Lê Phạm Hoàng Trung

MSSV: 21120157

Lab Homework 1

Ngày 14 tháng 10 năm 2022

Problem 1: Selection Sort

Ý tưởng:

Thuật toán Selection Sort sắp xếp một mảng bằng cách liên tục tìm phần tử nhỏ nhất (xét theo thứ tự tăng dần) từ phần chưa được sắp xếp và đặt nó ở đầu. Thuật toán duy trì hai mảng con trong một mảng nhất định. Mảng con đã được sắp xếp. Mảng con còn lại chưa được sắp xếp.

Thuật toán:

Các bước cơ bản của thuật toán này là:

Ở lượt thứ nhất, ta chọn trong dãy khoá a[1..n] ra khoá nhỏ nhất (khoá mọi khoá khác) và đổi giá trị của nó với a[1], khi đó giá trị khoá a[1] trở thành giá trị khoá nhỏ nhất.
Ở lượt thứ hai, ta chọn trong dãy khoá a[2..n] ra khoá nhỏ nhất và đổi giá trị của nó với a[2].
. . .
Ở lượt thứ i, ta chọn trong dãy khoá a[i..n] ra khoá nhỏ nhất và đổi giá trị của nó với a[i].
. . .
Làm tới lượt thứ n - 1, chọn trong hai khoá a[n-1], a[n] ra khoá nhỏ nhất và đổi giá trị của nó với a[n-1].

Ví dụ minh hoạ từng bước:

Ví dụ với mảng: arr = [64, 25, 12, 22, 11]

Code mẫu C++:

```
void SelectionSort(int a[], int n) {
        for (int i = 0; i < n; i++) {
                int minpos = i;
                for (int j = i + 1; j < n; j++) {
                        if (a[j] < a[minpos]) {
                                minpos = j;
                        }
                }
                if (a[minpos] < a[i]) {
                        swap(a[i], a[minpos]);
                }
```

```
            }
}
```

- Bước 1: [i, minpos] = [0, 0]
+ Sau khi chạy vòng for j, ta tìm được minpos = 4 (tức vị trí số 11), do 11 là số nhỏ nhất trong dãy [25, 12, 22, 11] nên minpos bây giờ có giá trị là 4. Do a[4] < a[0] nên tiến hành swap 2 thành phần này, cuối cùng ta được mảng arr = [11, 25, 12, 22, 64].

- Bước 2: [i, minpos] = [1, 1]
+ Sau khi chạy vòng for j, ta tìm được minpos = 2 (tức vị trí số 12), do 12 là số nhỏ nhất trong dãy [12, 22, 64] nên minpos bây giờ có giá trị là 2. Do a[2] < a[1] nên tiến hành swap 2 thành phần này, cuối cùng ta được mảng arr = [11, 12, 25, 22, 64].

- Bước 3: [i, minpos] = [2, 2]
+ Sau khi chạy vòng for j, ta tìm được minpos = 3 (tức vị trí số 22), do 22 là số nhỏ nhất trong dãy [22, 64] nên minpos bây giờ có giá trị là 3. Do a[3] < a[2] nên tiến hành swap 2 thành phần này, cuối cùng ta được mảng arr = [11, 12, 22, 25, 64].

- Bước 4: [i, minpos] = [3, 3]
+ Sau khi chạy vòng for j, do a[i] = a[minpos] = 25 nên ta không tìm được số nào nhỏ hơn a[minpos] trong dãy [64] nên minpos không thay đổi, cuối cùng ta được mảng arr = [11, 12, 22, 25, 64].

- Bước 5: [i, minpos] = [4, 4]
+ Sau khi chạy vòng for j, do a[i] = a[minpos] = 64 nên ta không tìm được số nào nhỏ hơn a[minpos] trong dãy [] nên minpos không thay đổi, cuối cùng ta được mảng arr = [11, 12, 22, 25, 64].

Đánh giá thuật toán:

Đối với Selection Sort, có thể coi phép so sánh (a[j] < a[minpos]) là phép toán O(1). Ở lượt thứ i, để chọn ra khoá nhỏ nhất bao giờ cũng cần n - i phép so sánh, số lượng phép so sánh này không hề phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra tổng số phép so sánh sẽ phải thực hiện là: (n - 1) + (n - 2) + . . . + 1 = n * (n - 1) / 2

Vậy thuật toán Selection Sort có độ phức tạp tính toán là O(n^2)

Độ phức tạp về không gian của Selection Sort là O(1) vì chỉ có mỗi biến minpos được sử dụng.

Kết quả thực nghiệm:

Selection Sort time run

```
2500
With n = 2500 and random data, time use is : 0.009
With n = 2500 and sorted data, time use is : 0.008
With n = 2500 and reverse data, time use is : 0.007
With n = 2500 and nearly sorted data, time use is : 0.007

Input key:
1
Welcome to Selection Sort :)
Input n:
10000
With n = 10000 and random data, time use is : 0.115
With n = 10000 and sorted data, time use is : 0.112
With n = 10000 and reverse data, time use is : 0.118
With n = 10000 and nearly sorted data, time use is : 0.112

Input key:
1
Welcome to Selection Sort :)
Input n:
25000
With n = 25000 and random data, time use is : 0.696
With n = 25000 and sorted data, time use is : 0.697
With n = 25000 and reverse data, time use is : 0.737
With n = 25000 and nearly sorted data, time use is : 0.699

Input key:
1
Welcome to Selection Sort :)
Input n:
100000
With n = 100000 and random data, time use is : 11.284
With n = 100000 and sorted data, time use is : 11.41
With n = 100000 and reverse data, time use is : 12.083
With n = 100000 and nearly sorted data, time use is : 11.423

Input key:
250000

Input key:
1
Welcome to Selection Sort :)
Input n:
250000
With n = 250000 and random data, time use is : 70.141
With n = 250000 and sorted data, time use is : 70.842
With n = 250000 and reverse data, time use is : 75.639
With n = 250000 and nearly sorted data, time use is : 70.881
```

Size / Array Type	2500	10000	25000	100000	250000
Random	0.009 s	0.115 s	0.696 s	11.284 s	70.141 s
Sorted	0.008 s	0.112 s	0.697 s	11.41 s	70.842 s
Reverse	0.007 s	0.118 s	0.737 s	12.083 s	75.639 s
Nearly Sorted	0.007 s	0.112 s	0.699 s	11.423 s	70.881 s

Problem 2: Insertion Sort

Ý tưởng:

Sắp xếp chèn (insertion sort) là một thuật toán sắp xếp bắt chước cách sắp xếp quân bài của những người chơi bài. Muốn sắp một bộ bài theo trật tự người chơi bài rút lần lượt từ quân thứ 2, so với các quân đứng trước nó để chèn vào vị trí thích hợp.

Thuật toán:

Các bước cơ bản của thuật toán này là:

Xét dãy khoá a[1..n]. Ta thấy dãy con chỉ gồm mỗi một khoá là a[1] có thể coi là đã sắp xếp rồi. Xét thêm a[2], ta so sánh nó với a[1], nếu thấy a[2] < a[1] thì chèn nó vào trước a[1]. Đối với a[3], ta lại xét dãy chỉ gồm 2 khoá a[1], a[2] đã sắp xếp và tìm cách chèn a[3] vào dãy khoá đó để được thứ tự sắp xếp. Một cách tổng quát, ta sẽ sắp xếp dãy a[1..i] trong điều kiện dãy a[1..i-1] đã sắp xếp rồi bằng cách chèn a[i] vào dãy đó tại vị trí đúng khi sắp xếp.

Ví dụ minh hoạ từng bước:

Ví dụ với mảng: arr = [12, 11, 13, 5, 6]

Code mẫu C++:

```
void InsertionSort(int a[], int n) {
        for (int i = 0; i < n; i++) {
                int j = i;
                while (j > 0 && a[j] < a[j - 1]) {
                        swap(a[j], a[j - 1]);
                        j--;
                }
        }
}
```

- Bước 1: [i, j] = [0, 0]
+ Sau khi chạy vòng while, ta thấy do a[j] là vị trí đầu nên không có a[j-1], nên mảng vẫn giữ nguyên, cuối cùng ta được mảng arr = [12, 11, 13, 5, 6].

- Bước 2: [i, j] = [1, 1]
+ Sau khi chạy vòng while, ta thấy do a[j] = 11 < a[j-1] = 12 nên ta swap 2 thành phần này với nhau, cuối cùng ta được mảng arr = [11, 12, 13, 5, 6].

- Bước 3: [i, j] = [2, 2]
+ Sau khi chạy vòng while, ta thấy do a[j] = 13 không bé hơn các thành phần ở vị trí nhỏ hơn nó, nên ta giữ nguyên mảng, cuối cùng ta được mảng arr = [11, 12, 13, 5, 6].

- Bước 4: [i, j] = [3, 3]
+ Sau khi chạy vòng while, ta thấy do a[j] = 5 bé hơn a[j-1] = 13 nên ta swap 2 thành phần này, mảng trở thành [11, 12, 5, 13, 6], sau đó j giảm xuống j = 2, ta thấy a[j] = 5 bé hơn a[j-1] = 12 nên ta lại tiếp tục swap 2 thành phần này, mảng trở thành [11, 5, 12, 13, 6], j lại tiếp tục giảm xuống j = 1, ta thấy a[j] = 5 bé hơn a[j-1] = 11 nên ta lại swap tiếp 2 thành phần này, mảng trở thành [5, 11, 12, 13, 6], cuối cùng ta được mảng arr = [5, 11, 12, 13, 6].

- Bước 5: [i, j] = [4, 4]
+ Sau khi chạy vòng while, ta thấy do a[j] = 6 bé hơn a[j-1] = 13 nên ta swap 2 thành phần này, mảng trở thành [5, 11, 12, 6, 13], sau đó j giảm xuống j = 3, ta thấy a[j] = 6 bé hơn a[j-1] = 12 nên ta lại tiếp tục swap 2 thành phần này, mảng trở thành [5, 11, 6, 12, 13], j lại tiếp tục giảm xuống j = 2, ta thấy a[j] = 6 bé hơn a[j-1] = 11 nên ta lại swap tiếp 2 thành phần này, mảng trở thành [5, 6, 11, 12, 13], j tiếp tục giảm còn j = 1, do a[j] = 6 không bé hơn a[j-1] nên mảng giữ nguyên, cuối cùng ta được mảng đã sắp xếp arr = [5, 6, 11, 12, 13].

Đánh giá thuật toán:

Đối với thuật toán Insertion Sort, thì chi phí thời gian thực hiện thuật toán phụ thuộc vào tình trạng dãy khoá ban đầu.

