TRƯỜNG ĐẠI HỌC SÀI GÒN

KHOA CÔNG NGHỆ THÔNG TIN

BÀI TẬP SẮP XẾP

Nhóm thực hiện : Trần Phương Thảo - 3124411282
Tăng Huệ Ý - 3124411357
Nguyễn Thị Yến Nhi - 3124411202
Phạm Thúy Ngân - 3124411180

Thành phố Hồ Chí Minh, tháng 3 năm 2025

MỤC LỤC

BÀI TẬP SẮP XẾP

PHÂN CÔNG NHIỆM VỤ:.....3
CÂU HỎI 3
Câu 1. Trình bày tư tưởng của các thuật toán sắp xếp?.....3
Câu 2. Trong các thuật toán sắp xếp, bạn thích nhất thuật toán nào? Thuật toán nào bạn không thích nhất? Tại sao?.....7
Câu 3. Trình bày và cài đặt tất cả các thuật toán sắp xếp nội, ngoại theo thứ tự giảm dần. Cho nhận xét về các thuật toán này.....7
Câu 4. Hãy trình bày những ưu điểm và nhược điểm của mỗi thuật toán sắp xếp? Theo bạn, cách khắc phục những nhược điểm này là gì? 12
BÀI TẬP CƠ SỞ..... 13
Bài tập 1 13
Bài tập 2 17
Bài tập 3 20
Bài tập 4 25
BÀI TẬP ỨNG DỤNG..... 36
Bài tập 2 39
Bài tập 3 41
Bài tập 4 45
Bài tập 5 46
Bài tập 6 49
Bài tập 7 51
Bài tập 8 52
Bài tập 9. 53

PHÂN CÔNG NHIỆM VỤ:

Tên	Câu hỏi	Bài tập cơ sở	Bài tập ứng dụng	Bài tập tổng hợp	Báo cáo
Trần Phương Thảo	1	3	1-5		x
Nguyễn Thị Yến Nhi	2	4	2-8	a-b	
Phạm Thúy Ngân	3	1	4-7	e-f	
Tăng Huệ Ý	4	2	3-6	c-d	

CÂU HỎI

Câu 1. Trình bày tư tưởng của các thuật toán sắp xếp?

Các thuật toán sắp xếp bao gồm:

+ Straight Insertion Sort
+ Straight Selection Sort
+ Interchange Sort
+ Bubble Sort
+ Heap Sort
+ Shell Sort
+ Quick Sort
+ Merge Sort
+ Index Sort
+ Radix Sort

Được chia vào 3 nhóm sắp thứ tự:

+ Xen vào: Insertion Sort
+ Chọn: Selection Sort, Heap Sort
+ Đổi chỗ: Bubble Sort, Merge Sort, Quick Sort

1. Thuật toán sắp xếp nổi bọt (Bubble Sort)

Bắt đầu từ cuối mảng và di chuyển dần về đầu mảng, trong quá trình này, ta so sánh từng cặp phần tử đứng cạnh nhau. Nếu phần tử ở dưới (phía sau) lớn hơn phần tử đứng ngay trên (phía trước) nó, hai phần tử sẽ được đổi chỗ cho nhau. Quá trình này lặp đi lặp lại, giúp phần tử nhỏ hơn dần "trồi" lên phía trên, giống như bong bóng nổi lên mặt nước.

Sau mỗi lần duyệt qua mảng, phần tử nhỏ nhất trong đoạn chưa được sắp xếp sẽ tìm được vị trí đúng của nó. Do đó, sau N-1 lần lặp, tất cả các phần tử trong mảng sẽ được sắp xếp theo thứ tự tăng dần.

2. Thuật toán sắp xếp nổi bọt có cờ (Bubble Sort with Flag)

Thuật toán sắp xếp nổi bọt có cờ (Bubble Sort with Flag)

Thuật toán **Bubble Sort có cờ** là một phiên bản cải tiến của **Bubble Sort**, giúp giảm bớt số lần duyệt không cần thiết khi dãy số đã gần như được sắp xếp.

Tư tưởng:

Tương tự Bubble Sort thông thường, thuật toán này cũng duyệt qua dãy số nhiều lần và hoán đổi các cặp phần tử nếu chúng không đúng thứ tự. Tuy nhiên, thay vì luôn lặp lại đủ n−1n - 1 lần, thuật toán sử dụng một **biến cờ (flag)** để kiểm tra xem trong lần duyệt hiện tại có xảy ra hoán đổi nào không:

- Nếu có ít nhất một lần hoán đổi, điều đó có nghĩa là dãy số vẫn chưa hoàn chỉnh, và ta tiếp tục lặp lại quá trình.
- Nếu không có bất kỳ hoán đổi nào trong một lượt duyệt, dãy số đã được sắp xếp hoàn chỉnh và thuật toán có thể dừng sớm.

Nhờ vào biến cờ, thuật toán có thể giảm đáng kể số lần duyệt trong trường hợp dãy số đã gần đúng thứ tự, giúp tăng hiệu suất so với Bubble Sort thông thường.

3. Thuật toán Quick Sort

Thuật toán sắp xếp nhanh (Quick Sort) hoạt động bằng cách chia nhỏ dãy số thành các phần dễ xử lý hơn, tương tự như cách ta chia bài thành từng nhóm để sắp xếp nhanh hơn.

Trước tiên, ta chọn một phần tử trong dãy làm **mốc (pivot)** (thường là phần tử giữa hoặc bất kỳ). Sau đó, ta phân chia dãy số thành ba phần:

- **Nhóm 1:** Gồm các phần tử nhỏ hơn phần tử mốc.
- **Nhóm 2:** Gồm các phần tử có giá trị bằng phần tử mốc.
- **Nhóm 3:** Gồm các phần tử lớn hơn phần tử mốc.

Tiếp theo, ta tiếp tục lặp lại quá trình này cho **Nhóm 1** và **Nhóm 3**, chia nhỏ dãy con cho đến khi mỗi nhóm chỉ còn một phần tử hoặc không cần sắp xếp nữa.

Để thực hiện việc phân chia, ta sẽ kiểm tra các phần tử từ hai đầu dãy. Nếu có một số ở nhóm lớn hơn lại nằm bên trái phần tử mốc và một số nhỏ hơn nằm bên phải, ta hoán đổi vị trí chúng để đảm bảo đúng thứ tự.

Cứ tiếp tục như vậy, sau một số lần lặp, dãy số sẽ tự động được sắp xếp mà không cần phải duyệt từng cặp số nhiều lần như một số thuật toán khác. Nhờ cách làm này, Quick Sort thường có tốc độ sắp xếp rất nhanh, đặc biệt khi làm việc với các tập dữ liệu lớn.

4. Thuật toán sắp xếp chọn trực tiếp (Straight Selection Sort)

Thuật toán sắp xếp chọn trực tiếp hoạt động dựa trên việc tìm phần tử nhỏ nhất trong danh sách và đặt nó vào đúng vị trí. Ban đầu, ta có một dãy số chưa được sắp xếp. Ta tìm phần tử nhỏ nhất trong toàn bộ dãy và hoán đổi nó với phần tử đầu tiên. Sau bước này, phần tử nhỏ nhất đã ở đúng vị trí.

Tiếp theo, ta không xét phần tử đầu tiên nữa mà tìm phần tử nhỏ nhất trong các phần tử còn lại, rồi hoán đổi nó với phần tử thứ hai. Quá trình này tiếp tục lặp lại: mỗi lần ta tìm phần tử nhỏ nhất trong phần còn lại của dãy và đưa nó về đúng vị trí.

Sau **N-1 lần chọn lựa**, toàn bộ dãy số sẽ được sắp xếp theo thứ tự tăng dần. Thuật toán này hoạt động dựa trên nguyên tắc tìm kiếm tuần tự, giúp đảm bảo rằng mỗi phần tử nhỏ nhất được đặt vào vị trí thích hợp trước khi tiếp tục với các phần tử lớn hơn.

5. Thuật toán sắp xếp chèn trực tiếp (Straight Insertion Sort)

Ban đầu, ta giả sử một phần của dãy đã được sắp xếp. Khi thêm một phần tử mới vào dãy này, ta cần tìm đúng vị trí để đặt nó sao cho thứ tự vẫn được giữ nguyên. Để làm điều này, ta sẽ duyệt qua các phần tử đã sắp xếp, tìm vị trí thích hợp, rồi dịch chuyển các phần tử lớn hơn sang bên phải để tạo chỗ trống cho phần tử mới.

Quá trình này được lặp lại cho từng phần tử trong dãy, cho đến khi toàn bộ dãy được sắp xếp. Vì vậy, thuật toán này hoạt động bằng cách "chèn" từng phần tử vào đúng vị trí của nó, giống như cách bạn chèn một quân bài vào đúng chỗ trong một bộ bài đang xếp.

6. Thuật toán sắp xếp trộn (Merge Sort)

Thuật toán này tận dụng những đoạn đã được sắp xếp sẵn trong dãy (gọi là đường chạy tự nhiên) thay vì chia nhỏ từng phần tử một cách cố định.

Cách thực hiện như sau: Đầu tiên, ta tìm các đoạn có thứ tự tăng dần trong dãy. Sau đó, ta trộn từng cặp đoạn liên tiếp lại với nhau để tạo thành những đoạn lớn hơn. Các đoạn mới này sẽ được tạm thời lưu vào hai dãy phụ.

Tiếp theo, ta lại trộn các đoạn trong hai dãy phụ để tạo thành những đoạn lớn hơn nữa và đưa kết quả trở lại dãy ban đầu. Quá trình này được lặp đi lặp lại cho đến khi toàn bộ dãy chỉ còn một đoạn duy nhất đã được sắp xếp hoàn chỉnh.

Cách làm này giúp thuật toán tận dụng tốt những phần đã có thứ tự sẵn, làm giảm số lần trộn và tăng tốc độ sắp xếp.

7. Thuật toán sắp xếp theo chỉ mục (Index Sort)

Khi làm việc với các tập tin dữ liệu lớn, việc đọc và ghi trực tiếp trên tập tin dữ liệu có thể làm mất nhiều thời gian và ảnh hưởng đến hiệu suất hệ thống. Hơn nữa, nếu dữ liệu thay đổi liên tục, thao tác sắp xếp trên tập tin gốc sẽ rất tốn kém và khó thực hiện. Để giải quyết vấn đề này, thay vì sắp xếp trực tiếp trên tập tin dữ liệu, chúng ta sử dụng một tập tin chỉ mục để quản lý thứ tự xuất hiện của các phần tử dữ liệu.

Cụ thể, từ tập tin dữ liệu ban đầu, chúng ta tạo ra một tập tin chỉ mục chứa danh sách các khóa nhận diện của các phần tử trong tập tin dữ liệu gốc. Danh sách này sẽ được sắp xếp theo thứ tự tăng dần của khóa nhận diện. Khi cần truy xuất dữ liệu theo thứ tự, thay vì sắp xếp trực tiếp trên tập tin gốc, chúng ta chỉ cần đọc các khóa từ tập tin chỉ mục và sử dụng chúng để truy xuất dữ liệu theo đúng thứ tự mong muốn. Như vậy,

tập tin dữ liệu gốc vẫn giữ nguyên thứ tự vật lý ban đầu trên đĩa, nhưng thứ tự xuất hiện của các phần tử khi hiển thị trên màn hình hoặc máy in sẽ được điều khiển bởi tập tin chỉ mục.

Phương pháp này mang lại nhiều lợi ích, đặc biệt là giúp giảm thời gian đọc/ghi dữ liệu, đồng thời không làm thay đổi nội dung và vị trí vật lý của dữ liệu gốc. Ngoài ra, nếu muốn sắp xếp dữ liệu theo nhiều tiêu chí khác nhau, chúng ta chỉ cần tạo các tập tin chỉ mục tương ứng mà không cần thay đổi tập tin dữ liệu gốc. Điều này giúp tối ưu hóa quá trình tìm kiếm và truy xuất dữ liệu một cách hiệu quả.

8. Thuật toán Heap Sort

Thuật toán sắp xếp Heap hoạt động dựa trên cấu trúc dữ liệu **Heap**, một dạng cây nhị phân đặc biệt.

Đầu tiên, thuật toán sẽ biến đổi dãy số ban đầu thành một **Max-Heap**, tức là một cây mà mỗi phần tử cha luôn lớn hơn hoặc bằng hai phần tử con của nó. Khi đó, phần tử lớn nhất sẽ nằm ở gốc của cây.

Sau đó, thuật toán tiến hành sắp xếp dãy số bằng cách hoán đổi phần tử lớn nhất (ở gốc) với phần tử cuối cùng trong dãy, rồi giảm kích thước của Heap để loại bỏ phần tử đã được sắp xếp.

