THỰC HÀNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Bài Tập Sắp Xếp ( Bài tập nhóm )

Phân công:

3123411316\_Nguyễn Lương Tấn Trung: Câu hỏi 3, btcs 3, btud 6, 4

3123411034\_Khương Thanh Bình: Câu hỏi 1, btcs 1, btud 2,8

3123411283\_Lê Minh Thịnh: Câu hỏi 2, btcs 2, btud 3,7

3123411215\_Trịnh Tâm Như: Câu hỏi 4, btcs 4, btud 1,5

**Câu hỏi:**

**Câu 1:**

Interchange Sort

Ý tưởng thuật toán:

Ý tưởng của thuật toán này là bắt cặp tất cả các phần tử trong dãy cần sắp xếp và đổi chỗ hai phần tử trong cặp nếu chúng nghịch thế (không thỏa điều kiện thứ tự).

Để bắt cặp tất cả các phần tử trong dãy, ta dùng 2 vòng lặp. Vòng lặp ngoài sẽ chạy từ đầu dãy đến phần tử kế cuối. Vòng lặp trong sẽ chạy từ phần tử tiếp theo của vị trí đang xét ở vòng lặp ngoài. Mỗi lần xét ta sẽ so sánh 2 phần tử trong cặp. Nếu chúng nghịch thế thì sẽ hoán đổi vị trí của chúng.

Bubble Sort

Ý tưởng thuật toán: *Khái niệm nghịch thế* "Nghịch thế" (inversion) là một khái niệm để mô tả một cặp các phần tử trong một danh sách mà thứ tự của chúng đã bị đảo ngược so với thứ tự trong danh sách đã được sắp xếp. Chúng ta có thể hiểu định nghĩa này bằng cách xem xét một cặp phần tử trong danh sách và xác định liệu chúng có đứng đúng vị trí so với nhau không. Nếu phần tử đứng trước có giá trị lớn hơn phần tử đứng sau mà lại có chỉ số nhỏ hơn, ta nói rằng ta đã tạo ra một cặp nghịch thế.

Thuật toán sắp xếp nổi bọt thực hiện bằng cách đổi chỗ cho 2 phần tử liên tiếp nếu 2 phần tử đó tạo thành 1 nghịch thế. Lặp lại điều này cho đến khi không còn 2 phần tử nào thỏa mãn. Có thể chứng minh đươc số lần lặp không quá N-1, do một phần tử chỉ có thể nổi lên trên không quá N-1 lần. Ưu điểm: Code đơn giản, dễ hiểu và không tốn nhiều bộ nhớ. Nhược điểm: Độ phức tạp O(n2) không đủ nhanh với dữ liệu lớn.

Insertion Sort

Ý tưởng thuật toán: Thuật toán Selection Sort là một thuật toán sắp xếp đơn giản và dễ hiểu. Nó hoạt động bằng cách tìm kiếm phần tử nhỏ nhất trong mảng và đổi chỗ nó với phần tử đầu tiên. Sau đó, nó tìm kiếm phần tử nhỏ nhất trong phần còn lại của mảng và đổi chỗ nó với phần tử thứ hai, và tiếp tục như vậy cho đến khi toàn bộ mảng đã được sắp xếp.

Heap Sort

Ý tưởng thuật toán: Thuật toán Heap sort là một kỹ thuật sắp xếp phân loại dựa trên cấu trúc dữ liệu Binary Heap. Heap sort giúp sắp xếp các phần tử trong danh sach sao cho phần tử lớn nhất được xếp vào cuối danh sách, và quá trình này sẽ lặp lại chio các phần tử còn lại trong danh sách. Heap sort thường được người dùng lựa chọn sử dụng nhiều nhờ có tốc độ chạy nhanh và không quá phức tạp.

Merge Sort

Ý tưởng thuật toán: Ý tưởng cơ bản của thuật toán Merge Sort là chia mảng cần sắp xếp thành các phần nhỏ hơn, sắp xếp từng phần nhỏ này, sau đó kết hợp chúng lại với nhau để tạo ra một mảng đã sắp xếp. Quá trình này được thực hiện bằng cách sử dụng kỹ thuật chia để trị.

Dưới đây là quá trình cụ thể:

1. **Bước 1 - Phân chia (Divide):** Chia mảng cần sắp xếp thành hai phần bằng

nhau (hoặc gần bằng nhau) theo giữa của mảng.

2. **Bước 2 - Sắp xếp (Conquer):** Sử dụng đệ quy để sắp xếp mỗi phần nhỏ đã chia

thành từng phần riêng biệt.

