TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN THỰC HÀNH

TÌM HIỂU VỀ CÁC THUẬT TOÁN TÌM KIẾM VÀ SẮP XẾP

Tác giả:
Huỳnh Lê Minh Thư
Lê Nguyễn Hoàng Uyên

Giảng viên:
Nguyễn Bảo Long

Ngày hoàn thành:
Ngày 24 tháng 10 năm 2022

Tên học phần:
Thực hành Cấu trúc dữ liệu và giải thuật

Mục lục

Chương 1

Báo cáo hoàn thành

1.1 Bảng phân công công việc

Người phụ trách	Phân công công việc	Hoàn thành
21280110 Huỳnh Lê Minh Thư	Tìm hiểu và trình bày: Binary Insertion sort, Heap sort, Merge sort, Flash sort	85%
	Cài đặt 12 thuật toán Sort	100%
	Xuất file txt và csv	100%
	Vẽ biểu đồ	100%
21280118 Lê Nguyễn Hoàng Uyên	Tìm hiểu và trình bày: Linear Search, Binary Search, Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Quick Sort, Counting Sort, Radix Sort	80%
	Cài đặt 2 thuật toán Search	100%
	Viết báo cáo	100%
	Kiểm tra lại thành phẩm	100%

1.2 Tự đánh giá điểm số

Dựa vào tỉ lệ hoàn thành trong bảng phân công công việc:

MSSV	Tên	Điểm
21280110	Huỳnh Lê Minh Thư	9.7
21280118	Lê Nguyễn Hoàng Uyên	9.5

Chương 2

Nội dung các thuật toán tìm kiếm và sắp xếp

2.1 Linear Search

2.1.1 Ý tưởng thuật toán

Duyệt qua tất cả các phần tử của dãy, trong quá trình duyệt nếu tìm thấy phần tử có giá trị khóa bằng với giá trị khóa cần tìm kiếm thì trả về vị trí của phần tử tìm được. Ngược lại, nếu duyệt tới hết dãy mà vẫn không có phần tử thỏa mãn yêu cầu thì trả về giá trị mặc định nào đó.

2.1.2 Mã giả

```
Function linear_search()
    Input: a[], key

    for i from 0 to n-1 do
        if a[i] == key return i
        else return -1
        end if
    end for

    Output: i|(a[i] = key)
End.
```

2.1.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	1	$O(1)$
Trung bình	$\frac{n}{2}$	$\frac{n}{2}$	$O(n)$
Tệ nhất	n	n	$O(n)$

Trường hợp xấu nhất, khi phần tử cần tìm là phần tử cuối cùng của mảng hay phần tử cần tìm không tồn tại trong mảng, mỗi phép so sánh này đều có thể thực hiện n lần. Do đó, độ phức tạp của thuật toán là $O(n)$.

2.1.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Phần tử cần tìm là $X = 3$. Các bước giải thuật toán sẽ được thực hiện như sau:

	A[0]	A[1]	A[2]	A[3]	A[4]
$i = 0$	13	8	2	3	5
$i = 1$	13	8	2	3	5
$i = 2$	13	8	2	3	5
$i = 3$	13	8	2	3	5

- Vòng lặp $i = 0$, $A[0] = 13$: bỏ qua.
- Vòng lặp $i = 1$, $A[1] = 8$: bỏ qua.
- Vòng lặp $i = 2$, $A[2] = 2$: bỏ qua.
- Vòng lặp $i = 3$, $A[3] = 3$: chọn, in ra màn hình $i = 3$.

  Dừng giải thuật

2.2 Binary Search

2.2.1 Ý tưởng thuật toán

Giải thuật tìm kiếm nhị phân là giải thuật dùng để tìm kiếm phần tử trong dãy đã được sắp xếp. Trong mỗi bước, ta tiến hành so sánh phần tử cần tìm với phần tử nằm ở chinh giữa dãy. Nếu hai phần tử bằng nhau thì thao tác tìm kiếm thành công và giải thuật kết thúc. Nếu chúng không bằng nhau thì tùy vào phần tử nào lớn hơn, giải thuật lặp lại bước so sánh trên với nửa đầu hoặc nửa sau của dãy và cứ tiếp tục như thế.

2.2.2 Mã giả

```
1  Function binary_search()
2      Input: n, key, left, right
3
4      for i from 0 to n-1
5          if(mid == key) return mid
6          else if(mid > key)
7              right = mid - 1
8          else
9              left = mid + 1
10         end if
11     end for
12
13     Output: mid
14 End.
```

2.2.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	1	$O(1)$
Trung bình	$n+1$	$n\log(n-1)+1$	$O(\log n)$
Tệ nhất	$n+1$	$n\log(n-1)+1$	$O(\log n)$

Trường hợp xấu nhất, khi phần tử ở cuối hay đầu dãy. Trong trường hợp này tổng các phép so sánh và phép gán là $\log n$. Do đó, độ phức tạp của thuật toán là $O(\log(n))$.

2.2.4 Ví dụ minh họa

Giả sử cho dãy $A = \{3, 4, 5, 8, 13\}$. Phần tử cần tìm là $X = 3$. Các bước giải thuật toán sẽ được thực hiện như sau:

- $left = 0, right = 4, mid = \frac{left+right}{2} = 2$

A[0]	A[1]	A[2]	A[3]	A[4]
3	4	5	8	13

So sánh: $A[mid] = 5 > X = 3$: Không phải X, cập nhật lại $right = mid - 1 = 1$

- $left = 0, right = 1, mid = \frac{left+right}{2} = 0$

A[0]	A[1]
3	4

So sánh: $A[mid] = 3$ và $X = 3$. Đúng giá trị X cần tìm

Dừng thuật toán.

2.3 Selection Sort

2.3.1 Ý tưởng thuật toán

Giả sử cần sắp xếp tăng dần một dãy số có n phần tử $a_0, a_1, a_2, .. , a_{n-1}$. Cho dãy a được chia làm hai phần: phần trên trái đã sắp xếp và phần bên phải chưa sắp xếp.

Chọn phần tử nhỏ nhất trong n phần tử của danh sách ban đầu. Tìm và đổi vị trí phần tử nhỏ nhất với phần tử đầu tiên trong danh sách. Lúc này, phần tử a_0 sẽ có giá trị nhỏ nhất trong danh sách.

Sau đó, xem danh sách cần sắp xếp hiện tại chỉ gồm $n - 1$ phần tử, bắt đầu từ phần tử thứ 2 trong danh sách ban đầu. Tức là danh sách hiện tại chỉ gồm $a_1, a_2, .. , a_{n-1}$.

Lặp lại quá trình trên cho danh sách hiện tại đến khi danh sách hiện tại chỉ còn một phần tử.

2.3.2 Mã giả

```
1   Function selection_sort()
2       Input: a[], n
3
4       for i from 0 to n-1 do
5           min = i;
6           for j from i+1 to n-1 do
7               if a[j] < a[min]
8                   min = j
9               end if
10          end for
11          if (min != i )
12              swap a[min] and a[i]
13          end if
14      end for
```

```
15
16          Output: sorted array a[ ]
17      End.
```

2.3.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	0	$\frac{n(n+1)}{2}$	$O(n^2)$
Trung bình	n	$\frac{n(n+1)}{2}$	$O(n^2)$
Tệ nhất	n	$\frac{n(n+1)}{2}$	$O(n^2)$

Với mỗi giá trị của i, giải thuật thực hiện $n-i-1$ phép so sánh và vì i chạy từ 0 cho tới $n-2$, giải thuật sẽ cần $(n-1)+(n-2)+..+1 = \frac{n(n-1)}{2}$ tức là $O(\frac{n}{2})$ phép so sánh. Trong mọi trường hợp, số lần so sánh của thuật toán là không thay đổi. Mỗi lần chạy của vòng lặp đối với biến i, có thể có nhiều nhất một lần đổi chỗ hai phần tử nên số lần đổi chỗ nhiều nhất của thuật toán là n. Như vậy, trong trường hợp tốt nhất, thuật toán cần 0 lần đổi chỗ, trung bình cần $\frac{n}{2}$ lần đổi chỗ và tệ nhất cần n lần đổi chỗ.

