ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

fit@hcmus

Môn học: Cấu trúc dữ liệu & Giải thuật
Lab 3: Sorting
Option: Set 2 (11 algorithms)

GVHD: Nguyễn Hải Minh

Trần Thị Thảo Nhi

Bùi Huy Thông

Lớp: 21CLC01

Nhóm thực hiện: Nhóm 12

21127297_Đỗ Phạm Thanh Huy

21127078_Nguyễn Duy Đăng Khoa

21127621_Âu Dương Khang

21127570_Trần Minh Đạt

Hồ Chí Minh, 6/2022

Mục lục

II. Information

System Information

Time of this report: 6/15/2022, 14:19:56
Machine name: LAPTOP-IQJVAU95
Machine Id: {56918907-EE1D-4536-ABB6-26F5F50908BC}
Operating System: Windows 10 Home Single Language 64-bit (10.0, Build 19044) (19041.vb_release.191206-1406)
Language: English (Regional Setting: English)
System Manufacturer: Acer
System Model: Nitro AN515-45
BIOS: V1.08 (type: UEFI)
Processor: AMD Ryzen 7 5800H with Radeon Graphics (16 CPUs), ~3.2GHz
Memory: 8192MB RAM
Available OS Memory: 7532MB RAM
Page File: 8315MB used, 10480MB available
Windows Dir: C:\Windows
DirectX Version: DirectX 12
DX Setup Parameters: Not found
User DPI Setting: 120 DPI (125 percent)
System DPI Setting: 120 DPI (125 percent)
DWM DPI Scaling: UnKnown
Miracast: Available, with HDCP
Microsoft Graphics Hybrid: Supported
DirectX Database Version: 1.3.7
DxDiag Version: 10.00.19041.0928 64bit Unicode

III. Algorithms presentation

1. Selection Sort

- *Idea:*
  - Mã giả cho Selection Sort:

```
for i=0 to length(arr):
        MinimumElement = arr[i]
        for each unsorted element:
                If arr[j] < MinimumElement
                        MinimumElement = arr[j]
        Swap(MinimumElement,arr[i])
```

- *Step-by-step descriptions:*

Example: arr[] = {64, 25, 12, 22, 11}

First Pass:

- Đối với vị trí đầu tiên trong mảng được sắp xếp, toàn bộ mảng được đi qua từ chỉ mục 0 đến 4 tuần tự. Vị trí đầu tiên nơi 64 được lưu trữ hiện tại, sau khi đi qua toàn bộ mảng, rõ ràng là 11 là giá trị thấp nhất.

64	25	12	22	11

- Do đó, thay thế 64 bằng 11. Sau một lần lặp lại 11, xảy ra là giá trị ít nhất trong mảng, có xu hướng xuất hiện ở vị trí đầu tiên của danh sách được sắp xếp.

11	25	12	22	64

Second Pass:

- Đối với vị trí thứ hai, nơi có 25, một lần nữa đi qua phần còn lại của mảng một cách tuần tự

11	**25**	12	22	64

- Sau khi đi qua, chúng tôi thấy rằng 12 là giá trị thấp thứ hai trong mảng và nó sẽ xuất hiện ở vị trí thứ hai trong mảng, do đó hoán đổi các giá trị này.

11	**12**	25	22	64

Third Pass:

- Bây giờ, đối với vị trí thứ ba, nơi 25 có mặt một lần nữa đi qua phần còn lại của mảng và tìm giá trị ít thứ ba hiện diện trong mảng.

11	12	**25**	22	64

- Trong khi đi qua, 22 xuất hiện là giá trị ít thứ ba và nó sẽ xuất hiện ở vị trí thứ ba trong mảng, do đó trao đổi 22 với phần tử có mặt ở vị trí thứ ba.

11	12	**22**	25	64

Fourth pass:

- Tương tự, đối với vị trí thứ tư đi qua phần còn lại của mảng và tìm phần tử ít thứ tư trong mảng
- Vì 25 là giá trị thấp thứ 4 do đó, nó sẽ đặt ở vị trí thứ tư.

11	12	22	**25**	64

Fifth Pass:

- Cuối cùng, giá trị lớn nhất hiện diện trong mảng sẽ tự động được đặt ở vị trí cuối cùng trong mảng
- Mảng kết quả là mảng được sắp xếp.

11	12	22	**25**	64

- *Complexity evaluation:*
  - Space complexity: O(1)
  - Time complexity: $O(n^2)$
    - Worst-case: $O(n^2)$
    - Average-case: $O(n^2)$
    - Best-case: $O(n^2)$
- *Selection Sort có biến thể:*
  - **Double Selection Sort:** Giúp tìm thấy cả giá trị tối thiểu và tối đa trong danh sách mỗi lần duyệt → Tiết kiệm được 25% chi phí tìm kiếm so với Selection Sort thông thường.
  - **Bingo Sort:** Tìm giá trị nhỏ nhất trong mảng và đem về cuối mỗi vòng lặp → Tối ưu hơn Selection Sort nếu trong mảng có nhiều giá trị trùng lặp

2. Insertion Sort

- *Idea:*
  - Để sắp xếp một mảng kích thước n theo thứ tự tăng dần:
  - Lặp từ mảng [1] đến mảng [n] trên mảng.
  - So sánh phần tử hiện tại (khóa) với phần tử trước đó của nó.
  - Nếu phần tử hiện tại nhỏ hơn phần tử trước đó của nó, hãy so sánh nó với các phần tử trước đó. Di chuyển các phần tử lớn hơn một vị trí lên để tạo không gian cho phần tử hoán đổi.
- *Step-by-step descriptions:*

Example: arr[]: {12, 11, 13, 5, 6}

12	**11**	**13**	**5**	**6**

First Pass:

- Ở đây, 12 lớn hơn 11 do đó chúng không theo thứ tự tăng dần và 12 không ở đúng vị trí của nó. Do đó, hoán đổi 11 và 12.
- Vì vậy, hiện tại 11 được lưu trữ trong một mảng con được sắp xếp.

12	**11**	**13**	**5**	**6**

- Ở đây, 12 lớn hơn 11 do đó chúng không theo thứ tự tăng dần và 12 không ở đúng vị trí của nó. Do đó, hoán đổi 11 và 12.
- Vì vậy, hiện tại 11 được lưu trữ trong một mảng con được sắp xếp.

11	**12**	**13**	**5**	**6**

Second Pass:

- Bây giờ, di chuyển đến hai yếu tố tiếp theo và so sánh chúng

11	12	13	5	6

- Ở đây, 13 lớn hơn 12, do đó cả hai yếu tố dường như theo thứ tự tăng dần, do đó, sẽ không có sự hoán đổi nào xảy ra. 12 cũng được lưu trữ trong một mảng con được sắp xếp cùng với 11

Third Pass:

- Bây giờ, hai phần tử có mặt trong mảng con được sắp xếp là 11 và 12
- Chuyển sang hai yếu tố tiếp theo là 13 và 5

11	12	13	5	6

- Cả 5 và 13 đều không có mặt đúng vị trí của chúng, vì vậy hãy trao đổi chúng.

11	12	5	13	6

- Sau khi hoán đổi, các phần tử 12 và 5 không được sắp xếp, do đó hoán đổi lại

11	5	12	13	6

- Ở đây, một lần nữa 11 và 5 không được sắp xếp, do đó trao đổi một lần nữa

5	11	12	13	6

- Ở đây, nó ở đúng vị trí của nó.

Fourth Pass:

- Bây giờ, các phần tử có mặt trong mảng con được sắp xếp là 5, 11 và 12
- Chuyển sang hai yếu tố tiếp theo 13 và 6

5	11	12	13	6

- Rõ ràng, chúng không được sắp xếp, do đó thực hiện hoán đổi giữa cả hai

5	11	12	6	13

- Bây giờ, 6 nhỏ hơn 12, do đó, hoán đổi một lần nữa

5	11	6	12	13

- Ở đây, cũng hoán đổi làm cho 11 và 6 unsorted do đó, trao đổi một lần nữa

5	6	11	12	13

- Cuối cùng, mảng được sắp xếp hoàn toàn.

5	6	11	12	13

- *Complexity evaluation:*
  - Space complexity: O(1)
  - Time complexity: $O(n^2)$
    - Worst-case: $O(n^2)$: Xảy ra trong trường hợp mảng hoàn toàn trái ngược với mảng được sort theo yêu cầu.
    - Average-case: $O(n^2)$: Xẩy ra trong trường hợp mảng ngẫu nhiên đã được sắp xếp một phần theo yêu cầu.

- Best-case: O(n): Xảy ra trong trường hợp mảng đã được sắp sẵn theo yêu cầu.

- *Thuật toán Insertion Sort có 2 biến thể là*
  - **Chèn nhị phân (Binary Insertion Sort):** Sử dụng phương pháp tìm kiếm nhị phân để tìm vị trí chèn phù hợp cho phần tử đang duyệt. Cuối cùng là tiến hành chèn → Giúp giảm thời gian chạy (running time) từ O(n) sang O(logn) và tối ưu hơn Insertion Sort.
  - **Shell Sort:** Dùng để tránh phải tráo đổi vị trí 2 phần tử quá xa nhau → Có thời gian chạy trong **Worst-case** và **Average-case** nhanh hơn Insertion Sort trong tập dữ liệu có kích cỡ trung bình O(n).
    (Xem mục 5 để biết thêm chi tiết về Shell Sort)
  - **Gnome Sort:** Thuật toán sắp xếp phân loại chèn nhưng không sử dụng vòng lặp lồng nhau → Tối ưu hơn Insertion Sort và thời gian chạy trung bình có xu hướng dần về O(n) nếu mảng được sắp xếp một phần hoặc gần như được sắp xếp.

3. Bubble Sort

- Idea:
  - Thuật toán sắp xếp bubble sort thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự (số sau bé hơn số trước với trường hợp sắp xếp tăng dần) cho đến khi dãy số được sắp xếp
  - **Bước 1**: Chạy vòng lặp từ i=0 đến i<n-1 và vòng lặp từ j=0 đến j<n-i-1
  - **Bước 2**: So sánh phần tử thứ j và phần tử thứ j+1, nếu a[j] > a[j+1] thì hoán đổi a[j] và a[j+1]
  - **Bước 3**: Tăng biến j++ rồi lặp lại bước 2 đến khi mảng được sắp xếp.
- Step-by-step descriptions:

  Example: Sắp xếp dãy số [5 1 4 2 8] này tăng dần.

  Lần lặp thứ 1:

  (**5 1** 4 2 8) –> (**1 5** 4 2 8)

  Ở đây, thuật toán sẽ so sánh hai phần tử đầu tiên, và đổi chỗ cho nhau do 5 > 1.

  (1 **5 4** 2 8) –> (1 **4 5** 2 8)

  Đổi chỗ do 5 > 4

  (1 4 **5 2** 8) –> (1 4 **2 5** 8)

  Đổi chỗ do 5 > 2

  (1 4 2 **5 8**) –> (1 4 2 **5 8**)

  Ở đây, hai phần tử đang xét đã đúng thứ tự (8 > 5), vậy ta không cần đổi chỗ.

Lần lặp thứ 2:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8)

Đổi chỗ do 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bây giờ, dãy số đã được sắp xếp, Nhưng thuật toán của chúng ta không nhận ra điều đó ngay được. Thuật toán sẽ cần thêm một lần lặp nữa để kết luận dãy đã sắp xếp khi và khi khi nó đi từ đầu tới cuối mà không có bất kỳ lần đổi chỗ nào được thực hiện.