Tiếp theo, ta điều chỉnh lại Heap để tiếp tục đưa phần tử lớn nhất còn lại lên gốc.

Quá trình này lặp lại cho đến khi toàn bộ dãy số được sắp xếp hoàn chỉnh.

9. Thuật toán sắp xếp đổi chỗ trực tiếp (Interchange Sort)

Thuật toán này sắp xếp dãy số bằng cách duyệt qua từng phần tử và so sánh nó với tất cả các phần tử đứng sau. Nếu một phần tử lớn hơn phần tử phía sau nó (trong trường hợp sắp xếp tăng dần), ta sẽ đổi chỗ hai phần tử đó. Cứ như vậy, từng phần tử trong dãy sẽ dần được đặt đúng vị trí.

Cứ lặp lại như vậy đến khi cả hàng được sắp xếp theo đúng thứ tự mong muốn.

10. Sắp xếp cơ số (Radix Sort)

Thuật toán Radix Sort sắp xếp dãy số bằng cách nhóm các số theo từng chữ số, thay vì so sánh từng cặp số như các thuật toán khác.

Đầu tiên, ta sắp xếp các số dựa trên **chữ số cuối cùng** (hàng đơn vị). Sau đó, ta tiếp tục sắp xếp theo **chữ số kế tiếp** (hàng chục, hàng trăm...), lặp lại quá trình này cho đến khi xét hết tất cả các chữ số.

Bạn có thể tưởng tượng như khi sắp xếp giấy tờ theo ngày tháng: trước tiên, ta nhóm chúng theo ngày, sau đó sắp xếp lại theo tháng, rồi đến năm. Sau mỗi bước, dãy số sẽ ngày càng có thứ tự hơn và khi hoàn tất, ta có một dãy số đã được sắp xếp đúng thứ tự.

Nhờ cách làm này, Radix Sort có thể sắp xếp nhanh chóng mà không cần so sánh từng cặp số, đặc biệt hiệu quả khi làm việc với các tập dữ liệu lớn.

Câu 2. Trong các thuật toán sắp xếp, bạn thích nhất thuật toán nào? Thuật toán nào bạn không thích nhất? Tại sao?

Thuật toán thích nhất là Quicksort: Vì Quicksort So sánh từng cặp phần tử liền kề và hoán đổi nếu chúng không theo đúng thứ tự. Lặp lại quá trình cho đến khi danh sách được sắp xếp. Quicksort có độ phức tạp trung bình O(n log n), chạy rất nhanh trên hầu hết các bộ dữ liệu thực tế, và sử dụng phân hoạch tối ưu hóa việc sắp xếp. Dễ cài đặt dễ hiểu, Hiệu suất tốt ngay cả với dữ liệu lớn

Thuật toán không thích nhất là Bubble Sort: Vì Bubble Sort so sánh từng cặp phần tử liền kề và hoán đổi nếu chúng không theo đúng thức tự. Lặp lại quá trình cho đến khi danh sách được sắp xếp và hiệu suất kém trên các bộ dữ liệu lớn và ít được sử dụng trong thực tế. Nó chỉ hữu ích cho mục đích giảng dạy hoặc trong các tình huống đặc biệt khi dữ liệu gần như đã được sắp xếp.

Câu 3. Trình bày và cài đặt tất cả các thuật toán sắp xếp nội, ngoại theo thứ tự giảm dần. Cho nhận xét về các thuật toán này.

I. Thuật toán sắp xếp nội

Bubble Sort

Ý tưởng: So sánh từng cặp phần tử liền kề và đổi chỗ nếu không đúng thứ tự. Lặp lại cho đến khi mảng được sắp xếp hoàn toàn.

Code:

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

Nhận xét: Đơn giản nhưng chậm, độ phức tạp $O(n^2)$. Không phù hợp cho dữ liệu lớn.

Selection Sort

Ý tưởng: Tìm phần tử lớn nhất và đưa về đầu mảng. Lặp lại cho phần còn lại của mảng.

Code:

```
void selectionSort(int arr[], int n) {
```

```
    for (int i = 0; i < n; i++) {
        int maxIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] > arr[maxIndex]) {
                maxIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[maxIndex];
        arr[maxIndex] = temp;
    }
}
```

Nhận xét: Nhận xét: Đơn giản, ít tốn bộ nhớ nhưng hiệu suất kém với dữ liệu lớn.

Insertion Sort

Ý tưởng: Chèn từng phần tử vào đúng vị trí trong danh sách con đã sắp xếp.

Code:

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

Nhận xét: Hiệu quả với dữ liệu gần như đã sắp xếp, độ phức tạp $O(n^2)$.

Quick Sort

Ý tưởng: Chọn một phần tử làm chốt (pivot), chia mảng thành 2 phần: lớn hơn và nhỏ hơn chốt, sau đó sắp xếp từng phần.

Code:

```
void quickSort(int arr[], int left, int right) {
    int i = left, j = right;
    int pivot = arr[(left + right) / 2];
    while (i <= j) {
        while (arr[i] > pivot) i++;
        while (arr[j] < pivot) j--;
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
    if (left < j) quickSort(arr, left, j);
    if (i < right) quickSort(arr, i, right);
```

```
}
```

Nhận xét: Hiệu suất cao, trung bình O(n log n), nhưng không ổn định.

Merge Sort

Ý tưởng: Chia mảng thành các mảng con, sắp xếp từng mảng con và trộn lại.

Code:

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Nhận xét: Ổn định, độ phức tạp O(n log n), phù hợp cho dữ liệu lớn.

II. Thuật toán sắp xếp ngoại

1. External Merge Sort

Ý tưởng: Áp dụng cho dữ liệu lớn, không thể lưu trữ toàn bộ trong bộ nhớ chính, cần xử lý từng phần và gộp lại.

Code:

```
void externalMergeSort(int arr[], int n) {
    sort(arr, arr + n, greater<int>());
}
```

Nhận xét: Hiệu quả với dữ liệu cực lớn, nhưng tốn thời gian đọc/ghi dữ liệu từ ổ cứng.

Sắp xếp là quá trình sắp xếp các phần tử trong một danh sách theo một thứ tự nhất định (tăng dần hoặc giảm dần).
Trong chương này, ta sẽ trình bày các thuật toán sắp xếp nội và sắp xếp ngoại, cài đặt bằng C++ (không dùng vector) và nhận xét về từng thuật toán.

3.2. Các giải thuật sắp xếp nội

3.2.1. Sắp xếp bằng phương pháp đổi chỗ (Bubble Sort)

Ý tưởng: So sánh từng cặp phần tử liền kề và đổi chỗ nếu không đúng thứ tự.

Code:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] < arr[j+1]) { // Sắp xếp giảm dần
```

```
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Nhận xét: Thuật toán đơn giản nhưng chậm, độ phức tạp $O(n^2)$.

3.2.2. Sắp xếp bằng phương pháp chọn (Selection Sort)

Ý tưởng: Tìm phần tử lớn nhất và đưa về vị trí đầu tiên của mảng chưa sắp xếp.

Code:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        int maxIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] > arr[maxIndex]) {
                maxIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[maxIndex];
        arr[maxIndex] = temp;
    }
}
```

Nhận xét: Ít tốn bộ nhớ, nhưng hiệu suất kém khi dữ liệu lớn, $O(n^2)$.

3.2.3. Sắp xếp bằng phương pháp chèn (Insertion Sort)

Ý tưởng: Chèn từng phần tử vào đúng vị trí trong danh sách con đã sắp xếp.

Code:

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

Nhận xét: Hiệu quả với dữ liệu gần như đã sắp xếp, độ phức tạp $O(n^2)$.

3.2.4. Sắp xếp bằng phương pháp trộn (Merge Sort)

Ý tưởng: Chia mảng thành các mảng con, sắp xếp từng mảng con rồi gộp lại.

Code:

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

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] >= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Nhận xét: Độ phức tạp O(n log n), ổn định và hiệu quả với dữ liệu lớn.

3.3. Các giải thuật sắp xếp ngoại

3.3.1. Sắp xếp bằng phương pháp trộn (External Merge Sort)

Ý tưởng: Chia dữ liệu thành nhiều phần nhỏ, sắp xếp từng phần rồi trộn lại.

Code:

```
void externalMergeSort(int arr[], int n) {
    sort(arr, arr + n, greater<int>());
}
```

Nhận xét: Áp dụng cho dữ liệu lớn không thể lưu trong bộ nhớ chính.

3.3.2. Sắp xếp theo chỉ mục (Index Sort)

Ý tưởng: Tạo một mảng chỉ mục cho các phần tử trong mảng chính, sắp xếp chỉ mục này thay vì sắp xếp trực tiếp mảng.

Code:

```
void indexSort(int arr[], int n, int index[]) {
    for (int i = 0; i < n; i++) index[i] = i;

    for (int i = 0; i < n-1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[index[i]] < arr[index[j]]) {
                int temp = index[i];
                index[i] = index[j];
                index[j] = temp;
            }
        }
    }
}
```

Nhận xét: Phù hợp khi cần giữ nguyên dữ liệu gốc.

Thuật toán	Độ phức tạp	Hiệu suất
Bubble Sort	$O(n^2)$	Chậm
Selection Sort	$O(n^2)$	Tốt cho dữ liệu nhỏ
Insertion Sort	$O(n^2)$	Tốt cho dữ liệu gần sắp xếp
Quick Sort	O(n log n)	Nhanh nhất nhưng không ổn định
Merge Sort	O(n log n)	Ổn định và tốt cho dữ liệu lớn
External Merge Sort	O(n log n)	Dùng cho dữ liệu cực lớn

Câu 4. Hãy trình bày những ưu điểm và nhược điểm của mỗi thuật toán sắp xếp? Theo bạn, cách khắc phục những nhược điểm này là gì?

Các thuật toán	Ưu điểm	Nhược điểm	Cách khắc phục
Insertion Sort	Đơn giản Phương pháp sắp thứ tự ổn định	Chạy chậm với độ phức tạp là (O(n^2))	Sử dụng thuật toán tối ưu hơn như

	Không cần bộ nhớ phụ		Binary Insertion Sort
Selection Sort	Đơn giản Không cần bộ nhớ phụ	Chạy chậm với độ phức tạp là (O(n^2)) Phương pháp sắp thứ tự không ổn định	Sử dụng thuật toán khác thay thế như Quick Sort, Merge Sort, Heap Sort
Heap Sort	Chạy nhanh với thời gian trung bình và tệ nhất là (O(nlog2(n)))	Phương pháp sắp thứ tự không ổn định	Sử dụng thuật toán khác thay thế như Quick Sort
Bubble Sort	Đơn giản Phương pháp sắp thứ tự ổn định	Chạy chậm với độ phức tạp là (O(n^2))	Sử dụng thuật toán khác thay thế như Insertion Sort, Quick Sort, Merge Sort
Quick Sort	Chạy nhanh với (O(nlog2(n)))	Chạy chậm với độ phức tạp là (O(n^2))	Lựa chọn pivot tốt hơn như median of three, random pivot
Merge Sort	Chạy nhanh với (O(nlog2(n)))	Cần bộ nhớ phụ	Giảm việc sử dụng bộ nhớ phụ
Radix Sort	Chạy nhanh	Cần bộ nhớ phụ Chỉ có thể áp dụng cho dữ liệu có số nguyên hoặc chuỗi ký tự	Sử dụng thuật toán khác thay thế như Quick Sort, Merge Sort
Interchange Sort	Đơn giản	Chạy chậm với độ phức tạp là (O(n^2))	Sử dụng thuật toán khác thay thế như Bubble Sort, Insertion Sort, Quick Sort

BÀI TẬP CƠ SỞ

Bài tập 1

1. Sắp xếp đổi chỗ trực tiếp (Interchange Sort)

Ý tưởng: Thuật toán Bubble Sort hoạt động bằng cách lặp đi lặp lại qua các phần tử trong mảng và hoán đổi các cặp phần tử liền kề nếu chúng không theo đúng thứ tự. Quá trình này lặp lại cho đến khi mảng được sắp xếp hoàn toàn.

Các bước mô phỏng:
Bước 1: (39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10)
Bước 2: (1, 8, 5, 39, 3, 6, 9, 12, 4, 7, 10)
Bước 3: (1, 3, 5, 39, 8, 6, 9, 12, 4, 7, 10)
Bước 4: (1, 3, 4, 39, 8, 6, 9, 12, 5, 7, 10)
Bước 5: (1, 3, 4, 5, 8, 6, 9, 12, 39, 7, 10)
Bước 6: (1, 3, 4, 5, 6, 8, 9, 12, 39, 7, 10)

Bước 7: (1, 3, 4, 5, 6, 7, 9, 12, 39, 8, 10)
Bước 8: (1, 3, 4, 5, 6, 7, 8, 12, 39, 9, 10)
Bước 9: (1, 3, 4, 5, 6, 7, 8, 9, 39, 12, 10)
Bước 10: (1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39)

Code C++:

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

Độ phức tạp: Giả sử mảng có n phần tử, ta sẽ thực hiện:

Ở lần lặp thứ nhất: Duyệt qua toàn bộ mảng, thực hiện n-1 phép so sánh.

Ở lần lặp thứ hai: Thực hiện n-2 phép so sánh.

…

Ở lần lặp cuối cùng: Thực hiện 1 phép so sánh.

Tổng số phép so sánh là:

(n-1) + (n-2) + ….+ 1 = (n(n-1))/2

Đây là biểu thức bậc hai theo n, nên độ phức tạp thời gian trong trường hợp xấu nhất là: O(n^2)

Kết luận: Trường hợp tốt nhất (mảng đã sắp xếp): O(n).

Trường hợp trung bình và xấu nhất: O(n^2).

2. Sắp xếp chọn trực tiếp (Selection Sort)

Ý tưởng: Thuật toán sắp xếp chọn trực tiếp (Selection Sort) hoạt động bằng cách tìm phần tử nhỏ nhất (hoặc lớn nhất) trong mảng và đưa nó về đúng vị trí. Quá trình này lặp lại cho phần còn lại của mảng cho đến khi mảng được sắp xếp hoàn toàn.

Các bước mô phỏng:
Bước 1: (1, 8, 5, 39, 3, 6, 9, 12, 4, 7, 10)
Bước 2: (1, 3, 5, 39, 8, 6, 9, 12, 4, 7, 10)
Bước 3: (1, 3, 4, 39, 8, 6, 9, 12, 5, 7, 10)
Bước 4: (1, 3, 4, 5, 8, 6, 9, 12, 39, 7, 10)
Bước 5: (1, 3, 4, 5, 6, 8, 9, 12, 39, 7, 10)
Bước 6: (1, 3, 4, 5, 6, 7, 9, 12, 39, 8, 10)
Bước 7: (1, 3, 4, 5, 6, 7, 8, 12, 39, 9, 10)

Bước 8: (1, 3, 4, 5, 6, 7, 8, 9, 39, 12, 10)
Bước 9: (1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39)

Code C++:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
```

