Đại học Khoa Học Tự Nhiên, ĐHQG-HCM

Khoa Công Nghệ Thông Tin
Bộ môn Thực hành Cấu Trúc Dữ Liệu và Giải Thuật

Báo cáo thuật toán sắp xếp

1. Trình bày các thuật toán đã cài đặt

1.1. Selection Sort

1.1.1. Ý tưởng của thuật toán

Sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất(giả sử với sắp xếp mảng tăng dần) trong đoạn chưa được sắp xếp và đổi cho phần tử nhỏ nhất đó với phần tử ở đầu đoạn chưa được sắp xếp(không phải đầu mảng). Thuật toán sẽ chia mảng làm 2 mảng con:

- Một mảng con đã được sắp xếp
- Một mảng con chưa được sắp xếp

Tại mỗi bước lặp của thuật toán, phần tử nhỏ nhất ở mảng con chưa được sắp xếp sẽ được di chuyển về đoạn đã sắp xếp.

1.1.2. Thuật toán

- **Bước 1:** Đặt MIN (vị trí của phần tử có giá trị nhỏ nhất) tại vị trí 0 của mảng.
- **Bước 2:** Tìm kiếm phần tử có giá trị tối thiểu từ vị trí MIN trong danh sách.
- **Bước 3:** Hoán đổi phần tử vừa tìm được với phần tử ở vị trí MIN.
- **Bước 4:** Tăng MIN để trỏ đến các phần tử tiếp theo.
- **Bước 5:** Lặp lại cho đến khi mảng đã được sắp xếp.

1.1.3. Mã giả

```
procedure selectionSort
   list  : mảng các phần tử
   n     : kích cỡ mảng

   for i = 1 to n - 1
   /* thiết lập phần tử hiện tại là min*/
      min = i;

      /* kiểm tra phần tử có là nhỏ nhất không */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* tráo đổi phần tử nhỏ nhất với phần tử hiện tại*/
      if indexMin ≠ i  then
         swap list[min] and list[i];
      end if

   end for

end
```

1.1.4. Đánh giá thuật toán

- Ta thấy, ở mỗi giai đoạn, ta cần lấy ra phần tử nhỏ nhất của phần chưa được sắp xếp, ban đầu phần này có n phần tử và ta cần $n-1$ phép so sánh, sau đó đưa phần tử này lên đầu thì phần chưa được sắp xếp chỉ còn $n-1$ phần tử, và cứ thế giảm dần về 1. Ta có số phép so sánh cần dùng là:

$$n-1+n-2+n-3+\ldots+2+1=\frac{n*(n-1)}{2}$$

- Do đó, số phép so sánh cần thực hiện là $\mathcal{O}(n^2)$. Mỗi giai đoạn ta cần một phép hoán vị, vì vậy ta cần $n-1$ phép hoán vị, hay $\mathcal{O}(n)$ phép hoán vị.

- Vậy độ phức tạp thời gian của thuật toán sắp xếp chọn là $\mathcal{O}(n^2)$.. Bộ nhớ cần dùng là $\mathcal{O}(1)$..
- Thuật toán sắp xếp chọn hoàn toàn không phụ thuộc vào các dữ liệu đầu vào, nghĩa là thời gian chạy không bị ảnh hưởng bởi cách dữ liệu được xếp. Do đó độ phức tạp trong các trường hợp tốt nhất, tệ nhất hay trung bình đều là $\mathcal{O}(n^2)$.
- **Tổng kết:**
  - Độ phức tạp thời gian:
    - Trường hợp tốt nhất: $\Omega(n^2)$
    - Trường hợp trung bình: $\Theta(n^2)$
    - Trường hợp xấu nhất: $\mathcal{O}(n^2)$
  - Độ phức tạp không gian: $\mathcal{O}(1)$

1.2. Merge Sort

1.2.1 .Ý tưởng của thuật toán:

Merge sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp.

- Hàm `merge()` được sử dụng để gộp hai nửa mảng.
- Hàm `merge(arr, l, m, r)` là tiến trình quan trọng nhất sẽ gộp hai nửa mảng thành 1 mảng sắp xếp, các nửa mảng là arr[l...m] và arr[m+1...r] sau khi gộp sẽ thành một mảng duy nhất đã sắp xếp.

1.2.2. Thuật toán

Cho hàm `mergeSort` có tham số đầu vào như sau `mergeSort(arr, l, m)`. Thực hiện lặp lại các bước dưới đây nếu `r > l`:

- **Bước 1:** Tìm chỉ số phần tử nằm giữa mảng để chia mảng thành 2 nửa: `middle m = (l+r)/2`
- **Bước 2:** Gọi đệ quy hàm `mergeSort` để cho mảng con đầu tiên: `mergeSort(arr, l, m)`
- **Bước 3:** Gọi đệ quy hàm `mergeSort` cho mảng con thứ hai: `mergeSort(arr, m+1, r)`
- **Bước 4**: So sánh từng phần tử trong hai mảng con ở **Bước 2** và **Bước 3**.
- **Bước 5:** Phần tử nào lớn hơn ta cho lại vào mảng chính.
- **Bước 6:** Quay lại **Bước 3** cho đến khi gặp phần tử cuối cùng ở một trong hai mảng con.
- **Bước 5:** Lấy các phần tử còn thừa ở hai mảng con vào lại mảng chính.

