LAB 4: Các thuật toán sắp xếp

Nguyễn Tiến Dũng - 23001585

Ngày 2 tháng 11 năm 2025

1. Lý thuyết về thuật toán sắp xếp

Bài toán sắp xếp (Sorting Problem):

- **Input:** Dãy n phần tử $\langle a_1, a_2, ..., a_n \rangle$
- **Output:** Một hoán vị $\langle a'_1, a'_2, ..., a'_n \rangle$ sao cho $a'_1 \leq a'_2 \leq ... \leq a'_n$

Phân loại thuật toán sắp xếp:

- **Comparison-based sorting:** Dựa trên phép so sánh giữa các phần tử
  - Cận dưới lý thuyết: $\Omega(n \log n)$
  - Ví dụ: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort
- **Non-comparison sorting:** Không dựa trên phép so sánh
  - Có thể đạt độ phức tạp tuyến tính $O(n)$ trong điều kiện đặc biệt
  - Ví dụ: Counting Sort, Radix Sort, Bucket Sort
- **In-place sorting:** Sử dụng $O(1)$ hoặc $O(\log n)$ bộ nhớ phụ
- **Stable sorting:** Giữ nguyên thứ tự tương đối của các phần tử bằng nhau

1.1 So sánh các thuật toán sắp xếp

Algorithm	Best Case	Average Case	Worst Case	Stable	In-place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	No
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	Yes

2 Các thuật toán sắp xếp cơ bản

2.1 Bubble Sort

Ý tưởng: Thuật toán *Bubble Sort* so sánh các cặp phần tử liên tiếp trong mảng và hoán đổi nếu sai thứ tự. Sau mỗi lượt lặp, phần tử lớn nhất trong đoạn chưa sắp xếp sẽ "nổi" về cuối mảng. Lặp lại cho đến khi mảng được sắp xếp.

a) Viết pseudocode cho thuật toán Bubble Sort.

Algorithm 1: Pseudocode Bubble Sort cơ bản

Input: Mảng $A[0 \dots n-1]$
Output: Mảng A được sắp xếp tăng dần
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-i-1$ **do**
    if $A[j] > A[j+1]$ **then**
      Swap $A[j]$ và $A[j+1]$;

b) Cài đặt thuật toán Bubble Sort trong C++.

```cpp
#include <iostream>
using namespace std;

void BubbleSort(int a[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

c) Tối ưu hóa thuật toán bằng cách dừng sớm khi mảng đã được sắp xếp.

Algorithm 2: Pseudocode Bubble Sort tối ưu

Input: Mảng $A[0 \dots n-1]$
Output: Mảng A được sắp xếp tăng dần
for $i \leftarrow 0$ **to** $n-1$ **do**
    flag $\leftarrow$ false;
    for $j \leftarrow 0$ **to** $n-i-1$ **do**
        if $A[j] > A[j+1]$ **then**
            Swap $A[j]$ và $A[j+1]$;
            flag $\leftarrow$ true;
    if *flag* = *false* **then**
        break;

```cpp
#include <iostream>
using namespace std;

void BubbleSort_Optimized(int a[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        bool flag = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                flag = true;
            }
        }
        if (!flag) break;
    }
}
```

d) So sánh số phép so sánh và số phép hoán đổi giữa phiên bản cơ bản và phiên bản tối ưu với các bộ test khác nhau.

Cách thực hiện

Để so sánh, ta sửa hai hàm để đếm:

- `count`: tổng số lần thực hiện phép so sánh điều kiện.
- `swap`: tổng số lần hoán đổi (swap).

Chạy với các bộ test:

1. Mảng đã sắp xếp xuôi: `{1,2,3,4,5}`
2. Mảng đã sắp xếp ngược: `{5,4,3,2,1}`

3. Mảng ngẫu nhiên: {3,1,4,2,5}
4. Mảng có phần tử trùng: {2,3,2,1,3}

Cài đặt đếm số phép

```
#include <iostream>
#include <cmath>
using namespace std;

void Bubblesort(int a[], int n, int &count, int &swap){
    int temp;
    count = swap = 0;
    for(int i = 0 ; i < n - 1; i++){
        for(int j = 0; j < n - i - 1; j++){
            count++;
            if(a[j] > a[j + 1]){
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                swap++;
            }
        }
    }
}
void Bubblesortflag(int a[], int n, int &count, int &swap){
    int temp;
    count = swap = 0;
    for(int i = 0 ; i < n - 1; i++){
        bool flag = false;
        for(int j = 0; j < n - i - 1; j++){
            count++;
            if(a[j] > a[j + 1]){
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                swap++;
                flag = true;
            }
        }
        if(!flag){
            break;
        }
    }
}
void nhapmang(int a[], int n){
    cout << "Nhap mang:" << endl;
    for(int i = 0; i < n; i++){
        cin >> a[i];
    }
}
```

