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

NHÓM 4

BÁO CÁO LAB 03

Thực hành DSA

MỤC LỤC

I. INFOMATION

NHÓM 04

STT	MSSV	Họ và tên	Email	SĐT
1	20120440	Lê Trần Bảo Châu	20120349@student.hcmus.edu.vn	
2	20120354	Đặng Huỳnh Cửu Quân	20120354@student.hcmus.edu.vn	0358766475
3	20120349	Ngô Hữu Phúc	20120440@student.hcmus.edu.vn	0834678755
4	19120557	Trần Tuấn Kiệt	19120557@student.hcmus.edu.vn	0706017675

II.INTRODUCTION

Bản báo có sẽ trình bày về:

- Chi tiết 11 giải thuật sắp sếp bao gồm: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort và Flash Sort.
- Thực nghiệm quá trình sắp xếp của các thuật toán trên với số lượng phần tử tăng dần

Qua đó, bản báo cáo sẽ cho chúng ta thấy một cái nhìn tổng quan về 11 thuật toán được nêu ở trên từ ý tưởng, các bước thực hiện, độ phúc tập không-thời gian. Đồng thời sẽ thể hiện tính áp dụng của chúng trong thực tế thông qua các biểu đồ thực nghiệm

Bản báo cáo được làm bởi 4 sinh viên Đặng Huỳnh Cửu Quân, Lê Trần Bảo Châu, Ngô Hữu Phúc, Trần Tuấn Kiệt. Như đã nói ở trên, trong khuôn khổ bài Lab 03, nhóm dự định sẽ có 11 thuật toán sẽ được thực hiện. Tuy nhiên vì một vài lí do của cá nhân trong tổ, số thuật toán thực hiện chỉ còn 8 thuật toán (trừ Quick Sort, Radix Sort và Merge Sort).

Để tiện cho việc chạy các thuật toán trên để so sánh hoặc sắp xếp 1 đối tượng nào đó, nhóm có tổ chức các source code trong 1 folder. Đồng thời có một file .exe để chạy các thuật toán (so sánh, sắp xếp) trên command line.

Bài báo cáo gồm 4 phần:

- Mô tả các thuật toán sắp xếp
- Thực nghiệm và các biểu đồ thực nghiệm
- Chia sẻ về cấu trúc file
- Ghi chú

III. ALGORITHM PRESENTATION

1. Selection sort

1.1 Sơ lược

- Ý tưởng: Selection Sort là một thuật toán sắp xếp được dựa trên việc so sánh tại chỗ bằng cách đi tìm phần từ có giá trị nhỏ nhất trong n phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu tiên của mảng con đã được sắp xếp. Sau đó bắt đầu từ vị trí thứ 2 với mảng con chưa được sắp xếp chỉ còn n-1 phần tử. Lặp lại quá trình này cho đến khi mảng hiện hành chỉ còn 1 phần tử.

1.2 Thuật toán

- Cho dãy gồm **n** phần tử
- Các bước thực hiện:
  - B1 : gán i=0
  - B2 : tìm phần tử array[min] nhỏ nhất trong dãy hiện hành từ array[i] đến array[n-1]
  - B3 : hoán vị a[min] và a[i]
  - B4 : nếu i<n-1, gán i=i+1. Lặp lại bước 2
  - Ngược lại, dừng thuật toán khi n-1 phần tử đã nằm đúng vị trí
- Minh họa:
  - Cho dãy array = {31, 23, 59, 47, 7}
  - Tìm phần tử nhỏ nhất trong array[0->4] và đổi chỗ nhó với phần tử đầu tiên của mảng hiện tại :
    7 31 23 59 47
  - Tìm phần tử nhỏ nhất trong array[1->4] và đổi chỗ nhó với phần tử đầu tiên của mảng hiện tại :
    7 23 31 59 47
  - Tìm phần tử nhỏ nhất trong array[2->4] và đổi chỗ nhó với phần tử đầu tiên của mảng hiện tại :
    7 23 31 59 47
  - Tìm phần tử nhỏ nhất trong array[3->4] và đổi chỗ nhó với phần tử đầu tiên của mảng hiện tại :
    7 23 31 47 59

1.3 Đánh giá

- Best-case: $O(n^2)$.
- Average case: $O(n^2)$.
- Worst case: $O(n^2)$.
- Space Complexity: O(1)..

2. Insertion sort

2.1 Sơ lược

- Ý tưởng: Thuật toán Insertion Sort thực thiện sắp xếp dãy số bằng cách duyệt từng phần tử và chèn phần tử được duyệt vào đúng vị trí trong mảng con đã sắp xếp (dãy từ vị trí đầu đến vị trí trước phần tử đang được duyệt) sao cho dãy số trong mảng con đã sắp xếp đó vẫn đảm bảo được tính chất của một dãy tăng dần (hoặc giảm dần).

2.2 Thuật toán

- Cho một mảng có **n** phần tử.
- Các bước thực hiện:
  - B1 : Duyệt từ phần tử thứ 2, lặp từ array[1] đến array[n-1] của mảng
  - B2 : So sánh phần tử đang được duyệt với phần tử trước đó
  - B3 : Nếu phần tử đang được duyệt nhỏ hơn phần tử trước đó, so sánh nó với các phần tử trước đó và di chuyển các phần tử lớn hơn lên một đơn vị để tạo khoảng trống cho phần tử được hoán đổi
- Minh Họa:
  - Cho dãy array = {47, 31, 59, 11, 23}
  - Gán biến k với giá trị lặp bắt đầu từ phần tử thứ 2 (array[1]) đến phần tử cuối cùng của mảng (array[4])
  - Khi k = 1, vì 31 nhỏ hơn 47 nên di chuyển 47 và chèn 31 vào trước 47
    31 47 59 11 23
  - Khi k = 2, 59 sẽ giữ vị trí như cũ vì 59 lớn hơn tất cả các phần tử trong array[0->1]
  - 31 47 59 11 23
  - Khi k = 3, 11 sẽ di chuyển về vị trí đều vì nó bé hơn tất cả các phần tử trong array[0->2] và tất cả các phần tử trong array[0->2] sẽ di chuyển vị trí lên thêm 1 đơn vị so với vị trí hiện tại của chúng
    11 31 47 59 23
  - Khi k = 4, 23 sẽ di chuyển đến vị trị sau 11 và các phần tử 31 đến 59 sẽ di chuyển lên thêm 1 đơn vị so với vị trí hiện tại của chúng
    11 23 31 47 59

2.3 Đánh giá:

- Best case: O(n).
- Average case: $O(n^2)$.
- Worst case: $O(n^2)$.
- Không gian bộ nhớ sử dụng: O(1).