3. **Bước 3 - Kết hợp (Combine):** Kết hợp hai phần đã sắp xếp thành một mảng đã

sắp xếp. Quá trình này là quá trình merge, trong đó các phần tử từ hai mảng con

được so sánh và sắp xếp lại thành một mảng lớn hơn.

Quick Sort

Ý tưởng thuật toán: Quick Sort hoạt động trên nguyên lý chia để trị. Chia mảng dữ liệu thành hai mảng dữ liệu con: mảng bên trái và mảng bên phải sao cho mảng bên trái gồm các phần tử có giá trị nhỏ hơn phần tử X (phần tử chốt) và mảng bên phải có các phần tử lớn hơn X. Nếu chúng ta sắp xếp được 2 mảng bên trái và bên phải thì dữ liệu ban đầu sẽ được sắp xếp toàn bộ. Để sắp xếp cho mảng bên trái và bên phải, ta cũng dùng ý tưởng chia dữ liệu để sao cho mảng bên trái và mảng bên phải có tối đa một phần tử thì mảng đã được sắp xếp.

Quá trình phân chia này diễn ra cho đến khi độ dài của các mảng con đều bằng 1. Với phương pháp đệ quy ta có thể sắp xếp nhanh các mảng con sau khi kết thúc chương trình ta được một mảng đã sắp xếp hoàn chỉnh.

Một số cách chọn phần tử chốt:

- Chọn phần tử đứng đầu hoặc đứng cuối làm phần tử chốt.

- Chọn phần tử đứng giữa danh sách làm phần tử chốt.

- Chọn phần tử trung vị trong ba phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt.

- Chọn phần tử ngẫu nhiên làm phần tử chốt. Tuy nhiên cách này rất dễ dẫn đến khả năng rơi vào các trường hợp đặc biệt.

Insertion Sort

Ý tưởng thuật toán: Phỏng theo cách làm của người chơi bài khi cần chèn thêm một con bài vào bộ bài cần sắp xếp

Thuật toán sắp xếp chèn thực hiện sắp xếp dãy số theo cách duyệt từng phần tử và chèn từng phần tử đó vào đúng vị trí trong mảng con(dãy số từ đầu đến phần tử phía trước nó) đã sắp xếp sao cho dãy số trong mảng sắp đã xếp đó vẫn đảm bảo tính chất của một dãy số tăng dần.

1. Khởi tạo mảng với dãy con đã sắp xếp có k = 1 phần tử(phần tử đầu tiên, phần tử

có chỉ số 0)

2. Duyệt từng phần tử từ phần tử thứ 2, tại mỗi lần duyệt phần tử ở chỉ số i thì đặt

phần tử đó vào một vị trí nào đó trong đoạn từ [0…i] sao cho dãy số từ [0…i]

vẫn đảm bảo tính chất dãy số tăng dần. Sau mỗi lần duyệt, số phần tử đã được sắp xếp k trong mảng tăng thêm 1 phần tử.

3. Lặp cho tới khi duyệt hết tất cả các phần tử của mảng

**Câu 2:**

Quick Sort là thuật toán tôi yêu thích vì thời gian xử lí khá nhanh. Với việc tận dụng một phần tử làm pivot và phân chia mảng thành các thành phần nhỏ, giúp giảm độ phức tạp còn O(nlogn).

Ngược lại, Interchange Sort là thuật toán tôi không thích. Mặc dù nó đơn giản nhưng sẽ không hiệu quả khi dùng trên dữ liệu lớn. Với độ phức tạp O(n2) trong các trường hợp xấu lẫn trung bình, việc so sánh và hoán đổi liên tục trở nên chập chạm so với QuickSort.

**Câu 3:**

1. Các thuật toán sắp xếp nội (Internal Sorting)

Sắp xếp nội là các thuật toán thực hiện trong bộ nhớ chính (RAM), phù hợp với dữ liệu có kích thước vừa phải.

a. Quick Sort (Sắp xếp nhanh)

Quick Sort là thuật toán chia để trị, chọn một phần tử làm chốt (pivot) và chia mảng thành hai phần.

Cài đăt:

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) { // 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);

}

}

b. Merge Sort (Sắp xếp trộn)

Merge Sort chia mảng thành các phần nhỏ, sắp xếp và trộn lại.

Cài đăt:

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) {

if (L[i] >= R[j]) { // Giảm dần

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

c. Heap Sort (Sắp xếp vun đống)

Cài đăt:

Heap Sort sử dụng cấu trúc heap để sắp xếp.

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]) // Giảm dần (max heap)

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(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--) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

d. Bubble Sort (Sắp xếp nổi bọt)

Bubble Sort so sánh và hoán đổi các phần tử liền kề.