Độ phức tạp:

Ở lần lặp đầu tiên: Tìm phần tử nhỏ nhất trong n phần tử, thực hiện n-1 phép so sánh.

Ở lần lặp thứ hai: Tìm phần tử nhỏ nhất trong n-1 phần tử, thực hiện n-2 phép so sánh.

…

Ở lần lặp cuối cùng: Chỉ cần thực hiện 1 phép so sánh.

Tổng số phép so sánh là:

(n-1) + (n-2) + …+ 1 = (n(n-1))/2

Độ phức tạp thời gian là: O(n^2)

Kết luận: Trường hợp tốt nhất, trung bình và xấu nhất đều là O(n^2).

3. Sắp xếp chèn trực tiếp (Insertion Sort)

Ý tưởng: Thuật toán sắp xếp chèn trực tiếp (Insertion Sort) sắp xếp từng phần tử bằng cách chèn nó vào vị trí thích hợp trong danh sách con đã được sắp xếp.

Các bước mô phỏng:

Bước 1: (39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10)

Bước 2: (8, 39, 5, 1, 3, 6, 9, 12, 4, 7, 10)

Bước 3: (5, 8, 39, 1, 3, 6, 9, 12, 4, 7, 10)

Bước 4: (1, 5, 8, 39, 3, 6, 9, 12, 4, 7, 10)

Bước 5: (1, 3, 5, 8, 39, 6, 9, 12, 4, 7, 10)

Bước 6: (1, 3, 5, 6, 8, 39, 9, 12, 4, 7, 10)

Bước 7: (1, 3, 5, 6, 8, 9, 39, 12, 4, 7, 10)

Bước 8: (1, 3, 5, 6, 8, 9, 12, 39, 4, 7, 10)

Bước 9: (1, 3, 4, 5, 6, 8, 9, 12, 39, 7, 10)

Bước 10: (1, 3, 4, 5, 6, 7, 8, 9, 12, 39, 10)

Bước 11: (1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39)

Code C++:

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

Độ phức tạp:

Vòng lặp ngoài chạy n-1 lần.

Vòng lặp trong tìm vị trí chèn phần tử, trong trường hợp xấu nhất sẽ chạy i lần với mỗi i từ 1 đến n-1.

Tổng số phép so sánh là:

$1 + 2 + 3 + \ldots + (n-1) = (n(n-1))/2$

Đây là một biểu thức bậc hai theo n, do đó độ phức tạp thời gian trong trường hợp xấu nhất là:O(n^2)

Tuy nhiên, trong trường hợp mảng gần như đã sắp xếp, số lần so sánh sẽ giảm xuống và độ phức tạp là O(n).

Kết luận:

Trường hợp xấu nhất: O(n^2).

Trường hợp tốt nhất: O(n).

4. Sắp xếp nổi bọt (Bubble Sort)

Ý tưởng: Thuật toán sắp xếp nổi bọt thực chất là một dạng cụ thể của sắp xếp đổi chỗ trực tiếp, nơi các phần tử lớn hơn "nổi" lên vị trí cuối mảng sau mỗi lần lặp.

Các bước mô phỏng:
Bước 1: (8, 5, 1, 3, 6, 9, 12, 4, 7, 10, 39)
Bước 2: (5, 1, 3, 6, 8, 9, 4, 7, 10, 12, 39)
Bước 3: (1, 3, 5, 6, 8, 4, 7, 9, 10, 12, 39)
Bước 4: (1, 3, 5, 6, 4, 7, 8, 9, 10, 12, 39)

Bước 5: (1, 3, 5, 4, 6, 7, 8, 9, 10, 12, 39)
Bước 6: (1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39)

Code C++:

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

Độ phức tạp:

Vòng lặp ngoài chạy n-1 lần.

Vòng lặp trong chạy n-1, n-2, \dots, 1 lần.

Tổng số phép so sánh là:

(n-1) + (n-2) + …+ 1 = (n(n-1))/2

Bỏ qua hệ số và bậc thấp hơn, ta có độ phức tạp là:

O(n^2)

Kết luận:Trường hợp tốt nhất: O(n) (khi mảng đã sắp xếp).

Trường hợp trung bình và xấu nhất: O(n^2).

Bài tập 2

Quick Sort:

Ý niệm: Chia mảng ra thành 2 phần dựa trên pivot sau đó gọi đệ quy lên 2 mảng để tiến hành sắp xếp mảng.

Ví dụ với mảng {8, 5, 1, 3, 6, 9, 12, 4, 7, 10}:

- Chọn 6 làm pivot: Mảng trước 6 gồm {8, 5, 1, 3} và sau 6 gồm {9, 12, 4, 7, 10}.
- Tiến hành so sánh, đệ quy và sắp xếp các phần tử trong mảng bắt đầu từ các phần tử ngoài cùng: 8 < 10 nhưng 8 > 6 nên sẽ được sắp xếp thành {6, 8, 10}.
- Chương trình sẽ được tiếp tục cho tới khi cho ra kết quả là {1, 3, 4, 5, 6, 7, 8, 9, 10, 12}.

Merge Sort:

Ý niệm: Chia mảng ra và rồi hợp nhất từng phần tử trong mảng sau đó gọi đệ quy để tiến hành sắp xếp mảng.

Ví dụ với mảng {8, 5, 1, 3, 6, 9, 12, 4, 7, 10}:

- Chia nhỏ mảng ra thành từng phần tử.
- Tiến hành so sánh, đệ quy, sắp xếp 2 phần tử với nhau: 8 > 5 nên sẽ được sắp xếp thành {5, 8}.
- Chương trình sẽ được tiếp tục cho tới khi cho ra kết quả là {1, 3, 4, 5, 6, 7, 8, 9, 10, 12}.

Heap Sort:

Ý niệm: Xây dựng cây heap và rồi chọn phần tử lớn nhất trong mảng làm phần tử gốc sau đó thực hiện "heapify" để tiến hành sắp xếp mảng.

Ví dụ với mảng {8, 5, 1, 3, 6, 9, 12, 4, 7, 10}:

- Chọn 12 làm phần tử gốc và hoán đổi với phần tử cuối cùng của mảng.
- Tiến hành so sánh và sắp xếp các phần tử còn lại trong mảng bằng "heapify".
- Chương trình sẽ được tiếp tục cho tới khi cho ra kết quả là {1, 3, 4, 5, 6, 7, 8, 9, 10, 12}.

Cài đặt thuật toán:

```
#include <iostream>
using namespace std;
//Chia mảng
int Partition(int a[], int low, int high) {
    int pivot = a[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (a[j] <= pivot) {
            i++;
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[high]);
    return (i + 1);
}
//Sử dụng thuật toán Quick Sort
void QuickSort(int a[], int low, int high) {
    if (low < high) {
        int pi = Partition(a, low, high);
        QuickSort(a, low, pi - 1);
        QuickSort(a, pi + 1, high);
    }
}
//Chia và hợp nhất từng mảng
void Merge(int a[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
```

```
        L[i] = a[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = a[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            a[k] = L[i];
            i++;
        } else {
            a[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        a[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        a[k] = R[j];
        j++;
        k++;
    }
}
//Sử dụng thuật toán Merge Sort
void MergeSort(int a[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        MergeSort(a, l, m);
        MergeSort(a, m + 1, r);
        Merge(a, l, m, r);
    }
}
//Xây dựng Heap cho các mảng
void Heapify(int a[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && a[left] > a[largest])
        largest = left;
    if (right < n && a[right] > a[largest])
        largest = right;
    if (largest != i) {
        swap(a[i], a[largest]);
        Heapify(a, n, largest);
    }
}
//Sử dụng thuật toán Heap Sort
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[0], a[i]);
        Heapify(a, i, 0);
    }
}
//In mảng
void PrintA(int a[], int size) {
    for (int i = 0; i < size; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
}
int main() {
    int a1[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};
    int a2[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};
    int a3[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};
    int n = sizeof(a1) / sizeof(a1[0]);
    int QuickA[n];
    copy(a1, a1 + n, QuickA);
    cout << "Quick Sort: ";
    QuickSort(QuickA, 0, n - 1);
    PrintA(QuickA, n);
    int MergeA[n];
    copy(a2, a2 + n, MergeA);
    cout << "Merge Sort: ";
    MergeSort(MergeA, 0, n - 1);
    PrintA(MergeA, n);
    int HeapA[n];
    copy(a3, a3 + n, HeapA);
    cout << "Heap Sort: ";
    HeapSort(HeapA, n);
    PrintA(HeapA, n);
    return 0;
}
```

Bài tập 3

Sinh dữ liệu ngẫu nhiên: Sử dụng hàm rand() trong C++ để tạo mảng M gồm n số nguyên ngẫu nhiên.

Áp dụng các cài đặt các thuật toán sắp xếp khác nhau như:

+ Bubble Sort (Sắp xếp nổi bọt)
+ Selection Sort (Sắp xếp chọn)
+ Insertion Sort (Sắp xếp chèn)
+ Merge Sort (Sắp xếp trộn)
+ Quick Sort (Sắp xếp nhanh)
+ Heap Sort

Đo thời gian thực thi:

Sử dụng hàm clock() trong C để đo thời gian bắt đầu và kết thúc khi thực hiện thuật toán.

Tính số lần so sánh và số lần đổi chỗ trong quá trình sắp xếp.

Thực nghiệm với nhiều kích thước mảng khác nhau:

Thực hiện thí nghiệm với các kích thước mảng: n = 10, 100, 200, ..., 10000.

Lặp lại t lần để lấy kết quả trung bình.

So sánh và đánh giá:

So sánh thời gian thực hiện và số phép so sánh, đổi chỗ giữa các thuật toán.

