ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

ĐỀ TÀI

TÌM HIỂU, TRÌNH BÀY VÀ CÀI ĐẶT CÁC THUẬT TOÁN SEARCH & SORT

Nhóm sinh viên thực hiện:

Họ và tên	***MSSV***	***Lớp***
Nguyễn Đặng Anh Thư	21280111	21KDL1B
Nguyễn Nhật Minh Thư	21280112	21KDL1B

TP.Hồ Chí Minh, 10 - 2022

Mục lục

Lời mở đầu

Lời đầu tiên, chúng em xin bày tỏ sự tôn trọng và biết ơn tới thầy Nguyễn Bảo Long - giảng viên phần Thực hành bộ môn Cấu trúc dữ liệu & giải thuật lớp 21KDL1 đã tạo cơ hội cho chúng em được học tập thông qua đồ án và giải đáp những thắc mắc của chúng em trong quá trình thực hiện.

Tìm kiếm và sắp xếp là hai vấn đề cơ bản trong cấu trúc dữ liệu và thiết kế thuật toán. Mặc dù để giải quyết chung cho một bài toán thế nhưng để áp dụng cho từng hoàn cảnh và từng bộ dữ liệu khác nhau, rất nhiều thuật toán tìm kiếm và sắp xếp khác nhau đã được phát triển với đa dạng cách tiếp cận. Trong đồ án này, chúng em sẽ trình bày về ý tưởng, cách cài đặt và thời gian thực hiện của tổng cộng 14 thuật toán, cụ thể là 2 thuật toán tìm kiếm: Linear Search, Binary Search và 12 thuật toán sắp xếp: Selection Sort, Insertion Sort, Binary Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, Flash Sort. Ý tưởng mà chúng em trình bày trong bài báo cáo chỉ là ý tưởng nguyên thủy của thuật toán và chưa trải qua các bước cải thiện về mặt thời gian.

Chúng em cam kết rằng đây là thành quả nghiên cứu của riêng chúng em chứ không sao chép từ bất kỳ nguồn nào.

Ký tên

Nguyễn Nhật Minh Thư
Nguyễn Đặng Anh Thư

Phân công công việc

Ngày bắt đầu	Nhiệm vụ	Mô tả nhiệm vụ	Deadline	Người thực hiện	Tỷ lệ hoàn thành
05/10/2022	Lập kế hoạch	- Liệt kê những công việc cần thực hiện - Phân chia công việc cho từng người	05/10/2022	Cả nhóm	100%
05/10/2022	Tìm hiểu & trình bày thuật toán	- Linear Search - Insertion, Binary Insertion, Bubble, Shaker, Shell, Heap Sort - Trình bày từng thuật toán gồm 4 phần: Ý tưởng chung, mã giả, nhận xét và ví dụ	15/10/2022	Anh Thư	100%
05/10/2022	Tìm hiểu & trình bày thuật toán	- Binary Search - Selection, Merge, Quick, Counting, Radix, Flash Sort - Trình bày từng thuật toán gồm 4 phần: Ý tưởng chung, mã giả, nhận xét và ví dụ	15/10/2022	Minh Thư	100%
15/10/2022	Cài đặt thuật toán	- Cài đặt hàm các thuật toán Search và Sort - Vẽ biểu đồ thể hiên thời gian với từng tình trạng dữ liệu và đưa ra nhận xét từng biểu đồ	19/10/2022	Anh Thư	100%
15/10/2022	Cài đặt thuật toán	- Cài đặt hàm Random với 3 tình trạng dữ liệu (Random, Sorted, Reversed) - Cài đặt hàm Run Time ghi nhận thời gian chạy thuật toán - Cài đặt hàm Write File .txt .csv chứa kết quả sau khi thực hiện thuật toán và thời gian thực hiện	19/10/2022	Minh Thư	100%
19/10/2022	Làm báo cáo	- Đưa các nội dung đã tìm hiểu được vào báo cáo - Viết lời mở đầu, lời kết và tạo mục lục	22/10/2022	Cả nhóm	100%
23/10/2022	Tổng kết	- Họp nhóm để đánh giá kết quả công việc đã đề ra - Xem lại tổng thể báo cáo trước khi nộp	24/10/2022	Cả nhóm	100%

Bảng 1: Bảng kế hoạch và phân công công việc đồ án

Phần A

Tìm hiểu và trình bày các thuật toán

Chương I

Thuật toán tìm kiếm

1 Linear Search

1.1 Ý tưởng chung

Tìm kiếm tuyến tính (Linear search) hay còn được gọi là thuật toán tìm kiếm tuần tự (sequential search algorithm). Đây là thuật toán tìm kiếm đơn giản nhất.

Trong thuật toán tìm kiếm tuyến tính, để tìm kiếm một phần tử cho trước trong một danh sách, chúng ta sẽ duyệt lần lượt từng phần tử của danh sách đó và so sánh với giá trị của phần tử mà chúng ta muốn tìm. Nếu tìm khớp thành công thì trả về chỉ mục của phần tử trong danh sách, còn nếu không tìm thấy thì quá trình tìm kiếm sẽ tiếp tục diễn ra cho tới khi tìm kiếm hết dữ liệu của danh sách và trả về -1 nếu không tìm được phần tử trong mảng.

Tìm kiếm tuyến tính được áp dụng cho các danh sách chưa được sắp xếp (unsorted), hoặc không có thứ tự (unordered), và tập dữ liệu cần thao tác nhỏ. Trong trường hợp cần tìm kiếm với một danh sách lớn hoặc nhiều lần chúng ta nên tìm một giải thuật khác hiệu quả hơn.

1.2 Mã giả

Mã giả: Linear Search

Input: A Mảng các phần tử
n Kích thước mảng
x Giá trị cần tìm

Output: i Vị trí phần từ cần tìm (-1 nếu không tồn tại)

```
1: procedure LINEARSEARCH(A, n, x)
2:     i ← 0
3:     while i < n do
4:         if A[i] = x then
5:             return i // Vị trí của x
6:         end if
7:         i ← i + 1
8:     end while
```

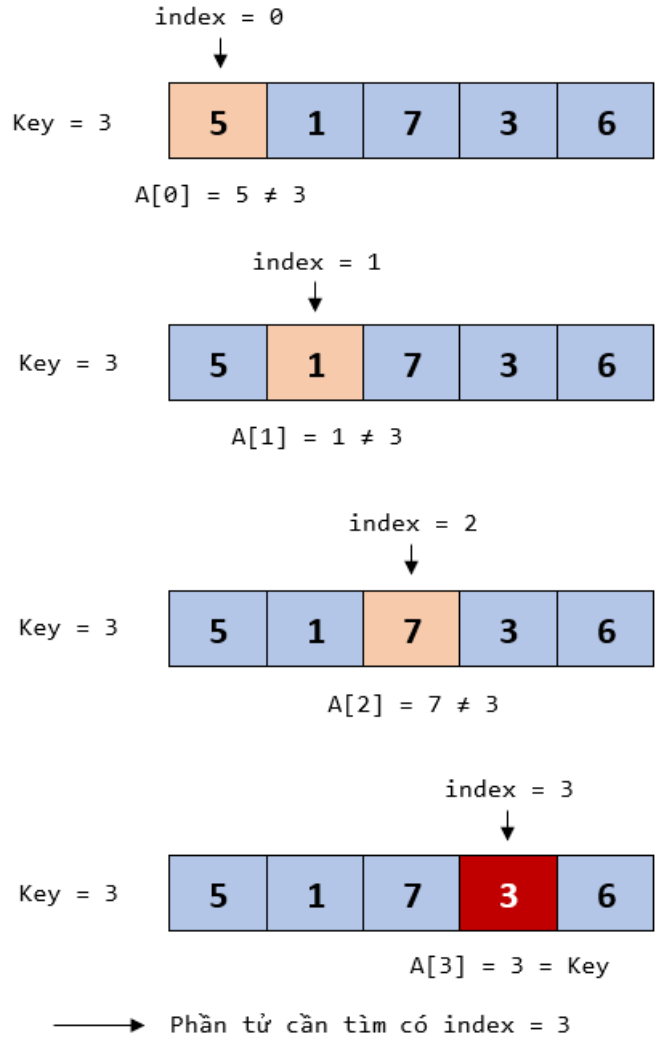
```
9:      return −1 // Không tìm thấy x
10: end procedure
```

1.3 Nhận xét

Độ phức tạp của thuật toán

- Trường hợp tốt nhất: $O(1)$ (Phần tử cần tìm ở vị trí đầu tiên của mảng)
- Trường hợp trung bình: $O(n)$
- Trường hợp xấu nhất: (Phần tử cần tìm ở vị trí cuối cùng hoặc không có trong mảng)
  - Thời gian thực hiện phép gán: $T_1 = O(1)$
  - Thời gian thực hiện phép lặp: $T_2 = O(n)$
    $\Rightarrow$ Chi phí thuật toán là: $T(n) = T_1 + T_2 = O(n)$

1.4 Ví dụ



Hình 1: Linear Search

2 Binary Search

2.1 Ý tưởng chung

Binary Search hay Tìm kiếm nhị phân là một trong những thuật toán tìm kiếm cơ bản, giúp tối ưu quá trình tìm kiếm so với Linear Search. Yêu cầu của thuật toán này là dãy phải được sắp xếp từ trước.

Cho trước một dãy A đã được sắp xếp tăng dần, ta cần tìm vị trí của phần tử có giá trị x. Ta sẽ so sánh giá trị của x với phần tử ở giữa (dãy được chia thành 2 phần là phần bên trái và phần bên phải). Nếu x khác phần tử ở giữa thì ta có thể loại bỏ một nửa dãy là nửa bên trái nếu x lớn hơn phần tử giữa và ngược lại bỏ nửa dãy bên phải nếu x nhỏ hơn phần tử giữa. Tiếp tục tìm kiếm ở nửa dãy còn lại với qui luật như trên và kết thúc khi tìm thấy phần tử cần tìm hoặc không tồn tại phần tử cần tìm trong dãy.

Mặc dù ý tưởng rất đơn giản nhưng việc thực hiện chính xác thuật toán Binary Search đòi hỏi phải chú ý đến một số điểm như điều kiện dừng và vị trí điểm giữa.

2.2 Mã giả

Mã giả: Binary Search

Input: A Mảng đã sắp xếp
n Kích thước mảng
x Giá trị cần tìm

Output: i Vị trí phần từ cần tìm (-1 nếu không tồn tại)

```
1:  procedure BINARYSEARCH(A, n, x)
2:      low ← 0
3:      high ← n − 1
4:      while low < high do
5:          mid ← (low + high)/2
6:          if A[mid] = x then
7:              return mid // Vị trí của x
8:          else if A[mid] < x then
9:              low ← mid + 1 // Chỉ cần tìm kiếm ở nửa dãy bên phải
10:         else
11:             high ← mid − 1 // Chỉ cần tìm kiếm ở nửa dãy bên trái
12:         end if
13:     end while
14:     return −1 // Không tìm thấy x
15: end procedure
```

2.3 Nhận xét

Độ phức tạp của thuật toán

- Trường hợp tốt nhất: $O(1)$ Phần tử cần tìm ở vị trí giữa của mảng

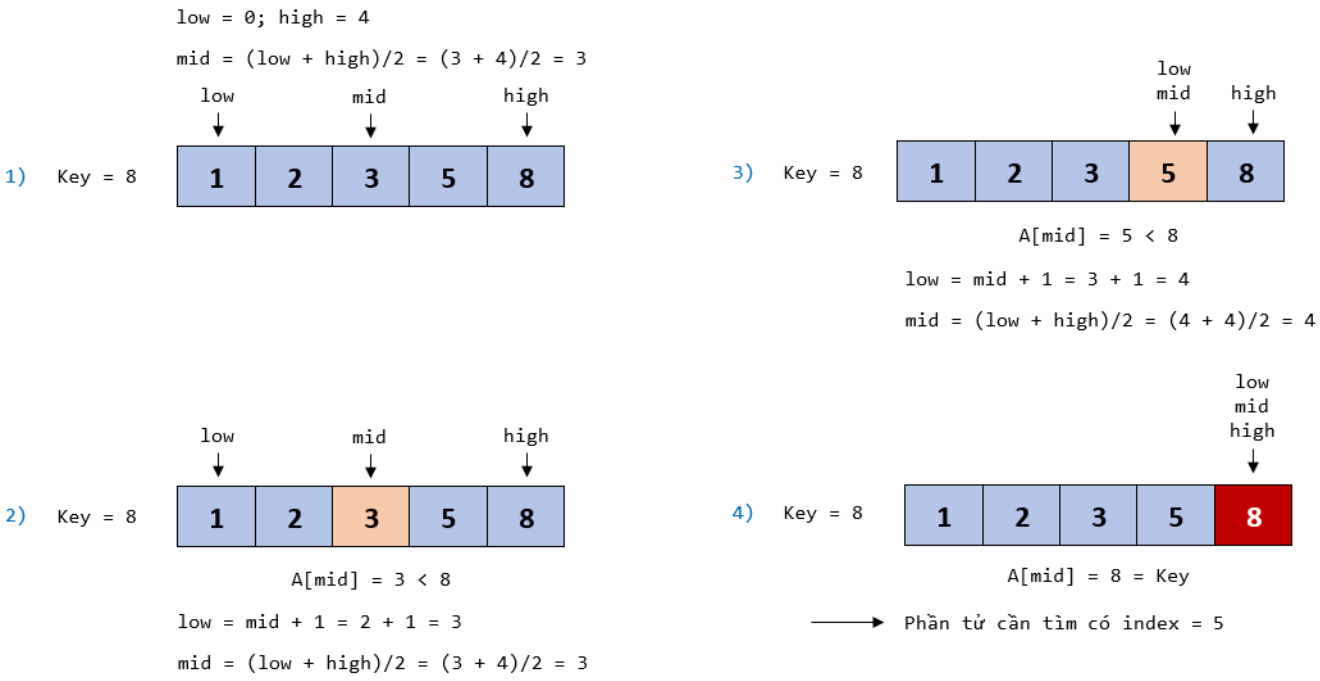
- Trường hợp trung bình: $O(log(n))$ ($log(n)$ ở đây sẽ được hiểu là $log_2(n)$)
- Trường hợp tệ nhất: Phần tử cần tìm là min/max hoặc không có trong mảng
  - Thời gian thực thi Binary Search trên một mảng gồm n phần tử: $T(n)$
  - Thời gian thực thi Binary Search trên một mảng gồm $\frac{n}{2}$ phần tử: $T(\frac{n}{2})$
  - Thời gian thực thi khi phần tử cần tìm ở giữa: $T_1 = C_1 = O(1)$
  - Thời gian thực thi khi cần cài đặt lại vị trí high và low: $T_2 = C_2 = O(1)$

$\Rightarrow$ Vậy ta có phương trình đệ quy:

$$T(n) = \begin{cases} C_1 & n = 1 \\ T\left(\frac{n}{2}\right) + C_2 & n > 1 \end{cases}$$

Giải phương trình trên bằng phương pháp truy hồi:
$T(n) = T\left(\frac{n}{2}\right) + C_2 = \left[T\left(\frac{n}{4}\right) + C_2\right] + C_2 = ... = T\left(\frac{n}{2^k}\right) + kC_2$
Quá trình đệ quy kết thúc khi: $\frac{n}{2^k} = 1 \Leftrightarrow k = log_2(n)$
$\Rightarrow T(n) = T(1) + kC_2 = C_1 + log_2(n)C_2$
Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O\left(log(n)\right)$

2.4 Ví dụ



Hình 2: Binary Search

Chương II

Thuật toán sắp xếp

1 Selection Sort

1.1 Ý tưởng chung

Selection Sort là một thuật toán sắp xếp có thể coi là đơn giản nhất, có ý tưởng khá dễ hiểu và giống với suy nghĩ tự nhiên của con người.

