TRƯỜNG ĐẠI HỌC SÀI GÒN

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO MÔN HỌC

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

BÀI TẬP NHÓM:

BÀI TẬP SẮP XẾP

SINH VIÊN THỰC HIỆN: Nguyễn Đỗ Huy – 3121411085

LỚP: DCT124C7

GVHD: ĐỖ NHƯ TÀI

Thành phố Hồ Chí Minh , Tháng 3 Năm 2025

MỤC LỤC

Câu hỏi

Câu 1.

Trình bày tư tưởng của các thuật toán sắp xếp?

- Thuật toán sắp xếp có thể chia thành hai loại:
  - Internal sorting: Tất cả phần tử cần sắp thứ tự phải ở trong bộ nhớ chính.
  - External sorting: Một số phần tử cần sắp thứ tự trong bộ nhớ chính, các phần tử còn lại ở bộ nhớ ngoài.
- Internal sorting có ba nhóm phương pháp:
  - Phương pháp xen vào (Insertion):
    - Sắp xếp chèn (Insertion Sort):
      - Giả sử danh sách đã được sắp xếp một phần, thuật toán sẽ lấy từng phần tử mới và chèn vào vị trí thích hợp trong phần đã sắp.
      - Cách hoạt động:
        1. Bắt đầu từ phần tử thứ hai, so sánh với các phần tử đứng trước.
        2. Dịch chuyển các phần tử lớn hơn về sau để tạo chỗ trống.
        3. Chèn phần tử hiện tại vào đúng vị trí.
      - Độ phức tạp: O(n^2) trong trường hợp xấu nhất, nhưng tốt nhất là O(n) nếu mảng đã gần như sắp xếp.
  - Phương pháp chọn (Selection):
    - Sắp xếp chọn (Selection Sort):
      - Lần lượt chọn phần tử nhỏ nhất trong danh sách chưa sắp xếp và đặt vào vị trí đúng.
      - Cách hoạt động:
        1. Duyệt qua danh sách, tìm phần tử nhỏ nhất.
        2. Hoán đổi phần tử nhỏ nhất với phần tử đầu của danh sách chưa sắp xếp.
        3. Lặp lại cho phần còn lại của danh sách.
      - Độ phức tạp: O(n^2) vì luôn phải tìm phần tử nhỏ nhất.
    - Sắp xếp đống (Heap Sort):
      - Sử dụng cấu trúc Heap (đống) để sắp xếp dữ liệu.
      - Cách hoạt động:
        1. Chuyển danh sách thành Heap (cây nhị phân).
        2. Lấy phần tử lớn nhất (gốc của Heap) và đặt vào cuối danh sách.
        3. Xây lại Heap và lặp lại quá trình cho đến khi danh sách được sắp xếp.
      - Độ phức tạp: O($n\log_2 n$).
  - Phương pháp đổi chỗ (Exchange):
    - Sắp xếp nổi bọt (Bubble Sort):

        - Lặp đi lặp lại việc so sánh và hoán đổi các cặp phần tử liên tiếp, đẩy phần tử lớn dần về cuối.
        - Cách hoạt động:
            1. Duyệt qua danh sách và hoán đổi nếu phần tử trước lớn hơn phần tử sau.
            2. Lặp lại cho đến khi không còn hoán đổi nào xảy ra.
        - Độ phức tạp: O(n^2).
    - Sắp xếp trộn (Merge Sort):
        - Chia danh sách thành hai nửa nhỏ hơn, sắp xếp từng nửa rồi trộn (merge) lại thành danh sách hoàn chỉnh.
        - Cách hoạt động:
            1. Chia danh sách thành hai nửa.
            2. Gọi đệ quy để sắp xếp hai nửa.
            3. Trộn hai nửa đã sắp xếp lại.
        - Độ phức tạp: O($n\log_2 n$).
    - Sắp xếp nhanh (Quick Sort):
        - Chọn một phần tử làm pivot và chia danh sách thành hai phần nhỏ hơn/lớn hơn pivot, sau đó sắp xếp đệ quy.
        - Cách hoạt động:
            1. Chọn một phần tử pivot.
            2. Chia danh sách thành hai phần: phần nhỏ hơn pivot và phần lớn hơn pivot.
            3. Gọi đệ quy để sắp xếp hai phần.
        - Độ phức tạp: Trung bình O($n\log_2 n$), nhưng trường hợp xấu nhất là O(n^2) nếu chọn pivot không tốt.

Câu 2.

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

- Trong các thuật toán sắp xếp, em thích nhất thuật toán sắp xếp nhanh (Quick Sort). Vì:
    - Nhanh hơn hầu hết các thuật toán khác trong thực tế, với độ phức tạp trung bình O($n\log_2 n$).
    - Cách hoạt động thông minh, chia để trị, giúp giảm số lần so sánh và hoán đổi.
    - Không cần dùng thêm bộ nhớ phụ nhiều như Merge Sort.
    - Được sử dụng rộng rãi trong thực tế, nhiều thư viện chuẩn cũng áp dụng nó.
- Còn thuật toán mà em không thích nhất là thuật toán sắp xếp nổi bọt (Bubble Sort). Vì:
    - Quá chậm, có độ phức tạp O(n^2), ngay cả khi danh sách gần như sắp xếp.

- Dù đơn giản, nhưng thực tế không hiệu quả, ít khi được sử dụng.
- Nếu cần một thuật toán dễ hiểu, thì Insertion Sort vẫn tốt hơn vì hiệu suất cao hơn trên danh sách gần 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.

- Các thuật toán sắp xếp nội:
  - Sắp xếp chèn (Insertion Sort)
    - Lần lượt chèn từng phần tử vào đúng vị trí trong danh sách đã sắp xếp.
    - Cài đặt:

```
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) { // Sắp xếp giảm dần
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

    - Nhận xét: Hiệu quả với danh sách nhỏ hoặc gần như sắp xếp.
  - Sắp xếp chọn (Selection Sort)
    - Tư tưởng: Tìm phần tử lớn nhất trong danh sách chưa sắp xếp và đưa lên đầu.
    - Cài đặt:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int maxIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] > arr[maxIdx]) // Tìm phần tử lớn nhất
                maxIdx = j;
        }
        swap(arr[i], arr[maxIdx]);
    }
}
```

    - Nhận xét: Chậm với dữ liệu lớn nhưng đơn giản.
  - Sắp xếp nổi bọt (Bubble Sort)
    - Tư tưởng: Lặp đi lặp lại việc hoán đổi cặp phần tử kề nhau nếu chưa đúng thứ tự.

- Cài đặt:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] < arr[j + 1]) { // Hoán đổi nếu chưa đúng thứ tự
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break; // Nếu không có swap nào, kết thúc sớm
    }
}
```

- Nhận xét: Không hiệu quả với dữ liệu lớn.

- Sắp xếp nhanh (Quick Sort)
  - Tư tưởng: Chọn một phần tử làm chốt (pivot), chia danh sách thành hai phần nhỏ hơn/lớn hơn pivot và sắp xếp đệ quy.
  - Cài đặt:

```
int partition(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) { // Sắp xếp giảm dần
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

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

- Nhận xét: Rất nhanh trong thực tế nhưng có thể chậm nếu chọn pivot không tốt.

- Sắp xếp trộn (Merge Sort)
  - Tư tưởng: Chia danh sách thành hai nửa, sắp xếp từng nửa rồi trộn lại.
  - Cài đặt:

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

    int* L = new int[n1]; // Cấp phát động
    int* R = new int[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++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    delete[] L; // Giải phóng bộ nhớ
    delete[] R;
}

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, tốt với dữ liệu lớn nhưng tốn bộ nhớ.
- Sắp xếp Heap (Heap Sort)
  - Tư tưởng: Dùng cấu trúc heap để tìm phần tử lớn nhất.
  - Cài đặt:

```cpp
void heapify(int arr[], int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

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

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