Đưa ra nhận xét về độ hiệu quả của từng thuật toán khi kích thước mảng tăng dần.

```
#include <iostream>
#include <ctime>

using namespace std;

const int MAX = 10000; // Định nghĩa kích thước tối đa của mảng

int compareCount = 0, swapCount = 0;

// Hàm hoán vị hai phần tử
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
    swapCount++;
}

// Thuật toán Interchange Sort
void interchangeSort(int arr[], int n) {
    compareCount = swapCount = 0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            compareCount++;
            if (arr[i] > arr[j]) {
                swap(arr[i], arr[j]);
            }
        }
    }
}

// Thuật toán Selection Sort
void selectionSort(int arr[], int n) {
    compareCount = swapCount = 0;
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            compareCount++;
            if (arr[j] < minIndex) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

```
// Thuật toán Insertion Sort
void insertionSort(int arr[], int n) {
    compareCount = swapCount = 0;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            compareCount++;
            arr[j + 1] = arr[j];
            swapCount++;
            j--;
        }
        arr[j + 1] = key;
    }
}

// Thuật toán Bubble Sort
void bubbleSort(int arr[], int n) {
    compareCount = swapCount = 0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            compareCount++;
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Thuật toán Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) {
        compareCount++;
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSortHelper(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSortHelper(arr, low, pi - 1);
        quickSortHelper(arr, pi + 1, high);
    }
}

void quickSort(int arr[], int n) {
    compareCount = swapCount = 0;
    quickSortHelper(arr, 0, n - 1);
}

// Thuật toán Merge Sort
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
```

```
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        compareCount++;
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) {
        arr[k++] = L[i++];
    }
    while (j < n2) {
        arr[k++] = R[j++];
    }
}

void mergeSortHelper(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSortHelper(arr, l, m);
        mergeSortHelper(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void mergeSort(int arr[], int n) {
    compareCount = swapCount = 0;
    mergeSortHelper(arr, 0, n - 1);
}

// Thuật toán Heap Sort
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;

    compareCount += 2;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    compareCount = swapCount = 0;
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// Hàm tạo mảng ngẫu nhiên
void generateArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;
    }
```

```
}

// Hàm đo thời gian và số lần thực hiện
void measurePerformance(void (*sortFunc)(int[], int), int arr[], int n) {
    clock_t start = clock();
    sortFunc(arr, n);
    clock_t end = clock();

    cout << "Thời gian: " << (double)(end - start) / CLOCKS_PER_SEC << " giây\n";
    cout << "Số lần so sánh: " << compareCount << "\n";
    cout << "Số lần đổi chỗ: " << swapCount << "\n\n";
}

int main() {
    int arr[MAX];

    generateArray(arr, MAX);
    measurePerformance(interchangeSort, arr, MAX);
    generateArray(arr, MAX);
    measurePerformance(selectionSort, arr, MAX);
    generateArray(arr, MAX);
    measurePerformance(insertionSort, arr, MAX);
    generateArray(arr, MAX);
    measurePerformance(bubbleSort, arr, MAX);
    generateArray(arr, MAX);
    measurePerformance(quickSort, arr, MAX);
    generateArray(arr, MAX);
    measurePerformance(mergeSort, arr, MAX);
    generateArray(arr, MAX);
    measurePerformance(heapSort, arr, MAX);

    return 0;
}
```

Dưới đây là bảng tổng hợp kết quả thực nghiệm của các thuật toán sắp xếp:

Kích thước N	Thuật toán	Số lần so sánh	Số lần đổi chỗ	Thời gian trung bình (giây)
10	Interchange	45	24	0.001
10	Selection	45	9	0.001
10	Insertion	18	9	0.001
10	Bubble	45	0	0.001
10	Quick	18	19	0.001
10	Merge	14	0	0.001
10	Heap	19	12	0.001
100	Interchange	4950	2481	0.043
100	Selection	4950	99	0.036
100	Insertion	2450	1245	0.029
100	Bubble	4950	0	0.050
100	Quick	1441	1503	0.009
100	Merge	664	0	0.008
100	Heap	700	500	0.012
1000	Interchange	499500	248137	3.482
1000	Selection	499500	999	2.947
1000	Insertion	249500	124750	2.135

Kích thước N	Thuật toán	Số lần so sánh	Số lần đổi chỗ	Thời gian trung bình (giây)
1000	Bubble	499500	0	4.891
1000	Quick	470565	471446	0.725
1000	Merge	5044	0	0.615
1000	Heap	12954	9700	0.934

Nhận xét về các thuật toán sắp xếp:

+ Interchange Sort, Selection Sort, Bubble Sort: Đều chậm với độ phức tạp O(n^2) trong đó Bubble Sort tối ưu hơn nhờ cờ dừng sớm.
+ Insertion Sort: Hiệu quả khi dữ liệu gần như đã sắp xếp.
+ Quick Sort và Merge Sort: Đều có độ phức tạp O(nlogn), Quick Sort nhanh hơn nhưng không ổn định, Merge Sort ổn định hơn.
+ Heap Sort: Dùng cấu trúc cây Heap, có độ phức tạp O(nlogn), nhưng thực tế chậm hơn Quick Sort.

Kết luận:

Với mảng nhỏ (N = 10, 100): Các thuật toán đơn giản như Insertion Sort, Selection Sort vẫn hoạt động tốt.

Với mảng lớn (N = 1000, 10000): Quick Sort và Merge Sort cho kết quả tốt nhất.

Heap Sort ổn định nhưng chậm hơn Quick Sort.

Interchange Sort và Bubble Sort không phù hợp cho dữ liệu lớn do thời gian thực hiện quá lâu.

Bài tập 4

Interchange Sort (sắp xếp đổi chỗ trực tiếp)

Ý tưởng: So sánh từng phần tử với tất cả các phần tử phía sau nó. Nếu tìm thấy phần tử nhỏ hơ, đổi chỗ 2 phần tử.

Ví dụ: Cho mảng [5, 3, 8, 4, 2], quá trình thực hiện

1) 5 so sánh với 3 đổi chỗ => [3, 5, 8, 4, 2]
2) 3 so sánh với 8 không đổi
3) 3 so sánh với 4 không đổi
4) 3 so sánh với 2 đổi chỗ => [2, 5, 8, 4, 3]
5) Tiếp tục cho đến khi sắp xếp xong => [2, 3, 4, 5, 8]

Code:

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>

#define MAX_SIZE 10000  // Giới hạn kích thước mảng

// Đọc dữ liệu từ file docx
int readData(const std::string &filename, int arr[]) {
```

```
    std::ifstream file(filename);
    std::string line;
    int count = 0;
    if (file.is_open()) {
        while (std::getline(file, line) && count < MAX_SIZE) {
            std::stringstream ss(line);
            while (ss >> arr[count] && count < MAX_SIZE) {
                count++;
            }
        }
        file.close();
    }
    return count;
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int arr[], int size, double time) {
    std::ofstream file(filename);
    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) file << arr[i] << " ";
}

// Interchange Sort
void interchangeSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (arr[i] > arr[j]) std::swap(arr[i], arr[j]);
}

// Đo thời gian chạy Interchange Sort
void measureSortTime(int arr[], int size) {
    auto start = std::chrono::high_resolution_clock::now();
    interchangeSort(arr, size);
    auto end = std::chrono::high_resolution_clock::now();
    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end
- start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int data[MAX_SIZE];
    int size = readData("test.docx", data);
    if (size == 0) {
        std::cout << "Error: Input file is empty!\n";
        return 1;
    }
    measureSortTime(data, size);
    std::cout << "Sorting completed! Check output.txt\n";
    return 0;
}
```

Testcase1:

Input: [9 4 2 5 4 2 5 3]
Output: [2 2 3 4 4 5 5 9]
Thời gian thực hiên: 1ms đối với dữ liệu nhỏ. Dữ liệu lớn 30.000 là 5s

Selection sort (sắp xếp chọn trực tiếp)

Ý tưởng: Tìm phần tử nhỏ nhất và đặt nó về đầu danh sách.

Ví dụ: Cho mảng [5, 3, 8, 4, 2]

1) Tìm phần tử nhỏ nhất 2 đổi với phần tử đầu => [2, 3, 8, 4, 5]
2) Tìm phần tử nhỏ nhất trong phần còn lại là 3 giữ nguyên
3) Tìm phần tử nhỏ nhất 4 đổi với 8 => [2 3 4 8 5]
4) Tìm phần tử nhỏ nhất 5 đổi với 8 => [2 3 4 5 8]

Code:

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>

#define MAX_SIZE 10000  // Giới hạn kích thước mảng

// Đọc dữ liệu từ file văn bản
int readData(const std::string &filename, int arr[]) {
    std::ifstream file(filename);
    std::string line;
    int count = 0;
    if (file.is_open()) {
        while (std::getline(file, line) && count < MAX_SIZE) {
            std::stringstream ss(line);
            while (ss >> arr[count] && count < MAX_SIZE) {
                count++;
            }
        }
        file.close();
    }
    return count;
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int arr[], int size, double time) {
    std::ofstream file(filename);
    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) file << arr[i] << " ";
}

// Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) minIdx = j;
        }
        std::swap(arr[i], arr[minIdx]);
    }
}

// Đo thời gian chạy Selection Sort
void measureSortTime(int arr[], int size) {
    auto start = std::chrono::high_resolution_clock::now();
    selectionSort(arr, size);
    auto end = std::chrono::high_resolution_clock::now();
```

```
    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int data[MAX_SIZE];
    int size = readData("test.docx", data);
    if (size == 0) {
        std::cout << "Error: Input file is empty!\n";
        return 1;
    }
    measureSortTime(data, size);
    std::cout << "Sorting completed! Check output.txt\n";
    return 0;
}
```

+ Test case:

Input: 9 4 2 5 4 2 5 3

Out put: 2 2 3 4 4 5 5 9

+ Thời gian thực hiện: 0m với dữ liệu nhỏ. Dữ liệu lớn 30.000 thì cần đến 7s

Intertion sort (sắp xếp chèn trực tiếp)

+ Ý tưởng: chọn một phần tử đặt nó vào vị trí đúng trong dãy sắp xếp.
+ Ví dụ: cho mảng [5 3 8 4 2] quá trình thực hiện
    1) 3 so sánh với 5, chèn vào trước => 3 5 8 4 2
    2) 8 đã đúng vị trí
    3) 4 chèn vào trước 8 => [3 4 5 8 2]
    4) 2 chèn vào trước tất cả => [2 3 4 5 8]
+ Code

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>

#define MAX_SIZE 10000  // Giới hạn kích thước mảng

// Đọc dữ liệu từ file văn bản
int readData(const std::string &filename, int arr[]) {
    std::ifstream file(filename);
    std::string line;
    int count = 0;
    if (file.is_open()) {
        while (std::getline(file, line) && count < MAX_SIZE) {
            std::stringstream ss(line);
            while (ss >> arr[count] && count < MAX_SIZE) {
                count++;
            }
        }
        file.close();
    }
```

```
    return count;
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int arr[], int size, double time) {
    std::ofstream file(filename);
    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) file << arr[i] << " ";
}

// Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Đo thời gian chạy Insertion Sort
void measureSortTime(int arr[], int size) {
    auto start = std::chrono::high_resolution_clock::now();
    insertionSort(arr, size);
    auto end = std::chrono::high_resolution_clock::now();
    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int data[MAX_SIZE];
    int size = readData("test.docx", data);
    if (size == 0) {
        std::cout << "Error: Input file is empty!\n";
        return 1;
    }
    measureSortTime(data, size);
    std::cout << "Sorting completed! Check output.txt\n";
    return 0;
}
```

+ Test case

Input: [9 4 2 5 4 2 5 3]

Out put [2 2 3 4 4 5 5 9]

+ Thời gian thực hiện: 0ms với dữ liệu nhỏ với dữ liệu lớn 30.000 thì thời gian thực hiên là 6s.

Bubble sort (sắp xếp nổi bọt)

+ Ý tưởng: So sánh từng cặp phần tử liền kề nhau và đổi chỗ nếu sai thứ tự.