Lần lặp thứ 3:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

- *Complexity evaluation*
  - Space complexity: O(1)
  - Time complexity: $O(n^2)$
    - Worst-case: $O(n^2)$: Xảy ra khi mảng đã được sắp xếp giảm dần.
    - Average-case: $O(n^2)$
    - Best-case: O(n): Xảy ra khi mảng đã được sort theo yêu cầu.
- *Biến thể của Bubble Sort:*
  - **Odd-even Sort:** Dùng cho các hệ thống truyên thông điệp → Đơn giản hơn Bubble Sort nhưng ít hiệu quả.
  - **Shell Sort:** Tác dụng như Bubble Sort, tuy nhiên đạt được hiệu suất tốt hơn so với Bubble Sort.
    (Xem chi tiết mục 4 để hiểu thêm về Shell Sort)
  - **Comb Sort:** Là sự cải thiện dựa trên Bubble Sort→ Mặc dù là sự cải thiện, tốt hơn Bubble Sort nhưng trong Worst-case và Avage-case vẫn là $O(n^2)$

4. Shaker Sort

- *Idea:*
  - Shaker Sort hay còn được gọi là thuật toán sắp xếp Cocktail, thực chất là một biến thể từ ý tưởng của Bubble Sort,. Thay vì chỉ đi từ bên trái tới bên phải mảng, Shaker Sort có hai giai đoạn:
    - Giai đoạn 1: thuật toán đi từ trái sang phải, đồng thời so sánh hai giá trị liền kề nhau, nếu giá trị bên trái lớn hơn giá trị bên phải thì đổi chỗ hai giá trị. Cuối cùng, giá trị lớn nhất mảng sẽ nằm ở cuối mảng.
    - Giai đoạn 2: thuật toán đi từ phải sang trái, bắt đầu tại phần tử liền kề phần tử vừa sắp xếp ở giai đoạn 1. Cũng sẽ diễn ra sự so sánh và đổi chỗ các phần tử nếu cần thiết.

- *Step-by-step descriptions:*

Example: Mảng A{3,2,6,12,10,8}

- Lần lặp thứ nhất:

{**3**,**2**,6,12,10,8}

3 > 2, swap {**2,3**}

{2,**3,6**,12,10,8}

3 < 6, giữ nguyên

{2,3,**6,12**,10,8}

6 < 12, giữ nguyên

{2,3,6,**12,10**,8}

12 > 10, swap {**10,12**}

{2,3,6,10,**12,8**}

12 > 8, swap {**8,12**}

Kết thúc

- Lần lặp thứ hai:

{2,3,6,**10,8,**12}

10 > 8, swap {**8,10}**

{2,3,**6,8,**10**,**12}

6 < 8, giữ nguyên

{2,**3,6,**8,10**,**12}

3 < 6, giữ nguyên

{**2,3**,6,8**,**10**,**12}

2 < 3, giữ nguyên

Kết thúc

- Lần lặp thứ ba: (lúc này ta thấy mảng đã được sắp xếp, nhưng thuật toán vẫn chạy lần lặp thứ ba, nếu không có việc đổi chỗ thì sẽ kết thúc thuật toán)

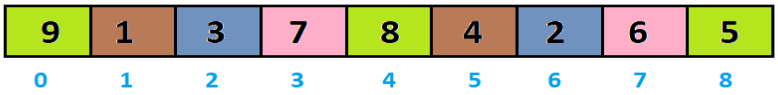
- *Complexity evaluation:*
  - Space complexity: O(1)
  - Time complexity: $O(n^2)$
    - Worst-case: $O(n^2)$: Xảy ra khi mảng đã được sắp xếp ngược yêu cầu.
    - Average-case: $O(n^2)$: Xảy ra khi mảng sắp xếp lộn xộn, không tăng dần cũng không giảm dần.
    - Best-case: O(n): Trường hợp tốt nhất của Shaker Sort xảy ra khi mảng đã được sắp xếp sẵn, khi đó chỉ xuất hiện một lần lặp và so sánh của n phần tử.
- Shaker Sort là biến thể dựa trên Bubble Sort, tuy độ phức tạp thời gian (time complexity) là như nhau, nhưng Shaker Sort nhanh hơn Bubble Sort ít nhất 2 lần.

5. Shell Sort

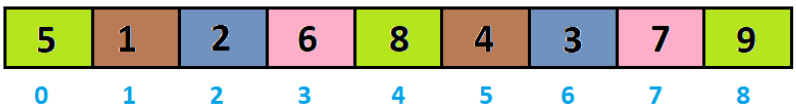
- *Idea:*
  - Shell Sort là một giải thuật sắp xếp mang lại hiệu quả cao dựa trên giải thuật sắp xếp chèn (Insertion Sort). Giải thuật này tránh các trường hợp phải tráo đổi vị trí của hai phần tử xa nhau trong giải thuật sắp xếp chọn (nếu như phần tử nhỏ hơn ở vị trí bên phải khá xa so với phần tử lớn hơn bên trái).
  - Đầu tiên, sử dụng giải thuật sắp xếp chọn trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn. Khoảng cách này được gọi là khoảng (interval).
  - interval sẽ nhận giá trị lần lượt là n/2, n/4, n/8 cho đến khi interval = 1.

- *Step-by-step descriptions:*

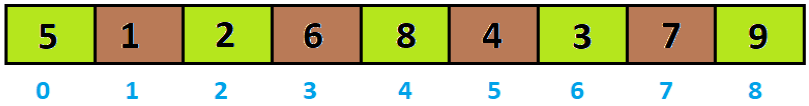
  Example: Mảng a = {9, 1, 3, 7, 8, 4, 2, 6, 5}
  - Sắp xếp những dãy con này theo cách sắp xếp chèn (Insertion Sort).



  - Sau khi sắp xếp các dãy con dãy sẽ thành.



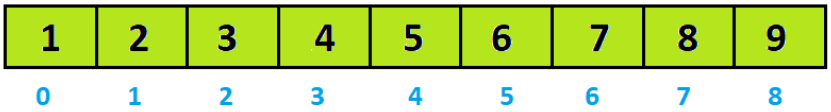
- Với interval = 9/4 = 2, ta sẽ chia dãy thành các dãy con với các số cách nhau một khoảng là interval: [5, 2, 8, 3, 9], [1, 6, 4, 7].



- Sau khi sắp xếp các dãy con dãy sẽ thành.

- Với interval = 9/8 = 1, lúc này interval = 1 ta áp dụng sắp xếp chèn với cả dãy a:

- Dãy sau khi sắp xếp là:



- *Complexity evaluation:*
  - Space complexity: O(1)
  - Time complexity: $O(n^2)$
    - Worst-case: $O(n^2)$
    - Average-case: O(nlogn)
    - Best-case: O(n)

- Shell Sort là một biến thể dựa trên Insertion Sort và Bubble Sort→ Là một sự tối ưu của Insertion Sort, có thể tráo đổi các phần tử ở xa nhau. Tuy nhiên việc xác định chính xác thời gian chạy của Shell Sort vẫn còn là vấn đề mở.
  (Có thể tham khảo thêm tại Wikipedia)
- *Shell Sort còn có biến thể*
  - **Dobosiewics Sort (Comb Sort):** Là biến thể dựa trên Shell Sort và có cách hoạt động như Bubble Sort → Tuy là biến thể nhưng vẫn hoạt động kém hiệu quả hơn Shell Sort trong Best-case, cả trong các trường hợp biến thể của Dobosiewics Sort (Shaker Pass, Brick Sort)

6. Heap Sort

- *Idea:*

  Bước 1: Ta xây dựng max heap từ dữ liệu input.

  Bước 2: Lấy phần tử rễ (root node) ra khỏi heap và đặt ở cuối mảng, thay vào vị trí đó là phần tử cuối heap.

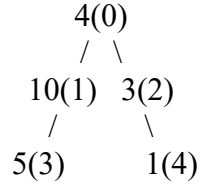
  Bước 3: Ta lặp lại bước 2 cho đến khi heap chỉ còn 1 phần tử.

- Mã giả cho Heap Sort:

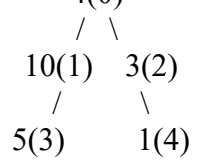
```
Max_Heapify(arr,i)
    leftchild = 2i + 1
    righchild = 2i + 2
    largest = largest(arr[parent],arr[leftchild],arr[rightchild])
    if A[i] != largest
        swap(A[i],largest)
        Max_Heapify(arr, parent)
End

BuildMaxHeap(arr)
    heap-size[arr] = length[arr]
    for i = (length[A]/2) downto 1
        do Max_Heapify(A, i)
End

HeapSort(A)
    BuildMaxHeap(A)
    for i = length[A] downto 2
        do swap A[1] with A[i]
        heap-size[A] = heap-size[A] – 1
            MaxHeapify(A, 1)
End
```

- *Step-by-step descriptions:*

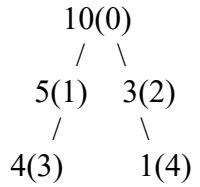
Example: Mảng A= {4, 10, 3, 5, 1}



- *Áp dụng quy trình heapify cho Index 1:*



- *Áp dụng quy trình heapify cho Index 0:*



- Quy trình heapify tự gọi mình là đệ quy để xây dựng đống theo cách từ trên xuống.

- *Complexity evaluation:*

- Space complexity: O(1)
- Time complexity: O(nlogn)
    - Time complexity of heapify is O(logn).
    - Time complexity of createAndBuildHeap() is O(n)
    - Worst-case: O(nlogn)
    - Average-case: O(nlogn)
    - Best-case: O(n): Xảy ra khi tất cả phần tử bằng nhau

- *Các biến thể của Heap Sort:*
    - **Bottom-up Heap Sort:** Dùng để giảm thiểu số phép so sánh cần thiết→Cải thiện hơn một chút về thời gian chạy so với Heap Sort thông thường. Tuy nhiên nếu so sánh với Quick Sort, thì thuật toán này dễ thực hiện và vượt trội hơn trong Avarage-case với n>=400 so với Quick Sort thông thường và với n>=16000 so. với Quick Sort dùng "Median of Three"
    - **Floyd'sHeap:** Như Heap Sort thông thường, nhưng Floyd's Heap sử dụng sàng lọc từ dưới lên(siftdown) thay vì từ trên xuống (siftup) → Tối ưu hơn và với độ phức tạp là O(n)
    - **Smooth Sort:** Biến thể và hoạt động dựa trên Heap Sort → Tối ưu hơn trong trường hợp mảng đã được sắp xếp một phần (có thời gian chạy gần tới O(n)). Tuy nhiên lại hiếm khi được dùng tới.
    - Ngoài ra còn có các biến thể khác của Heap Sort như**: Ternary Heap** (yêu cầu 3 phép so sánh) có thời gian chạy nhanh hơn một chút so với Heap Sort thông thường, **Memory-optimized Heap Sort** có sự cải thiệ hơn về mặt bộ nhớ, cải thiện hiệu suất, **Out-of-place Heap Sort** bằng việc cải thiện sắp xếp bằng cách loại bỏ Worst-case thì thời gian chạy nhanh hơn do không dùng đệ quy.
      (Ngoài ra còn một số biến thể khác có thể xem tại Wikipedia)