3. Bubble sort

3.1 Sơ lược:

- Ý tưởng: Bubble Sort là thuật toán thực hiện sắp xếp dãy số bằng cách lặp lại việc so sánh hai phần tử kề nhau, nếu chúng chưa đúng thứ tự theo độ tăng dần (hoặc giảm dần) thì đổi chỗ cho đến khi dãy số được sắp xếp hoàn chỉnh theo thứ tự tăng dần (hoặc giảm dần)

3.2 Thuật toán

- Sắp xếp từ trên xuống
    - Cho dãy cần sắp xếp có n phần tử. Tiến hành so sánh hai phần tử đầu từ trên xuống, nếu phần tử đứng trước lớn hơn phần tử đứng sau thì đổi chỗ chúng cho nhau. Lặp lại điều này với cặp phần tử tiếp theo và tiếp tục cho đến cuối tập hợp dữ liệu, tức so sánh phần tử n-1 và phần tử thứ n. Sau bước này, phần tử cuối cùng là phần tử lớn nhất của dãy.
    - Cho dãy cần sắp xếp có n phần tử. Tiến hành so sánh hai phần tử đầu từ trên xuống, nếu phần tử đứng trước lớn hơn phần tử đứng sau thì đổi chỗ chúng cho nhau. Lặp lại điều này với cặp phần tử tiếp theo và tiếp tục cho đến cuối tập hợp dữ liệu, tức so sánh phần tử n-1 và phần tử thứ n. Sau bước này, phần tử cuối cùng là phần tử lớn nhất của dãy.
- Sắp xếp tử dưới lên
    - Cho dãy cần sắp xếp có n phần tử. Tiến hành so sánh hai phần tử đầu từ dưới lên, nếu phần tử đứng sau lớn hơn phần tử đứng trước thì đổi chỗ chúng cho nhau. Lặp lại điều này với cặp phần tử tiếp theo và tiếp tục cho đến cuối tập hợp dữ liệu, tức so sánh phần tử thứ 1 và phần tử thứ 2. Sau bước này, phần tử đầu tiên là phần tử nhỏ nhất của dãy.
    - Sau đó, quay lại so sánh và đổi chỗ nếu cần 2 phần tử cuối cho đến khi gặp phần tử thứ 2. Khi không phải đổi chỗ bất cự cặp phần tử nào trong 1 lần duyệt thì danh sách đã được sắp xếp xong.
- Minh họa:
    - Cho dãy số array = {11, 7, 13, 9, 12}
    - Lần lặp đầu tiên:
    - So sánh 11 và 7, đổi chỗ cho nhau do 11 > 7
      7 11 13 9 12
    - So sánh 11 và 13, không đổi chỗ do đã đúng thứ tự
      7 11 13 9 12
    - So sánh 13 và 9, đổi chỗ cho nhau vì 13 > 5
      7 11 9 13 12
    - So sánh 13 và 12, đổi chỗ cho nhau vì 13 > 12

7 11 9 12 13
- Lần lặp thứ 2
  7 11 9 12 13 -> 7 11 9 12 13
  7 11 9 12 13 -> 7 9 11 12 13
  7 9 11 12 13 -> 7 9 11 12 13
  7 9 11 12 13 -> 7 9 11 12 13
- Lần lặp thứ 3
  7 9 11 12 13 -> 7 11 9 12 13
  7 9 11 12 13 -> 7 9 11 12 13
  7 9 11 12 13 -> 7 9 11 12 13
  7 9 11 12 13 -> 7 9 11 12 13
- Tại đây, không có cặp phần tử nào đổi chỗ cho nhau nên thuật toán sẽ dừng lại.

3.3 Đánh giá

- Best case : O(n)
- Average case : $O(n^2)$
- Worst case : $O(n^2)$
- Không gian bộ nhớ sử dụng : O(1)

4. Heap sort

4.1 Sơ lược

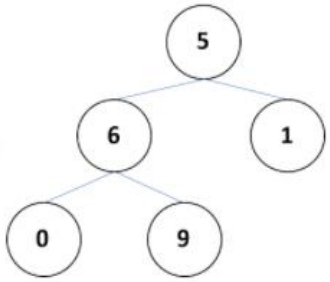
- Hay còn gọi là sắp xếp vun đống. Sử dụng kĩ thuật so sánh (comparison-based) dựa trên Binary Heap. Mang trong mình tính chất của cây nhị phân hoàn chỉnh (Complete Binary Tree: là cây mà mọi level được lấp đầy, trừ level cuối, và mọi node nằm từ trái sang).
- **Ý tưởng:** một Binary Heap có các node được sắp xếp theo thứ tự cụ thể. Đó là giá trị node parent luôn lớn hơn node con của nó (xét trong trường hợp tăng dần), được gọi là max-heap. Vì thế ở phần này ta sẽ trình bày dưới dạng mảng kèm dạng cây để dễ hình dung.

4.2 Thuật toán

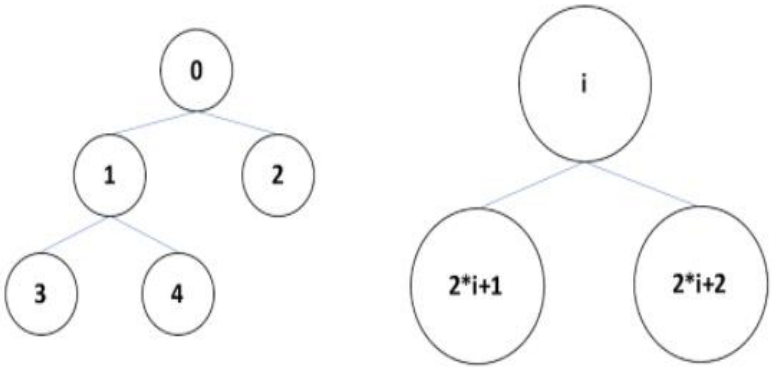
- Cách tạo ra Binary Heap Tree từ một mảng ban đầu:

Index	0	1	2	3	4
Array	5	6	1	0	9

Chuyển sang Binary Heap Tree



- Nếu ta thay bằng Index, ta có thể tính được index "i" có 2 node con lần lược là "2*i + 1" và "2*i + 2", ta gọi các phần tử này là các phần tử liên đới

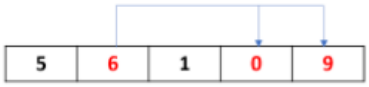


- Hành động biến đổi một cây như trên về dạng cấu trúc Heap được gọi là Heapify. Cấu trúc heap luôn luôn là dạng cây nhị phân hoàn chỉnh trong suốt qua trình biến đổi.
- Ban đầu ta cần xây dựng một max heap. Tức là gọi hàm heapify trên tất cả các node trừ node lá.
- Psedocode:

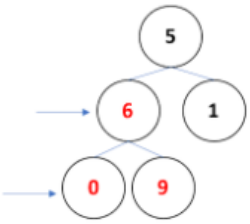
```
procedure create_max_heap(array, n) is
        loop (i = n/2, I >= 0, --i)
            heapify(array, n, i)


procedure heapify(array, n, i) is
        largest  i
        left <- 2*i + 1
        right <- 2*i + 2
        if left < n and array[left] > array[right]
            then largest = left
```

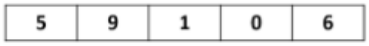
```
if left < n and array[right] > array[right]

        then largest = right

swap(array[largest], array[i])
heapify(array, n, largest)
```