1.2.3. Mã giả

- Hàm mergeSort:

```
procedure mergesort(biến arr là một mảng, n là số phần tử)
    /* Nếu đã dịch chuyển hết các phần tử */
    if (l ⩾ r) then:
        return;
    /* Tìm chỉ số nằm giữa để chia mảng thành 2 nửa */
    var m = (l + r - 1)/2;
    /* Gọi đệ quy hàm mergeSort cho nửa đầu tiên */
    mergeSort(a,l,m);
    /*Gọi đệ quy hàm mergeSort cho nửa thứ hai */
    mergeSort(a,m+1,r);
    /* Gộp 2 nửa mảng đã sắp xếp */
    merge(a,l,m,r);
end
```

- Hàm merge:

```
procedure merge(arr,l,m,r)
    var n1 = m - l + 1; /* Số phần tử nửa mảng đầu tiên */
    var n2 = r - m; /* Số phần tử nửa mảng còn lại */
    /* Tạo mảng tạm */
    L[n1], R[n2];
    /* Copy giá trị tới mảng tạm */
    for (i = 0 to i < n1) do:
        L[i] = arr[l+i];
    end for
    for (j = 0 to j < n2) do:
        R[j] = arr[m + 1 + j];
    end for
    var i , j = 0;
    var k = l; /* Chỉ số index của mảng được trộn */
    while (i < n1 and j < n2) do:
```

```
        if (L[i] ⩽ R[j]) then:
            arr[k] = L[i];
            i = i + 1;
        else:
            arr[k] = R[j];
            j++;
        k++;
    /* Copy giá trị còn lại của mảng L[] nếu có*/
    while (i < n1) do:
        arr[k] = L[i];
        i++;
        k++;
    /* Tương tự copy giá trị còn lại của mảng R[] nếu có*/
    while (j < n2) do:
        arr[k] = R[j];
        j++;
        k++;
end
```

1.2.4. Đánh giá thuật toán

- Độ phức tạp của thuật toán sắp xếp trộn là $\mathcal{O}(n \log n)$ ở mọi trường hợp. Ta có thể chứng minh nó như sau:
  - Tạm gọi $T(n)$ là thời gian thực thi của thuật toán sắp xếp trộn cho mảng có n phần tử.
  - Ta có hệ thức đệ quy sau: $T(n) = 2 * T(\frac{n}{2}) + n$. Vì ta cần giải bài toán con cho 2 mảng con với độ dài $\frac{n}{2}$, và tốn n cho phép trộn 2 mảng con.

$$T(n) = 4 * T\left(\frac{n}{4}\right) + n + \left(\frac{n}{2}\right) * 2 = 4 * T\left(\frac{n}{4}\right) + n + n = 2^{\log n} * T\left(\frac{n}{2^{\log n}}\right)$$

- Bộ nhớ cần dùng của thuật toán là $\mathcal{O}(n + \log n)$ = $\mathcal{O}(n)$, với n là độ dài mảng, và $\log n$ tầng đệ quy.
- Tổng kết:
  - Độ phức tạp thời gian:
    - Trường hợp tốt nhất: $\Omega(n \log n)$
    - Trường hợp trung bình: $\Theta(n \log n)$
    - Trường hợp xấu nhất: $\mathcal{O}(n \log n)$
  - Độ phức tạp không gian: $\mathcal{O}(n)$

1.3. Heap Sort

1.3.1. Ý tưởng của thuật toán

- Chia các phần tử thành 2 mảng con:
  - 1 mảng các phần tử đã được sắp xếp.
  - 1 mảng các phần tử chưa được sắp xếp.
- Trong mảng chưa được sắp xếp, các phần tử lớn nhất sẽ được tách ra và đưa vào mảng đã được sắp xếp. Điều cải tiến ở Heap sort so với Selection sort ở việc sử dụng cấu trúc dữ liệu heap thay vì tìm kiếm tuyến tính (linear-time search) như Selection sort để tìm ra phần tử lớn nhất.
- Heap sort là thuật toán in-place, nghĩa là không cần thêm bất cứ cấu trúc dữ liệu phụ trợ trong quá trình chạy thuật toán. Tuy nhiên, giải thuật này không có độ ổn định (stability).

1.3.2. Thuật toán

- **Bước 1:** Xây dựng một `heap` từ dữ liệu nhập vào (có thể là min-heap hoặc max-heap, ở đây em sử dụng max-heap) như sau:
  - Nút cha sẽ luôn lớn hơn tất cả các nút con, nút gốc của heap sẽ là phần tử lớn nhất.
  - Heap được tạo thành phải là một cây nhị phân đầy đủ, tức ngoại trừ các nút lá, ở cùng một cấp độ các nút nhánh không được thiếu.
- **Bước 2:** Lúc này, phần tử có giá trị lớn nhất nằm ở vị trí node gốc của `heap`. Ta thay nó bằng phần tử có vị trí cuối cùng của `heap`. Đồng thời giảm kích thước của `heap` đi 1.
- **Bước 3:** Sắp xếp lại `heap` sau khi loại bỏ node gốc(có giá trị lớn nhất) để tìm phần tử có giá trị lớn nhất tiếp theo.
- **Bước 4:** Lặp lại **B2** và **B3** cho đến khi kích thước của `heap` lớn hơn 1.

1.3.3. Mã giả

- Hàm heapify:

```
procedure heapify(a,n,i)
    largest = i; /* Khởi tạo phần tử lớn nhất là node gốc*/
    l = 2*i+1; /* Node trái */
    r = 2*i+2; /* Node phải */
    /* Nếu node con bên trái lớn hơn node gốc */
    if (l < n và a[l] > a[largest])
        largest = l;
    end if
    /* Nếu node con bên phải lớn hơn node lớn nhất */
    if (r < n && a[r] > a[largest])
        largest = r;
    end if

    /* Nếu node lớn nhất không phải node gốc */
    if (largest ≠ i)
        swap a[i] and a[largest];
        Heapify(a,n,largest);
    end if

end
```

- Hàm heapSort:

```
procedure heapSort(a[],n)
    /* Xây dựng heap */
    for(i = n/2 - 1 to i ≥ 0) do:
        heapify(a,n,i);
    end for
    /*Lần lượt xóa một phần tử ra khỏi heap */
    for (i = n-1 tới i > 0) do:
        /*Di chuyển node root tới vị trí cuối cùng*/
        swap a[0] and a[i];
        /*Gọi hàm heapify*/
        heapify(a,i,0);
    end for
end
```

1.3.4. Đánh giá thuật toán

- Độ phức tạp về thời gian của hàm heapify là $\mathcal{O}(n \log n)$. Độ phức tạp thời gian giai đoạn build heap là $\mathcal{O}(n)$ và độ phức tạp thời gian tổng thể của heap sort là $\mathcal{O}(n \log n)$ vì ta cần thực hiện n lần điều chỉnh heap, mỗi lần sẽ mất chi phí lớn nhất là độ cao của cây nhị phân , hay $\mathcal{O}(\log n)$.
- **Tổng kết:**
  - Độ phức tạp thời gian:
    - Trường hợp tốt nhất: $\Omega(n \log n)$
    - Trường hợp trung bình: $\Theta(n \log n)$
    - Trường hợp xấu nhất: $\mathcal{O}(n \log n)$
  - Độ phức tạp không gian: $\mathcal{O}(1)$

