	0.0542386	0.0691545	0.205453	157.282
20	0.0092951	0.386289	0.106511	0.138552	0.72842	566.928
21	0.0184296	0.884071	0.212872	0.275672	2.84715	2002.05
22	0.0379099	1.83809	0.419138	0.564521	10.5455	6926.607
23	0.0773284	3.68743	0.835535	1.1515	40.3051	23478.179
24	0.151107	7.48435	1.66434	2.34432	159.677	77966.228
25	0.306165	15.5024	3.351	4.80715	639.908	253657.06
26	0.692816	32.2234	6.65108	9.782	2721.61	808510.547
27	1.41536	68.9116	13.4186	19.9989	12284.776	2524775.3
28	2.92619	141.384	26.8963	40.7602	58849.362	7724281.4
29	6.4414	298.268	49.9681	109.78	299191.6	23152172.4
30	33.3113	541.841	112.12	142.886	1614321.9	67986661.7

- 黑色方框內的數字表示因執行時間過久而沒有完整完成 10 次測試，紅色數字為預測時間。
- 預測的時間是取「成長倍數」的斜率（去掉極端值），找出每項的成長倍數後，乘以第 k-1 項的執行時間，得到第 k 項的預測時間。
- Quick sort (Lomuto) 的平均成長倍數斜率為 1.06128766124163。
- Insertion sort 的平均成長倍數斜率為 0.979711732。

merge sort 程式碼 ：from GitHub

```cpp
void merge(int array[], int const left, int const mid, int const right) {
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    auto *leftArray = new int[subArrayOne], *rightArray = new int[subArrayTwo];

    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
    int indexOfMergedArray = left;

    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        } else {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) return;

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```

insertion sort 程式碼 ：from GitHub

```
27  void insertionSort(int array[], int size) {
28    for (int step = 1; step < size; step++) {
29      int key = array[step];
30      int j = step - 1;
31
32      // Compare key with each element on the left of it until an element smaller than
33      // it is found.
34      // For descending order, change key<array[j] to key>array[j].
35      while (key < array[j] && j >= 0) {
36        array[j + 1] = array[j];
37        --j;
38      }
39      array[j + 1] = key;
40    }
41  }
```

counting sort 程式碼 ：from GitHub

```
17  void countSort(int array[], int size) {
18    // The size of count must be at least the (max+1) but
19    // we cannot assign declare it as int count(max+1) in C++ as
20    // it does not support dynamic memory allocation.
21    // So, its size is provided statically.
22    int *output = new int[size];
23    int count[1001];
24    int max = array[0];
25
26    // Find the largest element of the array
27    for (int i = 1; i < size; i++) {
28      if (array[i] > max)
29        max = array[i];
30    }
31
32    // Initialize count array with all zeros.
33    for (int i = 0; i <= max; ++i) {
34      count[i] = 0;
35    }
36
37    // Store the count of each element
38    for (int i = 0; i < size; i++) {
39      count[array[i]]++;
40    }
41
42    // Store the cummulative count of each array
43    for (int i = 1; i <= max; i++) {
44      count[i] += count[i - 1];
45    }
46
47    // Find the index of each element of the original array in count array, and
48    // place the elements in output array
49    for (int i = size - 1; i >= 0; i--) {
50      output[count[array[i]] - 1] = array[i];
51      count[array[i]]--;
52    }
53
54    // Copy the sorted elements into original array
55    for (int i = 0; i < size; i++) {
56      array[i] = output[i];
57    }
58
59    delete []output;
60  }
```

quick sort (3-way) 程式碼 ：from GeeksforGeeks

```cpp
void swap(int* a, int* b){
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function partitions a[] in three parts
   a) a[l..i] contains all elements smaller than pivot
   b) a[i+1..j-1] contains all occurrences of pivot
   c) a[j..r] contains all elements greater than pivot */
void partition(int a[], int l, int r, int& i, int& j){
    i = l - 1, j = r;
    int p = l - 1, q = r;
    int v = a[r];

    while (true) {
        // From left, find the first element greater than
        // or equal to v. This loop will definitely
        // terminate as v is last element
        while (a[++i] < v);

        // From right, find the first element smaller than or equal to v
        while (v < a[--j])
            if (j == l) break;

        // If i and j cross, then we are done
        if (i >= j) break;

        // Swap, so that smaller goes on left greater goes on right
        swap(a+i, a+j);

        // Move all same left occurrence of pivot to beginning of array and keep count using p
        if (a[i] == v) {
            p++;
            swap(a+p, a+i);
        }

        // Move all same right occurrence of pivot to end of array and keep count using q
        if (a[j] == v) {
            q--;
            swap(a+j, a+q);
        }
    }

    // Move pivot element to its correct index
    swap(a+i, a+r);

    // Move all left same occurrences from beginning to adjacent to arr[i]
    j = i - 1;
    for (int k = l; k < p; k++, j--)
        swap(a+k, a+j);

