ĐẠI HỌC QUỐC GIA TP.HCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐỒ ÁN

HỌC PHẦN

THỰC HÀNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Ngày 24 tháng 10 năm 2022

MỤC LỤC

THÔNG TIN BÀI BÁO CÁO

Đề tài đồ án: Tìm hiểu về Search và Sort.

Giảng viên hướng dẫn: Nguyễn Bảo Long

Các thành viên trong nhóm:

1. Lê Võ Bảo Trâm (MSSV: 21280052)
2. Nguyễn Lưu Phương Ngọc Lam (MSSV: 21280096)

LỜI CẢM ƠN

Trước tiên, chúng em xin được gửi một lời cảm ơn sâu sắc và lòng biết ơn chân thành nhất dành đến cho thầy Nguyễn Bảo Long vì đã chuẩn bị và đầu tư những bài giảng tâm huyết, nhằm truyền đạt những kiến thức lập trình bổ ích mà không kém phần thú vị trong suốt thời gian thực học môn **Thực hành Cấu trúc dữ liệu và Giải thuật** vừa qua.

Trong quá trình nghiên cứu về các thuật toán tìm kiếm và sắp xếp, vì lí do thời gian và trình độ còn nhiều hạn chế, nhóm chúng em sẽ không thể tránh khỏi những thiếu sót. Như Socrates từng nói *"Tôi chỉ biết một điều là tôi không biết gì cả"* . Trên tinh thần cầu thị và sẵn sàng tiếp thu, ghi nhận mọi đóng góp để kịp thời sửa đổi, nhóm chúng em kính mong nhận được sự chỉ bảo và ý kiến đến từ quý thầy cô để có thể hoàn thiện sản phẩm của mình hơn.

Chúng em xin chân thành cảm ơn!

PHÂN CÔNG CÔNG VIỆC

Nội dung	Công việc	Yêu cầu	Người thực hiện	Tiến độ
Tìm hiểu và cài đặt thuật toán	➢ Tổng hợp các thông tin thuật toán lưu vào Notion. ➢ Nêu ý tưởng chung. ➢ Viết mã giả và chạy demo code minh họa cho mã giả đó. ➢ Cài đặt thuật toán (C++). ➢ Nhận xét và chứng minh độ phức tạp của thuật toán.	Đảm nhận các thuật toán: Linear Search, Selection Sort, Shaker Sort, Shell Sort, Merge Sort, Flash Sort, Heap Sort.	Bảo Trâm	04-12/10/2022
		Đảm nhận các thuật toán: Binary Search, Insertion Sort, Binary Insertion Sort, Bubble Sort, Quick Sort, Counting Sort, Radix Sort.	Ngọc Lam	04-12/10/2022
Kiểm thử và ghi nhận kết quả	➢ Kiểm tra kết quả của thuật toán đã cài đặt. ➢ Xử lý file	➢ Bổ sung các yếu tố còn thiếu theo yêu cầu đồ án. ➢ Xuất các file.txt lưu kết quả sau khi sắp xếp và xuất file Runtime,csv ghi nhận thời gian của từng thuật toán.	Bảo Trâm	13-15/10/2022

Xử lý và biểu diễn dữ liệu	➢ Vẽ biểu đồ theo từng tình trạng dữ liệu. ➢ Nhận xét biểu đồ.	➢ Biểu đồ đảm bảo các yêu cầu: trục tung thể hiện thời gian chạy, trục hoành thể hiện kích thước bộ dữ liệu và màu của đường biểu diễn tên thuật toán. ➢ Nhận xét tính	Ngọc Lam, Bảo Trâm	15-16/10/2022
Báo cáo	➢ Thống nhất nội dung báo cáo. ➢ Xây dựng dàn ý bản báo cáo.	Đảm bảo: trang bìa, mục lục, phân công công việc, tỷ lệ hoàn thành, tự đánh giá bài nộp trên thang 10đ và nội dung bài tập được yêu cầu.	Ngọc Lam	13-15/10/2022
	Thực hiện viết báo cáo bằng Latex.	Phối hợp viết Latex trên nền tảng Overleaf.	Ngọc Lam, Bảo Trâm	18-22/10/2022
Tổng kết	Kiểm tra nội dung và đảm bảo bài nộp.	➢ Kiểm tra nội dung báo cáo. ➢ Tổng duyệt thư mục nộp	Ngọc Lam, Bảo Trâm	19-22/10/2022

Họ và tên	MSSV	Đánh giá điểm
Lê Võ Bảo Trâm	21280052	10
Nguyễn Lưu Phương Ngọc Lam	21280096	10

Cách đánh giá dựa vào thái độ làm việc tích cực, hoàn thành deadline đúng thời gian đã quy định.
Tỷ lệ hoàn thành 95%

CHƯƠNG 1

THUẬT TOÁN TÌM KIẾM

1. Linear Search - Tìm kiếm tuyến tính

1.1. Ý tưởng

Đối chiếu giá trị cần tìm với giá trị của các phần tử trong mảng bằng cách duyệt lần lượt từ đầu đến cuối mảng. Xảy ra hai trường hợp: hoặc giá trị cần tìm tồn tại trong khoảng [0,n] thì trả về vị trí của nó, hoặc giá trị x không tồn tại trong mảng thì trả về -1.

1.2. Mã giả

```
procedure linear_search (array, value)
  for each item in the array
     if match item == value
        return the location of item
     end if
  end for
  return -1
end procedure
```

1.3. Nhận xét

Giả sử mảng a có n phần tử, tìm phần tử có giá trị x trong mảng a:

- Trường hợp tốt nhất: Giá trị cần tìm nằm ở đầu mảng Chỉ cần duy nhất 1 phép so sánh (a[0] ? x) $\Rightarrow$ **O(1)**

Trong cả hai trường hợp nêu trên đều phải duyệt mảng qua một vòng lặp, số lượng so sánh tối đa bằng n phép so sánh.

Do đó, độ phức tạp thời gian **trường hợp xấu nhất** của tìm kiếm tuyến tính là **O(n)**.

- Trường hợp Trung bình: gồm 2 khả năng có thể xảy ra:

+ Phần tử cần tìm nằm trong khoảng 0 đến n-1.

Số trường hợp có thể xảy ra: **n** (1)

Chi phí so sánh = Tổng các khả năng có thể tìm thấy x trong mảng ở vị trí bất kì:

$$\sum_{i=1}^{n} f(x) = 1+2+3+...+n = \frac{n(n+1)}{2}(*)$$

+ Phần tử cần tìm **không nằm trong mảng**.

Số trường hợp có thể xảy ra: **chỉ 1** (2)

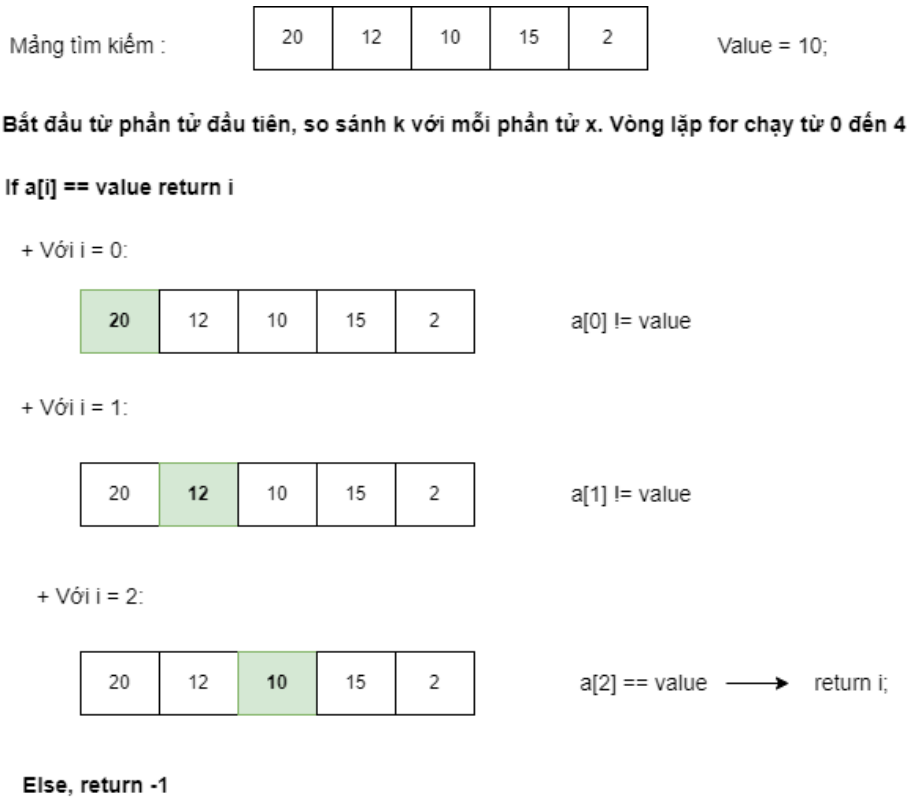
Chi phí so sánh = Một vòng lặp đối chiếu giá trị của x với từng phần tử trong mảng = O(n).

Từ (1) + (2) và (*), tổng chi phí dành cho trường hợp trung bình được tính bởi:

$$n+1 = \frac{n(n+1)}{2} \Leftrightarrow n+1 = n^2+3n \Leftrightarrow \frac{n^2+3n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

Vì vậy, độ phức tạp thời gian trường hợp trung bình của tìm kiếm tuyến tính là O(n).

1.4. Minh hoạ thuật toán



2. Binary Search

2.1. Ý tưởng

Làm sao để giảm bớt các so sánh không cần thiết khi mảng đã được sắp xếp theo một thứ tự xác định? Giải pháp của vấn đề này là một thuật toán tìm kiếm mới.

Thuật toán này hoạt động bằng cách chọn phần tử trung vị (phần tử ở vị trí chính giữa) để giảm bớt các phép so sánh (thừa).

Trong trường hợp phần tử cần tìm (x) khác phần tử trung vị (a[mid]) ta sẽ tiếp tục xét các trường hợp sau đây:

+ Nếu x < a[mid], ta sẽ tiếp tục tìm kiếm ở vùng nửa nhỏ hơn của mảng.

+ Ngược lại ta sẽ tìm kiếm trên nửa lớn hơn.

Kết thúc quá trình tìm kiếm mà không có phần tử nào bằng phần tử cần tìm, ta sẽ kết luận phần tử cần tìm không có trong dãy bằng lệnh trả về -1.

Có hai cách cài đặt tìm kiếm nhị phân: *lặp hoặc đệ quy*. Ở đây chúng ta tìm hiểu về phương thức lặp trước tiên, sau đó đến phần sắp xếp mảng bằng Binary Insertion sẽ tiếp cận phương pháp đệ quy.

2.2. Mã giả

```
Procedure binary_search(A, left, right, x)
       while left <= right
         mid = (left + right)/2
            if a[mid] == value
               return the midden location
            else if a[mid] < x
               left = mid + 1
            else right = mid - 1
            end if
       end while
    return -1
end procedure
```

2.3. Nhận xét

Giả sử mảng a có n phần tử đã được sắp xếp.