Cài đăt:

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]) // Giảm dần

swap(arr[j], arr[j + 1]);

}

2. Các thuật toán sắp xếp ngoại (External Sorting)

Sắp xếp ngoại được sử dụng khi dữ liệu quá lớn để chứa trong RAM, thường sử dụng bộ nhớ phụ (ổ cứng).

External Merge Sort (Sắp xếp trộn ngoài)

External Merge Sort chia dữ liệu thành các khối nhỏ, sắp xếp trong RAM, sau đó trộn từ các tệp tạm.

Cài đặt

3. Nhận xét về các thuật toán

a. Quick Sort

• Ưu điểm: Hiệu quả cao với độ phức tạp trung bình O(n log n), không cần bộ nhớ phụ.

• Nhược điểm: Hiệu suất tệ nhất O(n²) nếu pivot chọn không tốt, không ổn định (stable).

• Ứng dụng: Phù hợp với dữ liệu trong RAM, kích thước vừa phải.

b. Merge Sort

• Ưu điểm: Ổn định, độ phức tạp luôn là O(n log n), hoạt động tốt với danh sách liên kết.

• Nhược điểm: Cần bộ nhớ phụ O(n), chậm hơn Quick Sort trong thực tế.

• Ứng dụng: Dữ liệu liên kết hoặc khi cần tính ổn định.

c. Heap Sort

• Ưu điểm: Độ phức tạp O(n log n), không cần bộ nhớ phụ lớn, tận dụng cấu trúc heap.

• Nhược điểm: Không ổn định, chậm hơn Quick Sort trong thực tế.

• Ứng dụng: Khi cần ưu tiên hàng đợi hoặc sắp xếp trong không gian hạn chế.

d. Bubble Sort

• Ưu điểm: Đơn giản, dễ cài đặt, tốt cho dữ liệu nhỏ hoặc gần sắp xếp.

• Nhược điểm: Độ phức tạp O(n²), không hiệu quả với dữ liệu lớn.

• Ứng dụng: Học thuật hoặc dữ liệu rất nhỏ.

e. External Merge Sort

• Ưu điểm: Xử lý được dữ liệu lớn vượt quá RAM, độ phức tạp O(n log n).

• Nhược điểm: Phụ thuộc vào I/O đĩa (chậm), cần quản lý tệp phức tạp.

• Ứng dụng: Cơ sở dữ liệu lớn, hệ thống file.

**Câu 4:**

**1. Sắp xếp nổi bọt (Bubble Sort)**

* **Ưu điểm:** 
  + Dễ hiểu và dễ cài đặt.
  + Không cần bộ nhớ phụ trợ ngoài mảng hoặc danh sách gốc.
  + Tốt cho danh sách nhỏ, vì đơn giản và nhanh khi dữ liệu gần như đã sắp xếp.
* **Nhược điểm:** 
  + Độ phức tạp: O(n²) trong trường hợp xấu, làm cho thuật toán này chậm khi danh sách có số lượng phần tử lớn.
  + Không tối ưu: Mặc dù có thể cải tiến một chút với việc kiểm tra xem mảng đã được sắp xếp hay chưa, nhưng vẫn không hiệu quả đối với danh sách lớn.
* **Khắc phục nhược điểm:** 
  + Sử dụng các thuật toán sắp xếp khác như QuickSort hoặc MergeSort để giảm độ phức tạp thời gian.
  + Sử dụng kỹ thuật "tối ưu hóa" của Bubble Sort, bằng cách dừng lại sớm nếu không có sự hoán đổi nào trong một vòng lặp.

**2. Sắp xếp chọn (Selection Sort)**

* **Ưu điểm:** 
  + Đơn giản, dễ hiểu và cài đặt.
  + Chỉ cần không gian phụ trợ O(1).
  + Tốt cho các danh sách nhỏ hoặc các tình huống khi không cần tối ưu tốc độ.
* **Nhược điểm:** 
  + Độ phức tạp: O(n²) trong cả trường hợp tốt nhất, xấu nhất và trung bình.
  + Quá chậm đối với các danh sách lớn.
* **Khắc phục nhược điểm:** 
  + Giống như với Bubble Sort, chọn các thuật toán sắp xếp hiệu quả hơn như QuickSort, MergeSort, hoặc HeapSort.