- Mô tả thuật toán

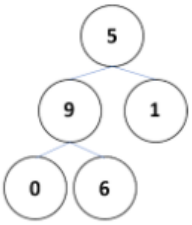


So sánh các phần tử liên đới: 9 > 6 > 0, nên swap 9 và 6

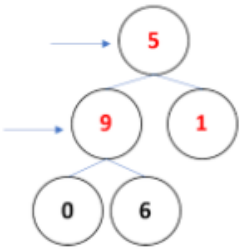


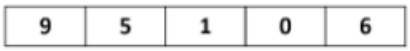


Sau khi swap. Value 6 nằm ở node lá, tức là không có phần tử liên đới. Ta tiếp tục xét đến index 0

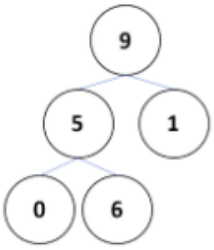


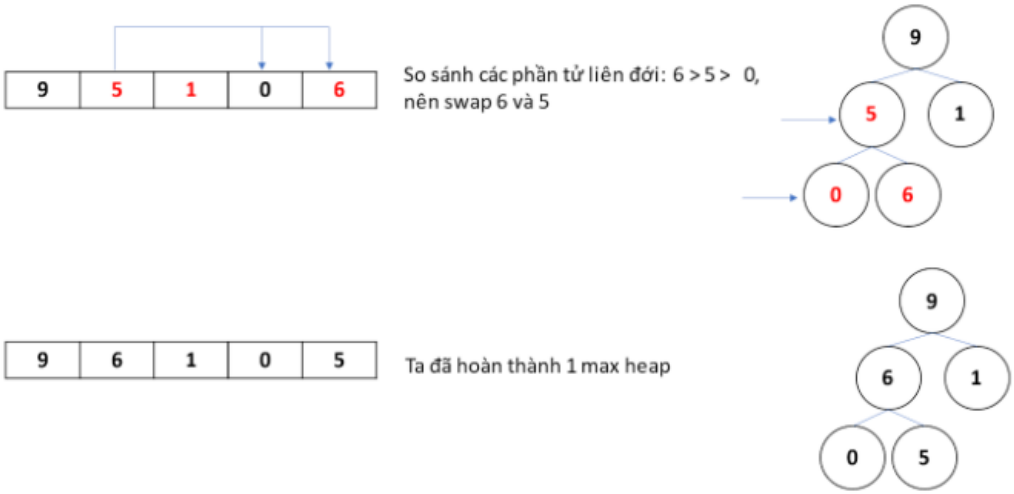
So sánh các phần tử liên đới: 9 > 5 > 1, nên swap 9 và 5



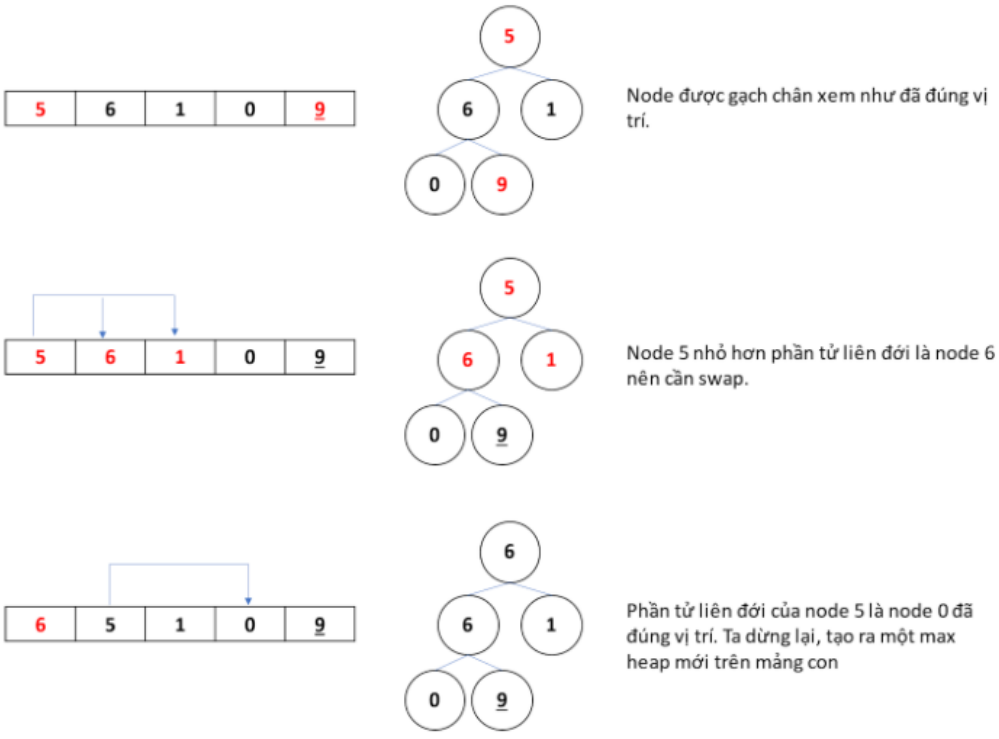


Ta thấy node 5 có phần tử liên đới, ta gọi đệ quy hàm heapify tại node này

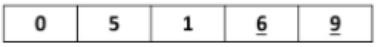


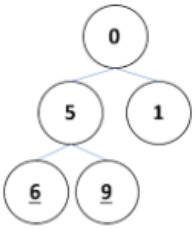


- Kết quả sau khi tạo một max heap là một dãy có phần tử lớn nhất đứng đầu. Ta sẽ dời về cuối mảng, từ đây, ta sẽ xem như có 2 mảng, mảng bên phải đã sắp xếp, mảng bên trái không còn là max heap ngay khi dời trước đó (tức là cần kiểm tra vị trí đầu tiên có phải là max heap hay không)

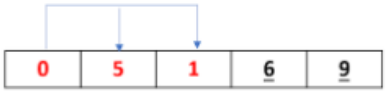


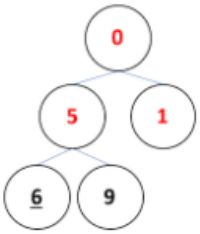
Swap max-heap về cuối mảng, node 6 đã nằm đúng vị trí:



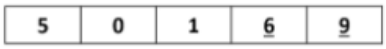


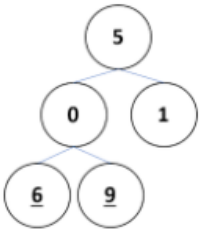
Việc swap trên đã khiến mảng con không còn là max heap, ta tiếp tục reshape



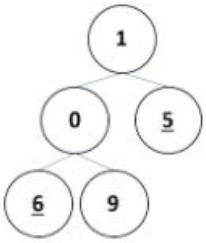


Sau khi swap node 5 và 0:

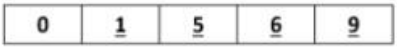


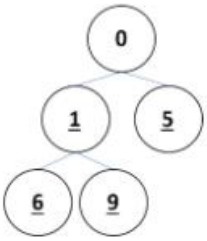


Sau khi swap 5 về cuối mảng con:



Việc swap trên tạo ra mảng con là max heap "có sẵn", tiếp tục swap 1 và 0





Lúc này điều kiện dừng là khi mảng con tạo ra có 1 phần tử → việc sắp xếp **kết thúc**.