Trường hợp tốt nhất ứng với dãy khoá đã sắp xếp rồi, mỗi lượt chỉ cần 1 phép so sánh a[j] < a[j - 1], và như vậy tổng số phép so sánh được thực hiện là n - 1. Phân tích trong trường hợp tốt nhất, độ phức tạp tính toán của InsertionSort là (n)

Trường hợp tồi tệ nhất ứng với dãy khoá đã có thứ tự ngược với thứ tự cần sắp thì ở lượt thứ i, cần có i - 1 phép so sánh và tổng số phép so sánh là:

1 + 2 + ... + (n - 1) + (n - 2) = n * (n - 1) / 2.

Vậy phân tích trong trường hợp tốt nhất, độ phức tạp tính toán của InsertionSort là (n^2)

Trường hợp các giá trị khoá xuất hiện một cách ngẫu nhiên, ta có thể coi xác suất xuất hiện mỗi khoá là đồng khả năng, thì có thể coi ở lượt thứ i, thuật toán cần trung bình i / 2 phép so sánh và tổng số phép so sánh là:

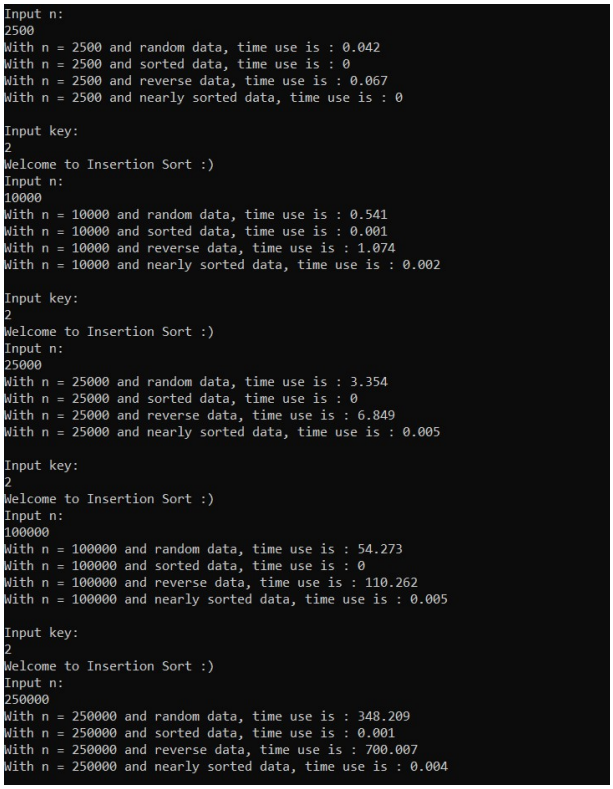
(1 / 2) + (2 / 2) + . . . + (n / 2) = (n + 1) * n / 4.

Vậy phân tích trong trường hợp trung bình, độ phức tạp tính toán của InsertionSort là (n^2)

Độ phức tạp không gian của thuật toán Insertion Sort là O(1) vì chỉ có một biến j được sử dụng.

Kết quả thực nghiệm:

Insertion Sort time run



Size / Array Type	2500	10000	25000	100000	250000
Random	0.042 s	0.541 s	3.354 s	54.273 s	348.209 s
Sorted	0 s	0.001 s	0 s	0 s	0.001 s
Reverse	0.067 s	1.074 s	6.849 s	110.262 s	700.007 s
Nearly Sorted	0 s	0.002 s	0.005 s	0.005 s	0.004 s

Problem 3: Bubble Sort

Ý tưởng:

Thuật toán sắp xếp bubble sort thúcj 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.

Thuật toán:

Các bước cơ bản của thuật toán này là:

Trong thuật toán sắp xếp nổi bọt, dãy các khoá sẽ được duyệt từ đầu dãy đến cuối dãy (từ a[1] về a[n]), nếu gặp hai khoá kế cận bị ngược thứ tự thì đổi chỗ của chúng cho nhau. Sau lần duyệt

như vậy, khoá nhỏ nhất trong dãy khoá sẽ được chuyển về vị trí đầu tiên và vấn đề trở thành sắp xếp dãy khoá từ a[2] tới a[n]:

Ví dụ minh hoạ từng bước:

Ví dụ với mảng: arr = [5, 1, 4, 2, 8]

Code mẫu C++:

```
void BubbleSort(int a[], int n) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < i; j++) {
                        if (a[j] > a[i]) {
                                swap(a[i], a[j]);
                        }
                }
        }
}
```

- Bước 1: [i, j] = [0, 0]
+ Do i = j = 0 nên ta tiếp tục qua vị trí i khác.

- Bước 2: [i, j] = [1, 0]
+ Do a[j] = 5 > a[i] = 1, do đó ta tiến hành swap 2 thành phần này, ta được mảng sau arr = [1, 5, 4, 2, 8].
+ Ta tiếp tục tăng j lên thì ta được j = 1 = i, nên ta bỏ qua và tiếp tục với vị trí i khác.

- Bước 3: [i, j] = [2, 0]
+ Do a[j] = 1 < a[i] = 4 nên mảng vẫn giữ nguyên: arr = [1, 5, 4, 2, 8].
+ Tiếp tục tăng j lên 1, ta có a[j] = 5 > a[i] = 4, nên ta tiến hành swap 2 thành phần này, ta được mảng arr = [1, 4, 5, 2, 8]. + Tiếp tục tăng j lên thì ta được j = 2 = i, nên ta bỏ qua và tiếp tục với vị trí i khác.

- Bước 4: [i, j] = [3, 0]
+ Do a[j] = 1 < a[i] = 2 nên mảng vẫn giữ nguyên: arr = [1, 4, 5, 2, 8].
+ Tiếp tục tăng j lên 1, ta có a[j] = 4 > a[i] = 2, nên ta tiến hành swap 2 thành phần này, ta được mảng arr = [1, 2, 5, 4, 8].
+ Tiếp tục tăng j lên 2, ta có a[j] = 5 > a[i] = 4, nên ta tiến hành swap 2 thành phần này, ta được mảng arr = [1, 2, 4, 5, 8].
+ Tiếp tục tăng j lên thì ta được j = 3 = i, nên ta bỏ qua và tiếp tục với vị trí i khác.

- Bước 5: [i, j] = [4, 0]
+ Do a[j] với j từ 0 đến 3 đều không có giá trị nào lớn hơn a[i] = 8, nên ta kết thúc vòng lặp và thuật toán.

Đánh giá thuật toán:

Đối với Bubble Sort, nếu ta xem phép so sánh a[j] > a[i] có độ phức tạp là O(1), thì ta có độ phức tạp thuật toán được tính như sau:

Với mỗi i, vòng lặp sẽ lại duyệt qua i-1 phần tử, ta có số lần duyệt như sau:

(1 - 1) + (2 - 1) + (3 - 1) + ... + (n - 2) + (n - 1) = n*(n - 1)/2

Kết quả thực nghiệm:

Bubble Sort time run

```
2500
With n = 2500 and random data, time use is : 0.037
With n = 2500 and sorted data, time use is : 0.007
With n = 2500 and reverse data, time use is : 0.069
With n = 2500 and nearly sorted data, time use is : 0.007

Input key:
3
Welcome to Bubble Sort :)
Input n:
10000
With n = 10000 and random data, time use is : 0.58
With n = 10000 and sorted data, time use is : 0.118
With n = 10000 and reverse data, time use is : 1.195
With n = 10000 and nearly sorted data, time use is : 0.114

Input key:
3
Welcome to Bubble Sort :)
Input n:
25000
With n = 25000 and random data, time use is : 3.431
With n = 25000 and sorted data, time use is : 0.696
With n = 25000 and reverse data, time use is : 6.888
With n = 25000 and nearly sorted data, time use is : 0.702

Input key:
3
Welcome to Bubble Sort :)
Input n:
100000
With n = 100000 and random data, time use is : 43.088
With n = 100000 and sorted data, time use is : 11.197
With n = 100000 and reverse data, time use is : 112.3
With n = 100000 and nearly sorted data, time use is : 11.159

Input key:
3
Welcome to Bubble Sort :)
Input n:
250000
With n = 250000 and random data, time use is : 176.776
With n = 250000 and sorted data, time use is : 69.957
With n = 250000 and reverse data, time use is : 704.043
With n = 250000 and nearly sorted data, time use is : 71.558
```

Size / Array Type	2500	10000	25000	100000	250000
Random	0.037 s	0.58 s	3.431 s	43.088 s	176.776 s
Sorted	0.007 s	0.118 s	0.696 s	11.197 s	69.957 s
Reverse	0.069 s	1.195 s	6.888 s	112.3 s	704.043 s
Nearly Sorted	0.007 s	0.114 s	0.702 s	11.159 s	71.558 s

Problem 4: Shell Sort

Ý tưởng:

Ý tưởng chính của thuật toán là phân chia dãy ban đầu thành những dãy con mà mỗi phần tử của dãy cách nhau 1 vị trí là h. Insertion sort áp dụng sau đó trên mỗi dãy con sẽ làm cho các phần tử được đưa về vị trí đúng tương đối (trong dãy con) 1 cách nhanh chóng.

Thuật toán:

Ví dụ minh hoạ từng bước:

Ví dụ với mảng: arr = [12, 34, 54, 2, 3]

Code mẫu C++:

```
void ShellSort(int arr[], int n)
{
        // Start with a big gap, then reduce the gap
        for (int gap = n / 2; gap > 0; gap /= 2)
        {
                for (int i = gap; i < n; i += 1)
                {
                        int temp = arr[i];
                        int j;
                        for (j=i; j >= gap && arr[j-gap] > temp; j-=gap){
                            arr[j] = arr[j - gap];
                        }
                        arr[j] = temp;
                }
        }
}
```

- Với mảng trên, do n = 5, nên ta có gap = 2.

* gap = 2
    - Ta có i = gap = 2
        + temp = arr[i] = arr[2] = 54
        + j = i = 2 >= gap = 2 arr[j - gap] = arr[0] = 12 không lớn hơn temp, kết thúc vòng lặp.
        + arr[j] = arr[2] = temp = 54. Mảng vẫn giữ nguyên arr = [12, 34, 54, 2, 3].
    - Ta có i += 1 = 3
        + temp = arr[i] = arr[3] = 2
        + j = i = 3 >= gap = 2 arr[j - gap] = arr[1] = 34 > temp = 2
            nên arr[j] = arr[3] = arr[j - gap] = arr[1] = 34.
            Mảng trở thành arr = [12, 34, 54, 34, 3]
        + j -= gap = 1 < gap nên kết thúc vòng lặp j.