Trường hợp tốt nhất: x nằm đúng ngay phần tử trung vị của mảng a[mid]

Chúng ta chỉ mất chi phí duy nhất cho 1 phép so sánh. $\Rightarrow O(1)$

Trường hợp tệ nhất: xảy ra khi x nằm ở vị trí đầu tiên hoặc cuối cùng của mảng

Độ phức tạp: $O(n * log_n)$

Trường hợp trung bình: có 2 khả năng xảy ra như sau:

1. Giá trị x có tồn tại trong mảng

Số trường hợp có thể xảy ra: n

Tổng quát hoá các trường hợp với length(a)=n:

2. Tìm kiếm theo phần tử trung vị được xác định ở lần 1:

$$length(a) = \frac{n}{2}$$

+ Tìm kiếm theo phần tử trung vị được xác định ở lần 2:

$$length(a) = \frac{\frac{n}{2}}{2} = \frac{n}{4}$$

+ Tìm kiếm theo phần tử trung vị được xác định ở lần 3:

$$length(a) = \frac{\frac{n}{4}}{2} = \frac{n}{8}$$

+ Tìm kiếm theo phần tử trung vị được xác định ở lần k:

$$length(a) = \frac{n}{2^k}$$

Sau k lần phân chia mảng thì ta được:

$$\begin{aligned} & length(a) = 1 \\ & \Leftrightarrow \frac{n}{2^k} = 1 \Leftrightarrow n = 2^k \\ & \Leftrightarrow log_2(n) = log_2(2^k) \\ & \Leftrightarrow log_2(n) = k * log_2 2 = k * 1 \\ & \Leftrightarrow k = log_2 n \end{aligned}$$

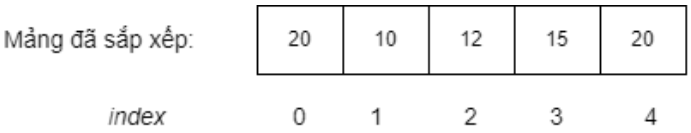
+ Giá trị x không tồn tại trong mảng

Số trường hợp có thể xảy ra: **chỉ 1**

Chi phí so sánh = Một vòng lặp đối chiếu giá trị của x với từng phần tử trong mảng = O(n).

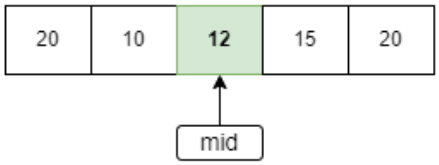
Do đó, độ phức tạp của Binary Search là: $O(\log_n)$

2.4. Minh hoạ thuật toán



- left = 0;
- right = 4

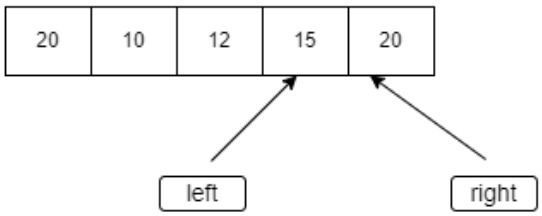
⟶ left < right (thỏa điều kiện WHILE; **mid = 2**



if a[mid] == value return mid. Else so sánh các phần tử khác.

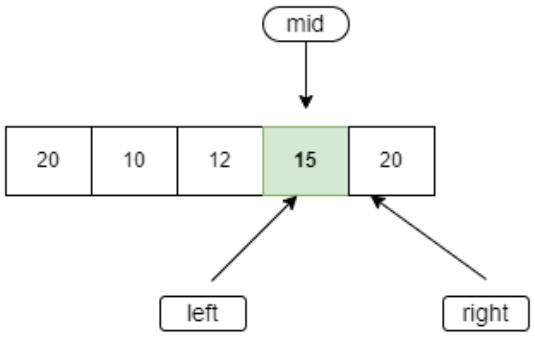
if value > a[mid], thì so sánh value với các phần từ chính giữa của các phần tử nằm bên phải mid.
Với left = mid +1, right = n.

else so sánh value với các phần tử chính giữa của các phần từ nằm bên trái của mid. Với left = 0, right = mid - 1.



- left = 3;
- right = 4;

⟶ left < right(thỏa mãn điều kiện WHILE); **mid = 3**;



a[mid] == value ⟶ **return mid;**

CHƯƠNG 2

THUẬT TOÁN SẮP XẾP

3. Selection Sort

3.1. Ý tưởng

Xét phần tử đầu tiên của dãy. Tìm phần tử nhỏ nhất trong các phần tử còn lại. Hoán đổi phần tử đầu tiên với phần tử nhỏ nhất này, ta được phần tử đầu tiên có vị trí đúng. Bỏ qua phần tử vừa được xét, tiếp tục xét đến phần tử kế tiếp và thực hiện đến hết dãy.

3.2. Mã giả

```
procedure selectionSort:
    for i from 0 to n-1:
    MinValue = i
        For j from i+1 to n:
                if (a[j] < a[MinValue]) then
                   MinValue = j;
                End if
        End For
        If (MinValue != i) then
             swap( a[i], a[MinValue]);
        End If
    END for
end procedure.
```

3.3. Nhận xét

Trường hợp tốt nhất (mảng đã sắp xếp):

Chỉ dùng duy nhất 1 vòng lặp đi đối chiếu giá trị giữa các phần tử mảng $\Rightarrow$ **O(n)**

Trường hợp trung bình (mảng ngẫu nhiên): $O(n^2)$

Trường hợp này sẽ được chứng minh tương tự như dưới đây.

Trường hợp tệ nhất: Mảng sắp xếp giảm dần

+ Vòng lặp toàn cục chiếm chi phí:

$$0+1+2+...+(n-1)=\sum_{i=1}^{n} f(x)=\frac{n(n-1)}{2}(*)$$

+ Trong lúc lặp toàn mảng ta thực hiện liên tiếp 3 nhiệm vụ với cùng chi phí: tìm min, so sánh min với i rồi hoán đổi min. Do đó để gọi swap ta cần chi phí:

$$3*(n-1)$$

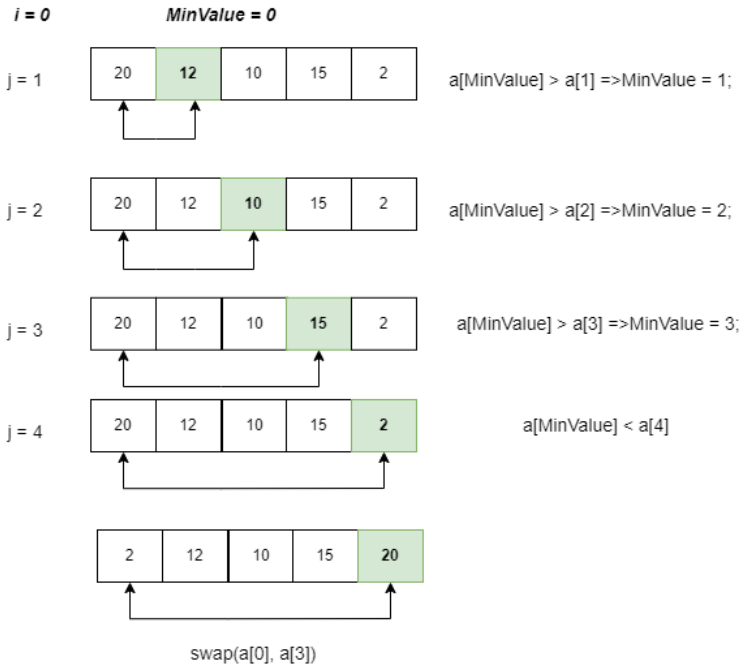
Vậy cần có chi phí tổng như sau:

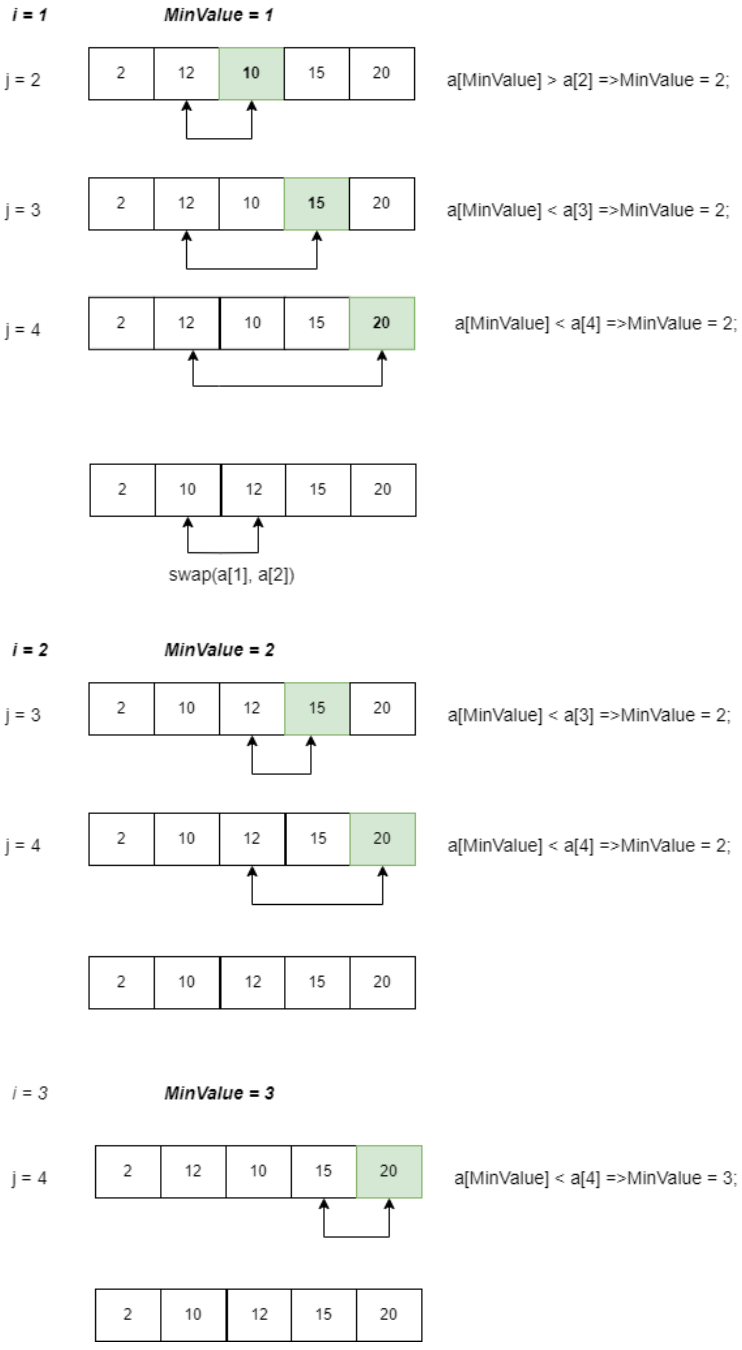
$$\frac{n*(n-1)}{2}+3(n-1)=\frac{n^2}{2}+\frac{5n}{2}-3=n^2+5n-6 \tag{2.1}$$

Vậy độ phức tạp của Sắp xếp chọn là: $O(n^2)$

- Độ ổn định: không.

3.4. Minh hoạ thuật toán





4. Insertion Sort

4.1. Ý tưởng

Lấy ra giá trị của từng phần tử để tìm vị trí thích hợp của nó trong mảng. Vị trí chính xác của phần tử đó được xác định khi đứng trước phần tử lớn hơn và sau phần tử nhỏ hơn hoặc bằng nó.

4.2. Mã giả

```
for i = 1 to length(A)
        x = A[i]
        j = i - 1
    while j >= 0 and A[j] > x
        A[j+1] = A[j]
        j = j - 1
    end while
A[j+1] = x
end for
```

4.3. Nhận xét

Trường hợp tốt nhất (mảng đã sắp xếp tăng dần):
Chỉ dùng duy nhất 1 vòng lặp đi đối chiếu giá trị giữa các phần tử mảng ⇒O(n)

Trường hợp trung bình (mảng ngẫu nhiên): $O(n^2)$
Trường hợp này sẽ được chứng minh tương tự như dưới đây.