```
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i); // Tạo Min-Heap

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Đưa phần tử nhỏ nhất ra cuối
        heapify(arr, i, 0);   // Gọi heapify để điều chỉnh Min-Heap
    }
}
```

  - Nhận xét: Hiệu quả nhưng chậm hơn Quick Sort trong thực tế.

- Các thuật toán sắp xếp ngoại được dùng khi dữ liệu quá lớn để chứa trong RAM, phải xử lý bằng ổ đĩa, SSD hoặc thiết bị lưu trữ bên ngoài.
  - Sắp xếp đa pha (Polyphase Merge Sort): Chia dữ liệu thành nhiều tệp con nhỏ, sắp xếp từng tệp, rồi trộn lại.
  - Sắp xếp phân phối (Distribution Sort): Chia dữ liệu thành nhiều vùng, sau đó xử lý từng vùng riêng lẻ.

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?

Thuật toán sắp xếp	Ưu điểm	Nhược điểm	Cách khắc phục
Insertion Sort (Sắp xếp chèn)	- Hiệu quả với dữ liệu nhỏ hoặc gần như sắp xếp (O(n) trong trường hợp tốt nhất). - Không cần bộ nhớ bổ sung.	- Với dữ liệu ngẫu nhiên, vẫn có độ phức tạp $O(n^2)$. - Không hiệu quả với dữ liệu lớn.	- Dùng Shell Sort (phiên bản nâng cấp của Insertion Sort) để tăng tốc độ.
Selection Sort (Sắp xếp chọn)	- Không cần bộ nhớ phụ, hoạt động tốt với bộ nhớ giới hạn. - Ít số lần hoán đổi hơn Bubble Sort.	- Vẫn có độ phức tạp $O(n^2)$. - Không tốt cho dữ liệu lớn.	- Dùng Heap Sort để cải thiện hiệu suất, vì Heap Sort cũng dựa trên phương pháp chọn nhưng có độ phức tạp $O(n\log_2 n)$.
Bubble Sort (Sắp xếp nổi bọt)	- Dễ cài đặt, không cần bộ nhớ phụ. - Hoạt động tốt với dữ liệu gần như đã sắp xếp (O(n) trong trường hợp tốt nhất).	- Chậm, hiệu suất $O(n^2)$ với dữ liệu ngẫu nhiên. - Không thích hợp cho dữ liệu lớn.	- Dùng thuật toán sắp xếp nhanh hơn như Quick Sort hoặc Merge Sort. - Dùng biến cờ (flag) để phát hiện nếu

			mảng đã sắp xếp, tránh vòng lặp không cần thiết.
Quick Sort (Sắp xếp nhanh)	- Nhanh, độ phức tạp trung bình $O(n\log_2 n)$. - Hiệu suất tốt cho dữ liệu ngẫu nhiên.	- Trường hợp xấu nhất là $O(n^2)$ nếu chọn điểm pivot không tốt. - Không ổn định (không giữ nguyên thứ tự các phần tử bằng nhau).	- Dùng chọn pivot thông minh (Median-of-Three hoặc Randomized Quick Sort). - Dùng Merge Sort nếu cần tính ổn định.
Merge Sort (Sắp xếp trộn)	- Luôn có độ phức tạp $O(n\log_2 n)$, không bị ảnh hưởng bởi dữ liệu đầu vào. - Ổn định, phù hợp khi cần bảo toàn thứ tự phần tử giống nhau.	- Cần bộ nhớ phụ (tốn $O(n)$ bộ nhớ). - Không hiệu quả khi làm việc với mảng nhỏ.	- Dùng Quick Sort nếu không cần ổn định và muốn tiết kiệm bộ nhớ. - Dùng Hybrid Sorting (như Tim Sort) để tối ưu cho các mảng nhỏ.
Heap Sort (Sắp xếp vun đống)	- Độ phức tạp $O(n\log_2 n)$ trong mọi trường hợp. - Không cần bộ nhớ phụ.	- Không ổn định. - Hiệu suất thực tế kém hơn Quick Sort do thao tác trên cấu trúc cây.	- Dùng Quick Sort nếu cần tốc độ cao hơn. - Dùng Merge Sort nếu cần ổn định.

Bài tập cơ sở

Bài tập 1.

Cho dãy n số nguyên sau:

39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10

a. Hãy mô phỏng các bước sắp xếp tăng dần dãy số trên bằng các giải thuật: sắp xếp đổi chỗ trực tiếp, sắp xếp chọn trực tiếp, sắp xếp chèn trực tiếp, Sắp xếp nổi bọt.

b. Cài đặt hoàn chỉnh các giải thuật trên.

c. Chứng minh độ phức tạp của các giải thuật đã triển khai.

Bài làm

a.

Sắp xếp đổi chỗ trực tiếp

Bước	Dãy số
Ban đầu	39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10
B1	1, 8, 5, 39, 3, 6, 9, 12, 4, 7, 10
B2	1, 3, 5, 39, 8, 6, 9, 12, 4, 7, 10
B3	1, 3, 4, 39, 8, 6, 9, 12, 5, 7, 10
B4	1, 3, 4, 5, 8, 6, 9, 12, 39, 7, 10
B5	1, 3, 4, 5, 6, 8, 9, 12, 39, 7, 10
B6	1, 3, 4, 5, 6, 7, 9, 12, 39, 8, 10
B7	1, 3, 4, 5, 6, 7, 8, 12, 39, 9, 10
B8	1, 3, 4, 5, 6, 7, 8, 9, 39, 12, 10
B9	1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39

Sắp xếp chọn trực tiếp

Bước	Dãy số
Ban đầu	39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10
B1	1, 8, 5, 39, 3, 6, 9, 12, 4, 7, 10
B2	1, 3, 5, 39, 8, 6, 9, 12, 4, 7, 10
B3	1, 3, 4, 39, 8, 6, 9, 12, 5, 7, 10
B4	1, 3, 4, 5, 8, 6, 9, 12, 39, 7, 10
B5	1, 3, 4, 5, 6, 8, 9, 12, 39, 7, 10
B6	1, 3, 4, 5, 6, 7, 9, 12, 39, 8, 10
B7	1, 3, 4, 5, 6, 7, 8, 12, 39, 9, 10
B8	1, 3, 4, 5, 6, 7, 8, 9, 39, 12, 10
B9	1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39

Sắp xếp chèn trực tiếp

Bước	Dãy số
Ban đầu	39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10
B1	8, 39, 5, 1, 3, 6, 9, 12, 4, 7, 10
B2	5, 8, 39, 1, 3, 6, 9, 12, 4, 7, 10

B3	1, 5, 8, 39, 3, 6, 9, 12, 4, 7, 10
B4	1, 3, 5, 8, 39, 6, 9, 12, 4, 7, 10
B5	1, 3, 5, 6, 8, 39, 9, 12, 4, 7, 10
B6	1, 3, 5, 6, 8, 9, 39, 12, 4, 7, 10
B7	1, 3, 5, 6, 8, 9, 12, 39, 4, 7, 10
B8	1, 3, 4, 5, 6, 8, 9, 12, 39, 7, 10
B9	1, 3, 4, 5, 6, 7, 8, 9, 12, 39, 10

Sắp xếp nổi bọt

Bước	Dãy số
Ban đầu	39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10
B1	8, 5, 1, 3, 6, 9, 12, 4, 7, 10, 39
B2	5, 1, 3, 6, 8, 9, 4, 7, 10, 12, 39
B3	1, 3, 5, 6, 8, 4, 7, 9, 10, 12, 39
B4	1, 3, 5, 6, 4, 7, 8, 9, 10, 12, 39
B5	1, 3, 5, 4, 6, 7, 8, 9, 10, 12, 39
B6	1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 39

b.

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

// Hàm in mảng
void printArray(const vector<int>& arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

// 1. Sắp xếp đổi chỗ trực tiếp (Interchange Sort)
void interchangeSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                swap(arr[i], arr[j]);
            }
        }
    }
}

// 2. Sắp xếp chọn trực tiếp (Selection Sort)
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    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;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}

// 3. Sắp xếp chèn trực tiếp (Insertion Sort)
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    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;
    }
}

// 4. Sắp xếp nổi bọt (Bubble Sort)
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int main() {
    vector<int> arr = {39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10};
    vector<int> temp;

    cout << "Dãy số ban đầu: ";
    printArray(arr);

    temp = arr;
    interchangeSort(temp);
    cout << "Sau khi sắp xếp đổi chỗ trực tiếp: ";
    printArray(temp);
```

```
    temp = arr;
    selectionSort(temp);
    cout << "Sau khi sắp xếp chọn trực tiếp: ";
    printArray(temp);

    temp = arr;
    insertionSort(temp);
    cout << "Sau khi sắp xếp chèn trực tiếp: ";
    printArray(temp);

    temp = arr;
    bubbleSort(temp);
    cout << "Sau khi sắp xếp nổi bọt: ";
    printArray(temp);

    return 0;
}
```

c.

Sắp xếp đổi chỗ trực tiếp:

- So sánh mọi cặp phần tử ⇒ $O(n^2)$.

Sắp xếp chọn trực tiếp:

- Tìm phần tử nhỏ nhất trong dãy chưa sắp xếp ⇒ $O(n^2)$.

Sắp xếp chèn trực tiếp:

- Trường hợp xấu nhất phải chèn tất cả các phần tử ⇒ $O(n^2)$.
- Trường hợp tốt nhất (dãy đã sắp xếp) ⇒ $O(n)$.

Sắp xếp nổi bọt:

- Luôn cần quét toàn bộ dãy để đảm bảo sắp xếp đúng ⇒ $O(n^2)$.

Bài tập 2.