1.4. Quick Sort

1.4.1. Ý tưởng của thuật toán

- Thuật toán sắp xếp quick sort là một thuật toán chia để trị(Divide and Conquer algorithm). Nó chọn một phần tử trong mảng làm điểm đánh dấu(pivot). Thuật toán sẽ thực hiện chia mảng thành các mảng con dựa vào pivot đã chọn. Dưới đây là một số cách để chọn pivot thường được sử dụng:
  - Luôn chọn phần tử đầu tiên của mảng.
  - Luôn chọn phần tử cuối cùng của mảng. (Ở đây em sử dụng cách này)
  - Chọn một phần tử random.
  - Chọn một phần tử có giá trị nằm giữa mảng(median element).
- Cho một mảng và một phần tử x là pivot. Đặt x vào đúng vị trí của mảng đã sắp xếp. Di chuyển tất cả các phần tử của mảng mà nhỏ hơn x sang bên trái vị trí của x, và di chuyển tất cả các phần tử của mảng mà lớn hơn x sang bên phải vị trí của x.

- Khi đó ta sẽ có 2 mảng con: mảng bên trai của x và mảng bên phải của x. Tiếp tục công việc với mỗi mảng con(chọn pivot, phân đoạn) cho tới khi mảng được sắp xếp.

1.4.2. Thuật toán

- **Bước 1:** Chọn một phần tử bất kì làm khóa pivot:
  - Ta thường chọn phần tử ngoài cùng bên phải làm pivot hoặc chọn bất kỳ một cách ngẫu nhiên và hoán đổi với phần tử ngoài cùng bên phải. Giả sử cho hai giá trị "Thấp" và "Cao" tương ứng với chỉ số đầu tiên và chỉ số cuối cùng.
- **Bước 2**: Phân vùng những phần tử nhỏ hơn khóa thì nằm bên trái của khóa, những phần tử lớn hơn khóa thì nằm bên phải của khóa và những phần tử bằng khóa có thể nằm bất cứ chỗ nào trên dãy.
- **Bước 3:** Chia mảng thành hai mảng con:
  - Mảng con trước vị trí pivot
  - Mảng con sau vị trí pivot
- **Bước 4:** Lặp lại các bước cho mảng con trái và phải một cách đệ quy đến khi mảng được sắp xếp.

1.4.3. Mã giả

- Hàm partition:

```
procedure partition (arr, low, high)
    /* pivot - Element at right most position */
    pivot = arr[high];
    i = (low - 1);  // Index of smaller element
    for (j = low to j ⩽ high-1) do:
         /*If current element is smaller than the pivot, swap the element with pivot*/
        if (arr[j] < pivot) then:
            i++;
            swap(arr[i], arr[j]);
        end if
    swap(arr[i + 1], arr[high]);
    return (i + 1);
end
```

- Hàm Quick Sort:

```
procedure quickSort(arr[],low, high)
    if (low < high) then:
    /*pivot_index is partitioning index, arr[pivot_index] is now at correct place in sorted array*/
        pivot_index = partition(arr, low, high);
        quickSort(arr, low, pivot_index - 1);  /* Before pivot_index*/
        quickSort(arr, pivot_index + 1, high); /* After pivot_index*/
    end if
 end
```

1.4.4. Đánh giá thuật toán

- Độ phức tạp của thuật toán sắp xếp nhanh phụ thuộc nhiều vào cách ta chọn phần tử chốt - pivot. Nếu ta chọn tốt, khi phần tử chốt chia mảng ra làm 2 phần có độ dài xấp xỉ bằng nhau thì độ phức tạp của Quick sort là $\mathcal{O}(n \log n)$. Ta có thể chứng minh nó như sau:
  - Tạm gọi $T(n)$ là thời gian thực thi của thuật toán Quick sort cho mảng có n phần tử.
  - Ta có hệ thức đệ quy sau: $T(n) = 2 * T(\frac{n}{2}) + n.$ Vì ta cần giải bài toán con cho 2 mảng con với độ dài $\frac{n}{2}$, và tốn n cho phép trộn 2 mảng con.

$$T(n) = 4 * T\left(\frac{n}{4}\right) + n + \left(\frac{n}{2}\right) * 2 = 4 * T\left(\frac{n}{4}\right) + n + n = 2^{\log n} * T\left(\frac{n}{2^{\log n}}\right) + n \log n = n + n \log n = \mathcal{O}(n \log n)$$

- Nhưng điều trên chỉ đúng khi ta chọn được các phần tử chốt tốt (khi phần tử là trung vị của dãy), nhưng nếu chọn không tốt thì độ phức tạp ở trường hợp xấu nhất là $\mathcal{O}(n^2)$(khi mà mỗi lần chọn phần tử chốt thì phân ra làm 2 dãy có độ dài 1 và $n - 1$). Ở trường hợp tốt nhất, thuật toán có đôp hức tạp $O(nlogn)$. Bộ nhớ cần dùng của thuật toán là $\mathcal{O}(n \log n)$ ở trường hợp trung bình, vì ta gọi đệ quy nên sẽ tốn chi phí bộ nhớ stack.
- Ở trường hợp tệ nhất, khi chọn chốt tệ, bộ nhớ cần dùng có thể lên tới $\mathcal{O}(n)$.Nếu ta chọn phần tử chốt một cách cố định thì rất dễ rơi vào trường hợp xấu,chẳng hạn nếu ta chọn phần tử đầu tiên làm phần tử chốt và mảng được sắp xếp giảm dần thì sẽ dẫn tới trường hợp mà ta đệ quy xâu n bước và dẫn tới tràn stack trong quá trình đệ quy.
- **Tổng kết:**
  - Độ phức tạp thời gian:
    - Trường hợp tốt nhất: $\Omega(n \log n)$
    - Trường hợp trung bình: $\Theta(n \log n)$

        - Trường hợp xấu nhất: $\mathcal{O}(n^2)$
    - Độ phức tạp không gian: $\mathcal{O}(n \log n)$