```
void inmang(int a[], int n){
    for (int i = 0; i < n; i++){
        cout << a[i] << " ";
    }
    cout << endl;
}
int main(){
    int n;
    int a[n], b[n];
    int count1, swap1, count2, swap2;
    cout << "Nhap so phan tu cua mang: ";
    cin >> n;
    nhapmang(a, n);

    for(int i = 0; i < n; i++){
        b[i] = a[i];
    }

    Bubblesort(a, n, count1, swap1);
    Bubblesortflag(b, n, count2, swap2);

    cout << "\nKet qua phien ban co ban:\n";
    inmang(a, n);
    cout << "So phep so sanh: " << count1 << endl;
    cout << "So phep hoan doi: " << swap1 << endl;

    cout << "\nKet qua phien ban toi uu:\n";
    inmang(b, n);
    cout << "So phep so sanh: " << count2 << endl;
    cout << "So phep hoan doi: " << swap2 << endl;

    return 0;
}
```

Bảng kết quả minh họa (n = 5)

Test case	Phiên bản	Số so sánh	Số hoán đổi	Ghi chú
Đã sắp xếp xuôi {1,2,3,4,5}	Cơ bản	10	0	chạy đủ vòng
	Tối ưu	4	0	dừng sớm sau 1 lượt
Đảo ngược {5,4,3,2,1}	Cơ bản	10	10	trường hợp xấu nhất
	Tối ưu	10	10	không khác biệt
Ngẫu nhiên {3,1,4,2,5}	Cơ bản	10	4	
	Tối ưu	8	4	giảm vài phép so sánh
Phần tử trùng {2,3,2,1,3}	Cơ bản	10	4	
	Tối ưu	7	4	dừng sớm hơn

Bảng 1: So sánh số phép so sánh và hoán đổi (ví dụ minh họa n=5)

Nhận xét

- Phiên bản tối ưu giúp **giảm nhiều phép so sánh** khi mảng đã (hoặc gần) sắp xếp, dẫn tới thời gian thực thi nhỏ hơn trong các trường hợp tốt.
- Trong trường hợp xấu nhất (mảng đảo ngược), cả hai phiên bản đều thực hiện số phép so sánh/hoán đổi tương đương và vẫn có độ phức tạp $O(n^2)$.
- Với mảng có phần tử trùng hoặc ngẫu nhiên, phiên bản tối ưu thường giảm được một số vòng, tức giảm một phần phép so sánh nhưng không thay đổi nhiều số hoán đổi tổng thể.

Kết luận

Bubble Sort là thuật toán đơn giản, dễ hiểu, nhưng hiệu quả kém với dữ liệu lớn do độ phức tạp $O(n^2)$. Tuy nhiên, biến thể tối ưu với cờ `flag` là một cải tiến nhỏ nhưng hữu ích khi dữ liệu gần như đã sắp xếp.

2.2 Selection Sort

Ý tưởng: Thuật toán *Selection Sort* tìm và đưa phần tử nhỏ nhất trong phần chưa sắp xếp và đặt nó vào vị trí đầu của phần chưa sắp xếp đó.

a. Pseudocode cho thuật toán Selection Sort

Algorithm 3: Selection Sort (Cơ bản)

Input: Mảng $A[1..n]$
Output: Mảng A sau khi đã được sắp xếp tăng dần
for $i \leftarrow 1$ **to** $n-1$ **do**
    $min \leftarrow i$;
    for $j \leftarrow i+1$ **to** n **do**
        if $A[j] < A[min]$ **then**
            $min \leftarrow j$;
    Swap $A[i]$ và $A[min]$;

b. Cài đặt C++

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