Cho dãy n số nguyên sau:

8, 5, 1, 3, 6, 9, 12, 4, 7, 10

a. Hãy mô phỏng các bước sắp xếp tăng dần dãy số trên bằng các giải thuật: sắp xếp nhanh (Quick Sort); Sắp xếp trộn trực tiếp (Merge Sort); Sắp xếp cây (Heap Sort).

b. Cài đặt hoàn chỉnh các giải thuật trên.

c. Chứng minh độ phức tạp của các giải thuật đã triển khai.

Bài làm

a.

Sắp xếp nhanh

Với dãy [8, 5, 1, 3, 6, 9, 12, 4, 7, 10]:

1. Chọn pivot = 6, chia thành hai phần:
   - Nhỏ hơn pivot: [5, 1, 3, 4]
   - Lớn hơn pivot: [8, 9, 12, 7, 10]
2. Sắp xếp phần trái [5, 1, 3, 4] với pivot = 3:
   - Nhỏ hơn pivot: [1]
   - Lớn hơn pivot: [5, 4]
   - Kết quả: [1, 3, 4, 5]
3. Sắp xếp phần phải [8, 9, 12, 7, 10] với pivot = 9:
   - Nhỏ hơn pivot: [8, 7]
   - Lớn hơn pivot: [12, 10]
   - Sắp xếp tiếp: [7, 8, 9, 10, 12]

⇨ Kết quả cuối cùng: [1, 3, 4, 5, 6, 7, 8, 9, 10, 12]

Sắp xếp trộn trực tiếp

Với dãy [8, 5, 1, 3, 6, 9, 12, 4, 7, 10]:

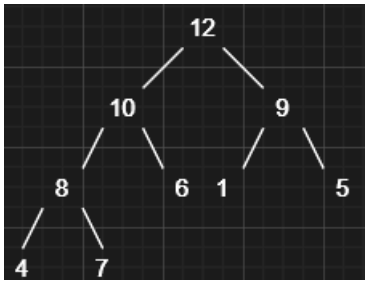
1. Chia thành hai nửa:
   - Trái: [8, 5, 1, 3, 6]
   - Phải: [9, 12, 4, 7, 10]
2. Sắp xếp nửa trái [8, 5, 1, 3, 6]:
   - Chia tiếp: [8, 5] và [1, 3, 6]
   - Sắp xếp từng phần và trộn lại: [1, 3, 5, 6, 8]
3. Sắp xếp nửa phải [9, 12, 4, 7, 10]:
   - Chia tiếp: [9, 12] và [4, 7, 10]
   - Sắp xếp từng phần và trộn lại: [4, 7, 9, 10, 12]
4. Trộn hai nửa:
   - [1, 3, 5, 6, 8] và [4, 7, 9, 10, 12]

⇨ Kết quả cuối cùng: [1, 3, 4, 5, 6, 7, 8, 9, 10, 12]

Sắp xếp cây

Với dãy [8, 5, 1, 3, 6, 9, 12, 4, 7, 10]:

1. Xây dựng max-heap:



2. Hoán đổi `12` với `7`, giảm heap size và điều chỉnh heap.
3. Tiếp tục hoán đổi phần tử lớn nhất với phần tử cuối, sắp xếp lại heap.

⇨ Kết quả cuối cùng: [1, 3, 4, 5, 6, 7, 8, 9, 10, 12].

b.

```cpp
#include <iostream>
#include <vector>

using namespace std;

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

// **1. Quick Sort**
int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[high];  // Chọn phần tử cuối làm pivot
    int i = low - 1; // Vị trí của phần tử nhỏ hơn pivot

    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 = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
```

```cpp
}

// **2. Merge Sort**
void merge(vector<int> &arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<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++];
        else arr[k++] = R[j++];
    }

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

void mergeSort(vector<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);
    }
}

// **3. Heap Sort**
void heapify(vector<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) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int> &arr) {
```

```
    int n = arr.size();

    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 in mảng**
void printArray(vector<int> &arr) {
    for (int num : arr) cout << num << " ";
    cout << endl;
}

// **Chương trình chính**
int main() {
    vector<int> arr = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

    // Quick Sort
    vector<int> quickArr = arr;
    quickSort(quickArr, 0, quickArr.size() - 1);
    cout << "Quick Sort: ";
    printArray(quickArr);

    // Merge Sort
    vector<int> mergeArr = arr;
    mergeSort(mergeArr, 0, mergeArr.size() - 1);
    cout << "Merge Sort: ";
    printArray(mergeArr);

    // Heap Sort
    vector<int> heapArr = arr;
    heapSort(heapArr);
    cout << "Heap Sort: ";
    printArray(heapArr);

    return 0;
}
```

c.

Quick Sort

- Trường hợp tốt nhất chọn pivot chia mảng thành hai phần gần bằng nhau, số bước gọi đệ quy giảm dần theo cấp số nhân. => $O(n\log_2 n)$.
- Trường hợp xấu nhất khi pivot luôn là phần tử nhỏ nhất hoặc lớn nhất, số lần gọi đệ quy là $O(n)$. => $O(n^2)$.

Merge Sort

- Mảng luôn được chia thành hai phần bằng nhau, không phụ thuộc vào dữ liệu đầu vào. => $O(n\log_2 n)$.

Heap Sort

- Tạo max-heap:
  - Mất O(n) thời gian để xây dựng heap.
- Heapify và trích xuất phần tử lớn nhất:
  - Mỗi lần lấy phần tử lớn nhất mất $O(n\log_2 n)$.
  - Thực hiện n lần, nên tổng cộng $O(n\log_2 n)$.

Bài tập 3.

Sử dụng hàm random trong C để tạo ra một dãy M có n số nguyên. Vận dụng các thuật toán sắp xếp để sắp xếp các phần tử của mảng M theo thứ tự tăng dần về mặt giá trị. Với cùng một dữ liệu như nhau, làm thế nào để biết thời gian thực hiện các thuật toán (đếm số lần so sánh, đổi chỗ)? Có nhận xét gì đối với các thuật toán sắp xếp này? Thực nghiệm mỗi trường hợp t lần với kích cỡ mảng n=10, 100, 200, …, 10000.

Bài làm

- Ta có:
  - Sinh mảng ngẫu nhiên (generateRandomArray).
  - Cài đặt thuật toán Quick Sort, Merge Sort và Heap Sort.
  - Đếm số lần so sánh và đổi chỗ bằng biến quickComparisons, heapSwaps...
  - Đo thời gian thực thi bằng chrono.
  - Chạy thử nghiệm với các giá trị n = 10, 100, ..., 10000.
- Cài đặt:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;
using namespace chrono;

// Biến đếm số lần so sánh và đổi chỗ
long long quickComparisons = 0, quickSwaps = 0;
long long mergeComparisons = 0;
long long heapComparisons = 0, heapSwaps = 0;
```

```cpp
// Hàm sinh mảng ngẫu nhiên
vector<int> generateRandomArray(int n) {
    vector<int> arr(n);
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;  // Số ngẫu nhiên từ 0 đến 9999
    }
    return arr;
}

// Hàm hoán đổi
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// **1. Quick Sort**
int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        quickComparisons++;
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
            quickSwaps++;
        }
    }
    swap(arr[i + 1], arr[high]);
    quickSwaps++;
    return i + 1;
}

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

// **2. Merge Sort**
void merge(vector<int> &arr, int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    vector<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) {
        mergeComparisons++;
        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 mergeSort(vector<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);
    }
}

// **3. Heap Sort**
void heapify(vector<int> &arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

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

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

void heapSort(vector<int> &arr) {
    int n = arr.size();
    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]);
        heapSwaps++;
        heapify(arr, i, 0);
```

```
    }
}

// **Hàm đo thời gian chạy thuật toán**
template <typename Func>
double measureTime(Func sortingFunction, vector<int> &arr) {
    auto start = high_resolution_clock::now();
    sortingFunction(arr);
    auto stop = high_resolution_clock::now();
    return duration<double, milli>(stop - start).count();
}

// **Chương trình chính**
int main() {
    srand(time(0));

    vector<int> sizes = {10, 100, 200, 500, 1000, 5000, 10000};

    cout << "n\tQuickSort (ms)\tQuickSort Comparisons\tQuickSort Swaps\t"
         << "MergeSort (ms)\tMergeSort Comparisons\t"
         << "HeapSort (ms)\tHeapSort Comparisons\tHeapSort Swaps\n";

    for (int n : sizes) {
        vector<int> arr = generateRandomArray(n);

        // Quick Sort
        quickComparisons = 0, quickSwaps = 0;
        vector<int> quickArr = arr;
        double quickTime = measureTime([&](vector<int> &a) { quickSort(a, 0,
a.size() - 1); }, quickArr);

        // Merge Sort
        mergeComparisons = 0;
        vector<int> mergeArr = arr;
        double mergeTime = measureTime([&](vector<int> &a) { mergeSort(a, 0,
a.size() - 1); }, mergeArr);

        // Heap Sort
        heapComparisons = 0, heapSwaps = 0;
        vector<int> heapArr = arr;
        double heapTime = measureTime([&](vector<int> &a) { heapSort(a); },
heapArr);