- Pseudocode:

```
procedure max_heap_sort(array, n) is
        create_max_heap(array, n)
        loop (i ← n-1, i > 0, --i)
            swap(array[0], array[i])
            heapify(array, i, 0)
```

Trong đó: Dòng cuối ta dùng "*heapify(array, i, 0)*" vì số phần tử của mảng con phát sinh ra bằng mảng trước đó trừ 1, kiểm tra vị trí "0" có phải là max-heap hay không. Vòng lặp dừng lại khi mảng con sinh ra có 1 phần tử

4.3 Đánh giá:

- Là thuật toán sắp xếp In-pace nên không cần mảng phụ.
- Unstable
- Độ phức tạp về mặt thời gian: hàm heapify là O(log(n)), thời gian chung của thuật toán là O(nlog(n)). Độ phức tạp về mặt thời gian tăng theo hàm logarit Hiệu quả.
- Dễ cài đặt

5. Shaker sort

5.1 Sơ lược:

- Là biến thể của bubble sort.
- Ý tưởng: Còn gọi là Cocktail sort hay Cocktail shaker sort. Như đã nói ở những phần trước, Bubble sort luôn "di chuyển" phần tử từ phải sang. Sau lần lặp đầu tiên ta có phần tử đầu ở đúng vị trí, sau lần 2 sẽ có phần tử kế đầu đúng vị trí và cứ tiếp tục như thế. Shaker sort duyệt mảng theo cả 2 hướng (bidirectional bubble sort).

5.2 Thuật toán:

- Mỗi lần lặp ta có thể chia làm 2 giai đoạn:
  - Từ trái sang: so sánh phần tử trái lớn hơn phần tử phải swap. Sau lần lặp này, phần tử lớn nhất nằm ở cuối mảng (nằm đúng vị trí). Thu hẹp mảng.
  - Từ phải sang: bắt đầu từ vị trí phía trước phần tử đã nằm đúng vị trí của giai đoạn trước, quay "ngược lại" để duyệt đến đầu mảng, phần tử nhỏ nhất sẽ nằm ở đầu mảng này. Thu hẹp mảng
- 2 giai đoạn này tương tự nhau, ta có thể lập trình giai đoạn nào trước cũng được. Trong bài này ta sẽ chạy từ trái sang trước.
- Việc thu hẹp mảng là vì phần tử đã nằm đúng vị trí, không cần phải xét lại sau này.
- Mô tả:

- Cho array: [8, 4, 0, 9, 5, 6, 2]
- Giai đoạn từ trái sang:
  - So sánh 8 và 4, swap hai phần tử này: [4, 8, 0, 9, 5, 6, 2]
  - So sánh 8 và 0, swap hai phần tử này: [4, 0, 8, 9, 5, 6, 2]
  - So sánh 8 và 9, không swap: [4, 0, 8, 9, 5, 6, 2]
  - So sánh 9 và 5, swap hai phần tử này: [4, 0, 8, 5, 9, 6, 2]
  - So sánh 9 và 6, swap hai phần tử này: [4, 0, 8, 5, 6, 9, 2]
  - So sánh 9 và 2, swap hai phần tử này: [4, 0, 8, 5, 6, 2, 9]
  - Phần tử 9 đã đúng vị trí : [4, 0, 8, 5, 6, 2, 9]
- Giai đoạn từ phải sang, lúc này ta sẽ bắt đầu tại 2:
  - So sánh 2 và 6, swap hai phần tử này: [4, 0, 8, 5, 2, 6, 9]
  - So sánh 2 và 5, swap hai phần tử này: [4, 0, 8, 2, 5, 6, 9]
  - So sánh 2 và 8, swap hai phần tử này: [4, 0, 2, 8, 5, 6, 9]
  - So sánh 2 và 0, không swap: [4, 0, 2, 8, 5, 6, 9]
  - So sánh 0 và 4, swap hai phần tử này: [0, 4, 2, 8, 5, 6, 9]
  - Phần tử này đầu tiên đã đúng vị trí: [0, 4, 2, 8, 5, 6, 9]
- Thu hẹp mảng này lại và bắt đầu tại 4
  - So sánh 4 và 2, swap hai phần tử này: [0, 2, 4, 8, 5, 6, 9]
  - So sánh 4 và 8, không swap: [0, 2, 4, 8, 5, 6, 9]
  - So sánh 8 và 5, swap hai phần tử này: [0, 2, 4, 5, 8, 6, 9]
  - So sánh 8 và 6, swap hai phần tử này: [0, 2, 4, 5, 6, 8, 9]
  - Phần tử 8 đã đúng vị trí: [0, 2, 4, 5, 6, 8, 9]
- Thu hẹp mảng này lại và bắt đầu tại 6
  - So sánh 6 và 5, không swap: [0, 2, 4, 5, 6, 8, 9]
  - So sánh 5 và 4, không swap: [0, 2, 4, 5, 6, 8, 9]
  - So sánh 4 và 2, không swap: [0, 2, 4, 5, 6, 8, 9]
- Nhận thấy rằng mảng con lúc này đã được sắp xếp, ta dừng thuật toán này.

- Pseudocode:

```
procedure shaker_sort(array, n) is
        bool flag
        start = 0
        end = n -1

        loop (condition = True)
            flag = False
            loop (i <-start; I < end; ++i)
                if array[i] > array[i+1] then
                    swap(a[i], a[i+1])
                    flag = True
        if flag == False then return sorted_array
```

```
end <- end-1
flag == False
loop (i <- end; i> start; --i)
        if array[i] < array[i-1] then
            swap(a[i], a[i-1])
            flag = True

if flag == False then return sorted_array
start <- start + 1
```

- Trong đó:
  - bool flag; // false: mảng đã được sắp xếp, thoát chương trình
  - start = 0; //chỉ số đầu của mảng
  - end = n - 1; // chỉ số cuối của mảng
  - end ← end-1, start ← start + 1: thu hẹp mảng

5.3 Đánh giá:

- Độ phức tạp về mặt thời gian:
  - Trường hợp tốt nhất: O(n) – mảng đã được sắp xếp tăng dần
  - Trường hợp trung bình và xấu nhất: O(n^2)
- Inplace, bộ nhớ sử dụng O(1)
- Stable
- So với bubble sort, Shaker sort nhanh khoảng 2 lần so với Bubble sort [1]

6. Counting sort

6.1 Sơ lược:

- Counting Sort là một thuật toán sắp xếp đơn giản, ổn định và hiệu quả với thời gian chạy tuyến tính; được áp dụng cho mảng các số nguyên không âm hoặc sort các ký tự theo bảng chữ cái bằng mã ASCII.
- Là một thuật toán non-comparison-based
- Ý tưởng : đếm số lượng các phần tử có giá trị khóa riêng biệt ; từ đó có thể suy ra vị trí đúng của mỗi phần tử trong mảng đã được sắp xếp

6.2 Thuật toán :

- Input: Array A[n] (n = 10)
  A[n] = [3, 9, 2, 6, 8, 7, 2 , 1 ,4 , 3]
- Output: Sorted Array B[n]
- Đầu tiên, ta tạo một mảng phụ k phần tử (k là khoảng phạm vi của phần tử trong mảng A, thường sẽ chọn k = A_{max}); B[i] = j (i<k; j là số lần xuất hiện phần tử i trong mảng A)
  - Ở ví dụ trên, ta có mảng B như sau :
    B = [0,1,2,2,1,0,1,1,1,1,1]
  - Có thể hiểu mảng B như sau :