Ý tưởng của thuật toán là dãy sẽ được chia thành hai phần gồm phần bên trái là đã sắp xếp và phần bên phải là chưa sắp xếp. Bắt đầu với dãy đã sắp xếp là rỗng và dãy chưa sắp xếp là dãy cần sắp xếp. Ta tìm phần tử nhỏ nhất trong dãy chưa sắp xếp, lấy phần tử đó ra từ dãy chưa sắp xếp và thêm vào cuối vào dãy đã sắp xếp, tiếp tục công việc đó cho đến khi dãy chưa sắp xếp không còn phần tử nào. Lặp lại việc này cho đến khi hết dãy ta thu được dãy đã sắp xếp.

1.2 Mã giả

Mã giả: Selection Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1:  procedure SELECTIONSORT(A, n)
2:      for i = 0 to n − 2 do
3:          Min ← i
4:          for j = i + 1 to n − 1 do
5:              // Tìm vị trí của Min trong dãy chưa sắp xếp
6:              if A[j] < A[Min] then
7:                  Min ← j
8:              end if
9:          end for
10:         // Đưa Min của dãy chưa sắp xếp thêm vào cuối dãy đã sắp xếp
11:         Swap(A[i],A[Min])
12:     end for
13: end procedure
```

1.3 Nhận xét

Độ phức tạp của thuật toán

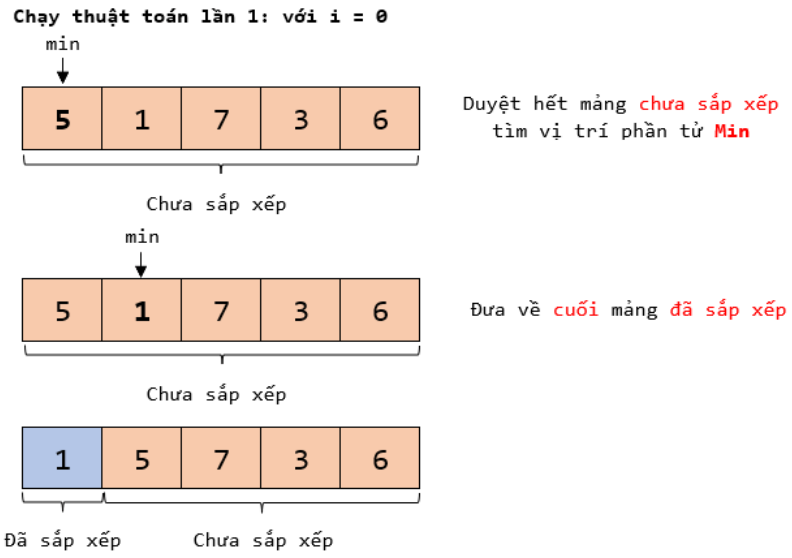
- Trường hợp tốt nhất: $O(n^2)$ Mảng được sắp xếp tăng dần
- Trường hợp trung bình: $O(n^2)$
- Trường hợp tệ nhất: Mảng được sắp xếp giảm dần
  - Vòng lặp bên ngoài chạy từ $i = 0 \to n-2$: $(n-1)$ lần
  - Thời gian thực hiện phép gán và đổi chỗ ở vòng lặp bên ngoài: $T_1 = C_1 = O(1)$
  - Vòng lặp bên trong chạy từ $j = i+1 \to n-1$: $(n-i-1)$ lần
  - Thời gian thực hiện phép so sánh và gán ở vòng lặp bên trong: $T_2 = C_2 = O(1)$

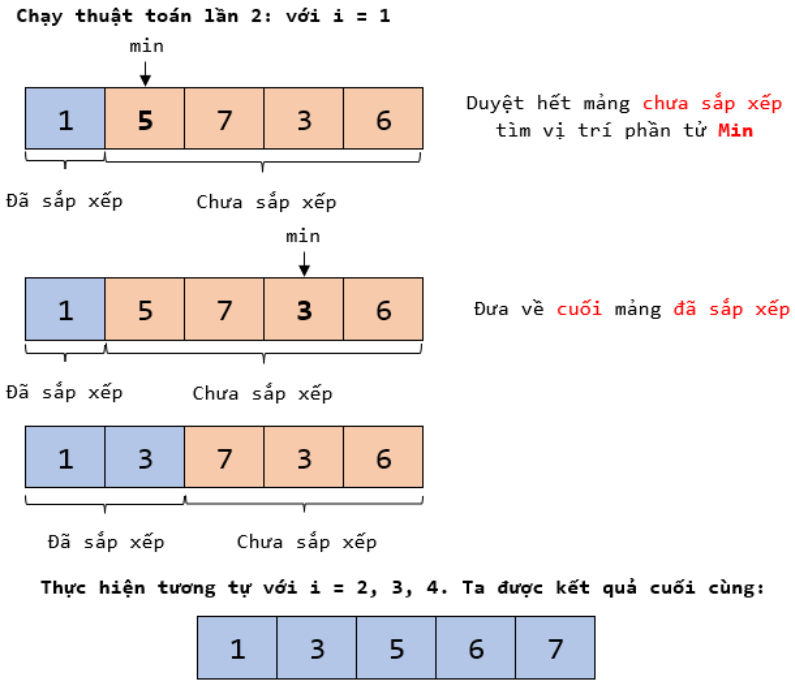
Thời gian thực thi Selection Sort trên một mảng gồm n phần tử:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} C_2\right) + C_1 = \sum_{i=0}^{n-2}(n-i-1)C_2 + C_1 \\ &= \left[(n-1)+(n-2)+\ldots+2+1\right]C_2 + (n-1)C_1 \\ &= \frac{n(n-1)}{2}C_2 + (n-1)C_1 = O(n^2) \end{aligned}$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

1.4 Ví dụ





Hình 1: Selection Sort

2 Insertion Sort

2.1 Ý tưởng chung

Insertion Sort (Sắp xếp chèn) là một thuật toán sắp xếp đơn giản hoạt động tương tự như cách chúng ta sắp xếp các lá bài tây trong bộ bài.

Giả sử rằng lá bài đầu tiên đã được sắp xếp, còn các lá còn lại chưa được sắp xếp. Sau đó, ta chọn một lá chưa được sắp xếp. Nếu lá đó lớn hơn lá đã sắp xếp, ta đặt nó ở bên phải, ngược lại đặt ở bên trái. Tương tự, các lá chưa được sắp xếp khác sẽ được lấy và đặt vào vị trí đúng của chúng.

Sắp xếp chèn được sử dụng khi số lượng phần tử nhỏ, hay khi mảng đầu vào gần như được sắp xếp, chỉ có một số phần tử bị đặt sai vị trí trong một mảng lớn hoàn chỉnh.

2.2 Mã giả

Mã giả: Insertion Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1: procedure INSERTIONSORT(A, n)
2:     for i = 1 to n − 1 do
3:         key ← A[i]
4:         j ← i − 1
```

```
5:        // Chèn key vào đúng vị trí của nó trong khoảng từ đầu mảng đến vị trí trước nó
6:        while j ≥ 0 and A[j] > key do
7:            A[j + 1] ← A[j]
8:            j ← j − 1
9:        end while
10:       A[j + 1] ← key
11:   end for
12: end procedure
```

2.3 Nhận xét

Độ phức tạp của thuật toán

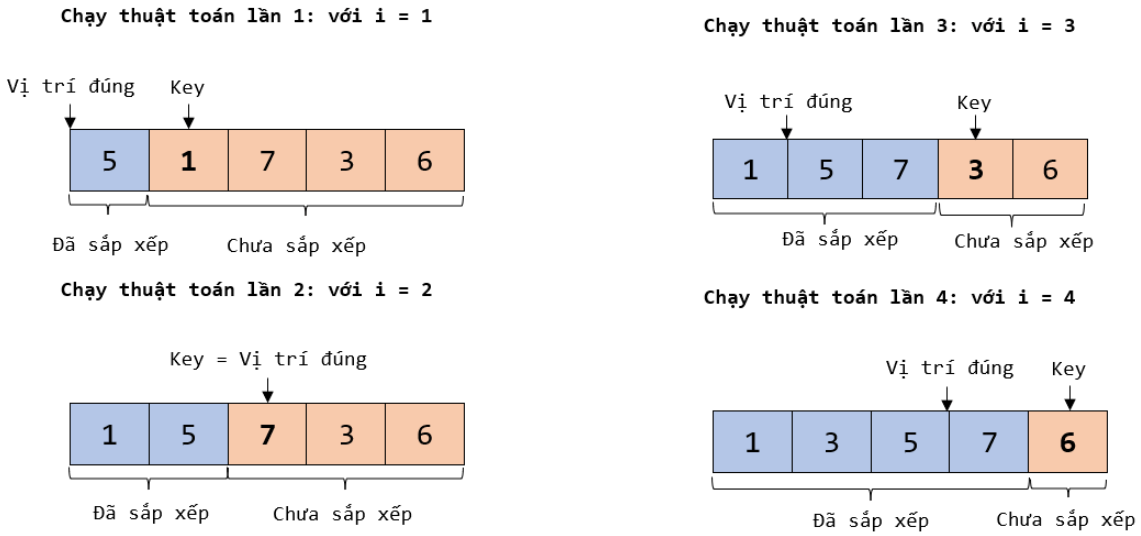
- Trường hợp tốt nhất: $O(n)$ Mảng được sắp xếp tăng dần
- Trường hợp trung bình: $O(n^2)$
- Trường hợp tệ nhất: Mảng được sắp xếp giảm dần
  - Vòng lặp bên ngoài chạy từ $i = 1 \to n - 1$: $(n - 1)$ lần
  - THời gian thực hiện các phép gán ở vòng lặp bên ngoài: $T_1 = C_1 = O(1)$
  - Vòng lặp bên trong chạy từ $j = i - 1 \to 0$: i lần; Do trong trường hợp xấu nhất đồng nghĩa với điều kiện lặp phải thỏa nhiều nhất có thể hay phải duyệt hết phần từ đầu mảng đến phần tử trước nó.
  - Các phép gán ở vòng lặp bên trong: $T_2 = C_2 = O(1)$

Thời gian thực thi Insertion Sort trên một mảng gồm n phần tử:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \left(\sum_{j=0}^{i-1} C_2 \right) + C_1 = \sum_{i=1}^{n-1} iC_2 + C_1 \\ &= [1 + 2 + ... + (n-2) + (n-1)]\, C_2 + (n-1)C_1 \\ &= \frac{n(n-1)}{2} C_2 + (n-1)C_1 = O(n^2) \end{aligned}$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

2.4 Ví dụ



Hình 2: Insertion Sort

3 Binary Insertion Sort

3.1 Ý tưởng chung

Binary Insertion Sort (Sắp xếp chèn nhị phân) là thuật toán sắp xếp có ý tưởng tương tự như Insertion Sort, nhưng thay vì sử dụng Linear Search để tìm vị trí đúng cần chèn phần tử vào, ta sẽ dùng Binary Search. Do đó, giảm số lần so sánh tìm vị trí pos để chèn phần tử từ $O(n)$ xuống $O(log(n))$

3.2 Mã giả

Để áp dụng Binary Search vào thuật toán sắp xếp này ta cần thay đổi một số chỗ để phù hợp với mục đích sắp xếp:

- Trường hợp phần tử cần chèn có vị trí tìm được là *mid* (Đã tồn tại) thì luôn chèn ra phía sau $mid + 1$ để mảng đảm bảo tính Stable (Vị trí tương đối giữa các phần tử có khóa giống nhau).

- Trường hợp phần tử cần chèn không tồn tại thì chèn vào vị trí *low*. Tại sao lại là *low* ? (Ở phần ví dụ minh họa sẽ giúp ta hiểu rõ hơn về lý do vì sao không tồn tại phần tử có giá trị *key* thì chèn vào *low*).

  - Khi $key > mid$ thì vị trí cần chèn chính là sau vị trí *mid* hay $low = mid + 1$
  - Khi $key < mid$ thì vị trí cần chèn chính là tại vị trí *mid* hay $low = mid$

Mã giả: Binary Insertion Search

```
1: // Binay Search như đã trình bày ở phần I
2: procedure BINARYSEARCH(A, low, high, key)
3:     while low ≥ high do
4:         mid ← (low + high)/2
5:         if A[mid] = key then
6:             return mid + 1 // Vì đã tồn tại phần tử có giá trị key nên ta chèn vào sau phần tử đó để mảng Stable
7:         else if A[mid] < key then
8:             low ← mid + 1 // Chỉ cần tìm kiếm ở nửa dãy bên phải
9:         else
10:            high ← mid − 1 // Chỉ cần tìm kiếm ở nửa dãy bên trái
11:        end if
12:    end while
13:    return low // Vị trí cần chèn khi chưa tồn tại phần tử có giá trị key
14: end procedure
15: // Binay Insertion Sort
Input:  A  Mảng chưa sắp xếp
        n  Kích thước mảng
Output: A  Mảng đã sắp xếp
16: procedure BINARYINSERTIONSORT(A, n)
17:    for i = 1 to n − 1 do
18:        key ← A[i]
19:        j ← i − 1
20:        pos ← binarySearch(A, 0, j, key) // Vị trí cần chèn key vào
21:        // Chèn key vào đúng vị trí của nó trong khoảng từ đầu mảng đến vị trí trước nó
22:        while j ≥ pos do
23:            A[j + 1] ← A[j]
24:            j ← j − 1
25:        end while
26:        A[j + 1] ← key
27:    end for
28: end procedure
```

3.3 Nhận xét

Độ phức tạp của thuật toán

- Trường hợp tốt nhất: $O(nlog(n))$ Mảng được sắp xếp tăng dần
- Trường hợp trung bình: $O(n^2)$
- Trường hợp tệ nhất: Mảng được sắp xếp giảm dần
  - Vòng lặp **for** bên ngoài chạy từ $i = 1 \rightarrow n - 1$: $(n - 1)$ lần
  - Các phép gán ở vòng lặp **for**: $T_1 = O(1)$