        // Xuất kết quả
        cout << n << "\t"
             << quickTime << "\t" << quickComparisons << "\t" << quickSwaps << "\t"
             << mergeTime << "\t" << mergeComparisons << "\t"
```

```
            << heapTime << "\t" << heapComparisons << "\t" << heapSwaps << "\n";
    }

    return 0;
}
```

- Nhận xét các thuật toán sắp xếp:
  - Quick Sort

n	Time (ms)	Comparisons	Swaps
10	0	20	18
100	0	767	500
200	0	2098	1109
500	0	4443	2629
1000	1.507	10406	5638
5000	0.999	72729	35526
10000	0.508	159300	86405

    - Nhận xét thuật toán Quick Sort
      - Khi n < 500, thời gian chạy gần như bằng 0 ms do kích thước nhỏ.
      - Khi n = 1000, thời gian bắt đầu tăng (~1.5 ms).
      - Khi n = 10000, số lần so sánh lên đến 159300, số lần hoán đổi 86405, nhưng thời gian chỉ khoảng 0.5 ms, cho thấy Quick Sort vẫn khá tối ưu.
      - So với Merge Sort và Heap Sort, Quick Sort có số lần hoán đổi lớn nhất, điều này làm tăng chi phí bộ nhớ cache.
    - Độ phức tạp
      - Trung bình $O(n\log_2 n)$, nhưng nếu chọn pivot không tốt, có thể lên $O(n^2)$.
  - Merge Sort

n	Time (ms)	Comparisons
10	0	25
100	0	540
200	0	1274
500	0	3847
1000	0	8726
5000	2.01	55173
10000	5.718	120440

    - Nhận xét thuật toán Merge Sort
      - Merge Sort có số lần so sánh thấp hơn Quick Sort, đặc biệt với n ≥ 5000.
      - Khi n = 10000, số lần so sánh 120440, thấp hơn Quick Sort (159300).

        - Tuy nhiên, thời gian chạy dài hơn (5.7 ms so với 0.5 ms của Quick Sort).
        - Điều này do Merge Sort sử dụng bộ nhớ bổ sung (O(n)), gây ảnh hưởng đến tốc độ.
    - Độ phức tạp
        - O(nlog$_2$n trong mọi trường hợp, nhưng tốn O(n) bộ nhớ phụ.
- Heap Sort

n	Time (ms)	Comparisons	Swaps
10	0	60	25
100	0	1262	581
200	0	2880	1340
500	0	8624	4062
1000	0	19116	9058
5000	2.105	119236	57118
10000	3.491	258346	124173

    - Nhận xét thuật toán Heap Sort
        - Heap Sort có số lần so sánh nhiều nhất trong tất cả các thuật toán.
        - Khi n = 10000, số lần so sánh 258346, gấp 2.15 lần Merge Sort (120440) và 1.62 lần Quick Sort (159300).
        - Số lần hoán đổi cao nhất (124173 swaps).
        - Thời gian chạy cao hơn Quick Sort (3.49 ms so với 0.5 ms).
    - Độ phức tạp
        - O(nlog$_2$n) nhưng với số lượng swap cao, làm ảnh hưởng đến hiệu suất bộ nhớ cache.
        - Không cần bộ nhớ phụ như Merge Sort.

Bài tập 4.

Hãy đo thời gian thi hành của mỗi giải thuật sắp xếp đã học trên cùng một bộ dữ liệu chứa khoảng 30.000 số nguyên ngẫu nhiên và trên cùng một máy tính của bạn.

• Dữ liệu đầu vào: Được lưu trên file văn bản.

• Dữ liệu đầu ra: Kết quả sắp xếp được lưu trên file văn bản.

• Các thuật toán sắp xếp cần đo thời gian: Sắp xếp đổi chỗ trực tiếp (Interchange Sort); Sắp xếp chọn trực tiếp (Selection Sort); Sắp xếp chèn trực tiếp (Insertion Sort); Sắp xếp nổi bọt (Bubble Sort); Sắp xếp nhanh (Quick Sort); Sắp xếp trộn (Merge Sort); Sắp xếp cây (Heap Sort)

Yêu cầu thực hiện:

• Đọc dữ liệu từ file văn bản.

• Đo thời gian chạy từng thuật toán trên cùng bộ dữ liệu.

• Lưu kết quả sắp xếp và thời gian chạy vào file văn bản.

• So sánh hiệu suất của các thuật toán sắp xếp.

Bài làm

- Tạo file dữ liệu input.txt chứa 30.000 số ngẫu nhiên

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>

using namespace std;

void generateRandomNumbers(const string &filename, int n) {
    ofstream file(filename);
    srand(time(0));

    for (int i = 0; i < n; i++) {
        file << rand() % 100000 << " "; // Số ngẫu nhiên từ 0 đến 99999
    }

    file.close();
}

int main() {
    generateRandomNumbers("input.txt", 30000);
    cout << "Đã tạo file input.txt chứa 30.000 số nguyên ngẫu nhiên." << endl;
    return 0;
}
```

- Cài đặt code để đọc file, sắp xếp và lưu kết quả

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <ctime>

using namespace std;

// Hàm đọc dữ liệu từ file
vector<int> readFromFile(const string &filename) {
    ifstream file(filename);
    vector<int> arr;
    int num;

    while (file >> num) {
```

```
        arr.push_back(num);
    }

    file.close();
    return arr;
}

// Hàm ghi dữ liệu vào file
void writeToFile(const string &filename, const vector<int> &arr, double timeTaken)
{
    ofstream file(filename);

    file << "Thời gian thực hiện: " << timeTaken << " ms\n";
    file << "Dãy số đã sắp xếp:\n";

    for (int num : arr) {
        file << num << " ";
    }

    file.close();
}

// 1 Interchange Sort (Sắp xếp đổi chỗ trực tiếp)
void interchangeSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                swap(arr[i], arr[j]);
            }
        }
    }
}

// 2 Selection Sort (Sắp xếp chọn trực tiếp)
void selectionSort(vector<int> &arr) {
    int n = arr.size();
    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;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

```cpp
// 3 Insertion Sort (Sắp xếp chèn trực tiếp)
void insertionSort(vector<int> &arr) {
    int n = arr.size();
    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;
    }
}

// 4 Bubble Sort (Sắp xếp nổi bọt)
void bubbleSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// 5 Quick Sort
void quickSort(vector<int> &arr, int left, int right) {
    if (left >= right) return;

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

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

// 6 Merge Sort
void merge(vector<int> &arr, int left, int mid, int right) {
```

```
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);

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

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

void mergeSort(vector<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);
}

// 7 Heap Sort
void heapify(vector<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) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int> &arr) {
    int n = arr.size();
    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);
    }
}
```

```cpp
// Hàm đo thời gian thực hiện
void measureSortTime(void (*sortFunction)(vector<int>&), vector<int> arr, const
string &filename) {
    clock_t start = clock();
    sortFunction(arr);
    clock_t end = clock();
    double timeTaken = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    writeToFile(filename, arr, timeTaken);
}

int main() {
    vector<int> arr = readFromFile("input.txt");

    measureSortTime(interchangeSort, arr, "interchange_sort.txt");
    measureSortTime(selectionSort, arr, "selection_sort.txt");
    measureSortTime(insertionSort, arr, "insertion_sort.txt");
    measureSortTime(bubbleSort, arr, "bubble_sort.txt");
    measureSortTime(heapSort, arr, "heap_sort.txt");

    return 0;
}
```

- So sánh hiệu suất của các thuật toán sắp xếp.
    - Bảng thời gian thực hiện của từng thuật toán

Thuật toán	Thời gian thực hiện (ms)
Heap Sort	10
Selection Sort	1799
Insertion Sort	1275
Bubble Sort	4102
Interchange Sort	4399

- Phân tích hiệu suất
    - Heap Sort - (Nhanh nhất: 10 ms)
        - Hiệu suất: Rất nhanh, chỉ mất 10 ms.
        - Lý do:
            - Chuyển đổi mảng thành max-heap, sau đó trích xuất phần tử lớn nhất và điều chỉnh lại heap.
            - Mặc dù có nhiều lần hoán đổi, nhưng chỉ mất O(n log n) thời gian, giúp tối ưu hóa tốc độ.
        - Ưu điểm:
            - Ổn định với dữ liệu lớn.
            - Không cần bộ nhớ bổ sung.
        - Nhược điểm:
            - Không phải là thuật toán ổn định (có thể thay đổi vị trí của các phần tử có giá trị bằng nhau).