1.5. Bubble Sort:

1.5.1. Ý tưởng của thuật toán

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

1.5.2. Thuật toán

- **Bước 1:** Khởi tạo vị trí dừng là vị trí cuối cùng
- **Bước 2:** Duyệt mảng từ phần tử đầu tiên của mảng đến vị trí dừng.
- **Bước 3:** So sánh phần tử ở vị trí đang duyệt với phần tử liền kề nó. Nếu phần tử vị trí duyệt lớn hơn, hoán đổi vị trí hai phần tử.
- **Bước 4:** Giảm vị trí dừng đi 1 và lặp lại **B2** cho đến khi vị trí dừng = 0

1.5.3. Mã giả

```
procedure BubbleSort(list)

   for tất cả phần tử trong list
      if list[i] > list[i+1]
         swap(list[i], list[i+1]);
      end if
   end for

   return list;

end
```

1.5.4. Đánh giá thuật toán

- Ta thấy, số phép so sánh cần thực hiện là $n-1$ ở lần lặp đầu tiên, sau đó là $n-2, n-3, \ldots$. Số phép hoán vị trong trường hợp xấu nhất sẽ bằng số phép so sánh, còn trong trường hợp tốt nhất là 0 phép hoán vị.
- Trong trường hợp xấu nhất sẽ có $\mathcal{O}(n^2)$ phép so sánh, $\mathcal{O}(n^2)$ phép hoán vị. Suy ra độ phức tạp của thuật toán là $\mathcal{O}(n^2)$ trong trường hợp xấu nhất.
- Trong trường hợp tốt nhất thì ta cần $\mathcal{O}(n2)$ phép so sánh, $\mathcal{O}(1)$ phép hoán vị. Suy ra độ phức tạp của thuật toán là $\mathcal{O}(n^2)$ trong trường hợp xấu nhất.
- Độ phức tạp của thuật toán là $\mathcal{O}(n^2)$ trong trường hợp trung bình.
- **Tổng kết:**
    - Độ phức tạp thời gian:
        - Trường hợp tốt nhất: $\Omega(n)$
        - Trường hợp trung bình: $\Theta(n^2)$
        - Trường hợp xấu nhất: $\mathcal{O}(n^2)$
    - Độ phức tạp không gian: $\mathcal{O}(1)$

1.6. Insertion Sort

1.6.1. Ý tưởng của thuật toán

- Ta có mảng ban đầu gồm phần tử A[0] xem như đã sắp xếp, ta sẽ duyệt từ phần tử 1 đến $n-1$, tìm cách chèn những phần tử đó vào vị trí thích hợp trong mảng ban đầu đã được sắp xếp.

1.6.2. Thuật toán

Giả sử cho mảng A có n phần tử chưa được sắp xếp. Các bước thực hiện của thuật toán áp dụng trên mảng A như sau:

- **Bước 1:** Đặt phần tử cần chèn là phần tử thứ 2. Khi đó các phần tử trước phần tử cần chèn đã được sắp xếp tăng dần.
- **Bước 2:** Duyệt các phần tử đứng trước phần tử cần chèn, bắt đầu từ vị trí kế bên giảm dần đến vị trí ban đầu.
- **Bước 3:** Tìm phần tử đầu tiên nhỏ hơn phần tử cần chèn.
- **Bước 4:** Dịch chuyển mảng con các phần tử ngay sau phần tử nhỏ hơn đầu tiên đến phần tử cần chèn sang phải một đơn vị. Đồng thời chèn phần tử cần chèn vào ngay sau phần tử đầu tiên. Nếu không có phần tử nào nhỏ hơn, dịch mảng và chèn phần tử vào vị trí đầu tiên.
- **Bước 5:** Tăng vị trí của phần tử cần chèn lên 1 đơn vị, lặp lại **B2** cho đến vị trí cuối mảng.

1.6.3. Mã giả

```
procedure insertionSort( A : mảng phần tử )
   var holePosition;
   var valueToInsert;

   for i = 1 tới i length(A) do:

      /* chọn một giá trị để chèn */
      valueToInsert = A[i];
      holePosition = i;

      /*xác định vị trí cho phần tử được chèn */

      while holePosition > 0 và A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1];
         holePosition = holePosition -1;
      end while

      /* chèn giá trị tại vị trí trên */
      A[holePosition] = valueToInsert;

   end for

end
```

1.6.4. Đánh giá thuật toán

- Ta thấy, ở mỗi giai đoạn, ta tìm vị trí thích hợp cho một phần tử ở phần chưa được sắp xếp. Việc tìm vị trí thích hợp này cần nhiều nhất là $\mathcal{O}(n)$ phép so sánh và gán (nếu vị trí thích hợp nằm ở đầu phần đã sắp), và ở trường hợp tốt nhất là $\mathcal{O}(1)$ phép so sánh và gán (nếu vị trí thích hợp nằm ở cuối phần đã sắp).
- Ở trường hợp tốt nhất, khi dãy đã được sắp xếp thì ta cần tổng cộng $\mathcal{O}(n)$ phép so sánh và $\mathcal{O}(n)$ phép gán. Vậy, độ phức tạp của sắp xếp chèn ở trường hợp tốt nhất là $\mathcal{O}(n)$.
- Ở trường hợp xấu nhất, khi dãy đã được sắp xếp giảm dần, thì mọi giai đoạn ta đều cần $\mathcal{O}(n)$ phép gán và $\mathcal{O}(n)$ phép so sánh, từ đó suy ra độ phức tạp ở trường hợp xấu nhất là $\mathcal{O}(n^2)$.
- Trung bình thì độ phức tạp của thuật toán sắp xếp chèn là $\mathcal{O}(n^2)$. Độ phức tạp không gian của thuật toán là $\mathcal{O}(1)$.
- Thuật toán sắp xếp chèn phụ thuộc rất nhiều vào dữ liệu đầu vào, khi dữ liệu đầu vào đã được sắp xếp hoặc gần như được sắp xếp thì thuật toán sẽ chạy rất nhanh (độ phức tạp $\mathcal{O}(n)$) , nhưng nếu dữ liệu ngẫu nhiên hoặc dữ liệu được sắp xếp ngược thì thuật toán sẽ chạy rất lâu ví độ phức tạp ở trường hợp tệ nhất là $\mathcal{O}(n^2)$
- **Tổng kết:**
  - Độ phức tạp thời gian:
    - Trường hợp tốt nhất: $\Omega(n)$
    - Trường hợp trung bình: $\Theta(n^2)$
    - Trường hợp xấu nhất: $\mathcal{O}(n^2)$
  - Độ phức tạp không gian: $\mathcal{O}(1)$

1.7. Binary-Insertion Sort

1.7.1. Ý tưởng của thuật toán

- Binary Insertion Sort sử dụng tìm kiếm nhị phân để tìm vị trí thích hợp để chèn thay vì tìm kiếm tuyến tính như Insertion sort.