+ Ta hoán đổi arr[j] = arr[1] = temp = 2. Mảng arr = [12, 2, 54, 34, 3].

- i += 1 = 4
  + temp = arr[i] = arr[4] = 3
  + j = 4 >= gap = 2 arr[j - gap] = arr[2] = 54 > temp = 3
    nên arr[j] = arr[4] = arr[j - gap] = arr[2] = 54
    Mảng trở thành arr = [12, 2, 54, 34, 54]
  + j -= gap = 2 >= gap = 2 arr[j - gap] = arr[0] = 12 >= temp = 3
    nên arr[j] = arr[2] = arr[j - gap] = arr[0] = 12
    Mảng trở thành arr = [12, 2, 12, 34, 54]
  + j -= gap = 0 < gap nên kết thúc vòng lặp j.
  + Ta hoán đổi arr[j] = arr[0] = temp = 3. Mảng arr = [3, 2, 12, 34, 54]
- i += 1 = 5 > n Kết thúc vòng lặp i.

* gap /= 2 = 1
  - Ta có i = gap = 1
    + temp = arr[i] = arr[1] = 2
    + j = i = 1 >= gap = 1 arr[j - gap] = arr[0] = 3 > temp = 2
      nên arr[j] = arr[1] = arr[j - gap] = arr[0] = 3.
      Mảng trở thành arr = [3, 3, 12, 34, 54]
    + j -= gap = 0 < gap nên kết thúc vòng lặp j.
    + Ta hoán đổi arr[j] = arr[0] = temp = 2. Mảng arr = [2, 3, 12, 34, 54]

Tới đây mảng đã được sort, nếu ta tiếp tục thực hiện thì điều kiện luôn đúng và không có thêm bất kỳ sự sắp xếp nào nữa.

Đánh giá thuật toán:

Trường hợp phức tạp tồi tệ nhất
Độ phức tạp trong trường hợp xấu nhất Shell Sort là O (n^2)

Độ phức tạp của trường hợp tốt nhất
Khi danh sách mảng nhất định đã được sắp xếp, tổng số phép so sánh của mỗi khoảng bằng kích thước của mảng đã cho.
Vì vậy, độ phức tạp của trường hợp tốt nhất là (n log (n))

Độ phức tạp của trường hợp trung bình
Shell sort với độ phức tạp trung bình phụ thuộc vào khoảng thời gian được chọn bởi lập trình viên. (n log (n) 2).
Độ phức tạp của trường hợp trung bình: O (n * log n)

Kết quả thực nghiệm:

Shell Sort time run

```
2500
With n = 2500 and random data, time use is : 0
With n = 2500 and sorted data, time use is : 0
With n = 2500 and reverse data, time use is : 0
With n = 2500 and nearly sorted data, time use is : 0

Input key:
4
Welcome to Shell Sort :)
Input n:
10000
With n = 10000 and random data, time use is : 0.002
With n = 10000 and sorted data, time use is : 0.001
With n = 10000 and reverse data, time use is : 0.001
With n = 10000 and nearly sorted data, time use is : 0.001

Input key:
4
Welcome to Shell Sort :)
Input n:
25000
With n = 25000 and random data, time use is : 0.006
With n = 25000 and sorted data, time use is : 0.001
With n = 25000 and reverse data, time use is : 0.001
With n = 25000 and nearly sorted data, time use is : 0.002

Input key:
4
Welcome to Shell Sort :)
Input n:
100000
With n = 100000 and random data, time use is : 0.024
With n = 100000 and sorted data, time use is : 0.005
With n = 100000 and reverse data, time use is : 0.007
With n = 100000 and nearly sorted data, time use is : 0.006

Input key:
4
Welcome to Shell Sort :)
Input n:
250000
With n = 250000 and random data, time use is : 0.061
With n = 250000 and sorted data, time use is : 0.012
With n = 250000 and reverse data, time use is : 0.019
With n = 250000 and nearly sorted data, time use is : 0.013
```

Size / Array Type	2500	10000	25000	100000	250000
Random	0 s	0.002 s	0.006 s	0.024 s	0.061 s
Sorted	0 s	0.001 s	0.001 s	0.005 s	0.012 s
Reverse	0 s	0.001 s	0.001 s	0.007 s	0.019 s
Nearly Sorted	0 s	0.001 s	0.002 s	0.006 s	0.013 s

Problem 5: Heap Sort

Ý tưởng:

Thuật toán Heap sort lấy ý tưởng giải quyết từ cấu trúc heap, cụ thể: Ta coi dãy cần sắp xếp là một cây nhị phân hoàn chỉnh, sau đó hiệu chỉnh cây thành dạng cấu trúc heap (vun đống) Dựa vào tính chất của cấu trúc heap, ta có thể lấy được ra phần từ lớn nhất hoặc nhỏ nhất của dãy, phần tử này chính là gốc của heap.

Thuật toán:

Các bước cơ bản của thuật toán này là:

- Thuật toán Heap Sort có 2 bước chính:

+ Thứ nhất: Heapify. Thủ tục Heapify vun cây gốc root thành đống trong điều kiện hai cây gốc 2i+2 và 2i +1 đã là đống rồi.

```
void Heapify(int a[], int n, int i) {
        int parent = i;
        int leftchild = 2 * i + 1;
        int rightchild = 2 * i + 2;

        if (leftchild < n && a[leftchild] > a[parent]) {
                parent = leftchild;
        }
        if (rightchild < n && a[rightchild] > a[parent]) {
                parent = rightchild;
        }
        if (parent != i) {
                swap(a[parent], a[i]);
                Heapify(a, n, parent);
        }
}
```

+ Thứ hai: Heap Sort. Thuật toán hoạt động dựa trên các nguyên tắc sau:

++ Phần tử lớn nhất được đặt ở nút gốc theo thuộc tính Max Heap.

++ Loại bỏ phần tử gốc và đặt nó ở cuối mảng nhị phân. Đặt phần tử cuối cùng của cây nhị phân vào chỗ trống.

++ Giảm kích thước của Heap đi 1 đơn vị.

++ Tạo cấu trúc dữ liệu Heap cho phần tử gốc để nút gốc chứa phần tử có giá trị lớn nhất.

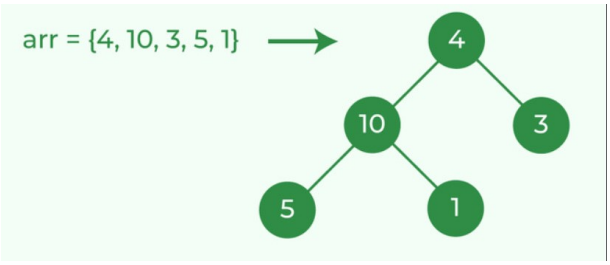
++ Lặp lại quá trình này cho đến khi tất cả phần tử của danh sách được sắp xếp đúng.

```
void HeapSort(int a[], int n) {
        for (int i = n / 2 - 1; i >= 0; i--) {
                Heapify(a, n, i);
        }
        for (int i = n - 1; i > 0; i--) {
                swap(a[i], a[0]);
                Heapify(a, i, 0);
        }
}
```

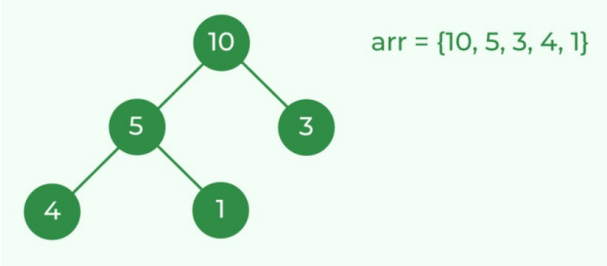
Ví dụ minh hoạ từng bước:

- Ví dụ đối với mảng arr[] = 4, 10, 3, 5, 1

- Đầu tiên, ta thấy đây là một cây nhị phân chưa được sắp xếp max heap qua thuật toán Heapify (do có đỉnh gốc là 4, bé hơn đỉnh con là 10, xem hình vẽ):



- Chúng ta sẽ sử dụng Heapify để chuyển cây trên về max heap, ta sẽ chuyển 4 với 10, sau đó lại xét thấy 4 vẫn bé hơn đỉnh con của nó là 5 nên ta chuyển thêm một lần nữa. - Sau khi đã sử dụng Heapify để vun cây mới theo đúng cấu trúc max heap, ta được cây mới như sau:



- Tới đây ta bắt đầu sử dụng thuật toán Heap Sort để sắp xếp mảng:

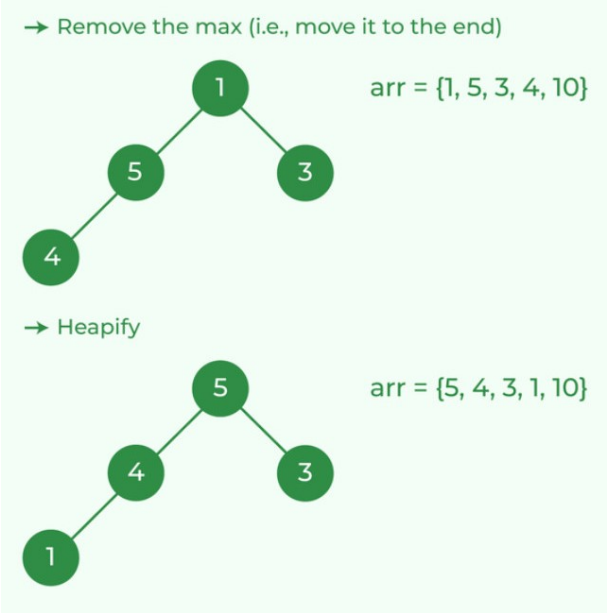
+ Bước 1:

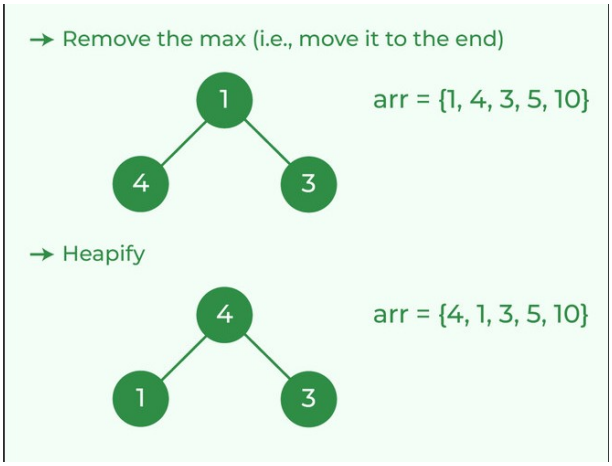
++ Chúng ta sẽ xoá phần tử lớn nhất bằng cách đưa phần tử lớn nhất về cuối mảng (ở đây phần tử lớn nhất là phần tử thứ 10). Ta sẽ tiến hành tráo đổi phần tử gốc rễ (root) với phần tử cuối mảng (đổi chỗ 10 và 1). Sau đó xoá đi phần tử 10 trong cây.