- Selection Sort - (1799 ms)
  - Hiệu suất: Chậm hơn Heap Sort ~180 lần.
  - Lý do:
    - Mỗi vòng lặp tìm phần tử nhỏ nhất và đưa về đầu mảng.
    - Cần $O(n^2)$ so sánh, không tối ưu cho dữ liệu lớn.
  - Ưu điểm:
    - Ít số lần hoán đổi hơn so với Bubble/Interchange Sort.
  - Nhược điểm:
    - Rất chậm với dữ liệu lớn.
- Insertion Sort - (1275 ms)
  - Hiệu suất: Nhanh hơn Selection Sort & Bubble Sort, nhưng vẫn chậm hơn Heap Sort ~127 lần.
  - Lý do:
    - Nếu mảng gần như đã sắp xếp, thuật toán hoạt động rất nhanh $O(n)$.
    - Nếu mảng ngẫu nhiên, hiệu suất vẫn $O(n^2)$.
  - Ưu điểm:
    - Tốt với dữ liệu gần như đã sắp xếp.
  - Nhược điểm:
    - Vẫn chậm với dữ liệu ngẫu nhiên lớn.
- Bubble Sort - (4102 ms)
  - Hiệu suất: Rất chậm, gấp 410 lần Heap Sort.
  - Lý do:
    - Luôn so sánh từng cặp và đổi chỗ, nên số lần hoán đổi rất lớn.
    - $O(n^2)$ trong mọi trường hợp.
  - Ưu điểm:
    - Dễ hiểu, dễ triển khai.
  - Nhược điểm:
    - Tệ nhất trong các thuật toán phổ biến, không nên dùng cho dữ liệu lớn.
- Interchange Sort - (4399 ms)
  - Hiệu suất: Chậm nhất, lâu hơn Heap Sort 439 lần.
  - Lý do:
    - Duyệt hết tất cả các cặp số để so sánh và đổi chỗ nếu cần.
    - $O(n^2)$ trong mọi trường hợp.
  - Ưu điểm:
    - Cực kỳ dễ hiểu.
  - Nhược điểm:
    - Không phù hợp với dữ liệu lớn.

Bài tập ứng dụng

Bài tập 1.

Cho dãy n số nguyên: $a_0, a_1, ..., a_{n-1}$

a. Hãy cho biết vị trí của k phần tử có giá trị lớn nhất trong dãy.

b. Sắp xếp các phần tử tăng dần theo tổng các chữ số của từng phần tử.

c. Hãy xóa tất cả các số nguyên tố có trong dãy.

Bài làm

a. Tìm vị trí k phần tử lớn nhất
   - Duyệt qua dãy, lưu cặp (giá trị, vị trí).
   - Sắp xếp giảm dần theo giá trị.
   - Chọn k phần tử đầu tiên và lấy vị trí của chúng.

b. Sắp xếp theo tổng chữ số
   - Viết hàm sumOfDigits(num) để tính tổng chữ số.
   - Dùng sort() với tiêu chí sắp xếp dựa trên tổng chữ số.
   - Nếu tổng chữ số bằng nhau, so sánh giá trị gốc để giữ thứ tự tăng dần.

c. Xóa số nguyên tố
   - Dùng isPrime(num) để kiểm tra số nguyên tố.
   - Sử dụng remove_if() để xóa số nguyên tố khỏi mảng.

Cài đặt

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

using namespace std;

// Hàm tính tổng các chữ số của một số
int sumOfDigits(int num) {
    int sum = 0;
    while (num > 0) {
        sum += num % 10;
        num /= 10;
    }
    return sum;
}

// Hàm kiểm tra số nguyên tố
bool isPrime(int num) {
    if (num < 2) return false;
```

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

// (a) Tìm vị trí của k phần tử lớn nhất
vector<int> findTopKElements(const vector<int>& arr, int k) {
    vector<pair<int, int>> indexedArr;
    for (int i = 0; i < arr.size(); i++) {
        indexedArr.push_back({arr[i], i});
    }

    sort(indexedArr.rbegin(), indexedArr.rend()); // Sắp xếp giảm dần
    vector<int> positions;
    for (int i = 0; i < k && i < indexedArr.size(); i++) {
        positions.push_back(indexedArr[i].second);
    }

    return positions;
}

// (b) Sắp xếp tăng dần theo tổng các chữ số
void sortByDigitSum(vector<int>& arr) {
    sort(arr.begin(), arr.end(), [](int a, int b) {
        int sumA = sumOfDigits(a);
        int sumB = sumOfDigits(b);
        return sumA < sumB || (sumA == sumB && a < b);
    });
}

// (c) Xóa tất cả số nguyên tố
void removePrimes(vector<int>& arr) {
    arr.erase(remove_if(arr.begin(), arr.end(), [](int num) {
        return isPrime(num);
    }), arr.end());
}

// Hàm hiển thị mảng
void printArray(const vector<int>& arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
```

```
    vector<int> arr = {29, 18, 12, 47, 56, 23, 34, 89, 77, 45};
    int k = 3;

    // (a) Tìm vị trí k phần tử lớn nhất
    vector<int> topKPositions = findTopKElements(arr, k);
    cout << "Vị trí của " << k << " phần tử lớn nhất: ";
    for (int pos : topKPositions) {
        cout << pos << " ";
    }
    cout << endl;

    // (b) Sắp xếp theo tổng chữ số
    vector<int> sortedArr = arr;
    sortByDigitSum(sortedArr);
    cout << "Sắp xếp theo tổng chữ số: ";
    printArray(sortedArr);

    // (c) Xóa số nguyên tố
    vector<int> filteredArr = arr;
    removePrimes(filteredArr);
    cout << "Dãy sau khi xóa số nguyên tố: ";
    printArray(filteredArr);

    return 0;
}
```

Bài tập 2.

Thông tin về mỗi số hạng của một dãy thức bậc n bao gồm: Hệ số – là một số thực, Bậc – là một số nguyên có giá trị từ 0 đến 100.

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

b. Với cấu trúc dữ liệu đã được định nghĩa, hãy vận dụng một thuật toán sắp xếp và cài đặt chương trình thực hiện việc sắp xếp các số hạng trong dãy theo thứ tự tăng dần của các bậc.

Bài làm

a. Định nghĩa cấu trúc dữ liệu
   - Dùng struct Term để lưu hệ số (coefficient) và bậc (exponent).
   - Dùng vector<Term> để lưu danh sách số hạng.

b. Sắp xếp dãy theo bậc
   - Dùng sort() trong STL, với điều kiện so sánh bậc (exponent).
   - In danh sách trước và sau khi sắp xếp để kiểm tra kết quả.

Cài đặt

```
#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>

using namespace std;

// (a) Định nghĩa cấu trúc dữ liệu
struct Term {
    double coefficient;  // Hệ số (số thực)
    int exponent;        // Bậc (số nguyên từ 0 đến 100)
};

// Hàm hiển thị danh sách số hạng
void printPolynomial(const vector<Term>& poly) {
    for (const auto& term : poly) {
        cout << term.coefficient << "x^" << term.exponent << "      ";
    }
    cout << endl;
}

// (b) Sắp xếp theo bậc tăng dần
void sortPolynomial(vector<Term>& poly) {
    sort(poly.begin(), poly.end(), [](const Term& a, const Term& b) {
        return a.exponent < b.exponent;
    });
}

int main() {
    // Khởi tạo dãy thức bậc n
    vector<Term> polynomial = {
        {3.5, 2}, {1.2, 5}, {4.0, 3}, {2.1, 0}, {5.6, 1}
    };

    cout << "Dãy thức trước khi sắp xếp:\n";
    printPolynomial(polynomial);

    // Sắp xếp theo bậc tăng dần
    sortPolynomial(polynomial);

    cout << "Dãy thức sau khi sắp xếp theo bậc tăng dần:\n";
    printPolynomial(polynomial);

    return 0;
}
```

Bài tập 3.

Thông tin về các phòng thi tại một hội đồng thi bao gồm: Số phòng – là một số nguyên có giá trị từ 1 đến 200, Nhà – là một chữ cái in hoa từ A → Z, Khả năng chứa – là một số nguyên có giá trị từ 10 → 250.

a. Hãy định nghĩa cấu trúc dữ liệu để lưu trữ các phòng thi này trong bộ nhớ trong của máy tính.

b. Với cấu trúc dữ liệu đã được định nghĩa, vận dụng các thuật toán sắp xếp và cài đặt chương trình thực hiện việc các công việc sau:

• Sắp xếp và in ra màn hình danh sách các phòng thi theo thứ tự giảm dần về Khả năng chứa.

• Sắp xếp và in ra màn hình danh sách các phòng thi theo thứ tự tăng dần theo Nhà (Từ A → Z), các phòng cùng một nhà thì sắp xếp theo thứ tự tăng dần theo Số phòng.

• Sắp xếp và in ra màn hình danh sách các phòng thi theo thứ tự tăng dần theo Nhà (Từ A → Z), các phòng cùng một nhà thì sắp xếp theo thứ tự giảm dần theo Khả năng chứa.