1.7.2. Thuật toán

- **Bước 1:** Đặt phần tử cần chèn là phần tử thứ 2. Khi đó các phần tử trước phần tử cần chèn đã được sắp xếp tăng dần.
- **Bước 2:** Duyệt các phần tử đứng trước phần tử cần chèn, bắt đầu từ vị trí kế bên giảm dần đến vị trí ban đầu.
- **Bước 3:** Tìm phần tử đầu tiên nhỏ hơn phần tử cần chèn bằng thuật toán Binary Search.
  - **3.1:** Bắt đầu với vị trí trái bằng 0, vị trí phải là vị trí ngay trước vị trí phần tử cần chèn.
  - **3.2:** Kiểm tra vị trí trái có lớn hơn hoặc bằng vị trí phải hay không. Nếu có trả về vị trí trái cộng 1 đơn vị nếu phần tử cần chèn lớn hơn phần tử vị trí trái, trả về vị trí trái nếu phần tử cần chèn nhỏ hơn hoặc bằng phần tử vị trí trái.
  - **3.3:** Tìm vị trí giữa bằng phần nguyên của trung bình cộng vị trí trái và vị trí phải.
  - **3.4:** Nếu phần tử vị trí giữa bằng phần tử cần chèn thì trả về vị trí trái cộng một đơn vị. Nếu phần tử vị trí giữa nhỏ hơn phần tử cần chèn thì quay lại bước 3.2 với vị trí trái bằng vị trí giữa cộng 1. Trường hợp còn lại thì quay lại bước 3.2 với

vị trí phải bằng vị trí giữa trừ 1.

- **Bước 4:** Dịch chuyển mảng con các phần tử ngay sau phần tử nhỏ hơn đầu tiên đến phần tử cần chèn sang phải một đơn vị.Đồng thời chèn phần tử cần chèn vào ngay sau phần tử đầu tiên. Nếu không có phần tử nào nhỏ hơn, dịch mảng và chèn phần tử vào vị trí đầu tiên.
- **Bước 5:** Tăng vị trí của phần tử cần chèn lên 1 đơn vị, lặp lại bước 2 cho đến vị trí cuối mảng.

1.7.3. Mã giả

- Hàm binarySearch

```
procedure binarySearch(int a, int item, int low, int high)
    if (high ⩽ low) then:
        return (item > a[low]) ? (low + 1):low;
    end if
    var mid = (low + high)/2;
    if (item == a[mid])
        return mid + 1;
    end if
    if (item > a[mid])
        return binarySearch(a,item, mid + 1, high)
    end if
return binarySearch(a,item,low,mid - 1);
```

- Hàm binaryInsertionSort

```
procedure binaryInsertionSort(int* a, int& n){
    var i, loc, j, k, selected;
    for (i = 1; to i < n) do:
        j = i - 1;
        selected = a[i];
    end for
        /* find location where selected sould be inseretd*/
        loc = binarySearch(a, selected, 0, j);
        /* Move all elements after location to create space*/
        while (j ⩾ loc) do:
            a[j + 1] = a[j];
            j--;
        end
        a[j + 1] = selected;
end
```

1.7.4. Đánh giá thuật toán

- Ta thấy, ở mỗi giai đoạn, ta tìm vị trí thích hợp cho một phần tử ở phần chưa được sắp xếp bằng thuật toán tìm kiếm nhị phân. Việc tìm vị trí thích hợp này cần $\mathcal{O}(\log n)$ phép so sánh. Sau đó ta cần chèn phần tử vào đúng vị trí của nó, việc chèn này nếu ở trường hợp tốt nhất là $\mathcal{O}(1)$ phép gán (nếu vị trí thích hợp nằm ở đầu phần đã sắp), còn trường hợp xấu nhất là $O(n)$phép gán (nếu vị trí thích hợp nằm ở cuối phần đã sắp).
- Ở trường hợp tốt nhất, khi dãy đã được sắp xếp thì ta cần tổng cộng $O(nlogn)$phép so sánh do tìm kiếm nhị phân và $\mathcal{O}(n)$ phép gán. Vậy, độ phức tạp của sắp xếp chèn ở trường hợp tốt nhất là $\mathcal{O}(n \log n)$.
- Ở trường hợp xấu nhất, khi dãy đã được sắp xếp giảm dần, thì mọi giai đoạn ta đều cần $O(n)$phép gán và có $\mathcal{O}(\log n)$ phép so sánh từ chặt nhị phân, từ đó suy ra độ phức tạp ở trường hợp xấu nhất là $\mathcal{O}(n^2 + n \log n) = \mathcal{O}(n^2)$.
- Giống như thuật toán sắp xếp chèn, thuật toán sắp xếp chèn nhị phân phụ thuộc rất nhiều vào dữ liệu đầu vào, khi dữ liệu đầu vào đã được sắp xếp hoặc gần như được sắp xếp thì thuật toán sẽ chạy rất nhanh(độ phức tạp $\mathcal{O}(\log n)$) , nhưng nếu dữ liệu ngẫu nhiên hoặc dữ liệu được sắp xếp ngược thì thuật toán sẽ chạy rất lâu vì độ phức tạp ở trường hợp tệ nhất là $\mathcal{O}(n^2)$.
- **Tổng kết:**
  - Độ phức tạp thời gian:
    - Trường hợp tốt nhất: $\Omega(n \log n)$
    - Trường hợp trung bình: $\Theta(n^2)$
    - Trường hợp xấu nhất: $\mathcal{O}(n^2)$
  - Độ phức tạp không gian: $\mathcal{O}(1)$