7. Merge Sort

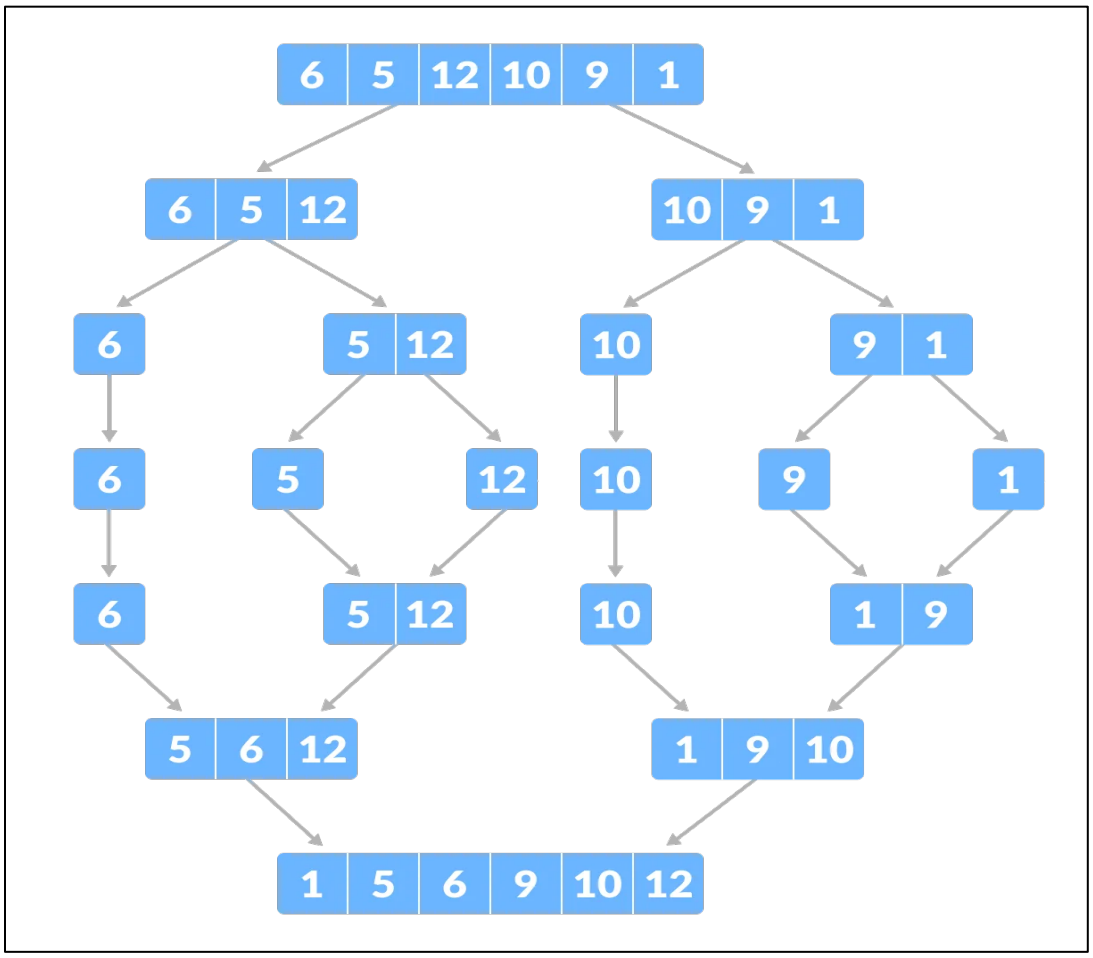
- *Idea:*
    - Hàm MergeSort() liên tục chia mảng thành hai nửa là các subarray cho đến khi thực hiện hàm MergeSort trên các subarray có kích thước bằng 1 (p=r).
    - Sau đó, hàm Merge() đi vào hoạt động và kết hợp các mảng được sắp xếp thành các mảng lớn hơn cho đến khi toàn bộ mảng được hợp nhất.
    - Mã giả Merge Sort:

```
MergeSort(A, p, r):
  if p > r
    return
  q = (p+r)/2
  mergeSort(A, p, q)
  mergeSort(A, q+1, r)
  merge(A, p, q, r)
```

- *Step-by-step descriptions:*

Example: Mảng A={6,5,12,10,9,1}

Hình ảnh minh hoạ mô tả quy trình MergeSort



- *Complexity evaluation:*
  - Space complexity: O(n)
  - Time complexity: O(nlogn) (xảy ra ở cả 3 trường hợp Worst-case, Avarage-case và Best-case)
- *Biến thể của Merge Sort:*
  - **3-way Merge Sort:** thay vì chia mảng thành 2 phần thì sẽ chia mảng thành 3 phần → Nhanh hơn Merge Sort thông thường và có độ phức tạp là $O(n\log_3 n)$
  - **Block Sort:** Dùng để kết hợp ít nhất 2 phép toán Merge và 1 phép Insertion Sort → Tối ưu hơn nhiều so với Merge Sort thông thường trong Best-case.

  (Ngoài ra có thể xem một số biến thể khác của Merge Sort tại Wikipedia)

8. Quick Sort

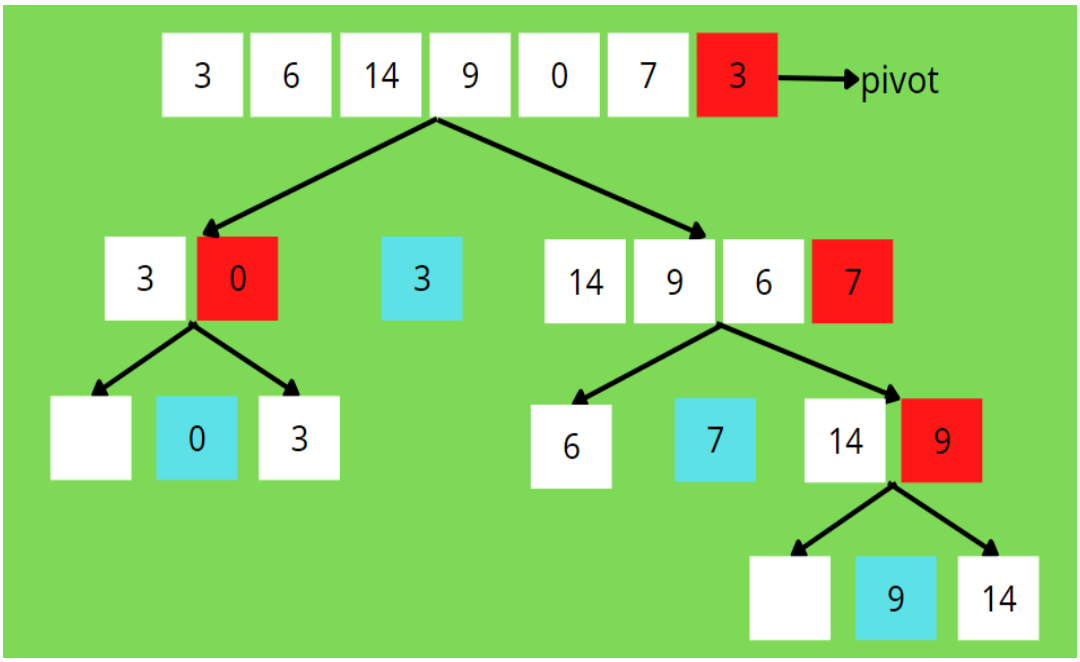
- *Idea:*
  - Chia mảng thành hai phần bằng cách so sánh từng phần tử của mảng với một phần tử được gọi là phần tử chốt. Một mảng bao gồm các phần tử nhỏ hơn hoặc bằng phần tử chốt và một mảng gồm các phần tử lớn hơn phần tử chốt. Quá trình phần chia này diễn ra cho đến khi độ dài của các mảng con đều bằng 1.

* Các bước thực hiện:

- Chọn phần tử cuối làm phần tử chốt (pivot)
- Khai báo biến i = low - 1
- Chạy vòng lặp duyệt mảng, nếu phần tử được xét nhỏ hơn pivot thì hoán đổi vị trí với phần tử thứ ++i. Duyệt đến phần tử trước pivot
- Hoán đổi pivot với phần tử giữa mảng (i + 1) để pivot vào giữa mảng
- Lúc này ta có 2 mảng con nhỏ hơn và lớn hơn pivot, sắp xếp mỗi mảng với các bước như trên

- *Step-by-step descriptions:*

Hình ảnh minh hoạ mô tả quy trình QuickSort:



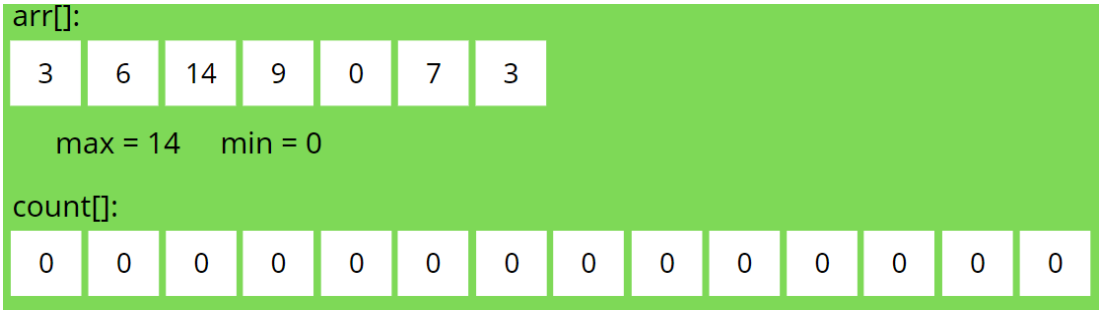
- *Complexity evaluation:*
  - Space complexity: O(logn)
  - Time complexity: $O(n^2)$

    - Worst-case: $O(n^2)$: Xảy ra khi chọn pivot là phần tử lớn nhất hoặc nhỏ nhất trong mảng
    - Average-case: O(nlogn): Xảy ra khi các phần tử trong mảng sắp xếp lộn xộn không tăng không giảm
    - Best-case: O(nlogn): Xảy ra khi chọn pivot là phần tử có giá trị trung bình hoặc gần trung bình.
- *Các biến thể của Quick Sort:*
  - Dựa trên việc chọn Pivot ở các vị trí khác nhau trên mảng như: Phần tử đầu tiền, phần tử cuối cùng, phần tử trung tâm hay phần tử ngẫu nhiên thì sẽ cho ra nhưng biến thể khác nhau cũng như là thời gian chạy khác nhau.
  - Three-way Radix Quick Sort: Là sự kết hợp của Radix Sort và Quick Sort → Có thời gian chạy trong Best-case là O(kn) (với $n<2^k$) và tối ưu hơn Quick Sort thông thường trong vài trường hợp.

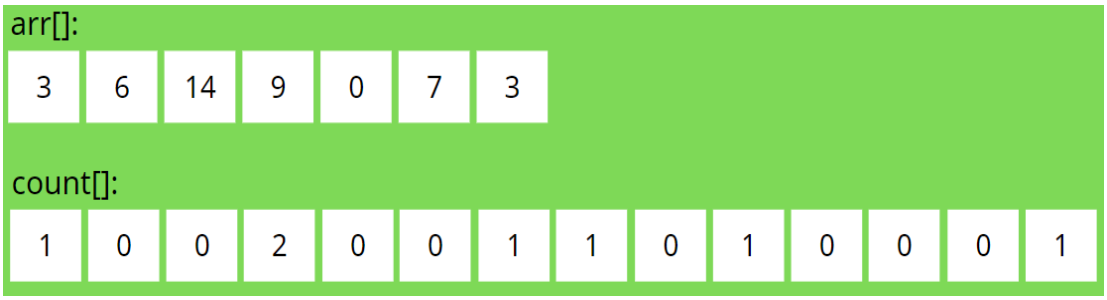
9. Counting Sort

- *Idea:*
  - Đếm số lần xuất hiện của các phần tử trong mảng đầu vào, lưu vào mảng trung gian. Sau đó thay đổi lại giá trị mảng trung gian và dựa vào đó mà sắp xếp các phần tử vào mảng mới. Cuối cùng trả về mảng mới cho mảng đầu vào.
- *Step-by-step descriptions:*

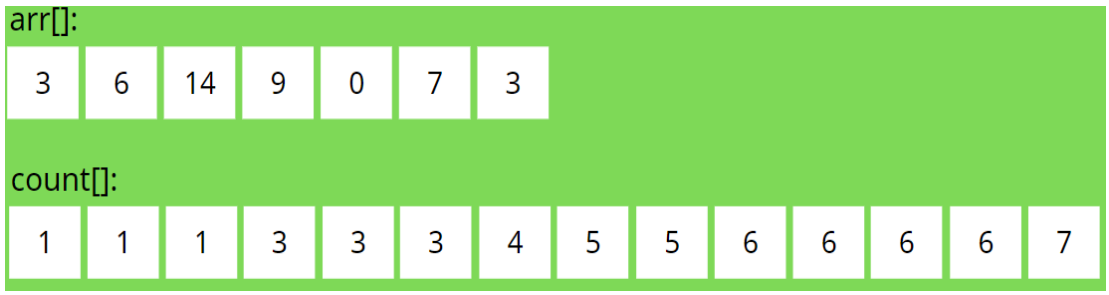
    Example: Mảng A={3, 6, 14, 9, 0, 7, 3}
  - Khởi tạo mảng mới, biến max, min chứa giá trị lớn nhất và nhỏ nhất của mảng đầu vào
  - Tạo mảng count[n] và lưu giá trị 0 cho cả mảng count[]