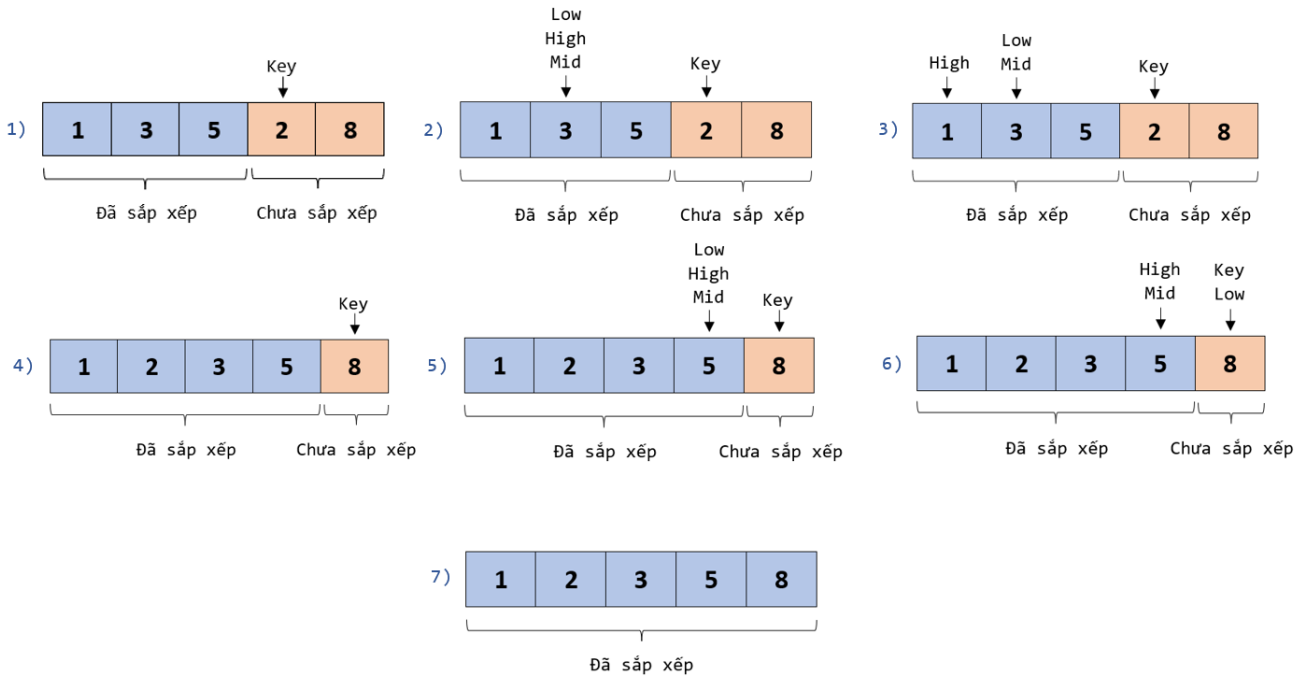
- Tìm vị trí **pos** trong khoảng $0 \to j = i - 1$ mất tối đa: $T_2 = log(i)$
- Vòng lặp **while** bên trong dời từ $j = i - 1 \to pos = 0$ (Vì trong trường tệ nhất là mảng sắp xếp giảm dần) mất tối đa: $T_3 = i$
- Các phép gán ở trong vòng lặp **while**: $T_4 = O(1)$

Thời gian thực thi Binary Insertion Sort trên một mảng gồm n phần tử:

$$T(n) = \sum_{i=1}^{n-1} T_1 + T_2 + T_3 + T_4 = \sum_{i=1}^{n-1} O(1) + log(i) + i + O(1) = \sum_{i=1}^{n-1} i \text{ (Vì } i \geq log(i) \geq O(1))$$
$$= [1 + 2 + ... + (n-2) + (n-1)] = \frac{n(n-1)}{2} = O(n^2)$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

3.4 Ví dụ



Hình 3: Binary Insertion Sort

4 Bubble Sort

4.1 Ý tưởng chung

Bubble Sort (Sắp xếp nổi bọt) là một thuật toán sắp xếp đơn giản, với thao tác cơ bản là so sánh hai phần tử kề nhau, nếu chúng chưa đứng đúng thứ tự thì đổi chỗ

Xuất phát từ phần tử cuối danh sách ta tiến hành so sánh với phần tử bên trái của nó. Nếu phần tử đang xét có khóa nhỏ hơn phần tử bên trái của nó ta tiến đưa nó về bên trái của dãy

bằng cách hoán vị với phần tử bên trái của nó. Tiếp tục thực hiện như thế đối với bài toán có n phần tử thì sau n - 1 bước ta thu được danh sách tăng dần.

4.2 Mã giả

Mã giả: Bubble Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1: procedure BUBBLESORT(A, n)
2:     for i = 0 to n − 2 do
3:         for j = n − 1 downto i + 1 do
4:             if A[j] < A[j − 1] then
5:                 Swap(A[j], A[j − 1]) // Đổi vị trí của A[j] và A[j − 1] cho nhau
6:             end if
7:         end for
8:     end for
9: end procedure
```

4.3 Nhận xét

Độ phức tạp của thuật toán

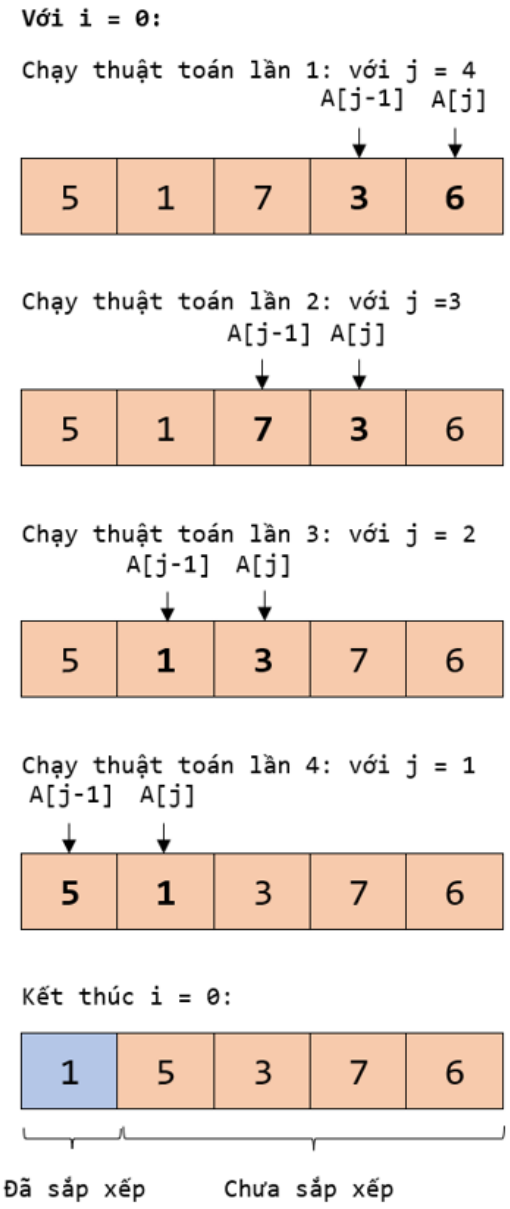
- Trường hợp tốt nhất: $O(n)$ Mảng được sắp xếp tăng dần
- Trường hợp trung bình: $O(n^2)$
- Trường hợp tệ nhất: Mảng được sắp xếp giảm dần
  - Vòng lặp **for** bên ngoài chạy từ $i = 0 \to n - 2$: $(n - 1)$ lần
  - Vòng lặp **for** bên trong chạy từ $j = n - 1 \to i + 1$: $(n - i - 1)$ lần
  - Đổi vị trí hai phần tử trong vòng lặp bên trong: $T_1 = O(1)$

Thời gian thực thi Bubble Sort trên một mảng gồm n phần tử:

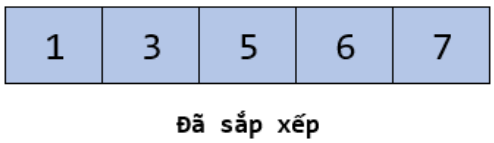
$$T(n) = \sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} T_1\right) = \sum_{i=0}^{n-2}(n-i-1)T_1$$
$$= [(n-1)+(n-2)+...+2+1]\,T_1 = \frac{n(n-1)}{2}T_1 = O(n^2)$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

4.4 Ví dụ



Thực hiện tương tự với i = 1, 2, 3 tương ứng j = 3, 2, 1. Ta được kết quả cuối cùng:



Hình 4: Bubble Sort

5 Shaker Sort

5.1 Ý tưởng chung

Shaker Sort là một cải tiến của Bubble Sort. Thay vì chỉ duyệt danh sách từ dưới lên và đưa phần tử nhỏ nhất về đầu mảng, Shaker Sort sẽ luân phiên đưa phần tử nhỏ nhất về đầu

mảng, sau đó đưa phần tử lớn nhất về cuối mảng. Nhờ việc đưa các phần tử về đúng vị trí ở cả hai đầu nên Shaker Sort sẽ làm giảm được độ lớn của mảng đang xét ở lần so sánh kế tiếp từ đó cải thiện thời gian sắp xếp dãy số.

5.2 Mã giả

Mã giả: Shaker Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1:  procedure SHAKERSORT(A, n)
2:      Left ← 0
3:      Right ← n − 1
4:      k ← 0
5:      while Left < Right do
6:          for i = Left to Right − 1 do
7:              if A[i] > A[i + 1] then
8:                  Swap(A[i], A[i + 1])
9:                  k ← i
10:             end if
11:         end for
12:         Right ← k
13:         for i = Right downto Left + 1 do
14:             if A[i] < A[i − 1] then
15:                 Swap(A[i], A[i − 1])
16:                 k ← i
17:             end if
18:         end for
19:         Left ← k
20:     end while
21: end procedure
```

5.3 Nhận xét

Độ phức tạp của thuật toán

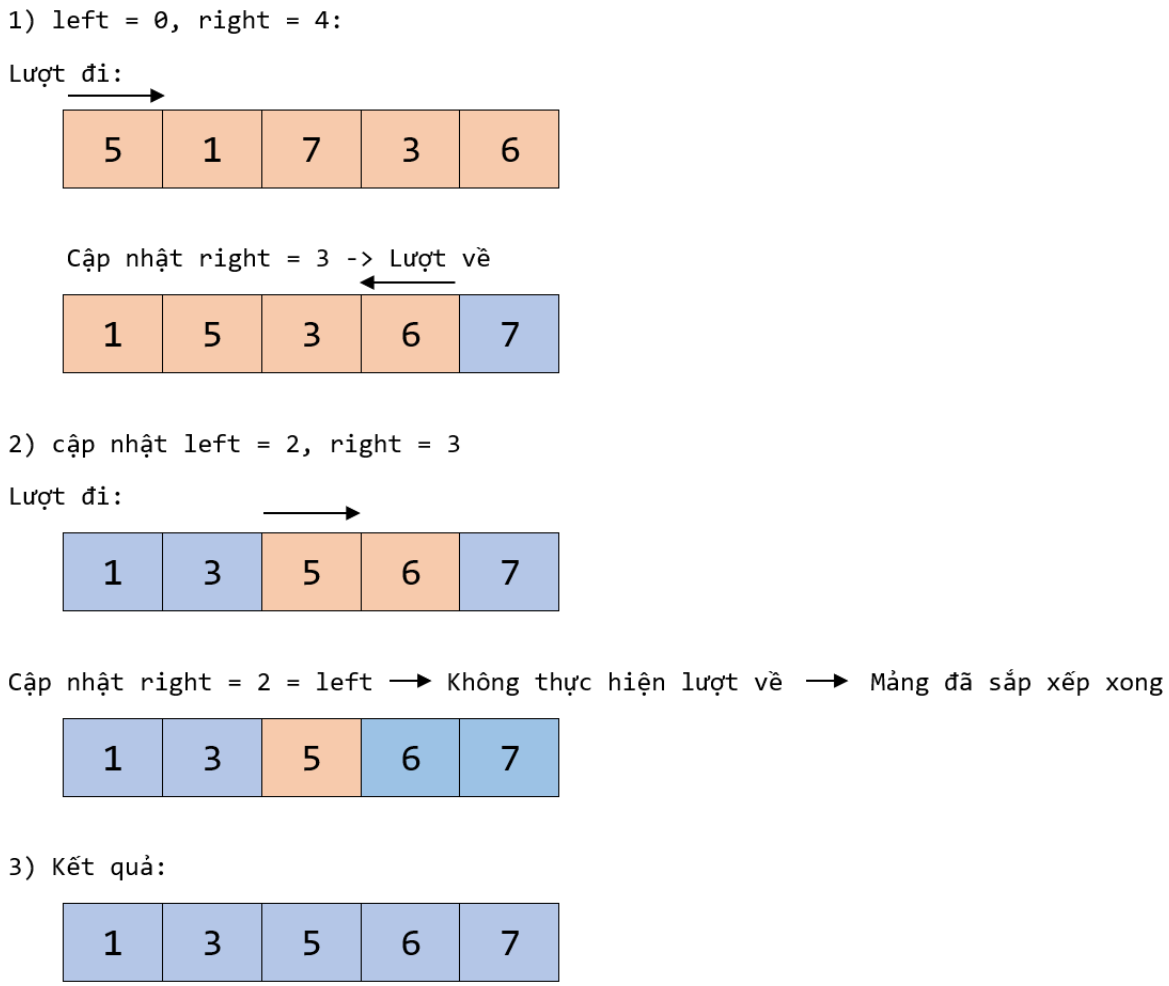
- Trường hợp tốt nhất: $O(n)$ Mảng được sắp xếp tăng dần
- Trường hợp trung bình: $O(n^2)$
- Trường hợp tệ nhất: Mảng được sắp xếp giảm dần
  - Vòng lặp **while** chạy từ $Left = 0 \rightarrow Right - 1 = n - 2$: $n - 1$ lần
  - Vòng lặp **for** thứ nhất chạy từ $i = Left = 0 \rightarrow Right - 1 = n - 2$: $(n - 1)$ lần
  - Vòng lặp **for** thứ hai chạy từ $i = Right = n - 1 \rightarrow Left + 1 = 1$: $(n - 1)$ lần
  - Chi phí **Swap** (đổi chỗ) và các phép gán: $T_1 = O(1)$

⇒ Hai vòng lặp **for** đều có số lần lặp và các câu lệnh **swap**, gán như nhau nên chi phí là bằng nhau. Thời gian thực thi Bubble Sort trên một mảng gồm n phần tử:

$$T(n) = \sum_{i=0}^{n-2}\left(2\sum_{j=0}^{n-2} T_1\right) = 2\sum_{i=0}^{n-2}(n-1)T_1$$
$$= 2\frac{(n-1)(n-1)}{2}T_1 = (n-1)^2 O(1) = O(n^2)$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

5.4 Ví dụ



Hình 5: Shaker Sort

6 Shell Sort

6.1 Ý tưởng chung

Shell Sort là một giải thuật sắp xếp dựa trên giải thuật sắp xếp chèn (Insertion sort). Ý tưởng chính của thuật toán là chia mảng ban đầu thành những mảng con mà mỗi phần tử của

mảng cách nhau 1 khoảng cách là gap. Áp dụng Insertion Sort trên mỗi mảng con để đưa các phần tử trong mảng về vị trí đúng tương đối (trong dãy con).

Sau đó tiếp tục giảm khoảng cách gap xuống 2 lần để tạo thành các mảng con mới và tiếp tục sắp xếp. Thuật toán dừng khi $gap = 1$, khi này mảng ban đầu đã được sắp xếp đúng.