2. Kết quả thực nghiệm thời gian chạy của thuật toán

2.1. Môi trường thực nghiệm

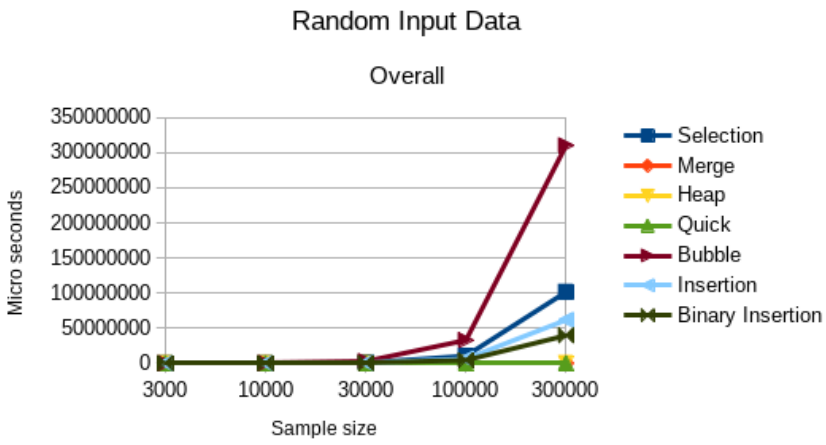
- Hệ điều hành và cấu hình máy sử dụng:
  - **OS**: Arch Linux - `Kernel: 5.10.3`
  - **CPU**: i5-8250U (8) @ 3.4GHz
  - **GPU**: Intel UHD Graphics 620
  - **Memory**: 8GB
- Ngôn ngữ lập trình sử dụng: C++.
- Code editor: Visual Studio Code.
- Chương trình được biên dịch bởi `GCC` (The GNU Compiler Collection) bằng cách thực thi lệnh như sau:

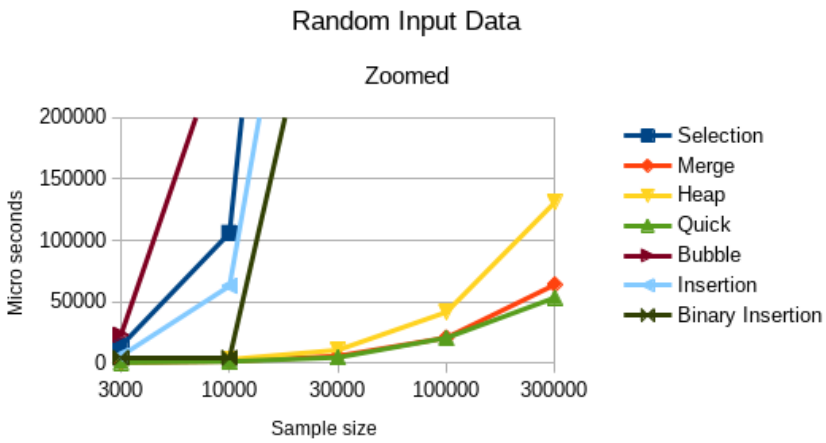
```
g++     main.cpp   -o main
```

- Thư viện dùng để đo thời gian: `chrono`.
- Công cụ vẽ biểu đồ: Libre Calc.
- Chú thích số liệu bảng thống kê:
  - Cột hàng ngang: Tên các thuật toán sắp xếp được thực nghiệm.
  - Cột hàng dọc: Cỡ mẫu dữ liệu đưa vào (3000,10000,30000,100000,300000).
  - Đơn vị thời gian đã đo: micro giây (microseconds - ms).
- Chú thích số liệu đồ thị:
  - Trục x: Cỡ mẫu dữ liệu đưa vào (3000,10000,30000,100000,300000).
  - Trục y: Thời gian thực thi quá trình sắp xếp (đơn vị: micro giây).

2.2. Dữ liệu đầu vào ngẫu nhiên

	Selection	Merge	Heap	Quick	Bubble	Insertion	Binary Insertion
3000	14254	578	1036	491	23209	5966	4328
10000	106041	1675	3135	1444	279402	62910	4328
30000	939702	5886	10739	4704	2784415	563331	386912
100000	10425610	20783	41614	20644	32780186	6232540	4272199
300000	10546352	20179	40497	16184	33075623	6476553	4525817