**3. Sắp xếp chèn (Insertion Sort)**

* **Ưu điểm:** 
  + Tốt cho các danh sách gần như đã sắp xếp (chạy nhanh hơn khi số phần tử cần sắp xếp ít).
  + Độ phức tạp tốt trong trường hợp tốt nhất: O(n) khi danh sách đã được sắp xếp.
  + Không cần bộ nhớ phụ trợ ngoài mảng hoặc danh sách gốc.
* **Nhược điểm:** 
  + Độ phức tạp: O(n²) trong trường hợp xấu (khi danh sách đảo ngược hoàn toàn).
  + Chậm khi số lượng phần tử lớn.
* **Khắc phục nhược điểm:** 
  + Sử dụng MergeSort hoặc QuickSort cho các danh sách lớn.
  + Chọn sử dụng Insertion Sort khi danh sách gần như đã sắp xếp hoặc có kích thước nhỏ.

**4. Sắp xếp nhanh (QuickSort)**

* **Ưu điểm:** 
  + Độ phức tạp trung bình: O(n log n), nhanh và hiệu quả đối với danh sách lớn.
  + Quản lý bộ nhớ tốt, có thể sắp xếp "in-place" mà không cần không gian phụ trợ lớn.
* **Nhược điểm:** 
  + Độ phức tạp trong trường hợp xấu: O(n²) khi chọn phân tách không tốt (ví dụ, nếu phân tách luôn chọn phần tử lớn nhất hoặc nhỏ nhất).
  + Không ổn định (các phần tử có giá trị bằng nhau có thể thay đổi vị trí sau khi sắp xếp).
* **Khắc phục nhược điểm:** 
  + Chọn phần tử phân tách tốt: Sử dụng các phương pháp phân tách như "median-of-three" (chọn phần tử trung bình của ba phần tử) để tránh trường hợp xấu.
  + Cải thiện sự ổn định bằng cách sử dụng các thuật toán sắp xếp ổn định, như MergeSort.

**5. Sắp xếp trộn (MergeSort)**

* **Ưu điểm:** 
  + Độ phức tạp: O(n log n) trong tất cả các trường hợp, nên rất nhanh cho danh sách lớn.
  + Ổn định: Các phần tử có giá trị bằng nhau không thay đổi vị trí.
  + Tốt cho sắp xếp các tập dữ liệu lớn và có thể sử dụng với các dữ liệu không thể truy cập trực tiếp (ví dụ, sắp xếp dữ liệu trên đĩa).
* **Nhược điểm:** 
  + Cần bộ nhớ phụ trợ O(n), do sử dụng phương pháp chia đôi danh sách.
  + Chậm hơn so với các thuật toán sắp xếp "in-place" như QuickSort hoặc HeapSort đối với danh sách nhỏ.
* **Khắc phục nhược điểm:** 
  + Chọn sử dụng MergeSort khi sắp xếp tập dữ liệu lớn hoặc khi cần sự ổn định.
  + Cân nhắc sử dụng QuickSort hoặc HeapSort cho trường hợp danh sách nhỏ hoặc yêu cầu tối ưu không gian bộ nhớ.

**6. Sắp xếp đống (HeapSort)**

* **Ưu điểm:** 
  + Độ phức tạp: O(n log n) trong mọi trường hợp.
  + In-place: Không cần bộ nhớ phụ trợ ngoài mảng gốc.
  + Tốt cho các ứng dụng yêu cầu sự ổn định của độ phức tạp thời gian.
* **Nhược điểm:** 
  + Chậm hơn QuickSort trong thực tế (do có thể có thêm nhiều thao tác so với QuickSort).
  + Không ổn định (các phần tử có giá trị giống nhau có thể thay đổi vị trí).
* **Khắc phục nhược điểm:** 
  + HeapSort có thể được sử dụng khi bộ nhớ là yếu tố quan trọng, và khi không yêu cầu sự ổn định.
  + Nếu yêu cầu sự ổn định, hãy chuyển sang MergeSort hoặc Insertion Sort.

**Bài tập cơ sở:**

**Câu 1:**

*a) Các thuật toán sắp xếp*

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

Thuật toán này duyệt qua từng phần tử, so sánh nó với các phần tử còn lại và hoán đổi nếu cần.

**Các bước sắp xếp dãy số:**

* Bước 1: So sánh 39 với từng số phía sau, đổi chỗ với số nhỏ hơn.
* Bước 2: Tiếp tục với số tiếp theo sau khi đã đặt số nhỏ nhất lên đầu.
* Bước 3: Lặp lại bước 2 cho đến khi danh sách hoàn chỉnh.

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

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

**2. Sắp xếp chọn trực tiếp (Selection Sort)**

Tìm phần tử nhỏ nhất trong danh sách và đưa nó về đầu. Lặp lại với danh sách còn lại.