6.2 Mã giả

Mã giả: Shell Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1:  procedure SHELLSORT(A, n)
2:      gap ← n/2
3:      while gap > 0 do
4:          for i = gap to n − 1 do
5:              temp ← A[i]
6:              j ← i
7:              // Insertion Sort đối với các phần tử cách nhau 1 khoảng là gap
8:              while j ≥ gap and A[j − gap] > temp do
9:                  A[j] ← A[j − gap]
10:                 j ← j − gap
11:             end while
12:             A[j] = temp
13:         end for
14:         gap ← gap/2
15:     end while
16: end procedure
```

6.3 Nhận xét

Độ phức tạp của thuật toán

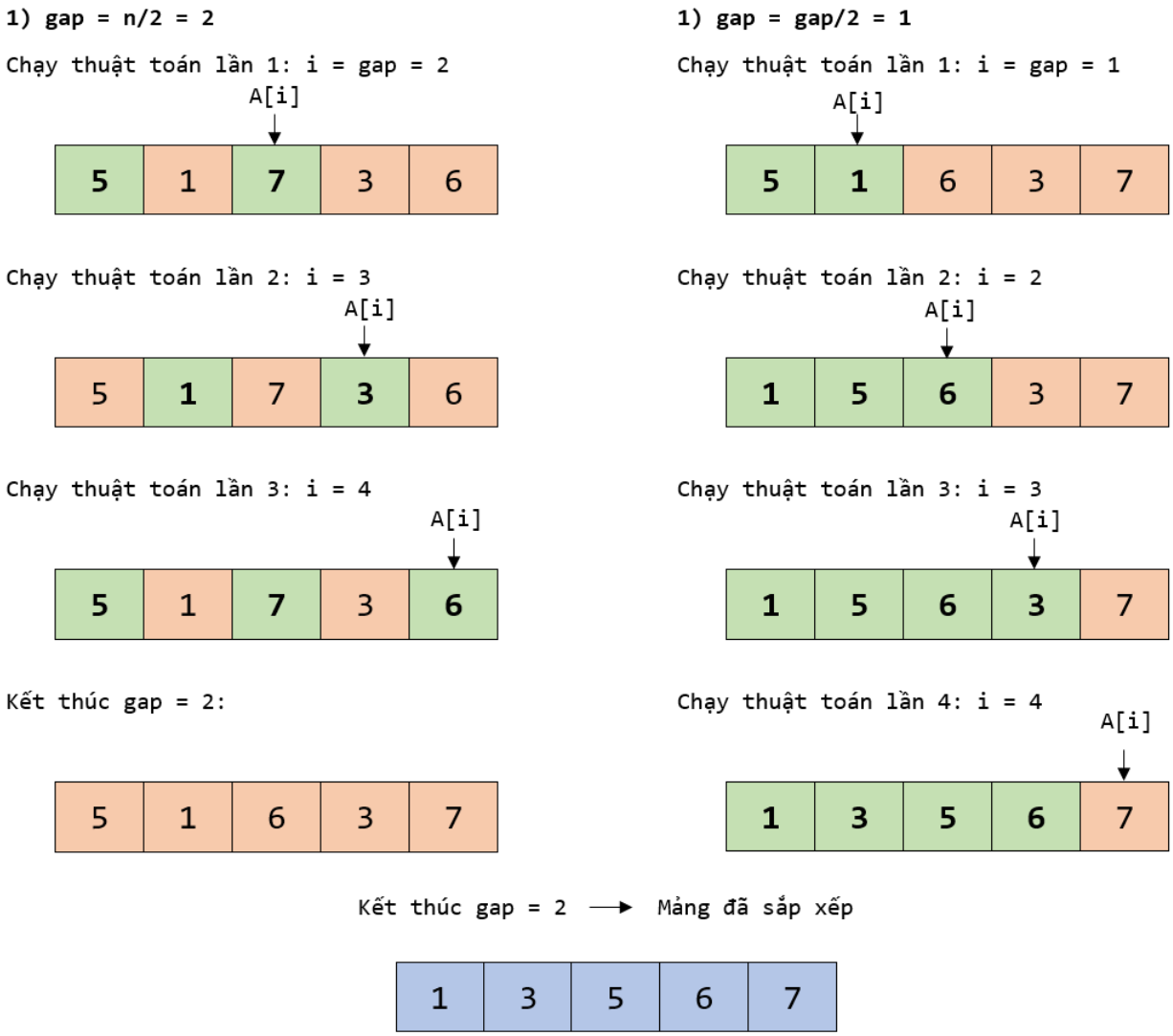
- Trường hợp tốt nhất: $O(nlog(n))$ Mảng đã được sắp xếp
- Trường hợp trung bình: $O(nlog(n))$
- Trường hợp tệ nhất: Khi $gap = 2$ thì hai mảng con $A_1 = [x_0, x_2, ..., x_{n-2}]$ và $A_2 = [x_1, x_3, ..., x_{n-1}]$ đã được sắp xếp độc lập. Đồng thời $x_{n-2} < x_1$ hay mọi phần tử $\in A_2$ đều lớn hơn mọi phần tử $\in A_1$. Và vòng lặp kết thúc khi $gap = 1$ thì ta thực hiện Insertion Sort với toàn bộ mảng $[x_0, x_1, ..., x_{n-2}, x_{n-1}]$. Khi duyệt đến:
  - Phần tử $x_i \in A_1$ sẽ cần **Swap** $\frac{i}{2}$ lần.
  - Phần tử $x_j \in A_2$ sẽ không cần **Swap** vì như đã nói ở trên mọi phần tử $\in A_2$ đều lớn hơn mọi phần tử $\in A_1$.
  - Chi phí các phép gán là: $T_1 = O(1)$

$\Rightarrow$ Số lần **Swap** của các phần tử $x_i \in A_1 = [x_0, x_2, ..., x_{n-2}]$ lần lượt là $\frac{i}{2} : \left[0, 1, ..., \frac{n-2}{2}\right]$
Thời gian thực thi **Shell Sort** trên một mảng gồm n phần tử :

$$T(n) = \sum_{i=0}^{\frac{n-2}{2}} T_1 = \left(0 + 1 + 2 + ... + \frac{n-2}{2}\right) = \frac{1}{2} * \frac{n-2}{2} * \left(\frac{n-2}{2} + 1\right)$$
$$= \frac{n(n-2)}{8} = O(n^2)$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

6.4 Ví dụ



Hình 6: Shell Sort

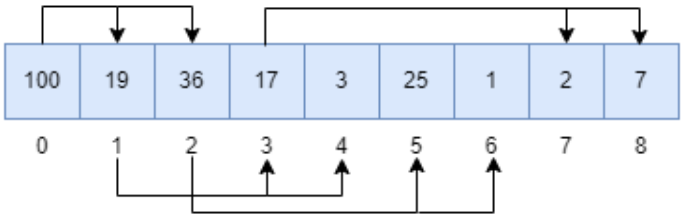
7 Heap Sort

7.1 Ý tưởng chung

Heap Sort (Sắp xếp vun đống) là thuật toán sắp xếp dựa trên cấu trúc dữ liệu binary heap (đống nhị phân). Giải thuật này được xem là bản cải tiến của Selection Sort khi thay vì tìm kiếm tuyến tính (linear-time search) để tìm ra phần tử lớn nhất, Heap Sort sẽ sử dụng cấu trúc dữ liệu heap.

Chia mảng ban đầu thành 2 mảng con: Một mảng bao gồm các phần tử đã sắp xếp và một mảng bao gồm các phần tử còn lại chưa được sắp xếp. Ban đầu, mảng đã sắp xếp là mảng rỗng, mảng chưa sắp xếp chính là mảng ban đầu.

Đầu tiên, xây dựng một cây nhị phân hoàn chỉnh từ mảng ban đầu, sau đó đưa nó về dạng max-heap, cuối cùng đưa phần tử lớn nhất (root của max-heap) về cuối mảng đã sắp xếp và loại bỏ nó khỏi heap. Lặp lại quá trình tương tự cho các phần tử còn lại. Thuật toán kết thúc khi mảng ban đầu chỉ còn 1 phần tử, đưa phần tử này về đúng vị trí và kết thúc thuật toán.



Hình 7: Minh họa xây dựng cây nhị phân từ mảng

7.2 Mã giả

Mã giả: Heap Sort

```
1:  procedure HEAPIFY(A, n, i)
2:      Largest ← i // Khởi tạo node lớn nhất là node gốc
3:      Left ← 2i + 1 // Vị trí node trái của i
4:      Right ← 2i + 2 // Vị trí node phải của i
5:      if Left < n and A[Left] > A[Largest] then
6:          Largest ← Left
7:      end if
8:      if Right < n and A[Right] > A[Largest] then
9:          Largest ← Right
10:     end if
```

```
    if Largest ≠ i then // Nếu node lớn nhất khác node gốc, cập nhật lại và tiếp tục đi xuống các node con phía dưới để cập nhật (nếu có)
        Swap(A[i], A[Largest])
        Heapify(A, n, Largest)
    end if
end procedure
Input: A  Mảng chưa sắp xếp
       n  Kích thước mảng
Output: A  Mảng đã sắp xếp
procedure HEAPSORT(A, n)
    for i = n/2 − 1 downto 0 do // Xây dựng Max Heap
        Heapify(A, n, i)
    end for
    for i = n − 1 downto 0 do
        Swap(A[0], A[i]) // Đưa phần tử lớn nhất về cuối mảng
        Heapify(A, i, 0) // Xây dựng Max Heap với các phần tử còn lại
    end for
end procedure
```

7.3 Nhận xét

Độ phức tạp của thuật toán

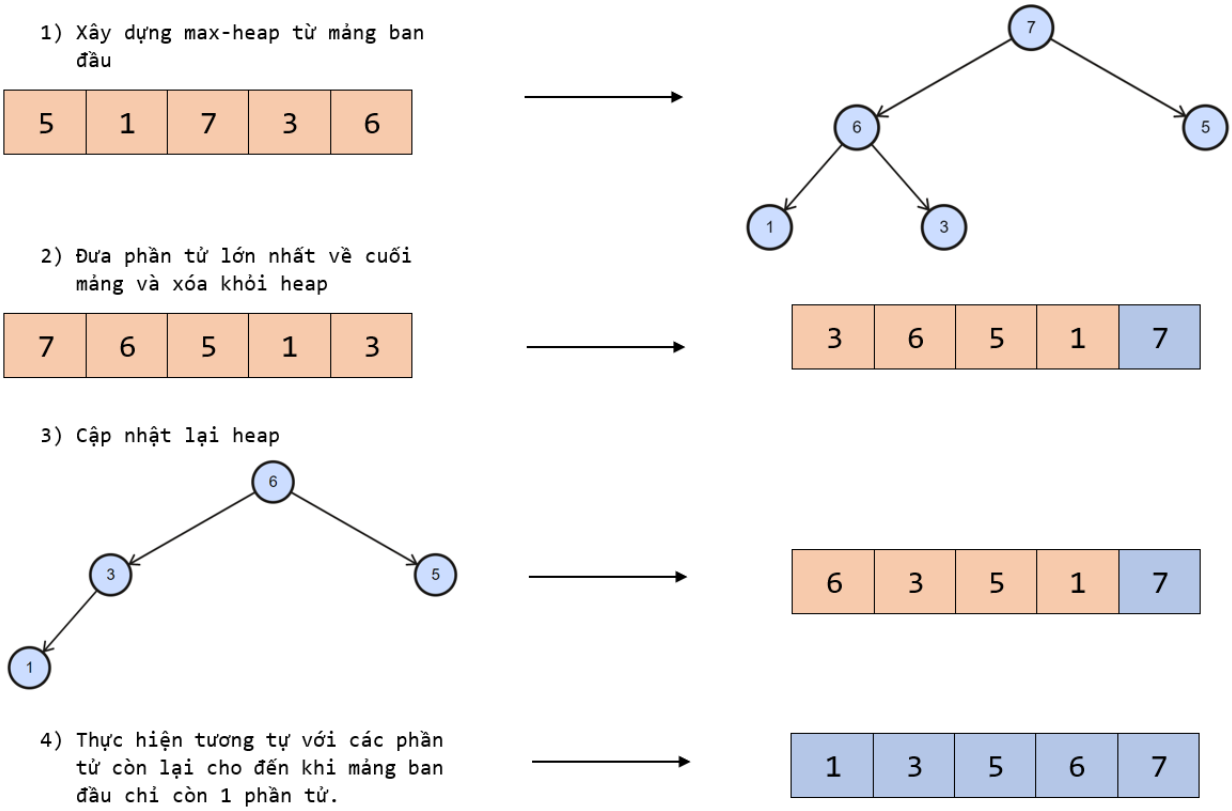
- Trường hợp tốt nhất: $O(n)$ Khi mảng gồm n phần tử có giá trị giống hệt nhau
- Trường hợp trung bình: $O(nlog(n))$
- Trường hợp tệ nhất: Khi mảng gồm n phần tử có giá trị khác nhau
  - Chi phí cho hàm **Heapify** cụ thể như sau:
    - Chi phí tìm và thay đổi vị trí của **parent node** và **child node** nếu cần (**parent node** > **child node**).
    - Nếu không phải thay đổi vị trí thì hàm kết thúc chi phí là: $O(1)$
    - Nếu phải thay đổi thì thực hiện lại hàm **Heapify** cho **child node** vừa thay đổi. Số lần thay đổi tối đa chính là chiều cao của cây là $h = [log(n+1)] + 1$: $O(h) = O(n)$
  - Vòng lặp **for** đầu tiên để xây dựng **Max Heap** có số lần lặp là: $\frac{n}{2}$ lần
  - Vòng lặp **for** thứ hai thực hiện đưa phần tử lớn nhất về cuối và thực hiện xây dựng **Max Heap** đối với các phần tử còn lại. Dễ dàng thấy việc này được thực hiện từ cuối về đầu mảng nên số lần lặp là: n lần

Thời gian thực thi **Heap Sort** trên một mảng gồm n phần tử :

$$T(n) = O(log(n)) * \frac{n}{2} + O(log(n)) * n = O(nlog(n)$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(nlog(n))$

7.4 Ví dụ



Hình 8: Heap Sort

8 Merge Sort

8.1 Ý tưởng chung

Đây là thuật toán sắp xếp hiệu quả, sử dụng kỹ thuật chia để trị tức là chia vấn đề lớn thành các vấn đề con. Khi giải quyết được vấn đề con thì sẽ giải quyết được vấn đề lớn. Thuật toán Merge Sort có hai hướng tiếp cận chính là Top-Down và Bottom-Up. Ở đây chúng ta chỉ đi vào tìm hiểu về Top-Down Merge Sort.

Ta chia mảng cần sắp xếp thành hai nửa trái và phải. Sau đó gọi đệ quy Merge Sort cho nửa bên trái và nửa bên phải. Việc gọi đệ quy sẽ được lặp lại cho đến khi mảng con không thể chia nhỏ thêm nữa (Mảng chỉ còn 1 phần tử). Cuối cùng là hợp nhất (Merge) 2 nửa dãy đã sắp xếp thành một. Hành động hợp nhất là quá trình lấy hai mảng được sắp xếp nhỏ hơn và kết hợp chúng để tạo ra một mảng lớn hơn.

8.2 Mã giả

Mã giả: Top - Down Merge Sort