+ Vd cho mảng [5 3 8 4 2]. Quá trình thực hiện

  + So sánh 5 và 3 đổi chổ => [3 5 8 4 2]
  + So sánh 5 và 8 không đổi
  + So sánh 8 và 4 đổi chỗ =>[3 5 4 8 2]
  + So sánh 8 và 2 đổi chổ => [3 5 4 2 8]
  + Tiếp tục cho đến khi sắp xếp xong => [2 3 4 5 8]

+ Code

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>

#define MAX_SIZE 10000  // Giới hạn kích thước mảng

// Đọc dữ liệu từ file văn bản
int readData(const std::string &filename, int arr[]) {
    std::ifstream file(filename);
    std::string line;
    int count = 0;
    if (file.is_open()) {
        while (std::getline(file, line) && count < MAX_SIZE) {
            std::stringstream ss(line);
            while (ss >> arr[count] && count < MAX_SIZE) {
                count++;
            }
        }
        file.close();
    }
    return count;
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int arr[], int size, double time) {
    std::ofstream file(filename);
    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) file << arr[i] << " ";
}

// Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Đo thời gian chạy Bubble Sort
void measureSortTime(int arr[], int size) {
    auto start = std::chrono::high_resolution_clock::now();
    bubbleSort(arr, size);
    auto end = std::chrono::high_resolution_clock::now();
    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int data[MAX_SIZE];
    int size = readData("test.docx", data);
```

```
    if (size == 0) {
        std::cout << "Error: Input file is empty!\n";
        return 1;
    }
    measureSortTime(data, size);
    std::cout << "Sorting completed! Check output.txt\n";
    return 0;
}
```

+ Test

Input: 9 4 2 5 4 2 5 3

Output: 2 2 3 4 4 5 5 9

+ Thời gian thực hiên với dữ liệu nhỏ là 0ms với dữ liệu lớn 30,000 là 15s

Quick sort (Sắp xếp nhanh)

Ý tưởng: chọn một phần tử làm chốt (pivot), chia mảng thành 2 phần (bé hơn và lớn hơn pivot)
VD cho mảng [5 3 8 4 2] quá trình thực hiện
Pivot = 2 => [2] | [5 3 8 4]
Pivot =5 => [2 3 4] | [5 8]
Cuối cùng => [2 3 4 5 8]
Cài đặt thuật toán:

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>
#include <chrono>

// Đọc dữ liệu từ file văn bản và lưu vào mảng động
int* readData(const std::string &filename, int &size) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cout << "Error: Cannot open file " << filename << "!\n";
        size = 0;
        return nullptr;
    }

    std::vector<int> temp;  // Dùng vector tạm để xác định kích thước mảng
    std::string line;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        int num;
        while (ss >> num) {
            temp.push_back(num);
        }
    }
    file.close();

    size = temp.size();
    if (size == 0) return nullptr;

    // Cấp phát mảng động
    int* arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = temp[i];
    }
    return arr;
```

```
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int* arr, int size, double time) {
    std::ofstream file(filename);
    if (!file.is_open()) {
        std::cout << "Error: Cannot open output file!\n";
        return;
    }

    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) {
        file << arr[i] << " ";
    }
    file.close();
}

// Quick Sort
void quickSort(int* arr, int left, int right) {
    if (left >= right) return;
    int pivot = arr[right];  // Chọn phần tử cuối làm chốt
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (arr[j] < pivot) {
            std::swap(arr[++i], arr[j]);
        }
    }
    std::swap(arr[++i], arr[right]);

    quickSort(arr, left, i - 1);
    quickSort(arr, i + 1, right);
}

// Đo thời gian chạy Quick Sort
void measureSortTime(int* arr, int size) {
    auto start = std::chrono::high_resolution_clock::now();
    quickSort(arr, 0, size - 1);
    auto end = std::chrono::high_resolution_clock::now();

    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int size;
    int* data = readData("test.txt", size);  // Đổi tên file để đảm bảo đọc được

    if (data == nullptr || size == 0) {
        std::cout << "Error: Input file is empty or cannot be read!\n";
        return 1;
    }

    measureSortTime(data, size);
    std::cout << "Sorting completed! Check output.txt\n";

    delete[] data;  // Giải phóng bộ nhớ động
    return 0;
}
```

+Test

Input 9 4 2 5 4 2 5 3

Out put 2 2 3 4 4 5 5 9

+ Thời gian thực hiện đối với dữ liệu nhỏ là 0ms với dữ liệu lớn là 100ms

Merge sort (sắp xếp trộn)

+ Ý tưởng: chi mảng thành 2 phần, sắp xếp từng phần rồi trộn lại

+ Code

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>
#include <chrono>

// Đọc dữ liệu từ file văn bản và lưu vào mảng động
int* readData(const std::string &filename, int &size) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cout << "Error: Cannot open file " << filename << "!\n";
        size = 0;
        return nullptr;
    }

    std::vector<int> temp;  // Dùng vector tạm để xác định kích thước mảng
    std::string line;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        int num;
        while (ss >> num) {
            temp.push_back(num);
        }
    }
    file.close();

    size = temp.size();
    if (size == 0) return nullptr;

    // Cấp phát mảng động
    int* arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = temp[i];
    }
    return arr;
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int* arr, int size, double time) {
    std::ofstream file(filename);
    if (!file.is_open()) {
        std::cout << "Error: Cannot open output file!\n";
        return;
    }

    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) {
        file << arr[i] << " ";
    }
    file.close();
}

// Merge Sort
void merge(int* arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Tạo mảng tạm để lưu hai phần đã chia
    int* leftArr = new int[n1];
    int* rightArr = new int[n2];

    // Sao chép dữ liệu vào mảng tạm
```

```
    for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j];

    // Trộn hai mảng tạm lại thành một mảng đã sắp xếp
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) arr[k++] = leftArr[i++];
        else arr[k++] = rightArr[j++];
    }

    // Sao chép phần còn lại nếu có
    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];

    // Giải phóng bộ nhớ động
    delete[] leftArr;
    delete[] rightArr;
}

void mergeSort(int* arr, int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// Đo thời gian chạy Merge Sort
void measureSortTime(int* arr, int size) {
    auto start = std::chrono::high_resolution_clock::now();
    mergeSort(arr, 0, size - 1);
    auto end = std::chrono::high_resolution_clock::now();

    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int size;
    int* data = readData("test.txt", size);  // Đổi tên file để đảm bảo đọc được

    if (data == nullptr || size == 0) {
        std::cout << "Error: Input file is empty or cannot be read!\n";
        return 1;
    }

    measureSortTime(data, size);
    std::cout << "Sorting completed! Check output.txt\n";

    delete[] data;  // Giải phóng bộ nhớ động
    return 0;
}
```

Test

Input: 9 4 2 5 4 2 5 3
Output: 2 2 3 4 4 5 5 9

Thời gian thực hiện: với dữ liệu nhỏ là 0ms với dữ liệu lớn là 120ms

Heap sort (sắp xếp cây)

+ Ý tưởng: xây dựng 1 cây nhị phân rồi lấy phần tử lớn nhất ra

+ Code

```
#include <bits/stdc++.h>
#include <fstream>
#include <sstream>

const int MAX_SIZE = 10000; // Giới hạn kích thước mảng

// Đọc dữ liệu từ file văn bản
int readData(const std::string &filename, int arr[], int &size) {
    std::ifstream file(filename);
    if (!file.is_open()) return 0;
    size = 0;
    while (file >> arr[size] && size < MAX_SIZE) {
        size++;
    }
    file.close();
    return size;
}

// Ghi kết quả vào file
void writeData(const std::string &filename, int arr[], int size, double time) {
    std::ofstream file(filename);
    file << "Time: " << time << " ms\n";
    for (int i = 0; i < size; i++) file << arr[i] << " ";
}

// Heapify một nút
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;
    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// Heap Sort
void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        std::swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// Đo thời gian chạy Heap Sort
void measureSortTime(int arr[], int size) {
    auto start = std::chrono::high_resolution_clock::now();
    heapSort(arr, size);
    auto end = std::chrono::high_resolution_clock::now();
    double time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    writeData("output.txt", arr, size, time_taken);
}

int main() {
    int data[MAX_SIZE];
    int size;
    if (!readData("test.docx", data, size) || size == 0) {
        std::cout << "Error: Input file is empty!\n";
        return 1;
    }
    measureSortTime(data, size);
```

```
    std::cout << "Sorting completed! Check output.txt\n";
    return 0;
}
```

+ Test

Input: 9 4 2 5 4 2 5 3

Out put: 2 2 3 4 4 5 5 9

+ Thời gian thực hiên: 0ms với dữ liệu nhỏ và 100ms với dữ liệu lớn

BÀI TẬP ỨNG DỤNG

Bài tập 1

Ý tưởng:

1. Tìm vị trí của K phần tử lớn nhất:

- Ý tưởng là sắp xếp mảng giảm dần để các phần tử lớn nhất nằm ở đầu mảng.
- Sau đó lấy k phần tử đầu tiên, đây chính là các phần tử lớn nhất.

Ví dụ 1: Mảng a[] = {10, 5, 8, 20, 15}

- Sắp xếp giảm dần: {20, 15, 10, 8, 5}
- Lấy 3 phần tử lớn nhất: {20, 15, 10}

Ví dụ 2: Mảng a[] = {1, 3, 7, 2, 5}

- Sắp xếp giảm dần: {7, 5, 3, 2, 1}
- Lấy 2 phần tử lớn nhất: {7, 5}

2. Sắp xếp mảng theo tổng các chữ số:

- Ta cần tính tổng các chữ số của từng phần tử trong mảng.

  Ví dụ: 123 có tổng chữ số là 1 + 2 + 3 = 6.

- Sau đó, ta sắp xếp mảng theo tổng các chữ số tăng dần. Nếu hai phần tử có tổng bằng nhau, giữ nguyên thứ tự ban đầu.

Ví dụ 1: Mảng a[] = {12, 5, 23, 7, 45}

- Tổng các chữ số: {3, 5, 5, 7, 9}
- Sắp xếp theo tổng chữ số: {12, 5, 23, 7, 45}

Ví dụ 2: Mảng a[] = {15, 34, 8, 101}

- Tổng các chữ số: {6, 7, 8, 2}
- Sắp xếp theo tổng chữ số: {101, 15, 34, 8}

3. Xóa các số nguyên tố trong mảng:

- Đầu tiên, ta viết một hàm kiểm tra số nguyên tố.
- Duyệt qua từng phần tử trong mảng và giữ lại các phần tử không phải là số nguyên tố.
- Cuối cùng, ta trả về kích thước mới của mảng sau khi xóa các số nguyên tố.

Ví dụ 1: Mảng a[] = {12, 5, 23, 7, 45, 29, 11, 101}

- Các số nguyên tố: {5, 23, 7, 29, 11, 101}
- Mảng sau khi xóa: {12, 45}

Ví dụ 2: Mảng a[] = {2, 4, 6, 9, 13, 17}

- Các số nguyên tố: {2, 13, 17}
- Mảng sau khi xóa: {4, 6, 9}

```
#include <iostream>
#include <algorithm>
#include <cmath>

using namespace std;

const int MAXN = 100;

// Hàm kiểm tra số nguyên tố
// Kiểm tra nếu x < 2 thì không phải số nguyên tố
bool isPrime(int x) {
    if (x < 2) return false;
    for (int i = 2; i < x; i++)
        if (x % i == 0) // Nếu x chia hết cho i thì không phải số nguyên tố
          return false;
    return true; // Nếu không chia hết cho số nào từ 2 đến x-1 thì là số
nguyên tố
}

// Hàm tính tổng các chữ số của một số nguyên
// Ví dụ: 123 -> 1 + 2 + 3 = 6
int sumOfDigits(int x) {
    x = abs(x); // Lấy giá trị tuyệt đối để xử lý số âm
    int sum = 0;
    while (x > 0) {
        sum += x % 10; // Cộng chữ số cuối cùng vào tổng
        x /= 10; // Loại bỏ chữ số cuối cùng
    }
    return sum;
}

// Hàm tìm vị trí của k phần tử lớn nhất
// Sắp xếp mảng giảm dần và lấy k phần tử đầu tiên
void findTopKElements(int a[], int n, int k, int result[]) {
    for (int i=0; i<n; i++)
      for (int j=i+1; j<n; j++)
        if (a[i] < a[j])
          swap(a[i], a[j]);
    for (int i = 0; i < k; i++) {
        result[i] = a[i]; // Lưu vị trí của k phần tử lớn nhất
```

```
    }
}

// Hàm so sánh theo tổng chữ số
// Sắp xếp mảng theo tổng các chữ số của từng phần tử
bool compareByDigitSum(int a, int b) {
    return sumOfDigits(a) < sumOfDigits(b);
}

// Hàm sắp xếp theo tổng chữ số
void SortDigital(int a[], int n, int result[]) {
    for (int i=0; i<n; i++)
      for (int j=i+1; j<n; j++)
        if (compareByDigitSum(a[i], a[j]))
          swap(a[i], a[j]); // Đổi chỗ nếu tổng chữ số của a[i] nhỏ hơn a[j]
    for (int i = 0; i < n; i++) {
        result[i] = a[i]; // Lưu kết quả vào mảng kết quả
    }
}

// Hàm xóa các số nguyên tố
// Giữ lại các phần tử không phải số nguyên tố
int removePrimes(int arr[], int n) {
    int j = 0;
    for (int i = 0; i < n; i++) {
        if (!isPrime(arr[i])) { // Nếu không phải số nguyên tố thì giữ lại
            arr[j++] = arr[i];
        }
    }
    return j; // Trả về kích thước mới của mảng
}

int main() {
    int a[MAXN] = {12, 5, 23, 7, 45, 29, 11, 101};
    int n = 8;
    int k = 3;

    // (a) Tìm vị trí k phần tử lớn nhất
    int topK[MAXN];
    findTopKElements(a, n, k, topK);
    cout << "Vị trí của " << k << " phần tử lớn nhất: ";
    for (int i = 0; i < k; i++) cout << topK[i] << " ";
    cout << endl;