Bài làm

a. Định nghĩa cấu trúc Room
   - Dùng struct Room để lưu số phòng, nhà và khả năng chứa.
   - Dùng vector<Room> để lưu danh sách phòng.

b. Cài đặt các thuật toán sắp xếp
   - Sắp xếp giảm dần theo khả năng chứa (sortByCapacity()).
   - Sắp xếp theo Nhà (A → Z), cùng nhà thì theo số phòng tăng dần (sortByBuildingAndRoom()).
   - Sắp xếp theo Nhà (A → Z), cùng nhà thì theo khả năng chứa giảm dần (sortByBuildingAndCapacity()).

Cài đặt

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

using namespace std;

// (a) Định nghĩa cấu trúc dữ liệu phòng thi
struct Room {
    int roomNumber;  // Số phòng (1 → 200)
    char building;   // Nhà (A → Z)
    int capacity;    // Khả năng chứa (10 → 250)
};

// Hàm hiển thị danh sách phòng thi
```

```cpp
void printRooms(const vector<Room>& rooms) {
    for (const auto& room : rooms) {
        cout << "Nhà " << room.building << " - Phòng " << room.roomNumber
             << " - Sức chứa: " << room.capacity << endl;
    }
    cout << endl;
}

// (b.1) Sắp xếp giảm dần theo khả năng chứa
void sortByCapacity(vector<Room>& rooms) {
    sort(rooms.begin(), rooms.end(), [](const Room& a, const Room& b) {
        return a.capacity > b.capacity;  // Sắp xếp giảm dần
    });
}

// (b.2) Sắp xếp tăng dần theo Nhà, nếu cùng Nhà thì tăng theo Số phòng
void sortByBuildingAndRoom(vector<Room>& rooms) {
    sort(rooms.begin(), rooms.end(), [](const Room& a, const Room& b) {
        if (a.building == b.building)
            return a.roomNumber < b.roomNumber;  // Số phòng tăng dần
        return a.building < b.building;  // Nhà tăng dần từ A → Z
    });
}

// (b.3) Sắp xếp tăng dần theo Nhà, nếu cùng Nhà thì giảm dần theo Khả năng chứa
void sortByBuildingAndCapacity(vector<Room>& rooms) {
    sort(rooms.begin(), rooms.end(), [](const Room& a, const Room& b) {
        if (a.building == b.building)
            return a.capacity > b.capacity;  // Khả năng chứa giảm dần
        return a.building < b.building;  // Nhà tăng dần từ A → Z
    });
}

int main() {
    // Danh sách phòng thi mẫu
    vector<Room> rooms = {
        {101, 'B', 100}, {102, 'A', 150}, {103, 'B', 90}, {201, 'A', 200},
        {202, 'C', 120}, {301, 'C', 250}, {203, 'C', 180}, {204, 'A', 80}
    };

    cout << "Danh sách phòng thi ban đầu:\n";
    printRooms(rooms);

    // (b.1) Sắp xếp theo khả năng chứa giảm dần
    sortByCapacity(rooms);
    cout << "Sắp xếp giảm dần theo khả năng chứa:\n";
    printRooms(rooms);
```

```
    // (b.2) Sắp xếp theo Nhà (tăng dần), cùng nhà thì theo Số phòng (tăng dần)
    sortByBuildingAndRoom(rooms);
    cout << "Sắp xếp theo Nhà (A → Z), cùng nhà thì theo Số phòng tăng dần:\n";
    printRooms(rooms);

    // (b.3) Sắp xếp theo Nhà (A → Z), cùng nhà thì theo Khả năng chứa giảm dần
    sortByBuildingAndCapacity(rooms);
    cout << "Sắp xếp theo Nhà (A → Z), cùng nhà thì theo Khả năng chứa giảm
dần:\n";
    printRooms(rooms);

    return 0;
}
```

Bài tập 4.

Cho ma trận hai chiều m dòng n cột, các phần tử là các số nguyên dương.

a. Tìm số nguyên tố lớn nhất trong ma trận.

b. Tìm những dòng của ma trận có chứa giá trị nguyên tố.

c. Tìm những dòng của ma trận chỉ chứa các số nguyên tố.

Bài làm

a. Tìm số nguyên tố lớn nhất
   - Duyệt từng phần tử trong ma trận.
   - Nếu phần tử là số nguyên tố và lớn hơn giá trị lớn nhất hiện tại, cập nhật giá trị lớn nhất.

b. Tìm các dòng có ít nhất một số nguyên tố
   - Duyệt từng dòng, nếu thấy ít nhất một số nguyên tố thì lưu lại chỉ số dòng đó.

c. Tìm các dòng chỉ chứa toàn số nguyên tố
   - Duyệt từng dòng, nếu tất cả phần tử trong dòng đều là số nguyên tố thì lưu lại chỉ số dòng đó.

Cài đặt

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

// Hàm kiểm tra số nguyên tố
bool isPrime(int num) {
    if (num < 2) return false;
```

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

// (a) Tìm số nguyên tố lớn nhất trong ma trận
int findLargestPrime(const vector<vector<int>>& matrix) {
    int maxPrime = -1;
    for (const auto& row : matrix) {
        for (int num : row) {
            if (isPrime(num) && num > maxPrime) {
                maxPrime = num;
            }
        }
    }
    return maxPrime;
}

// (b) Tìm các dòng có ít nhất một số nguyên tố
vector<int> findRowsWithPrime(const vector<vector<int>>& matrix) {
    vector<int> rows;
    for (int i = 0; i < matrix.size(); i++) {
        for (int num : matrix[i]) {
            if (isPrime(num)) {
                rows.push_back(i);
                break;  // Chỉ cần tìm thấy 1 số nguyên tố là đủ
            }
        }
    }
    return rows;
}

// (c) Tìm các dòng chỉ chứa số nguyên tố
vector<int> findRowsWithOnlyPrimes(const vector<vector<int>>& matrix) {
    vector<int> rows;
    for (int i = 0; i < matrix.size(); i++) {
        bool allPrime = true;
        for (int num : matrix[i]) {
            if (!isPrime(num)) {
                allPrime = false;
                break;
            }
        }
        if (allPrime) {
            rows.push_back(i);
        }
```

```
    }
    return rows;
}

// Hàm in ma trận
void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int num : row) {
            cout << num << "\t";
        }
        cout << endl;
    }
}

// Hàm in danh sách các dòng
void printRows(const vector<int>& rows, const string& message) {
    cout << message;
    if (rows.empty()) {
        cout << "Không có dòng nào." << endl;
    } else {
        for (int row : rows) {
            cout << "Dòng " << row << " ";
        }
        cout << endl;
    }
}

int main() {
    // Ma trận mẫu
    vector<vector<int>> matrix = {
        {4, 6, 7, 8},
        {10, 17, 19, 22},
        {29, 31, 37, 41},
        {9, 15, 21, 25}
    };

    cout << "Ma trận ban đầu:\n";
    printMatrix(matrix);

    // (a) Số nguyên tố lớn nhất trong ma trận
    int maxPrime = findLargestPrime(matrix);
    if (maxPrime != -1) {
        cout << "Số nguyên tố lớn nhất trong ma trận: " << maxPrime << endl;
    } else {
        cout << "Không có số nguyên tố trong ma trận." << endl;
    }
```

```
    // (b) Dòng có ít nhất một số nguyên tố
    vector<int> rowsWithPrime = findRowsWithPrime(matrix);
    printRows(rowsWithPrime, "Các dòng chứa ít nhất một số nguyên tố: ");

    // (c) Dòng chỉ chứa số nguyên tố
    vector<int> rowsWithOnlyPrimes = findRowsWithOnlyPrimes(matrix);
    printRows(rowsWithOnlyPrimes, "Các dòng chỉ chứa số nguyên tố: ");

    return 0;
}
```

Bài tập 5.

Cho ma trận hai chiều m dòng n cột, các phần tử là các số nguyên dương.

a. Tìm dòng có tổng lớn nhất.

b. Sắp xếp các dòng sao cho dòng có tổng các phần tử lớn hơn sẽ nằm phía trên.

Bài làm

a. Tìm dòng có tổng lớn nhất
   - Duyệt từng dòng, tính tổng.
   - Lưu lại dòng có tổng lớn nhất.

b. Sắp xếp dòng theo tổng giảm dần
   - Dùng sort() với tiêu chí tổng dòng giảm dần.

Cài đặt

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

using namespace std;

// Hàm tính tổng của một dòng
int rowSum(const vector<int>& row) {
    int sum = 0;
    for (int num : row) {
        sum += num;
    }
    return sum;
}

// (a) Tìm dòng có tổng lớn nhất
int findMaxSumRow(const vector<vector<int>>& matrix) {
    int maxSum = -1, maxRowIndex = -1;
    for (int i = 0; i < matrix.size(); i++) {
```

```
        int sum = rowSum(matrix[i]);
        if (sum > maxSum) {
            maxSum = sum;
            maxRowIndex = i;
        }
    }
    return maxRowIndex;
}

// (b) Sắp xếp các dòng theo tổng giảm dần
void sortRowsBySum(vector<vector<int>>& matrix) {
    sort(matrix.begin(), matrix.end(), [](const vector<int>& a, const vector<int>&
b) {
        return rowSum(a) > rowSum(b); // Sắp xếp giảm dần theo tổng dòng
    });
}

// Hàm in ma trận
void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int num : row) {
            cout << num << "\t";
        }
        cout << endl;
    }
}

int main() {
    // Ma trận mẫu
    vector<vector<int>> matrix = {
        {4, 6, 7, 8},
        {10, 17, 19, 22},
        {29, 31, 37, 41},
        {9, 15, 21, 25}
    };

    cout << "Ma trận ban đầu:\n";
    printMatrix(matrix);

    // (a) Tìm dòng có tổng lớn nhất
    int maxRowIndex = findMaxSumRow(matrix);
    if (maxRowIndex != -1) {
        cout << "Dòng có tổng lớn nhất: " << maxRowIndex << endl;
    }

    // (b) Sắp xếp dòng theo tổng giảm dần
    sortRowsBySum(matrix);
```

```
    cout << "Ma trận sau khi sắp xếp dòng theo tổng giảm dần:\n";
    printMatrix(matrix);

    return 0;
}
```

Bài tập 6.