```
    }
}
```

c. Giải thích độ phức tạp

Dù mảng đã được sắp xếp hay chưa, thuật toán vẫn phải duyệt toàn bộ các phần tử để tìm phần tử nhỏ nhất trong mỗi vòng lặp. Do đó, số phép so sánh không thay đổi, luôn là:

$$\frac{n(n-1)}{2}$$

⇒ Độ phức tạp thời gian là $O(n^2)$ cho cả **Best case** và **Worst case**.

d. Chứng minh số lần hoán đổi là $n-1$

Trong mỗi vòng lặp ngoài (từ 1 đến $n-1$):

- Ta chọn ra phần tử nhỏ nhất trong phần chưa sắp xếp.
- Nếu phần tử đó chưa đúng vị trí, ta chỉ hoán đổi đúng một lần.

⇒ Mỗi vòng lặp ngoài có tối đa một lần hoán đổi. ⇒ Tổng số lần hoán đổi là : n - 1. Đây cũng là số lần hoán đổi tối thiểu, vì mỗi lần hoán đổi giúp đặt đúng vị trí cho ít nhất một phần tử.

2.3 Insertion Sort

Ý tưởng : Xây dựng mảng sắp xếp bằng cách chèn từng phần tử từ mảng chưa sắp xếp vào vị trí thích hợp.

a.Viết pseudocode cho thuật toán Insertion Sort

Algorithm 4: Insertion Sort

Input: Mảng $A[1..n]$
Output: Mảng A sau khi đã được sắp xếp tăng dần
for $i \leftarrow 2$ **to** n **do**
    $key \leftarrow A[i]$;
    $j \leftarrow i-1$;
    while $j > 0$ ***and*** $A[j] > key$ **do**
        $A[j+1] \leftarrow A[j]$;
        $j \leftarrow j-1$;
    $A[j+1] \leftarrow key$;

b.Cài đặt thuật toán Insertion Sort

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

c.Cài đặt phiên bản Binary Insertion Sort (sử dụng binary search để tìm vị trí chèn). Phân tích độ phức tạp và so sánh với phiên bản thường.

```
int binarysearch(int a[], int m, int left, int right){
    while (left <= right){
        int mid = left + (right - left)/2;
        if (a[mid] > m)
            right = mid - 1;
        else
            left = mid + 1;
        }
    return left;
}
void binaryInsertionsort(int a[], int n){
    for (int i = 1; i <  n; i++){
        int m = a[i];
        int pos = binarysearch(a, m, 0, i-1);
        for(int j = i - 1; j >= pos; j--){
            a[j+1] = a[j];
        }
        a[pos] = m;
    }
}
```

Thuật toán	Tìm vị trí chèn	Dịch phần tử	Tổng
Insertion Sort	$O(n)$	$O(n)$	$O(n^2)$
Binary Insertion Sort	$O(\log n)$	$O(n)$	$O(n^2)$

Bảng 2: So sánh độ phức tạp của Insertion Sort và Binary Insertion Sort

d.Giải thích tại sao Insertion Sort hiệu quả với Mảng nhỏ (n < 50) và mảng gần như đã sắp xếp.

1. Mảng nhỏ ($n < 50$):
   - Overhead của thuật toán đơn giản rất thấp.
   - O(n^2) chưa trở nên tồi tệ với n nhỏ.
2. Mảng gần như sắp xếp:
   - Số lần dịch các phần tử rất ít, gần như chỉ chèn một vài phần tử.
   - Thời gian thực tế gần O(n) thay vì O(n^2).

   ⇒ Insertion Sort là lựa chọn lý tưởng cho mảng nhỏ hoặc mảng gần như đã sắp xếp nhờ tính đơn giản và hiệu quả trên dữ liệu gần sắp xếp.

3 Các thuật toán sắp xếp hiệu quả

3.1 Merge Sort

Ý tưởng: Sử dụng kỹ thuật chia để trị (Divide and Conquer):

- **Chia:** Chia mảng thành 2 nửa
- **Trị:** Sắp xếp đệ quy từng nửa
- **Hợp nhất:** Trộn 2 mảng con đã sắp xếp

a. Viết pseudocode cho thuật toán Merge Sort và hàm Merge.

Algorithm 5: Thuật toán Merge Sort và hàm Merge

Input: Mảng A, chỉ số $left, right$
Output: Mảng A sau khi được sắp xếp tăng dần
Function `Merge`$(A, left, mid, right)$**:**
    $n_1 \leftarrow mid - left + 1$;
    $n_2 \leftarrow right - mid$;
    Tạo mảng tạm $L[1..n_1]$, $R[1..n_2]$;
    for $i \leftarrow 1$ ***to*** n_1 **do**
        $L[i] \leftarrow A[left + i - 1]$;
    end
    for $j \leftarrow 1$ ***to*** n_2 **do**
        $R[j] \leftarrow A[mid + j]$;
    end
    $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow left$;
    while $i \leq n_1$ ***and*** $j \leq n_2$ **do**
        if $L[i] \leq R[j]$ **then**
            $A[k] \leftarrow L[i]$; $i \leftarrow i + 1$;
        end
        else
            $A[k] \leftarrow R[j]$; $j \leftarrow j + 1$;
        end
        $k \leftarrow k + 1$;
    end
    while $i \leq n_1$ **do**
        $A[k] \leftarrow L[i]$;
        $i \leftarrow i + 1$; $k \leftarrow k + 1$;
    end
    while $j \leq n_2$ **do**
        $A[k] \leftarrow R[j]$;
        $j \leftarrow j + 1$; $k \leftarrow k + 1$;
    end
Function `MergeSort`$(A, left, right)$**:**
    if $left < right$ **then**
        $mid \leftarrow \lfloor (left + right)/2 \rfloor$;
        `MergeSort`$(A, left, mid)$;
        `MergeSort`$(A, mid + 1, right)$;
        `Merge`$(A, left, mid, right)$;
    end

b. Công thức truy hồi và phân tích bằng Master Theorem

Giả sử mảng có n phần tử. Mỗi lần ta chia mảng làm 2 nửa, mỗi nửa có thời gian chạy $T(n/2)$. Khi hợp lại, ta tốn thời gian $O(n)$ (vì phải duyệt qua cả hai nửa để trộn).

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Áp dụng **Master Theorem:**

$$a = 2, \quad b = 2, \quad f(n) = n$$

Ta có:

$$\log_b a = \log_2 2 = 1$$

$$f(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

Vì $f(n) = \Theta(n^{\log_b a})$, theo **Trường hợp 2 của Master Theorem**:

$$T(n) = \Theta(n \log n)$$

Độ phức tạp thời gian:

$$\begin{cases} \text{Trung bình: } O(n \log n) \\ \text{Xấu nhất: } O(n \log n) \\ \text{Tốt nhất: } O(n \log n) \end{cases}$$

c. Cài đặt Merge Sort trong C++ với cả hai cách:

- Top-down (đệ quy)