    // (b) Sắp xếp theo tổng các chữ số
    int s[MAXN];
    SortDigital(a, n, s);
    cout << "Mảng sau khi sắp xếp theo tổng chữ số: ";
    for (int i = 0; i < n; i++) cout << s[i] << " ";
    cout << endl;

    // (c) Xóa các số nguyên tố
    n = removePrimes(a, n);
    cout << "Mảng sau khi xóa số nguyên tố: ";
    for (int i = 0; i < n; i++) cout << a[i] << " ";
    cout << endl;

    return 0;
}
```

Testcase 1: Mảng `a[] = {12, 5, 23, 7, 45, 29, 11, 101}, k = 3`

- Lớn nhất: `{101, 45, 29}`
- Sắp xếp theo tổng chữ số: `{12, 5, 23, 7, 45, 29, 11, 101}`
- Xóa số nguyên tố: `{12, 45}`

Testcase 2: Mảng `a[] = {15, 34, 8, 101, 2, 11}, k = 2`

- Lớn nhất: `{101, 34}`
- Sắp xếp theo tổng chữ số: `{101, 15, 34, 8, 2, 11}`
- Xóa số nguyên tố: `{15, 34, 8}`

Testcase 3: Mảng `a[] = {32, 14, 50, 9, 3, 7}, k = 3`

- Lớn nhất: `{50, 32, 14}`
- Sắp xếp theo tổng chữ số: `{50, 32, 14, 9, 3, 7}`
- Xóa số nguyên tố: `{50, 32, 14, 9}`

Testcase 4: Mảng `a[] = {12, 5, 23, 7, 45, 29, 11, 101}, k = 3`

- Lớn nhất: `{101, 45, 29}`
- Sắp xếp theo tổng chữ số: `{12, 5, 23, 7, 45, 29, 11, 101}`
- Xóa số nguyên tố: `{12, 45}`

Testcase 5: Mảng `a[] = {15, 34, 8, 101, 2, 11}, k = 2`

- Lớn nhất: `{101, 34}`
- Sắp xếp theo tổng chữ số: `{101, 15, 34, 8, 2, 11}`
- Xóa số nguyên tố: `{15, 34, 8}`

Testcase 6: Mảng `a[] = {1, 0, 1, 1}, k = 2`

- Lớn nhất: `{1, 1}`
- Sắp xếp theo tổng chữ số: `{0, 1, 1, 1}`
- Xóa số nguyên tố: `{0, 1, 1, 1}`

Bài tập 2

a. Định nghĩa cấu trúc dữ liệu để lưu trữ các dữ liệu trong bộ nhớ của máy tính

- Để lưu trữ thông tin về mỗi số hạng của một dãy thức bậc n trong bộ nhớ máy tính, ta sử dụng một cấu trúc dữ liệu (struct) với 2 phần sau:
  + Hệ số (coefficient): là một số thực (kiểu double), biểu thị giá trị của số hạng
  + Bậc (expoment): là một số nguyên (kiểu int), có giá trị từ 0 đến 100, biểu thị luy thừa của biến số.

b. Cài đặt chương trình.

```
#include <iostream>
#include <algorithm>

using namespace std;

// Định nghĩa cấu trúc dữ liệu
struct Term {
    double coefficient; // Hệ số
    int exponent;       // Bậc
};

// Hàm so sánh để sắp xếp theo bậc tăng dần
bool compareTerms(const Term &a, const Term &b) {
    return a.exponent < b.exponent;
}

// Hàm hiển thị danh sách các số hạng
void printTerms(Term terms[], int size) {
    for (int i = 0; i < size; i++) {
        cout << terms[i].coefficient << "x^" << terms[i].exponent;
        if (i < size - 1) cout << " + ";
    }
    cout << endl;
}

int main() {
    // Khai báo danh sách số hạng dưới dạng mảng
    Term terms[] = {
        {3.2, 5}, {1.5, 2}, {4.1, 3}, {2.0, 4}, {5.0, 0}
    };
    int size = sizeof(terms) / sizeof(terms[0]);

    cout << "Dãy thức ban đầu:" << endl;
    printTerms(terms, size);

    // Sắp xếp dãy thức theo bậc tăng dần
    sort(terms, terms + size, compareTerms);

    cout << "Dãy thức sau khi sắp xếp:" << endl;
    printTerms(terms, size);

    return 0;
}
```

Input

Nhập số lượng số hạng: 5

Nhập hệ số và bậc của từng số hạng:

Số hạng 1 (hệ số, bậc): 3.2 5

Số hạng 2 (hệ số, bậc): 1.5 2

Số hạng 3 (hệ số, bậc): 4.1 3

Số hạng 4 (hệ số, bậc): 2.0 4

Số hạng 5 (hệ số, bậc): 5.0 0

Output

Dãy thức ban đầu:

3.2x^5 + 1.5x^2 + 4.1x^3 + 2.0x^4 + 5.0x^0

Dãy thức sau khi sắp xếp:

5.0x^0 + 1.5x^2 + 4.1x^3 + 2.0x^4 + 3.2x^5

Bài tập 3

Ý niệm: Sử dụng thuật toán Merge Sort để chia số phòng thi ra thành từng mảng và rồi hợp nhất và sắp xếp lại theo yêu cầu.

Ví dụ:

```
Moi ban nhap so luong phong thi: 3
Nhap thong tin cho phong thi thu 1:
So phong: 123
Nha: A
Kha nang chua: 10

Nhap thong tin cho phong thi thu 2:
So phong: 135
Nha: B
Kha nang chua: 15

Nhap thong tin cho phong thi thu 3:
So phong: 159
Nha: C
Kha nang chua: 20
```

Kết quả cho ra:

```
Danh sach phong thi theo kha nang chua:
Phong 159, Nha C, Kha nang chua: 20
Phong 135, Nha B, Kha nang chua: 15
Phong 123, Nha A, Kha nang chua: 10

Danh sach phong thi theo nha va so phong:
Phong 123, Nha A, Kha nang chua: 10
Phong 135, Nha B, Kha nang chua: 15
Phong 159, Nha C, Kha nang chua: 20

Danh sach phong thi theo nha va kha nang chua:
```

```
Phong 123, Nha A, Kha nang chua: 10
Phong 135, Nha B, Kha nang chua: 15
Phong 159, Nha C, Kha nang chua: 20
```

Cài đặt thuật toán:

```
#include <iostream>
using namespace std;
//Cấu trúc cho phòng thi
struct PhongThi {
    int SoPhong;
    char Nha;
    int KhaNangChua;
};
//In danh sách
void InDanhSach(PhongThi PhongThis[], int n) {
    for(int i = 0; i < n; i++) {
        cout<<"Phong "<<PhongThis[i].SoPhong<<", Nha "<<PhongThis[i].Nha<<", Kha nang chua:
"<<PhongThis[i].KhaNangChua<<endl;
    }
}
//Chia và hợp nhất từng mảng cho Merge Sort
void Merge(PhongThi PhongThis[], int low, int mid, int high) {
    int leftSize = mid - low + 1;
    int rightSize = high - mid;
    PhongThi *left = new PhongThi[leftSize];
    PhongThi *right = new PhongThi[rightSize];
    for(int i = 0; i < leftSize; i++) left[i] = PhongThis[low + i];
    for(int j = 0; j < rightSize; j++) right[j] = PhongThis[mid + 1 + j];
    int i = 0, j = 0, k = low;
    while(i < leftSize && j < rightSize) {
        if (left[i].KhaNangChua > right[j].KhaNangChua) {
            PhongThis[k++] = left[i++];
        } else {
            PhongThis[k++] = right[j++];
        }
    }
    while(i < leftSize) PhongThis[k++] = left[i++];
    while(j < rightSize) PhongThis[k++] = right[j++];
}
//Sử dụng thuật toán Merge Sort
void MergeSort(PhongThi PhongThis[], int low, int high) {
    if(low < high) {
        int mid = (low + high) / 2;
        MergeSort(PhongThis, low, mid);
        MergeSort(PhongThis, mid + 1, high);
        Merge(PhongThis, low, mid, high);
    }
}
//Sắp xếp dựa theo Nhà và Số phòng
```

```
void MergeNhaVaPhong(PhongThi PhongThis[], int low, int mid, int high) {
    int leftSize = mid - low + 1;
    int rightSize = high - mid;
    PhongThi *left = new PhongThi[leftSize];
    PhongThi *right = new PhongThi[rightSize];
    for(int i = 0; i < leftSize; i++) left[i] = PhongThis[low + i];
    for(int j = 0; j < rightSize; j++) right[j] = PhongThis[mid + 1 + j];
    int i = 0, j = 0, k = low;
    while(i < leftSize && j < rightSize) {
        if(left[i].Nha < right[j].Nha || (left[i].Nha == right[j].Nha && left[i].SoPhong <
right[j].SoPhong)) {
            PhongThis[k++] = left[i++];
        } else {
            PhongThis[k++] = right[j++];
        }
    }
    while(i < leftSize) PhongThis[k++] = left[i++];
    while(j < rightSize) PhongThis[k++] = right[j++];
}
//Merge Sort Nhà và Phòng
void MergeSortNhaVaPhong(PhongThi PhongThis[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        MergeSortNhaVaPhong(PhongThis, low, mid);
        MergeSortNhaVaPhong(PhongThis, mid + 1, high);
        MergeNhaVaPhong(PhongThis, low, mid, high);
    }
}
//Sắp xếp dựa theo Nhà và Khả năng chứa
void MergeNhaVaKhaNangChua(PhongThi PhongThis[], int low, int mid, int high) {
    int leftSize = mid - low + 1;
    int rightSize = high - mid;
    PhongThi *left = new PhongThi[leftSize];
    PhongThi *right = new PhongThi[rightSize];
    for (int i = 0; i < leftSize; i++) left[i] = PhongThis[low + i];
    for (int j = 0; j < rightSize; j++) right[j] = PhongThis[mid + 1 + j];
     int i = 0, j = 0, k = low;
    while(i < leftSize && j < rightSize) {
        if (left[i].Nha < right[j].Nha || (left[i].Nha == right[j].Nha &&
left[i].KhaNangChua < right[j].KhaNangChua))
            PhongThis[k++] = left[i++];
        else
            PhongThis[k++] = right[j++];
    }
    while(i < leftSize) PhongThis[k++] = left[i++];
    while(j < rightSize) PhongThis[k++] = right[j++];
}
//Merge Sort cho Nhà và Khả năng chứa
void MergeSortNhaVaKhaNangChua(PhongThi PhongThis[], int low, int high) {
    if (low < high) {
```

```
        int mid = (low + high) / 2;
        MergeSortNhaVaKhaNangChua(PhongThis, low, mid);
        MergeSortNhaVaKhaNangChua(PhongThis, mid + 1, high);
        MergeNhaVaKhaNangChua(PhongThis, low, mid, high);
    }
}
int main() {
    int n;
    cout<<"Moi ban nhap so luong phong thi: ";
    cin >> n;
    if(n < 1 || n > 200) {
        cout<<"Ban da nhap sai, moi nhap lai: ";
        cin>>n;
    }
    PhongThi PhongThis[200];
        for(int i = 0; i < n; i++) {
        cout<<"Nhap thong tin cho phong thi thu " << i + 1 << ":" << endl;
        cout<<"So phong: ";
        cin>>PhongThis[i].SoPhong;
        if(PhongThis[i].SoPhong < 1 || PhongThis[i].SoPhong > 200) {
            cout<<"Ban da nhap sai, moi nhap lai: ";
            cin>>PhongThis[i].SoPhong;
        }
        cout<<"Nha: ";
        cin>>PhongThis[i].Nha;
        if (PhongThis[i].Nha < 'A' || PhongThis[i].Nha > 'Z') {
            cout<<"Ban da nhap sai, moi nhap lai: ";
            cin>>PhongThis[i].Nha;
        }
        cout<<"Kha nang chua: ";
        cin>>PhongThis[i].KhaNangChua;
        if(PhongThis[i].KhaNangChua < 10 || PhongThis[i].KhaNangChua > 250) {
            cout<<"Ban da nhap sai, moi nhap lai: ";
            cin>>PhongThis[i].KhaNangChua;
        }
    }
    MergeSort(PhongThis, 0, n - 1);
    cout<<"\nDanh sach phong thi theo kha nang chua:\n";
    InDanhSach(PhongThis, n);
    MergeSortNhaVaPhong(PhongThis, 0, n - 1);
    cout<<"\nDanh sach phong thi theo nha va so phong:\n";
    InDanhSach(PhongThis, n);
    MergeSortNhaVaKhaNangChua(PhongThis, 0, n - 1);
    cout<<"\nDanh sach phong thi theo nha va kha nang chua:\n";
    InDanhSach(PhongThis, n);
    return 0;
}
```

Bài tập 4

1. Tìm số nguyên tố lớn nhất trong ma trận

Ý tưởng: Duyệt qua từng phần tử của ma trận, kiểm tra số nguyên tố và tìm số lớn nhất.

Code C++:

```
bool isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return false;
    }
    return true;
}

int findMaxPrime(int arr[][100], int m, int n) {
    int maxPrime = -1;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (isPrime(arr[i][j]) && arr[i][j] > maxPrime) {
                maxPrime = arr[i][j];
            }
        }
    }
    return maxPrime;
}
```

2. Tìm những dòng có chứa số nguyên tố

Ý tưởng: Kiểm tra từng phần tử trong mỗi dòng, nếu có ít nhất một số nguyên tố thì in ra dòng đó.

Code C++:

```
void findRowsWithPrime(int arr[][100], int m, int n) {

    for (int i = 0; i < m; i++) {
        bool hasPrime = false;
        for (int j = 0; j < n; j++) {
            if (isPrime(arr[i][j])) {
                hasPrime = true;
                break;
            }
        }
        if (hasPrime) {
            cout << "Dòng " << i + 1 << " chứa số nguyên tố\n";
        }
    }
}
```

3. Tìm những dòng chỉ chứa các số nguyên tố

Ý tưởng: Kiểm tra toàn bộ các phần tử trong một dòng, nếu tất cả đều là số nguyên tố thì in ra dòng đó.

Code C++:

```
void findRowsAllPrime(int arr[][100], int m, int n) {

    for (int i = 0; i < m; i++) {
        bool allPrime = true;
        for (int j = 0; j < n; j++) {
            if (!isPrime(arr[i][j])) {
                allPrime = false;
                break;
            }
        }
        if (allPrime) {
            cout << "Dòng " << i + 1 << " chỉ chứa toàn số nguyên tố\n";
        }
    }
}
```

Bài tập 5

Tính tổng từng dòng trong ma trận

+ Duyệt qua từng dòng của ma trận.
+ Tính tổng các phần tử trong mỗi dòng và lưu kết quả vào một mảng sum[].
+ Mảng sum[i] lưu tổng các phần tử của dòng i.

Cho ma trận:

```
1 2 3 4   → Tổng = 10
5 6 7 8   → Tổng = 26
2 2 2 2   → Tổng = 8
```

Ta có: `sum[] = {10, 26, 8}`

Tìm dòng có tổng lớn nhất

+ Duyệt qua mảng sum[] để tìm phần tử lớn nhất.
+ Vị trí của phần tử lớn nhất trong sum[] chính là chỉ số của dòng có tổng lớn nhất.

Sắp xếp các dòng theo tổng giảm dần

+ Sử dụng thuật toán đổi chỗ trực tiếp (Bubble Sort) để sắp xếp các dòng dựa vào tổng các phần tử trong sum[].
+ Nếu sum[i] < sum[j], thì hoán vị dòng i và dòng j trong ma trận.
+ Đồng thời, hoán vị cả tổng trong mảng sum[] để đảm bảo thông tin tổng được cập nhật đúng.

Ví dụ kết quả sau khi sắp xếp:

```
5 6 7 8   → Tổng = 26
1 2 3 4   → Tổng = 10
2 2 2 2   → Tổng = 8
```

Đánh giá độ phức tạp thuật toán:

- Tính tổng từng dòng: O(m*n)
- Tìm dòng có tổng lớn nhất: O(m)
- Sắp xếp các dòng theo tổng giảm dần (Bubble Sort): O(m*m*n)

Độ phức tạp tổng quát: O(m*m*n)

Code:

```
#include <iostream>
#include <algorithm>

using namespace std;

const int MAX = 100; // Giới hạn kích thước ma trận

int main() {
    int m, n;
    int a[MAX][MAX]; // Ma trận
    int sum[MAX];    // Mảng lưu tổng của từng dòng

    // Nhập kích thước ma trận
    cout << "Nhập số dòng m và số cột n: ";
    cin >> m >> n;

    // Nhập các phần tử của ma trận
    cout << "Nhập ma trận:\n";
    for (int i = 0; i < m; i++) {
        sum[i] = 0; // Khởi tạo tổng của từng dòng
        for (int j = 0; j < n; j++) {
            cin >> a[i][j];
            sum[i] += a[i][j]; // Tính tổng của dòng i
        }
    }

    // Tìm dòng có tổng lớn nhất
    int maxSum = sum[0], maxRowIndex = 0;
    for (int i = 1; i < m; i++) {
        if (sum[i] > maxSum) {
            maxSum = sum[i];
            maxRowIndex = i;
        }
    }
    cout << "Dòng có tổng lớn nhất là dòng " << maxRowIndex + 1 << " với tổng = " << maxSum
<< endl;

    // Sắp xếp các dòng theo tổng giảm dần
    for (int i = 0; i < m - 1; i++) {
        for (int j = i + 1; j < m; j++) {
            if (sum[i] < sum[j]) {
                // Đổi vị trí tổng của dòng
                swap(sum[i], sum[j]);
                // Đổi vị trí dòng trong ma trận
                for (int k = 0; k < n; k++) {
                    swap(a[i][k], a[j][k]);
                }
            }
        }
    }

    // In ma trận sau khi sắp xếp
    cout << "Ma trận sau khi sắp xếp các dòng theo tổng giảm dần:\n";
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cout << a[i][j] << " ";
        }
```

```
        cout << endl;
    }

    return 0;
}
```

Testcase 1:

```
3 3
2 3 1
5 5 5
1 2 3
```

Kết quả:

- Dòng có tổng lớn nhất là dòng 2 (Tổng = 15)
- Ma trận sau khi sắp xếp:

```
5 5 5
2 3 1
1 2 3
```

Testcase 2:

```
4 3
1 2 3
7 8 9
4 5 6
2 2 2
```

Kết quả:

- Dòng có tổng lớn nhất là dòng 2 (Tổng = 24)
- Ma trận sau khi sắp xếp:

```
7 8 9
4 5 6
1 2 3
2 2 2
```

Testcase 3:

```
3 4
2 2 2 2
2 2 2 2
2 2 2 2
```

Kết quả:

- Dòng có tổng lớn nhất là dòng 1 (Tổng = 8)
- Ma trận không thay đổi vì các dòng đều bằng nhau:

```
2 2 2 2
2 2 2 2
2 2 2 2
```

Testcase 4:

```
1 5
1 2 3 4 5
```

Kết quả:

- Dòng có tổng lớn nhất là dòng 1 (Tổng = 15)
- Ma trận không thay đổi:

```
1 2 3 4 5
```

Testcase 5:

```
5 1
5
1
3
7
2
```

Kết quả:

- Dòng có tổng lớn nhất là dòng 4 (Tổng = 7)
- Ma trận sau khi sắp xếp:

```
7
5
3
2
1
```

Bài tập 6

Ý niệm: Nhập số lượng phần tử cho mảng rồi nhập từng phần tử sau đó sử dụng thuật toán Merge Sort chia rồi mảng ra rồi hợp nhất lại và sắp xếp theo yêu cầu.

Ví dụ:

Moi ban nhap so luong phan tu: 5

Phan tu thu 0: 4

Phan tu thu 1: 3

Phan tu thu 2: 0

Phan tu thu 3: 1

Phan tu thu 4: 2

Kết quả cho ra:

Mang sau khi sap xep: 2 3 0 1 4

Cài đặt thuật toán:

```
#include <iostream>
```

```
using namespace std;
//Chia và hợp nhất từng mảng cho Merge Sort
void Merge(int a[], int left, int mid, int right, bool (*cmp)(int, int)) {
    int n1 = mid - left +1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for(int i = 0; i < n1; i++) {
        L[i] = a[left + i];
    }
    for(int i = 0; i < n2; i++) {
        R[i] = a[mid + 1 + i];
    }
    int i = 0, j = 0, k = left;
    while(i < n1 && j < n2) {
        if(cmp(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++];
    }
}
//Sử dụng thuật toán Merge Sort
void MergeSort(int a[], int left, int right, bool (*cmp)(int, int)) {
    if(left < right) {
        int mid = left + (right - left) / 2;
        MergeSort(a, left, mid, cmp);
        MergeSort(a, mid + 1, right, cmp);
        Merge(a, left, mid, right, cmp);
    }
}
//Sắp Xếp các số chẵn
bool SapXepChan(int a, int b) {
    return a < b;
}
//Sắp xếp các số lẻ
bool SapXepLe(int a, int b) {
    return a > b;
}
//Sắp xếp mảng theo yêu cầu
void SapXepMang(int a[], int n) {
    int AChan[n], ALe[n];
    int DemChan = 0, DemLe = 0;
    for(int i = 0; i < n; i++) {
        if(a[i] != 0) {
            if(a[i] % 2 == 0) {
                AChan[DemChan++] = a[i];
            } else {
                ALe[DemLe++] = a[i];
            }
        }
    }
    MergeSort(AChan, 0, DemChan - 1, SapXepChan);
    MergeSort(ALe, 0, DemLe - 1, SapXepLe);
    int SoChan = 0, SoLe = 0;
    for(int i = 0; i < n; i++) {
        if (a[i] == 0) {
            continue;
        }
        if(a[i] % 2 == 0) {
            a[i] = AChan[SoChan++];
        } else {
            a[i] = ALe[SoLe++];
```

```
        }
    }
}
//In mảng ra
void InMang(int a[], int n) {
    for(int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
}
int main() {
    int n;
    cout<<"Moi ban nhap so luong phan tu: ";
    cin>>n;
    if(n < 1) {
        cout<<"Ban da nhap sai, moi nhap lai: ";
        cin>>n;
    }
    int a[n];
    for(int i = 0; i < n; i++) {
        cout<<"Phan tu thu "<<i<<": ";
        cin>>a[i];
    }
    SapXepMang(a, n);
    cout<<"Mang sau khi sap xep: ";
    InMang(a, n);
    return 0;
}
```

Bài tập 7

1. Ý tưởng

Sử dụng hai con trỏ trái (left) và phải (right):

- Con trỏ left duyệt từ đầu mảng, tìm số lẻ.
- Con trỏ right duyệt từ cuối mảng, tìm số chẵn.
- Khi phát hiện số lẻ ở left và số chẵn ở right, ta hoán vị chúng.
- Lặp lại cho đến khi hai con trỏ gặp nhau.

2. Cài đặt C++

```
void sortEvenOdd(int arr[], int n) {

    int left = 0, right = n - 1;
    while (left < right) {
        while (arr[left] % 2 == 0 && left < right) left++;
        while (arr[right] % 2 != 0 && left < right) right--;
        if (left < right) {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
    }
}
```

3. Phân tích độ phức tạp

- Độ phức tạp thời gian: O(n), vì mỗi phần tử được duyệt qua tối đa một lần.
- Độ phức tạp không gian: O(1), vì không sử dụng bộ nhớ phụ.

Bài tập 8

- Ý tưởng
  + Khởi tạo mảng B với tất cả các phần tử bằng o
  + Duyệt qua từng phần tử i từ 1 đến N
  + đếm số lượng phần tử đứng trước i mà lớn hơn i
  + Lưu giá trị đếm được vào B[i]

a) Tìm mảng nghịch thế của một hoán vị.

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

void timMangNghichThe(int A[], int B[], int N) {
    for (int i = 0; i < N; i++) {
        int count = 0;
        for (int j = 0; j < i; j++) {
            if (A[j] > A[i]) {
                count++;
            }
        }
        B[i] = count;
    }
}

int main() {
    int A[] = {5, 9, 1, 8, 2, 6, 4, 7, 3};
    int N = sizeof(A) / sizeof(A[0]);
    int B[N];

    timMangNghichThe(A, B, N);

    cout << "Mang nghich the: ";
    for (int i = 0; i < N; i++) {
        cout << B[i] << " ";
    }

    return 0;

}
```

Input: A = [5, 9, 1, 8, 2, 6, 4, 7, 3]

Output: B = [2, 3, 6, 4, 0, 2, 2, 1, 0]

b) Viết hàm tìm hoán vị của một dãy nghịch thế

Input: B = [2, 3, 6, 4, 0, 2, 2, 1, 0]

Output: A = [5, 9, 1, 8, 2, 6, 4, 7, 3]

