ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO THỰC HÀNH

MÔN HỌC: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

CHỦ ĐỀ: TÌM HIỂU VÀ THỰC HÀNH CÁC THUẬT TOÁN SẮP XẾP

NHÓM 02

21120499 - Nguyễn Duy Long

21120504 - Nguyễn Phương Nam

21120505 - Bùi Thị Thanh Ngân

21120521 - Nguyễn Phúc Phát

Giảng viên hướng dẫn: Thầy Lê Đình Ngọc

--> -->

MỤC LỤC

1️⃣ Selection Sort

1. Ý tưởng thuật toán:

```
Mỗi bước sẽ di chuyển một phần tử nhỏ nhất sang bên trái, từ đó mảng sẽ dần được
chia làm 2 phần:

Bên trái là mảng đã được sắp xếp tăng dần.

Bên phải là mảng chưa được sắp xếp.
```

2. Psuedocode:

```
For i: 0 đến n – 2

Tìm min_index là phần tử nhỏ nhất trong khoảng i đến n – 1

Swap a(i) và a(min_index).
```

3. Độ phức tạp:

```
Không gian: O(1) vì các phép toán được thực hiện trên mảng ban đầu.

Thời gian: O(n2) cho tất cả các trường hợp vì các vòng lặp for đều phải chạy đủ.
```

4. Biến thể và cải tiến:

```
Heap sort sử dụng cùng ý tưởng tìm các giá trị lớn nhất nhỏ nhất, nhỏ nhất nhưng
dùng đến cấu trúc heap nên độ phức tạp giảm còn O(nlogn).

Double selection sort, tìm cùng lúc giá trị lớn nhất và nhỏ nhất sau đó di chuyển
các giá trị này đến đầu và cuối mảng.
```

2️⃣ Insertion Sort

1. Ý tưởng thuật toán:

```
Mỗi bước sẽ di chuyển một phần tử nhỏ nhất sang bên trái, từ đó mảng sẽ dần được
chia làm 2 phần:
```

2. Psuedocode:

```
For i: 1 đến n – 1
```

3. Độ phức tạp:

```

```

4. Biến thể và cải tiến:

```

```

3️⃣ Bubble Sort

1. Ý tưởng thuật toán:

```
xuất phát từ cuối mảng, đổi chỗ các 2 phần tử gần nhau để đưa phần tử nhỏ hơn
trong cặp phần tử đó về vị
trí đứng đầu dãy hiện hành sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở
lần xử lý thứ i sẽ có vị
trí đầu dãy là i. Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét
```

2. Psuedocode:

```
 n <- độ dài của mảng

   for i = 0 to n-1 do:

      for j = n-1 downto 0 do:

Tiến hành so sánh 2 phần tử kề cận nhau

         if list[j] > list[j+1] then

Đưa phần tử nhỏ hơn ra phía trước

            swap( list[j], list[j+1] )

         end if

      end for
```

3. Độ phức tạp:

```
Không gian: O(1) vì các phép toán được thực hiện trên mảng ban đầu.

Thời gian: O(n2) ngay cả khi mảng đã được sắp xếp.
```

4. Biến thể và cải tiến:

```
Trong mỗi vòng lặp của biến j ở trên, kiểm tra xem nếu không có phép hoán vị nào
được thực hiện tức mảng đã đúng vị trí ta sẽ dừng thuật toán ngay lập tức. Trong
trường hợp tốt nhất mảng đã được sắp xếp độ phức tạp về thời gian là 0(n)
```

4 Shaker Sort

1. Ý tưởng thuật toán:

```
Mỗi bước sẽ di chuyển một phần tử nhỏ nhất sang bên trái, từ đó mảng sẽ dần được
chia làm 2 phần:
```

2. Psuedocode:

```
For i: 1 đến n – 1
```

3. Độ phức tạp:

```

```

4. Biến thể và cải tiến:

```

```

5 Shell Sort

1. Ý tưởng thuật toán:

```
shell sort là một biến thể cải tiến hơn của insertion sort.  Thuật toán sử dụng
insertion sort lên các phần tử cách xa nhau sau đó thu hẹp dần khoảng cách này.
Như vậy mảng sẽ được chia thành các mảng con với các phần tử có khoảng cách là h,
sắp xếp các mảng con này bằng insertion sort và lặp lại các bước trên với khoảng
cách thu hẹp dần thì ta được mảng có thứ tự.
```