```
1:  // Hàm Merge để hợp nhất hai mảng con thành một mảng chính
2:  procedure MERGE(A, Aux, Left, Mid, Right)
3:      // Sao chép các phần tử ở mảng chính vào mảng phụ
4:      for i = Left to Right − 1 do
5:          Aux[i] = A[i]
6:      end for
7:      i ← Left // Index đầu mảng bên trái
8:      j ← Mid + 1 // Index đầu mảng bên phải
9:      k ← Left // Index đầu mảng chính để sao chép phần tử thứ j hoặc k vào
10:     // So sánh hai phần tử ở mỗi mảng con và sau đó kết hợp vào mảng chính
11:     while i ≤ Mid and j ≤ Right do
12:         if Aux[i] ≤ Aux[j] then
13:             A[k] ← Aux[i]
14:             k ← k + 1
15:             i ← i + 1
16:         else
17:             A[k] ← Aux[j]
18:             k ← k + 1
19:             j ← j + 1
20:         end if
21:     end while
22:     // Sao chép những phần tử còn lại của mảng bên trái vào mảng chính
23:     while i ≤ Mid do
24:         A[k] ← Aux[i]
25:         k ← k + 1
26:         i ← i + 1
27:     end while
28:     // Sao chép những phần tử còn lại của mảng bên phải vào mảng chính
29:     while j ≤ Right do
30:         A[k] ← Aux[j]
31:         k ← k + 1
32:         j ← j + 1
33:     end while
34: end procedure
35: // Hàm cài đặt chính của thuật toán Merge Sort
```

Input:

A	Mảng chưa sắp xếp
Aux	Mảng phụ
Left	Vị trí phần tử bên trái
Right	Vị trí phần tử bên phải

Output: A Mảng đã sắp xếp

```
36: procedure MERGESORT(A, Aux, Left, Right)
37:     if Right ≤ Left then return
38:     end if
39:     Mid ← (left + right)/2
40:     mergeSort(A, Aux, Left, Mid)
41:     mergeSort(A, Aux, Mid + 1, Right)
42:     merge(A, Aux, Left, Mid, Right)
43: end procedure
```

8.3 Nhận xét

Độ phức tạp của thuật toán

- Trường hợp tốt nhất: $O(nlog(n))$ Mảng được sắp xếp tăng dần
- Trường hợp trung bình: $O(nlog(n))$
- Trường hợp tệ nhất: Mảng được sắp xếp giảm dần
  - Thời gian để chạy hàm **mergeSort** với mảng có n và n/2 phần tử là:$T(n)$ và $T(\frac{n}{2})$
  - Ở đây có hai hàm **mergeSort** cho hai mảng từ $Left \to Mid$ và $Mid + 1 \to Right$ là: $2T(\frac{n}{2})$
  - Chi phí cho phép so sánh điều kiện dừng của đệ quy và các phép gán: $C_1 = O(1)$
  - Thời gian để chạy hàm **merge**: Có tất cả ba vòng lặp đơn, trong đó gồm hai vòng lặp **for** và một vòng lặp **while** đều có số lần lặp tối đa là $\frac{n}{2}$ lần
    $\Rightarrow$ Vậy tóm lại hàm **merge** có chi phí là: $T_1 = O(n)$

Thời gian thực thi Merge Sort trên một mảng gồm n phần tử được biểu diễn dưới dạng phương trình đệ quy như sau:

$$T(n) = \begin{cases} C_1 & n = 1 \\ 2T\left(\frac{n}{2}\right) + T_1 & n > 1 \end{cases}$$

Giải phương trình trên bằng phương pháp truy hồi:

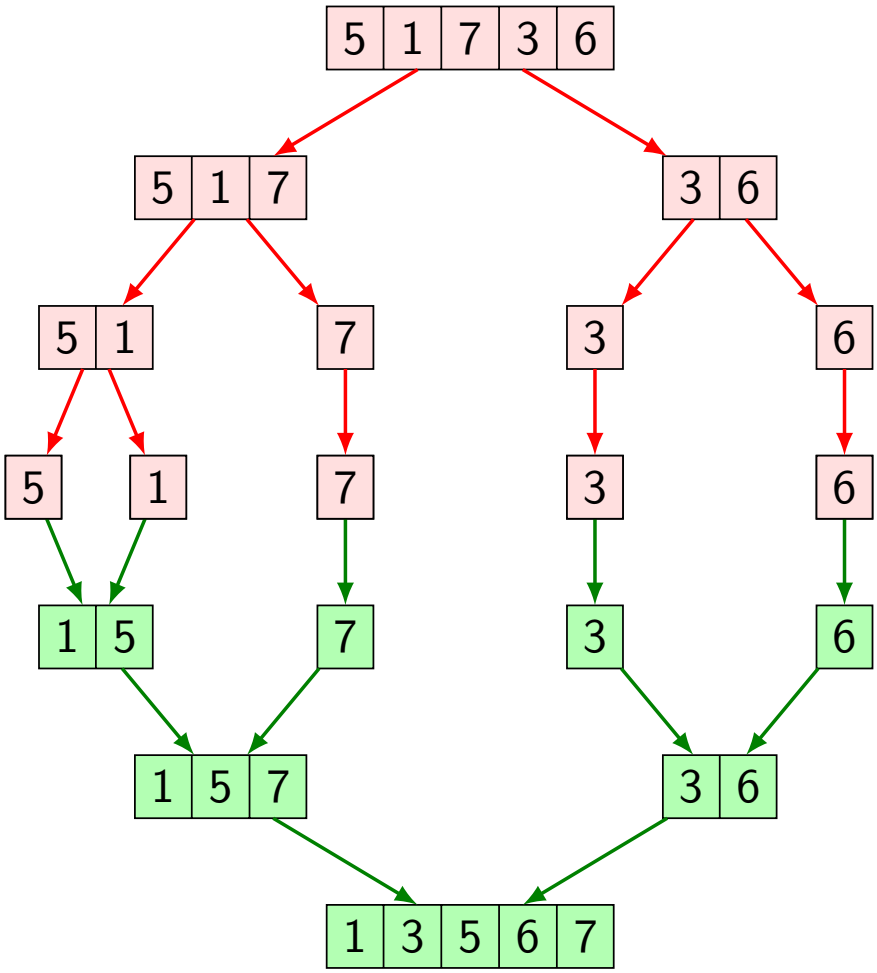
$$T(n) = 2T\left(\frac{n}{2}\right) + T_1 = 2\left[2T\left(\frac{n}{4}\right) + T_1\right] + T_1 = ... = 2^kT\left(\frac{n}{2^k}\right) + (2k-1)T_1$$

Quá trình đệ quy kết thúc khi: $\frac{n}{2^k} = 1 \Leftrightarrow k = log_2(n)$

$$\begin{aligned} \Rightarrow T(n) &= 2^kC_1 + (2k-1)T_1 = nC_1 + (2log_2(n) - 1)T_1 \\ &= nO(1) + (2log_2(n) - 1)O(n) \leq O(n) + 2log_2(n)O(n) = O(nlog(n)) \end{aligned}$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(nlog(n))$

8.4 Ví dụ



9 Quick Sort

9.1 Ý tưởng chung

Đây là thuật toán sắp xếp hiệu quả, sử dụng kỹ thuật chia để trị giống như thuật toán Merge Sort. Hoạt động bằng cách lựa chọn phần tử chốt (Pivot) và tiến hành chia dãy thành 2 phần: Phần bên trái là những phần tử có giá trị nhỏ hơn Pivot và phần bên phải là những phần tử có giá trị lớn hơn Pivot. Lặp lại quá trình như trên đối với các phần nhỏ hơn cụ thể là phần bên trái và phần bên phải ta thu được dãy đã sắp xếp xong.

9.2 Mã giả

Mã giả: Quick Sort

```
1:  // Phân vùng cho các phần tử lớn hay nhỏ hơn Pivot
2:  procedure PARTITION(A, Left, Right)
3:      i ← Left, j ← Right
4:      Pivot = A[(i + j)/2]
5:      while i < j do
6:          while A[i] < Pivot do
7:              i ← i + 1
8:          end while
9:          while A[j] > Pivot do
10:             j ← j − 1
11:         end while
12:         if i < j then
13:             Swap(A[i], A[j])
14:             i ← i + 1
15:             j ← j − 1
16:         else
17:             break
18:         end if
19:     end while
20:     Swap(A[i], A[Right]) return i
21: end procedure
22: // Hàm cài đặt chính của thuật toán Quick Sort
          A     Mảng chưa sắp xếp
Input:  Left    Vị trí phần tử bên trái
        Right   Vị trí phần tử bên phải
Output:  A    Mảng đã sắp xếp
23: procedure QUICKSORT(A, Aux, Left, Right)
24:     if Right ≤ Left then return
25:     end if
26:     i ← Partition(A, Left, Right)
27:     quickSort(A, Left, i − 1)
28:     quickSort(A, i, Right)
29: end procedure
```

9.3 Nhận xét

Độ phức tạp của thuật toán

- Trường hợp tốt nhất: $O(nlog(n))$ Khi **Pivot** có giá trị là trung vị hoặc gần trung vị của mảng
- Trường hợp trung bình: $O(nlog(n))$
- Trường hợp tệ nhất: Khi **Pivot** có giá trị là Min hoặc Max của mảng. Lúc này ta phải chạy hàm **quickSort** cho hai mảng có (n-1) và 1 phần tử.
  - Thời gian để chạy hàm **quickSort** với mảng có n và n-1 phần tử là: $T(n)$ và $T(n-1)$

- Chi phí cho phép so sánh điều kiện dừng của đệ quy và các phép gán: $T(1) = C_1 = O(1)$
- Chọn được **Pivot** là Min hoặc Max thì ta phải chạy hàm **quickSort** cho hai mảng có (n-1) và 1 phần tử. Tức là $T(n) = T(n-1) + T(1) \leq T(n)$ (Bỏ qua vì $T(1) = C_1 = O(1)$)
- Thời gian để chạy hàm **Partition**:
  - Vì **Pivot** có giá trị là Min hoặc Max của mảng nên hàm Partition chỉ có tác dụng là dời tất cả các phần tử còn lại qua bên phải **Pivot** nếu **Pivot** là Min và ngược lại là qua bên trái nếu **Pivot** là Max. Ta thấy để làm được điều này thì ta chỉ cần thời gian tuyến tính vì chỉ cần đi hết mảng và so sánh với **Pivot** sau đó đổi vị trí nếu cần.

$\Rightarrow$ Vậy tóm lại hàm **Partition** có chi phí là: $T_1 = O(n)$

Thời gian thực thi **Quick Sort** trên một mảng gồm n phần tử được biểu diễn dưới dạng phương trình đệ quy như sau:

$$T(n) = \begin{cases} C_1 & n = 1 \\ T(n-1) + T_1 & n > 1 \end{cases}$$

Giải phương trình trên bằng phương pháp truy hồi:

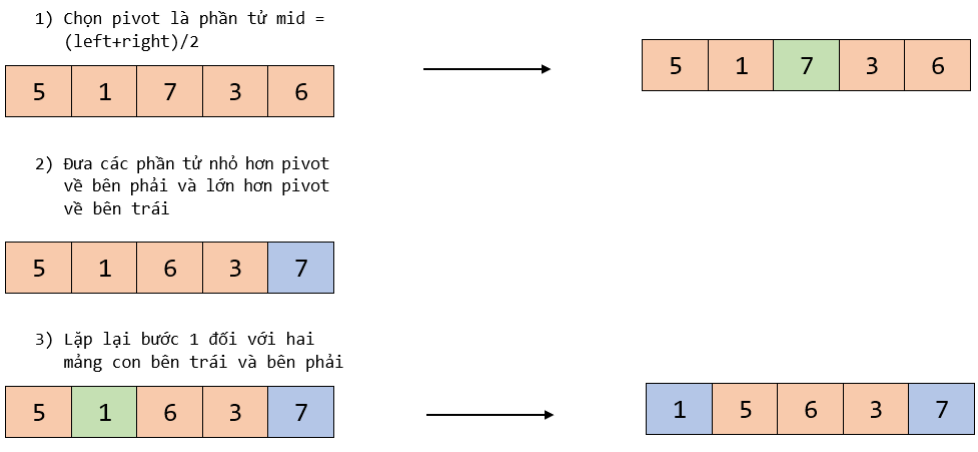
$$T(n) = T(n-1) + T_1 = [T(n-2) + T_1] + T_1 = T(n-2) + 2T_1 = ... = T(n-k) + kT_1$$

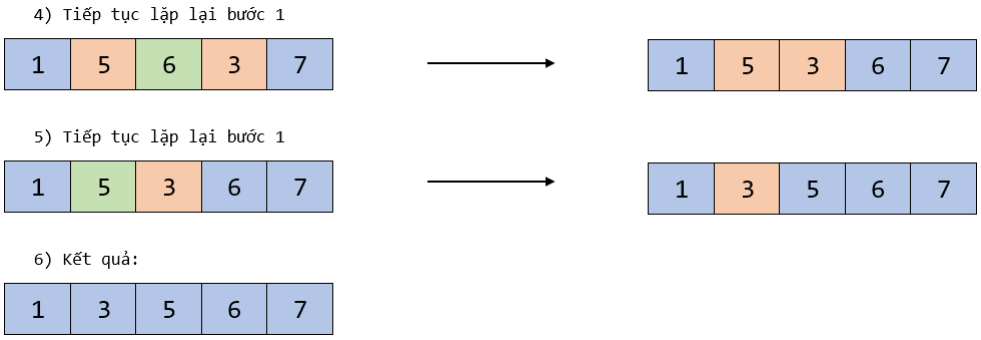
Quá trình đệ quy kết thúc khi: $n - k = 1 \Leftrightarrow k = n - 1$

$$\Rightarrow T(n) = T(1) + (n-1)T_1 = O(1) + (n-1)O(n) = O(n^2)$$

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

9.4 Ví dụ





Hình 9: Quick Sort

10 Counting Sort

10.1 Ý tưởng chung

Counting Sort là một thuật toán sắp xếp có độ phức tạp tuyến tính, khác với những thuật toán sắp xếp khác thì Counting Sort yêu cầu đầu vào là một dãy các số tự nhiên có **Range = Giá trị lớn nhất Mảng - Giá trị nhỏ nhất Mảng**. Ta không cần phải so sánh các phần tử với nhau mà sẽ dựa vào số lần xuất hiện của mỗi giá trị khác nhau, từ đó tính toán được vị trí của mỗi phần tử sau khi sắp xếp.

10.2 Mã giả

Mã giả: Counting Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1:  procedure CountingSort(A, n, Range)
2:      Count[Max(A, n) + 1] ← 0 // Khởi tạo mảng Count gồm Max + 1 phần tử có giá trị bằng 0
3:      Aux[n] ← 0 // Khởi tạo mảng Aux gồm n phần tử có giá trị bằng 0
4:      for i = 0 to n − 1 do
5:          Count[A[i]] ← Count[A[i]] + 1 // Đếm số lần xuất hiện của giá trị A_i
6:      end for
7:      // Chuyển từ đếm số lần xuất hiện sang index vị trí sau vị trí cuối cùng của A_i
8:      for i = 1 to Range + 1 do
9:          Count[i] ← Count[i] + Count[i − 1]
10:     end for
11:     for i = n − 1 downto 0 do
12:         Aux[Count[A[i]] − 1] ← A[i]
13:         Count[A[i]] ← Count[A[i]] − 1
14:     end for
15:     for i = 0 to n − 1 do
16:         A[i] ← Aux[i]
17:     end for
18: end procedure
```