        - B[0] = 0 là số lần xuất hiện phần tử 0 trong mảng A
        - B[1] = 1 là số lần xuất hiện phần tử 1 trong mảng A
        - . . .
        - B[9] = 1 là số lần xuất hiện phần tử 9 trong mảng A
- Tiếp theo, sửa đổi mảng B sao cho B[i] là chặn trên của phần tử i sau khi sắp xếp:
    - B[i] = B[i] + B[i-1]
    - B = [0,1,3,5,6,6,7,8,9,10]
- Cuối cùng, ta duyệt mảng A. Cho các phần tử i vào đúng vị trí của nó trong mảng sắp xếp dựa theo B. Mỗi lần đưa i vào đúng vị trí, B[i] sẽ giảm đi 1 cho phần tử i tiếp theo.
    - A[n] = [3, 9, 2, 6, 8, 7, 2, 1, 4, 3]
    - Duyệt từ 0 đến n:
        - A[0] = 3, B[3] = 5 => 3 sẽ ở vị trí thứ 4 trong mảng đã sắp xếp C=[0, 0, 0, 0, 4, 0, 0, 0, 0, 0]
        - A[1] = 9, B[9] = 10 => 9 sẽ ở vị trí thứ 9 trong mảng đã sắp xếp C=[0, 0, 0, 0, 4, 0, 0, 0, 0, 9]
        - A[2] = 2, B[2] = 3 => 2 sẽ ở vị trí thứ 2 trong mảng đã sắp xếp C=[0, 0, 2, 0, 4, 0, 0, 0, 0, 9]
        - A[3] = 6, B[6] = 7 => 6 sẽ ở vị trí thứ 6 trong mảng đã sắp xếp C=[0, 0, 2, 0, 4, 0, 6, 0, 0, 9]
        - A[4] = 8, B[8] = 9 => 8 sẽ ở vị trí thứ 8 trong mảng đã sắp xếp C=[0, 0, 2, 0, 4, 0, 6, 0, 8, 9]
        - A[5] = 7, B[7] = 8 => 7 sẽ ở vị trí thứ 7 trong mảng đã sắp xếp C=[0, 0, 2, 0, 4, 0, 6, 7, 8, 9]
        - A[6] = 2, B[2] = 2 => 2 sẽ ở vị trí thứ 1 trong mảng đã sắp xếp (Vì lần duyệt A[2], B[2] đã được duyệt nên B[2] --) C=[0, 2, 2, 0, 4, 0, 6, 7, 8, 9]
        - A[7] = 1, B[1] = 1 => 1 sẽ ở vị trí thứ 0 trong mảng đã sắp xếp C=[1, 2, 2, 0, 4, 0, 6, 7, 8, 9]
        - A[8] = 4, B[4] = 6=> 4 sẽ ở vị trí thứ 5 trong mảng đã sắp xếp C=[1, 2, 2, 0, 4, 4, 6, 7, 8, 9]
        - A[9] = 3, B[3] = 4 => 3 sẽ ở vị trí thứ 3 trong mảng đã sắp xếp (B[3] --) C=[1, 2, 2, 3, 4, 4, 6, 7, 8, 9]
- Mảng cuối cùng cho ta mảng đã sắp xếp C=[1, 2, 2, 3, 4, 4, 6, 7, 8, 9]

Pseudo Code [2]:

```
function CountingSort(input, k)

    count ← array of k + 1 zeros
```

```
output ← array of same length as input

for i = 0 to length(input) - 1 do
   j = key(input[i])
   count[j] += 1

for i = 1 to k do
   count[i] += count[i - 1]

for i = length(input) - 1 downto 0 do
   j = key(input[i])
   count[j] -= 1
   output[count[j]] = input[i]

return output
```

6.3 Đánh giá:

Counting Sort sử dụng khoảng k trong việc duyệt mảng dẫn đến độ phức tạp của thuật toán ngoài phụ thuộc tuyến tính n, nó còn phụ thuộc vào K

- Time Complexity: O(n+k)
- Space Complexity: O(n+k)
- Counting sẽ hiệu hơn trong trường hợp khoảng phạm vi dữ liệu đầu vào không lớn đáng kể so với số lượng phần tử phải sắp xếp [3]
- Ngoài ra, còn có một thuật toán cải tiến của Counting Sort, E-Counting Sort
- Bằng việc cải tiến vòng lặp trong đoạn duyệt phần tử để đưa vào đúng vị trí trong mảng sắp xếp, E-Counting Sort chiếm ít bộ nhớ hơn Counting Sort. Đồng thời nó cũng yêu cầu ít số lần thực hiện hơn. [4]

7. Shell sort

7.1 Sơ lược:

- Shell Sort là thuật toán sắp xếp tại chỗ (In-place Comparison Sorting) có tính hiệu quả cao và là một biến thể của Insertion Sort.
- Thay vì di chuyển phần tử lên 1 vị trí trong Insertion Sort, Shell Sort lại sắp xếp các phần tử trên 1 khoảng xa trước tiên rồi thu nhỏ dần khoảng cách đó lại. Từ đó tránh được trường hợp phải hoán vị 2 phần tử có khoảng cách xa trong Insertion Sort.
- Ý tưởng: Như đã nói ở trên , Shell sort sẽ tập trung sắp xếp những phần tử có khoảng cách xa trước tiên. Sau đó sẽ tiếp tục thu nhỏ dần khoảng cách này lại đến khi bằng 1. Khi đó, các phần tử ở trên mảng cũ sẽ được sắp xếp theo thứ tự.

7.2 Thuật toán:

- Trước tiên, ta gọi "khoảng cách" để thực hiện sắp xếp trong Shell Sort là **gap** hoặc **Interval**. (Ở đây sẽ sử dụng **gap**).
- Input : Array A[] = [4,1,6,10,3,2,5] (n=7).
- Output: Sorted Array A[].
- "Khoảng cách" đầu tiên để thực hiện sắp xếp là **gap** = n/2 = 7/2 = 3 :
  - gap = 3.
  - Ta sẽ sắp xếp các phần tử có khoảng cách = gap =3.
  - [4,10,5], [1,3], [6,2].
- Các Mảng con sau khi sắp xếp sẽ thành [4,5,10], [1,3], [2,6].
  Mảng A lúc này A[] = [4, 1, 2, 5, 3, 6, 10]
- Ta tiếp tục giảm khoảng cách so sánh giữa các phần tử
  gap = n/4 = 7/4 = 1
- Lúc này khoảng cách so sánh là 1, tức mảng sắp xếp lúc này cũng chính là A[] sau khi sắp sếp lần 1: A[] = [1, 2, 3, 4, 5, 6, 10]
- Về cách sắp xếp :
  - Bắt đầu từ vị trí i = gap tức A[i]
  - So sánh A[i] với A[i-gap]
    - Nếu A[i] > A[i-gap], giữ nguyên
    - Nếu A[i] < A[i-gap], hoán vị A[i] với A[i-gap]
  - Sau đó tiếp tục tăng i (miễn i < n)