    // Move all right same occurrences from end to adjacent to arr[i]
    i = i + 1;
    for (int k = r - 1; k > q; k--, i++)
        swap(a+i, a+k);
}

void quicksort(int a[], int l, int r){
    if (r <= l) return;

    int i, j;
    // Note that i and j are passed as reference
    partition(a, l, r, i, j);

    // Recur
    quicksort(a, l, j);
    quicksort(a, i, r);
}
```

quick sort (Hoare) 程式碼 ：from GeeksforGeeks

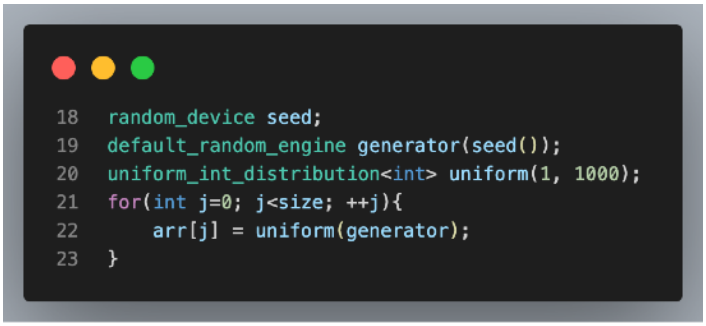
```
6  void swap(int* a, int* b){
7      int t = *a;
8      *a = *b;
9      *b = t;
10 }
11
12 /* This function takes first element as pivot, and places
13    all the elements smaller than the pivot on the left side
14    and all the elements greater than the pivot on
15    the right side. It returns the index of the last element
16    on the smaller side*/
17 int partition(int arr[], int low, int high){
18     int pivot = arr[low];
19     int i = low - 1, j = high + 1;
20
21     while (true) {
22         // Find leftmost element greater than or equal to pivot
23         do {
24             i++;
25         } while (arr[i] < pivot);
26
27         // Find rightmost element smaller than or equal to pivot
28         do {
29             j--;
30         } while (arr[j] > pivot);
31
32         // If two pointers met.
33         if (i >= j) return j;
34
35         swap(arr+i, arr+j);
36     }
37 }
38
39 /* The main function that implements QuickSort
40  arr[] --> Array to be sorted,
41   low  --> Starting index,
42   high  --> Ending index */
43 void quickSort(int arr[], int low, int high){
44     if (low < high) {
45         /* pi is partitioning index, arr[p] is now at right place */
46         int pi = partition(arr, low, high);
47
48         // Separately sort elements before
49         // partition and after partition
50         quickSort(arr, low, pi);
51         quickSort(arr, pi + 1, high);
52     }
53 }
```

quick sort (Lomuto) 程式碼 ：from GeeksforGeeks

```c
// A utility function to swap two elements
void swap(int* a, int* b){
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition(int arr[], int low, int high){
    int pivot = arr[high]; // pivot
    // Index of smaller element and indicates the right position of pivot found so far
    int i = (low- 1);

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high){
    if (low < high) {
        /* pi is partitioning index, arr[p] is now at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

心得、疑問、與遇到的困難

- 如果要得到均勻分佈的亂數就不能使用標準函式庫中提供的 rand()，而是必須到 <random> 中取得亂數種子產生器 (random_device) 和亂數產生器 (default_random_engine)，最後再宣告一個離散型均勻分布的機率函數，並設定亂數的最大值與最小值。



```
18  random_device seed;
19  default_random_engine generator(seed());
20  uniform_int_distribution<int> uniform(1, 1000);
21  for(int j=0; j<size; ++j){
22      arr[j] = uniform(generator);
23  }
```

- 在搜尋均勻分佈函數時，有資料提到不適合用 random_device 來產生大量亂數，可能用來產生 2^k 個亂數已經影響到電腦效能了，不確定實驗結果是否因此有偏差。

  Source : https://blog.gtwang.org/programming/cpp-random-number-generator-and-probability-distribution-tutorial/

  由於 `std::random_device` 在產生亂數時會消耗掉系統的熵（entropy），所以當熵耗盡時就會影響到亂數的產生，不適合用來產生大量的亂數。

  > *熵（entropy）的意思可以想像成系統上的「隨機性資訊總量」，電腦可以靠著一些有隨機性的設備來產生隨機的亂數，例如系統行程 ID、滑鼠移動軌跡、網路封包等，但是這些設備是有限的，而其所產生出來的隨機資訊也只能使用一次，若使用第二次就不是隨機的，因此系統上的隨機性資訊總量有一定的值，消耗太快而耗盡的話，就要等待這些設備再度產生。*

- 在 counting sort 的程式碼裡面，作者直接指定了 output 與 count 陣列的大小，並在註解表示是因為 C++ 不支援動態記憶體規劃，但我記得 C++ 版本的 malloc 是透過 new & delete 來完成的，因此就稍微更動了一下程式碼，count 的部分因為已經知道數字隨機分配的大小範圍，所以就直接設了 (1000+1)，編譯過後有執行成功，不知道是不是有其他原因讓原作者選擇直接給定數字。