2. Psuedocode:

```
Chọn h = n / 2

While h >  0:

For i: từ h đến n – 1:

(Selection sort cho mảng từ 0 – i với bước chạy là h)\

Chọn temp = a(i)

Đẩy tất cả các phần tử lớn hơn temp lên h đơn vị

Chèn temp vào trị ví thích hợp

h / 2
```

3. Độ phức tạp:

```
Không gian: sắp xếp tại chỗ nên là O(1)

Thời gian: độ phức tạp của shell sort tùy thuộc vào h ta chọn, với những h thích
hợp ta có thể tối ưu shell sort hơn nữa. Với h = h/2 ta chọn ở trên:

Trường hợp tệ nhất là khi shell sort trở thành insertion sort, lúc này độ phức tạp
thời gian là O(n2)

Trường hợp tốt nhất là khi mảng đã được sắp xếp sẵn thì độ phức tạp sẽ là O(nlogn)

Trung bình thời gian chạy của shell sort sẽ là O(nlogn)
```

4. Biến thể và cải tiến:

```
Dobosiewicz sort

Shaker sort

Insertion sort
```

6 Quick Sort

1. Ý tưởng thuật toán:

2. Psuedocode:

3. Độ phức tạp:

4. Biến thể và cải tiến:

7 Heap Sort

1. Ý tưởng thuật toán:

2. Psuedocode:

3. Độ phức tạp:

4. Biến thể và cải tiến:

8 Merge Sort

1. Ý tưởng thuật toán:

```
sử dụng thuật toán chia để trị xử lý 2 nữa mảng rồi trộn chúng lại với nhau. Cụ
thể:

Mảng được chia đôi thành 2 phần, ta sắp xếp 2 phần mảng này trước.

Trộn 2 mảng đã có thứ tự này lại bằng cách lấy lần lượt số nhỏ nhất ở đầu 2 mảng
bỏ vào mảng chính.
```

2. Psuedocode:

```
Hàm trộn 2 mảng có thứ tự left và right vào mảng chính:

While 2 cả mảng left và right còn phần tử:

Lấy lần lượt phần ở đầu 2 mảng

So sánh 2 phần tử này

Phần tử nào nhỏ hơn thì lấy bỏ vào mảng chính

Nếu mảng left còn phần tử:

Bỏ phần còn lại đó vào phía sau mảng chính

Nếu mảng right còn phần tử:

Bỏ phần còn lại đó vào phía sau mảng chính

Hàm mergeSort:

Nếu mảng có n <= 1 thì dừng

Chia mảng a thành 2 phần bằng nhau là

Mảng left
```

```
Mảng right

Thực hiện gọi đệ qui sắp xếp 2 mảng này

Gọi hàm trộn 2 mảng này về mảng chính
```

3. Độ phức tạp:

```
Không gian:O(n) sử dụng để lưu trữ 2 mảng con.

Thời gian:. Thời gian chạy của merge sort khá ổn định, trong tất cả các trường hợp đều là O(nlogn).
```

4. Biến thể và cải tiến:

```
các biến thể của merge sort chủ yếu tập trung vào việc giảm độ phức tạp về không gian và giảm số lần copy phần tử

Block sort: là một in-place sorting với độ phức tạp ổ định là O(nlogn)

Katajainen et al: cũng là một in-place sorting với độ phức tạp O(nlogn) chưa được ổn định
```

9 Radix Sort

1. Ý tưởng thuật toán:

```
Khác với các thuật toán trước, Radix sort là một thuật toán tiếp cận theo một hướng hoàn toàn khác. Nếu như trong các thuật toán khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử thì Radix sort lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó nó còn có tên là Postman’s sort. Nó không hề quan tâm đến việc so sánh giá trị của phần tử và bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

Coi các phần tử trong mảng sắp xếp được cấu thành từng các lớp có độ ưu tiên khác nhau. Ví dụ, các số tự nhiên chia thành các lớp như: hàng đơn vị, hàng chục, hàng trăm, hàng nghìn,

Bước đầu tiên ta sắp xếp dãy các phần tử bằng cách so sánh các phần tử ở lớp có độ ưu tiên thấp nhất (ví dụ các chữ số hàng đơn vị). Số nào có hàng đơn vị thấp hơn thì ta đưa lên trên. Như vậy các số có hàng đơn vị là 0 ở trên cùng, sau đó đến các số có hàng đơn vị là 1,…

Sau bước 1, ta thu được 1 thứ tự sắp xếp mới. Ta lại làm tương tự với các lớp kế
```

```
tiếp (chữ số thuộc hàng chục, hàng trăm,…)cuối cùng ta sẽ có dãy đã sắp xếp.
```