**Các bước sắp xếp dãy số:**

* Bước 1: Tìm số nhỏ nhất (1) và đổi chỗ với số đầu tiên (39).
* Bước 2: Tìm số nhỏ nhất trong phần còn lại và đổi chỗ với phần tử thứ hai.
* Bước 3: Lặp lại bước 2 cho đến khi danh sách hoàn chỉnh.

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

**3. Sắp xếp chèn trực tiếp (Insertion Sort)**

Duyệt qua từng phần tử và chèn vào vị trí đúng trong phần đã sắp xếp.

**Các bước sắp xếp dãy số:**

* Bước 1: Lấy số thứ hai (8), đặt vào vị trí phù hợp so với số đầu tiên.
* Bước 2: Lấy số thứ ba (5), chèn vào vị trí thích hợp so với phần trước.
* Bước 3: Lặp lại bước 2 cho đến khi danh sách hoàn chỉnh.

Mô phỏng:

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

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

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

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

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

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

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

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

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

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

**4. Sắp xếp nổi bọt (Bubble Sort)**

Duyệt qua danh sách nhiều lần, mỗi lần đưa số lớn nhất trong phần chưa sắp xếp lên cuối danh sách.

**Các bước sắp xếp dãy số:**

* Bước 1: So sánh 39 và 8, đổi chỗ. Tiếp tục đổi chỗ trong danh sách.
* Bước 2: Lặp lại cho đến khi không cần đổi chỗ nữa.

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

*b) Cài đặt*

#include <iostream>

#include <vector>

using namespace std;

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]);

}

}

}

}

void selectionSort(vector<int>& arr) {

int n = arr.size();

for (int i = 0; i < n; i++) {

int min\_idx = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

swap(arr[i], arr[min\_idx]);

}

}

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;

}

}

void bubbleSort(vector<int>& arr) {

int n = arr.size();

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

}

}

}

}

void printArray(const vector<int>& arr) {

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

int main() {

vector<int> testArray = { 39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10 };

vector<int> arr1 = testArray;

interchangeSort(arr1);

cout << "Interchange Sort: ";

printArray(arr1);

vector<int> arr2 = testArray;

selectionSort(arr2);

cout << "Selection Sort: ";

printArray(arr2);

vector<int> arr3 = testArray;

insertionSort(arr3);

cout << "Insertion Sort: ";

printArray(arr3);

vector<int> arr4 = testArray;

bubbleSort(arr4);

cout << "Bubble Sort: ";

printArray(arr4);

return 0;

}

*c) Độ phức tạp*

1. **Sắp xếp đổi chỗ trực tiếp (Interchange Sort)**
   * Số lần so sánh: O(n^2)
   * Số lần hoán vị: O(n^2) (trong trường hợp xấu nhất)
2. **Sắp xếp chọn trực tiếp (Selection Sort)**
   * Số lần so sánh: O(n^2)
   * Số lần hoán vị: O(n) (tốt hơn Interchange Sort)
3. **Sắp xếp chèn trực tiếp (Insertion Sort)**
   * Tốt nhất: O(n) (nếu danh sách đã sắp xếp)
   * Trung bình & xấu nhất: O(n^2)
4. **Sắp xếp nổi bọt (Bubble Sort)**
   * Trung bình & xấu nhất: O(n^2)
   * Tốt nhất: O(n) (nếu danh sách đã sắp xếp)

**Câu 2:**

1. Quick Sort

Quick Sort chọn một phần tử làm "chốt" (pivot), phân vùng mảng thành hai phần: các phần tử nhỏ hơn chốt và lớn hơn chốt, rồi đệ quy sắp xếp các phần.

* **Bước 1**: Chọn pivot (giả sử chọn phần tử cuối: 10)
  + Dãy ban đầu: [8, 5, 1, 3, 6, 9, 12, 4, 7, 10]
  + Phân vùng: [8, 5, 1, 3, 6, 9, 4, 7] | [10] | [12]
* **Bước 2**: Đệ quy bên trái [8, 5, 1, 3, 6, 9, 4, 7], chọn pivot = 7
  + Phân vùng: [5, 1, 3, 6, 4] | [7] | [8, 9]
* **Bước 3**: Tiếp tục phân vùng nhỏ hơn:
  + [5, 1, 3, 6, 4] (pivot = 4) → [1, 3] | [4] | [5, 6]
  + [1, 3] (pivot = 3) → [1] | [3]
  + [5, 6] (pivot = 6) → [5] | [6]
  + [8, 9] (pivot = 9) → [8] | [9]
* **Kết quả cuối**: [1, 3, 4, 5, 6, 7, 8, 9, 10, 12]

void quicksort(int a[], int l, int r) {

int i = l, j = r;

int x = a[(l + r) / 2];

while (i <= j) {

while (a[i] < x) i++;

while (a[j] > x)j--;

if (i <= j) {

swap(a[i], a[j]);

i++;

j--;

}

}

if (i < r) quicksort(a, i, r);

if (l < j)quicksort(a, l, j);

}



1. Merge Sort

Merge Sort chia mảng thành các phần nhỏ nhất (1 phần tử), sau đó trộn các phần đã sắp xếp lại.