Trường hợp tệ nhất (mảng sắp xếp giảm dần)
Chi phí cho một lần duyệt mảng:

$$0+1+2+...+n-1=\sum_{0}^{n-1} f(x)=\frac{n(n-1)}{2} \tag{2.2}$$

Chi phí hoán vị trong một mảng:
Với $n!=n*(n-1)!$ ta chỉ hoán vị hai phần tử nên được:

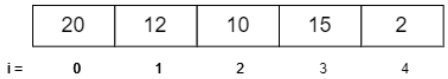
$$2*(n-1) \tag{2.3}$$

Từ (2.2) và (2.3) ta được: $\frac{n(n-1)}{2}+2*(n-1)=n^2+n-2$
Vì vậy, độ phức tạp thời gian trường hợp tệ nhất của thuật toán sắp xếp chèn là: $O(n^2)$
- Độ ổn định: có.

4.4. Minh hoạ thuật toán

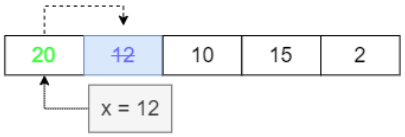
Insertion Sort

Mảng ban đầu:

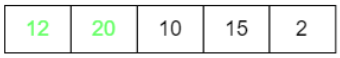


Duyệt từ vị trí i = 1 đến hết mảng i = (n-1), tức là vòng lặp sẽ đi qua các giá trị 12, 10, 15 và 2.
Lưu trữ a[i] bằng biến x. Coi phần tử đầu tiên là một mảng con được sắp xếp
và phần còn lại là một mảng con chưa được sắp xếp.

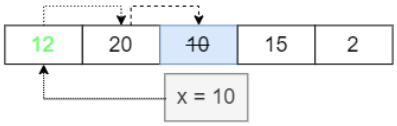
Tại i=1, ta có a[1] < a[0] (vì 12 < 20), nên ta chèn 12 lên trước 20 bằng cách:
đưa 20 lên vị trí thứ i và đẩy 12 về vị trí thứ i - 1.



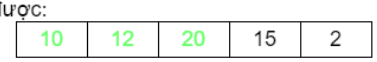
Ta được:



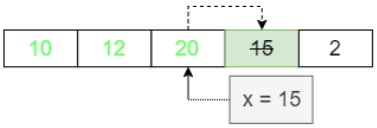
Tại i=2, ta có a[2] < a[1] và a[2] < a[0] (tức là chúng ta không tìm thấy bất kỳ phần tử nào
nhỏ hơn hoặc bằng 10, chúng ta sẽ chèn nó vào đầu mảng được sắp xếp),
ta thực hiện chèn 20 rồi 12 lên 1 vị trí so với ban đầu và đưa 10 về i = 0:



Ta được:

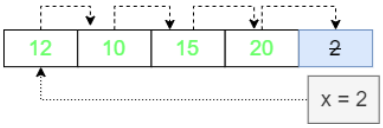


Tại i=3, ta có a[3] < a[2] (vì 15 < 20), xử lý tương tự, ta được:

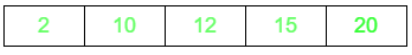


Ta được:

Tại i=4, vì sau tất cả các phép so sánh không tìm thấy phần tử nhỏ hơn hoặc bằng a[4], cho nên
a[4] là phần tử bé nhất nên cần chèn vào đầu mảng và đẩy tất cả phần tử còn lại lên một đơn vị.



Cuối cùng ta được mảng đã sắp xếp như sau:



5. Binary Insertion Sort

5.1. Ý tưởng

Là cải tiến của thuật toán sắp xếp chèn. Nhưng thay vì sử dụng tìm kiếm tuyến tính để tìm vị trí nơi phần tử sẽ được chèn vào, chúng ta tận dụng thứ tự đã được sắp xếp trong đoạn a[0] đến a[i-1] bằng thuật toán tìm kiếm nhị phân, qua đó làm giảm số lượng phép so sánh trong mỗi lần duyệt mảng.

5.2. Mã giả

```
procedure binarySearch(Array, x, left, right)
    if left > right
   return False
 else
   mid = (left + right) / 2
    if x == arr[mid]
           return mid
      else if x > arr[mid]
          return binarySearch(arr, x, mid + 1, right)
      else
       return binarySearch(arr, x, left, mid - 1)
end procedure

procedure binaryInsertionSort(Array)
    for i = 1 to length(Array) do:
        key = Array[i]
        pos = binarySearch(Array, key, 0, i-1)
        j = i
        while j >= pos
            Array[j] = Array[j-1]
            j = j-1
        Array[pos] = key
    end for
end procedure
```

5.3. Nhận xét

Đây là một phiên bản tối ưu hơn của sắp xếp chèn. Với số lượng các phép so sánh giảm từ n^2 về $n * logn$

Độ phức tạp thời gian của thuật toán:

- Trường hợp tốt nhất: một vòng lặp kiểm tra vị trí đúng của các giá trị **O(n)**

- Trường hợp trung bình: O($n * log_n$)

- Trường hợp xấu nhất: O($n * log_n$)

Với việc sử dụng 1 vòng lặp để kiểm tra vị trí đúng của các giá trị ta tiêu tốn **O(n)** chi phí. Song song với đó, hàm Binary Search được gọi xử lý mất chi phí: **O(logn)**

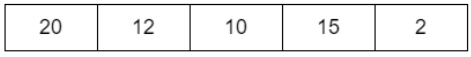
Do đó ta được tổng chi phí để sắp xếp chèn nhị phân là: **O(n*logn)**

- Độ ổn định: có.

5.4. Minh hoạ thuật toán

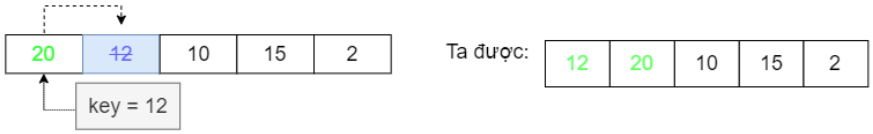
Binary Insertion Sort

Mảng ban đầu:

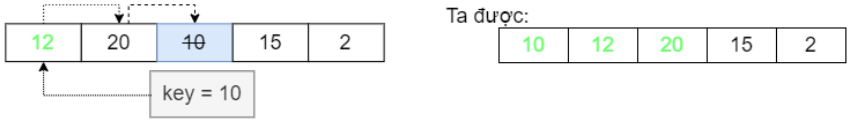


Duyệt từ vị trí i = 1 đến hết mảng i = (n-1). Mỗi lần lặp ta cần lưu trữ giá trị của a[i] bằng biến key. Giả định rằng phần tử đầu tiên là một mảng con đã được sắp xếp và phần còn lại là một mảng con chưa được sắp xếp.

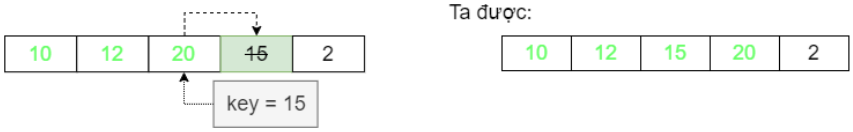
Tại i=1, chúng ta sử dụng **tìm kiếm nhị phân** để tìm phần tử lớn hơn a[i] ở vùng đã sắp xếp, dễ thấy 20>12 nên ta chèn 12 lên trước 20 bằng cách: đưa 20 lên vị trí thứ i và đẩy 12 về vị trí thứ i - 1.



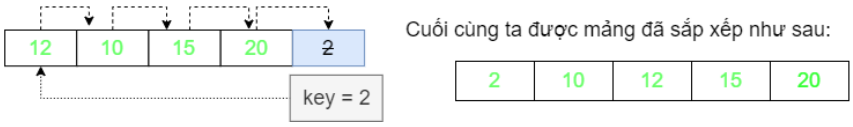
Tại i=2, gọi hàm tìm kiếm nhị phân để tìm phần tử chỉ lớn hơn 10 trong phần được sắp xếp. Trong trường hợp này, phần tử cần tìm là 12. Sau đó, ta thực hiện chèn 20 rồi 12 lên 1 vị trí so với ban đầu và đưa 10 về i = 0.



Tại i=3, xử lý tương tự:



Tại i=4, thông qua thuật toán tìm kiếm nhị phân không tìm thấy phần tử nào nhỏ hơn hoặc bằng a[4], cho nên cần chèn a[4] vào đầu mảng và đẩy tất cả phần tử còn lại lên một đơn vị.



6. Bubble Sort

6.1. Ý tưởng

Thuật toán sắp xếp Bubble Sort thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự(số sau bé hơn số trước với trường hợp sắp xếp tăng dần) cho đến khi dãy số được sắp xếp.

6.2. Mã giả

```
proceduce bubble_sort(a)
  for i = 0 to length(a)-1 do:
```

```
            for j = length(a) downto i+1 do:
                    if a[i]>a[j]
                            swap(a[i],a[j]);
                    end if
             end for
             end while
        end proceduce
```

6.3. Nhận xét

Trường hợp tốt nhất (mảng đã được sắp xếp): O(n) Trường hợp xấu nhất = Trường hợp trung bình = $O(n^2)$ Chi phí cho việc duyệt mảng từ 0 đến n-1 là:

$$0+1+2+...+n-1=\sum_{0}^{n-1} f(x)=\frac{n(n-1)}{2} \tag{2.4}$$

Hay

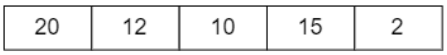
$$f(x)=\frac{1}{2}(n^2-n)\approx O(n^2) \tag{2.5}$$

- Độ ổn định: có.

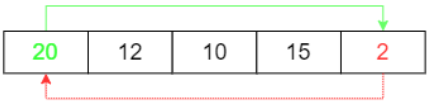
6.4. Minh hoạ thuật toán

Bubble Sort

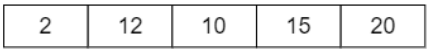
Mảng ban đầu:



Duyệt từ đầu mảng đi tới và cuối mảng đi về để so sánh hai phần tử a[i] và a[j].
Lần 1: Dễ thấy a[0] > a[4], đó đây là một nghịch thế cần xử lý.
Vì vậy, chúng ta cần dịch chuyển chúng về vị trí đúng và tiếp tục duyệt mảng.

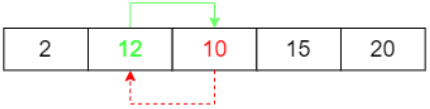


Ta có mảng:



Lần 2: Xét thấy a[1] < a[3], không thoả điều kiện nên giữ nguyên mảng và tiếp tục lặp để xét.

Lần 3: Xét thấy a[1] > a[2], thực hiện phép hoán vị tương tự ta được:



Ta có mảng:

Mảng đã được sắp xếp nhưng thuật toán vẫn chạy để kiểm tra tính đúng đắn về vị trí của các giá trị thông qua điều kiện đặt ra.

7. Merge Sort

7.1. Ý tưởng

Merge sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Hàm merge() được sử dụng để gộp hai nửa mảng. Hàm merge(arr, l, m, r) là tiến trình quan trọng nhất sẽ gộp hai nửa mảng thành 1 mảng sắp xếp, các nửa mảng là arr[l...m] và arr[m+1...r] sau khi gộp sẽ thành một mảng duy nhất đã sắp.