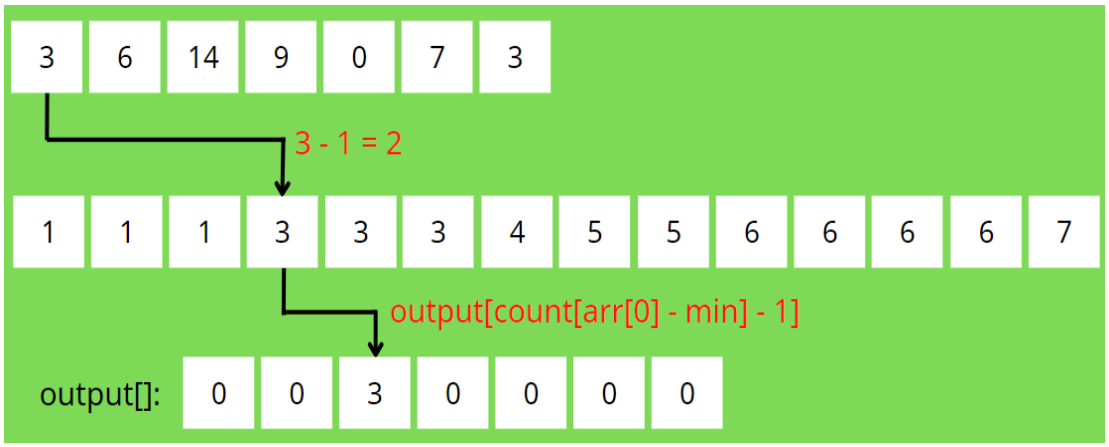
  - Lưu số lần xuất hiện của các phần tử của mảng đầu vào vào mảng count[]



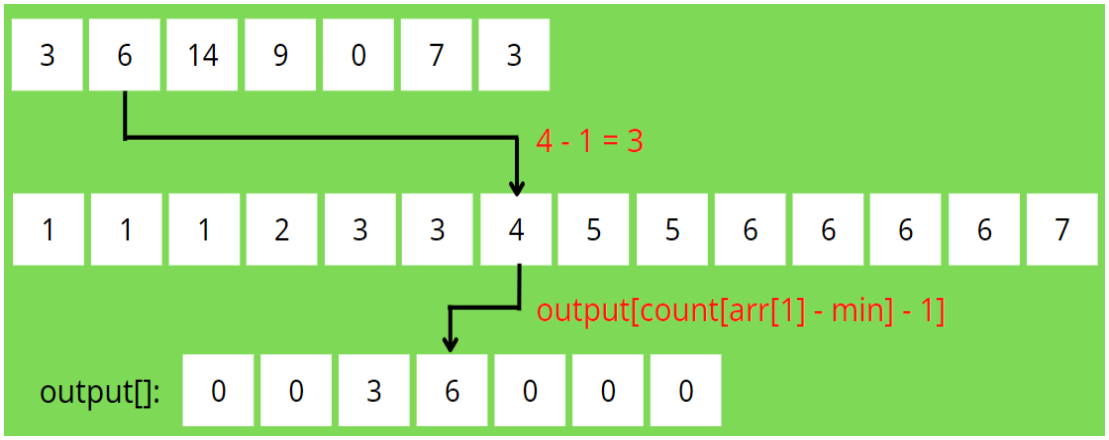
- Thay đổi giá trị mảng count[] với count[i] += count[i - 1] để count[] lúc này chứa vị trí của phần tử để lưu vào mảng mới



- Xây dựng mảng mới dựa trên chỉ số của mảng count[]
- Với phần tử 3:



- Với phần tử 6:



- Cứ như vậy cho đến hết mảng đầu vào
- Sao chép mảng output[] vào mảng đầu vào

- *Complexity evaluation:*
  - Space complexity: O(k)
  - Time complexity: O(n+k) Trong mọi trường hợp

* Với k là độ rộng của các phần tử (k = phần tử lớn nhất - phần tử nhỏ nhất)

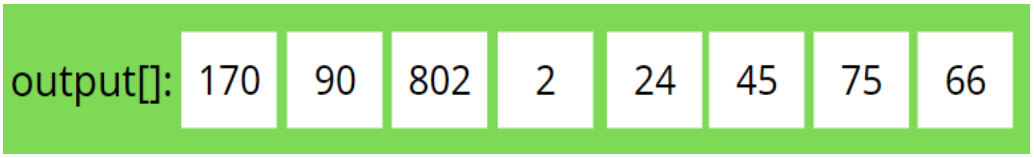
10. Radix Sort

- *Idea:*
  - Giả sử phần tử a_i trong mảng a_0, a_1... a_{n-1} là một số nguyên có tối đa m chữ số, sau đó phân loại và sắp xếp mảng lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm... cho đến hàng thứ m.

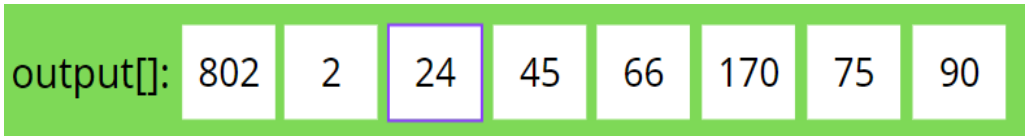
  <u>* Các bước thực hiện</u>:
  - Tìm giá trị lớn nhất của mảng làm biến trung gian cho m chữ số tối đa
  - Khởi tạo mảng trung gian là output[] với kích thước n phần tử
  - Tạo mảng count[10] chứa số lượng các chữ số tự nhiên của hàng đơn vị của các số
  - Thay đổi giá trị mảng count[] với count [i] += count[i - 1] để count[] lúc này chứa vị trí của phần tử để lưu vào output[]
  - Xây dựng mảng output[] dựa trên chỉ số mảng count[]
  - Sao chép mảng output[] vào mảng đầu vào
  - Lặp lại bước 2 với lần lượt hàng chục, hàng trăm,... đến hàng thứ m thì dừng
- *Step-by-step descriptions:*

  Example: Mảng A={170,45,75,90,802,24,2,66}

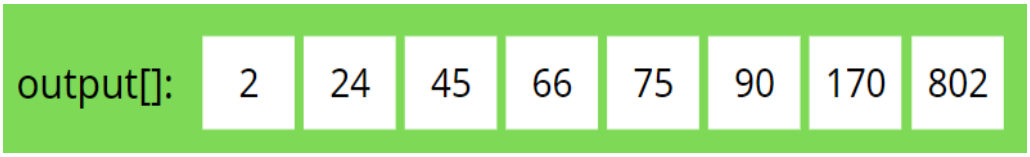
  *Hàng đơn vị:



  *Hàng chục:



  *Hàng trăm:



- *Complexity evaluation:*
  - Space complexity: O(n+k)
  - Time complexity: O(k*n)

- Worst-case: $O(n^2)$: Xảy ra khi tất cả phần tử đều có cùng số chữ số và một phần tử có số chữ số lớn đảng kể.
- Average-case: O(p(n + d)) (với p là số chữ số được xét và mỗi chữ số có tối đa d giá trị khác nhau)
- Best-case: O(a(n + b)): Khi tất cả các phần tử đều có cùng số chữ số.

(Nếu b = O(n) thì the time complexity là O(a*n))

- Biến thể của Radix Sort:
  - MSD Radix Sort (Most significant digit): Bắt đầu sắp xếp từ đầu chuỗi, giống chuỗi Quick Sort → Thời gian chạy tốt hơn và tối ưu hơn Radix Sort với Best-case là O(n)
  - LSD Radix Sort (Least significant digit): Bắt đầu sắp xếp từ cuối chuỗi như Radix Sort thông thường

11. Flash Sort

- *Idea:*

Bước 1: Phân lớp dữ liệu, ta có thể chia nhỏ thêm các bước như sau (với a[] là mảng cần được sắp xếp có n phần tử):

- Tìm giá trị nhỏ nhất của các phần tử trong mảng(minVal) và vị trí phần tử lớn nhất của các phần tử trong mảng(max).
- Khởi tạo 1 vector L có m phần tử
- Đếm số lượng phần tử các lớp theo quy luật, phần tử a[i] sẽ thuộc lớp

$$k = 1+(m - 1) * (a[i]-minVal) / (a[max] - minVal).$$

- Tính vị trí kết thúc của phân lớp thứ j theo công thức L[j] = L[j] + L[j - 1] (j tăng từ 1 đến m - 1).

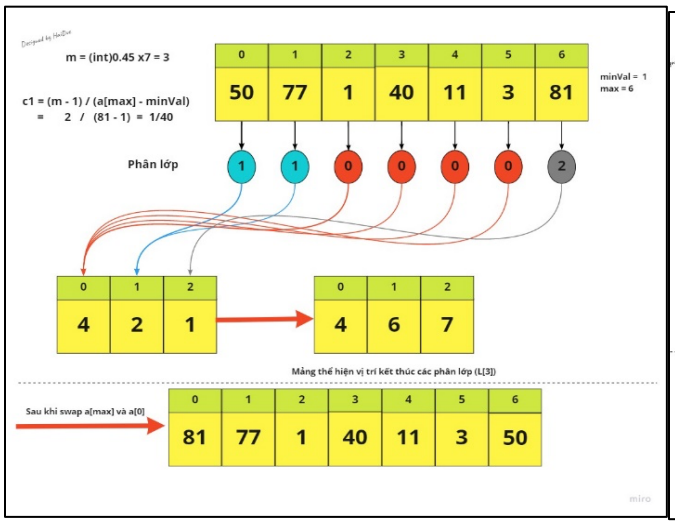
Bước 2: Hoán vị toàn cục.

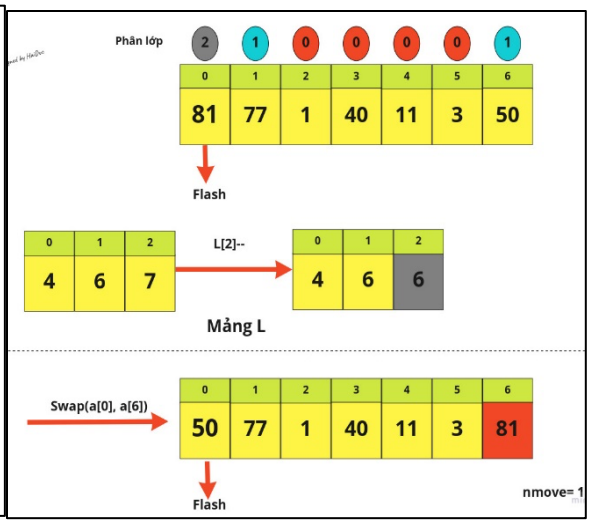
- Việc này sẽ hình thành các chu trình hoán vị: mỗi khi ta đem một phần tử ở đâu đó đến một vị trí nào đó thì ta phải nhấc phần tử hiện tại đang chiếm chỗ ra, và tiếp tục với phần tử bị nhấc ra và đưa đến chỗ khác cho đến khi quay lại vị trí ban đầu thì hoàn tất vòng lặp.

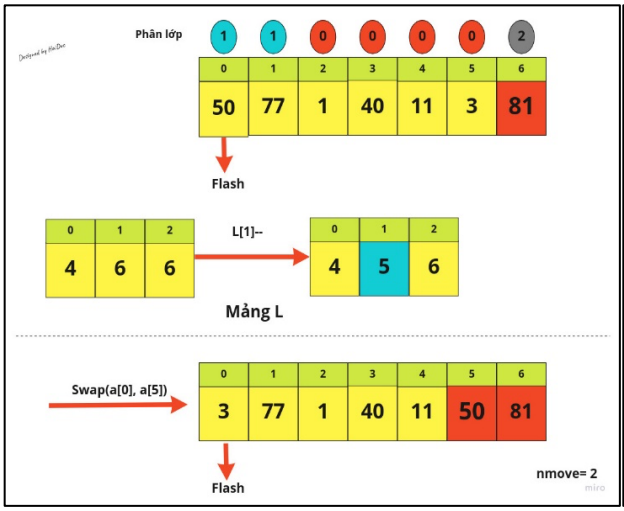
Bước 3: Dùng Insertion Sort sắp xếp lại mảng đang bị chia thành các lớp.