```
void merge(int a[], int left, int mid, int right){
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int *L = new int[n1];
    int *R = new int[n2];

    for(int i = 0; i < n1; i++){
        L[i] = a[left + i];
    }
    for(int j = 0; j < n2; j++){
        R[j] = a[mid + 1 + j];
    }
    int i = 0, j = 0;
    while(i < n1 && j < n2){
        if (L[i] <= R[j]){
            a[left++] = L[i++];
        }
        else{
            a[left++] = R[j++];
        }
    }
    while(i < n1) a[left++] = L[i++];
    while(j < n2) a[left++] = R[j++];

    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);
    }
}
```

- Bottom-up (lặp)

```
void mergesortiterative(int a[], int n){
    for(int iter = 1; iter < n; iter*=2){
        for(int left = 0; left < n - iter + 1; left +=  2*
           iter){
            int mid = left + iter - 1;
            int right = min(left + 2 *iter - 1, n-1);
            merge(a,left, mid, right);
        }
    }
}
```

d. Chứng minh Merge Sort là ổn định

Định nghĩa. Một thuật toán sắp xếp gọi là *ổn định* nếu với mọi cặp phần tử có khóa bằng nhau, thứ tự tương đối của chúng trong kết quả sắp xếp giữ nguyên như thứ tự ban đầu. **Ý tưởng.** Merge Sort chia mảng thành hai nửa, sắp xếp đệ quy hai nửa rồi ghép lại bằng hàm `merge`. Nếu khi so sánh phần tử từ hai nửa, ta chọn phần tử từ nửa trái trước khi hai khóa bằng nhau (ví dụ dùng điều kiện `L[i] <= R[j]`), thì thao tác ghép này bảo toàn thứ tự tương đối của các phần tử có khóa bằng nhau. Do đó Merge Sort ổn định.

Chứng minh (quy nạp).

- *Cơ sở:* Với $n = 0$ hoặc $n = 1$ mảng đã sắp xếp và ổn định.
- *Bước quy nạp:* Giả sử với mọi $m < n$, Merge Sort sắp xếp ổn định. Xét mảng có n phần tử. Sau khi chia, hai nửa có kích thước nhỏ hơn n và theo giả thiết quy nạp chúng được sắp xếp ổn định. Áp dụng luận điểm: nếu `merge` khi gặp hai phần tử bằng nhau luôn lấy phần tử từ nửa trái trước, thì thao tác ghép hai dãy đã sắp xếp ổn định sẽ cho kết quả ổn định. Do đó toàn bộ Merge Sort trên n phần tử là ổn định.

Nhận xét thực hành: Trong cài đặt, cần sử dụng so sánh dạng `if (L[i] <= R[j])` để đảm bảo tính ổn định. Nếu dùng `<` thì thuật toán có thể không ổn định.

e. Vì sao Merge Sort cần $O(n)$ bộ nhớ phụ?

Trong quá trình trộn (*merge*), thuật toán cần sử dụng hai mảng tạm L và R để lưu trữ hai nửa của mảng con. Giả sử ta đang gộp hai mảng con có kích thước lần lượt là n_1 và n_2, khi đó:

$$n_1 + n_2 = n$$

Tổng kích thước bộ nhớ cần cấp phát cho hai mảng tạm là $O(n)$.

Sau khi quá trình gộp hoàn tất, các phần tử từ hai mảng tạm L và R được sao chép ngược trở lại vào mảng chính A.

→ Do đó, bộ nhớ phụ của Merge Sort là $O(n)$.

3.2 Quick Sort

Ý tưởng : Sử dụng kỹ thuật chia để trị với phương pháp partition:

- **Chọn pivot :** Chọn một phần tử làm pivot
- **Partition :** Chia mảng thành 2 phần: nhỏ hơn pivot và lớn hơn pivot
- **Đệ quy :** Sắp xếp đệ quy hai phần

a. Viết pseudocode cho thuật toán Quick Sort và hàm Partition.

Algorithm 6: Thuật toán Quick Sort và hàm Partition (chọn pivot đầu mảng)

Input: Mảng a, chỉ số n, m
Output: Mảng a sau khi được sắp xếp tăng dần
Function `Partition`(a, n, m)**:**
  $pivot \leftarrow a[n]$;
  $i \leftarrow n + 1$;
  $j \leftarrow m$;
  while *true* **do**
    while $i \leq j$ ***and*** $a[i] \leq pivot$ **do**
      $i \leftarrow i + 1$;
    end
    while $i \leq j$ ***and*** $a[j] > pivot$ **do**
      $j \leftarrow j - 1$;
    end
    if $i > j$ **then**
      break;
    end
    swap$(a[i], a[j])$;
  end
  swap$(a[n], a[j])$ `// Đưa pivot về đúng vị trí`
  return j;
Function `QuickSort`(a, n, m)**:**
  if $n < m$ **then**
    $p \leftarrow$ `Partition`(a, n, m);
    `QuickSort`$(a, n, p - 1)$;
    `QuickSort`$(a, p + 1, m)$;
  end

b. Cài đặt Quick Sort với các phương pháp chọn pivot khác nhau (đầu mảng, cuối mảng, ngẫu nhiên).

1. Đầu mảng