10.3 Nhận xét

Độ phức tạp của thuật toán

- Trường hợp tốt nhất: $O(n)$ Khi $k = Range = 1$, tức là các phần tử trong mảng đều có giá trị bằng nhau
- Trường hợp trung bình: $O(n + k)$ (Với $k = Range$)
- Trường hợp tệ nhất: Khi $k = Range$ quá lớn, tức là $Max - Min$ quá lớn so với số phần tử của mảng là n
  - Vòng lặp **for** thứ 1 chạy từ $i = 0 \to n - 1$: n lần
  - Vòng lặp **for** thứ 2 chạy từ $i = 1 \to Range$: $k = Range$ lần
  - Vòng lặp **for** thứ 1 chạy từ $i = n - 1 \to 0$: n lần
  - Vòng lặp **for** thứ 1 chạy từ $i = 0 \to n - 1$: n lần
  - Chi phí cho phép gán bên trong và ngoài vòng lặp đều là: $T = O(1)$

$\Rightarrow$ Vậy tóm lại hàm **countingSort** có chi phí là:

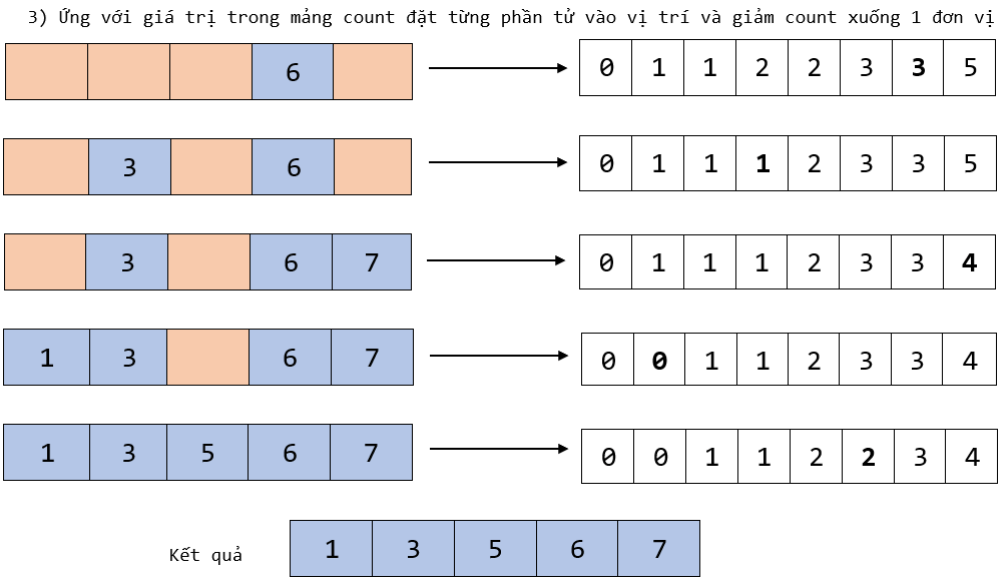
$$T(n) = nO(1) + kO(1) + nO(1) + nO(1) = 3O(n) + O(k) = O(n + k)$$

Vì chi phí thuật toán **countingSort** có hai tham số là n và k nên chi phí thuật toán sắp xếp mảng có n phần tử sẽ phụ thuộc vào k, cụ thể:

- Khi $k = O(n)$ thì $O(n + k) = O(n)$ (Trường hợp tốt nhất thì thời gian sắp xếp là tuyến tính $O(n)$)
- Khi $k = O(n^2)$ thì $O(n + k) = O(n^2)$ (Không còn vượt trội so với các thuật toán sắp xếp khác)
- Khi $k > O(n^2)$ thì thuật toán lúc này trở nên vô dụng vì chi phí quá lớn

Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n + k)$

10.4 Ví dụ



Hình 10: Counting Sort

11 Radix Sort

11.1 Ý tưởng chung

Radix Sort là một thuật toán rất hiệu quả và tốc độ thực thi nhanh, việc sắp xếp không dựa vào so sánh các phần tử với nhau giống như Counting Sort. Giả sử ta có dãy tự nhiên có n phần tử và các phần tử có tối đa d chữ số, ta sẽ lần lượt phân loại các phần tử trong dãy lần lượt theo hàng đơn vị (chữ số có ít ý nghĩa nhất), hàng chục,... đến hàng thứ d.

11.2 Mã giả

Mã giả: Radix Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Range Max(A)-Min(A)
Output: A Mảng đã sắp xếp

```
1: procedure COUNTINGSORT(A, n, Range = 10, div)
2:     Count[Range + 1] ← 0 // Khởi tạo mảng Count gồm Range + 1 phần tử có giá trị bằng 0
3:     Aux[n] ← 0 // Khởi tạo mảng Aux gồm n phần tử có giá trị bằng 0
4:     for i = 0 to n − 1 do
5:         Count[(A[i]/div) mod 10] + 1 // Đếm số lần xuất hiện của mỗi giá trị
6:     end for
```

```
    // Chuyển từ đếm số lần xuất hiện sang index vị trí sau vị trí cuối cùng của A_i
    for i = 1 to Range + 1 do
        Count[i] ← Count[i] + Count[i − 1]
    end for
    for i = n − 1 downto 0 do
        Aux[Count[(A[i]/div) mod 10] − 1] ← A[i]
        Count[A[i]] ← Count[A[i]] − 1
    end for
    for i = 0 to n − 1 do
        A[i] ← Aux[i]
    end for
end procedure
// Cài đặt hàm radixSort
procedure RADIXSORT((A, n))
    m ← max(A, n)// Hàm max tìm max của mảng A có n phần tử
    // Thực hiện hàm countingSort cho từng chữ số
    div ← 1
    while (m/div) > 0 do
        countingSort(A, n, div)
        div ← 10 ∗ div
    end while
end procedure
```

11.3 Nhận xét

Độ phức tạp của thuật toán

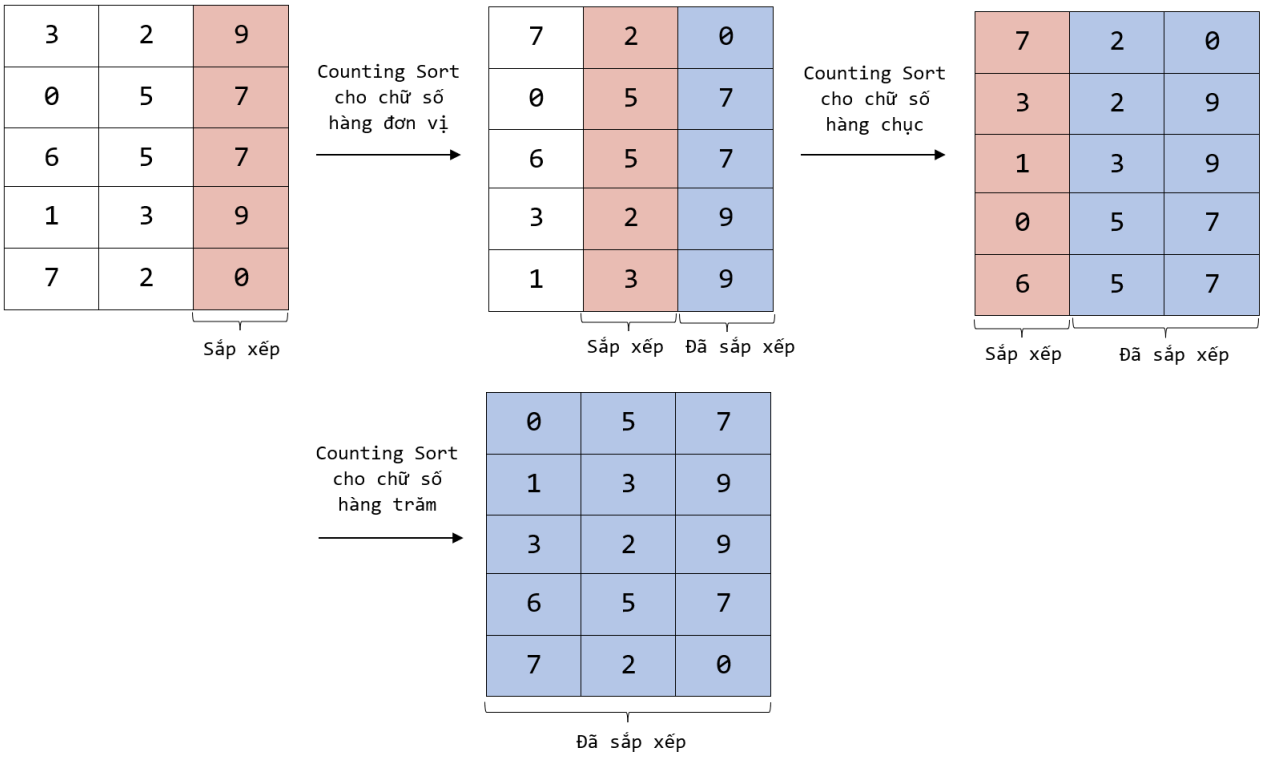
- Trường hợp tốt nhất: $O(n)$ Khi d là số rất nhỏ và hữu hạn (Với d = Số chữ số của phần tử lớn nhất trong mảng)
- Trường hợp trung bình: $O(nd)$ hay $O(d(n+k))$
- Trường hợp tệ nhất: Khi d quá lớn so với với n là số phần tử của mảng
  - Hàm tìm **max** của mảng có chi phí: $T_1 = O(n)$
  - Vòng lặp **while** chạy số lần tương ứng với số chữ số của $m = max(A, n)$: d lần
  - Hàm **countingSort** có chi phí là $T_2 = O(n+k)$ như đã tính ở trên

⇒ Vậy tóm lại hàm **radixSort** có chi phí là:

$$T(n) = T_1 + dT_2 = O(n) + dO(n+k) = O\left(d(n+k)\right)$$

Vì chi phí thuật toán **radixSort** có ba tham số là n, k và d. Nhưng vì $k = Range$ của các số có một chữ số nên $k \leq 10$ nên chi phí thuật toán sắp xếp mảng có n phần tử trong trường hợp tệ nhất sẽ là: $O(d(n+k))$ hay $O(nd)$

11.4 Ví dụ



Hình 11: Radix Sort

12 Flash Sort

12.1 Ý tưởng chung

Flash sort là một thuật toán sắp xếp có tốc độ rất nhanh và thực hiện sắp xếp bằng cách phân lớp dữ liệu với số lớp $m = 0.43n$ sẽ là tối ưu. Sau đó tính số phần tử của mỗi lớp rồi bắt đầu phân lớp từng phần tử vào đúng lớp của phần tử đó. Phần tử thứ i sẽ thuộc phân lớp thứ $k = \textbf{floor}\left((m-1)\frac{A[i]-min}{max-min}\right)$. Như vậy với công thức trên ta có k luôn nằm trong khoảng $[0, m-1]$.

Sau khi tính được phân lớp của phần tử thì ta tiến hành đưa phần tử đó vào trong phân lớp đó, đồng thời lấy ra một phần tử trong phần lớp vừa bỏ vào vì số lượng phần tử trong một lớp đã cố định từ đầu. Ta tiếp tục lặp lại quá trình trên cho đến khi không có phần tử nào ở sai lớp.

Việc phân lớp cũng đồng nghĩa với việc là các phần tử trong lớp sau sẽ đều lớn hơn các phần tử của lớp trước nên cuối cùng ta chỉ cần sắp xếp thứ tự các phần tử trong từng lớp bằng **Insertion Sort** là thu được mảng đã sắp xếp.

12.2 Mã giả

Mã giả: Flash Sort

Input: A Mảng chưa sắp xếp
n Kích thước mảng
Output: A Mảng đã sắp xếp

```
1: procedure FLASHSORT(A, n)
2:     // Tìm Min và Index của phần tử là Max của mảng A
3:     minValue ← Min(A, n)
4:     indexMaxValue ← IndexOfMax(A, n)
5:     m ← floor(0.43n)
6:     Class[m] = 0 // Tạo mảng Class gồm m phần tử có giá trị 0
7:     if A[indexMaxValue] = minValue then
8:         return
9:     end if
10:    // Tính phân lớp thứ k của phần tử A_i
11:    for i = 0 to n − 1 do
12:        k = (m − 1)floor [(a[i] − minValue)/(A[indexMaxValue] − minValue)]
13:        Class[k] ← Class[k] + 1 // Tính số phần tử của mỗi phân lớp
14:    end for
15:    for i = 1 to m − 1 do
16:        Class[i] ← Class[i] + Class[i − 1] // Chuyển từ số phần tử mỗi lớp sang index sau
       phần tử cuối cùng mỗi lớp
17:    end for
18:    Swap(A[indexMaxValue], A[0])
19:    NMove ← 0
20:    J ← 0
21:    k = m − 1
22:    // Chỉ cần di chuyển tối đa (n-1) phần tử trong mảng để các phần tử được về đúng lớp
23:    while NMove < n − 1 do
24:        // Đảm bảo rằng lớp có đủ các phần tử thuộc về lớp đó không cần phải duyệt tiếp
25:        while J > Class[k] − 1 do
26:            J ← J + 1
27:            k = (m − 1)floor [(a[i] − minValue)/(A[indexMaxValue] − minValue)]
28:        end while
29:        FLASH ← A[J]
30:        // Tìm phân lớp của FLASH và đưa vào đúng phân lớp. Đồng thời HOLD giữ
       phần tử bị FLASH thế chỗ sau đó lặp lại quá trình trên với FLASH = HOLD
31:        while J ≠ Class[k] do
32:            k = (m − 1)floor [(FLASH − minValue)/(A[indexMaxValue] − minValue)]
33:            Class[k] ← Class[k] − 1
34:            HOLD ← A[Class[k]]
35:            A[Class[k]] = FLASH
36:            FLASH = HOLD
37:            NMove ← NMove + 1
38:        end while
39:    end while
40:    insertionSort(A, n) // Vì các phần tử đã về đúng phân lớp nên việc sắp xếp bằng
       insertionSort sẽ rất nhanh
41: end procedure
```