2.3.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

A[0]	A[1]	A[2]	A[3]	A[4]
13	8	2	3	5
2	8	13	3	5
2	8	13	3	5

- Vòng lặp $i = 0$, dãy: 13,8,2,3,5, số phần tử của dãy: 5

  Giả sử phần tử nhỏ nhất: $min = A[0] = 13$

  Tìm được phần tử nhỏ nhất thực tế :$min = A[2] = 2$

  Hoán vị $A[0]$ và $A[2]$, được dãy số mới: 2,8,13,3,5

  Loại phần tử $A[0]$ ra khỏi dãy

A[0]	A[1]	A[2]	A[3]	A[4]
2	8	13	3	5
2	3	13	8	5
2	3	13	8	5

- Vòng lặp $i = 1$, dãy: 2,8,13,3,5, số phần tử của dãy: 4

  Giả sử phần tử nhỏ nhất :$min = A[1] = 8$

  Tìm được phần tử nhỏ nhất thực tế :$min = A[3] = 3$

  Hoán vị $A[1]$ và $A[3]$, được dãy số mới: 2,3,13,8,5

  Loại phần tử $A[1]$ ra khỏi dãy

A[0]	A[1]	A[2]	A[3]	A[4]
2	3	13	8	5
2	3	5	8	13
2	3	5	8	13

- Vòng lặp $i = 2$, dãy: 2,3,13,8,5, số phần tử của dãy: 3

  Giả sử phần tử nhỏ nhất::$min = A[2] = 13$

  Tìm được phần tử nhỏ nhất thực tế :$min = A[4] = 5$

  Dãy số mới: 2,3,5,8,13

  Loại phần tử $A[2]$ ra khỏi dãy

A[0]	A[1]	A[2]	A[3]	A[4]
2	3	5	8	13
2	3	5	8	13

- Vòng lặp $i = 3$, dãy: 2,3,5,8,13, số phần tử của dãy: 2

  Giả sử phần tử nhỏ nhất::$min = A[3] = 8$

  Không hoán vị trong hợp này do phần tử giả sử nhỏ nhất và phần tử nhỏ nhất thực tế cũng nằm ở một vị trí trong dãy.

  Dãy số mới: 2,3,5,8,13

  Loại phần tử $A[3]$ ra khỏi dãy

A[0]	A[1]	A[2]	A[3]	A[4]
2	3	5	8	13
2	3	5	8	13

- Vòng lặp $i = 4$, dãy: 2,3,5,8,13, số phần tử của dãy: 1

Dừng giải thuật

Vậy sau khi sắp xếp tăng: 2,3,5,8,13

2.4 Insertion Sort

2.4.1 Ý tưởng thuật toán

Giả sử cần sắp xếp tăng dần một danh sách có n phần tử $a_0, a_1, .. , a_{n-1}$

Giả sử đoạn $a[0]$ trong danh sách đã được sắp xếp. Bắt đầu từ phần tử thứ $i = 1$, tức là a_1. Tìm cách chèn phần tử a_i vào vị trí thích hợp của đoạn đã được sắp xếp để có dãy mới $a_0, .. , a_i$ trở nên có thứ tự. Vị trí này chính là vị trí giữa hai phần tử a_{k-1} và a_k thỏa $a_{k-1} < a_i < a_k$ $(0 \leq k \leq i)$. Lưu ý, khi $k = 0$ thì có nghĩa là a_i cần chèn vào trước vị trí đầu tiên trong danh sách.

Lặp lại quá trình trên ở mỗi lần lặp với mỗi lần lặp thì tăng i lên 1 đến khi $i < n$ thì dừng quá trình lặp.

2.4.2 Mã giả

```
Function Insertion_Sort()
    Input: a[], n, key

    for i from 1 to n - 1 do
        key = a[i]
        j = i      1
        while (j >= 0 && a[j] > key)
            a[j + 1] = a[j]
            j = j - 1
        end while
        a[j + 1] = key
    end for

    Output: sorted array a[ ]
End.
```

2.4.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	n	$O(n)$
Trung bình	1	$\frac{n^2}{4}$	$O(n^2)$
Tệ nhất	1	$\frac{(n-1)(n-2)}{2}$	$O(n^2)$

Thuật toán sắp xếp chèn là một thuật toán sắp xếp ổn định và là thuật toán nhanh nhất trong số các thuật toán sắp xếp cơ bản.

Với mỗi giá trị của i, chúng ta cần thực hiện so sánh khóa hiện tại $a[i]$ với nhiều nhất là i khóa và vì i chạy từ 1 tới $n-1$, nên chúng ta phải thực hiện nhiều nhất: $1+2+..+n-1 = \frac{n(n-1)}{2}$, tức là $O(n^2)$ phép so sánh tương tự như thuật toán sắp xếp chọn. Tuy nhinhiên, vòng lặp while không phải lúc nào cũng được thực hiện và nếu thực hiện thì cũng không nhất định là lặp i lần, nên trên thực tế thuật toán sắp xếp chèn nhanh hơn so với thuật toán xếp chọn. Trong trường hợp tốt nhất, thuật toán chỉ cần sử dụng đúng n lần so sánh và 0 lần đổi chỗ. Trên thực tế, một mảng bất kỳ gồm nhiều mảng con đã được sắp xếp nên thuật toán chèn hoạt động khá hiệu quả. Thuật toán toán sắp xếp chèn là thuật toán nhanh nhất trong các thuật toán sắp xếp cơ bản (đều có độ phức tạp $O(n^2)$).

2.4.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

	A[0]	A[1]	A[2]	A[3]	A[4]
	13	8	2	3	5
$j = 1$	8	13	2	3	5

Vòng lặp $i = 1$, dãy: 13,8,2,3,5. Lưu lại giá trị $saved = A[1] = 8$.

Vòng lặp $j = 1$. So sánh $saved = 8$ và $A[0] = 13$. Do $A[0] > saved$ nên dời $A[0]$ thành $A[1]$, $A[0]$ được gán bằng giá trị của *saved* (tức là giá trị ban đầu của $A[1]$). Dãy trở thành: 8,13,2,3,5.

	A[0]	A[1]	A[2]	A[3]	A[4]
	8	13	2	3	5
$j = 2$	8	2	13	3	5
$j = 1$	2	8	13	3	5

Vòng lặp $i = 2$, dãy: 8,13,2,3,5. Lưu lại giá trị $saved = A[2] = 2$.

Vòng lặp $j = 2$. So sánh $saved = 2$ và $A[1] = 13$. Do $A[1] > saved$ nên dời $A[1]$ thành $A[2]$, $A[1]$ được gán bằng giá trị của *saved* (tức là giá trị ban đầu của $A[1]$). Dãy trở thành: 8,2,13,3,5.

Vòng lặp $j = 1$. So sánh $saved = 2$ và $A[0] = 8$. Do $A[0] > saved$ nên dời $A[0]$ thành $A[1]$, $A[0]$ được gán bằng giá trị của *saved* (tức là giá tri ban đầu của $A[2]$). Dãy trở thành: 2,8,13,3,5.

Vòng lặp $i = 3$**, dãy: 2,8,13,3,5**. Lưu lại giá trị $saved = A[3] = 3$

Vòng lặp $j = 3$. So sánh $saved = 3$ và $A[2] = 13$. Do $A[2] > saved$ nên dời $A[2]$ thành $A[3]$, $A[2]$ được gán bằng giá trị của *saved* (tức là giá trị ban đầu của $A[3]$). Dãy trở thành: 2,8,3,13,5.

Vòng lặp $j = 2$. So sánh $saved = 3$ và $A[1] = 8$. Do $A[1] > saved$ nên dời $A[1]$ thành $A[2]$, $A[1]$ được gán bằng giá trị của *saved* (tức là giá trị ban đầu của $A[3]$). Dãy trở thành: 2,3,8,13,5.

Vòng lặp $j = 1$. So sánh $saved = 3$ và $A[0] = 2$, nhỏ hơn nên giữ nguyên vị trí. Dãy trở thành: 2,3,8,13,5.