```
int partition_low(int a[], int n, int m){
    int pivot = a[n];
    int i = n + 1;
    int j = m;
    while(true){
        while(i <= j && a[i] <= pivot){
            i++;
        }
        while(i <= j && a[j] > pivot){
            j--;
        }
        if(i>j) break;
        swap(a[i], a[j]);
    }
    swap(a[n], a[j]);
    return j;
}
```

2. Cuối mảng

```
int partition_high(int a[], int n, int m){
    int pivot1 = a[m];
    int i = n;
    int j = m-1;
    while(true){
        while(i <=j && a[i] <= pivot1){
            i++;
        }
        while(i <=j && a[j] > pivot1){
            j--;
        }
        if(i > j) break;
        swap(a[i], a[j]);
    }
    swap(a[i], a[m]);
    return i;
}
```

3. Ngẫu nhiên

```
int partition(int a[], int n, int m){
    int pivot2 = n + rand() % (m-n+1);
    int pivot3 = a[pivot2];
    int i=n, j=m;
    while(true){
        while(i <= j && a[i] <= pivot3){
            i++;
        }
        while(i <= j && a[j] > pivot3){
            j--;
        }
        if(i > j) break;
```

```
        swap(a[i], a[j]);
    }
    swap(a[pivot2], a[j]);
    return j;
}
```

Trường hợp mảng	Pivot đầu/cuối	Pivot ngẫu nhiên	Giải thích
Mảng ngẫu nhiên	Hiệu quả trung bình $\mathbf{O(n \log n)}$	$\mathbf{O(n \log n)}$	Cả hai hoạt động tốt, mảng được chia tương đối cân bằng.
Mảng đã sắp xếp	Rất kém $\mathbf{O(n^2)}$	$\mathbf{O(n \log n)}$	Pivot đầu/cuối luôn chọn phần tử nhỏ nhất hoặc lớn nhất → chia mảng mất cân bằng.
Mảng có nhiều phần tử trùng	Gần $\mathbf{O(n^2)}$	$\mathbf{O(n \log n)}$ (có thể cải tiến)	Pivot đầu/cuối dễ chọn phần tử trùng, gây nhiều bước so sánh dư thừa.

Bảng 3: So sánh hiệu suất Quick Sort theo cách chọn Pivot

4 Bài tập

4.1
Cho một mảng các chuỗi, sắp xếp chúng theo thứ tự từ điển nhưng không phân biệt hoa thường.