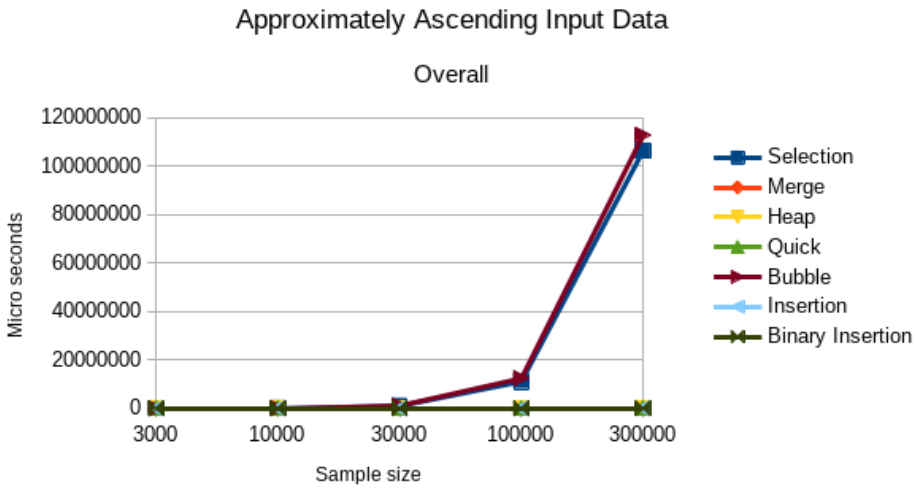


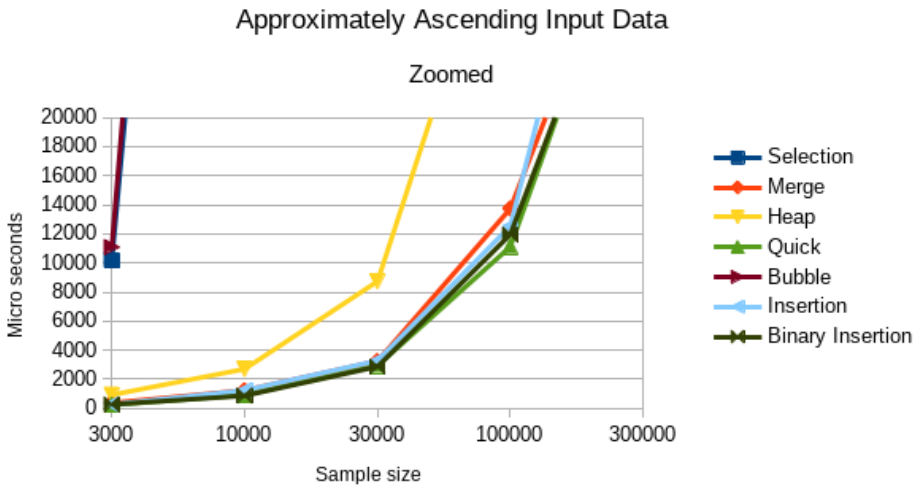


- **Nhận xét:**
  - Thuật toán cho kết quả chạy nhanh nhất: Quick Sort.
  - Thuật toán cho kết quả chạy chậm nhất: Bubble Sort
- **Giải thích:**
  - Với độ phức tạp thuật toán $\mathcal{O}(n^2)$ do việc liên tục hoán vị các phần tử cho nhau đã làm tăng thời gian chạy của thuật toán Bubble Sort lên đáng kể.

2.3. Dữ liệu đầu vào có thứ tự gần như tăng dần

	Selection	Merge	Heap	Quick	Bubble	Insertion	Binary Insertion
3000	10164	381	928	241	11086	274	276
10000	109103	1239	2703	923	120356	1236	871
300000	966670	3247	8732	2940	1073528	3278	2872
100000	11035946	13738	36699	11116	12361861	12535	11957
300000	106266388	37288	101976	36536	112746101	48404	35890

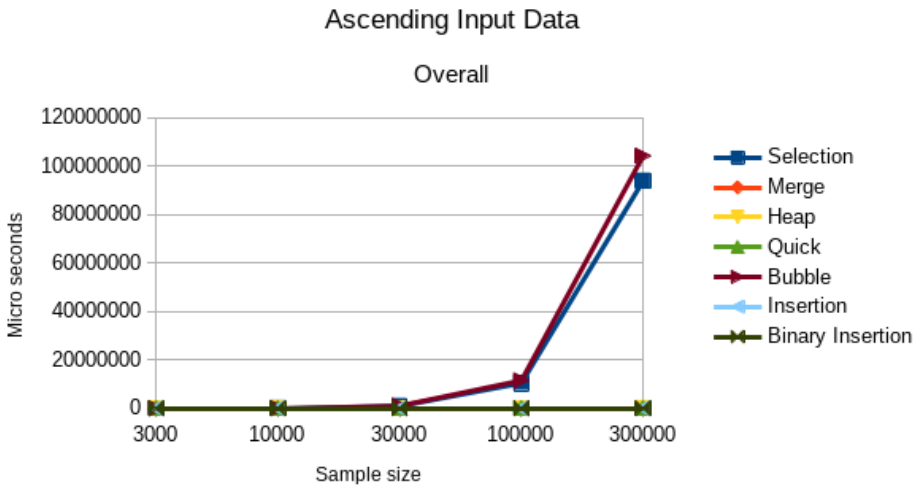


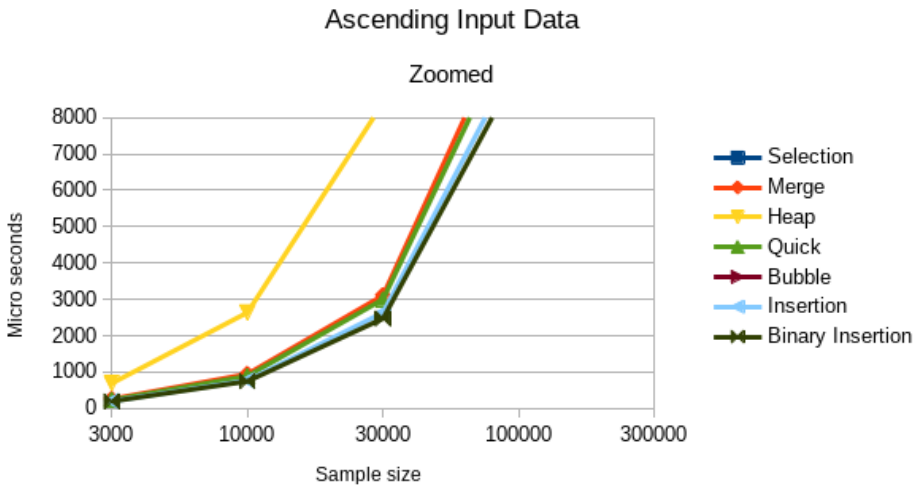


- **Nhận xét:**
  - Thuật toán cho kết quả chạy nhanh nhất: Quick Sort.
  - Thuật toán cho kết quả chạy chậm nhất: Bubble Sort, Selection Sort.
- **Giải thích:**
  - Ở đây ta thấy Selection và Bubble Sort có tốc độ gần như tương đương nhau (chậm nhất). Vì mảng chỉ gần như được sắp xếp tăng dần nên thuật toán vẫn cần phải nhiều lần gán thay đổi vị trí phần tử làm tăng thời gian chạy.