++ Sau đó ta thấy cây không còn giữ cấu trúc max heap, nên ta tiến hành Heapify lại cây.

+ Bước 2:

++ Sau khi Heapify lại cây ta có phần tử lớn nhất nằm ở root có giá trị 5, ta lại tiếp tục sử dụng Heap Sort để đưa nó về cuối mảng (ở vị trí phần tử có giá trị 1). Sau đó xoá đi phần tử có giá trị 5 trong cây và tiến hành Heapify lại cho cây.



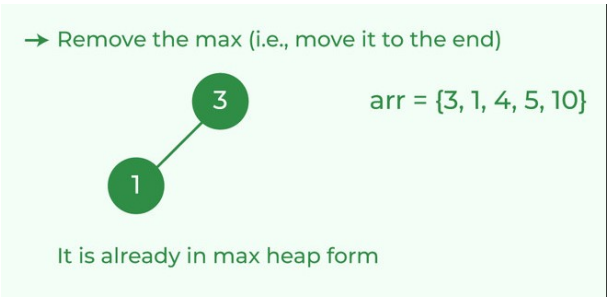


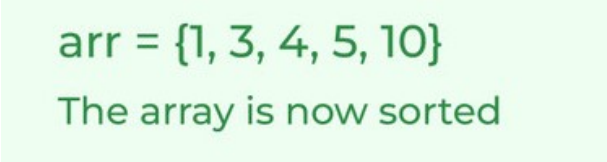
+ Bước 3:

++ Sau khi Heapify lại cây ta có phần tử lớn nhất nằm ở root có giá trị 4, ta lại tiếp tục sử dụng Heap Sort để đưa nó về cuối mảng (ở vị trí phần tử có giá trị 3). Sau đó xoá đi phần tử có giá trị 4 trong cây và tiến hành Heapify lại cho cây.

+ Bước 4:

++ Sau khi Heapify lại cây ta có phần tử lớn nhất nằm ở root có giá trị 3, ta lại tiếp tục sử dụng Heap Sort để đưa nó về cuối mảng (ở vị trí phần tử có giá trị 1). Sau đó xoá đi phần tử có giá trị 3 trong cây và tiến hành Heapify lại cho cây. Lúc này mảng đã được sắp xếp nên thuật toán dừng lại, mảng cuối cùng là arr = [1, 3, 4, 5, 10]





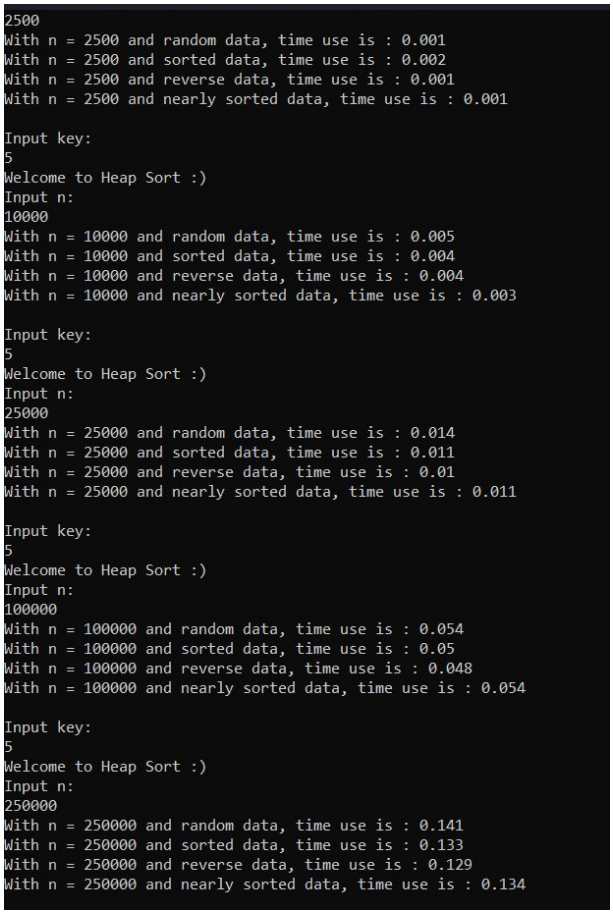
Đánh giá thuật toán:

Về độ phức tạp của thuật toán, ta đã biết rằng cây nhị phân hoàn chỉnh có n nút thì chiều cao của nó là [lg(n)] + 1. Trong trường hợp xấu nhất Hopify phải thực hiện tìm đường đi từ nút gốc tới nút lá ở xa nhất thì đường đi tìm được cũng chỉ dài bằng chiều cao của cây nên thời gian thực hiện một lần gọi Hopify là O(lgn). Từ đó có thể suy ra, trong trường hợp xấu nhất, độ phức tạp của HeapSort cũng chỉ là O(nlgn). Việc đánh giá thời gian thực hiện trung bình phức tạp hơn, ta chỉ ghi nhận một kết quả đã chứng minh được là độ phức tạp trung bình của HeapSort cũng là O(nlgn).

Kết quả thực nghiệm:

Size / Array Type	2500	10000	25000	100000	250000
Random	0.001 s	0.005 s	0.014 s	0.054 s	0.141 s
Sorted	0.002 s	0.004 s	0.011 s	0.05 s	0.133 s
Reverse	0.001 s	0.004 s	0.01 s	0.048 s	0.129 s
Nearly Sorted	0.001 s	0.003 s	0.011 s	0.054 s	0.134 s

Heap Sort time run



```
2500
With n = 2500 and random data, time use is : 0.001
With n = 2500 and sorted data, time use is : 0.002
With n = 2500 and reverse data, time use is : 0.001
With n = 2500 and nearly sorted data, time use is : 0.001

Input key:
5
Welcome to Heap Sort :)
Input n:
10000
With n = 10000 and random data, time use is : 0.005
With n = 10000 and sorted data, time use is : 0.004
With n = 10000 and reverse data, time use is : 0.004
With n = 10000 and nearly sorted data, time use is : 0.003

Input key:
5
Welcome to Heap Sort :)
Input n:
25000
With n = 25000 and random data, time use is : 0.014
With n = 25000 and sorted data, time use is : 0.011
With n = 25000 and reverse data, time use is : 0.01
With n = 25000 and nearly sorted data, time use is : 0.011

Input key:
5
Welcome to Heap Sort :)
Input n:
100000
With n = 100000 and random data, time use is : 0.054
With n = 100000 and sorted data, time use is : 0.05
With n = 100000 and reverse data, time use is : 0.048
With n = 100000 and nearly sorted data, time use is : 0.054

Input key:
5
Welcome to Heap Sort :)
Input n:
250000
With n = 250000 and random data, time use is : 0.141
With n = 250000 and sorted data, time use is : 0.133
With n = 250000 and reverse data, time use is : 0.129
With n = 250000 and nearly sorted data, time use is : 0.134
```

Problem 6: Merge Sort

Ý tưởng:

Chia mảng lớn thành những mảng con nhỏ hơn bằng cách chia đôi mảng lớn và tiếp tục chia đôi các mảng con cho đến khi mảng con nhỏ nhất chỉ còn 1 phần tử.

So sánh 2 mảng con có cùng mảng cơ sở (khi chia đôi mảng lớn thành 2 mảng con thì mảng lớn đó gọi là mảng cơ sở của 2 mảng con đó).

Khi so sánh chúng vừa sắp xếp vừa ghép 2 mảng con đó lại thành mảng cơ sở, tiếp tục so sánh và ghép các mảng con lại đến khi còn lại mảng duy nhất, đó là mảng đã được sắp xếp.

Thuật toán:

Các bước cơ bản của thuật toán này là:

Thuật toán merge sort sẽ có 2 phần chính là chia ra làm 2 và hợp nhất (merge) lại. Đầu tiên, mảng sẽ gọi đệ quy phân chia ra làm 2 liên tục đến khi không còn chia nhỏ được nữa sau đó merge từng phần đã được sắp xếp lại. Các bước thực hiện như sau:

1. Tìm điểm giữa để chia mảng thành hai nửa: Ở giữa m = (l + r) / 2
2. Hợp nhất nửa đầu của mảng: Gọi mergeSort(arr, l, m)
3. Hợp nhất cho nửa sau của mảng: Gọi mergeSort(arr, m + 1, r)
4. Hợp nhất hai nửa được sắp xếp ở bước 2 và 3: Gọi merge(arr, l, m, r)

Ví dụ minh hoạ từng bước:

- Code mẫu:

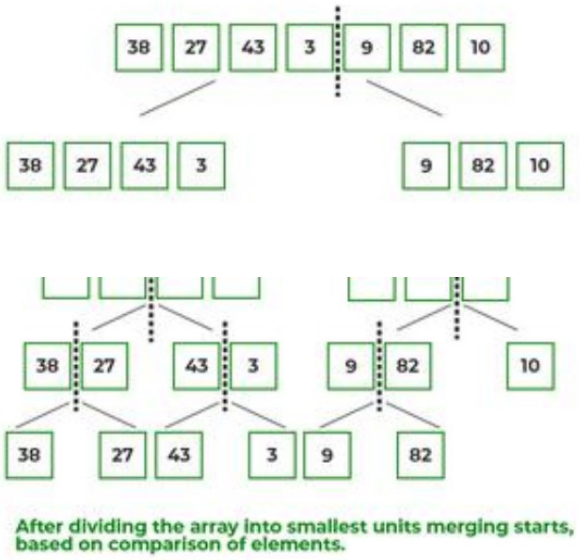
```
void Merge(int arr[], int l, int m, int r)
{
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Tạo các mảng tạm */
        auto* L = new int[125000], * R = new int[125000];

        /* Copy dữ liệu sang các mảng tạm */
        for (i = 0; i < n1; i++)
                L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
                R[j] = arr[m + 1 + j];

        /* Gộp hai mảng tạm vừa rồi vào mảng arr*/
        i = 0; // Khởi tạo chỉ số bắt đầu của mảng con đầu tiên
        j = 0; // Khởi tạo chỉ số bắt đầu của mảng con thứ hai
        k = l; // IKhởi tạo chỉ số bắt đầu của mảng lưu kết quả
        while (i < n1 && j < n2)
        {
                if (L[i] <= R[j])
                {
                        arr[k] = L[i];
                        i++;
                }
                else
                {
                        arr[k] = R[j];
```

```
                        j++;
                }
                k++;
        }

        /* Copy các phần tử còn lại của mảng L vào arr nếu có */
        while (i < n1)
        {
                arr[k] = L[i];
                i++;
                k++;
        }

        /* Copy các phần tử còn lại của mảng R vào arr nếu có */
        while (j < n2)
        {
                arr[k] = R[j];
                j++;
                k++;
        }
        delete[] L;
        delete[] R;
}

void MergeSort(int a[], int left, int right) {
        if (left < right) {
                int mid = left + (right - left) / 2;
                MergeSort(a, left, mid);
                MergeSort(a, mid + 1, right);
                Merge(a, left, mid, right);
        }
}
```

- Ví dụ với mảng arr = [38, 27, 43, 3, 9, 82, 10].