7.2. Mã giả

```
function Merge (int a[], int left, int mid, int right)
   int i = left;
   int j = mid + 1;
   int k = 0;
   int n = right - left + 1;
   int *B = new int[n];
   while ((i < mid + 1) && (j < right + 1))
       if (a[i] < a[j])
       {
           B[k] = a[i];
           k++; i++;
       }
       else
       {
           B[k] =a[j];
           k++; j++;
       }
       end if
   end while
   while (i <mid+1)
       B[k] = a[i];
       k++; i++;
   end while
   while (j < right+1)
       B[k] = a[j];
       k++; j++;
   end while
   i = left;
   for(k = 0; k <n; k++)
       a[i] = B[k];
       i++;
   end for
```

```
end function.

procedure MergeSort(arr, left, right):
    if left < right
        return mid = (left+right)/2
    MergeSort(arr, left, mid)
    MergeSort(arr, mid+1, right)
    Merge(arr, left, mid, right)
end procedure
```

7.3. Nhận xét

Gọi T(n) là thời gian thực hiện MergeSort một danh sách n phần tử thì $\frac{n}{2}$ là thời gian thực hiện MergeSort một danh sách $T\frac{n}{2}$ phần tử.

Khi n = 1 phần tử thì chương trình chỉ làm một việc duy nhất là return(L), việc này tốn O(1) = C1 thời gian.

Trong trường hợp n > 1, chương trình phải thực hiện gọi đệ quy MergeSort hai lần cho L1 và L2 với độ dài $\frac{n}{2}$ do đó thời gian để gọi hai lần đệ quy này là $2T\frac{n}{2}$. Ngoài ra còn phải tốn thời gian cho việc chia danh sách L thành hai nửa bằng nhau và trộn hai danh sách kết quả (Merge). Người ta xác định được thời gian để chia danh sách và Merge là $O(n) = C2n$. Vậy ta có phương trình đệ quy như sau:

$$T(n)\begin{cases} C1 & n=1 \\ 2T\frac{n}{2} + C2n & n>1 \end{cases} \tag{2.6}$$

$T(n) = 2T\frac{n}{2} + C2n = 2[2T\frac{n}{2} + C2\frac{n}{2}] + C2n = 4T\frac{n}{4} + 2.C2n = 4[2T\frac{n}{8} + C2\frac{n}{4}] + C2n = 2^iT\frac{n}{2^i} + i.C2n$

Giả sử $n = 2^k$ và khi kết thúc ta có $i = k \rightarrow$T(n) = $2^kT(1) + k.C2n$

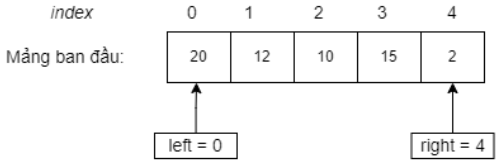
Lại có: $n = 2^k \rightarrow$k = logn,T(1) = C1

$\rightarrow$T(n) = $2^kT(1) + kC2n = n.C1 + C2n.logn = O(n.logn)$

Vậy độ phức tạp của Merge Sort là O(n.logn)

- Độ ổn định: có.

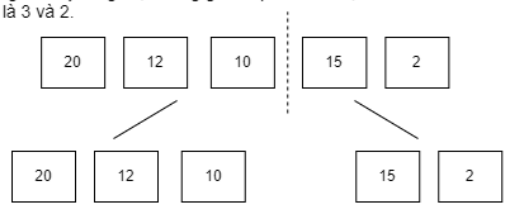
7.4. Minh hoạ thuật toán



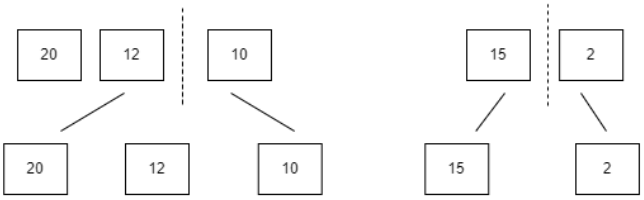
-Kiểm tra xem chỉ số bên trái (left) của mảng có nhỏ hơn chỉ số bên phải (right) hay không, nếu có thì hãy tính điểm giữa của nó (mid).

mid = (left + right)/2 = (0+4)/2 =2

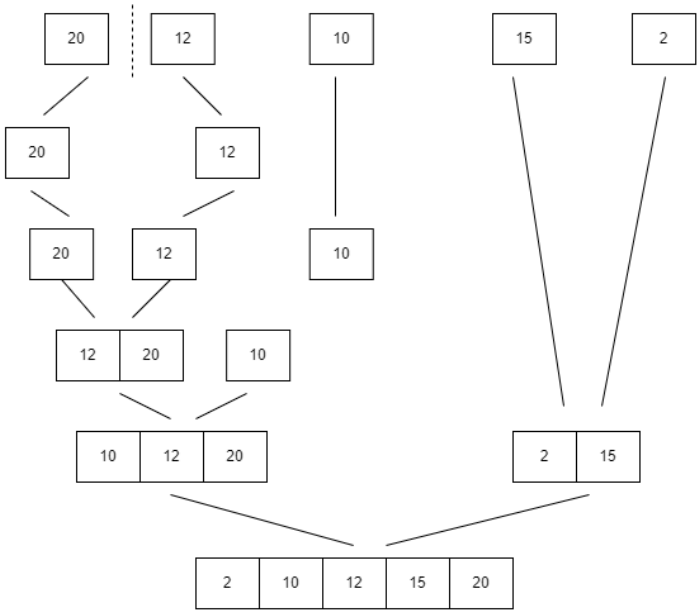
Chia lặp đi lặp lại toàn bộ mảng thành các nửa bằng nhau, cho đến khi mảng chỉ còn một phần tử.
Ở đây, chúng ta thấy rằng một mảng gồm 5 phần tử được chia thành hai mảng có kích thước tương ứng là 3 và 2.



Bây giờ, một lần nữa, hãy tìm chỉ số bên trái nhỏ hơn chỉ số bên phải cho cả hai mảng, nếu thấy có, sau đó lại tính điểm giữa cho cả hai mảng. iếp tục chia hai mảng này thành các nửa khác, cho đến khi đạt được các đơn vị nguyên tử của mảng và không thể phân chia thêm nữa.



Sau khi chia mảng thành các đơn vị nhỏ nhất, hãy bắt đầu hợp nhất các phần tử lại dựa trên so sánh kích thước của các phần tử
Trước hết, hãy so sánh phần tử cho từng danh sách và sau đó kết hợp chúng thành một danh sách khác theo cách được sắp xếp



Sau lần hợp nhất cuối cùng, mảng đã được sắp xếp.

8. Heap Sort

8.1. Ý tưởng

Có mối liên hệ nào giữa cấu trúc cây và sắp xếp thứ tự không? Heap Sort ra đời để trả lời cho câu hỏi đó.

Ta coi dãy cần sắp xếp là một cây nhị phân hoàn chỉnh, sau đó hiệu chỉnh cây thành dạng cấu trúc heap. Dựa vào tính chất của cấu trúc heap, ta có thể lấy được ra phần tử lớn nhất hoặc nhỏ nhất của dãy, phần tử này chính là gốc của heap. Giảm số lượng phần tử của cây nhị phân và tái cấu trúc heap. Đưa phần tử đỉnh heap về đúng vị trí của dãy ở cuối mảng, sau đó giảm số lượng phần tử của mảng (không xét tới phần tử cuối nữa). Tái cấu trúc heap và lặp lại việc lấy phần tử gốc của cấu trúc heap cho tới khi danh sách ban đầu chỉ còn 1 phần tử. Đưa phần tử này về đúng vị trí và kết thúc thuật toán.

8.2. Mã giả

```
function Heapify(A as array, n as int, i as int)
{
    max = i
    leftchild <- 2i + 1
    rightchild <- 2i + 2
    if (leftchild <= n) and (A[i] < A[leftchild])
        max = leftchild
    else
        max = i
    end if
    if (rightchild <= n) and (A[max] > A[rightchild])
        max = rightchild
    end
    IF (max != i)
        swap(A[i], A[max])
        Heapify(A, n, max)
    END IF
}
end function
procedure Heapsort(A as array)
{
   n = length(A)
   for i = n/2 -1 downto 0
     Heapify(A, n ,i)
   end for
   for i = n downto 2
     exchange A[1] with A[i]
```

```
        A.heapsize = A.heapsize - 1
        Heapify(A, i, 0)
      end for
} end procedure
```

8.3. Nhận xét

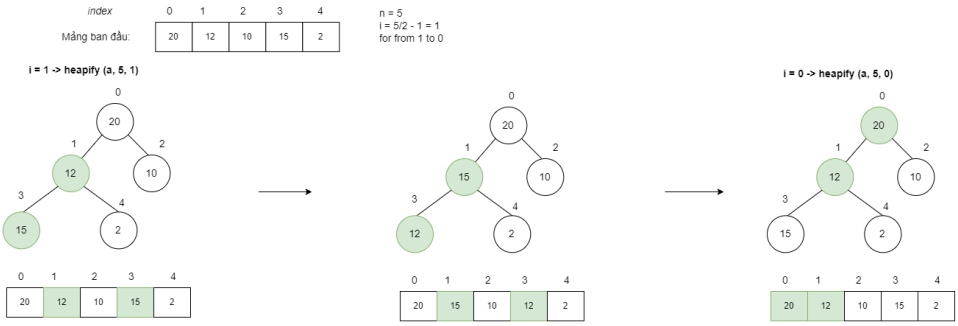
Độ phức tạp của heapify là O(logn), độ phức tạp của việc tạo và build heap là O(n). Trong mọi trường hợp Heap Sort đều có độ phức tạp là O(nlogn).

$log(n)+log(n-1)+log(n-2)+....log(1)llog(x)+log(y)=log(x*y) \Leftrightarrow log(n*(n-1)*(n-2)*...*2*1)=$

Sử dụng xấp xỉ của Stirling: $log(n!)=n*log(n)-n+O(log(n))$
- Độ ổn định: không.

8.4. Minh hoạ thuật toán

Xây dựng Max Heap



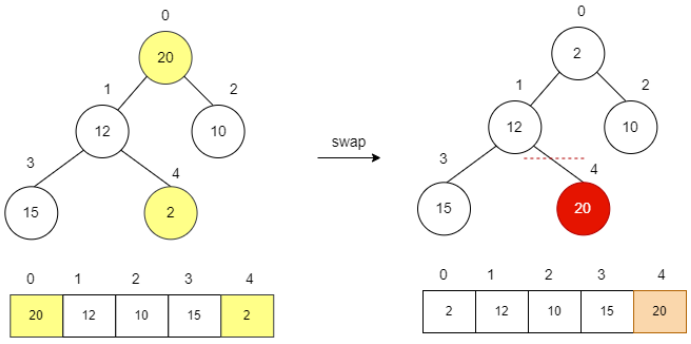
+ Cây thỏa mãn thuộc tính Max-Heap, nên phần tử lớn nhất được lưu trữ tại nút gốc.