Cho mảng một chiều gồm n phần tử là các số nguyên không âm.

• Hãy sắp xếp các số chẵn trong mảng theo thứ tự tăng dần.

• Sắp xếp các số lẻ theo thứ tự giảm dần.

• Các số 0 giữ nguyên vị trí.

Bài làm

Bước 1: Phân loại số chẵn, số lẻ, số 0

- Duyệt qua mảng, lưu:
  - Số chẵn vào evens.
  - Số lẻ vào odds.
  - Vị trí của số 0 vào zeroPositions.

Bước 2: Sắp xếp

- Số chẵn được sắp xếp tăng dần.
- Số lẻ được sắp xếp giảm dần.

Bước 3: Xây dựng lại mảng

- Duyệt qua mảng ban đầu.
- Nếu vị trí thuộc zeroPositions, bỏ qua (giữ nguyên số 0).
- Nếu gặp số chẵn, thay bằng giá trị trong evens (tăng dần).
- Nếu gặp số lẻ, thay bằng giá trị trong odds (giảm dần).

Cài đặt

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

using namespace std;

// Hàm sắp xếp theo yêu cầu
```

```cpp
void sortArray(vector<int>& arr) {
    vector<int> evens, odds;
    vector<int> zeroPositions;

    // Bước 1: Phân loại số chẵn, số lẻ, vị trí số 0
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == 0) {
            zeroPositions.push_back(i);
        } else if (arr[i] % 2 == 0) {
            evens.push_back(arr[i]);
        } else {
            odds.push_back(arr[i]);
        }
    }

    // Bước 2: Sắp xếp số chẵn tăng dần, số lẻ giảm dần
    sort(evens.begin(), evens.end());          // Sắp xếp chẵn tăng dần
    sort(odds.rbegin(), odds.rend());          // Sắp xếp lẻ giảm dần

    // Bước 3: Lắp các số đã sắp xếp vào mảng, giữ nguyên số 0
    int evenIndex = 0, oddIndex = 0;
    for (int i = 0; i < arr.size(); i++) {
        if (find(zeroPositions.begin(), zeroPositions.end(), i) !=
zeroPositions.end()) {
            continue;  // Giữ nguyên số 0
        } else if (!evens.empty() && arr[i] % 2 == 0) {
            arr[i] = evens[evenIndex++];
        } else {
            arr[i] = odds[oddIndex++];
        }
    }
}

// Hàm hiển thị mảng
void printArray(const vector<int>& arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {3, 0, 2, 5, 4, 0, 7, 8, 0, 6, 1};

    cout << "Mảng ban đầu: ";
    printArray(arr);
```

```
    // Sắp xếp theo yêu cầu
    sortArray(arr);

    cout << "Mảng sau khi sắp xếp: ";
    printArray(arr);

    return 0;
}
```

Bài tập 7.

Cho mảng một chiều gồm n phần tử là các số nguyên dương.

• Hãy sắp xếp sao cho các phần tử chẵn ở đầu, các phần tử lẻ về cuối.

• Yêu cầu độ phức tạp là O(n).

Bài làm

Cách tiếp cận (O(n))

- Dùng hai con trỏ (left và right):
    - left trỏ vào vị trí cần đặt số chẵn.
    - right trỏ vào vị trí cần đặt số lẻ.
- Duyệt mảng một lần:
    - Nếu gặp số chẵn, đặt vào left, tăng left.
    - Nếu gặp số lẻ, giữ nguyên.
- Sau vòng lặp, các số chẵn nằm trước, số lẻ nằm sau.

Ý tưởng

- Duyệt mảng một lần (O(n)).
- Dùng biến left để đánh dấu vị trí đặt số chẵn.
- Nếu gặp số chẵn, hoán đổi nó với phần tử ở left, sau đó tăng left.

Độ phức tạp

- Duyệt mảng một lần (O(n)).
- Không dùng sort(), không cần mảng phụ.

Cài đặt

```
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

// Hàm sắp xếp chẵn lên đầu, lẻ xuống cuối (O(n))
void sortEvenOdd(vector<int>& arr) {
    int left = 0; // Vị trí chèn số chẵn

    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] % 2 == 0) { // Nếu là số chẵn
            swap(arr[left], arr[i]);
            left++; // Tăng vị trí đặt số chẵn
        }
    }
}

// Hàm hiển thị mảng
void printArray(const vector<int>& arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {3, 8, 5, 2, 4, 7, 6, 9, 10, 1};

    cout << "Mảng ban đầu: ";
    printArray(arr);

    // Sắp xếp theo yêu cầu
    sortEvenOdd(arr);

    cout << "Mảng sau khi sắp xếp (chẵn đầu, lẻ cuối): ";
    printArray(arr);

    return 0;
}
```

Bài tập 8.

Với mỗi hoán vị $A(a_1, a_2, a_3, \dots, a_N)$ của N số tự nhiên đầu tiên, ta ký hiệu: $B = (b_1, b_2, \dots, b_N)$ là một mảng, trong đó:

b_i = số lượng phần tử đứng trước số i và lớn hơn i, $\forall 1 \le i \le N$.

Mảng B được gọi là mảng nghịch thế của mảng A.

Ví dụ: Nếu A là một hoán vị của 9 số tự nhiên đầu tiên: A = (5, 9, 1, 8, 2, 6, 4, 7, 3)

Thì mảng nghịch thế sẽ là: B = (2, 3, 6, 4, 0, 2, 2, 1, 0)

Yêu cầu bài toán

a. Viết hàm tìm mảng nghịch thế của một hoán vị.

b. Viết hàm tìm hoán vị của một mảng nghịch thế (nếu có).

Bài làm

a. Tìm mảng nghịch thế B từ hoán vị A
   - Duyệt từng phần tử A[i], tìm các phần tử A[j] > A[i] với j < i.
   - Lưu kết quả vào B[A[i] - 1] vì A là hoán vị {1..N}.
   - Độ phức tạp: $O(n^2)$, có thể tối ưu $O(n\log_2 n)$ với Fenwick Tree.

b. Tìm hoán vị A từ B
   - Dùng tập hợp set<int> chứa số từ {1..N}.
   - Duyệt ngược từ N-1 → 0, chọn số thứ B[i] trong tập hợp để xây dựng A.
   - Độ phức tạp: $O(n\log_2 n)$.

Cài đặt

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

// (a) Tìm mảng nghịch thế từ hoán vị A
vector<int> findInversionArray(const vector<int>& A) {
    int N = A.size();
    vector<int> B(N, 0);

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < i; j++) {
            if (A[j] > A[i]) {
                B[A[i] - 1]++; // Lưu vào chỉ số A[i] - 1 vì A là hoán vị của
{1..N}
            }
        }
    }

    return B;
}

// (b) Tìm hoán vị A từ mảng nghịch thế B (nếu có)
vector<int> findPermutationFromInversionArray(const vector<int>& B) {
    int N = B.size();
    vector<int> A(N, 0);
```

```
    set<int> availableNumbers; // Tập hợp các số từ 1 đến N

    for (int i = 1; i <= N; i++) {
        availableNumbers.insert(i);
    }

    for (int i = N - 1; i >= 0; i--) {
        if (B[i] >= availableNumbers.size()) {
            return {}; // Không thể tạo hoán vị hợp lệ
        }

        auto it = availableNumbers.begin();
        advance(it, B[i]); // Lấy số ở vị trí B[i]
        A[i] = *it;
        availableNumbers.erase(it);
    }

    return A;
}

// Hàm hiển thị mảng
void printArray(const vector<int>& arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    // (a) Tìm mảng nghịch thế từ hoán vị A
    vector<int> A = {5, 9, 1, 8, 2, 6, 4, 7, 3};
    cout << "Hoán vị A: ";
    printArray(A);

    vector<int> B = findInversionArray(A);
    cout << "Mảng nghịch thế B: ";
    printArray(B);

    // (b) Tìm hoán vị A từ mảng nghịch thế B
    vector<int> reconstructedA = findPermutationFromInversionArray(B);
    cout << "Hoán vị tái tạo từ B: ";
    if (!reconstructedA.empty()) {
        printArray(reconstructedA);
    } else {
        cout << "Không tồn tại hoán vị hợp lệ.\n";
    }
```

```
    return 0;
}
```

Bài tập 9.