- Tìm điểm ở giữa, m = l + (r - l)/2, với l = 0, r = sizeof(arr) = 7. Sau khi tính ta được m = 3. Do đó ta sẽ chia mảng thành 2 phần, phần bên trái từ [l, m] = [0, 3] tức là [38, 27, 43, 3]. Phần bên phải là [m+1,r] = [4,7] tức là [9, 82, 10]. Sau đó trong các phần bên trái và bên phải này, tiếp tục gọi đệ quy để phân chia ra thành 2 phần cho đến khi không còn phân chia được nữa thì dừng lại. Sau khi đã phân chia ta được các phần nhỏ như sau:

- Sau khi đã phân chia xong, ta tiến hành merge các phần con lại với nhau:
    + Với 2 phần 38, 27. Ta merge thành [27, 38].
    + Với 2 phần 43, 3. Ta merge thành [3, 43].
    + Với 2 phần 9, 82. Ta merge thành [9, 82].
- Sau đó ta lại merge tiếp 2 phần lớn hơn:



+ Với 2 phần [27, 38] và [3, 43] ta merge thành [3, 27, 38, 43].
+ Với 2 phần [9, 82] và [10], ta merge thành [9, 10, 82].

- Tiếp tục merge tiếp 2 phần lớn hơn nữa: [3, 27, 38, 43] và [9, 10, 82], ta được mảng [3, 9, 10, 27, 38, 43, 82]. Và đây cũng là mảng đã được sắp xếp.

Đánh giá thuật toán:

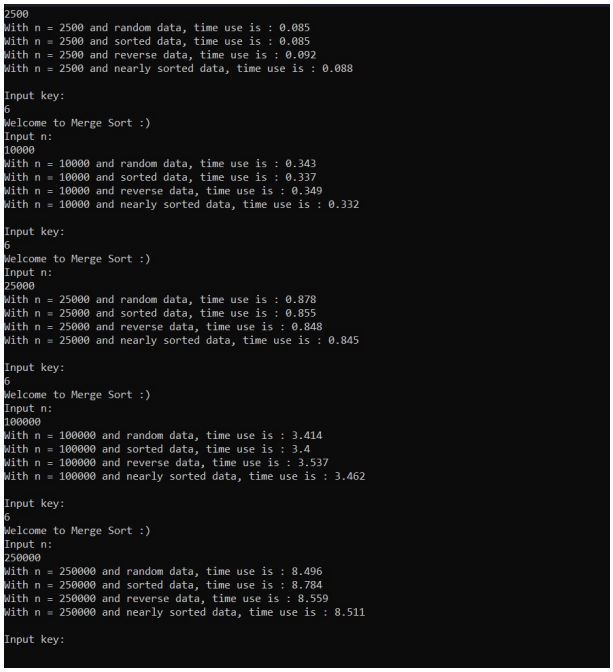
- Độ phức tạp thời gian của Merge Sort là O(nLogn) trong cả 3 trường hợp (xấu nhất, trung bình và tốt nhất) vì Merge Sort luôn chia mảng thành hai nửa và mất thời gian tuyến tính để hợp nhất hai nửa.

- Độ phức tạp không gian là O(n).

Kết quả thực nghiệm:

Size / Array Type	2500	10000	25000	100000	250000
Random	0.085 s	0.343 s	0.878 s	3.414 s	8.496 s
Sorted	0.085 s	0.337 s	0.855 s	3.4 s	8.784 s
Reverse	0.092 s	0.349 s	0.848 s	3.537 s	8.559 s
Nearly Sorted	0.088 s	0.332 s	0.845 s	3.462 s	8.511 s

Merge Sort time run



Problem 7: Quick Sort

Ý tưởng:

Quicksort là thuật toán ứng dụng ý tưởng chia nhỏ để trị. Đầu tiên nó chia mảng đầu vào thành hai mảng con một nửa nhỏ hơn, một nửa lớn hơn dựa vào một phần tử trung gian. Sau đó, nó sắp xếp đệ quy các mảng con để giải quyết bài toán.

Thuật toán:

Các bước cơ bản của thuật toán này là:

- Chọn một phần tử trong mảng làm phần tử trung gian để chia đôi mảng gọi là pivot. Thông thường ta thường chọn phần tử đầu tiên, phần tử ở giữa mảng hoặc phần tử cuối cùng của mảng để làm pivot.
- Phân đoạn: di chuyển phần tử có giá trị nhỏ hơn pivot về một bên, tất cả các phần tử có giá trị lớn hơn pivot xếp về một bên (các giá trị bằng pivot có thể đi theo một trong hai cách).
- Sau bước phân đoạn di chuyển pivot về đúng vị trí giữa của 2 mảng con
- Áp dụng đệ quy các bước phân đoạn trên cho hai mảng con để thực hiện sắp xếp
- Dừng đệ quy khi mảng con có kích thước bằng 0 hoặc 1, khi đó mảng con đã được sắp xếp.

Ví dụ minh hoạ từng bước:

- Code mẫu C++:

```
int partition(int arr[], int low, int high, int pivot) {
        int i = low;
```

```
        int j = low;
        while (i <= high) {
                if (arr[i] > pivot) {
                        i++;
                }
                else {
                        swap(arr[i], arr[j]);
                        i++;
                        j++;
                }
        }
        return j - 1;
}

void QuickSort(int arr[], int low, int high) {
        if (low < high) {
                int pivot = arr[high];
                int pos = partition(arr, low, high, pivot);

                QuickSort(arr, low, pos - 1);
                QuickSort(arr, pos + 1, high);
        }
}
```

- Ví dụ với mảng a = [9, 3, 4, 2, 1, 8].

- Bước 1:

+ Xét QuickSort(a, 0, 5). Ta có pivot = a[5] = 8.

+ pos = partition(a, 0, 5, 8)

+ [i, j] = [0, 0]

+ Chạy vòng while:

++ a[0] = 9 > pivot = 8 => i++ = 1

++ a[1] = 3 < pivot = 8 => swap(a[i], a[j]) ta được mảng a=[3, 9, 4, 2, 1, 8], i++ = 2 và j++ = 1.

++ a[2] = 4 < pivot = 8 => swap(a[i], a[j]) ta được mảng a=[3, 4, 9, 2, 1, 8], i++ = 3, j++ = 2.

++ a[3] = 2 < pivot = 8 => swap(a[i], a[j]) ta được mảng a=[3, 4, 2, 9, 1, 8], i++ = 4, j++ = 3.

++ a[4] = 1 < pivot = 8 => swap(a[i], a[j]) ta được mảng a=[3, 4, 2, 1, 9, 8], i++ = 5, j++ = 4.

++ a[5] = 8 = pivot => swap(a[i], a[j]) ta được mảng a=[3, 4, 2, 1, 8, 9], i++ = 6, j++ = 5. Kết thúc while

+ Khi đó pos sẽ có giá trị j - 1 = 4.

- Bước 2:

+ Xét QuickSort(a, 0, 3) và QuickSort(a, 5, 5), do trường hợp 2 low = high = 5 nên ta chỉ xét trường hợp đầu. Ta có pivot = a[3] = 1.

+ pos = partition(a, 0, 3, 1).
+ [i, j] = [0, 0]
+ Chạy vòng while:
++ a[0] = 3 > pivot = 1 => i++ = 1.
++ a[1] = 4 > pivot = 1 => i++ = 2.
++ a[2] = 2 > pivot = 1 => i++ = 3.
++ a[3] = 1 = pivot => swap(a[i], a[j]) ta được mảng a=[1, 4, 2, 3, 8, 9], i++ = 4, j++ = 1. Kết thúc while.
+ Khi đó pos sẽ có giá trị j - 1 = 0.

-Bước 3:
+ Xét QuickSort(a, 1, 3) và QuickSort(a, 0, -1), do trường hợp 2 low > high nên ta chỉ xét trường hợp đầu. Ta có pivot = a[3] = 3.
+ pos = partition(a, 1, 3, 3).
+ [i, j] = [1, 1].
+ Chạy vòng while:
++ a[1] = 4 > pivot => i++ = 2.
++ a[2] = 2 < pivot => swap(a[i], a[j]) ta được mảng a=[1, 2, 4, 3, 8, 9], i++ = 3 và j++ = 2.
++ a[3] = 3 = pivot => swap(a[i], a[j]) ta được mảng a=[1, 2, 3, 4, 8, 9]. i++ = 4 và j++ = 3.
+ Khi đó pos sẽ có giá trị j - 1 = 2.
- Bước 4:
+ Xét QuickSort(a, 1, 1) và QuickSort(a, 3, 3). Do không trường hợp nào thoả mãn nên ta kết thúc thuật toán. Ta có mảng đã được sắp xếp a = [1, 2, 3, 4, 8, 9].