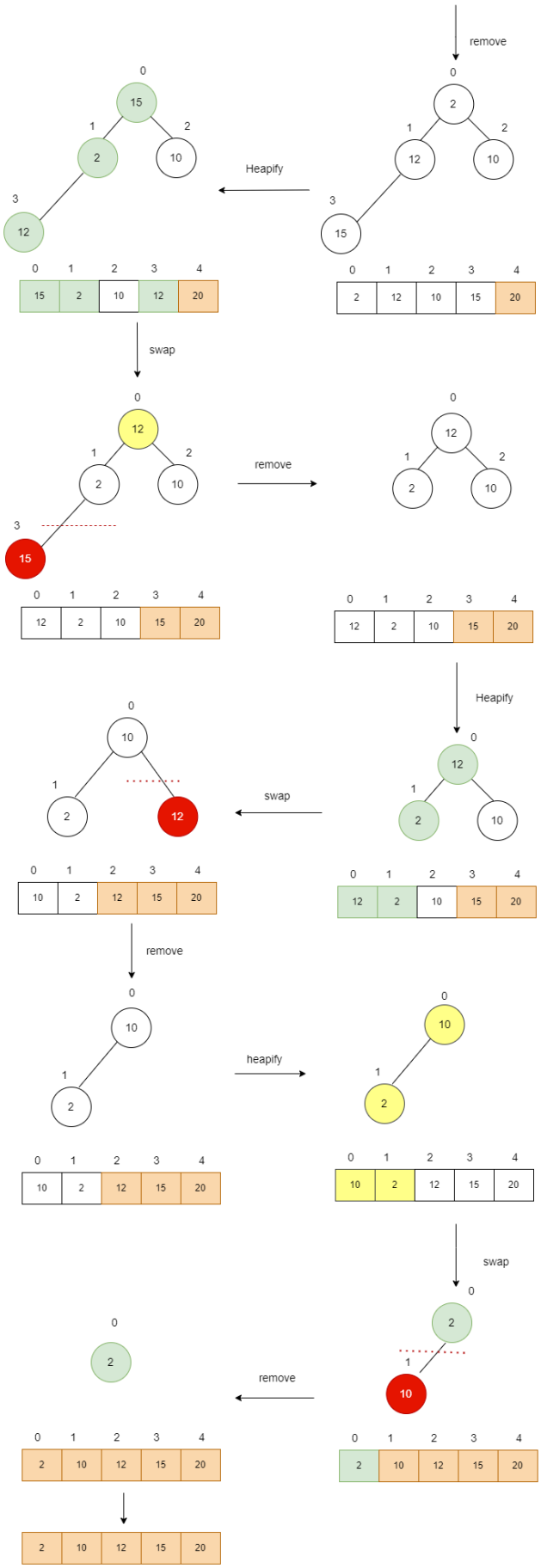
swap: Loại bỏ phần tử gốc và đặt ở cuối mảng (vị trí thứ n) Đặt mục cuối cùng của cây (đống) vào chỗ trống.

remove: Giảm kích thước của heap đi 1.

Heapify: Giải phóng phần tử gốc một lần nữa để chúng ta có phần tử cao nhất ở gốc.

Quá trình này được lặp lại cho đến khi tất cả các mục của danh sách được sắp xếp.




remove
Heapify
swap
remove
Heapify
swap
remove
heapify
swap
remove

9. Shell Sort

9.1. Ý tưởng

Thuật toán Shell Sort có thể nói là một cải tiến của thuật toán Insertion Sort, thuật toán này cho phép so sánh và hoán vị các phần tử nằm xa nhau thay vì chỉ được xét những phần tử kề nhau như thuật toán Insertion Sort. Ban đầu, ta sẽ có một dãy các số gọi là các "gaps", ta lần lượt duyệt qua các gaps này theo thứ tự giảm dần và ta sẽ chỉ so sánh các phần tử cách nhau một đoạn đúng bằng gap tương ứng.

9.2. Mã giả

```
procedure shell_sort(array, n)
    h = 1;
   while h < length(array) /3 :
                  h = ( interval * 3 ) + 1
   end while loop
   while h > 0 :
       for ( i = h; i < length(array); i++):
             insertion_value = array[i]
                   j = i;
             while j > h-1 and array[j-h] >= insertion_value:
                   array[j] = array[j-gap]
                   j = j-gap
              end while loop
                 array[j] = insertion_value
       end for loop
       h = (h-1)/3;
   end while loop
end procedure
```

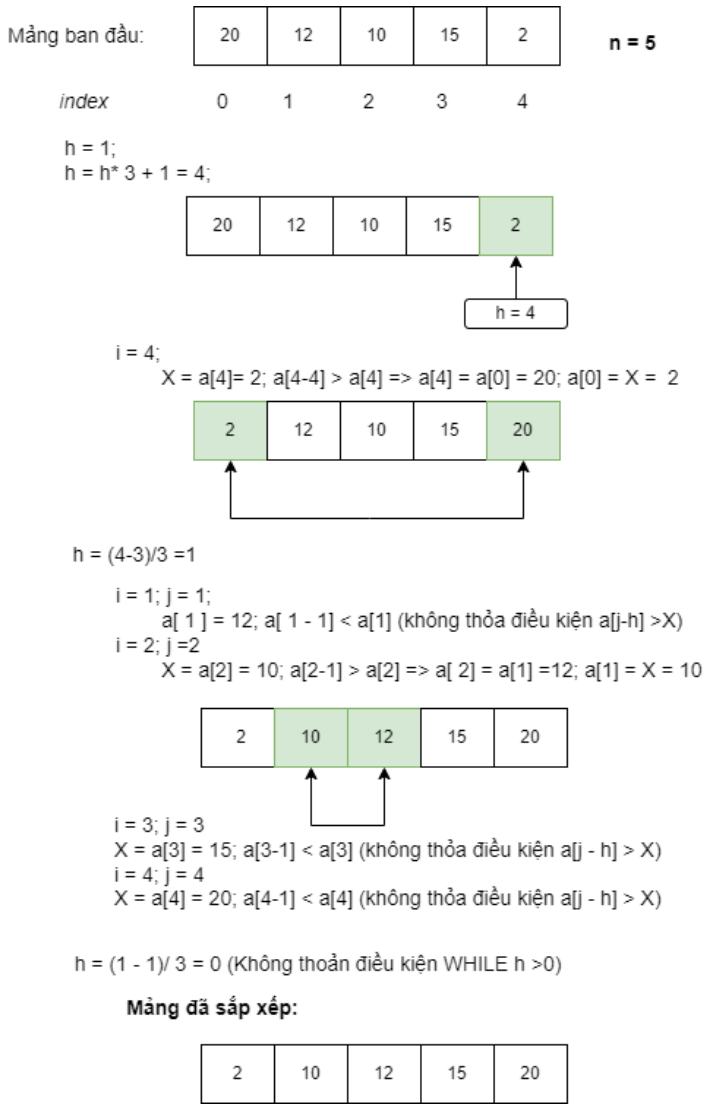
9.3. Nhận xét

Độ phức tạp của thuật toán Shell Sort phụ thuộc phần lớn vào cách ta chọn dãy gaps. Với một số dãy, việc đánh giá độ phức tạp của Shell Sort vẫn là một bài toán mở, chưa có lời giải.
Bộ nhớ cần dùng của thuật toán là O(1 + m), với m là độ dài dãy gaps.
Trong phần cài đặt thuật toán này, chúng tôi chọn dãy gaps có dạng:
g = {1750, 701, 301, 132, 57, 23, 10, 4, 1}
Dãy này được đề xuất bởi Ciura vào năm 2001.

Trường hợp tốt nhất: O($nlog_n$)
Trường hợp trung bình: O($n(log_n)^2$)
Trường hợp tệ nhất: O($n(log_n)^2$)

- Độ ổn định: có.

9.4. Minh hoạ thuật toán



10. Shaker Sort

10.1. Ý tưởng

Shaker Sort là một thuật toán cải tiến từ Bubble Sort. Ở Bubble Sort mỗi khi sắp xếp thì ta chỉ đưa ra các phần tử hướng về phía sau. Shaker Sort thì sau khi duyệt từ đầu mảng tới cuối mảng thì ta sẽ thực hiện một lần duyệt từ cuối về đầu mảng, với cách làm này thì các phần tử có thể được di chuyển theo hướng đi về trước. Sau mỗi lần lặp như thế, phần tử lớn nhất sẽ được đưa về cuối, phần tử nhỏ nhất sẽ được đưa lên đầu. Vì vậy, ta không cần duyệt hết lại từ đầu mảng.

10.2. Mã giả

```
        procedure ShakerSort(arr, n):
               left = 0; right = n-1;
               WHILE left < right:
                   FOR (i = left ; i <= right - 1; i++):
                       IF a[i] > a[i+1]
                           swap(a[i], a[i+1])
                       END IF
                   END FOR
                   right--;
                   FOR (j = right ; j <= left + 1; j--):
                       IF a[j-1] > a[j]
                           swap(a[j], a[j-1])
                       END IF
                   END FOR
                   left ++;
               END WHILE
    end procedure
```

10.3. Nhận xét

Độ phức tạp của thuật toán Shaker Sort hoàn toàn giống với Bubble Sort, chỉ khác duy nhất là thời gian chạy sẽ nhanh hơn vì số bước thực hiện sẽ ít hơn, nhưng nhìn chung khi nó về độ phức tạp trong mọi trường hợp thì nó vẫn là $O(n^2)$.

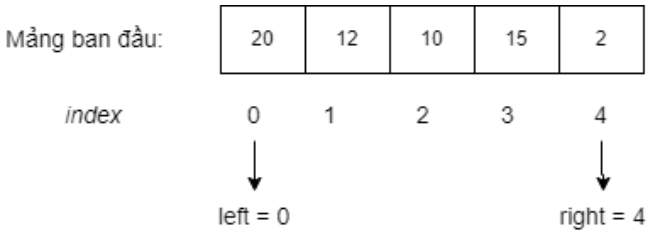
Trường hợp tốt nhất: O(n)

Trường hợp trung bình: O(n^2)

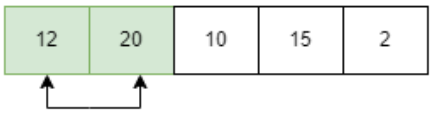
Trường hợp tệ nhất: O(n^2)

- Độ ổn định: có.

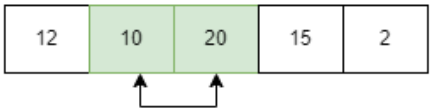
10.4. Minh hoạ thuật toán



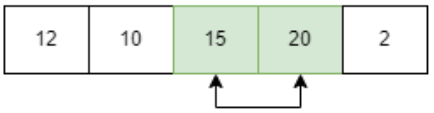
i = left = 0;
a[0] > a[1]=> swap (a[0], a[1]);



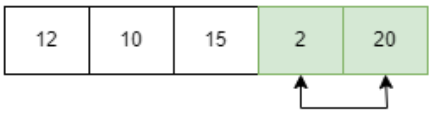
i = 1
a[1] > a[2]=> swap (a[1], a[2]);



i = 2
a[2] > a[3]=> swap (a[2], a[3]);

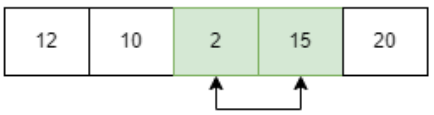


i = 3
a[3] > a[4]=> swap (a[3], a[4]);

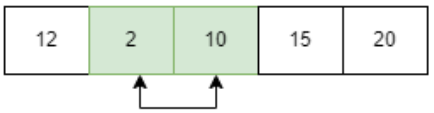


right = right -1 =3;

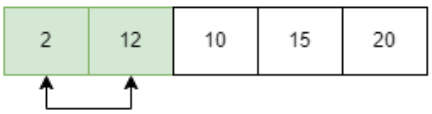
j = right = 3;
a[2] > a[3]=> swap (a[2], a[3]);



j = 2
a[1] > a[2]=> swap (a[1], a[2]);

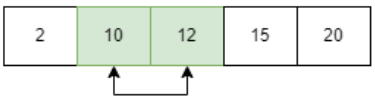


j = 1
a[0] > a[1]=> swap (a[0], a[1]);



left = left +1 = 1;

i = left = 1;
a[1] > a[2]=> swap (a[1], a[2]);



i = 2
a[2] > a[3] không thỏa điều kiện

i = 3
a[3] > a[4] không thỏa điều kiện

right = right - 1 =2

j = 2
a[1] > a[2] không thỏa điều kiện

left = left + 1 = 2

left = right = 2; Không thỏa điều kiện WHILE (left < right). Kết thúc vòng lặp.
Mảng đã sắp xếp:

11. Quick Sort

11.1. Ý tưởng

Dùng "chia để trị" làm hướng giải. Chia một mảng lớn thành 2 mảng con: 1 mảng có phần tử nhỏ, 1 mảng có phần tử lớn.