```
#include <bits/stdc++.h>
using namespace std;

bool compareIgnoreCase(const string&a, const string&b){
    string A = a, B = b;
    transform(A.begin(), A.end(), A.begin(), ::tolower);
    transform(B.begin(), B.end(), B.begin(), ::tolower);
    return A < B;
}
int partitionquicksort(vector<string>& a, int n, int m){
    string pivot = a[m];
    int i = n - 1;
    for(int j = n; j < m ; j++){
        if(compareIgnoreCase(a[j], pivot)){
            i++;
            swap(a[i],a[j]);
        }
    }
    swap(a[i+1],a[m]);
    return i+1;
}
void quicksort(vector<string>& a, int n, int m){
    if(n < m){
        int p = partitionquicksort(a, n, m);
        quicksort(a, n, p - 1);
        quicksort(a, p + 1, m);
    }
}
void merge(vector<string>& a, int left, int mid, int right){
    int n1 = mid - left + 1;
```

```
    int n2 = right - mid;
    vector<string> L(n1), R(n2);
    for(int i = 0; i < n1; i++){
        L[i] = a[left + i];
    }
    for(int j = 0; j < n2; j++){
        R[j] = a[mid + 1 + j];
    }
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2){
        if(compareIgnoreCase(L[i], R[j])){
            a[k++] = L[i++];
        }
        else{
            a[k++] = R[j++];
        }
    }
    while(i < n1) a[k++] = L[i++];
    while(j < n2) a[k++] = R[j++];
}
void mergesort(vector<string>& 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);
    }
}
void Bubblesort(vector<string>& a, int n){
    int temp;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n-i-1; j++){
            if(compareIgnoreCase(a[j+1], a[j])){
                swap(a[j], a[j+1]);
            }
        }
    }
}
void selectionsort(vector<string>& a, int n){
    for(int i = 0 ;i < n-1; i++){
        int min = i;
        for(int j = i+1; j < n; j++){
            if(compareIgnoreCase(a[j], a[min])){
                min = j;
            }
        }
        swap(a[i], a[min]);
    }
}
void insectionsort(vector<string>& a, int n){
    for(int i = 0;i < n-1; i++){
```

```
        string key = a[i+1];
        int j = i;
        while(j >= 0 && compareIgnoreCase(key, a[j])){
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = key;
    }
}
int main(){
    vector<string> a = {"Banana", "date", "cherry", "apple"};
    // quicksort(a, 0, a.size() - 1);
    // mergesort(a, 0 , a.size() - 1);
    // Bubblesort(a, a.size());
    // selectionsort(a, a.size());
    insectionsort(a, a.size());
    for(int k = 0; k < a.size(); k++){
        cout << a[k] << " ";
    }
}
```

4.2

Cho hai mảng đã sắp xếp A[n] và B[m], trộn chúng thành một mảng sắp xếp C[n + m] trong thời gian O(n + m).

```
#include <bits/stdc++.h>
using namespace std;

vector<int> mergearray(const vector<int>& a, const vector<int>& b)
   {
    vector<int> C(a.size() + b.size());
    int i = 0, j = 0, k = 0;
    while(i < a.size() && j < b.size()){
        if(a[i] < b[j]){
            C[k++] = a[i++];
        }
        else{
            C[k++] = b[j++];
        }
    }
    while(i < a.size()) C[k++] = a[i++];
    while(j < b.size()) C[k++] = b[j++];
    return C;
}
void bubblesortarray(vector<int>& a, int n){
    int temp;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n-i-1; j++){
            if(a[j] > a[j+1]){
                swap(a[j], a[j+1]);
            }
        }
```

```
    }
}
void selectionsortarray(vector<int>& a, int n){
    for(int i = 0; i < n-1; i++){
        int min = i;
        for (int j = i+1; j < n; j++){
            if(a[j] < a[min]){
                min = j;
            }
        }
        swap(a[i], a[min]);
    }
}
void insertionsortarray(vector<int>& a, int n){
    for(int i = 0; i < n-1; i++){
        int k = a[i+1];
        int j = i;
        while(i >= 0 && a[j] > k){
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = k;
    }
}
int partition_low(vector<int>& a, int n, int m){
    int pivot = a[n];
    int i = n+1;
    int j = m;
    while(true){
        while(i <= j && a[i] < pivot){
            i++;
        }
        while(i <= j && a[j] > pivot){
            j--;
        }
        if(i > j) break;
        swap(a[i], a[j]);
    }
    swap(a[n], a[j]);
    return j;
}
void quicksortarray(vector<int>& a, int n, int m){
    if(n < m){
        int p = partition_low(a,n,m);
        quicksortarray(a, n, p);
        quicksortarray(a, p+1, m);
    }
}
int main(){
    vector<int> a = {1,4,6,9,10};
    vector<int> b = {2,3,8,14,15};
```