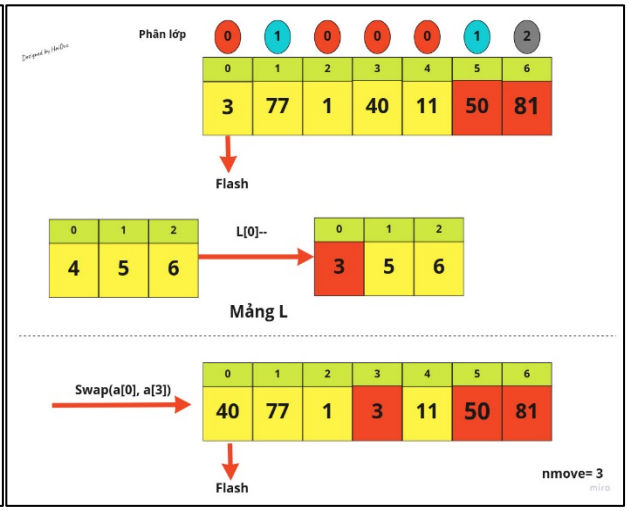
- *Step-by-step descriptions:*

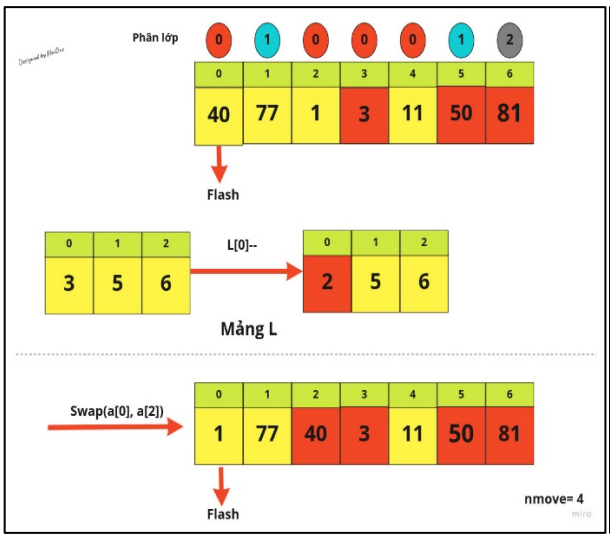
Hình ảnh minh hoạ mô tả quy trình Flash Sort

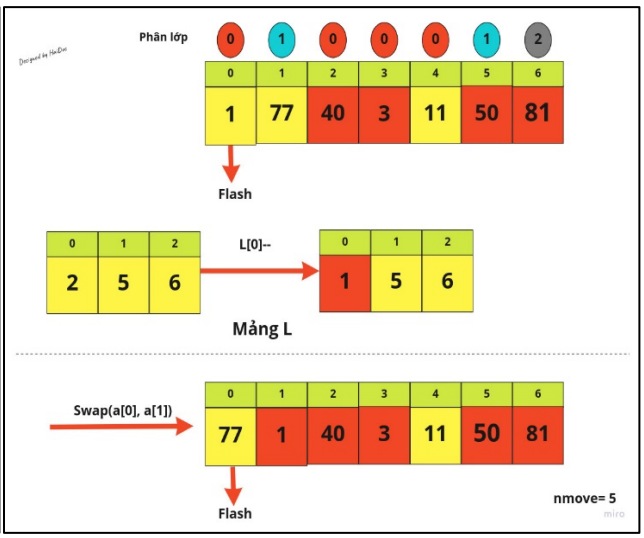

m = (int)0.45 x7 = 3
c1 = (m - 1) / (a[max] - minVal)
= 2 / (81 - 1) = 1/40
minVal = 1
max = 6
Phân lớp
Mảng thể hiện vị trí kết thúc các phân lớp (L[3])
Sau khi swap a[max] và a[0]

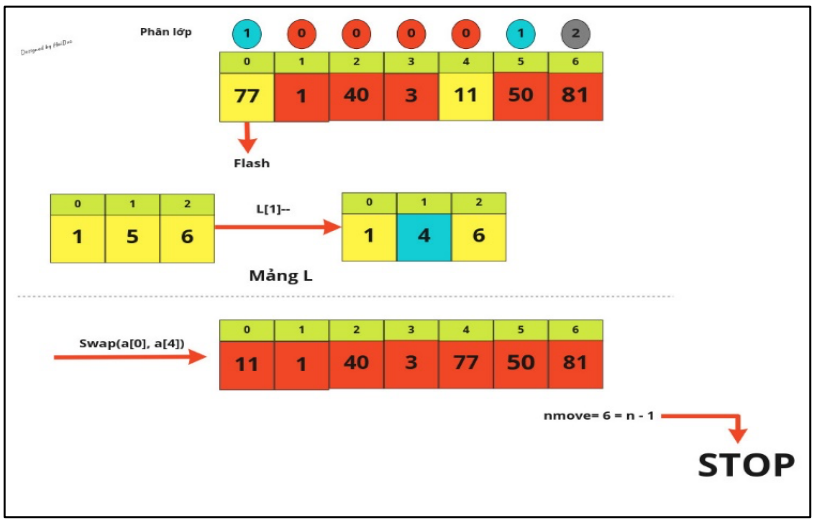

Phân lớp
Flash
L[2]--
Mảng L
Swap(a[0], a[6])
Flash
nmove= 1


Phân lớp
Flash
L[1]--
Mảng L
Swap(a[0], a[5])
Flash
nmove= 2


Phân lớp
Flash
L[0]--
Mảng L
Swap(a[0], a[3])
Flash
nmove= 3


Phân lớp
Flash
L[0]--
Mảng L
Swap(a[0], a[2])
Flash
nmove= 4


Phân lớp
Flash
L[0]--
Mảng L
Swap(a[0], a[1])
Flash
nmove= 5



- *Complexity evaluation:*
  - Space complexity: O(n)
  - Time complexity: O(n)
    - Worst-case: $O(n^2)$
    - Average-case: O(n)
    - Best-case: O(n)

IV. Experimental results and comments

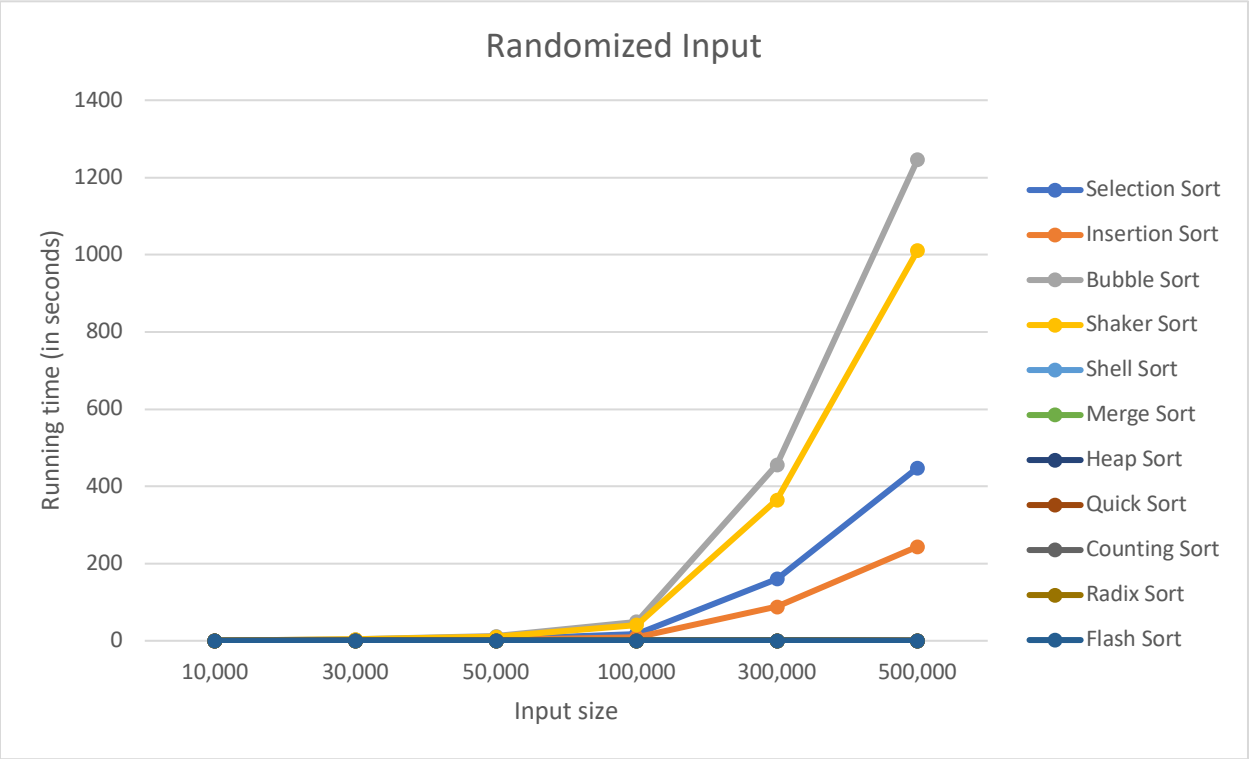
***Note:** Thời gian chạy được nhóm đo bằng cách lấy trung bình cộng của ba lần chạy. Pivot của Quick Sort được chọn theo công thức pivot = arr[$\frac{low+high}{2}$]

1. Randomized Data

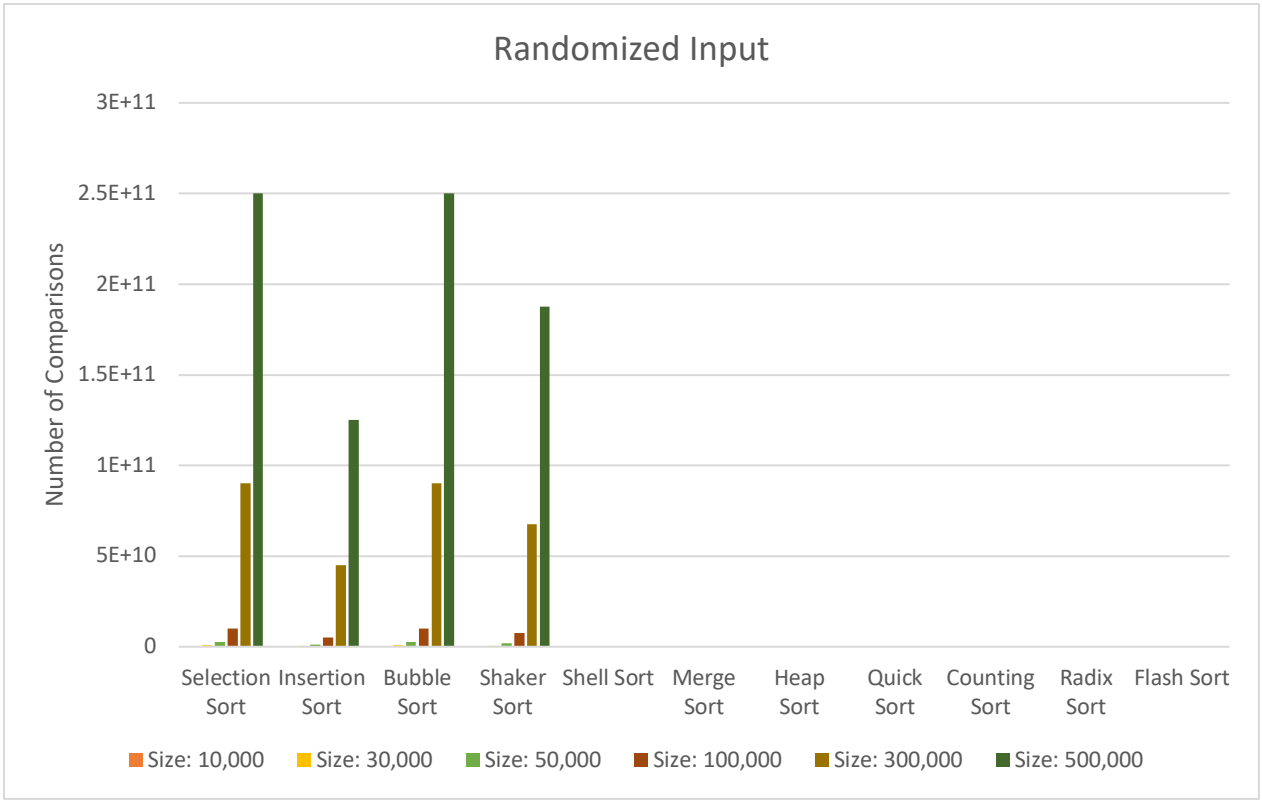
Sau đây là bảng dữ liệu nhóm thu thập được

