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

\newpage
\pagebreak

MỤC LỤC

3.11. Flash Sort

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 to n - 2

        min_index = chỉ số phần tử nhỏ nhất trong khoảng i + 1 đến n - 1

        swap (a(i), a(min_index))
```

3. Độ phức tạp:

```
    Không gian: O(1)

    Thời gian: O(n^2)
```

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:

```
    1. Chèn phần tử thứ hai vào vị trí thích hợp trong mảng con đã được sắp xếp.
    2. Chèn phần tử thứ ba vào vị trí thích hợp trong mảng con đã được sắp xếp.
    3. Lặp lại cho đến khi chèn phần tử cuối cùng vào vị trí thích hợp trong mảng
con đã được sắp xếp.
```

2. Psuedocode:

```
     for i = 1 to n - 1
        x = a[i]
        j = i - 1
        Duyệt j và tìm vị trí thích hợp cho x, đồng thời dịch các phần tử sang
phải để tạo chỗ cho x
        Chèn x vào vị trí thích hợp
```

3. Độ phức tạp:

```
        Không gian: O(1)

        Thời gian: O(n^2)

        Trung bình, thuật toán sắp xếp chèn – Insertion sort có độ  phức tạp là
O(n^2)
        Trường hợp tốt nhất là với đầu vào đã được sắp xếp đúng thứ tự. Trường hợp
xấu là dãy bị đảo ngược thứ tự hoàn toàn.
```

4. Biến thể và cải tiến:

```
    Shell sort sử dụng cùng ý tưởng tìm các giá trị lớn nhất nhỏ nhất.
    Áp dụng trong flash sort.
    Có thể dùng binary search để giảm số lần so sánh.
```

3️⃣ Bubble Sort

1. Ý tưởng thuật toán:

```
    1. So sánh 2 phần tử liền kề, nếu phần tử đứng trước lớn hơn phần tử đứng sau
thì hoán đổi chỗ 2 phần tử này.
    2. Lặp lại cho đến khi không còn phần tử nào cần hoán đổi chỗ.
```

2. Psuedocode:

```
        for i = 0 to n - 2
            for j = n - 1 downTo i + 1
                if a[j] < a[j - 1]
                    swap (a[j], a[j - 1])
```

3. Độ phức tạp:

```
    Không gian: O(1)

    Thời gian: O(n^2) đối với trường hợp tệ nhất, O(n) đối với trường hợp tốt
nhất.
```

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 O(n)

    Biến thể là Recursive Bubble Sort và Shaker Sort.
```

4 Shaker Sort

1. Ý tưởng thuật toán:

```
    1. So sánh 2 phần tử liền kề, nếu phần tử đứng trước lớn hơn phần tử đứng sau
thì hoán đổi chỗ 2 phần tử này.
    2. Lặp lại cho đến khi không còn phần tử nào cần hoán đổi chỗ.
    3. Lặp lại bước 1 và 2 nhưng lần này so sánh từ phải sang trái.
```

2. Psuedocode:

```
    for i = 0 to n - 2
        // Trong mỗi vòng lặp, ta sẽ tìm được 1 phần tử đúng vị trí
        // Duyệt từ trái sang phải
        for j = n - 1 to i + 1
            if a[j] < a[j - 1]
                swap (a[j], a[j - 1])
```

```
        // Duyệt từ phải sang trái
        for j = i + 1 to n - 1
            if a[j] < a[j - 1]
                swap (a[j], a[j - 1])
```

3. Độ phức tạp:

```
    Không gian: O(1)

    Thời gian: O(n^2)
    Có độ phức tạp tương tự như Bubble Sort nhưng có thể tối ưu hơn về thời gian
trong trường hợp tốt nhất.
```

5️⃣ 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:

```
    h = n / 2

    While h >  0:

        For i = h đến n – 1:

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

            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:

```
    1. Chọn một phần tử làm pivot.
    2. Đưa các phần tử nhỏ hơn pivot về bên trái pivot, các phần tử lớn hơn pivot
về bên phải pivot.
    3. Lặp lại bước 1 và 2 cho đến khi không còn phần tử nào cần sắp xếp.
```

2. Psuedocode:

```
  quickSort(a, l, r)
    if l < r
        p = partition(a, l, r)
        quickSort(a, l, p - 1)
        quickSort(a, p + 1, r)

    partition(a, l, r)
```

```
    pivot = a[r]
    i = l - 1
    for j = l to r - 1
        if a[j] <= pivot
            i = i + 1
            swap a[i] and a[j]
    swap a[i + 1] and a[r]
    return i + 1
```

3. Độ phức tạp:

```
    Không gian: O(logn)

    Thời gian: O(nlogn)
```

4. Biến thể và cải tiến:

```
    Quick sort 3-way: sử dụng 3 pivot để chia mảng thành 3 phần.

    Quick sort random: chọn pivot ngẫu nhiên.

    Quick sort median: chọn pivot là phần tử ở giữa mảng.
```

7 Heap Sort

1. Ý tưởng thuật toán:

```
    1. Tạo max - heap từ mảng.
    2. Lấy phần tử lớn nhất tại vị trí 0 và đưa về cuối mảng, giảm kích thước của
mảng đi 1.
    3. heapify lại mảng.
    4. Lặp lại bước 2 và 3 cho đến khi kích thước của mảng bằng 1.
```

2. Psuedocode:

```
    buildMaxHeap(a, n)
    for i = n - 1 to 1
        swap a[0] and a[i]
        heapify(a, i, 0)
```

```
    heapify(a, n, i)
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and a[l] > a[largest]
        largest = l
    if r < n and a[r] > a[largest]
        largest = r
    if largest != i
        swap a[i] and a[largest]
        heapify(a, n, largest)
```

3. Độ phức tạp:

```
    Không gian: O(1)

    Thời gian: O(nlogn)
```

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 bằng đệ qui cho đến khi mảng chỉ còn 1 phần
tử.

    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
```

🔟 Counting Sort

1. Ý tưởng thuật toán:

```
    1. Tìm phần tử lớn nhất trong mảng
    2. Tạo mảng mới có kích thước bằng phần tử lớn nhất + 1, khởi tạo các phần tử
bằng 0
    3. Lưu số lần xuất hiện của các phần tử trong mảng tại vị trí tương ứng trong
mảng mới
    4. Cộng dồn các phần tử trong mảng mới
    5. Tìm chỉ số của mỗi phần tử trong mảng gốc trong mảng mới.
    6. Đưa các phần tử vào mảng mới theo thứ tự tìm được ở bước 5
    7. Giảm số lần xuất hiện của các phần tử trong mảng mới
```

2. Psuedocode:

```
    max = phần tử lớn nhất trong mảng

    count = mảng mới có kích thước bằng max + 1, khởi tạo các phần tử bằng 0

    for i = 0 to n-1 do:

        count[a[i]] = count[a[i]] + 1

    end for

    for i = 1 to max do:

        count[i] = count[i] + count[i-1]

    end for

    for i = n-1 to 0 do:

        b[count[a[i]]] = a[i]

        count[a[i]] = count[a[i]] - 1

    end for
```

3. Độ phức tạp:

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

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

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).
    1. Tìm phần tử lớn nhất và nhỏ nhất trong mảng
    2. Tính giá trị m theo công thức : m = (n * alpha) / (max - min ), alpha thường là 0.45
    3. Tạo mảng mới có kích thước bằng m, khởi tạo các phần tử bằng 0
    4. Lặp qua mảng gốc, tính chỉ số của phần tử tại vị trí i trong mảng mới theo công thức: index = (m - 1) * (a[i] - min) / (max - min)
    5. Tăng giá trị của phần tử tại vị trí index trong mảng mới lên 1
    6. Lặp qua mảng mới, tính vị trí bắt đầu của các phân lớp bằng cách cộng dồn các phần tử trong mảng mới
    7. Hoán đổi a[max] với a[0]
    8. Lặp  và hoán đổi để đưa các phần tử về đúng phân lớp
    9. Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự bằng thuật toán Insertion Sort
```

2. Psuedocode:

```
    min = phần tử nhỏ nhất trong mảng

    max = phần tử lớn nhất trong mảng

    m = (n * alpha) / (max - min )

    count = mảng mới có kích thước bằng m, khởi tạo các phần tử bằng 0

    for i = 0 to n-1 do:
```

```
        index = (m - 1) * (a[i] - min) / (max - min)

        count[index] = count[index] + 1

    end for

    for i = 1 to m-1 do:

        count[i] = count[i] + count[i-1]

    end for

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

    for i = 0 to m-1 do:

        while i < count[i] do:

            index = (m - 1) * (a[i] - min) / (max - min)

            swap(a[i], a[count[index]])

            count[index] = count[index] + 1

        end while

    end for

    for i = 0 to m-1 do:

        insertionSort(a[count[i-1]], count[i] - count[i-1])

    end for
```

3. Độ phức tạp:

```
    Không gian: O(1) vì thực hiện tại chỗ.
    Thời gian: có độ phức tạp trung bình là O(n) .

    Nhìn lại toàn bộ các giai đoạn của thuật toán, ta thấy như sau:
    - Giai đoạn phân lớp đòi hỏi độ phức tạp O(n) và O(m)
    - Giai đoạn Hoán vị 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)
    - Giai đoạn Insertion_Sort đổi hỏi độ phức tạp O(n2/m) ( mỗi 1 phân lớp đòi
hỏi độ phức tạp O((n/m)2) và m phân lớp đòi hỏi O(m*(n/m)2) )

    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.
```

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	:---:	:---:	:---:	:---:	:---:	:---:
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	:---:	:---:	:---:	:---:	:---:	:---:
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	:---:	:---:	:---:	:---:	:---:	:---:

Data size	10.000	30.000	50.000	100.000	300.000	500.000
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	:---:	:---:	:---:	:---:	:---:	:---:

Data size	10.000	30.000	50.000	100.000	300.000	500.000
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

1️⃣ Tổ chức đồ án

```
├──source
│   |___releases
│   |   |___a.exe
│   |___main.cpp
│   |___library.h
│   |___def_command.cpp
│   |___def_algorithms.cpp
│   |___def_supFuncs.cpp
│   |___dataGenerator.cpp
|___report.pdf
```

`source` là thư mục chứa mã nguồn của đồ án.

- `releases` là thư mục chứa các file thực thi của đồ án.
- `main.cpp` là file chứa hàm `main()`, validate các tham số đầu vào và gọi các hàm thực thi tương ứng
- `library.h` là file header chứa các khai báo hàm và các thư viện cần thiết, được chia làm 4 phần: + I. khai báo các hàm thuật toán + II. khai báo các hàm command + III. khai báo các hàm tạo dữ liệu + IV. khai báo các hàm hỗ trợ khác
- `def_command.cpp` là file định nghĩa các hàm command, thực hiện các yêu cầu và xuất kết quả ra file
- `def_algorithms.cpp` là file định nghĩa các hàm đo thời gian chạy và số lần so sánh của các thuật toán
- `def_supFuncs.cpp` là file định nghĩa các hàm hỗ trợ khác
- `dataGenerator.cpp` là file định nghĩa các hàm tạo dữ liệu

`report.pdf` là file báo cáo đồ án.

2️⃣ Các ghi chú

- Thư viện sử dụng : `iostream`, `fstream`, `ctime`, `cstdlib`, `string.h`, `cmath`

- `iostream` : thư viện chuẩn c++ dùng để nhập xuất dữ liệu
- `fstream` : thư viện chuẩn c++ dùng để đọc ghi file
- `ctime` : thư viện chuẩn c++ dùng để đo thời gian chạy
- `cstdlib` : thư viện chuẩn c++ dùng để sử dụng hàm `rand()`
- `string.h` : thư viện chuẩn c dùng để xử lý chuỗi
- `cmath` : thư viện chuẩn c++ dùng để sử dụng hàm `log()`, `pow()`

VI. Tài liệu tham khảo

1. selection sort 2. insertion sort 3. bubble sort 4. shell sort 5. heap sort 6. merge sort 7. quick sort 8. counting sort 9. radix sort 10.1 flash sort 10.2 flash sort 11. shaker sort