```
    vector<int> D = a;
    D.insert(D.end(), b.begin(), b.end());

    quicksortarray(D, 0, D.size()-1);
    cout << "Quick Sort array" << endl;
    for(int k = 0; k < D.size(); k++){
        cout << D[k] << " ";
    }
    cout << " " << endl;

    bubblesortarray(D, D.size()-1);
    cout <<"Bubble Sort array : " << endl ;
        for(int k = 0; k < D.size(); k++){
        cout << D[k] << " ";
    }
    cout << " " << endl;

    selectionsortarray(D, D.size()-1);
    cout <<"Selection Sort array : " << endl ;
        for(int k = 0; k < D.size(); k++){
        cout << D[k] << " ";
    }
    cout << " " << endl;

    insertionsortarray(D, D.size()-1);
    cout <<"Insertion Sort array : " << endl ;
    for(int k = 0; k < D.size(); k++){
        cout << D[k] << " ";
    }
    cout << " " << endl;

    cout << "Merge Sort array : "  << endl;
    vector<int> C = mergearray(a,b);
    for(int m = 0; m < a.size() + b.size(); m++){
        cout << C[m] << " ";
    }
}
```

4.3

Cho một mảng số nguyên dương intervals với intervals[i] = [starti , endi] đại diện cho các khoảng thời gian. Hãy hợp nhất tất cả các khoảng thời gian chồng lấn và trả về mảng chứa các khoảng thời gian đã hợp nhất được sắp xếp theo thứ tự tăng dần của start time.

```
#include<bits/stdc++.h>
using namespace std;

void bubblesort(vector<vector<int>>& a){
    for(int i = 0; i < a.size() - 1; i++){
        for(int j = 0; j < a.size() - 1 - i; j++){
            if(a[j][0] > a[j+1][0]){
                swap(a[j], a[j+1]);
            }
```

```
        }
    }
}
vector<vector<int>> interval1(vector<vector<int>>& a){
    bubblesort(a);

    vector<vector<int>> bubble;
    for(auto a1 : a){
        if(bubble.empty() || bubble.back()[1] < a1[0]){
            bubble.push_back(a1);
        }
        else{
            bubble.back()[1] = max(bubble.back()[1], a1[1]);
        }
    }
    return bubble;
}

void selectionsort(vector<vector<int>>& b){
    for(int i = 0; i < b.size()-1; i++){
        int min = i;
        for(int j = i +1; j < b.size(); j++){
            if(b[j] < b[min]){
                min = j;
            }
        }
        swap(b[i], b[min]);
    }
}
vector<vector<int>> interval2(vector<vector<int>>& b){
    selectionsort(b);

    vector<vector<int>> selection;
    for(auto b1 : b){
        if(selection.empty() || selection.back()[1] < b1[0]){
            selection.push_back(b1);
        }else{
            selection.back()[1] = max(selection.back()[1], b1[1]);
        }
    }
    return selection;
}

void insertionsort(vector<vector<int>>& c){
    for(int i = 0; i < c.size()-1; i++){
        vector<int> c1 = c[i+1];
        int j = i;
        while(j >= 0 && c[j+1] < c[j]){
            c[j+1] = c[j];
            j--;
        }
```

```
            c[i+1] = c1;
        }
    }
    vector<vector<int>> interval3(vector<vector<int>>& c){
        insertionsort(c);
        vector<vector<int>> insertion;
        for(auto c2 :c){
            if(insertion.empty() || insertion.back()[1] < c2[0]){
                insertion.push_back(c2);
            }else{
                insertion.back()[1] = max(insertion.back()[1], c2[1]);
            }
        }
        return insertion;
    }

    void merge(vector<vector<int>>& d, int left, int mid, int right){
        int n1 = mid- left + 1;
        int n2 = right - mid;
        vector<vector<int>> L(n1), R(n2);
        for(int i=0; i < n1; i++){
            L[i] = d[left+i];
        }
        for(int j = 0; j < n2; j++){
            R[j] = d[mid+1+j];
        }
        int i = 0, j = 0, k = left;
        while(i < n1 && j < n2){
            if(L[i][0] < R[j][0]){
                d[k++] = L[i++];
            }else{
                d[k++] = R[j++];
            }
        }
        while(i < n1) d[k++] = L[i++];
        while(j < n2) d[k++] = R[j++];
    }
    void mergesort(vector<vector<int>>& d, int left, int right){
        if(left < right) {
            int mid = left + (right - left)/2;
            mergesort(d, left, mid);
            mergesort(d, mid+1, right);
            merge(d,left,mid,right);
        }
    }
    vector<vector<int>> interval4(vector<vector<int>>& d){
        mergesort(d,0,d.size()-1);

        vector<vector<int>> merge1;
        for(auto d1 : d){
            if(merge1.empty() || merge1.back()[1] < d1[0]){
```