Pseudo Code

```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]  // Ciura gap sequence

# Start with the largest gap and work down to a gap of 1
# similar to insertion sort but instead of 1, gap is being used in each step
foreach (gap in gaps)
{
    # Do a gapped insertion sort for every elements in gaps
    # Each gap sort includes (0..gap-1) offset interleaved sorting.
    for (offset = 0; offset < gap; offset++)
      for (i = offset; i < n; i += gap)
      {
          # save a[i] in temp and make a hole at position i
          temp = a[i]
          # shift earlier gap-sorted elements up until the correct location for a[i] is found
          for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
          {
              a[j] = a[j - gap]
          }
          # put temp (the original a[i]) in its correct location
          a[j] = temp
      }
```

```
}
```

7.3 Đánh giá

- Shell Sort phụ thuộc nhiều vào cách chia khoảng cách
- Đối với cách chia như trên thì độ phúc tạp thời gian (Time Complextity) là $O(n^2)$
- Tùy vào cách chia khoảng cách sẽ dẫn đến độ phúc tạp thời gian nhỏ hơn, ví dụ:
  - Với cách chia gap = $\frac{n}{k^2}$ => $O(n^2)$
  - gap = $2\left(\frac{n}{2^{k+1}}\right)+1$ => $O(n^{\frac{3}{2}})$
  - gap = $2^k - 1$ => $O(n^{\frac{3}{2}})$
  - . . .

8. Flash Sort

8.1 Sơ lược:

- Hầu hết các thuật toán sort dựa trên Classify Elements (tức phân loại phần tử) đều yêu cầu cần có bộ nhớ phụ lớn, mặc dù thời gian sắp xếp lại nhanh hơn các thuật toán sắp sếp dựa theo so sánh, điển hình là Bucket sort [5]
- Flash Sort chính là cải tiến của Bucket Sort khi nó chỉ yêu cầu bộ nhớ phụ (auxiliary memory) nhỏ hơn 0.45n (với n là số phần tử trong mảng cần sắp xếp) [6]
- Ý tưởng: Flash sort dựa trên sự hoán vị tại chỗ (int situ permutation) trong 1 khoảng lớn các phần tử và sắp xếp lại từng khoảng nhỏ hơn. [5]

8.2 Thuật toán:

- Input: Array A[] = [9,8,7,6,9,4,3] (n=7)
- Output: Sorted Array A[]
- Thuật toán chung: giả sử n phần tử của mảng cần được sắp xếp được chia vào m lớp nhỏ hơn (buckets). Sau đó sắp lai các lớp theo đúng thứ tự và sắp xếp lại từng lớp đó.
- Flash Sort có 3 phần chính : Phân lớp (classification), Hoán vị (Permutaiton) và Sắp xếp (Straight Insertion)
- Với Phân lớp :
  - Ta chia n phần tử thành m lớp với công thức như sau :
    m = $(int)(0{,}45 * n) = (int)(0{,}45 * 7) = 3$
    Lúc này t có được số lượng lớp là 3 lớp
  - Với mỗi phần tử A(i) (i chạy từ 0 đến n), ta có thể tính được A(i) thuộc vào lớp nào bằng công thức :
    $K(A(i))=1+(int)((m-1)(A(i)-A_{min})/(A_{max}-A_{min}))$ [6]

Từ đó ta có thể tính được số lượng phần tử mỗi lớp (Trung bình mỗi lớp sẽ có khoảng xấp xỉ n/m phần tử. Tuy nhiên trên thực tế, lớp thứ m chỉ có duy nhất 1 phần tử là A_{max}).

- Với mảng A như trên t tính được mảng các phần tử mỗi lớp như sau :
  Bucket[] = [3, 3, 1
  Tức là lớp thứ 1 có 3 phần tử, lớp thứ 2 có 3 phần tử, lớp thứ 3 có 1 phần tử (ứng với A_{max})
- Sau đó, ta tính vị trí kết thúc của mỗi lớp trên mảng cần sắp xếp theo công thức:
  Bucket[i] = Bucket[i]+ Bucket[i+1] (i chạy từ 1 đến m-1)
  => Bucket[] = [3,6,7]
  Có thể hiểu như sau: lớp 1 sẽ là số phần tử trong lớp đó, lớp 2 sẽ kết thúc tại vị trí i=6 và lớp 3 tương tự, sẽ kết thúc tại vị trí i=7 (tức vị trí phần tử cuối cùng)

- Với Hoán vị:
  - Tại bước này sẽ di chuyển các phần tử vào đúng vị trí phân lớp của nó bằng phương pháp Hoán vị.
  - Tuy nhiên, khi hoán vị sẽ có 1 phần tử sai chỗ bị thay thế phần tử đúng chỗ. Vậy phần tử sai đó sẽ ở đâu?. Ta có thể sử dụng 1 biến để tạm giữ phần tử sai đó , tạm gọi là **hold**.
  - Ta sẽ đặt A(i) (i chạy từ 0 đến n) vào đúng vị trí phân lớp của mình bằng công thức như trên :
    $K(A(i))=1+(int)((m-1)(A(i)-A_{min})/(A_{max}-A_{min}))$
  - Sau khi hoán vị A(i) với phần tử bị nhầm lớp *(Bucket[K(A(i))])* thì t giảm Bucket[K(A(i))] xuống 1.
    Có thể hiểu rằng ta hoán vị A(i) với phần tử tại vị trí cuối cùng của mỗi lớp. Sau đó dịch vị trí này xuống 1 để chèn phần tử tiếp theo vào.
  - Việc hoán vị này sẽ diễn ra trong nhiều quá trình. Quá trình đầu tiên kết thúc khi lớp đầu tiên lấp đầy.
    Quá trình mới sẽ bắt đầu với phần tử hoán vị đầu tiên là A(j) thõa mãn
    $j < L(K(A(j)))$ [6]
  - Kết thúc khi đã thực hiện đủ n-1 bước hoặc có 1 phần tử không bị dịch chuyển, tức nó nằm đúng vị trí.
  - Với mảng A như trên, ta có thể biểu diễn toàn bộ quá trình hoán vị như sau :
    - A[0] = 9 , ta tính được K(A[0]) = 3 => A[0] thuộc lớp cuối (lớp thứ 3) . Ta hoán vị A[0] với A[bucket[2] - 1] = A[6] = 3. Lúc này t giảm Bucket[2] – 1. = 6

- Phần tử hoán bị tiếp theo là A 3 (phần tử vừa bị hoán vị). Ta tính được K(3) = 1=> phần tử 3 thuộc lớp thứ 1. Ta hoán vị phàn tử này với A[bucket[0]-1] = A[2] = 7. Ta giảm Bucket[0] - 1 =2
- Phần tử hoán bị tiếp theo là 7. K(7) = 2 => 7 thuộc lớp 2 => Hoán vị 7 với A[bucket[1] -1]=A[5] = 4. Ta giảm Bucket[1] -1 = 5.
- . . .
- Cứ hoán vị như vậy đến khi đủ n-1 bước hoặc có 1 phần tử không không bị hoán vị
- Mảng A lúc này là : A[] = [3, 5, 4, 6, 7, 8, 9]