Cho thông tin về một sinh viên bao gồm:

• Mã số – là một số nguyên dương.

• Họ và đệm – là một chuỗi có tối đa 20 ký tự.

• Tên sinh viên – là một chuỗi có tối đa 40 ký tự.

• Ngày, tháng, năm sinh – là các số nguyên dương.

• Phái – là "Nam" hoặc "Nữ".

• Điểm trung bình – là các số thực có giá trị từ 0.00 → 10.00.

a. Viết chương trình nhập vào danh sách sinh viên (ít nhất là 10 sinh viên), không nhập trùng mã giữa các sinh viên với nhau và lưu trữ danh sách này vào một tập tin có tên SINHVIEN.DAT.

b. Sau đó vận dụng các thuật toán sắp xếp đã học để sắp xếp danh sách sinh viên theo thứ tự tăng dần theo Mã sinh viên. In danh sách sinh viên trong tập tin SINHVIEN.DAT sau khi sắp xếp ra màn hình.

c. Tạo các tập tin chỉ mục theo các khóa trong các trường hợp sau:

• Chỉ mục sắp xếp theo Mã sinh viên tăng dần.

• Chỉ mục sắp xếp theo Tên sinh viên từ A → Z, nếu cùng tên thì sắp xếp Họ và đệm theo thứ tự A → Z.

• Chỉ mục sắp xếp theo Điểm trung bình giảm dần.

d. Lưu các tập tin chỉ mục theo các khóa nêu trên vào trong ba tập tin tương ứng là:

• SVMASO.IDX (sắp xếp theo Mã sinh viên)

• SVTH.IDX (sắp xếp theo Tên sinh viên)

• SVDTB.IDX (sắp xếp theo Điểm trung bình)

e. Dựa vào các tập tin chỉ mục, mở và đọc danh sách sinh viên trong tập tin SINHVIEN.DAT theo đúng thứ tự sắp xếp ứng với từng tập tin chỉ mục.

f. Nhận xét về cách thực hiện việc sắp xếp dữ liệu trên tập tin theo chỉ mục.

Bài làm

a. Sắp xếp danh sách sinh viên theo mã số
   - Đọc danh sách từ file.
   - Sắp xếp tăng dần theo maSo.
   - Ghi kết quả ra màn hình.

b. Tạo tập tin chỉ mục
   - Lưu chỉ mục của danh sách theo từng tiêu chí: maSo, ten, diemTB.
   - Chỉ lưu chỉ mục (index), không ghi toàn bộ dữ liệu.

c. Đọc danh sách theo chỉ mục
   - Mở file chỉ mục, đọc danh sách theo thứ tự lưu trong file.
   - Hiển thị danh sách đã được sắp xếp theo chỉ mục.

Cài đặt

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

using namespace std;

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

// Hàm nhập danh sách sinh viên
void nhapDanhSach(vector<SinhVien>& danhSach) {
    int n;
    cout << "Nhập số lượng sinh viên (tối thiểu 10): ";
    cin >> n;
    cin.ignore();
    danhSach.resize(n);

    ofstream file("SINHVIEN.DAT", ios::binary | ios::trunc);
    if (!file) {
        cout << "Không thể mở file để ghi!\n";
        return;
    }

    for (int i = 0; i < n; i++) {
        cout << "Nhập thông tin sinh viên " << i + 1 << ":\n";
        cout << "Mã số: "; cin >> danhSach[i].maSo;
```

```
        cin.ignore();
        cout << "Họ và đệm: "; getline(cin, danhSach[i].hoVaDem);
        cout << "Tên: "; getline(cin, danhSach[i].ten);
        cout << "Ngày sinh: "; cin >> danhSach[i].ngaySinh;
        cout << "Tháng sinh: "; cin >> danhSach[i].thangSinh;
        cout << "Năm sinh: "; cin >> danhSach[i].namSinh;
        cin.ignore();
        cout << "Phái (Nam/Nữ): "; getline(cin, danhSach[i].phai);
        cout << "Điểm trung bình: "; cin >> danhSach[i].diemTB;
        cin.ignore();

        file.write(reinterpret_cast<char*>(&danhSach[i]), sizeof(SinhVien));
    }

    file.close();
    cout << "Lưu danh sách sinh viên vào SINHVIEN.DAT thành công!\n";
}

// Hàm đọc danh sách sinh viên từ file
void docDanhSach(vector<SinhVien>& danhSach) {
    ifstream file("SINHVIEN.DAT", ios::binary);
    if (!file) {
        cout << "Không thể mở file!\n";
        return;
    }

    danhSach.clear();
    SinhVien sv;
    while (file.read(reinterpret_cast<char*>(&sv), sizeof(SinhVien))) {
        danhSach.push_back(sv);
    }

    file.close();
}

// Hàm hiển thị danh sách sinh viên
void hienThiDanhSach(const vector<SinhVien>& danhSach) {
    cout << "\nDanh sách sinh viên:\n";
    for (const auto& sv : danhSach) {
        cout << sv.maSo << " | " << sv.hoVaDem << " " << sv.ten
             << " | " << sv.ngaySinh << "/" << sv.thangSinh << "/" << sv.namSinh
             << " | " << sv.phai << " | Điểm TB: " << sv.diemTB << endl;
    }
}

// Hàm tạo tập tin chỉ mục
void taoChiMuc(const vector<SinhVien>& danhSach, const string& tenFile,
```

```
                bool (*cmp)(const SinhVien&, const SinhVien&)) {
    vector<int> chiMuc(danhSach.size());
    for (size_t i = 0; i < danhSach.size(); i++) chiMuc[i] = i;

    sort(chiMuc.begin(), chiMuc.end(), [&](int a, int b) {
        return cmp(danhSach[a], danhSach[b]);
    });

    ofstream file(tenFile, ios::binary);
    for (int index : chiMuc) {
        file.write(reinterpret_cast<char*>(&index), sizeof(int));
    }
    file.close();
}

// Các hàm so sánh
bool cmpMaSo(const SinhVien& a, const SinhVien& b) {
    return a.maSo < b.maSo;
}

bool cmpTen(const SinhVien& a, const SinhVien& b) {
    if (a.ten != b.ten) return a.ten < b.ten;
    return a.hoVaDem < b.hoVaDem;
}

bool cmpDiemTB(const SinhVien& a, const SinhVien& b) {
    return a.diemTB > b.diemTB;
}

// Hàm đọc danh sách theo chỉ mục
void docTheoChiMuc(const vector<SinhVien>& danhSach, const string& tenFile) {
    ifstream file(tenFile, ios::binary);
    if (!file) {
        cout << "Không thể mở file chỉ mục!\n";
        return;
    }

    vector<SinhVien> dsSapXep;
    int index;
    while (file.read(reinterpret_cast<char*>(&index), sizeof(int))) {
        dsSapXep.push_back(danhSach[index]);
    }
    file.close();

    hienThiDanhSach(dsSapXep);
}
```

```
int main() {
    vector<SinhVien> danhSach;
    nhapDanhSach(danhSach);

    // Đọc lại danh sách từ file để đảm bảo dữ liệu nhất quán
    danhSach.clear();
    docDanhSach(danhSach);

    // Sắp xếp danh sách theo mã số và hiển thị
    sort(danhSach.begin(), danhSach.end(), cmpMaSo);
    cout << "\nDanh sách sau khi sắp xếp theo Mã số:\n";
    hienThiDanhSach(danhSach);

    // Tạo các tập tin chỉ mục
    taoChiMuc(danhSach, "SVMASO.IDX", cmpMaSo);
    taoChiMuc(danhSach, "SVTH.IDX", cmpTen);
    taoChiMuc(danhSach, "SVDTB.IDX", cmpDiemTB);

    // Đọc danh sách theo từng chỉ mục
    cout << "\nDanh sách theo Mã số tăng dần:\n";
    docTheoChiMuc(danhSach, "SVMASO.IDX");

    cout << "\nDanh sách theo Tên (A → Z):\n";
    docTheoChiMuc(danhSach, "SVTH.IDX");

    cout << "\nDanh sách theo Điểm trung bình giảm dần:\n";
    docTheoChiMuc(danhSach, "SVDTB.IDX");

    return 0;
}
```