```
            merge1.push_back(d1);
        }else{
            merge1.back()[1] = max(merge1.back()[1], d1[1]);
        }
    }
    return merge1;
}

int partition(vector<vector<int>>& e, int n, int m){
    vector<int> pivot = e[m];
    int i = n-1;
    for(int j = n; j < m; j++){
        if(e[j][0] < pivot[0]){
            i++;
            swap(e[i], e[j]);
        }
    }
    swap(e[i+1], e[m]);
    return i+1;
}
void quicksort(vector<vector<int>>& e, int n, int m){
    if(n < m){
        int p = partition(e,n,m);
        quicksort(e,n,p-1);
        quicksort(e,p+1,m);
    }
}
vector<vector<int>> interval5(vector<vector<int>>& e){
    quicksort(e,0,e.size()-1);

    vector<vector<int>> quick;
    for(auto e1 : e){
        if(quick.empty() || quick.back()[1] < e1[0]){
            quick.push_back(e1);
        }else{
            quick.back()[1] = max(quick.back()[1], e1[1]);
        }
    }
    return quick;
}
int main(){
    cout << "Bubble Sort " << endl;
    vector<vector<int>> a = {{1,3},{2,6},{8,10},{15,18}};
    for(auto a2 : interval1(a)){
        cout << "[" << a2[0] << "," << a2[1] <<"]" ;
    }
    cout << endl;

    cout << "Selection Sort" << endl;
    vector<vector<int>> b = {{1,6},{3,10},{9,20},{36,71}};
    for(auto b2 : interval2(b)){
```

```
        cout << "[" << b2[0] << "," << b2[1] << "]";
    }
    cout << endl;

    cout << "Insertion Sort" << endl;
    vector<vector<int>> c = {{1,5},{3,20},{12,
       34},{45,62},{65,70}};
    for(auto c3 : interval3(c)){
        cout << "[" << c3[0] << "," << c3[1] << "]";
    }
    cout << endl;

    cout << "Merge sort "<< endl;
    vector<vector<int>> d = {{1,5},{4,10},{13,18},{32,60}};
    for(auto d2 : interval4(d)){
        cout << "[" << d2[0] << "," << d2[1] << "]";
    }
    cout << endl;

    cout << "Quick Sort" << endl;
    vector<vector<int>> e = {{1,6},{4,25},{45,97},{222,855}};
    for(auto e2 : interval5(e)){
        cout << "[" << e2[0] << "," << e2[1] << "]";
    }
    cout << endl;
}
```

4.4
Hybrid Sort: Thiết kế một thuật toán sắp xếp kết hợp nhiều thuật toán:

- Sử dụng Quick Sort làm thuật toán chính.
- Chuyển sang Insertion Sort khi mảng con nhỏ hơn 10 phần tử.
- Sử dụng Median-of-three (chọn trung vị trong ba phần tử đầu, giữa, cuối làm pivot).
- So sánh với Quick Sort thuần.

```
#include <bits/stdc++.h>
using namespace std;

const int INSERTION_THRESHOLD = 10;

void insertionSort(vector<int>& arr, int low, int high) {
    for (int i = low + 1; i <= high; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= low && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
```

```
    }
}

int medianOfThree(vector<int>& arr, int low, int high) {
    int mid = (low + high) / 2;
    int a = arr[low], b = arr[mid], c = arr[high];
    if ((a > b) != (a > c)) return low;
    else if ((b > a) != (b > c)) return mid;
    else return high;
}

int partition(vector<int>& arr, int low, int high) {
    int medianIndex = medianOfThree(arr, low, high);
    swap(arr[medianIndex], arr[high]);
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void hybridSort(vector<int>& arr, int low, int high) {
    while (low < high) {
        if (high - low + 1 < INSERTION_THRESHOLD) {
            insertionSort(arr, low, high);
            break;
        } else {
            int pi = partition(arr, low, high);
            if (pi - low < high - pi) {
                hybridSort(arr, low, pi - 1);
                low = pi + 1;
            } else {
                hybridSort(arr, pi + 1, high);
                high = pi - 1;
            }
        }
    }
}
int partitionSimple(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            swap(arr[i], arr[j]);
```

```
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionSimple(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    const int N = 100000;
    vector<int> arr1(N), arr2(N);
    srand(time(nullptr));
    for (int i = 0; i < N; ++i) {
        int x = rand() % N;
        arr1[i] = x;
        arr2[i] = x;
    }

    auto start = chrono::high_resolution_clock::now();
    hybridSort(arr1, 0, N - 1);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> hybridTime = end - start;

    start = chrono::high_resolution_clock::now();
    quickSort(arr2, 0, N - 1);
    end = chrono::high_resolution_clock::now();
    chrono::duration<double> quickTime = end - start;

    cout << fixed << setprecision(5);
    cout << "Hybrid Sort time: " << hybridTime.count() << " s\n";
    cout << "Quick Sort time:  " << quickTime.count() << " s\n";

    cout << "Giong nhau: " << boolalpha << (arr1 == arr2) << "\n";
    return 0;
}
```