- Với Sắp xếp :
  - Ở bước này, ta sắp xếp cục bộ ở mỗi lớp sử dụng Insertion Sort [7]
  - Ta có mảng A đã được sắp xếp như sau
    A[] = [3, 4, 5, 6, 7, 8, 9].
- Code [6]

```
DIMENSION A(1),L(1)
C ============================ CLASS FORMATION =====
ANMIN=A(1)
NMAX=1
DO I=1,N
    IF( A(I).LT.ANMIN) ANMIN=A(I)
    IF( A(I).GT.A(NMAX)) NMAX=I
END DO
IF (ANMIN.EQ.A(NMAX)) RETURN C1=(M - 1) / (A(NMAX) - ANMIN)
DO K=1,M
     L(K)=0
END DO
DO I=1,N
     K=1 + INT(C1 * (A(I) - ANMIN))
     L(K)=L(K) + 1 END DO DO K=2,M L(K)=L(K) + L(K - 1)
END DO
HOLD=A(NMAX)
A(NMAX)=A(1)
A(1)=HOLD
C ============================== PERMUTATION =====
NMOVE=0
J=1
K=M
DO WHILE (NMOVE.LT.N - 1)
    DO WHILE (J.GT.L(K))
```

```
            J=J + 1
            K=1 + INT(C1 * (A(J) - ANMIN))
        END DO FLASH=A(J)
        DO WHILE (.NOT.(J.EQ.L(K) + 1))
            K=1 + INT(C1 * (FLASH - ANMIN))
            HOLD=A(L(K))
            A(L(K))=FLASH
            FLASH=HOLD
            L(K)=L(K) - 1
            NMOVE=NMOVE + 1
        END DO
    END DO
    C ========================= STRAIGHT INSERTION =====
    DO I=N-2,1,-1
        IF (A(I + 1).LT.A(I)) THEN
            HOLD=A(I)
            J=I
            DO WHILE (A(J + 1).LT.HOLD)
                A(J)=A(J + 1) J=J + 1
            END DO
            A(J)=HOLD
    ENDIF
    END DO
    C ============================ RETURN,END FLASH1 =====
    RETURN
```

8.3 Đánh giá:

- Time Complexity: O(n).
- Unstable.
- Space Complexity: O(n).

9. Quick Sort

9.1 Sơ lược:

- Thuật toán chia để trị (Divide and Conquer), chọn một phần tử làm phần tử pivot, và các phần tử còn lại được chia theo thứ tự: nhỏ hơn xếp bên trái pivot, lớn hơn xếp bên phải pivot. Từ mảng lớn này sẽ chia thành 2 mảng con và cũng chọn pivot cho mảng con này.

9.2 Thuật Toán

- Tùy vào pivot mà thời gian chạy sẽ khác nhau. Vì vậy mà cách chọn pivot cũng rất quan trọng, Có rất nhiều cách chọn pivot:
  - Chọn đầu, cuối, hoặc nói chung là chọn random.
  - Chọn median of 3 (thường được nhiều người sử dụng)
-
- Cách chọn pivot median: Giả sử cho mảng: [1, 9, 6, 2, 9, 0, 8, 6, 2, 5] ta chọn cuối, giữa và đầu mảng là các phần tử có giá trị: 1, 9, 5,. Ta quyết định chọn 5 vì 1< 5 < Tuy nhiên cách chọn này là để tránh trường hợp xấu nhất mà không phải để trở thành trường hợp tốt nhất
- Mô tả thuật toán:
  - Cho mảng n=5 A = {57,23,45,6,11}
  - Ta chọn phần tử lính canh (**pivot**) bằng phương pháp median-three-pivot
    Ở mảng A này pivot = 45
    Mảng A lúc này A = [11,23,45,6,57]
  - Cho I chạy từ 0 đến n nếu A[i] > pivot thì chuyển A[i] về bên phải pivot . Ngược lại, nếu A[i] < pivot thì di chuyển về bên trái pivot.
    Mảng A lúc này A=[11,23,6,45,57]
  - Tiếp tục đệ quy mảng (lặp lại 2 bước như trên) với bên trái và bên phải pivot
    [11,23,6] và [57]
  - Mảng bên phải chỉ có một phần tử thì dừng đệ qui
  - Mảng bên trái, ta tiếp tục chọn pivot bằng phương pháp như trên
    [6,11,23]
  - Cho I chạy từ 0 đến n (với n là số phần tử của mảng lúc này) nếu A[i] > pivot thì chuyển A[i] về bên phải pivot . Ngược lại, nếu A[i] < pivot thì di chuyển về bên trái pivot.
    [6,11,23]
    Đệ quy mạng này lần 2
    [6] và [23]
  - Mỗi mảng chỉ có 1 phần tử => thuật toán kết thúc
    Mảng A lúc này đã được sắp xếp
    A = [6,11,23,45,57]

9.3 Đánh giá:

- Độ phức tạp về mặt thời gian:
- Tốt nhất: pivot sẽ luôn chia 2 mảng này thành 2 mảng con bằng nhau hoặc chỉ chênh lệch 1 đơn vị: O(nLog(n))

- Xấu nhất: khi mảng đã được sắp xếp theo tăng dần hoặc giảm dần, pivot ở đây là phần tử đầu hoặc cuối. Khi chia mảng thì mảng con một bên sẽ không có phần tử nào. Mảng còn lại chứa "đầy ắp" các phần tử. Khi đó việc chia mảng không có ý nghĩa: O(n^2).
- Trung bình: O(nLog(n))
- Tính chất:
  - Không stable
  - Inplace
- Để tránh xảy ra trường hợp xấu nhất như đã nêu ở trên, ta cần quan sát sơ qua mảng để có thể chọn pivot vừa ý (không cần là pivot tốt nhất). Để làm được ta cần có kiến thức về quick sort.

10. Merge Sort

10.1 Sơ lược:

- Ta đã biết về Quick sort có đặc điểm đặc biệt là chia để trị (Divide and Conquer), và merge sort cũng chia sẽ chung đặc điểm này. Chia mảng ra làm 2 phần (nếu phần tử chẵn thì 2 mảng con sau khi chia sẽ bằng nhau về số lượng phần tử, nếu là lẻ thì ta phải quy định mảng trái nhiều hơn mảng phải 1 phần tử hay ngược lại, điều này duy trì trong suốt quá trình thực hiện thực toán), các mảng nhỏ hơn này tiếp tục chia nhỏ ra nữa và cứ thế. Sau giai đoạn này, ta sẽ thực hiện tiếp giai đoạn tiếp theo đúng theo tên gọi: merge các mảng nhỏ này lại làm một. Tức là có thể sử dụng đệ quy

10.2 Thuật toán:

- Mô tả thuật toán:
  - Cho mảng gồm n = 10 phần tử:
    A = [23, 54, 12, 6, 7,10, 11, 0, 27, 47]
  - Ta duyệt lần lượt cây con bên phải và cây con bên trái bằng cách gọi đệ quy hàm sort

    - [23, 54, 12, 6, 7,10, 11, 0, 27, 47]
      - /
    \
    - [23,54,12,6,7] [10,11,0,27,47]
    - / \ / \
    - [23,54] [12,6,7] [10,11] [0,27,47]
    - / \ / \ / \ / \