Ứng dụng đệ quy, sắp xếp hai mảng con:

Chọn một phần tử để so sánh, gọi là phần tử Key (Pivot), từ trong mảng đầu tiên.

Phân vùng và sort mảng con trong phân vùng làm sao cho các phần tử lớn hơn từ phần tử Key nằm sau (bên phải) và các phần tử bé hơn phần tử Key nằm trước (bên trái).

Cuối cùng là đệ quy sử dụng các bước trên cho các mảng với phần tử bé hơn và phân tách với các phần tử lớn hơn sau khi phân vùng.

11.2. Mã giả

Theo Sơ đồ phân vùng Lomuto:

```
procedure partition(A, start, end) :
    pivot = A[end]
        i = start - 1
    for j = start to end do
```

```
            if A[j] < pivot then
                i = i + 1
                swap A[i] with A[j]
        swap A[i+1] with A[end]
        return i + 1
end procedure
procedure quicksort(A, start, end) :
        if start >= end then return;
            else
                pi = partition(A, start, end)
                quicksort(A, start, pi - 1)
                quicksort(A, pi + 1, end)
end procedure
```

11.3. Nhận xét

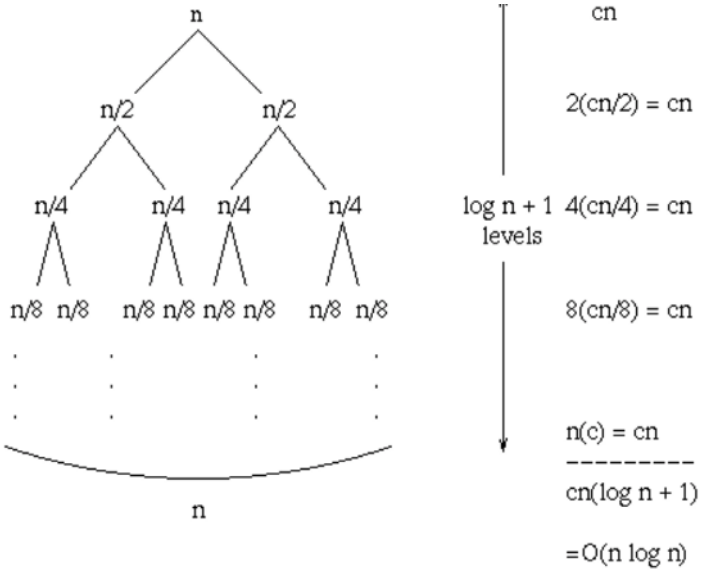
Cho mảng a có n phần tử. Gọi $T(n)$ là hàm truy hồi xác định độ phức tạp của thuật toán. Chia mảng thành hai nhóm có kích thước khác nhau, chúng phụ thuộc vào việc chọn pivot trong quá trình phân hoạch mảng. Giả sử sau phân hoạch phần tử a nằm trong mảng con bên trái và phần tử n - i - 1 nằm ở mảng con bên phải. Ta có:

- Size of LeftSubArray = i
- Size of RightSubArray = n - i - 1.

Vậy ta được công thức: $T(n) = T(i) + T(n-i-1)$

- Trường hợp tốt nhất: O(n*logn) xảy ra khi chọn phần tử trung vị của mảng làm pivot, kích thước của cả hai mảng con bằng một nửa kích thước mảng ban đầu:

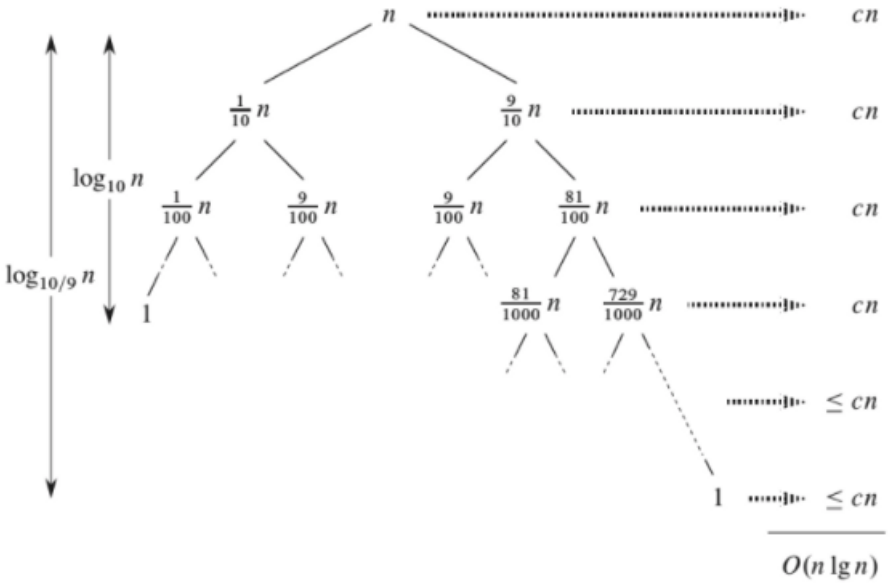
$$T(n) = T(\frac{n}{2}) + T(n-1-\frac{n}{2}) + C_n = T(\frac{n}{2}) + T(\frac{n}{2}-1) + C_n \approx 2T(\frac{n}{2}) + C_n \qquad (2.7)$$



- Trường hợp xấu nhất: $O(n^2)$ xảy ra khi tìm ra phần tử lớn nhất hoặc nhỏ nhất của mảng ngay tại pivot. Kích thước mảng con sau khi phân hoạch không cân bằng nữa mà trở thành n-1 và 1.

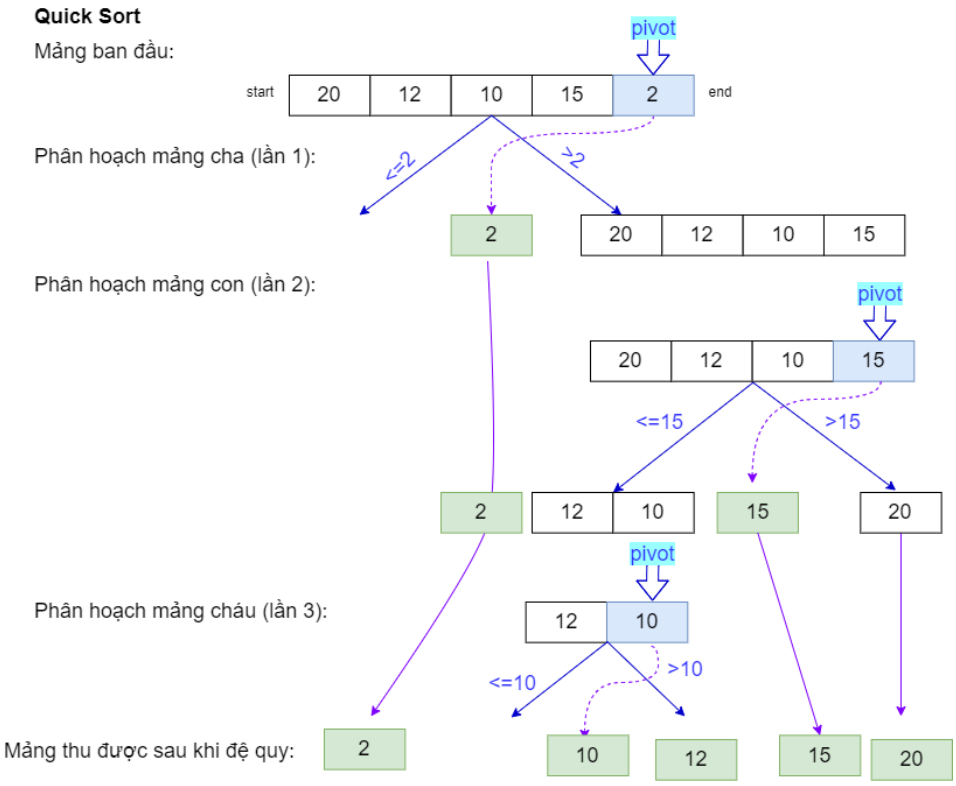
$$
\begin{aligned}
&T(n)\\
&\quad = T(n-1)+C_n\\
&\quad = T(n-2)+C_(n-1)+C_n\\
&\quad = T(n-3)+C_(n-2)+C_(n-1)+C_n\\
&\quad = ...\\
&\quad = T(1)+2C+3C+...+C_(n-3)+C_(n-2)+C_(n-1)+C_n\\
&\quad = C+2C+3C+...+C_(n-3)+C_(n-2)+C_(n-1)+C_n\\
&\quad = C(1+2+3+..+n-3+n-2+n-1+n)\\
&\quad \Rightarrow T(n) = C_(n*\frac{(n+1)}{2}) = O(n^2)
\end{aligned}
\tag{2.8}
$$

- Trường hợp trung bình: $O(n*logn)$ Giả sử quá trình phân hoạch mảng tạo ra hai mảng con không cân theo tỷ lệ 9:1. Ta có: $T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + C_n$



- Độ ổn định: không.

11.4. Minh hoạ thuật toán



12. Radix Sort

12.1. Ý tưởng

Không dùng cách phép so sánh phần tử như các phương pháp so sánh thường dùng mà xoay quanh việc áp dụng một phương pháp sort ổn định như counting sort làm chương trình con để sắp xếp các giá trị phần tử theo thứ tự các chữ số như trên mảng đã cho.

12.2. Mã giả

```
Radix-Sort(A, d)
   for j = 1 to d do
        int count[10] = {0};
        for i = 0 to n do
            count[key of(A[i]) in pass j]++
        for k = 1 to 10 do
            count[k] = count[k] + count[k-1]
        for i = n-1 downto 0 do
            result[ count[key of(A[i])] ] = A[j]
            count[key of(A[i])]--
        for i=0 to n do
            A[i] = result[i]
```

```
            end for(j)
        end func
```

12.3. Nhận xét

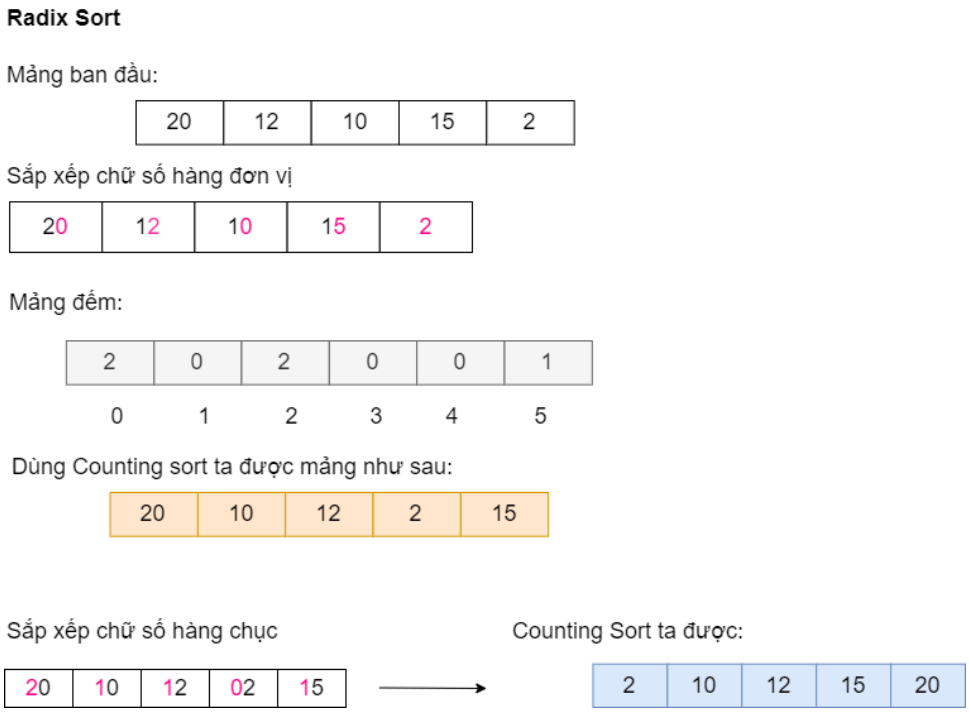
Trường hợp tốt nhất: Mảng đã được sắp xếp => O(n)

Trường hợp xấu nhất: Mảng sắp xếp giảm dần, counting sort được gọi k lần với k là số lần xuất hiện của chữ số lớn nhất trong mảng đã tách chữ số. Khi đó độ phức tạp là. O(n*k)

Trường hợp trung bình: Mảng sắp xếp ngẫu nhiên, tương tự ta gọi Counting Sort k lần. Cho nên độ phức tạp là: O(n*k)

Độ ổn định: không.