Data order: Randomized Data												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statics	Running time (s)	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparson
Selection Sort	0.181666667	100009999	1.605333333	900029999	4.504666667	2500049999	17.96866667	10000099999	160.4333333	90000299999	447.9763333	2.5E+11
Insertion Sort	0.097	49910727	0.886666667	451741854	2.454666667	1250833930	9.789666667	4979695941	88.06166667	45007509081	243.8606667	1.25161E+11
Bubble Sort	0.407	100009999	4.052666667	900029999	12.19666667	2500049999	49.059	10000099999	456.3883333	90000299999	1245.881	2.5E+11
Shaker Sort	0.361	75069951	3.584	675809271	10.37266667	1874899996	41.33666667	7481365404	364.8863333	67658719100	1010.583333	1.87653E+11
Shell Sort	0.002	643070	0.007	2228844	0.013	4492379	0.029333333	10192328	0.101	35064797	0.177	63394919
Merge Sort	0.006333333	583679	0.018	1937642	0.031	3383188	0.062	7165641	0.188333333	23382925	0.311666667	40382208
Heap Sort	0.004333333	638212	0.014333333	2150475	0.025	3770973	0.053	8045584	0.177666667	26490795	0.308	45968848
Quick Sort	0.002	317011	0.01	849943	0.017666667	1461459	0.037333333	3012966	0.121333333	10319381	0.210333333	18052023
Counting Sort	0	59986	0.001	179988	0.001	282756	0.002	532756	0.005333333	1532755	0.009	2532761
Radix Sort	0.001	100052	0.004	360065	0.007333333	600065	0.014	1200065	0.043333333	3600065	0.072666667	6000065
Flash Sort	0	77454	0.001	231893	0.001333333	395988	0.003	708157	0.012	2192245	0.020666667	3509038

Với bảng dữ liệu trên, ta sẽ xây dựng được biểu đồ đường để biểu diễn thời gian chạy của các thuật toán:



Biểu đồ biểu diễn số phép so sánh của mỗi thuật toán:



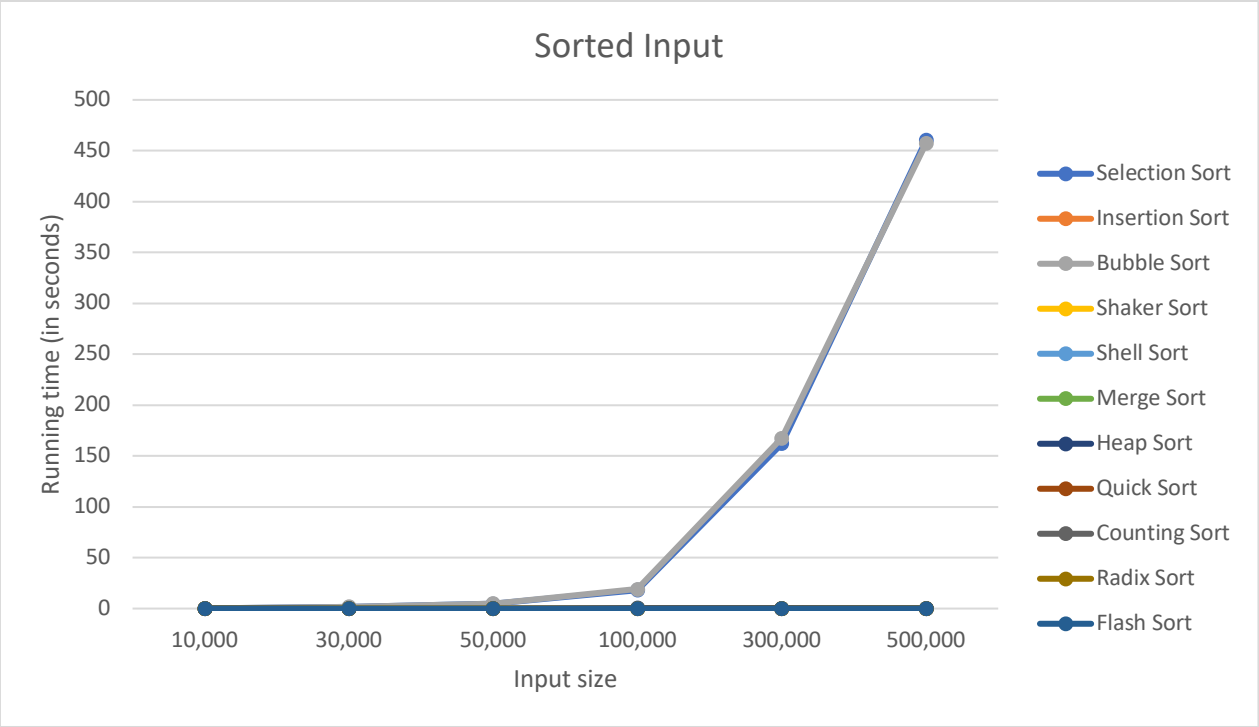
- Như ta có thể thấy, thuật toán chạy nhanh nhất đối với dữ liệu ngẫu nhiên là **Counting Sort**. Điều này càng rõ hơn khi số lượng phần tử ngày càng lớn, vì Counting Sort là thuật toán không sử dụng phép so sánh nên sẽ nhanh hơn các thuật toán so sánh như Merge Sort, Quick Sort.
- Thuật toán chạy chậm nhất là **Bubble Sort**, và sự khác biệt này càng rõ hơn khi số phần tử của dữ liệu ngày càng lớn dần. Những thuật toán có thời gian chạy tăng đáng kể khi số phần tử dữ liệu rất lớn còn bao gồm: **Shaker Sort, Selection Sort, Insertion Sort.** Điều này đúng về mặt lý thuyết vì độ phức tạp của cả 4 thuật toán trong average case đều là $O(n^2)$.

2. Sorted Data

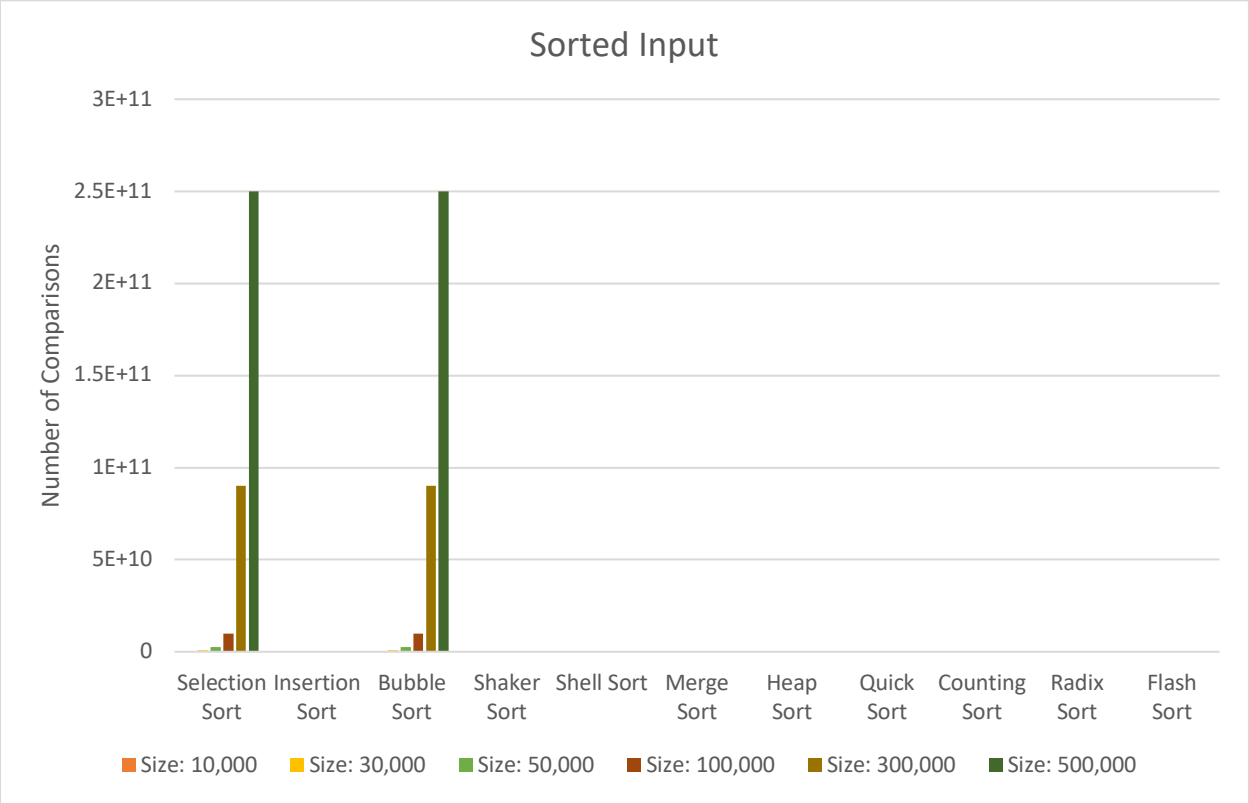
Sau đây là bảng dữ liệu nhóm thu thập được

Data order: Sorted data												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statistics	Running time (s)	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.181333333	100009999	1.630666667	900029999	4.567666667	2500049999	18.21966667	10000099999	162.1533333	90000299999	460.4843333	2.5E+11
Insertion Sort	0	29998	0	89998	0	149998	0.001	299998	0.002	899998	0.003666667	1499998
Bubble Sort	0.188333333	100009999	1.684333333	900029999	4.811	2500049999	19.01466667	10000099999	167.449	90000299999	457.34	2.5E+11
Shaker Sort	0	19999	0	59999	0	99999	0.001	199999	0.001333333	599999	0.001666667	999999
Shell Sort	0.001	360042	0.002666667	1170050	0.004333333	2100049	0.001	4500051	0.034333333	15300061	0.055333333	25500058
Merge Sort	0.005	475242	0.016666667	1559914	0.027333333	2722826	0.057	5745658	0.159666667	18645946	0.269666667	32017850
Heap Sort	0.004	670329	0.013333333	2236648	0.024	3925351	0.456	8365080	0.166666667	27413230	0.287333333	47404886
Quick Sort	0.001333333	161659	0.002	531788	0.007	923557	0.014	1947097	0.046	6319424	0.078	10888350
Counting Sort	0	50001	0.001	150001	0.001	250001	0.002	500001	0.005333333	1500001	0.008666667	2500001
Radix Sort	0.001	100052	0.004333333	360065	0.007	600065	0.014333333	1200065	0.053333333	4200078	0.088666667	7000078
Flash Sort	0	107993	0.001333333	323993	0.002	539993	0.003666667	1079993	0.011333333	3239993	0.019333333	5399993

Biểu đồ biểu diễn thời gian chạy của thuật toán:



Biểu đồ biểu diễn số phép so sánh:



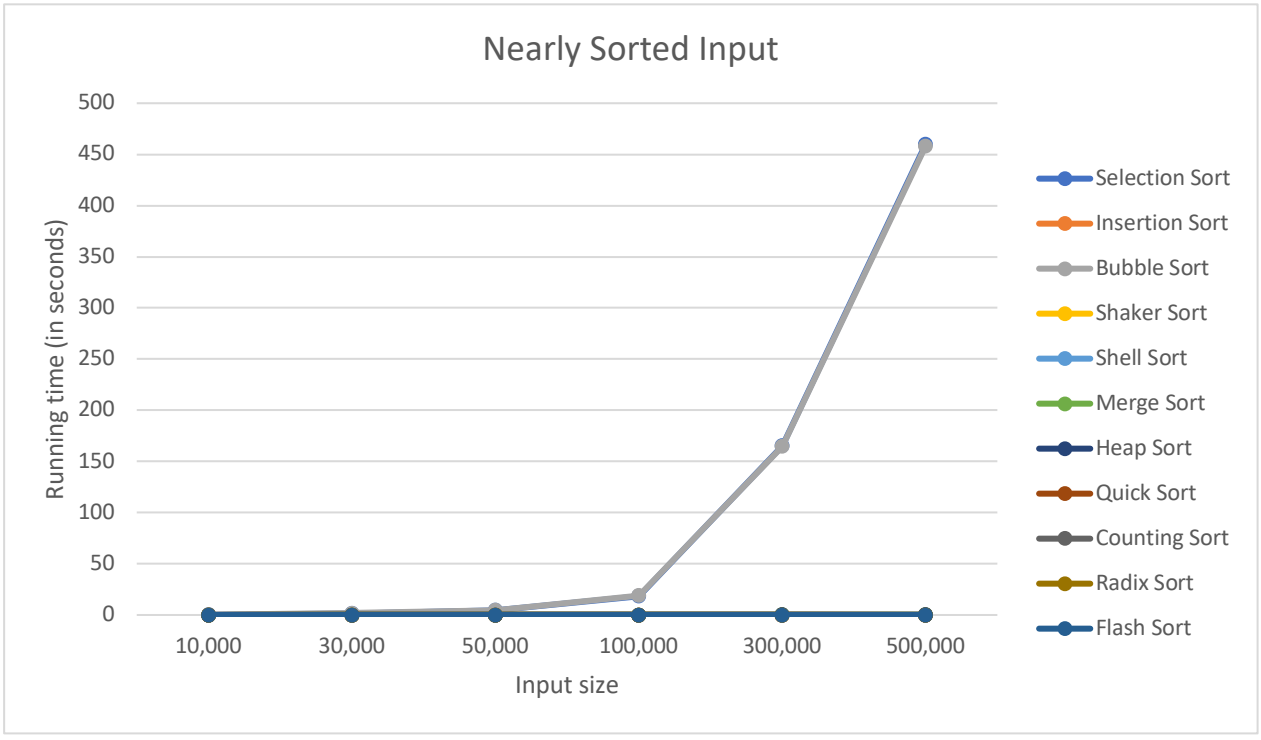
- Đối với dữ liệu được sắp xếp sẵn (trường hợp tốt nhất), thuật toán chạy chậm nhất là **Bubble Sort và Selection Sort**. Vì cả hai thuật toán đều phải so sánh mọi phần tử của dãy với mọi phần tử khác của dãy, mặc dù dãy đã được sắp xếp tăng dần.
- Những thuật toán còn lại thì được thực hiện rất nhanh chóng, với sự khác biệt nhỏ giữa các thuật toán với nhau. Nhưng khi số lượng phần tử lớn dần, ta thấy được hai thuật toán nhanh nhất là **Counting Sort và Shaker Sort.**
    - **Counting Sort** có độ phức tạp O(n) và nhanh hơn các thuật toán khác khi số lượng phần tử lớn dần.
    - Vì dữ liệu được sắp xếp sẵn, **Shaker Sort** chỉ thực hiện một vòng lặp để kết thúc (vì không xuất hiện sự hoán vị phần tử) cho nên cũng có độ phức tạp O(n).

3. Nearly Sorted Data

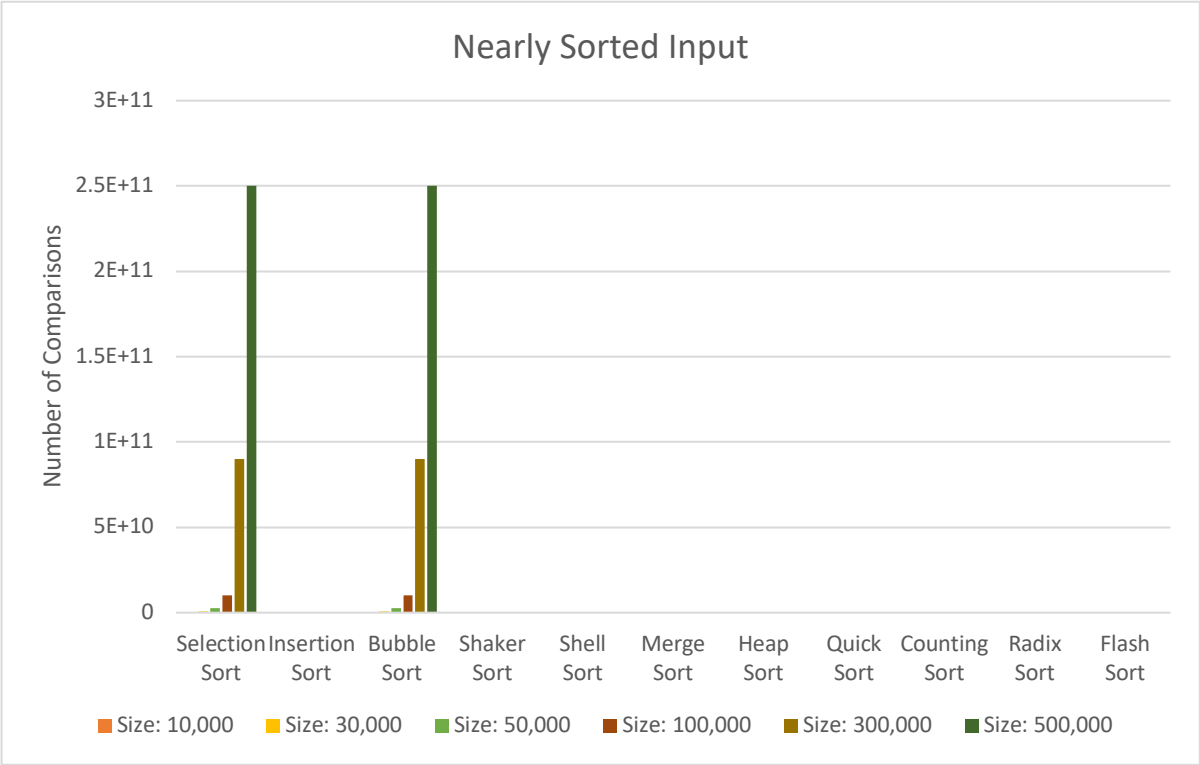
Sau đây là bảng dữ liệu nhóm thu thập được

Data order: Nearly sorted data												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statistics	Running time (s)	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.183666667	100009999	1.658666667	900029999	4.593666667	2500049999	18.322	10000099999	165.2776667	90000299999	459.9953333	2.5E+11
Insertion Sort	0	141586	0.001333333	522366	0.001	582574	0.002	838514	0.002666667	1207074	0.004	1780162
Bubble Sort	0.192333333	100009999	1.728666667	900029999	4.785333333	2500049999	19.16133333	10000099999	164.6416667	90000299999	458.1866667	2.5E+11
Shaker Sort	0.001	259831	0.003	779831	0.004	1299831	0.008	2999775	0.013	6599879	0.025	12999831
Shell Sort	0.001	401647	0.003333333	1337275	0.005333333	2282688	0.011	4695996	0.034	15419695	0.056333333	25638940
Merge Sort	0.006	511133	0.017333333	1657989	0.028	2832194	0.056	5840562	0.164666667	18758454	0.27	32109004
Heap Sort	0.004	669764	0.014	2236644	0.024666667	3925185	0.051333333	8365118	0.168	27413276	0.288	47404918
Quick Sort	0	161695	0.002	531832	0.003	923597	0.006333333	1947137	0.019666667	6319464	0.033333333	10888374
Counting Sort	0	59686	0.001	178350	0.001	281976	0.002	527851	0.005333333	1527475	0.008333333	2531132
Radix Sort	0.001	100052	0.004	360065	0.007333333	600065	0.015	1200065	0.053	4200078	0.087333333	7000078
Flash Sort	0	107965	0.002	323957	0.002	539955	0.004	1079959	0.011333333	3239963	0.018333333	5399959

Biểu đồ biểu diễn thời gian chạy của các thuật toán:



Biểu đồ biểu diễn số phép so sánh:



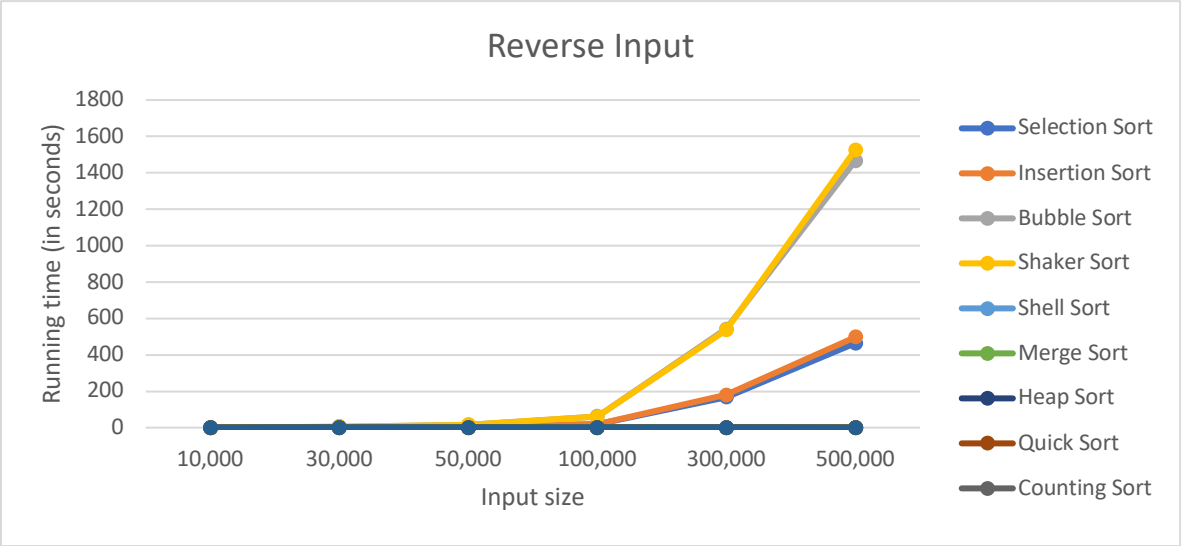
- Đối với kiểu dữ liệu gần được sắp xếp, thuật toán chậm và sử dụng nhiều phép so sánh nhất là **Bubble Sort và Selection Sort**. Vì cả hai thuật toán đều phải so sánh mọi phần tử của dãy với mọi phần tử khác của dãy thì mới kết thúc.
- Các thuật toán còn lại vẫn không có sự khác biệt to lớn về thời gian chạy và phép so sánh sử dụng. Nhưng hai thuật toán nhanh nhất là **Counting Sort và Insertion Sort**.
    - Counting Sort có độ phức tạp O(n) và có thời gian càng nhanh hơn các thuật toán khác khi số lượng phần tử ngày càng lớn.
    - Đối với Insertion Sort, là do thuật toán này thích ứng với tỷ lệ giữa dãy số đã sắp xếp và dãy chưa sắp xếp, nên khi dữ liệu input gần được sắp xếp, Insertion Sort có thời gian chạy rất ngắn.