Vòng lặp $i = 4$**, dãy: 2,3,8,13,5**. Lưu lại giá trị $saved = A[4] = 5$

Vòng lặp $j = 4$. So sánh $saved = 5$ và $A[3] = 13$. Do $A[3] > saved$ nên dời $A[3]$ thành $A[4]$, $A[3]$ được gán bằng giá trị của *saved* (tức là giá trị ban đầu của $A[4]$). Dãy trở thành: 2,3,8,5,13.

Vòng lặp $j = 3$. So sánh $saved = 5$ và $A[2] = 8$. Do $A[2] > saved$ nên dời $A[2]$ thành $A[3]$, $A[2]$ được gán bằng giá trị của *saved* (tức là giá trị ban đầu của $A[4]$). Dãy trở thành: 2,3,5,8,13.

Vòng lặp $j = 2$. So sánh $saved = 5$ và $A[1] = 3$ nên giữ nguyên vị trí. Dãy trở thành: 2,3,5,8,13.

	A[0]	A[1]	A[2]	A[3]	A[4]
	2	8	13	3	5
$j = 3$	2	8	3	13	5
$j = 2$	2	3	8	13	5
$j = 1$	2	3	8	13	5

	A[0]	A[1]	A[2]	A[3]	A[4]
	2	8	13	3	5
$j = 3$	2	3	8	5	13
$j = 2$	2	3	5	8	13
$j = 1$	2	3	5	8	13

Vòng lặp $j = 1$. So sánh $saved = 5$ và $A[1] = 2$ nên giữ nguyên vị trí. Dãy trở thành: 2,3,5,8,13.

Dừng giải thuật

Vậy sau khi sắp xếp tăng, dãy: 2,3,5,8,13

2.5 Binary Insertion Sort

2.5.1 Ý tưởng thuật toán

Binary Insertion Sort là một thuật toán sắp xếp tương tự như Insertion Sort, nhưng thay vì dùng Linear Search để tìm vị trí cần chèn một phần tử, ta sử dụng Binary Search.

2.5.2 Mã giả

```
Function Binary_Insertion_Sort ( )
    Input: a[ ], n

    for i from 1 to n-1 do
        key = a[i]
        pos = BinarySearchPos(a, i, key)
        j = i
        while (j > pos) do
            swap a[j] and a[j      1]
            j = j      1
        end while
    end for

    Output: sorted array a[ ]
End.
```

2.5.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	n	$O(n)$
Trung bình	$n \log n$	$n \log n$	$O(n \log n)$
Tệ nhất	$n \log n$	$n \log n$	$O(n \log n)$

Binary Insertion Sort được sử dụng để giảm các phép so sánh trong Insertion Sort. Độ phức tạp của thuật toán ở trường hợp tệ nhất là $O(n \log n)$, xảy ra ở trường hợp tương tự Binary Search và Insertion Sort.

2.5.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

- Với $i = 1$: mảng đã sắp xếp là $\{13\}$, mảng chưa sắp xếp là $\{8, 5\}$
  $key = A[1] = 8$ thì $pos = BinarySearchPos(A, 1, 8) = 0$
  Vòng lặp while để chèn 8 vào $A[0]$

- Với $i = 2$: mảng đã sắp xếp là $\{8, 13\}$, mảng chưa sắp xếp là $\{2, 3, 5\}$
  $\{key = A[2] = 2\}$ thì $pos = BinarySearch_pos(A, 2, 2) = 0$
  Vòng lặp while để chèn 2 vào $A[0]$

- Với $i = 3$: mảng đã sắp xếp là $\{2, 8, 13\}$, mảng chưa sắp xếp là $\{3, 5\}$.
  $key = A[3] = 3$ thì $pos = BinarySearch_pos(A, 3, 3) = 1$
  Vòng lặp while để chèn 3 vào $A[1]$

- Với $i = 4$: mảng đã sắp xếp là $\{2, 3, 8, 13\}$, mảng chưa sắp xếp là $\{5\}$
  $key = A[4] = 5$ thì $pos = BinarySearch_pos(A, 4, 5) = 2$
  Vòng lặp while để chèn 5 vào $A[2]$

- Vậy sau khi sắp xếp tăng: $A = \{2, 3, 5, 8, 13\}$

2.6 Bubble Sort

2.6.1 Ý tưởng thuật toán

Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn hoặc lớn hơn trong cặp phần tử đó về đúng ở đầu dãy hiện hành. Sau đó, sẽ không xét đến nó ở bước tiếp theo, do vậy, ở lần xử lý thứ i, phần tử đầu được xét sẽ có vị trí là i. Lặp lại các bước xử lý trên cho đến khi không còn cặp phần tử nào để xét.

2.6.2 Mã giả

```
Function Bubble_Sort()
    Input: a[], n

    for i from 0 to n-2 do
        for i from n-1 to i-1 do
            if(a[j] < a[j - 1])
                swap a[j] and a[j-1]
            end if
        end for
    end for

    Output: sorted array a[ ]
End.
```

2.6.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	0	n	$O(n)$
Trung bình	n	$\frac{n(n-1)}{2}$	$O(n^2)$
Tệ nhất	n	$\frac{n(n-1)}{2}$	$O(n^2)$

Độ phức tạp thuật toán là $\frac{n(n-1)}{2} = O(n^2)$, bằng số lần so sánh và số lần đổi chỗ nhiều nhất của giải thuật (trong trường hợp xấu nhất, khi các phần tử trong mảng được sắp xếp lộn xộn, không tăng dần cũng không giảm dần). Giải thuật sắp xếp nổi bọt là giải thuật chậm nhất trong số các giải thuật sắp xếp cơ bản, nó còn chậm hơn

giải thuật sắp xếp đổi chỗ trực tiếp mặc dù có số lần so sánh bằng nhau, nhưng do đổi chỗ hai phần tử kề nhau nên số lần đổi chỗ nhiều hơn.

2.6.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

	A[0]	A[1]	A[2]	A[3]	A[4]
Ban đầu	13	8	2	3	5
Lần 1	2	13	8	3	5
	2	13	8	3	5
	2	13	3	8	5
	2	3	13	8	5
Lần 2	2	3	13	8	5
	2	3	13	5	8
	2	3	5	13	8
	2	3	5	13	8
Lần 3	2	3	5	13	8
	2	3	5	8	13
	2	3	5	8	13
	2	3	5	8	13

Vậy đã sắp xếp tăng là: 2,3,5,8,13

2.7 Shaker Sort

2.7.1 Ý tưởng thuật toán

Shaker Sort là thuật toán cải tiến của Bubble Sort, không chỉ làm các phần tử nhỏ nhất nổi lên trên mà đồng thời làm các phần tử lớn nhất chìm xuống dưới (dùng kĩ thuật chặn hai đầu để thu hẹp khoảng cần duyệt)

Thuật toán được chia làm hai giai đoạn:

- Giai đoạn đầu sắp xếp các phần tử từ trái sang phải, giống như Bubble Sort. Trong vòng lặp, các phần tử liền kề được so sánh và nếu giá trị bên trái lớn hơn giá trị bên phải, thì các giá trị sẽ được hoán đổi. Vào cuối lần lặp đầu tiên, số lớn nhất sẽ nằm ở cuối mảng.

- Giai đoạn thứ hai kiểm tra mảng theo hướng ngược lại - bắt đầu từ phần tử ngay trước phần tử được sắp xếp lớn nhất của dãy hiện tại và kiểm tra ngược lại phần tử ở đầu mảng. Cũng giống như ở giai đoạn trên, các phần tử liền kề được so sánh và được hoán đổi nếu giá trị bên trái lớn hơn giá trị bên phải.