2.4. Dữ liệu đầu vào có thứ tự tăng dần

	Selection	Merge	Heap	Quick	Bubble	Insertion	Binary Insertion
3000	9428	258	689	234	10607	199	194
10000	105998	944	2636	887	115834	769	745
300000	937965	3107	8404	2998	1035496	2634	2473
100000	10403635	11229	32039	10810	11506515	9756	9345
300000	93876032	35981	100514	36936	104166814	33346	32561

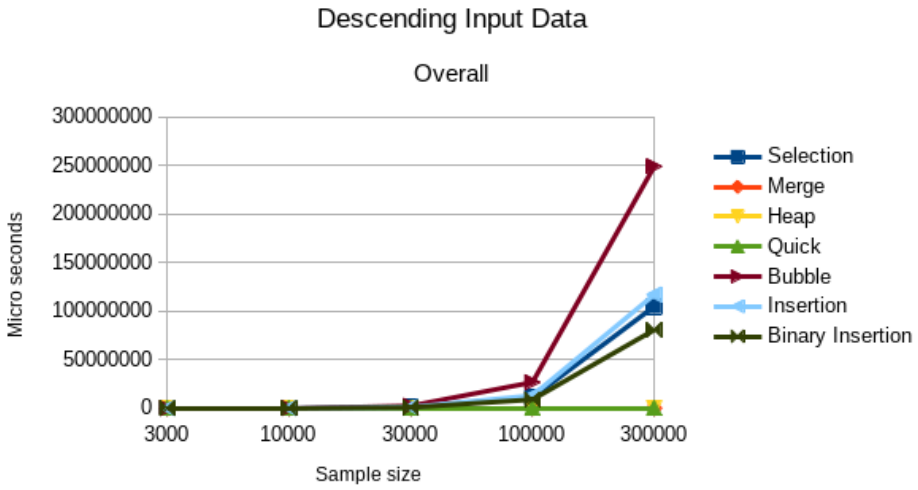


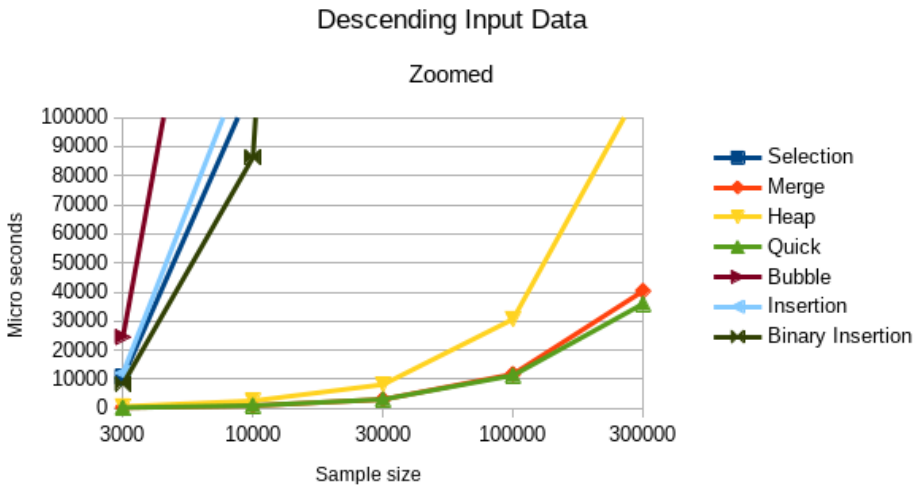


- **Nhận xét:**
  - Thuật toán cho kết quả chạy nhanh nhất: Binary Insertion Sort.
  - Thuật toán cho kết quả chạy chậm nhất: Bubble Sort, Selection Sort.
- **Giải thích:**
  - Với độ phức tạp thuật toán $\mathcal{O}(n^2)$ do việc liên tục hoán vị các phần tử cho nhau đã làm tăng thời gian chạy của thuật toán Bubble Sort lên đáng kể.

2.5. Dữ liệu đầu vào có thứ tự giảm dần

	Selection	Merge	Heap	Quick	Bubble	Insertion	Binary Insertion
3000	10704	269	669	244	24505	12093	8312
10000	111730	936	2595	961	265738	126072	86324
300000	991351	3082	8178	3102	2374859	1131777	772675
100000	11028355	11767	30664	11348	26650020	13023872	8837028
300000	104665903	40341	111905	35953	248948337	116882526	80584883





- **Nhận xét:**
  - Thuật toán cho kết quả chạy nhanh nhất: Quick Sort.
  - Thuật toán cho kết quả chạy chậm nhất: Bubble Sort.
- **Giải thích**:
  - Với độ phức tạp thuật toán $\mathcal{O}(n^2)$ do việc liên tục hoán vị các phần tử cho nhau đã làm tăng thời gian chạy của thuật toán Bubble Sort lên đáng kể.

3. Nhận xét tổng thể

3.1. Dựa vào trạng thái sắp xếp và kiểu dữ liệu đầu vào

Căn cứ vào 4 loại dữ liệu đầu vào, ta có thể dễ dàng nhận thấy được:

- **Thuật toán nhanh nhất:** Quick sort.
- **Thuật toán chậm nhất:** Bubble sort.

3.2. Phân loại

Ta có bảng thống kê độ phức tạp và tính ổn định của các thuật toán như sau:

Thuật toán	Tốt nhất	Trung bình	Xấu nhất	Bộ nhớ	Tính ổn định
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Không
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Có
Heap Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Không
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Không
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Có
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Có
Binary-Insertion Sort	$\Omega(n \log n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Có

Từ đó ta có thể phân loại các thuật toán sắp xếp theo:

- Độ phức tạp của thuật toán:
  - Thuật toán có độ phức tạp $\mathcal{O}(n^2)$:
    - Selection Sort
    - Insertion Sort
    - Binary Insertion Sort
    - Bubble Sort
  - Thuật toán có độ phức tạp $\mathcal{O}(n \log n)$:
    - Heap Sort
    - Merge Sort
    - Quick Sort
- Độ ổn định của thuật toán: Một thuật toán sắp xếp được gọi là ổn định nếu sau khi tiến hành sắp xếp vị trí tương đối giữa các phần tử bằng nhau không bị thay đổi.
  - Các thuật toán ổn định:
    - Insertion Sort
    - Binary-Insertion Sort

    - Bubble Sort
    - Merge Sort

  Trong các thuật toán ổn định, thuật toán Merge Sort chạy nhanh nhất.

- Các thuật toán không ổn định:
    - Selection Sort
    - Quick Sort
    - Heap Sort

  Trong các thuật toán không ổn định, thuật toán Quick Sort chạy nhanh nhất.

4. Tài liệu tham khảo

https://www.geeksforgeeks.org/sorting-algorithms/

https://www.tutorialspoint.com/data_structures_algorithms/

https://www.bigocheatsheet.com/

https://github.com/leduythuccs/Sorting-Algorithms