12.3 Nhận xét

Độ phức tạp của thuật toán

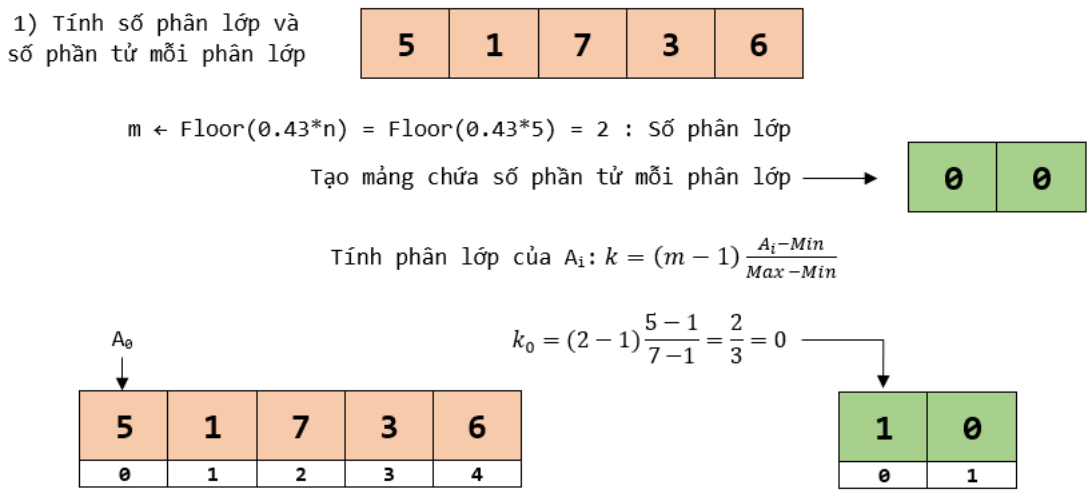
- Trường hợp tốt nhất: $O(n)$
- Trường hợp trung bình: $O(n)$
- Trường hợp tệ nhất: Khi giá trị các phần tử phân chỉ tập trung vào một số **Class** nhất định
  - Tìm **minValue,indexMaxValue** tốn chi phí: $O(n)$
  - Vòng lặp **for** thứ nhất chạy từ $i = 1 \to n - 1$ tốn chi phí: $O(n)$
  - Vòng lặp **for** thứ hai chạy từ $i = 1 \to m - 1 = 0.43n - 1$ tốn chi phí: $O(m) = O(0.45n) = O(n)$
  - Cả ba vòng lặp **while** đều dừng khi thực hiện được $n - 1$ lần lặp vì cả hai biến **NMove** và **J** đều duyệt qua các phần tử tối đa một lần
  - **insertionSort** sẽ thực hiện qua m *Class* và trong trường hợp xấu nhất thì *Class* đầu hoặc cuối có $n-1$ phần tử sắp xếp ngược (giảm dần). Tức là chi phí tệ nhất của **Insertion Sort** cho $n - 1$ ta đã chứng minh ở trên là: $O(n^2)$
  - Các phép gán, so sánh và hoán đổi vị trí tốn chi phí: $O(1)$

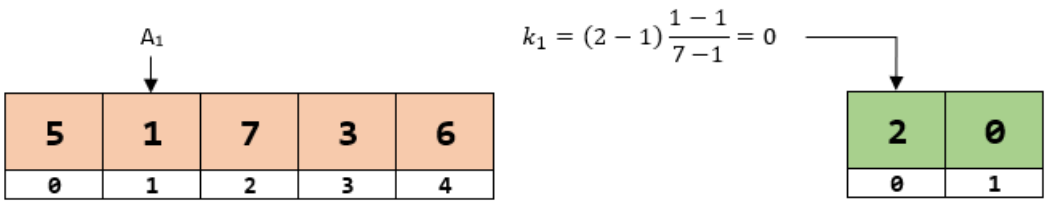
Thời gian thực thi **Flash Sort** trên một mảng gồm n phần tử:

$$T(n) = O(n) + O(m) + O(n) + O(n^2) = O(n^2)$$

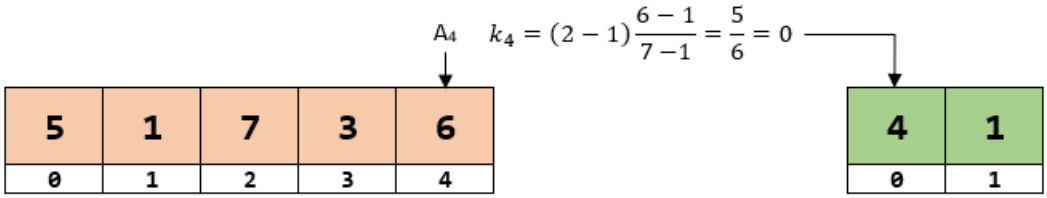
Vậy trong trường hợp tệ nhất thì thuật toán có độ phức tạp: $O(n^2)$

12.4 Ví dụ

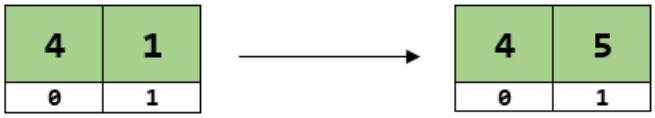


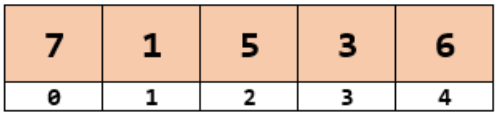


...



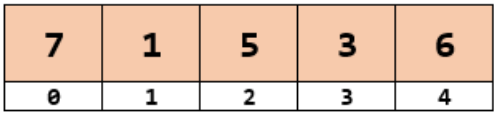
Cộng dồn mảng để tạo thành mảng chứa index+1 của phần tử cuối cùng mỗi phân lớp



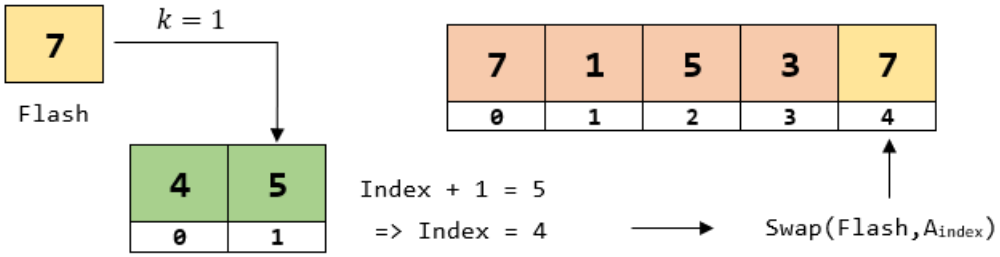


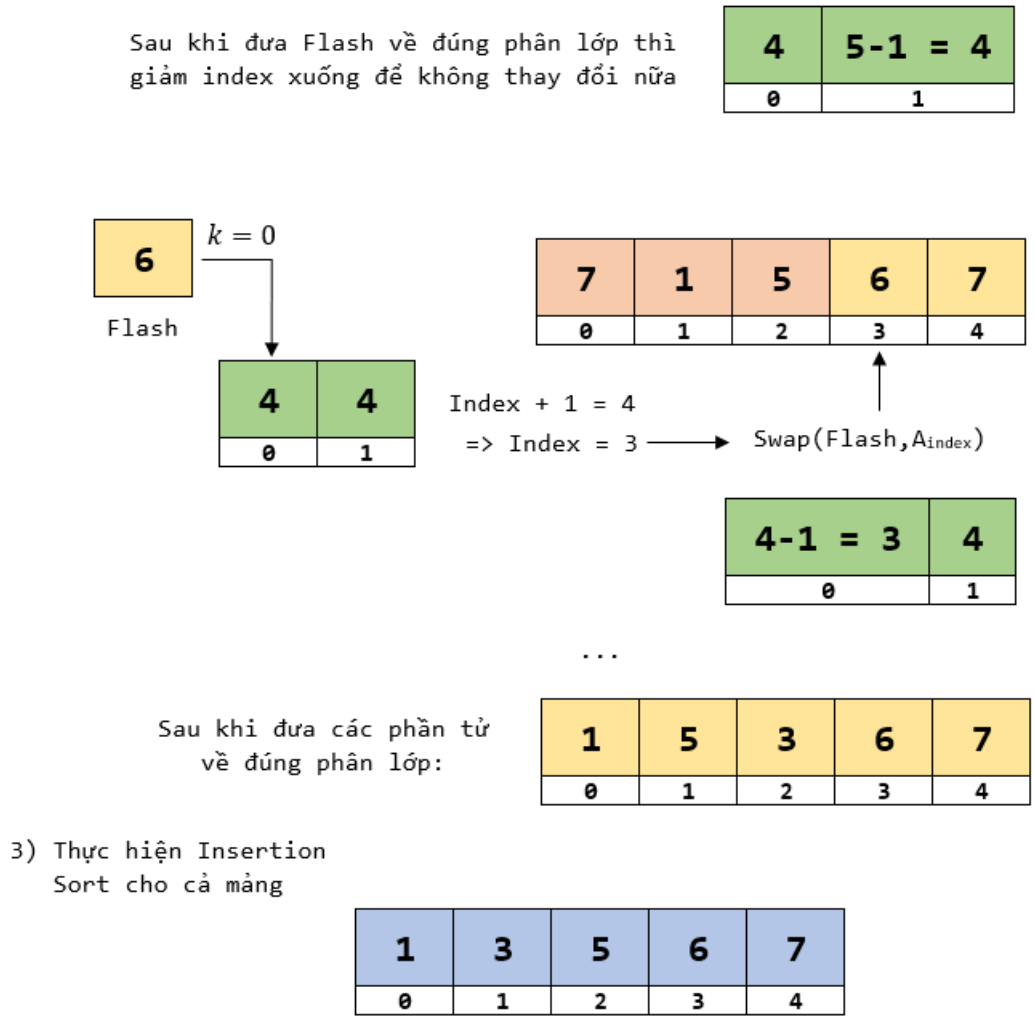
Hoán đổi vị trí A_{max} và A_0

2) Đi tìm phân lớp đúng của mỗi phần tử



Tính phân lớp của Flash: $k = (m-1)\frac{A_i - Min}{Max - Min}$





Hình 12: Flash Sort

Phần B

Cài đặt

Thời gian chạy thuật toán

Sau khi tiến hành xây dựng hàm **runTime** để tính thời gian chạy của 12 thuật toán sắp xếp với 3 trạng thái (Sorted: Đã sắp xếp (Tăng dần); Reversed: Sắp xếp ngược (Giảm dần); Random: Ngẫu nhiên) và 5 kích thước đầu vào là mảng gồm 1000,3000,10000,30000,100000 phần tử. Ta có kết quả thể hiện trong bảng dưới đây:

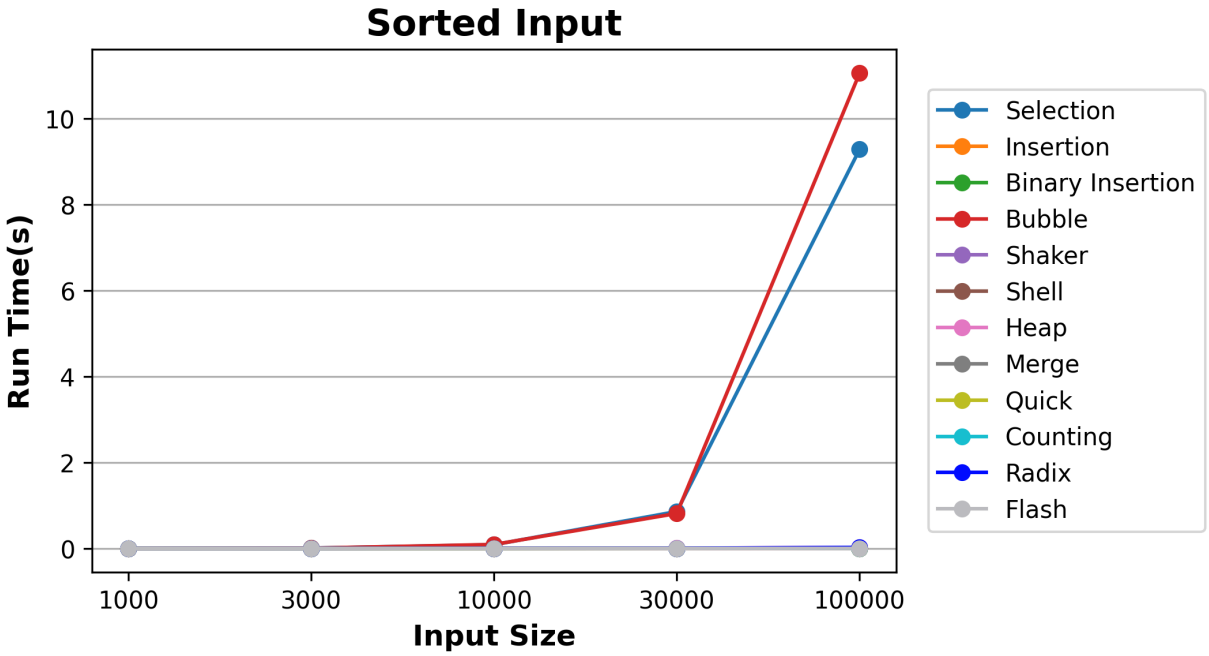
Input State	Input Size	Selection	Insertion	Binary Insertion	Bubble	Shaker	Shell	Heap	Merge	Quick	Counting	Radix	Flash
Sorted	1000	0.0013568	4.10E-06	3.01E-05	0.000885	2.10E-06	3.65E-05	0.000149	7.07E-05	5.05E-05	4.47E-05	0.000131	5.06E-05
Sorted	3000	0.0093891	9.40E-06	0.0001084	0.008253	5.70E-06	9.96E-05	0.000489	0.000207	0.000111	7.62E-05	0.000365	0.000155
Sorted	10000	0.0884122	3.21E-05	0.0004351	0.094238	1.90E-05	0.00042	0.002754	0.000846	0.000622	0.000359	0.002291	0.000426
Sorted	30000	0.860217	0.000102	0.0014746	0.815371	9.74E-05	0.001363	0.007115	0.00232	0.001299	0.000655	0.004109	0.001247
Sorted	100000	9.28012	0.000344	0.0060371	11.052	0.000376	0.005741	0.025492	0.00916	0.004669	0.002413	0.026353	0.004234
Reversed	1000	0.0014133	0.001163	0.0011027	0.005652	0.003635	5.36E-05	0.000223	9.73E-05	7.13E-05	3.05E-05	0.000143	5.37E-05
Reversed	3000	0.0104631	0.016424	0.0127745	0.037965	0.031725	0.000175	0.000504	0.00023	0.000131	0.000121	0.000693	0.000141
Reversed	10000	0.107535	0.203757	0.120404	0.365503	0.409643	0.000668	0.00286	0.001265	0.000471	0.000277	0.001633	0.000395
Reversed	30000	0.98066	1.14259	0.989905	3.37475	3.42085	0.004507	0.006849	0.004008	0.001519	0.00079	0.004953	0.001232
Reversed	100000	9.93076	11.8232	10.5604	34.7836	33.9543	0.007642	0.025324	0.010471	0.007762	0.00278	0.01968	0.005521
Random	1000	0.0011062	0.001003	0.0008509	0.003371	0.004014	0.000123	0.001758	0.000119	0.000121	2.69E-05	0.000225	5.36E-05
Random	3000	0.0098135	0.00587	0.0069128	0.061494	0.06208	0.000595	0.000988	0.000461	0.000526	0.000115	0.000417	0.000164
Random	10000	0.139523	0.087077	0.05949	0.371868	0.304614	0.001724	0.002546	0.002143	0.00137	0.0003	0.001588	0.000546
Random	30000	0.955702	0.531109	0.542169	3.4679	2.61299	0.008168	0.008407	0.006076	0.004134	0.000625	0.004628	0.001378
Random	100000	9.96942	6.25819	5.44544	38.4509	28.5554	0.02527	0.031222	0.02137	0.014565	0.002119	0.015098	0.00588

Bảng 1: Thời gian chạy các thuật toán

Biểu đồ

Dựa vào Bảng dữ liệu đã có ta tiến hành vẽ biểu đồ với ba trạng thái dữ liệu là:

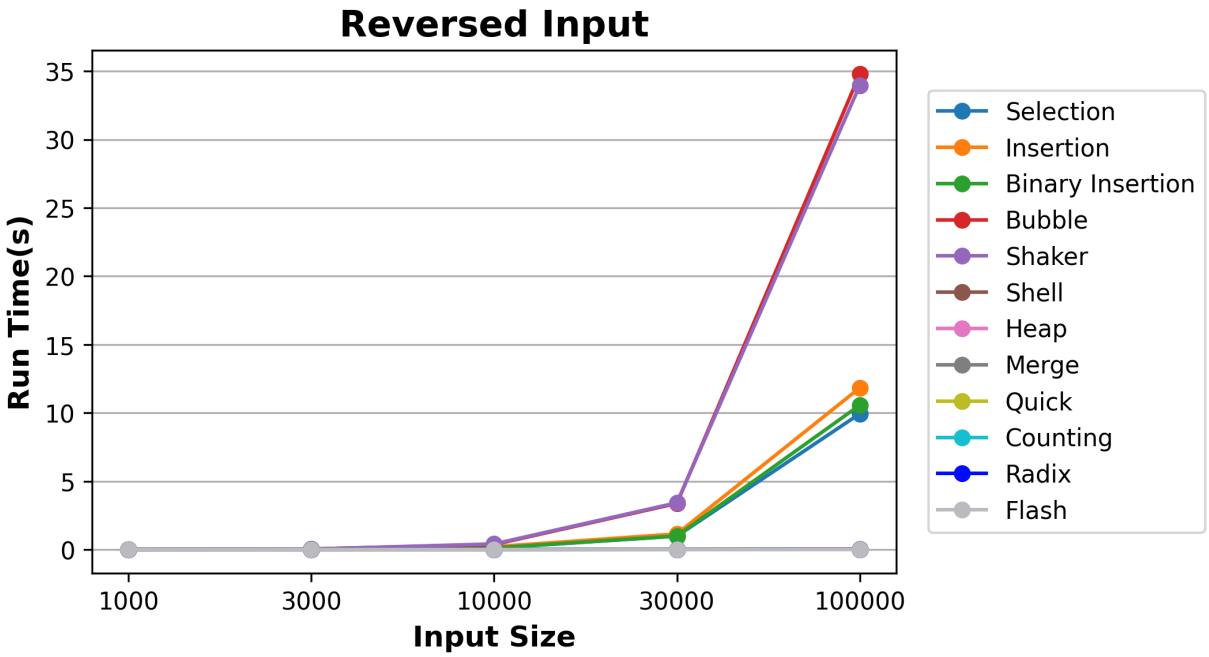
- **Sorted**
- **Reversed**
- **Random**



Biểu đồ 1: Biểu đồ thời gian chạy thuật toán sắp xếp đối với dữ liệu đã được sắp xếp

Nhận xét:

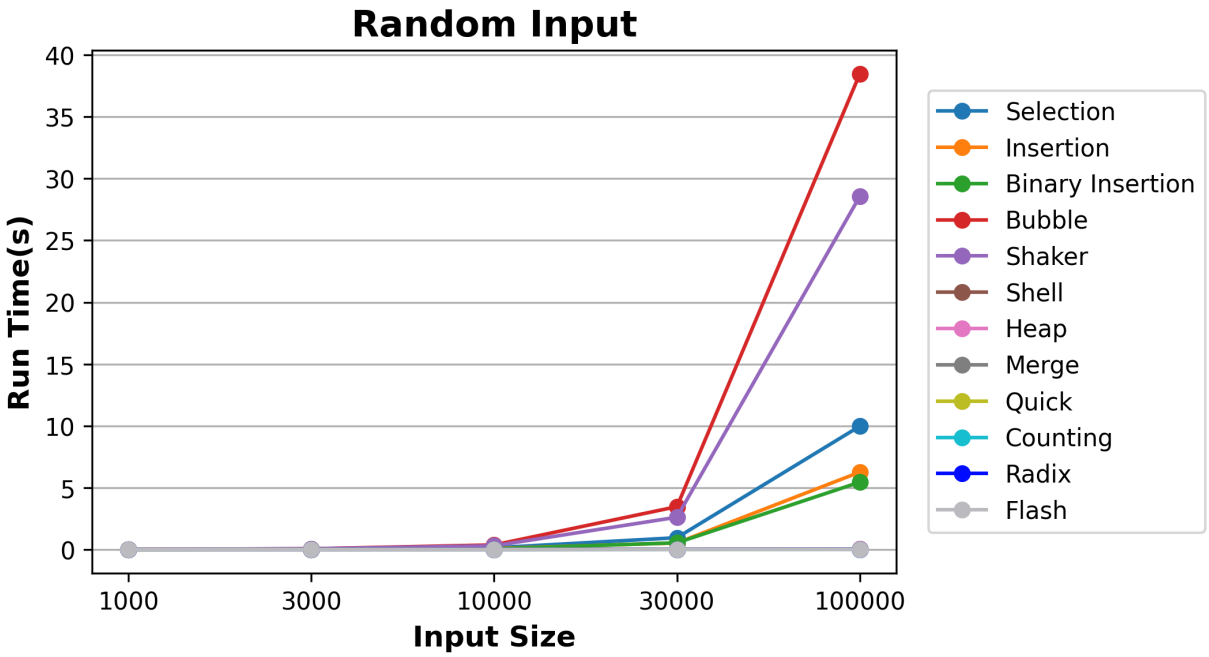
- Đối với dữ liệu đã được sắp xếp sẵn, về mặt thời gian có thể chia làm hai nhóm chính:
  - Nhóm chạy cực nhanh (dưới 0,03s với tất cả input có size khác nhau): Tất cả các thuật toán trừ hai thuật toán Selection Sort và Bubble Sort.
  - Nhóm chạy chậm: Selection Sort và Bubble Sort (có độ phức tạp $O(n^2)$) có thời gian chạy chậm hơn khoảng 9-10 lần so với các thuật toán còn lại.
- Trong input từ 1000 đến 10000 phần tử thì thời gian chạy của tất cả các thuật toán đều rất nhanh và không chênh lệch nhiều. Sự khác biệt chỉ xuất hiện đối với input từ 30000 phần tử trở lên, nhóm chạy chậm Selection Sort và Bubble Sort có màn thể hiện kém hơn hẳn với thời gian chạy lên đến 0,86s và khi input tăng gấp 3,3 lần thì thời gian chạy cũng tăng gấp 12,8 lần.
- Xét các thuật toán còn lại có thời gian chạy nhanh hơn: Insertion (dù có độ phức tạp $O(n^2)$ dẫn đầu khi chỉ mất tối đa 0,0003s, lí do là vì khi dữ liệu được sắp xếp tăng dần sẵn Insertion Sort chỉ tốn thời gian duyệt tất cả phần tử từ đầu đến cuối ($O(n)$) nhưng không tốn thời gian chèn phần tử vào vị trí đúng của nó, xếp ngay sau Insertion là Shaker Sort và Counting Sort. Đứng chót bảng là Radix Sort với 0,026s.



Biểu đồ 2: Biểu đồ thời gian chạy thuật toán sắp xếp đối với dữ liệu sắp xếp ngược (giảm dần)

Nhận xét:

- Đối với dữ liệu đã được sắp xếp ngược (giảm dần), về mặt thời gian có thể chia làm ba nhóm chính:
  - Nhóm chạy cực nhanh (dưới 0,03s): Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, Flash Sort.
  - Nhóm chạy trung bình (khoảng 12s): Selection Sort, Insertion Sort, Binary Insertion Sort.
  - Nhóm chạy chậm (khoảng 35s): Bubble Sort và Shaker Sort.
- Xét nhóm thuật toán chạy rất nhanh: Counting Sort dẫn đầu khi chỉ mất tối đa 0,002s, tiếp theo là Flash Sort và Quick Sort. Đứng vị trí cuối trong nhóm này là Heap Sort với 0,025s. Thời gian chạy của các thuật toán trong nhóm này cho dù là nhanh nhất hay chậm nhất đều ít hơn nhiều so với hai nhóm còn lại, cụ thể ít hơn 400 lần so với nhóm trung bình và 1167 lần so với nhóm chậm. Ngoài ra thời gian chạy của nhóm thuật toán này được duy trì khá ổn định và không có quá nhiều sự chênh lệch khi chạy input size 1000 và input size 100000 (gấp 100 lần).
- Xét nhóm thuật toán chạy trung bình: Selection Sort dẫn đầu với 9,93s, theo sau là Binary Insertion Sort và đứng vị trí cuối là Insertion Sort với 11,82s. Thời gian chạy của nhóm thuật toán này có độ chênh lệch tương đối lớn (chênh lệch từ 6-8 lần) và ngày càng tăng khi input size tăng.
- Xét nhóm thuật toán chạy chậm: Bubble Sort chạy chậm nhất với 34,78s, tiếp đó là Shaker Sort với 33,95s. Thời gian chạy của nhóm thuật toán này có độ chênh lệch rất lớn (chênh lệch từ 9-11 lần) và ngày càng tăng khi input size tăng (tăng nhanh hơn so với nhóm trung bình).



Biểu đồ 3: Biểu đồ thời gian chạy thuật toán sắp xếp đối với dữ liệu ngẫu nhiên

Nhận xét:

- Đối với dữ liệu ngẫu nhiên, thời gian chạy của các thuật toán có độ phân tán rõ ràng hơn so với dữ liệu đã sắp xếp tăng dần và giảm dần nên sẽ chia thành 4 nhóm để nhận xét được cụ thể hơn:
  - Nhóm chạy siêu nhanh (khoảng 0,005s): Flash Sort và Counting Sort.
  - Nhóm chạy khá nhanh (khoảng 0,03s): Shell Sort, Heap Sort, Merge Sort, Quick Sort và Radix Sort.
  - Nhóm chạy trung bình (khoảng 10s): Selection Sort, Insertion Sort và Binary Insertion Sort.
  - Nhóm chạy chậm (khoảng 40s): Bubble Sort và Shaker Sort.
- Xét nhóm thuật toán chạy siêu nhanh: Counting Sort đứng đầu chỉ với 0,002s, theo sát là Flash Sort với 0,005s. Thời gian chạy của nhóm thuật toán này được duy trì khá ổn định và không có quá nhiều sự chênh lệch giữa tất cả các input size (chênh lệch từ 2-3 lần so với input size trước nhỏ hơn).
- Xét nhóm thuật toán chạy khá nhanh: Quick Sort đứng đầu với 0,02s, tiếp theo là Radix Sort và Merge Sort. Đứng vị trí cuối trong nhóm này là Heap Sort với 0,03s. Thời gian chạy của nhóm thuật toán này cũng được duy trì khá ổn định và không có quá nhiều sự chênh lệch giữa các input size (chênh lệch từ 3-4 lần so với input size trước nhỏ hơn).
- Xét nhóm thuật toán chạy trung bình: Binary Insertion Sort đứng đầu với 5,44s, tiếp theo là Insertion Sort và ở vị trí cuối là Selection Sort với 9,96s. Từ nhóm này thời gian chạy đã bắt đầu biến động và có sự chênh lệch tương đối lớn giữa các input size (chênh lệch từ 7-10 lần so với input size trước nhỏ hơn).

- Xét nhóm thuật toán chạy chậm vẫn là hai cái tên quen thuộc: Bubble Sort chậm nhất với 38,45s và Shaker Sort với 28,55s. Ở nhóm này thời gian chạy biến động rõ rệt và chênh lệch nhiều so với các input size trước (chênh lệch từ 10-12 lần so với input size trước nhỏ hơn).

Lời kết và đánh giá kết quả

Trải nghiệm ba tuần thực hiện đồ án: Tìm hiểu và cài đặt các thuật toán tìm kiếm và sắp xếp đã giúp chúng em học hỏi được nhiều điều mới mẻ. Từ nguyên lý hoạt động của từng thuật toán tìm kiếm và sắp xếp cho đến cách áp dụng chúng vào giải quyết cho từng trường hợp khác nhau, đồng thời chúng em cũng đã có cơ hội rèn luyện thêm những kĩ năng mềm như nghiên cứu và làm việc nhóm.

Trong quá trình làm việc, do thời gian thực hiên và chỉnh sửa không nhiều cũng như trình độ của nhóm còn có hạn, nên dù chúng em đã rất cố gắng nhưng khó tránh khỏi sai sót xuất hiện. Vì vậy chúng em rất mong nhận được những góp ý và chia sẻ từ quý thầy cô để nhóm có thể cải thiện bài báo cáo, rút kinh nghiệm và thực hiện các đồ án sau này tốt hơn.

Vì nhận thấy những nỗ lực, cố gắng của chính bản thân trong việc tìm hiểu và hoàn thành đồ án và xét thấy đồ án của nhóm đã đáp ứng đầy đủ các yêu cầu của thầy đề ra trong đề bài nên chúng em xin phép tự đánh giá kết quả bài nộp là: **10 điểm**

Chúng em xin chân thành cảm ơn!

Ký tên

Nguyễn Nhật Minh Thư
Nguyễn Đặng Anh Thư

Tài liệu tham khảo

[1] Donald Knuth, *"The Art of Computer Programming"*, Third Edition. Addison–Wesley, 1997. ISBN: 0-201-89685-0

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *"Introduction to Algorithms"*, Third Edition. The MIT Press, 2009. ISBN: 978-0-262-03384-8.

[3] Bùi Tiến Lên, *"DSA"*. Truy cập: 05/10/2022 tại `https://drive.google.com/drive/folders/0B3x--lrLA0r2a2dhakVwbE5FMUU?resourcekey=0-k4aLP77yxfPIRHwNtWSwug`.

[4] Thomas Cormen and Devin Balkcom, *"Insertion Sort"*. Truy cập: 07/10/2022 tại `https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort`.

[5] Milos Simic, *"Binary insertion Sort"*. Truy cập: 09/10/2022 tại `https://www.baeldung.com/cs/binary-insertion-sort`.

[6] Thomas Cormen and Devin Balkcom, *"Overview of merge sort"*. Truy cập: 10/10/2022 tại `https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort`.

[7] Thomas Cormen and Devin Balkcom, *"Overview of quicksort"*. Truy cập: 10/10/2022 tại `https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort`.

[8] Karl-Dietrich Neubert (1998-2002), *"A Collection of FlashSort Implementations"*. Truy cập: 13/10/2022 tại `http://ww.neubert.net/Flacodes/FLACodes.html`.

[9] GeeksforGeeks, *"Shell Sort"*. Truy cập: 13/10/2022 tại `https://www.geeksforgeeks.org/shellsort/`.

[10] GeeksforGeeks, *"Overview of quicksort"*. Truy cập: 13/10/2022 tại `https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort`.

[11] GeeksforGeeks, *"Counting Sort"*. Truy cập: 13/10/2022 tại `https://www.geeksforgeeks.org/counting-sort/`.

[12] Shreya Shah, *"Time Space Complexity of Counting Sort"*. Truy cập: 14/10/2022 tại `https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/#:~:text=The%20time%20complexity%20of%20counting%20sort%20algorithm%20is%20O(n,of%20values%20to%20be%20sorted.`.

[13] NickHounsome, *"Date and time utilities"*. Truy cập: 19/10/2022 tại `https://en.cppreference.com/w/cpp/chronol`.

[14] Jake VanderPlas, *"Simple Line Plots"*. Truy cập: 20/10/2022 tại `https://jakevdp.github.io/PythonDataScienceHandbook/04.01-simple-line-plots.html`.

[15] Malte Skarupke, *"I Wrote a Faster Sorting Algorithm"*. Truy cập: 20/10/2022 tại `https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/`.