* **Bước 1**: Chia:
  + [8, 5, 1, 3, 6, 9, 12, 4, 7, 10]
  + [8, 5, 1, 3, 6] | [9, 12, 4, 7, 10]
  + [8, 5] | [1, 3, 6] | [9, 12] | [4, 7, 10]
  + [8] | [5] | [1] | [3, 6] | [9] | [12] | [4] | [7, 10]
* **Bước 2**: Trộn:
  + [5, 8] | [1, 3, 6] | [9, 12] | [4, 7, 10]
  + [1, 3, 5, 6, 8] | [4, 7, 9, 12, 10]
  + [1, 3, 4, 5, 6, 7, 8, 9, 10, 12]
* **Kết quả cuối**: [1, 3, 4, 5, 6, 7, 8, 9, 10, 12]

void merge(int a[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int\* L = new int[n1];

int\* R = new int[n2];

for (int i = 0; i < n1; i++)

L[i] = a[left + i];

for (int j = 0; j < n2; j++)

R[j] = a[mid + 1 + j];

int i = 0, j = 0, k = left;

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++;

}

delete[] L;

delete[] R;

}

void mergeSort(int a[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(a, left, mid);

mergeSort(a, mid + 1, right);

merge(a, left, mid, right);

}

}



1. Heap Sort

**Bước 1: Xây dựng max-heap (createHeap)**

Hàm createHeap gọi insertHeap để xây dựng max-heap từ mảng ban đầu. Quá trình bắt đầu từ phần tử không lá (tức là từ vị trí (n+1)/2 - 1 đến 0).

* **Dãy ban đầu**: [8, 5, 1, 3, 6, 9, 12, 4, 7, 10]
* **Kích thước mảng (n)**: 10
* **Vị trí bắt đầu (k)**: (10 + 1) / 2 - 1 = 5 - 1 = 4
* **Cấu trúc cây heap**: Với mảng 10 phần tử, các nút lá là từ index 5 đến 9, nên ta heapify từ index 4 ngược về 0.

**Lặp qua từng k từ 4 đến 0:**

* **k = 4 (phần tử 6)**:
  + Gọi insertHeap(a, 4, 9) để đảm bảo subtree với gốc tại 6 là max-heap.
  + Con trái: a[9] = 10, con phải: a[10] (không tồn tại).
  + So sánh: 6 < 10 → hoán đổi 6 và 10 → [8, 5, 1, 3, 10, 9, 12, 4, 7, 6].
  + Cập nhật r = 9, p = (9-1)/2 = 4 (vòng lặp kết thúc vì p < l).
  + Dãy: [8, 5, 1, 3, 10, 9, 12, 4, 7, 6].
* **k = 3 (phần tử 3)**:
  + Gọi insertHeap(a, 3, 9).
  + Con trái: a[7] = 7, con phải: a[8] = 6.
  + So sánh: 3 < 7 → hoán đổi 3 và 7 → [8, 5, 1, 7, 10, 9, 12, 4, 6, 6].
  + r = 7, p = (7-1)/2 = 3 (vòng lặp tiếp tục).
  + So sánh: 7 > 5 → hoán đổi 7 và 5 → [8, 7, 1, 5, 10, 9, 12, 4, 6, 6].
  + r = 3, p = (3-1)/2 = 1 (vòng lặp tiếp tục).
  + So sánh: 7 < 8 → dừng.
  + Dãy: [8, 7, 1, 5, 10, 9, 12, 4, 6, 6].
* **k = 2 (phần tử 1)**:
  + Gọi insertHeap(a, 2, 9).
  + Con trái: a[5] = 9, con phải: a[6] = 12.
  + So sánh: 1 < 9 → hoán đổi 1 và 9 → [8, 7, 9, 5, 10, 1, 12, 4, 6, 6].
  + r = 5, p = (5-1)/2 = 2 (vòng lặp tiếp tục).
  + So sánh: 9 < 7 → dừng.
  + Dãy: [8, 7, 9, 5, 10, 1, 12, 4, 6, 6].
* **k = 1 (phần tử 7)**:
  + Gọi insertHeap(a, 1, 9).
  + Con trái: a[3] = 5, con phải: a[4] = 10.
  + So sánh: 7 < 10 → hoán đổi 7 và 10 → [8, 10, 9, 5, 7, 1, 12, 4, 6, 6].
  + r = 4, p = (4-1)/2 = 1 (vòng lặp tiếp tục).
  + So sánh: 7 < 8 → dừng.
  + Dãy: [8, 10, 9, 5, 7, 1, 12, 4, 6, 6].
* **k = 0 (phần tử 8)**:
  + Gọi insertHeap(a, 0, 9).
  + Con trái: a[1] = 10, con phải: a[2] = 9.
  + So sánh: 8 < 10 → hoán đổi 8 và 10 → [10, 8, 9, 5, 7, 1, 12, 4, 6, 6].
  + r = 1, p = (1-1)/2 = 0 (vòng lặp tiếp tục).
  + So sánh: 8 < 10 → dừng.
  + Dãy: [10, 8, 9, 5, 7, 1, 12, 4, 6, 6].
* **Max-heap hoàn chỉnh**: [10, 8, 9, 5, 7, 1, 12, 4, 6, 6]
  + Kiểm tra: 10 (gốc) > 8 và 9 (con), 8 > 5 và 7, 9 > 1 và 12 (sai → cần điều chỉnh).