2.7.2 Mã giả

```
1   Function shaker_sort() {
2       Input: a[], n, left = 0, k = 0, right = n - 1
3
4       while (left < right) do
5           for i from 0 to n-2 do
6               if a[i] > a[i+1]
```

```
                    swap a[i] and a[i+1]
                    k++
                end if
            end for

            right = k

            for i from left+1 to k do
                if (a[i] < a[i-1])
                    swap a[i] and a[i-1]
                    k = i
                end if
            end for

            left = k
        end while

        Output: sorted array a[ ]
    End.
```

2.7.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	2	n	$O(n)$
Trung bình	n	$\frac{n(n-1)}{2}$	$O(n^2)$
Tệ nhất	n	$\frac{n(n-1)}{2}$	$O(n^2)$

2.7.4 Ví dụ minh họa

Giả sử cho dãy $A = \{9,5,1,4,3\}$, dãy có 5 phần tử. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

Lần 1: $left = 0$, $right = 4$, $k = 0$. Vì $left < right$ thỏa mãn điều kiện thực hiện thuật toán

	A[0]	A[1]	A[2]	A[3]	A[4]	k
Ban đầu	9	5	1	4	3	0
$i = 0$	5	9	1	4	3	1
$i = 1$	5	1	9	4	3	2
$i = 2$	5	1	4	9	3	3
$i = 3$	5	1	4	3	9	4

$$right = k = 4$$

	A[0]	A[1]	A[2]	A[3]	A[4]	k
Ban đầu	5	1	4	3	9	4
$i = 4$	5	1	4	3	9	4
$i = 3$	5	1	3	4	9	3
$i = 2$	5	1	3	4	9	3
$i = 1$	1	5	3	4	9	1

$$left = k = 1$$

Lần 2: $left = 1$, $right = 4$. $k = 1$. Vì $left < right$ thỏa mãn điều kiện thực hiện thuật toán

	A[0]	A[1]	A[2]	A[3]	A[4]	k
Ban đầu	1	5	3	4	9	1
$i = 0$	1	5	3	4	9	1
$i = 1$	1	3	5	4	9	2
$i = 2$	1	3	4	5	9	3
$i = 3$	1	3	4	5	9	3

$$left = k = 3$$

	A[0]	A[1]	A[2]	A[3]	A[4]	k
Ban đầu	1	5	3	4	9	1
$i = 0$	1	5	3	4	9	1
$i = 1$	1	3	5	4	9	2
$i = 2$	1	3	4	5	9	3
$i = 3$	1	3	4	5	9	3

$$right = k = 3$$

Lần 3: $left = right = 3$. Dừng thuật toán

Như vậy, dãy tăng dần là: 1, 3, 4, 5, 9

2.8 Shell Sort

2.8.1 Ý tưởng thuật toán

Chia dãy A thành h dãy con

$a_0, a_{0+h}, a_{0+2h}, ..$

$a_1, a_{1+h}, a_{1+2h}, ..$

$a_2, a_{2+h}, a_{2+2h}, ..$

Sắp xếp từng dãy con bằng cách sử dụng phương pháp chèn trực tiếp Insertion Sort.

2.8.2 Mã giả

```
// h = 2
Function Shell_Sort()
    Input: a[], n

    for gap from n/2 to 0 do      // divide into 2
        // Sort all elements in each interval
        for i from gap to n-1 do
            temp = a[i]
            for (j = i to (j >= gap & a[j-gap] > temp) do
                a[i] = a [j-gap]
                j = j-gap
            end for
        end for
        gap = gap/2
    end for

    Output: sorted array a[ ]
End.
```

2.8.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	n	$O(n \log n)$
Trung bình	$n \log n$	$n \log n$	$O(n \log n)$
Tệ nhất	$n \log n$	$n \log n$	$O(n^2)$

Độ phức tạp của Shell Sort phụ thuộc vào khoảng được chọn. Các độ phức tạp trên khác nhau đối với các trình tự gia tăng khác nhau được chọn. Trình tự tăng tốt nhất là không xác định.

Theo định lý Poonen, độ phức tạp trong trường hợp xấu nhất cho kiểu sắp xếp này là $O(\frac{(n \log n)^2}{\log \log n n^2})$ hoặc $O(\frac{(n \log n)^2}{\log \log n})$ hoặc $O(n(\log n)^2)$

2.8.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

A[0]	A[1]	A[2]	A[3]	A[4]
13	8	2	3	5
2	8	13	3	5
2	3	5	8	13

Trong vòng lặp đầu tiên, $n = 5$ thì các phần tử nằm trong khoảng $gap = \frac{n}{2} = 2$ sẽ hoán vị với nhau nếu chúng không theo thứ tự.$A[0]$ được so sánh với $A[2]$, $A[1]$ được so sánh với $A[3]$, ..

Với $gap = \frac{n}{4} = 1$ lúc này ta áp dụng với cả dãy A

Vậy sau khi sắp xếp tăng: $A = \{2, 3, 5, 8, 13\}$

2.9 Heap Sort

2.9.1 Ý tưởng thuật toán

Ý tưởng cơ bản của Heap Sort là thực hiện sắp xếp thông qua việc tạo các Binary Heap. Trong đó, sắp xếp tăng sẽ tạo max heap với key ở nút cha bao giờ cũng lớn hơn ở nút con. Giải thuật này chia làm 2 giai đoạn:

- Bước 1: Tạo heap từ dãy ban đầu. Bây giơ ta thu đươc phân tư A[0] co gia tri lơn nhât trong mang, chinh la node gôc cua heap. Đôi chô phân tư nay vơi phân tư cuôi cung cua mang.
- Bước 2: Sau khi đôi chô, ta tiêp tuc tao max heap mơi với cac phân tư trong mang trư phân tư cuôi cung vi nó đa đươc săp xêp đung vi tri.
- Lặp lại 2 bước trên đến khi tât ca cac phân tư trong mang đươc săp xêp đung theo thư tư tăng dân.

2.9.2 Mã giả

```
Function Heap_Sort ()
    Input: a[ ], n

    for i from n/2 - 1 to 0 do
        heapify(a,n,i)
    end for

    for i from n-1 to 0 do
        swap a[0] and a[i]
        heapify(a,n,0)
    end for

    Output: sorted array a[ ]
End.
```

2.9.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	n	$O(n)$
Trung bình	$n \log n$	$n \log n$	$O(n \log n)$
Tệ nhất	$n \log n$	$n \log n$	$O(n \log n)$

Hàm $heapify()$ có độ phức tạp tỷ lệ với chiều cao của cây heap. Mà một heap có n nút sẽ có chiều cao là $O(\log n)$. Do đó, $heapify()$ có $T(n) = O(\log n)$. Trong hàm $Heap_Sort()$, vòng for lặp n lần nên độ phức tạp của hàm $Heap_Sort()$ là $O(n \log n)$.

2.9.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

- Vòng for đầu tiên dùng để hiệu chỉnh mảng A thành heap thỏa điều kiện $a_i a_{2i+1}$ và $a_i a_{2i+2}$. Sau vòng for, mảng A vẫn giữ như ban đầu $A = \{13, 8, 2, 3, 5\}$
- Vòng for thứ 2: sắp xếp dãy dựa vào heap

Heap	Hoán vị	Loại	Sắp xếp	Ghi chú
13,8,2,3,5	13,5			
5,8,2,3,13		13	13	Lấy 13 ra khỏi heap và thêm vào dãy đã sắp
5,8,2,3	5,8			Hiệu chỉnh lại heap
8,5,2,3	8,3			
3,5,2,8		8	8,13	Lấy 8 ra khỏi heap và thêm vào dãy đã sắp
3,5,2	3,5			Hiệu chỉnh lại heap
5,3,2		5	5,8,13	Lấy 5 ra khỏi heap và thêm vào dãy đã sắp
3,2		3	3,5,8,13	Lấy 3 ra khỏi heap và thêm vào dãy đã sắp
2		2	2,3,5,8,13	Lấy 2 ra khỏi heap và thêm vào dãy đã sắp

- Vậy sau khi sắp xếp tăng: $A = \{2, 3, 5, 8, 13\}$.

2.10 Merge Sort

2.10.1 Ý tưởng thuật toán

Giả sử cho dãy ban đầu $a_1, a_2, .., a_n$. Ta có thể coi dãy đã cho là tập hợp các dãy con có thứ tự. Với dãy có n phần tử, ta phân hoạch thành n dãy con. Vì mỗi dãy con chỉ có 1 phần tử nên nó là dãy có thứ tự.

Merge Sort liên tục chia mảng thành hai nửa cho đến khi trên một mảng con chỉ có 1 phần tử. Sau đó ta merge các mảng đã sắp xếp thành các mảng lớn hơn cho đến khi toàn bộ mảng được hợp nhất.