12.4. Minh hoạ thuật toán



13. Counting Sort

13.1. Ý tưởng

Đếm số lần xuất hiện của từng phần tử duy nhất (key) trong mảng, phân phối lại mảng chính thông qua việc ánh xạ các số đếm được lưu trong mảng phụ để xác định vị trí từng key trong mảng được sắp xếp.

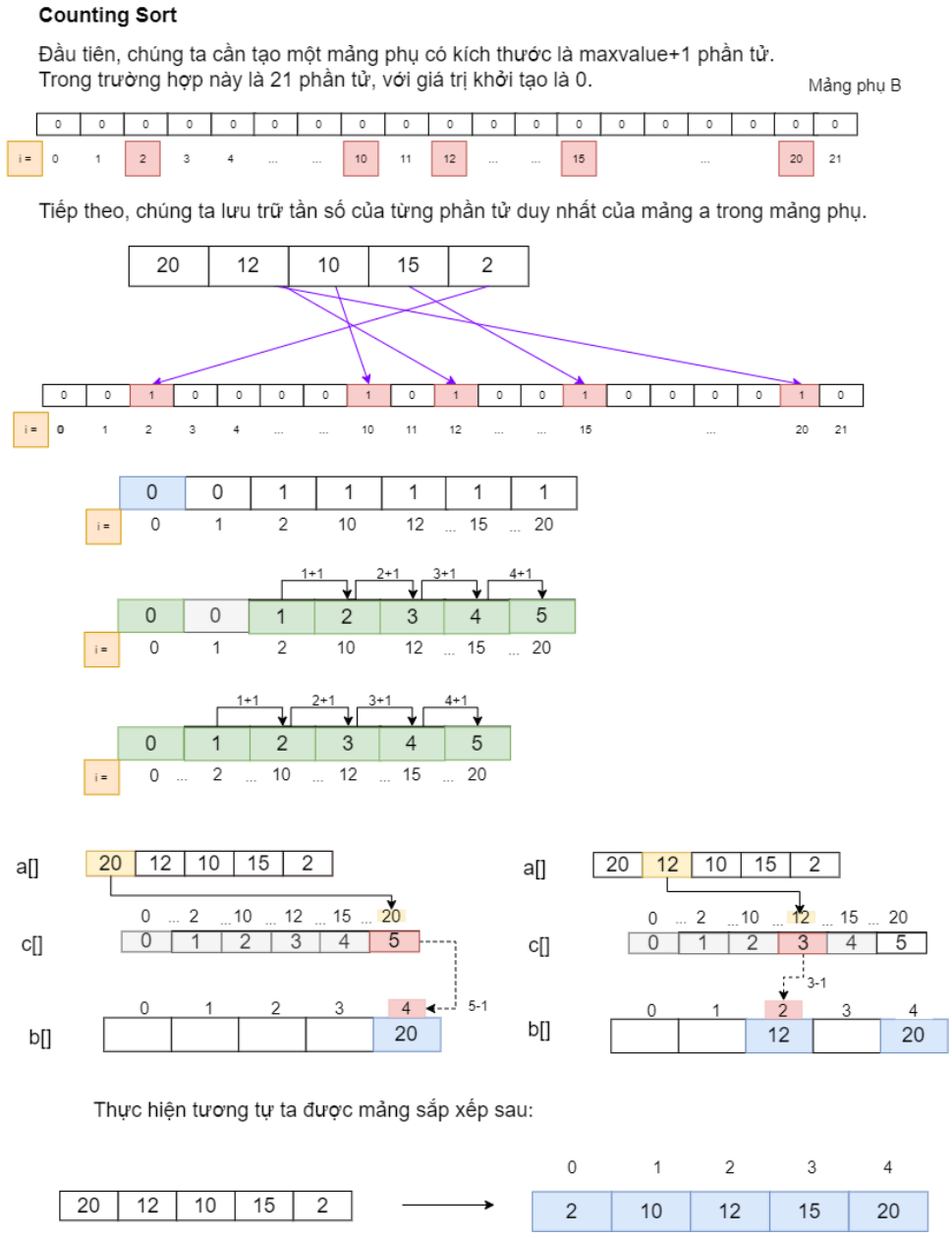
13.2. Mã giả

```
function countingSort(array, min, max):
    count: array of (max - min + 1) elements
    initialize count with 0
        for each number in array do
            count[number - min] := count[number - min] + 1
        end for
    z := 0
        for i from min to max do
            while ( count[i - min] > 0 ) do
                array[z] := i
                z := z+1
                count[i - min] := count[i - min] - 1
            end while
        end for
end function
```

13.3. Nhận xét

Trường hợp tốt nhất: O(n+k)
Trường hợp tệ nhất: O(n+k)
Trường hợp trung bình: O(n+k)
Độ ổn định: có.

13.4. Minh hoạ thuật toán



14. Flash Sort

14.1. Ý tưởng

Đồng thời thực hiện phân phối và so sánh giá trị phần tử bằng cách chia các phần tử vào m lớp khác nhau với quy định $class[x] = \frac{(m-1)}{a[max]a[minVal]}$ với $m = 0.45$. Sau khi quá trình phân phối kết thúc chúng ta dùng thuật toán sắp xếp chèn để sắp xếp lại mảng.

14.2. Mã giả

```
procedure FlashSort(a,n)
```

```
    int max = 0, min = a[0]
    int m = (int) 0.45*n
    int *L = new int[m]
        function findminValue(a) return min;
        function findmaxIndex(a) return max;
            if(min==a[max]) then return a
    int classA = (m-1)/(a[max]-min)
        for k = 0 to m: createArray(L,m,0);
        for j = 0 to n:
            k = (int)classA*(a[j]-min)
            L[k]++
        for p = 1 to m: L[p] += L[p-1]
        k = m-1, j = 0, moved = 0;
             while(moved < (n-1)):
                while(j > L[k]-1):
                    j++
                    k = classA * (a[j]-min)
                end while
                if (k<0) stop
            flash = a[j]
                while (j!=L[k])
                    k = classA*(flash-min)
                    hold = a[t=L[k]--]
                    a[t]= flash
                    flash = hold
                    moved++
                end while
            end while
    InsertionSort(a,n)
end procedure
```

14.3. Nhận xét

Quá trình phân phối mảng có độ phức tạp là O(n) vì chúng ta dùng một dòng for để duyệt mỗi phần tử một lần.

Quá trình sắp xếp:

+ Phân lớp: gồm $\frac{n}{m}$ phần tử với m lớp

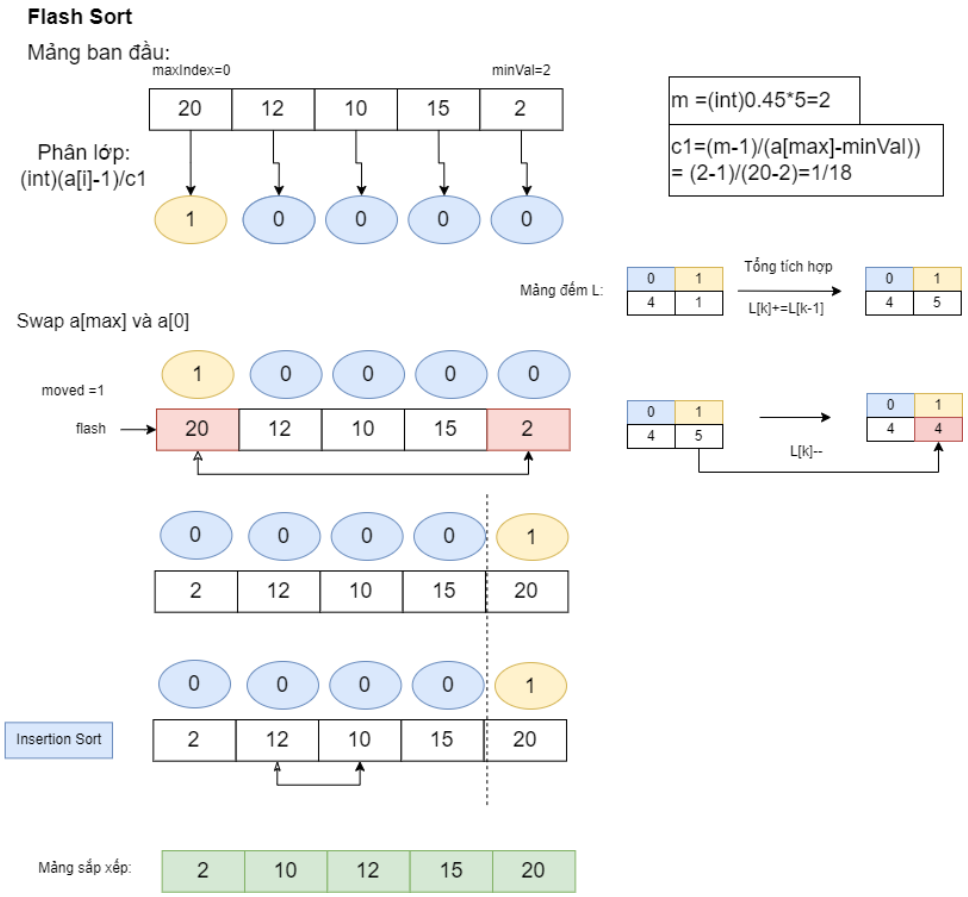
+ Sắp xếp chèn: $O(n^2)$

$\Rightarrow$O($\frac{n^2}{m^2}$)* m = O($\frac{n^2}{m}$)

Với m = 0.45n thì **độ phức tạp của thuật toán là:** O($\frac{n^2}{0.45n}$)$\approx O(n)$

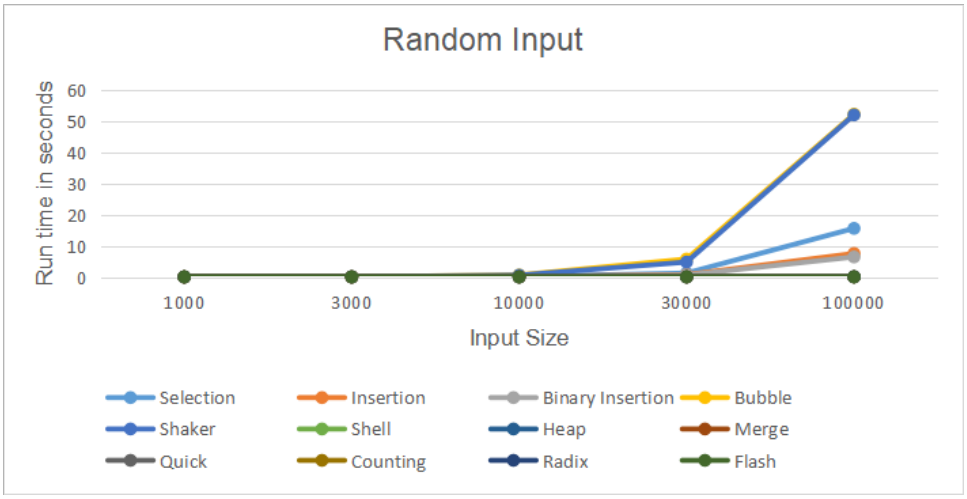
- Độ ổn định: không.