**Bước 2: Sắp xếp (heapSort)**

* **Dãy ban đầu sau createHeap**: [10, 8, 9, 5, 7, 1, 12, 4, 6, 6] (max-heap chưa hoàn hảo).
* Lặp từ k = 9 đến 1:
  + **k = 9**: Hoán đổi a[0] = 10 với a[9] = 6 → [6, 8, 9, 5, 7, 1, 12, 4, 6, 10].
    - Gọi insertHeap(a, 0, 8) để điều chỉnh heap với kích thước 9.
    - p = (8-1)/2 = 3, so sánh: 6 < 5 → dừng.
    - Dãy: [6, 8, 9, 5, 7, 1, 12, 4, 6, 10].
  + **k = 8**: Hoán đổi a[0] = 6 với a[8] = 6 → [6, 8, 9, 5, 7, 1, 12, 4, 6, 10].
    - Gọi insertHeap(a, 0, 7) (kích thước 8).
    - p = (7-1)/2 = 3, so sánh: 6 < 5 → dừng.
    - Dãy: [6, 8, 9, 5, 7, 1, 12, 4, 6, 10].
  + **k = 7**: Hoán đổi a[0] = 6 với a[7] = 4 → [4, 8, 9, 5, 7, 1, 12, 6, 6, 10].
    - Gọi insertHeap(a, 0, 6) (kích thước 7).
    - p = (6-1)/2 = 2, so sánh: 4 < 9 → hoán đổi → [9, 8, 4, 5, 7, 1, 12, 6, 6, 10].
    - r = 2, p = (2-1)/2 = 0, so sánh: 4 < 9 → dừng.
    - Dãy: [9, 8, 4, 5, 7, 1, 12, 6, 6, 10].
  + **k = 6**: Hoán đổi a[0] = 9 với a[6] = 12 → [12, 8, 4, 5, 7, 1, 9, 6, 6, 10].
    - Gọi insertHeap(a, 0, 5) (kích thước 6).
    - p = (5-1)/2 = 2, so sánh: 12 > 4 → dừng.
    - Dãy: [12, 8, 4, 5, 7, 1, 9, 6, 6, 10].
  + **k = 5**: Hoán đổi a[0] = 12 với a[5] = 1 → [1, 8, 4, 5, 7, 12, 9, 6, 6, 10].
    - Gọi insertHeap(a, 0, 4) (kích thước 5).
    - p = (4-1)/2 = 1, so sánh: 1 < 8 → hoán đổi → [8, 1, 4, 5, 7, 12, 9, 6, 6, 10].
    - r = 1, p = (1-1)/2 = 0, so sánh: 1 < 8 → dừng.
    - Dãy: [8, 1, 4, 5, 7, 12, 9, 6, 6, 10].
  + **k = 4**: Hoán đổi a[0] = 8 với a[4] = 7 → [7, 1, 4, 5, 8, 12, 9, 6, 6, 10].
    - Gọi insertHeap(a, 0, 3) (kích thước 4).
    - p = (3-1)/2 = 1, so sánh: 7 > 1 → dừng.
    - Dãy: [7, 1, 4, 5, 8, 12, 9, 6, 6, 10].
  + **k = 3**: Hoán đổi a[0] = 7 với a[3] = 5 → [5, 1, 4, 7, 8, 12, 9, 6, 6, 10].
    - Gọi insertHeap(a, 0, 2) (kích thước 3).
    - p = (2-1)/2 = 0, so sánh: 5 < 4 → dừng.
    - Dãy: [5, 1, 4, 7, 8, 12, 9, 6, 6, 10].
  + **k = 2**: Hoán đổi a[0] = 5 với a[2] = 4 → [4, 1, 5, 7, 8, 12, 9, 6, 6, 10].
    - Gọi insertHeap(a, 0, 1) (kích thước 2).
    - p = (1-1)/2 = 0, so sánh: 4 > 1 → dừng.
    - Dãy: [4, 1, 5, 7, 8, 12, 9, 6, 6, 10].
  + **k = 1**: Hoán đổi a[0] = 4 với a[1] = 1 → [1, 4, 5, 7, 8, 12, 9, 6, 6, 10].
    - Gọi insertHeap(a, 0, 0) (kích thước 1).
    - Vòng lặp không chạy (p < l).
    - Dãy: [1, 4, 5, 7, 8, 12, 9, 6, 6, 10].