2.10.2 Mã giả

```
1   Function merge_sort()
2       Input: a[], left, right
3
4       if (left < right) do
5           mid = (left + right)/2
6           merge_sort(a, left, mid)
7           merge_sort(a, mid + 1, right)
8           Merge(a, left, mid, right)
9       end if
10
11      Output: sorted array a[ ]
12  End.
```

2.10.3 Nhận xét thuật toán

- $T(n)$ là thời gian thực hiện Merge Sort trên một dãy n phần tử
- $T(\frac{n}{2})$ là thời gian thực hiện Merge Sort trên một dãy $\frac{n}{2}$ phần tử
- Khi dãy L có độ dài $n = 1$ thì chương trình chỉ thực hiện lệnh trả về có thời gian là $T = C_1$
- Khi dãy L có độ dài $n > 1$ thì chương trình gọi đệ quy hàm Merge Sort hai lần cho hai dãy L_1, L_2 có độ dài là $\frac{n}{2}$ do đó thời gian thực thi sẽ là $2T(\frac{n}{2})$
- Ngoài ra còn phải tốn thời gian cho việc chia danh sách và trộn lại danh sách. Có thể dễ dàng thấy thời gian là $T = C(n, 2)$
- Vậy ta có phương trình đệ quy như sau:

$$\mathrm{T(n)} = \begin{cases} \mathrm{C_1} & n = 1 \quad (2.1a) \\ 2T(\frac{n}{2}) + C_2 n & n > 1 \quad (2.1b) \end{cases}$$

$T(n) = 2T(\frac{n}{2}) + C_2 n$

$T(n) = 2[2T(\frac{n}{4}) + C_2 \frac{n}{2}] = 4T(\frac{n}{4}) + 2C_2 n$

$T(n) = 4[2T(\frac{n}{8}) + C_2 \frac{n}{4}] = 8T(\frac{n}{8}) + 3C_2 n \; ... \; T(n) = 2^i T(\frac{n}{2_i}) + iC_2 n$

- Quá trình thay thế này kết thúc khi $\frac{n}{2_i} = 1$ suy ra $2^i = n, i = \log n$

- Khi đó ta có: $T(n) = nT(1) + \log nC_2n = C_1n + C_2n\log n = O(n\log n)$

	Độ phức tạp
Tốt nhất	$O(n\log n)$
Trung bình	$O(n\log n)$
Tệ nhất	$O(n\log n)$

2.10.4 Ví dụ minh họa

Giả sử cho dãy $A = \{13, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

- Chia đôi dãy ban đầu thành 2 dãy: $A_1 = \{13, 8, 2\}$ và $A_2 = \{3, 5\}$.
- Với A_1 và A_{22} ta vẫn chia đôi thành 2 dãy: $A_{11} = \{13, 8\}$ và $A_{12} = \{2\}$; $A_{21} = \{3\}$ và $A_{22} = \{5\}$
- Tương tự A_{11} thành 2 dãy: $A_{111} = \{13\}$ và $A_{112} = \{8\}$
- Sau đó ta bắt đầu merge 2 dãy A_{111} và A_{112} đã sắp xếp thành 1 dãy đã sắp xếp tăng dần $A_{11} = \{8, 13\}$. Làm tương tự, ta cũng có dãy $A_{22} = \{3, 5\}$
- Tiếp tục merge A_{11} và A_{12} thành $A_1 = \{2, 8, 13\}$
- Và cuối cùng khi merge A_1 và A_2, ta có dãy: $A = \{2, 3, 5, 8, 13\}$

Vậy sau khi sắp xếp tăng: $A = \{2, 3, 5, 8, 13\}$

2.11 Quick Sort

2.11.1 Ý tưởng thuật toán

Ý tưởng của thuật toán như sau: Đây cũng là một hướng tiếp cận "chia nhỏ bài toán"

- Bước 1: Chia dãy a thành hai dãy con bên trái a_{left} và bên phải a_{right} rồi so sánh từng phần tử của a với một phần tử được chọn (gọi là phần tử chốt) và mục đích của chúng ta là đưa phần tử chốt về đúng vị trí của nó.
- Bước 2: Sắp xếp cho các dãy a_{left} và a_{right}
- Bước 3: Nối hai dãy trái và phải với nhau a = $a_{left}a_{right}$

2.11.2 Mã giả

```
1   Function quick_sort()
2       Input: a, left, right
3
4       index = partition(a, left, right)
5       if (left < index - 1) do
6           quick_sort(a, left, index      1)
7       end if
8
9       if (index < right) do
10          quick_sort(a, index, right)
11      end if
12
```

```
13          Output: sorted array a[]
14      End.
15
16      Function partition() {
17          Input: a[], left, right
18
19          mid = a[(left + right) / 2]
20          i = left
21          j = right
22
23          while (i <= j) do
24              while (a[i] < mid) do
25                  i++
26              end while
27              while (a[j] > mid) do
28                  j--
29              end while
30
31              if (i <= j) do
32                  swap(a[i], a[j])
33                  i++
34                  j--
35              end if
36          end while
37
38          Output: i
39      End.
```

Trong đó, hàm *partition*() phân hoạch dãy a trong đoạn từ $left$ đến $right$ sau đó về vị trí *index* dùng để chia đôi mảng, khi chọn phần tử chốt đứng giữa mảng,

2.11.3 Nhận xét thuật toán

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	1	$n \log n$	$O(n \log n)$
Trung bình	1	$n \log n$	$O(n \log n)$
Tệ nhất	1	$\frac{n^2}{2}$	$O(n^2)$

Trường hợp xấu nhất xảy ra khi quá trình phân hoạch luôn chọn phần tử lớn nhất hoặc nhỏ nhất làm trục xoay. Nếu chúng ta xem xét chiến lược phân vùng ở trên trong đó phần tử cuối cùng luôn được chọn làm trục xoay, trường hợp xấu nhất sẽ xảy ra khi mảng đã được sắp xếp theo thứ tự tăng hoặc giảm. Việc này có thể khiến độ phức tạp thuật toán là $O(n^2)$.

$$T(n) = T(0) + T(n-1) + \theta(n) \text{ tương đương với } T(n) = T(n-1) + \theta(n)$$

2.11.4 Ví dụ minh họa

Giả sử cho dãy $A = \{6, 5, 3, 2, 8\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

$$mid = A[\tfrac{left+right}{2}] = A[2] = 3$$

- Phân hoạch đoạn từ $left = 0$ đến $right = 4$:

  $i = 0, j = 4$. $A[i]$ đi trước, tiến về bên phải nếu gặp được phần tử có giá trị lớn hơn *mid* thì dừng lại; ở đây $A[i]$ dừng ở $A[0]$, ghi nhận $i = 0$. $A[j]$ đi sau,

A[0]	A[1]	A[2]	A[3]	A[4]
6	5	3	2	8
2	5	3	6	8

tiến về bên trái nếu gặp được phần tử có giá trị nhỏ hơn *mid* thì dừng lại; ở đây $A[j]$ dừng ở $A[3]$, ghi nhận $j = 3$. Vì $i < j$ nên hoán vị $A[0]$ và $A[3]$, ghi nhận $i = 1, j = 2$.

A[0]	A[1]	A[2]	A[3]	A[4]
2	5	3	6	8
2	3	5	6	8

- Phân hoạch đoạn từ $left = 1$ đến $right = 2$:

  $i = 1, j = 2$. $A[i]$ đi trước, tiến về bên phải nếu gặp được phần tử có giá trị lớn hơn *mid* thì dừng lại; ở đây $A[i]$ dừng ở $A[1]$, ghi nhận $i = 2$. $A[j]$ đi sau, tiến về bên trái nếu gặp được phần tử có giá trị nhỏ hơn *mid* thì dừng lại; ở đây $A[j]$ dừng ở $A[2]$, ghi nhận $j = 1$. Vì $i > j$ nên dừng thuật toán

Vậy sau khi sắp xếp tăng: $A = \{2, 3, 5, 6, 8\}$

2.12 Counting Sort

2.12.1 Ý tưởng thuật toán

Counting Sort là một thuật toán sắp xếp sắp xếp các phần tử của một mảng bằng cách đếm số lần xuất hiện của mỗi phần tử duy nhất trong mảng. Số đếm được lưu trữ trong một mảng phụ và việc sắp xếp được thực hiện bằng cách ánh xạ số đếm như một chỉ số của mảng phụ. Nó dựa trên giả thiết rằng, các khoá cần sắp xếp là các số tự nhiên giới hạn trong một khoảng nào đó, chẳng hạn từ 1 đến n.