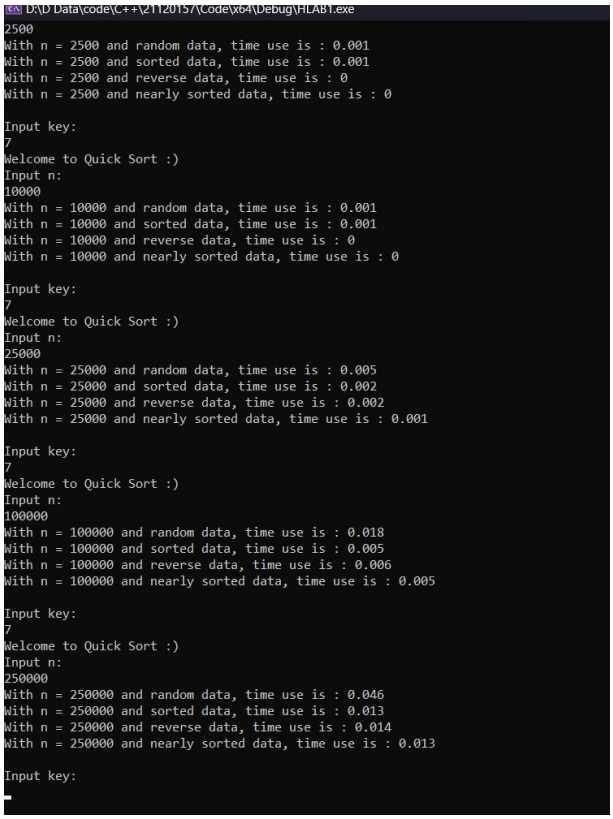
Đánh giá thuật toán:

- Trường hợp xấu nhất xảy ra khi mỗi lần phân đoạn, ta chọn phải phần tử lớn nhất hoặc nhỏ nhất làm chốt. Khi đó độ phức tạp thuật toán là O(n^2).
- Trường hợp tốt nhất khi mỗi lần phân đoạn, hai mảng con có độ dài bằng nhau. Khi đó độ phức tạp thuật toán là O(nlogn).
- Độ phức tạp không gian O(logn).

Kết quả thực nghiệm:

Size / Array Type	2500	10000	25000	100000	250000
Random	0.001 s	0.001 s	0.005 s	0.018 s	0.046 s
Sorted	0.001 s	0.001 s	0.002 s	0.005 s	0.013 s
Reverse	0 s	0 s	0.002 s	0.006 s	0.014 s
Nearly Sorted	0 s	0 s	0.001 s	0.005 s	0.013 s

Quick Sort time run



Problem 8: Radix Sort

Ý tưởng:

Thuật toán Radix Sort được thực hiện dựa trên ý tưởng phân phát thư của bưu điện (sẽ được giải thích chi tiết ở dưới) nếu một danh sách đã được sắp xếp hoàn chỉnh thì từng phần tử cũng sẽ được sắp xếp hoàn chỉnh dựa trên giá trị của các phần tử đó. Thuật toán này yêu cầu danh sách cần được sắp xếp để có thể so sánh thứ tự các vị trí vì thế sắp xếp theo cơ số không giới hạn ở tập số nguyên (ta có thể dễ dàng đưa dạng xâu về cơ số nhị phân).

Thuật toán:

Ta biết rằng, để chuyển một khối lượng thư lớn đến tay người nhận ở nhiều địa phương khác nhau, bưu điện thường tổ chức một hệ thống phân loại thư phân cấp. Trước tiên, các thư đến cùng một tỉnh, thành phố sẽ được sắp chung vào một lô để gửi đến tỉnh thành tương ứng. Bưu điện các tỉnh thành này lại thực hiện công việc tương tự. Các thư đến cùng một quận, huyện sẽ được xếp vào chung một lô và gửi đến quận, huyện tương ứng. Cứ như vậy, các bức thư sẽ được trao đến tay người nhận một cách có hệ thông mà công việc sắp xếp thư không quá nặng nhọc.

Mô phỏng lại qui trình trên, gia sử mỗi phần tử ai trong day a[0], a[1], . . . , a[n-1] la một số nguyên co tối đa m chữ số. Phân loai cac phần tử nay lần lượt theo cac chữ số hang đơn vi, hang

chuc, hang trăm, . . . , tương tự việc phân loai thư theo tinh thanh, quận huyện, phường xa, . . .

Các bước cơ bản của thuật toán này là:

- Code mẫu C++:

```
void RadixSort(int* a, int n)
{
        int i, m = a[0], exp = 1;
        auto* b = new int[250000];
        for (i = 0; i < n; i++)
        {
                if (a[i] > m)
                        m = a[i];
        }
        while (m / exp > 0)
        {
                int bucket[10] = { 0 };
                for (i = 0; i < n; i++)
                        bucket[a[i] / exp % 10]++;
                for (i = 1; i < 10; i++)
                        bucket[i] += bucket[i - 1];
                for (i = n - 1; i >= 0; i--)
                        b[--bucket[a[i] / exp % 10]] = a[i];
                for (i = 0; i < n; i++)
                        a[i] = b[i];
                exp *= 10;
        }
        delete[] b;
}
```

Ví dụ minh hoạ từng bước:

- Ví dụ với mảng a = [964, 354, 368, 128, 495, 121].

- Đầu tiên ta sẽ chạy vòng for để tìm phần tử lớn nhất. Ta có được phần tử lớn nhất m = a[0] = 964.

- Ta bắt đầu chạy vòng lặp while:
    + Ta có m/exp = 964 > 0 nên điều kiện đúng.
    + Tạo mảng bucket[10] = 0 cho lần duyệt đầu tiên cho hàng đơn vị.
    + Chạy vòng for:
        ++ Ta có bucket[a[0]/1%10]++ = bucket[4]++ => bucket[4] = 1.
        ++ Ta có bucket[a[1]/1%10]++ = bucket[4]++ => bucket[4] = 2.
        ++ Ta có bucket[a[2]/1%10]++ = bucket[8]++ => bucket[8] = 1.
        ++ Ta có bucket[a[3]/1%10]++ = bucket[8]++ => bucket[8] = 2.

++ Ta có bucket[a[4]/1%10]++ = bucket[5]++ => bucket[5] = 1.
++ Ta có bucket[a[5]/1%10]++ = bucket[1]++ => bucket[1] = 1.
++ Sau khi chạy vòng lặp đầu tiên ta được mảng bucket = [0, 1, 0, 0, 2, 1, 0, 0, 2, 0].

+ Chạy vòng for thứ hai:
++ bucket[1] += bucket[0] => bucket[1] = 1.
++ bucket[2] += bucket[1] => bucket[2] = 1.
++ bucket[3] += bucket[2] => bucket[3] = 1.
++ bucket[4] += bucket[3] => bucket[4] = 3.
++ Tiếp tục đến khi hết vòng lặp ta được bucket = [0, 1, 1, 1, 3, 4, 4, 4, 6, 6].

+ Chạy vòng for thứ ba:
++ b[–bucket[a[5]/1%10] = a[5] => b[0] = 121.
++ b[–bucket[a[4]/1%10] = a[4] => b[3] = 495.
++ b[–bucket[a[3]/1%10] = a[3] => b[5] = 128.
++ b[–bucket[a[2]/1%10] = a[2] => b[4] = 368.
++ b[–bucket[a[1]/1%10] = a[1] => b[2] = 354.
++ b[–bucket[a[0]/1%10] = a[0] => b[1] = 964.
++ i<0 nên ta kết thúc vòng lặp.

+ Chạy vòng for thứ ba để gán các giá trị của b cho mảng a. Ta được mảng a = [121, 964, 354, 495, 368, 128].

+ Tới đây ta đã sắp xếp lại mảng theo hàng đơn vị.

- Ta tiếp tục sắp xếp với hàng chục và hàng trăm. Đến khi số lớn nhất chia cho exp không còn lớn hơn 0 (tức m/exp <= 0) thì ta dừng lại và kết thúc thuật toán. Ta được mảng sau khi đã sắp xếp a = [121, 128, 354, 368, 495, 964].

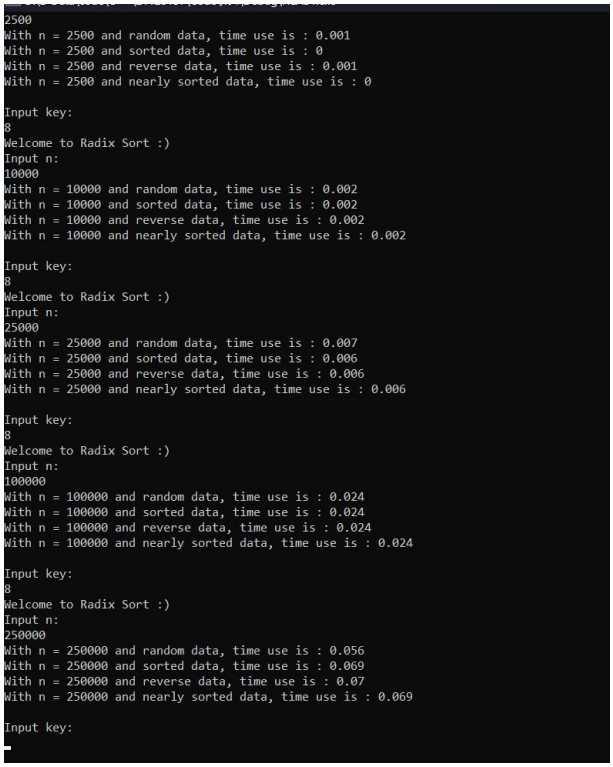
Đánh giá thuật toán:

Với Radix Sort, ta hoàn toàn có thể làm trên hệ cơ số R khác chứ không nhất thiết phải làm trên hệ nhị phân (ý tưởng cũng tương tự như trên), tuy nhiên quá trình phân đoạn sẽ không phải chia làm 2 mà chia thành R đoạn. Về độ phức tạp của thuật toán, ta thấy để phân đoạn bằng một bit thì thời gian sẽ là C.n để chia tất cả các đoạn cần chia bằng bit đó (C là hằng số). Vậy tổng thời gian phân đoạn bằng z bit sẽ là C.n.z. Trong trường hợp xấu nhất, độ phức tạp của Radix Sort là O(n.z). Và độ phức tạp trung bình của Radix Sort là O(n.min(z, lgn)).