2. Psuedocode:

```
max <- số chữ số của phần tử lớn nhất

table <- mảng các queue gồm 10 phần tử( các lớp từ 0->9)

for k = 0 to max-1 do:

for i = 0 to i-1 do:

unit <- chữ số của hàng thứ k

thêm a[i] vào queue table[unit]

end for

i = 0

j = 0

Gán các phần tử trong queue vào mảng theo thứ tự các lớp từ 0 ->9

while j < 10 do:

while table[j] có phần tử do:

a[i] <- lấy ra phần tử đầu trong queue

i = i +1

j = j + 1

end for
```

3. Độ phức tạp:

```
Không gian: O(n) vì sử dụng hàng đợi để lưu các phần tử.

Thời gian: O(max*n) = O(n) như nhau trên mọi trường hợp, không có trường hợp xấu
nhất lẫn tốt nhất

Nhận xét: Thuật toán với độ phức tạp tuyến tính nên rất thích hợp cho việc sắp xếp
các dãy có rất nhiều phần tử, nhất là khi các khóa sắp xết không quá dài so với số
lượng phần tử
```

4. Biến thể và cải tiến:

🔟 Counting Sort

1. Ý tưởng thuật toán:

2. Psuedocode:

3. Độ phức tạp:

4. Biến thể và cải tiến:

1️⃣1️⃣ Flash Sort

1. Ý tưởng thuật toán:

```
Tư tưởng chính của thuật toán là dựa trên sự phân lớp phần tử (Subclasses
Arrangement). FlashSort bao gồm ba khối logic: Phân loại các phần tử (Elements
Classification); Phân bố các phần tử vào đúng các phân lớp (Elements Permutation);
Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự (Elements Ordering).
```

2. Psuedocode:

3. Độ phức tạp:

```
Không gian: O(1) vì thực hiện tại chỗ.

Thời gian: có độ phức tạp trung bình là O(n) .Gọi m là số lớp, trường hợp tốt nhất
mỗi lớp gần như có cùng kích thước độ phức tạp là m*O(1). Trường hợp xấu nhất tất
cả các phần tử đều nằm trong chỉ 1 vài nhóm độ phức tạp là O(n2).  Vậy nên theo
khảo sát thì số phân lớp tối ưu nhất là m = 0.45n với n là số lượng phần tử của
mảng.
```

4. Biến thể và cải tiến:

```

```

IV. Kết quả thực nghiệm

Bảng thống kê

Data order: Randomized Data - Running time (ms)

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:
11. Flash Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Randomized Data - Comparisons

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Sorted Data - Running time (ms)

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Sorted Data - Comparisons

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Reverse Data - Running time (ms)

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Reverse Data - Comparisons

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Nearly Sorted Data - Running time (ms)

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Data order: Nearly Sorted Data - Comparisons

Data size	**10.000**	**30.000**	**50.000**	**100.000**	**300.000**	**500.000**
1. Selection Sort	:---:	:---:	:---:	:---:	:---:	:---:
2. Insertion Sort	:---:	:---:	:---:	:---:	:---:	:---:
3. Bubble Sort	:---:	:---:	:---:	:---:	:---:	:---:
4. Shaker Sort	:---:	:---:	:---:	:---:	:---:	:---:
5. Shell Sort	:---:	:---:	:---:	:---:	:---:	:---:
6. Heap Sort	:---:	:---:	:---:	:---:	:---:	:---:
7. Merge Sort	:---:	:---:	:---:	:---:	:---:	:---:
8. Quick Sort	:---:	:---:	:---:	:---:	:---:	:---:
9. Counting Sort	:---:	:---:	:---:	:---:	:---:	:---:
10. Radix Sort	:---:	:---:	:---:	:---:	:---:	:---:

Đồ thị minh họa

Nhận xét chung

V. Tổ chức đồ án và các lưu ý

VI. Tài liệu tham khảo