2.12.2 Mã giả

```
1   Function CountingSort()
2       Input: a[], n
3
4       max = max(a[])
5       count[max + 1] = [0]
6       for i from 0 to n-1 do
7           count[a[i]]++
8       end for
9
10      for i form 1 to max do
11          count[i] = count[i] + count[i-1]
12      end for
13
14      temp[n]
15
16      for i from 0 to n-1 do
17          temp[count[a[i]] - 1] = a[i]
18          count[a[i]] = count[a[i]] - 1
19      end for
20
21      for i from 0 to n-1 do  //copy data from temp to a
22          a[i] = temp[i]
23      end for
```

```
24
25          Output: Sorted Array a[]
26      End.
```

2.12.3 Nhận xét thuật toán

	Các phép gán	Độ phức tạp
Tốt nhất	$O(n)$	$O(n)$
Trung bình	$O(n)$	$O(n)$
Tệ nhất	$O(n)$	$O(n)$

Có 4 vòng lặp chính trong hàm $counting_sort()$. Vòng lặp thứ nhất, thứ ba và thứ tư lặp n lần với n là kích thước mảng. Vòng lặp thứ hai lặp max lần với max là giá trị lớn nhất trong mảng a.

Trong tất cả trường hợp, mỗi vòng lặp đều chạy giống nhau nên độ phức tạp tổng thể là:

$$T(n) = 3O(n) + O(max) = O(n + max)$$

2.12.4 Ví dụ minh họa

Giả sử cho dãy $A = \{9, 8, 2, 3, 5\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

A[0]	A[1]	A[2]	A[3]	A[4]
9	8	2	3	55

Tìm giá trị lớn nhất của dãy, bằng cách: gán $A[0]$ bằng giá trị lớn nhất, rồi chạy vòng lặp for với i chạy từ 1 đến $n-1$, xét từng phần tử với giá trị lớn nhất hiện tại, nếu lớn hơn thì gán vào max. Thực hiện lần lượt, thu được kết quả: $max = 9$

Khởi tạo mảng $count[]$ có kích thước là $max + 1 = 10$ và gán cho tất các giá trị của các phần tử có trong dãy đó đều bằng 0.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Lưu trữ tổng tích lũy của các phần tử trong mảng count. Giá trị của các phần tử trong mảng count cũng là vị trí chính xác mà các phần tử của mảng A sẽ đứng sau khi sắp xếp.

Ta đặt từng giá trị của mảng A vào đúng vị trí khi đã sắp xếp của nó dựa vào mảng count. Với $A[0] = 9$, vị trí của $A[0]$ là $count[A[0]] - 1 = 4$ nên $temp[4] = 9$.

Sau khi đặt mỗi phần tử vào đúng vị trí của nó, giảm số lượng của nó 1 đơn vị. $count[A[0] = count[A[0] - 1 = 4$

Thực hiện tương tự cho các phần tử còn lại.

Vậy sau khi sắp xếp tăng: $A = \{2, 3, 5, 8, 9\}$

0	1	2	3	4	5	6	7	8	9
0	0	1	2	2	3	3	3	4	5

0	1	2	3	4
2	3	5	8	9

2.13 Radix Sort

2.13.1 Ý tưởng thuật toán

Các phương pháp sắp xếp trước đây chỉ sử dụng phương pháp so sánh thử và sai

Radix Sort là phương pháp sắp xếp sử dụng thông tin của khóa.

Cụ thể là phương pháp này sẽ nhóm các khóa có cùng chữ số thứ *i* thành một nhóm đây là bước phân bố

Bước kế tiếp là kết hợp các nhóm này lại thành một dãy duy nhất. Quá trình phân bố-kết hợp được thực hiện cho đến khi hết các chữ số

Vậy có hai phương pháp chính: Least significant digit đi từ phải qua trái và Most significant digit đi từ trái qua phải

2.13.2 Mã giả

Trong đó, hàm partition( ) phân hoạch dãy a trong đoạn từ left đến right sau đó về vị trí index dùng để chia đôi mảng, khi chọn phần tử chốt đứng giữa mảng,

```
Function radix_sort()
    Input: a, n, b[n], exp= 1

    m = max(a,n) // MAX a from 1 to n
    while (m / exp > 0) do
        bucket[10] = [0]
        for (i = 0 to n-1) do
            bucket[a[i] / exp % 10]++
        end for

        for (i = 1 to 9) do
            bucket[i] += bucket[i - 1]
        end for

        for (i = n-1 to 0) do
            b[--bucket[a[i] / exp % 10]] = a[i]
        end for

        for (i = 0 to n-1) do
            a[i] = b[i]
        end for

        exp = exp*10    //di chuyn  n  ch  s  tip theo
    end while

    Output: sorted array a[]
End.
```

	Các phép gán	Các phép so sánh	Độ phức tạp
Tốt nhất	n	n	$O(n)$
Trung bình	n	n	$O(n)$
Tệ nhất	n	n	$O(n)$

2.13.3 Nhận xét thuật toán

Xét một mảng gồm n phần tử, các phần tử trong mảng có tối đa k chữ số thì số lần chia nhóm các phần tử là k lần. Trong mỗi lần chia nhóm và gộp lại thành mảng, các phần tử chỉ được xét đúng 1 lần. Vì vậy, độ phức tạp của thuật toán Radix sort là $O(2kn) = O(n)$

2.13.4 Ví dụ minh họa

Giả sử cho dãy $A = \{103, 68, 233, 37, 51\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

Với m là số lớn nhất trong mảng nên $m = 233$. Kết quả là vòng lặp sẽ lặp 3 lần.

Với $exp = 1$, ta so sánh chữ số hàng đơn vị của dãy A

Vòng for thứ nhất, ta chia mảng A thành 10 nhóm có cùng chữ số hàng đơn vị và lưu số lượng phần tử của các nhoám đó vào mảng bucket.

Bucket	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	0	1	0	2	0	0	0	1	1	0

Vòng for thứ hai, $bucket[i] = bucket[i] + bucket[i-1]$

Bucket	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	0	1	1	3	3	3	3	4	5	5

Vòng for thứ ba, xếp các phần tử của mảng A tăng dần theo chữ số hàng đơn vị vào mảng bucket

Vòng for thứ tư, sao chép mảng b[] sang mảng A[]

Với $exp = 10$, ta làm tương tự các bước trên cho hàng chục.

Với $exp = 10$, ta làm tương tự các bước trên cho hàng trăm.

Vậy sau khi sắp xếp tăng: $A = \{37, 51, 68, 103, 233\}$

2.14 Flash Sort

2.14.1 Ý tưởng thuật toán

Ý tưởng cơ bản đằng sau flashsort là trong tập dữ liệu có phân bố đã biết, có thể dễ dàng ước tính ngay vị trí cần đặt phần tử sau khi sắp xếp khi phạm vi của tập hợp đã biết. Gồm có 3 bước:

Bucket	[0]	[1]	[2]	[3]	[4]
	51	233	103	37	68

A	[0]	[1]	[2]	[3]	[4]
	51	233	103	37	68

- Bước 1: Phân lớp dữ liệu, tức là dựa trên giả thiết dữ liệu tuân theo 1 phân bố nào đó, chẳng hạn phân bố đều, để tìm 1 công thức ước tính vị trí (lớp) của phần tử sau khi sắp xếp
- Bước 2: Hoán vị toàn cục, tức là dời chuyển các phần tử trong mảng về lớp của mình.
- Bước 3: Sắp xếp cục bộ, tức là để sắp xếp lại các phần tử trong phạm vi của từng lớp. Sau bước hoán vị toàn cục, mảng của chúng ta hiện tại sẽ được chia thành các lớp(thứ tự các phần tử trong lớp vẫn chưa đúng) do đó để đạt được trạng thái đúng thứ tự thì khoảng cách phải di chuyển của các phần tử là không lớn vì vậy Insertion Sort sẽ là thuật toán thích hợp nhất để sắp xếp lại mảng có trạng thái như vậy.