Kết quả thực nghiệm:

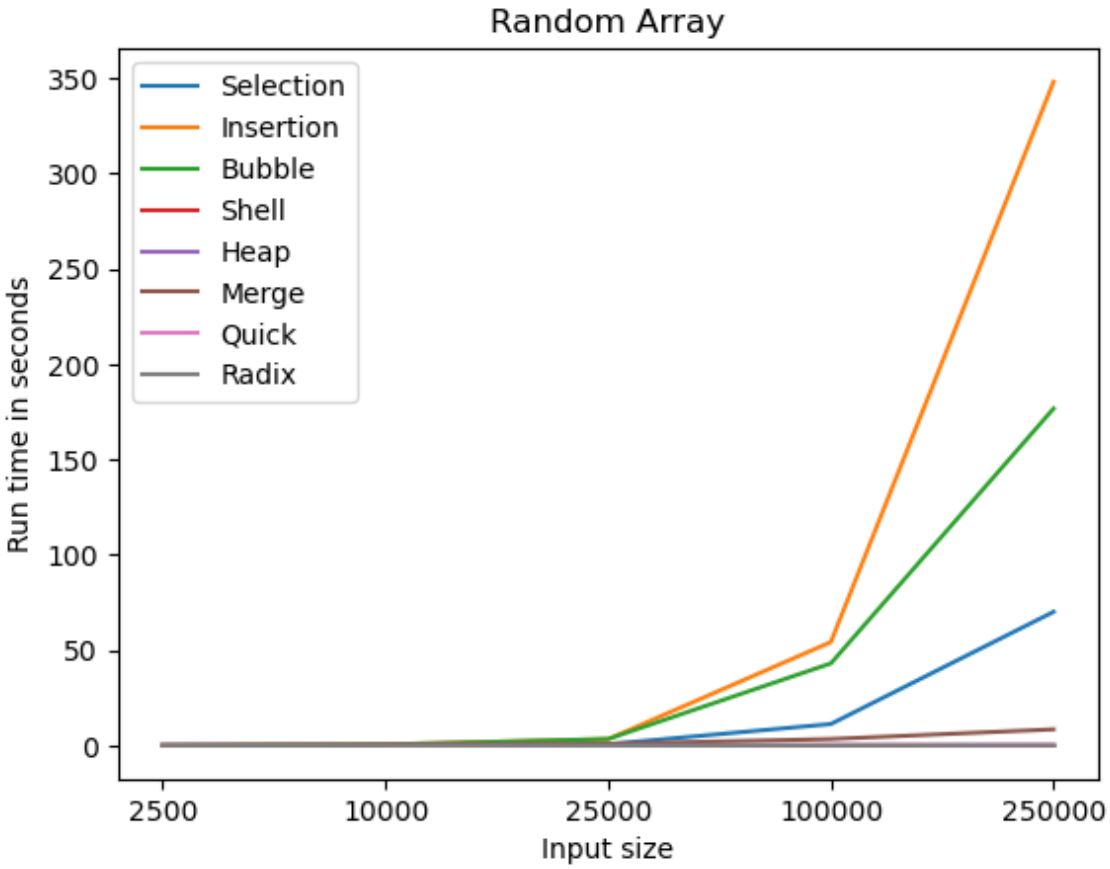
Size / Array Type	2500	10000	25000	100000	250000
Random	0.001 s	0.002 s	0.007 s	0.024 s	0.056 s
Sorted	0 s	0.002 s	0.006 s	0.024 s	0.069 s
Reverse	0.001 s	0.002 s	0.006 s	0.024 s	0.07 s
Nearly Sorted	0 s	0.002 s	0.006 s	0.024 s	0.069 s

Radix Sort time run



Problem 8: Tổng kết và đánh giá thuật toán

Random Array



- Code mẫu Python dùng để vẽ đồ thị:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Random array
random_selection = np.array([0.009, 0.115, 0.696, 11.284, 70.141 ])
random_insertion = np.array([0.042, 0.541, 3.354, 54.273, 348.209])
random_bubble = np.array([0.037, 0.58, 3.431, 43.088, 176.776])
random_shell = np.array([0, 0.002, 0.006, 0.024, 0.061])
random_heap = np.array([0.001, 0.005, 0.014, 0.054, 0.141])
random_merge = np.array([0.085, 0.343, 0.878, 3.414, 8.496 ])
random_quick = np.array([0.001, 0.001, 0.005, 0.018, 0.046])
random_radix = np.array([0.001, 0.002, 0.007, 0.024, 0.056])

plt.plot(random_selection, label = "Selection")
plt.plot(random_insertion, label = "Insertion")
plt.plot(random_bubble, label = "Bubble")
plt.plot(random_shell, label = "Shell")
plt.plot(random_heap, label = "Heap")
```

```
plt.plot(random_merge, label = "Merge")
plt.plot(random_quick, label = "Quick")
plt.plot(random_radix, label = "Radix")
plt.title("Random Array")
plt.xticks([0, 1, 2, 3, 4],[2500, 10000, 25000, 100000, 250000])
plt.xlabel("Input size")
plt.ylabel("Run time in seconds")
plt.legend()
```

- Nhận xét đối với mảng Random Array:

+ Các thuật toán có tốc độ ổn định trong khoảng từ 2500 đến dưới 25000 phần tử.

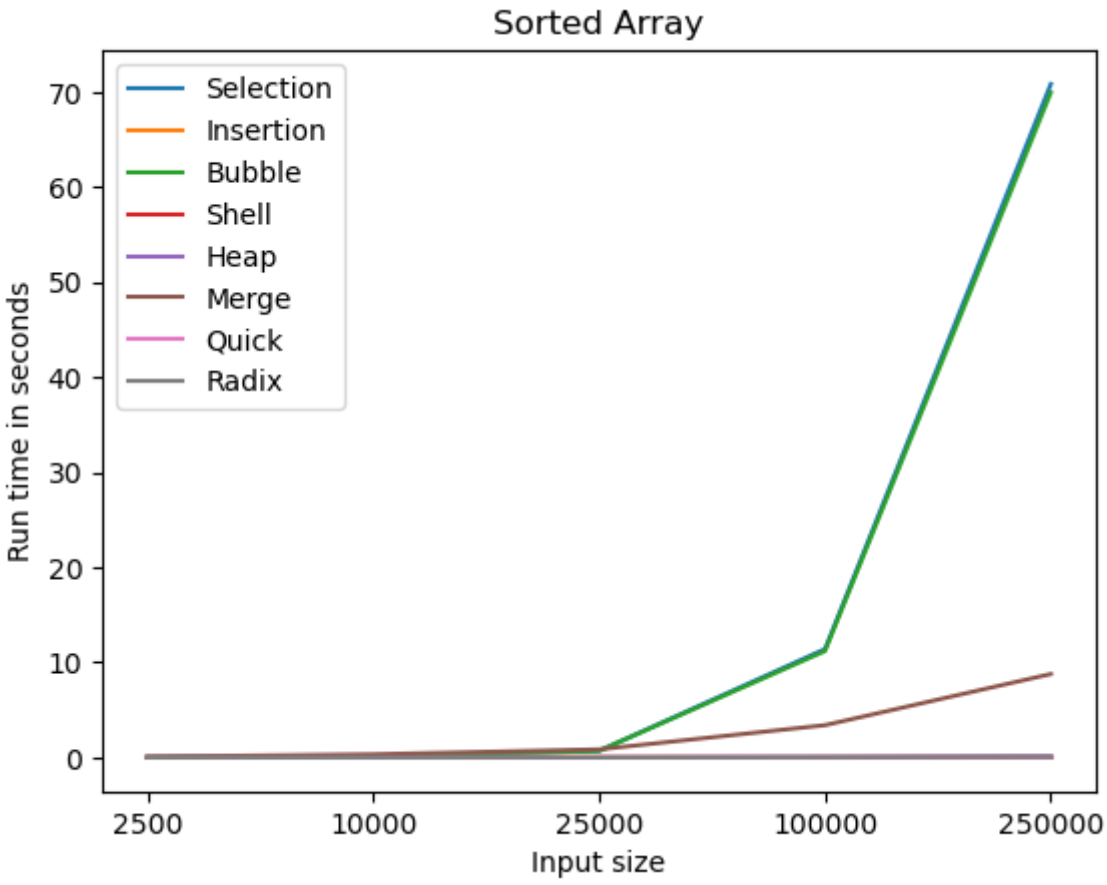
+ Đến mảng có hơn 25000 phần tử, các thuật toán có sự phân hoá rõ rệt.

+ Tổng quan, các thuật toán Shell, Heap và Quick và Radix Sort có tốc độ nhanh và ổn định từ 2500 phần tử cho đến 250000 phần tử.

+ Các thuật toán Selection, Insertion và Bubble có tốc độ chậm trong mảng Random từ mảng có 25000 phần tử trở lên.

+ Thuật toán chậm nhất trên mảng Random là thuật toán Insertion Sort (theo như thực nghiệm).

Sorted Array



- Code mẫu Python dùng để vẽ đồ thị:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Sorted array
sorted_selection = np.array([0.008, 0.112, 0.697, 11.41, 70.842])
sorted_insertion = np.array([0, 0.001, 0, 0, 0.001])
sorted_bubble = np.array([0.007, 0.118, 0.696, 11.197, 69.957])
sorted_shell = np.array([0, 0.001, 0.001, 0.005, 0.012])
sorted_heap = np.array([0.002, 0.004, 0.011, 0.05, 0.133])
sorted_merge = np.array([0.085, 0.337, 0.855, 3.4, 8.784 ])
sorted_quick = np.array([0.001, 0.001, 0.002, 0.005, 0.013])
sorted_radix = np.array([0, 0.002, 0.006, 0.024, 0.069])

plt.plot(sorted_selection, label = "Selection")
plt.plot(sorted_insertion, label = "Insertion")
plt.plot(sorted_bubble, label = "Bubble")
plt.plot(sorted_shell, label = "Shell")
plt.plot(sorted_heap, label = "Heap")
plt.plot(sorted_merge, label = "Merge")
plt.plot(sorted_quick, label = "Quick")
plt.plot(sorted_radix, label = "Radix")
plt.title("Sorted Array")
plt.xticks([0, 1, 2, 3, 4],[2500, 10000, 25000, 100000, 250000])
plt.xlabel("Input size")
plt.ylabel("Run time in seconds")
plt.legend()
```

- Nhận xét đối với mảng Sorted Array:

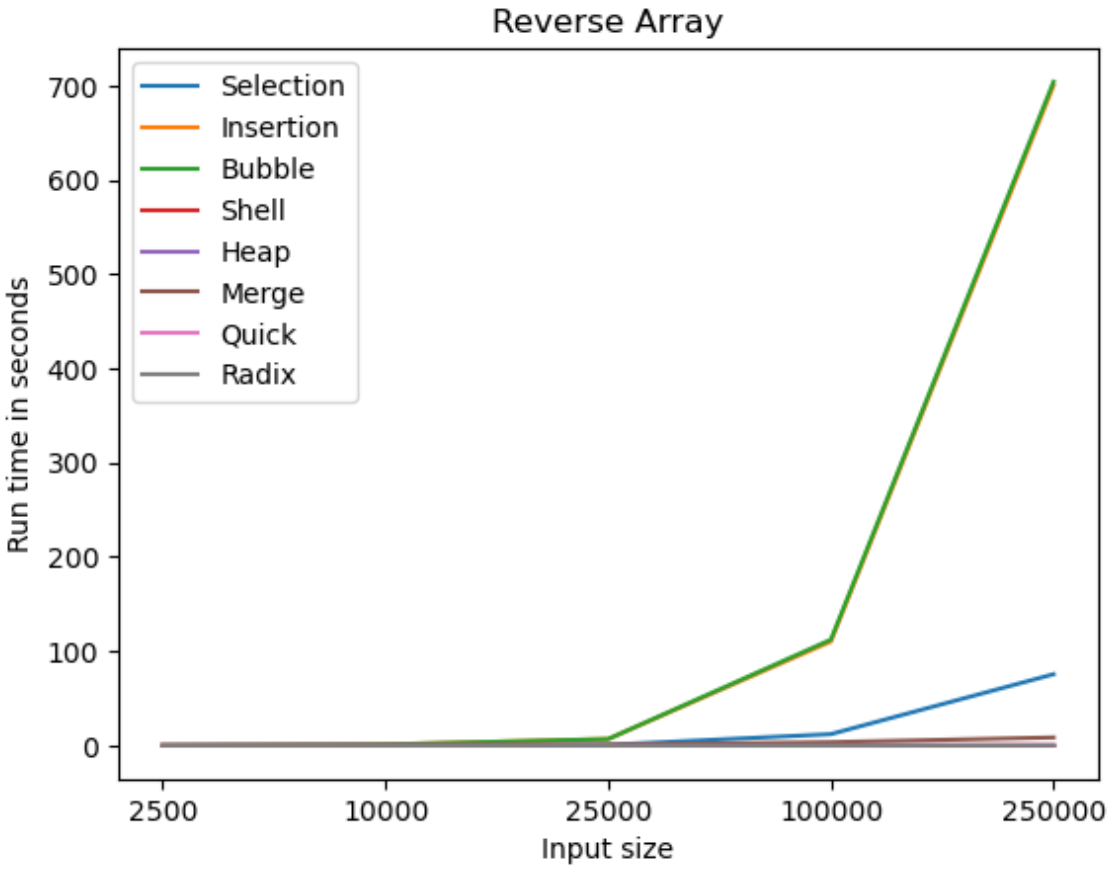
+ Tổng quan, các thuật toán có độ ổn định trong khoảng 2500 đến 25000 phần từ.

+ Các thuật toán Shell, Heap và Quick và Radix Sort có tốc độ nhanh và ổn định từ 2500 phần tử cho đến 250000 phần tử.

+ Các thuật toán Selection, Merge và Bubble có tốc độ chậm trong mảng Random từ mảng có 25000 phần tử trở lên.

+ Thuật toán chậm nhất trên mảng Sorted là thuật toán Selection Sort và Bubble Sort (theo như thực nghiệm).

Reverse Array



- Code mẫu Python dùng để vẽ đồ thị:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Reverse array
reverse_selection = np.array([0.007, 0.118, 0.737, 12.083, 75.639 ])
reverse_insertion = np.array([0.067, 1.074, 6.849, 110.262, 700.007])
reverse_bubble = np.array([0.069, 1.195, 6.888, 112.3, 704.043])
reverse_shell = np.array([0, 0.001, 0.001, 0.007, 0.019])
reverse_heap = np.array([0.001, 0.004, 0.01, 0.048, 0.129])
reverse_merge = np.array([0.092, 0.349, 0.848, 3.537, 8.559 ])
reverse_quick = np.array([0, 0, 0.002, 0.006, 0.014 ])
reverse_radix = np.array([0.001, 0.002, 0.006, 0.024, 0.07])

plt.plot(reverse_selection, label = "Selection")
plt.plot(reverse_insertion, label = "Insertion")
plt.plot(reverse_bubble, label = "Bubble")
plt.plot(reverse_shell, label = "Shell")
plt.plot(reverse_heap, label = "Heap")
plt.plot(reverse_merge, label = "Merge")
plt.plot(reverse_quick, label = "Quick")
```

```
plt.plot(reverse_radix, label = "Radix")
plt.title("Reverse Array")
plt.xticks([0, 1, 2, 3, 4],[2500, 10000, 25000, 100000, 250000])
plt.xlabel("Input size")
plt.ylabel("Run time in seconds")
plt.legend()
```

- Nhận xét các thuật toán đối với Reverse Array:

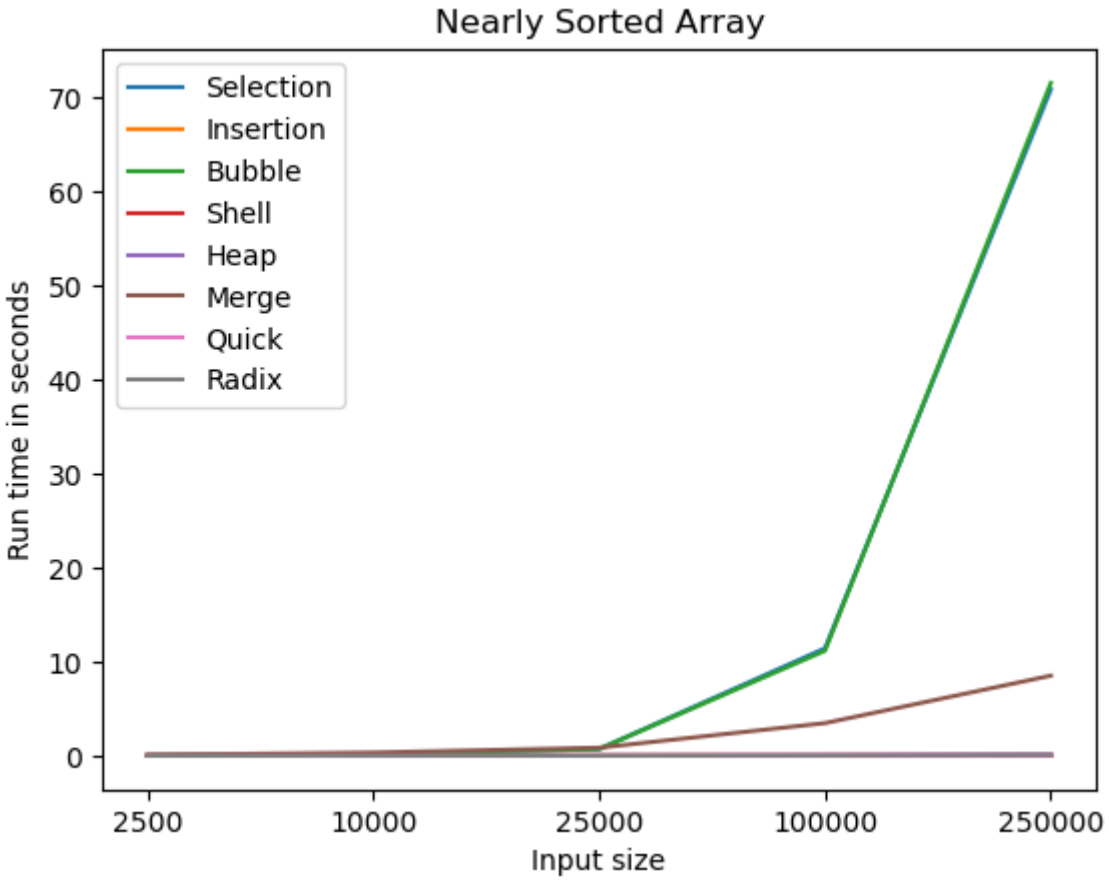
+ Tổng quan, các thuật toán có tốc độ ổn định trong khoảng từ 2500 đến dưới 25000 phần tử trong mảng Reverse Array.

+ Các thuật toán có tốc độ nhanh và ổn định trừ thuật toán Bubble Sort.

+ Mảng từ 25000 phần tử trở lên tốc độ chạy của các thuật toán mới bắt đầu có sự phân hoá (2 thuật toán Bubble và Selection trở nên chậm hơn nhiều so với các thuật toán khác).

+ Thuật toán chậm nhất trên mảng Reverse là thuật toán Bubble Sort (theo như thực nghiệm).

Nearly Sorted Array



- Code mẫu Python dùng để vẽ đồ thị:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
#Nearly sorted array
nearly_sorted_selection = np.array([0.007, 0.112, 0.699, 11.423, 70.881])
nearly_sorted_insertion = np.array([0, 0.002, 0.005, 0.005, 0.004])
nearly_sorted_bubble = np.array([0.007, 0.114, 0.702, 11.159, 71.558])
nearly_sorted_shell = np.array([0, 0.001, 0.002, 0.006, 0.013])
nearly_sorted_heap = np.array([0.001, 0.003, 0.011, 0.054, 0.134])
nearly_sorted_merge = np.array([0.088, 0.332, 0.845, 3.462, 8.511])
nearly_sorted_quick = np.array([0, 0, 0.001, 0.005, 0.013])
nearly_sorted_radix = np.array([0, 0.002, 0.006, 0.024, 0.069])

plt.plot(nearly_sorted_selection, label = "Selection")
plt.plot(nearly_sorted_insertion, label = "Insertion")
plt.plot(nearly_sorted_bubble, label = "Bubble")
plt.plot(nearly_sorted_shell, label = "Shell")
plt.plot(nearly_sorted_heap, label = "Heap")
plt.plot(nearly_sorted_merge, label = "Merge")
plt.plot(nearly_sorted_quick, label = "Quick")
plt.plot(nearly_sorted_radix, label = "Radix")
plt.title("Nearly Sorted Array")
plt.xticks([0, 1, 2, 3, 4],[2500, 10000, 25000, 100000, 250000])
plt.xlabel("Input size")
plt.ylabel("Run time in seconds")
plt.legend()
```

- Nhận xét các thuật toán đối với Nearly Sorted Array:

+ Tổng quan, các thuật toán có tốc độ ổn định trong khoảng từ 2500 đến dưới 25000 phần tử trong mảng Reverse Array.

+ Các thuật toán có tốc độ nhanh và ổn định trừ thuật toán Bubble Sort và Merge Sort.

+ Mảng từ 25000 phần tử trở lên tốc độ chạy của các thuật toán mới bắt đầu có sự phân hoá (2 thuật toán Bubble và Merge trở nên chậm hơn nhiều so với các thuật toán khác).

+ Thuật toán chậm nhất trên mảng Nearly Sorted là thuật toán Bubble Sort (theo như thực nghiệm).

Nguồn tham khảo

Website: https://www.geeksforgeeks.org/

Book: Giải thuật và Lập trình thầy Lê Minh Hoàng, ĐH Sư phạm Hà Nội 1999-2002