4. Reverse Data

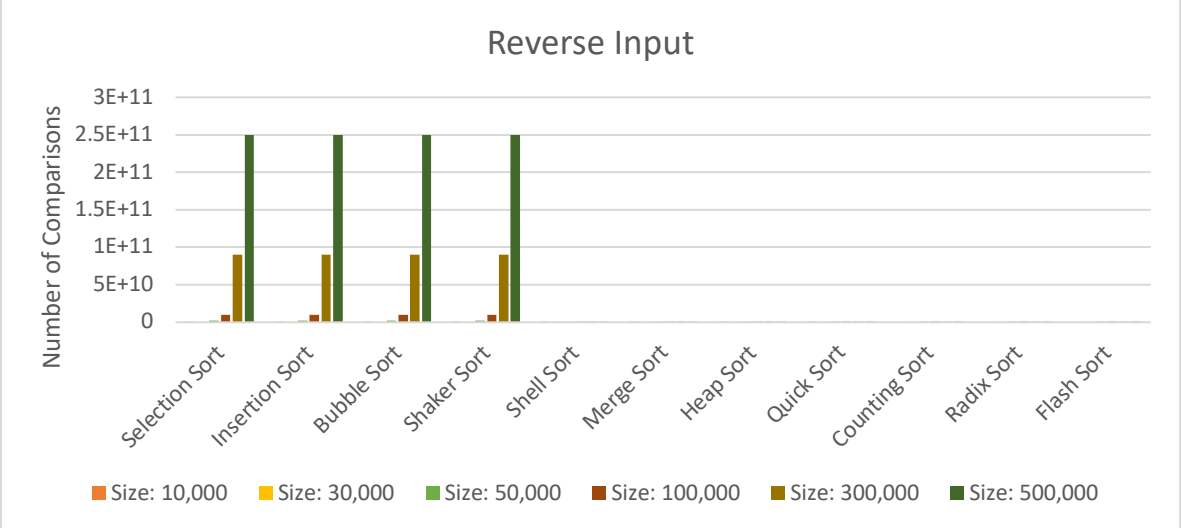
Sau đây là bảng dữ liệu nhóm thu thập được

Data order: Reverse data												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statistics	Running time (s)	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.183333333	100009999	1.654333333	900029999	4.607	2500049999	18.31133333	10000099999	167.7656667	90000299999	464.4023333	2.5E+11
Insertion Sort	0.197666667	100009999	1.801666667	900029999	5.031	2500049999	19.799	10000099999	180.264	90000299999	497.5276667	2.5E+11
Bubble Sort	0.611	100009999	5.511666667	900029999	15.27766667	2500049999	62.08025	10000099999	542.8506667	90000299999	1467.554667	2.5E+11
Shaker Sort	0.606666667	100000000	5.51	900000000	15.672	2500000000	59.812	10000000000	536.8136667	90000000000	1527.174667	2.5E+11
Shell Sort	0.001	475175	0.003	1554051	0.006	2844628	0.014	6089190	0.045	20001852	0.078	33857581
Merge Sort	0.005666667	476441	0.017333333	1573465	0.028	2733945	0.052666667	5767897	0.160333333	18708313	0.270666667	32336409
Heap Sort	0.004	606771	0.012666667	2063324	0.023	3612724	0.048	7718943	0.154666667	25569379	0.271666667	44483348
Quick Sort	0	171656	0.002	561785	0.003333333	973554	0.006666667	2047094	0.0214	6619421	0.036	11388347
Counting Sort	0	60000	0.001	180000	0.001	300000	0.002	600000	0.005333333	1800000	0.009	3000000
Radix Sort	0.001	100052	0.004	360065	0.007	600065	0.014	1200065	0.052333333	4200078	0.088666667	7000078
Flash Sort	0	90502	0.001	271502	0.001333333	452502	0.003	905002	0.010666667	2715002	0.017	4525002

Biểu đồ biểu diễn thời gian chạy của các thuật toán:



Biểu đồ biểu diễn số phép so sánh các thuật toán:



- Đối với dữ liệu bị sắp xếp ngược, **Shaker Sort và Bubble Sort** là hai thuật toán chạy chậm nhất, và cũng thuộc 4 loại thuật toán có số phép so sánh nhiều nhất. Điều này đúng vì:
    - Bubble Sort sẽ phải so sánh mọi phần tử của dãy với mọi phần tử khác, đồng thời còn phải thực hiện n lần hoán vị phần tử.
    - Và vì Shaker Sort về cơ bản là một biến thể của Bubble Sort, cho nên nó cũng là thuật toán chậm nhất trong trường hợp này.

- Thuật toán chạy nhanh nhất và thực hiện ít phép so sánh nhất là **Counting Sort**, điều này càng rõ hơn khi số lượng phần tử phải so sánh lớn dần.

5. Nhận xét chung

	Độ phức tạp			**Sử dụng bộ nhớ thêm ?**	**Thuật toán ổn định?**	**Thuật toán có so sánh?**	**Ghi chú**
	Best case	**Average case**	**Worst case**				
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Không	Có	Thời gian chạy chậm, đôi khi được ưa chuộng vì sử dụng ít bộ nhớ và số lượng phép hoán vị luôn < n
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Có	Có	Có tốc độ vượt trội khi số lượng dữ liệu thấp, hoặc khi kiểu dữ liệu gần như được sắp xếp
Bubble Sort	O(n) (nếu thuật toán được tối ưu hóa)	$O(n^2)$	$O(n^2)$	O(1)	Có	Có	Thuật toán sắp xếp cơ bản nhất, thời gian chạy rất lâu, có best case bằng O(n) nếu được tối ưu
Shaker Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Có	Có	Là biến thể dựa trên Bubble Sort, thời gian chạy vẫn chậm, nhưng nhanh gấp đôi Bubble Sort
Shell Sort	O(n)	O(nlogn)	$O(n^2)$	O(1)	Không	Có	Là biến thể dựa trên Insertion Sort, thời gian chạy khá nhanh nhưng dựa trên độ lớn các mảng chia nhỏ
Heap Sort	O(n)	O(nlogn)	O(nlogn)	O(1)	Không	Có	Là thuật toán có thời gian khá nhanh, kể cả khi số lượng phần tử lớn dần. Tuy nhiên,

							thuật toán không ổn định
Merge Sort	O(nlog)	O(nlogn)	O(nlog)	O(n)	Có	Có	Là thuật toán có thời gian khá nhanh, kể cả khi số lượng phần tử lớn dần. Thuật toán cần sử dụng bộ nhớ thêm
Quick Sort	O(nlog)	O(nlogn)	$O(n^2)$	O(logn)	Không	Có	Có thời gian chạy khá nhanh, hiếm khi chạy trong $O(n^2)$. Chưa tối ưu hoá bộ nhớ
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Có	Không	Có thời gian chạy rất nhanh, vấn đề lớn gặp phải là khó điều khiển được mảng dữ liệu tạo thêm.
Radix Sort	O(a(n+b))	O(p(n+d))	O(n*k)	O(n+k)	Có	Không	Giải quyết vấn đề bộ nhớ mảng thêm của Counting Sort. Tuy nhiên, độ phức tạp của radix sort bị ảnh hưởng nhiều bởi kiểu dữ liệu (int,float,string,…)
Flash Sort	O(n)	O(n)	$O(n^2)$	O(1)	Không	Có	Có thời gian chạy trong Average-case là tốt nhất trong các thuật toán sắp xếp

- p là số chữ số được xét và mỗi chữ số có tối đa d giá trị khác nhau

Kết luận:

- Qua kết quả thực nghiệm, thuật toán Counting Sort là nhanh nhất nói chung, dựa trên 4 kiểu sắp xếp dữ liệu. Khi số lượng phần tử lớn dần (tiệm cận), sự khác biệt này ngày càng rõ và Counting Sort sẽ vượt trội hơn các thuật toán phổ biến như Quick Sort hay Merge Sort. Tuy nhiên, vấn đề lớn của thuật toán này nằm ở độ lớn của mảng tạo thêm (vì độ lớn của mảng = giá trị phần tử max của dãy dữ liệu).
- Thuật toán Bubble Sort là chậm nhất trong các thuật toán, và thời gian chạy tăng theo cấp số nhân khi số lượng phần tử lớn dần (tiệm cận).

V. Project organization and Programming notes:

- Source codes được chia làm thành 3 file tuỳ vào từng chức năng khác nhau như:
  - 12.cpp: Chứa dữ liệu nội dụng được yêu cầu về các Command, Input và Ouput
  - GenerateData.h: Chứa các hàm dùng để xử lí việc tạo ra dữ liệu đầu vào được ngẫu nhiên theo từng chức năng khác nhau tương ứng từng hàm có trong file.
  - Sort.h: Chứa các hàm khác nhau tương ứng các loại hình Sort về việc sắp xếp dữ liệu đã cho theo thứ tự tăng dần.
- Ngoài ra, còn sử dụng các thư viện khác nhau trong việc xây dựng như:

```
#include <iostream>
#include <string>
#include <iomanip>
#include <fstream>
#include "Sorting.h"
```

```
#include <iostream>
#include <time.h>
#include <stdlib.h>
```

- Trong đó có các viện đặc biệt như thư viện <time.h> dùng để đo thời gian chạy của các thuật toán sắp xếp và thư viện <iomanip> giúp lấy hàm setprecious để xuất chi tiết phần thập phân trong việc xem xét Running Time của các thuật toán sắp xếp.

VI. List of References

- Thuật toán Selection Sort:
  - Ví dụ và Space complexity: https://www.geeksforgeeks.org/selection-sort/
  - Tìm biến thể:
    - https://en.wikipedia.org/wiki/Selection_sort#Variants/
    - https://xlinux.nist.gov/dads/HTML/bingosort.html
- Thuật toán Insertion Sort:
  - Tìm biến thể: https://duongdinh24.com/thuat-toan-sap-xep-chen/
  - Ý tưởng, ví dụ và code thuật toán: https://www.geeksforgeeks.org/insertion-sort/
- Thuật toán Bubble Sort:
  - Ví dụ và độ phức tạp: https://nguyenvanhieu.vn/thuat-toan-sap-xep-bubble-sort/
  - Thực hiện code: https://www.geeksforgeeks.org/bubble-sort/
  - Tìm biến thể:
  - https://en.wikipedia.org/wiki/Odd–even_sort
  - https://en.wikipedia.org/wiki/Comb_sort
- Thuật toán Shaker Sort/Cocktail Sort:
  - Ý tưởng và code thuật toán https://www.geeksforgeeks.org/cocktail-sort/
  - Độ phức tạp: https://www.javatpoint.com/cocktail-sort
- Thuật toán Shell Sort:
  - Ý tưởng và ví dụ: https://codelearn.io/learning/cau-truc-du-lieu-va-giai-thuat/856660
  - Tìm biến thể:
  - https://en.wikipedia.org/wiki/Shellsort#Computational_complexity
  - http://thomas.baudel.name/Visualisation/VisuTri/dobosort.html

- Thuật toán Heap Sort:
  - https://www.geeksforgeeks.org/heap-sort/
  - Tìm biến thể: https://en.wikipedia.org/wiki/Heapsort#Algorithm
    - Bottom-up Heap Sort: https://www.sciencedirect.com/science/article/pii/030439759390364Y
- Thuật toán Quick Sort:
  - Ý tưởng và thực hiện code: https://www.geeksforgeeks.org/quick-sort/
- Thuật toán Merge Sort:
  - Ví dụ và code thuật toán: https://www.programiz.com/dsa/merge-sort/
  - Tìm biến thể: https://www.geeksforgeeks.org/3-way-merge-sort/
- Thuật toán Radix Sort:
  - Thực hiện code: https://www.geeksforgeeks.org/radix-sort/
  - Time complexity: https://www.simplilearn.com/tutorials/data-structure-tutorial/radix-sort
  - Tìm biến thể: https://www.geeksforgeeks.org/msd-most-significant-digit-radix-sort/
- Thuật toán Counting Sort:
  - Thực hiện code: https://nguyenvanhieu.vn/counting-sort/
- Thuật toán Flash Sort:
  - https://en.wikipedia.org/wiki/Flashsort
  - Thự hiện code ,ý tưởng và ví dụhttps://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh/
- Các tài liệu tham khảo khác:
  - Thực hiện code: https://github.com/HaiDuc0147/sortingAlgorithm/tree/main/reportSort/