2.14.2 Mã giả

```
Function flash_sort()
    Input: a[], n

    // Layering Data
    MinVal = min(a[])    // The smallest value in the
    Max = max(a[])       // The location of the smallest value in the
array       m = 0.45*n
    L[m] = [0]
    if (a[max] = minVal) end function.
    c1 = (m - 1) / (a[max] - minVal)

    for (i = 0 to n-1) do
        k = c1 * (a[i] - minVal)
        L[k] = L[k]+1
    end for

    for (i = 1 to m) do
        L[i] = L[i] +L[i-1]
    end for

    swap(a[max], a[0])

    // Hoan vi toan cuc
    nmove = 0, j = 0, t = 0, k = m-1

    while (nmove < n - 1) do
        while (j > l[k] - 1) do
            j = j+1
            k = c1*(a[j] - minVal)
        end while
        flash = a[j]
        if (k < 0) end while

        while (j != L[k]) do
            k = c1*(flash - minVal)
            L[k] = L[k] -1
```

```
36                  swap(flash,a[L[k]])
37                  nmove = nmove +1
38              end while
39          end while
40
41          // Sap xep cuc bo
42          insertion_sort(a, n)
43
44          Output: Sorted array a[]
45      End.
```

2.14.3 Nhận xét thuật toán

	Các phép so sánh	Độ phức tạp
Tốt nhất	$O(n)$	$O(n)$
Trung bình	$O(n)$	$O(n)$
Tệ nhất	$O(n)$	$O(n)$

Trường hợp xấu nhất xảy ra khi quá trình phân hoạch luôn chọn phần tử lớn nhất hoặc nhỏ nhất làm trục xoay. Nếu chúng ta xem xét chiến lược phân vùng ở trên trong đó phần tử cuối cùng luôn được chọn làm trục xoay, trường hợp xấu nhất sẽ xảy ra khi mảng đã được sắp xếp theo thứ tự tăng hoặc giảm. Việc này có thể khiến độ phức tạp thuật toán là $O(n^2)$.

Nhìn lại toàn bộ các giai đoạn của thuật toán, ta thấy như sau:

- Bước phân lớp dữ liệu đòi hỏi độ phức tạp $O(n)$ và $O(m)$
- Bước hoán vị toàn cục đòi hỏi độ phức tạp $O(n)$ (vì mỗi phần tử chỉ phải đổi chỗ đúng một lần, và n lần cho n phần tử)
- Bước sắp xếp cục bộ đỏi hỏi độ phức tạp $O(\frac{n^2}{m})$ (mỗi 1 phân lớp đòi hỏi độ phức tạp $O((\frac{n}{m})^2)$ và m phân lớp đòi hỏi $O(m * (\frac{n}{m})^2))$

$$T(n) = a_1 n + a_2 m + a_3 \frac{n^2}{m}$$

$$T'(n) = a_2 - a_3 (\frac{n}{m})^2$$

$$m = n(\frac{a_3}{a_2})^{\frac{1}{2}}$$

2.14.4 Ví dụ minh họa

Giả sử cho dãy $A = \{54, 76, 3, 26, 5, 83\}$. Hãy sắp xếp dãy số đã cho theo thứ tự tăng dần. Các bước giải thuật toán sẽ được thực hiện như sau:

Bước 1: Phân lớp dữ liệu

- $max = 83, minVal = 3, m = 3$
- $c1 = \frac{m-1}{a[max]-minVal} = \frac{2}{83-3} = \frac{1}{40}$
- Khởi tạo mảng L có m phần tử ứng với m lớp

- Đếm số lượng phần tử các lớp, $a[i]$ sẽ thuộc lớp$k = int((m-1) * (a[i] - minVal)$

L	0	1	2
	4	2	1

Tính vị trí kết thúc của từng phân lớp

Hoán vị $A[max]$ và $A[0]$

Bước 2: hoán vị toàn cục: tính từng phần tử của a phải nằm ở phân lớp nào, rồi bỏ nó vào vị trí của phân lớp đó.

- $flash = A[j] = A[0] = 83$ thì $k = int(c_1 * (flash - minVal)) = int(\frac{1}{40} * (83 - 3)) = 2$ nên ta hoán đổi $flash$ cho $a[L[k] - 1] = A[6]$
- $flash = A[j] = A[0] = 54$ thì $k = int(c_1 * (flash - minVal)) = int(\frac{1}{40} * (54 - 3)) = 1$ nên ta hoán đổi $flash$ cho $a[L[k] - 1] = A[5]$
- $flash = A[j] = A[0] = 5$ thì $k = int(c_1 * (flash - minVal)) = int(\frac{1}{40} * (5 - 3)) = 0$ nên ta hoán đổi $flash$ cho $a[L[k] - 1] = A[3]$
- Tiếp tục như vậy cho đến khi tất cả phần tử của a nằm đúng nhóm phân lớp của chúng

Bước 3: sắp xếp cục bộ:

- Nhìn tổng thể, ta thấy rằng mảng a đã được sắp xếp tăng dần theo từng nhóm phân lớp. Nghĩa là, mọi phần tử thuộc phân lớp 0 đều bé hơn phần tử ở phân lớp 1 và mọi phần tử thuộc phân lớp 1 đều bé hơn phần tử ở phân lớp 2.
- Tuy nhiên, thứ tự các phần tử trong lớp vẫn chưa đúng. Vì vậy, ta có thể dùng Insertion Sort để sắp xếp lại mảng a theo đúng thứ tự do khoảng cách phải di chuyển của giữa các phần tử trong lớp là không lớn