- 23 54 [12,6] 7 10 11 [0,27] 47
- \ / / \ | \ / / \ |
- [23,54] 12 6 | [10,11] 0 27 |
- \ \ / | \ \ / |
- \ [6,12] | \ [0,27] |
- \ / | \ / |
- [6,12,23,54] | [0,10,11,27] |
  - \ | \ |
  - [6,7,12,23,54] [0,10,11,27,47]
    - \ /
      - \ /
  - [0,6,7,10,11,12,23,27,47,54]

- Mỗi lần gọi đệ quy, ta sẽ tách mảng thành 2 mảng con trái và phải
- o sánh từng phần tử => merge theo chiều từ bé đến lớn
- Ta được dãy sắp xếp như trên

10.3 Đánh giá:

- Độ phức tạp về mặt thời gian:
- Trọng mọi trường hợp tốt nhất, xấu nhất và trung bình, thuật toán chia mảng ra theo cách giống nhau và gộp lại theo thời gian tuyến tính. Từ đó ta suy ra được: O(nLog(n))
- Tính chất:
  - Stable
  - Chia để trị
- Inplace? Tùy vào version của merge sort. Với thuật toán mà nhóm trình bày ở đây, ta sẽ sort ngay trên chính mảng.
- Thường thấy nhất là trong danh sách liên kết. Vì lúc này các phần tử rời nhau, ko thể truy xuất ngẫu nhiên, nhưng có thể chèn vào giữa mà không phải dời như mảng. Để truy xuất tới một node bất kì, ta phải duyệt qua các node cha của nó, việc sử dụng một số thuật toán khác ví dụ như Quick sort đòi hỏi nhiều chi phí hơn merge sort. Còn nếu trong array, thì quick sort sẽ là sự lựa chọn tốt hơn.
- Không thật sự nhanh bằng các thuật toán khác.
- Thuật toán phức tạp, nếu chỉ sort trên một mảng, ta nên dùng các thuật toán khác sẽ tiết kiệm thời gian viết code hơn. Còn nếu sort trên linked list, thì đây là sự lựa chọn tuyệt vời

11.Radix Sort

11.1 Sơ lược:

- Ý tưởng: là thuật toán dễ hiểu nhất. Ngay cả đứa trẻ lớp 1 cũng có thể chạy tay ngon lành. Radix sort sẽ thực hiện từng chữ số phải nhất đến trái nhất. Sử dụng counting sort như là chương trình con để sort.

11.2 Thuật toán:

- Mô tả thuật toán:
  - Cho mảng gồm n=10 phần tử A=[12,9,7,3,2,2,10,11,100,34]
  - Ta sẽ chọn cơ số hàng đơn vị trước để gộp nhóm
  - Mảng sau khi gộp lần đầu :
    A = [10, 100, 11, 12, 2, 2, 3, 34, 7, 9]
  - Tiếp theo, chọn cơ số hàng chục để gộp nhóm
  - Mảng sau khi gộp lần 2:
    A = [100, 2, 2, 3, 7, 9, 10, 11, 34]
  - Đến cơ số hàng trăm
  - Mảng sai khi gộp lần 3:
    A = [2, 2, 3, 7, 9, 10, 11, 34, 100]
  - Mảng lúc này đã được sắp xếp
  - Ở mỗi bước gộp nhóm, ta sử dụng Counting Sort để đếm số lần xuất hiện của các phần tử hàng đơn vị, chục, trăm, ... Từ đó suy ra vị trí chính xác mỗi phần tử trong nhóm gộp.

V. FILE ORGANIZATION

File structure:

```
04:.
|   data.txt
|   DataGenerator.cpp
|   Main.cpp
|   Main.exe
|   Main.h
|
\---Sort
        BubbleSort.cpp
        Coutingsort.cpp
        FlashSort.cpp
        HeapSort.cpp
        InsertionSort.cpp
        MergeSort.cpp
        QuickSort.cpp
        RadixSort.cpp
        SelectionSort.cpp
        ShakerSort.cpp
        ShellSort.cpp
```

- Folder Sort bao gồm các thuật toán sắp xếp được cài được
- Main.cpp: file cài đặt các hàm trong việc sử dụng Command Line
- Main.exe: file thực thi sử dụng Command Line
- Main.h: chứa các thư viện và các hàm cần thiết trong quá trình cài đặt các thuật toán sắp xếp và Main.cpp
- DataGenerator.cpp: file cài đặt các hàm tạo dữ liệu một cách ngẫu nhiên bao gồm: tạo ngẫu nhiên hoàn toàn, tạo mảng đã sắp xếp, tạo mảng gần như được sắp xếp và tạo mảng đảo ngược từ mảng đã sắp xếp.
- Data.txt: một file ví dụ cho việc lấy dữ liệu từ file
- Ngoài ra, khi chạy các Command line, sẽ có them các file :
    - Output.txt: file chứa mảng đã sắp xếp
    - Input_1.txt: mảng sắp xếp từ mảng tạo ngẫu nhiên hoàn toàn
    - Input_2.txt: mảng sắp xếp từ mảng tạo ngẫu nhiên gần được sắp xếp
    - Input_3.txt: mảng sắp xếp từ mảng tạo ngẫu nhiên đã được sắp xếp
    - Input_4.txt: mảng sắp xếp từ mảng tạo ngẫu nhiên đảo ngược sắp xếp

Việc sắp xếp như vậy sẽ giúp nhóm trong quá trình thực nghiệm và cài đặt dễ dàng debug hơn, dễ nhận biết và sửa được thuật toán sắp xếp bị lỗi. Ngoài ra, gom tất cả các thuật toán sắp xếp vào trong 1 thu mục sẽ gọn gàng hơn so với việc để ở ngoai.

Một số lib sử dụng thêm:

- <fstream> : Đọc và ghi file
- <chrono> : Đo thời gian thực hiện của các hàm

VI. REFERENCE

[1] "Wikipedia," [Trực tuyến]. Available: https://en.wikipedia.org/wiki/Cocktail_shaker_sort.

[2] "https://en.wikipedia.org," Wikipedia, [Trực tuyến]. Available: https://en.wikipedia.org/wiki/Counting_sort.

[3] 12 Sep 2021. [Trực tuyến]. Available: https://www.geeksforgeeks.org/counting-sort/.

[4] K. Bajpai and A. Kots, "http://citeseerx.ist.psu.edu/," July 2014. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.681.4955&rep=rep1&type=pdf.

[5] K.-D. Neubert. [Trực tuyến]. Available: https://pages.cs.wisc.edu/~deppeler/cs400/readings/Sorting/flashpap.pdf.

[6] K.-D. Neubert, "https://pages.cs.wisc.edu/," Feb 1998. [Trực tuyến]. Available: https://pages.cs.wisc.edu/~deppeler/cs400/readings/Sorting/Dr.%20Dobb's%20Journal%20February%201998_%20The%20Flashsort1%20Algorithm.pdf.

[7] "Wikipedia," [Trực tuyến]. Available: https://en.wikipedia.org/wiki/Flashsort#Performance.