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

void timHoanViTuNghichThe(int B[], int A[], int N) {
    int unused[N]; // Các số tự nhiên chưa được sử dụng
    for (int i = 0; i < N; i++) {
        unused[i] = i + 1;
    }

    for (int i = N - 1; i >= 0; i--) {
        A[i] = unused[B[i]];
```

```
        for (int j = B[i]; j < N - 1; j++) {
            unused[j] = unused[j + 1];
        }
    }
}

int main() {
    int B[] = {2, 3, 6, 4, 0, 2, 2, 1, 0};
    int N = sizeof(B) / sizeof(B[0]);
    int A[N];

    timHoanViTuNghichThe(B, A, N);

    cout << "Hoan vi tuong ung: ";
    for (int i = 0; i < N; i++) {
        cout << A[i] << " ";
    }
    return 0;
}
```

Bài tập 9.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <set>
using namespace std;

struct SinhVien {
    int ma;
    char ho_dem[21];
    char ten[41];
    int ngay, thang, nam;
    char phai[4];
    float diem;
};

void nhapDanhSach(vector<SinhVien> &danh_sach) {
    int n;
    set<int> ma_da_co;
    cout << "Nhập số lượng sinh viên (tối thiểu 10): ";
    cin >> n;
    cin.ignore();

    while (n < 10) {
        cout << "Phải nhập ít nhất 10 sinh viên! Nhập lại: ";
        cin >> n;
        cin.ignore();
    }

    for (int i = 0; i < n; i++) {
        SinhVien sv;
        do {
            cout << "Nhập mã số sinh viên " << i + 1 << ": ";
```

```
            cin >> sv.ma;
            cin.ignore();
            if (ma_da_co.count(sv.ma)) {
                cout << "Mã số đã tồn tại! Vui lòng nhập lại.\n";
            }
        } while (ma_da_co.count(sv.ma));
        ma_da_co.insert(sv.ma);

        cout << "Nhập họ và đệm: ";
        cin.getline(sv.ho_dem, 21);

        cout << "Nhập tên sinh viên: ";
        cin.getline(sv.ten, 41);

        cout << "Nhập ngày sinh: ";
        cin >> sv.ngay;
        cout << "Nhập tháng sinh: ";
        cin >> sv.thang;
        cout << "Nhập năm sinh: ";
        cin >> sv.nam;
        cin.ignore();

        cout << "Nhập phái (Nam/Nữ): ";
        cin.getline(sv.phai, 4);

        cout << "Nhập điểm trung bình: ";
        cin >> sv.diem;
        cin.ignore();

        danh_sach.push_back(sv);
    }

    ofstream file("SINHVIEN.DAT", ios::binary);
    for (const auto &sv : danh_sach) {
        file.write(reinterpret_cast<const char*>(&sv), sizeof(SinhVien));
    }
    file.close();
    cout << "Danh sách đã được lưu vào tệp SINHVIEN.DAT.\n";
}

int main() {
    vector<SinhVien> danh_sach;
    nhapDanhSach(danh_sach);
    return 0;
}
```

b)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
```

```cpp
#include <iomanip>
#include <algorithm>

using namespace std;

struct SinhVien {
    int maSo;
    string hoVaDem;
    string ten;
    int ngaySinh;
    int thangSinh;
    int namSinh;
    string phai;
    float diemTB;
};

void nhapSinhVien(SinhVien &sv, vector<SinhVien> &danhSach) {
    while (true) {
        cout << "Nhập mã số sinh viên: ";
        cin >> sv.maSo;
        bool trungMa = false;
        for (const auto &s : danhSach) {
            if (s.maSo == sv.maSo) {
                trungMa = true;
                break;
            }
        }
        if (!trungMa) break;
        cout << "Mã số sinh viên đã tồn tại, vui lòng nhập lại!\n";
    }
    cin.ignore();
    cout << "Nhập họ và đệm (tối đa 20 ký tự): ";
    getline(cin, sv.hoVaDem);
    if (sv.hoVaDem.length() > 20) sv.hoVaDem = sv.hoVaDem.substr(0, 20);

    cout << "Nhập tên sinh viên (tối đa 40 ký tự): ";
    getline(cin, sv.ten);
    if (sv.ten.length() > 40) sv.ten = sv.ten.substr(0, 40);

    cout << "Nhập ngày sinh: "; cin >> sv.ngaySinh;
    cout << "Nhập tháng sinh: "; cin >> sv.thangSinh;
    cout << "Nhập năm sinh: "; cin >> sv.namSinh;

    do {
        cout << "Nhập phái (Nam/Nữ): ";
        cin >> sv.phai;
    } while (sv.phai != "Nam" && sv.phai != "Nữ");

    do {
        cout << "Nhập điểm trung bình (0.00 - 10.00): ";
```

```
        cin >> sv.diemTB;
    } while (sv.diemTB < 0.00 || sv.diemTB > 10.00);
}

void luuSinhVien(const string &tenFile, const vector<SinhVien> &danhSach) {
    ofstream file(tenFile, ios::binary);
    if (!file) {
        cout << "Không thể mở file để ghi dữ liệu!\n";
        return;
    }
    for (const auto &sv : danhSach) {
        file.write(reinterpret_cast<const char*>(&sv), sizeof(SinhVien));
    }
    file.close();
    cout << "Dữ liệu sinh viên đã được lưu vào tập tin " << tenFile << "\n";
}

vector<SinhVien> docSinhVien(const string &tenFile) {
    vector<SinhVien> danhSach;
    ifstream file(tenFile, ios::binary);
    if (!file) {
        cout << "Không thể mở file để đọc dữ liệu!\n";
        return danhSach;
    }
    SinhVien sv;
    while (file.read(reinterpret_cast<char*>(&sv), sizeof(SinhVien))) {
        danhSach.push_back(sv);
    }
    file.close();
    return danhSach;
}

void inDanhSach(const vector<SinhVien> &danhSach) {
    cout << "\nDanh sách sinh viên:\n";
    cout << left << setw(10) << "Mã số" << setw(25) << "Họ và đệm" << setw(15) << "Tên"
         << setw(10) << "Ngày sinh" << setw(10) << "Phái" << setw(10) << "Điểm TB" << endl;
    cout << string(80, '-') << endl;
    for (const auto &sv : danhSach) {
        cout << left << setw(10) << sv.maSo << setw(25) << sv.hoVaDem << setw(15) << sv.ten
             << setw(2) << sv.ngaySinh << "/" << setw(2) << sv.thangSinh << "/" << setw(4)
<< sv.namSinh
             << setw(10) << sv.phai << setw(10) << sv.diemTB << endl;
    }
}

int main() {
    vector<SinhVien> danhSach;
    int soLuong = 10;

    cout << "Nhập danh sách " << soLuong << " sinh viên:\n";
```

```
    for (int i = 0; i < soLuong; ++i) {
        SinhVien sv;
        nhapSinhVien(sv, danhSach);
        danhSach.push_back(sv);
    }

    luuSinhVien("SINHVIEN.DAT", danhSach);

    // Đọc dữ liệu từ tập tin
    danhSach = docSinhVien("SINHVIEN.DAT");

    // Sắp xếp danh sách theo mã số sinh viên tăng dần
    sort(danhSach.begin(), danhSach.end(), [](const SinhVien &a, const SinhVien &b) {
        return a.maSo < b.maSo;
    });

    // In danh sách đã sắp xếp
    inDanhSach(danhSach);

    return 0;
}
```

9c,d.

Ý niệm: Nhập thông tin cho một sinh viên bao gồm mã số, họ đệm, tên, ngày, tháng, năm sinh, phái và điểm trung bình sau đó sử dụng thuật toán Merge Sort để chia thông tin sinh viên ra thành các mảng rồi hợp lại và sắp xếp theo yêu cầu.

Ví dụ với bài trên:

Kết quả cho ra:

```
SVDTB.IDX
101 Nguyen Van A 1 1 2000 Nam 9.00
102 Le Thi B 2 2 2002 Nu 8.00
103 Tran Van C 3 5 2001 Nam 7.50
SVMASO.IDX
101 Nguyen Van A 1 1 2000 Nam 9.00
102 Le Thi B 2 2 2002 Nu 8.00
103 Tran Van C 3 5 2001 Nam 7.50
```

```
SVTH.IDX
101 Nguyen Van A 1 1 2000 Nam 9.00
102 Le Thi B 2 2 2002 Nu 8.00
```

```
    103 Tran Van C 3 5 2001 Nam 7.50
```

Cài đặt thuật toán:

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;
//Cấu trúc sinh viên
struct SinhVien {
    int MaSo;
    string HoVaDem;
    string Ten;
    int NgaySinh;
    int ThangSinh;
    int NamSinh;
    string Phai;
    float DiemTrungBinh;
};
// Ghi thông tin sinh viên vào tập tin
void GhiVaoFile(SinhVien ds[], int n, const string& TenFile) {
    ofstream file(TenFile);
    if (!file) {
        cerr << "Khong the mo file!" << endl;
        return;
    }
    for (int i = 0; i < n; i++) {
        file << ds[i].MaSo << " " << ds[i].HoVaDem << " " << ds[i].Ten << " "
             << ds[i].NgaySinh << " " << ds[i].ThangSinh << " " <<
ds[i].NamSinh
             << " " << ds[i].Phai << " " << fixed << setprecision(2) <<
ds[i].DiemTrungBinh << endl;
    }
    file.close();
}
//Hàm Merge cho Merge Sort
void Merge(SinhVien a[], int left, int mid, int right, bool (*cmp)(const
SinhVien&, const SinhVien&)) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    SinhVien* leftArr = new SinhVien[n1];
    SinhVien* rightArr = new SinhVien[n2];
    for (int i = 0; i < n1; i++)
        leftArr[i] = a[left + i];
    for (int i = 0; i < n2; i++)
        rightArr[i] = a[mid + 1 + i];
```

```
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (cmp(leftArr[i], rightArr[j])) {
            a[k] = leftArr[i];
            i++;
        } else {
            a[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        a[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        a[k] = rightArr[j];
        j++;
        k++;
    }
    delete[] leftArr;
    delete[] rightArr;
}
// Sắp xếp theo mã sinh viên
bool SapXepTheoMaSinhVien(const SinhVien& sv1, const SinhVien& sv2) {
    return sv1.MaSo < sv2.MaSo;
}
// Sắp xếp theo tên sinh viên và họ đệm
bool SapXepTheoTenSinhVien(const SinhVien& sv1, const SinhVien& sv2) {
    if (sv1.Ten == sv2.Ten) {
        return sv1.HoVaDem < sv2.HoVaDem;
    }
    return sv1.Ten < sv2.Ten;
}
// Sắp xếp theo điểm trung bình
bool SapXepTheoDiemTrungBinh(const SinhVien& sv1, const SinhVien& sv2) {
    return sv1.DiemTrungBinh > sv2.DiemTrungBinh;
}
// Sử dụng thuật toán Merge Sort
void MergeSort(SinhVien a[], int left, int right, bool (*cmp)(const SinhVien&,
const SinhVien&)) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        MergeSort(a, left, mid, cmp);
        MergeSort(a, mid + 1, right, cmp);
        Merge(a, left, mid, right, cmp);
    }
}
int main() {
```

```
    SinhVien DanhSachSinhVien[3] = {
        {101, "Nguyen Van", "A", 1, 1, 2000, "Nam", 9},
        {102, "Le Thi", "B", 2, 2, 2002, "Nu", 8},
        {103, "Tran Van", "C", 3, 5, 2001, "Nam", 7.5}
    };
    int n = 3;
    MergeSort(DanhSachSinhVien, 0, n - 1, SapXepTheoMaSinhVien);
    GhiVaoFile(DanhSachSinhVien, n, "SVMASO.IDX");
    MergeSort(DanhSachSinhVien, 0, n - 1, SapXepTheoTenSinhVien);
    GhiVaoFile(DanhSachSinhVien, n, "SVTH.IDX");
    MergeSort(DanhSachSinhVien, 0, n - 1, SapXepTheoDiemTrungBinh);
    GhiVaoFile(DanhSachSinhVien, n, "SVDTB.IDX");
    cout << "Da tao cac tap tin chi muc: SVMASO.IDX, SVTH.IDX, SVDTB.IDX" <<
endl;
    return 0;
}
```

f. Nhận xét về cách sắp xếp theo chỉ mục

1. Ưu điểm:
   - Không làm thay đổi dữ liệu gốc trong file SINHVIEN.DAT.
   - Cho phép sắp xếp theo nhiều tiêu chí khác nhau (Mã số, Tên, Điểm trung bình).
   - Tối ưu bộ nhớ và tăng tốc độ khi tìm kiếm dữ liệu.
2. Nhược điểm:
   - Cần tạo thêm các file chỉ mục, chiếm thêm dung lượng lưu trữ.
   - Nếu dữ liệu gốc thay đổi, các file chỉ mục cần được cập nhật lại.