Vậy sau khi sắp xếp tăng: $A = \{3, 5, 26, 37, 54, 76, 83\}$

Chương 3

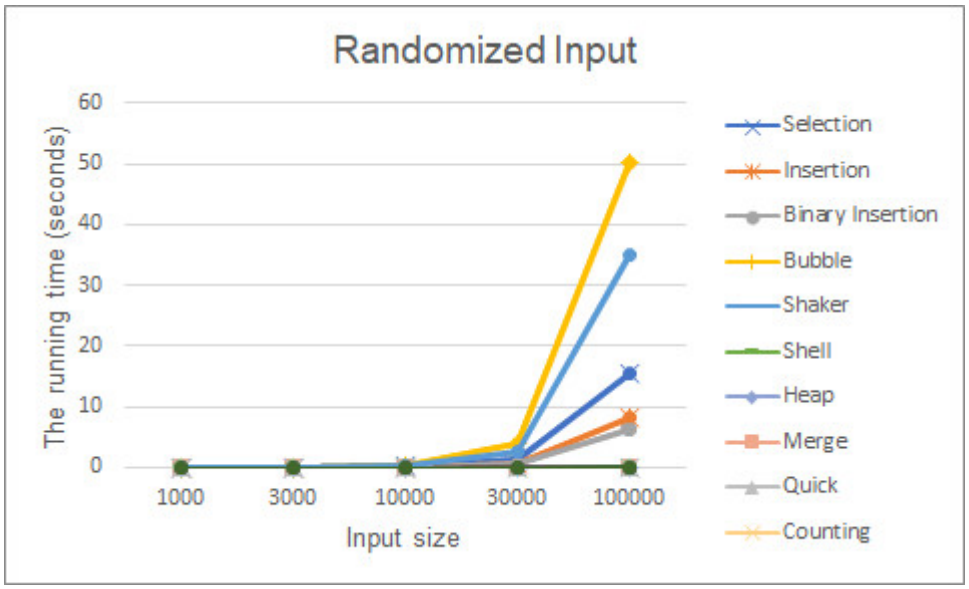
Đánh giá thuật toán

3.1 Nội dung cài đặt

Source Code và File cài đặt đã được đính kèm trong thư mục nộp bài

3.2 Đánh giá

3.2.1 Mảng ngẫu nhiên



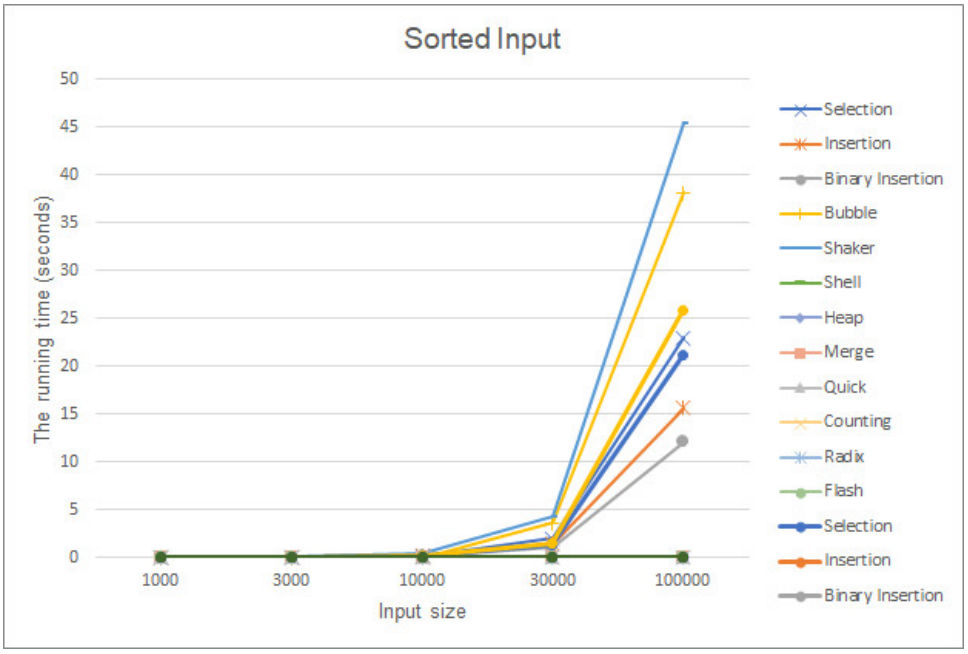
Thời gian chạy của các thuật toán tăng dần, tỉ lệ thuận với kích thước của mảng.

Trong mảng random có kích thước 100000 phần tử, 3 thuật toán có thời gian chạy chậm nhất là Bubble Sort, Shaker Sort và Selection Sort với thời gian lần lượt là 50.248, 35.099 và 15.389 giây. Bên cạnh đó, 3 thuật toán có thời gian chạy nhanh nhất là Flash sort,Counting Sort và Radix Sort với thời gian đều xấp xỉ 0 giây ở mọi kích thươc phần tử được so sánh.

Lý giải cho điều này: 3 thuật toán Bubble Sort, Shaker Sort và Selection Sort đều có độ phức tạp là $O(n^2)$ còn Flash sort, Counting Sort và Radix Sort đều có độ phức tạp là O(n). Thế nhưng Bubble Sort, Shaker Sort và Selection Sort vẫn đuọc sử dụng rộng rãi vì tính thông dụng, dễ cài đặt. Counting Sort và Radix Sort lại ít được sử

dụng do chúng chỉ sắp xếp được những số không âm - 1 hạn chế rất lớn ở các thuật toán sắp xếp.

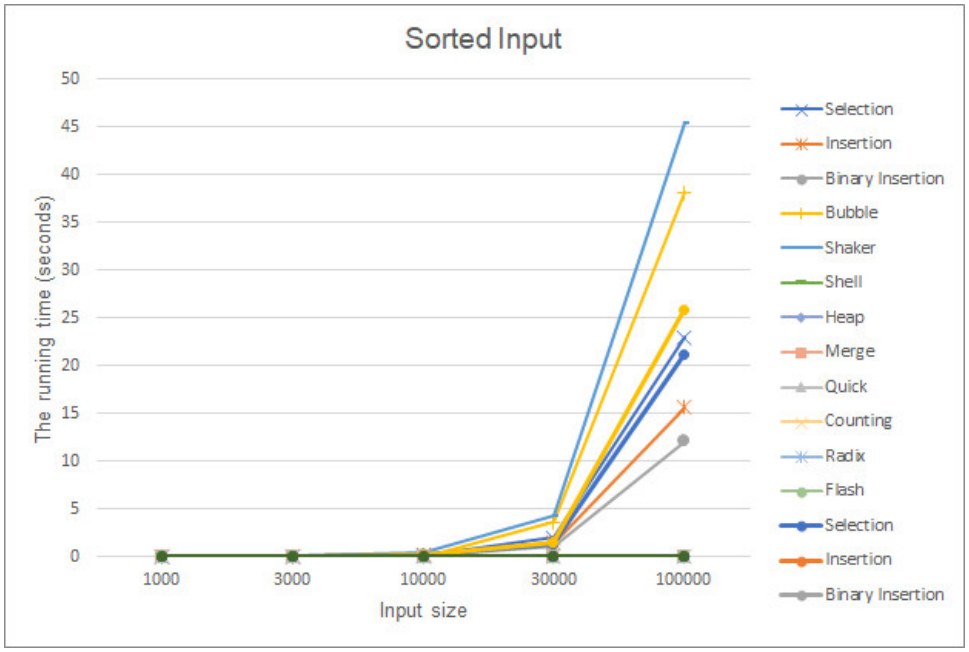
3.2.2 Mảng đã sắp xếp



Trong trường hợp mảng đã được sắp xếp, có 2 thuật toán có thời gian chạy chậm nhất là Bubble Sort và Selection Sort với thời gian lần lượt là 25.876 và 21.208 giây. Tất cả các thuật toán còn lại thời gian chạy cho cả mảng có 100000 dều chưa tới 0.05 giây.

Lý giải cho điều này: Bubble Sort và Selection Sort chưa nhận diện được mảng đã sắp xếp mà phải chạy tuần tự từng phần tử. Shaker Sort là thuật toán được cải tiến từ Bubble Sort và với mảng đã được sắp xếp thì sự cải tiến đó đã thể hiện một cách rõ rệt.

3.2.3 Mảng sắp xếp ngược



Trong trường hợp mảng được sắp xếp ngược có 3 thuật toán chậm nhất là Shaker Sort, Bubble Sort và Selection Sort với thời gian lần lượt là 45.44, 38.17 và 23.002 giây. 2 thuật toán có thời gian chạy nhanh nhất là Flash sort và Quick Sort đều cho thời gian ở tất cả dữ liệu đầu vào là 0 giây.

điều này cho thấy: cả Flash sort Quick Sort rất hiệu quả đối với mảng ngược. Tuy nhiên, trái ngược với Quick Sort thì Flash sort lại rất ít được sử dụng vì cách cài đặt phức tạp dù độ phức tạp về thời gian là O(n).

3.2.4 Kết luận

Không nên áp dụng 1 thuật toán sort cho mọi bài toán. Tùy vào bài toán với bối cảnh cụ thể, với giới hạn về thời gian, tài nguyên và đặc điểm dữ liệu mà ta có thể áp dụng các thuật toán sort khác nhau. đối với những bài toán đơn giản như sắp xếp danh sách theo điểm trung bình của 1 lớp học thì ta nên sử dụng Bubble Sort hay Selection Sort vì chúng vừa đơn giản vừa dễ cài đặt. Ta không nhất thiết phải sử dụng các thuật toán top chạy nhanh vì hầu hết chúng rất phức tạp, cài đặt dễ sai.

Phụ lục A

Tài liệu tham khảo

PGS.TS Đỗ Văn Nhơn và ThS. Trịnh Quốc Sơn (2015). *Giáo trình Cấu trúc Dữ liệu và Giải thuật*, Nxb. Đại học Quốc Gia TP. Hồ Chí Minh, TP Hồ Chí Minh

Walls and Mirrors (2014). *Data Abstraction And Problem Solving with C++*

Bùi Tiến Lên (2021). *Elementary Sorting Methods*, Khoa Công nghệ Thông tin - Trường Đại học Khoa học Tự Nhiên - ĐHQG TP.HCM, TP Hồ Chí Minh. https://drive.google.com/drive/folders/1mTuksP2rlwrwGBORXWFiLyhPJpqoYHKU (10/2022)

Geeksforgeeks (10/03/2022). *Sorting Algorithm*. URL: https://www.geeksforgeeks.org/sorting-algorithms/?ref=lbp (10/2022)