14.4. Minh hoạ thuật toán



CHƯƠNG 3

BIỂU ĐỒ



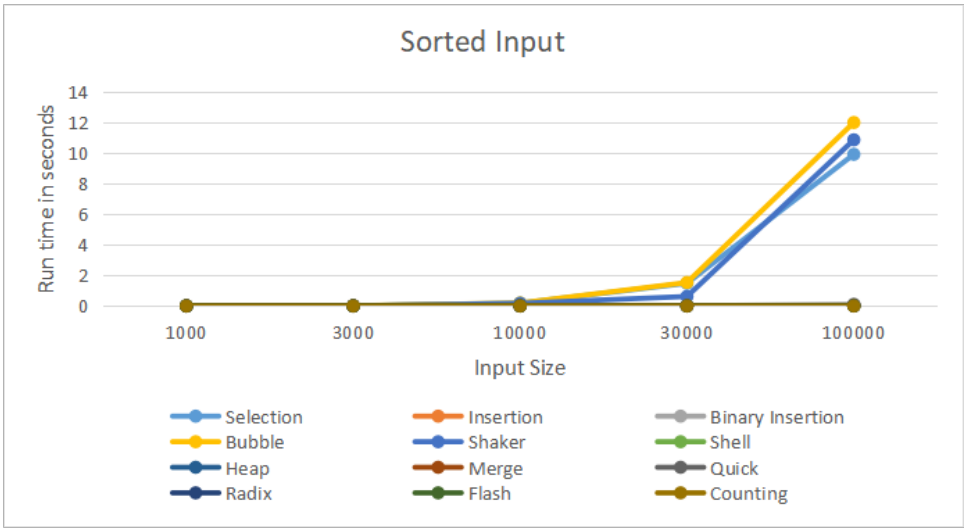
Nhận xét:

Đồ thị này có phần khó xem vì một số thuật toán với độ phức tạp O(n^2) như Bubble Sort, Shaker Sort hay Selection Sort có thời gian chạy quá lớn, làm ta khó nhìn thấy phần còn lại. Qua đây ta thấy trong các thuật toán sắp xếp cơ bản với độ phức tạp O(n^2) thì Bubble Sort là thuật toán tệ nhất, dù có tối ưu nó bằng Shaker Sort cũng không cải thiện tốc độ lắm.

Thuật toán Selection Sort sẽ là lâu nhất trong các thuật toán còn lại, chậm gần gấp đôi so với Insertion. Thuật toán Binary Insertion Sort khá tốt, có thể nói là thuật toán chạy nhanh nhất trong các thuật toán O(n^2) ngoại trừ Shell Sort.

Đa số đều là các thuật toán có độ phức tạp O(*nlogn*) hoặc tuyến tính. Rất bất ngờ đó là thuật toán Shell Sort có tốc độ nhanh hơn Merge Sort. Trong 3 thuật toán Merge Sort, Heap Sort, Quick Sort thì Quick Sort có tốc độ nhanh nhất, rồi tới Merge Sort và sau cùng là Heap Sort. Thuật toán Quick Sort nếu cài đặt tốt, không bị rơi vào trường hợp chọn chốt tệ thì sẽ có tốc độ thực thi rất nhanh chóng. Thuật toán nhanh nhất chính là Counting Sort, với độ phức tạp tuyến tính cùng hằng số lập trình nhỏ, thuật toán có thời gian chạy nhanh, dễ cài đặt nhưng chỉ khả dụng nếu giới hạn dữ liệu nhỏ. Thuật toán nhanh tiếp theo là Flash Sort, về trung bình

cũng có độ phức tạp tuyến tính, tuy nhiên do Flash Sort có hằng số lập trình cao nên vẫn chậm hơn Counting Sort rất nhiều, tuy vậy Flash Sort có thể dùng được khi dữ liệu lớn và có tính ngẫu nhiên cao. Một điểm trừ của nó là khó cài đặt. Trong các thuật toán ổn định, thuật toán Counting Sort có tốc độ nhanh nhất, Bubble Sort có tốc độ chậm nhất. Với các thuật toán không ổn định thì Flash Sort có tốc độ nhanh nhất và Selection Sort chậm nhất.



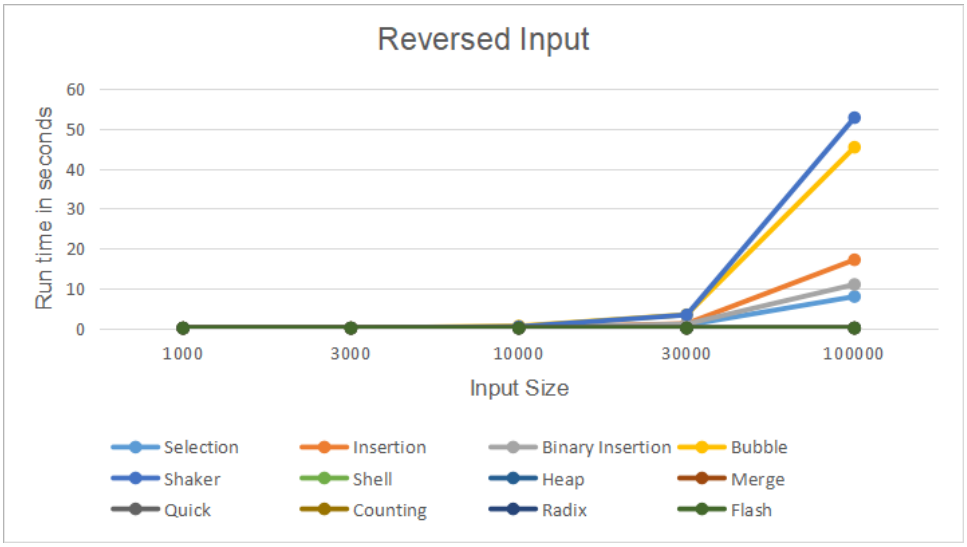
Nhận xét:

Đồ thị này có phần khó xem vì một số thuật toán với độ phức tạp O(n^2) như Bubble Sort hay Shaker Sort có thời gian chạy quá lớn dù dữ liệu đã sắp xếp, làm ta khó nhìn thấy phần còn lại. Qua đây ta thấy trong các thuật toán sắp xếp cơ bản với độ phức tạp O(n^2) thì Bubble sort là thuật toán tệ nhất, dù có tối ưu nó bằng Shaker Sort cũng không cải thiện tốc độ lắm, nhưng nếu ta dùng cải tiến rằng khi mảng đã được sắp xếp thì tốc độ sẽ rất nhanh. Với thuật toán Selection Sort có thời gian chạy nhanh hơn Bubble Sort và Shaker Sort một chút.

Những thuật toán còn lại chủ yếu đều có độ phức tạp O($nlogn$) hay O(n) ở trường hợp mảng đã được sắp xếp. Trong đó chậm nhất chính là Merge Sort với độ phức tạp O($nlogn$). Tiếp theo chính là Binary-Insertion Sort, thuật toán này chạy nhanh hơn Merge Sort một chút bởi vì có hằng số lập trình thấp hơn Merge sort nhiều.

Tiếp theo là Heap Sort và Quick Sort, cũng giống với dữ liệu ngẫu nhiên, thuật toán Heap Sort chạy nhanh ngang thời gian với Merge Sort..Radix Sort và Flash Sort cũng là 2 thuật toán cuối cùng có độ phức tạp O($nlogn$) và O($nlogk$) trước khi ta tới với những thuật toán có độ phức tạp O(n) trong trường hợp này.

Thuật toán Shell Sort, mặc dù không tốn bất kì phép chèn nào nhưng Shell Sort chạy lâu hơn nhiều so với các thuật toán O(n) khác đó là vì Shell Sort phải duyệt qua mảng nhiều lần.



Nhận xét:

Vẫn dẫn đầu các thuật toán chạy chậm nhất đó chính là Bubble Sort và Shaker Sort, đây cũng là trường hợp khiến cho hai thuật toán này sử dụng nhiều phép hoán vị nhất, đó đó ta thấy thời gian chạy của 2 thuật toán này gấp nhiều lần so với Selection Sort. Một điều đặc biệt đó là cải tiến của cả 2 thuật toán này đều chạy lâu hơn so với khi không được cải tiến.

Thuật toán Insertion Sort chạy hơn thuật toán Selection Sort. Tiếp theo đó là thuật toán Binary Insertion Sort. Trong các thuật toán còn lại, đa số đều là các thuật toán có độ phức tạp O($nlogn$) hoặc tuyến tính. Giống với những dữ liệu khác, ba thuật toán Merge Sort, Heap Sort và Quick Sort vẫn giữ đúng thứ tự thời gian chạy. Với dữ liệu này, ta thấy thuật toán Radix Sort có tốc độ chạy nhanh hơn thuật toán Flash Sort một chút. Thuật toán nhanh nhất vẫn là Counting Sort.

KẾT LUẬN

Thuật toán Tìm kiếm và So sánh là một trong những chủ đề thảo luận chính diễn ra trong nhiều thế kỷ của lĩnh vực lập trình. Có thể nói rằng mỗi thuật toán có những ưu điểm và nhược điểm riêng biệt. Việc phân tích và thí nghiệm dựa trên các phương án cài đặt cho ta dễ dàng tiếp cận và hiểu sâu hơn về các mặt lợi - hại của từng thuật toán. Qua đó hệ thống lại kiến thức lập trình, xây dựng tư duy sử dụng thuật toán sao cho phù hợp với vấn đề cần giải quyết.

Như vậy, nghiên cứu về thuật toán Tìm kiếm và So sánh mang ý nghĩa hết sức quan trọng cả trong thực tiễn lẫn khoa học máy tính. Vì lẽ đó, nhóm chúng em đã tìm hiểu và trình bày những nội dung khái quát nhất về 2 loại thuật toán tìm kiếm và 12 loại thuật toán so sánh góp phần củng cố các giá trị nền tảng và tạo tiền đề để phát triển các thuật toán sau này.

TÀI LIỆU THAM KHẢO

[1] Frank M. Carrano, Timothy Henry (2006), "Data Abstraction and Problem Solving with C++": Walls and Mirrors 6th, Pearson.

[2] "Everything About Sorting Algorithms", Interview Kickstart,

[3] Source code về Heap Sort tại https://www.programiz.com

[4] Source code về Heap Sort tại https://www.geeksforgeeks.org

[5] nguyenvanquan7826 (2014) "Cách tính độ phức tạp thuật toán – Algorithm complexity

[6] Sửa lỗi với https://stackoverflow.com