**Kết quả cuối**

Dãy sau khi sắp xếp: [1, 4, 5, 6, 6, 7, 8, 9, 10, 12].

void insertHeap(int a[], int l, int r) {

int p = (r - 1) / 2; // Cha của phần tử r

while (p >= l) {

if (a[r] <= a[p]) {

break; // Thoát nếu phần tử r đã đúng vị trí

}

swap(a[r], a[p]);

r = p;

p = (r - 1) / 2; // Cập nhật cha mới

}

}

void createHeap(int a[], int n) {

for (int k = (n + 1) / 2 - 1; k >= 0; k--) {

insertHeap(a, k, n - 1);

}

}

void heapSort(int a[], int n) {

createHeap(a, n); // Xây dựng heap ban đầu

for (int k = n - 1; k >= 1; k--) {

swap(a[0], a[k]); // Đưa phần tử lớn nhất về cuối

insertHeap(a, 0, k - 1); // Điều chỉnh heap với kích thước giảm dần

}

}



**Câu 3:**

1. Cách thực hiện trong mã nguồn

a. Tạo dãy số ngẫu nhiên

• Hàm generateArray: Tạo mảng với n phần tử, mỗi phần tử là số ngẫu nhiên trong khoảng [0, 99] bằng rand() % 100.

• Seed được khởi tạo bằng srand(time(NULL)) trong main(), đảm bảo dữ liệu ngẫu nhiên thay đổi mỗi lần chạy.

b. Sao chép mảng

• Hàm copyArray: Đảm bảo mỗi thuật toán sắp xếp sử dụng cùng một bản sao của mảng gốc, giúp so sánh công bằng.

c. Đo lường hiệu suất

• Thời gian: Sử dụng chrono::high\_resolution\_clock để đo thời gian thực thi (đơn vị: mili giây).

• Số lần so sánh (comp): Được đếm trong mỗi thuật toán bằng biến long long& comp.

• Số lần đổi chỗ (swapCount): Được đếm khi gọi hàm swap() hoặc thực hiện gán trực tiếp (tùy thuật toán).

• Hàm testSort: Gói gọn quá trình đo lường và in kết quả cho từng thuật toán.

d. Các thuật toán sắp xếp

• Bubble Sort: So sánh và đổi chỗ các phần tử liền kề.

• Selection Sort: Tìm phần tử nhỏ nhất trong đoạn chưa sắp xếp và đổi chỗ.

• Insertion Sort: Chèn từng phần tử vào vị trí thích hợp.

• Quick Sort: Chia mảng quanh pivot và sắp xếp đệ quy.

• Merge Sort: Chia mảng thành các phần nhỏ, trộn lại theo thứ tự.

• Heap Sort: Xây heap và trích xuất phần tử lớn nhất dần dần.

e. Thực nghiệm

• Kích thước mảng: n = 10, 100, 1000, 5000, 10000.

• Mỗi kích thước được thử nghiệm một lần với cùng dữ liệu ngẫu nhiên.

2. Phân tích hiệu suất dựa trên mã

Dựa trên cách cài đặt, tôi sẽ dự đoán xu hướng hiệu suất (thời gian, số lần so sánh, đổi chỗ) và giải thích lý do.

a. Bubble Sort

• Độ phức tạp: O(n²).

• Thời gian: Tăng nhanh theo n², rất chậm với n lớn (5000, 10000).

• So sánh: Gần (n-1)\*(n-2)/2 lần trong trường hợp trung bình, vì so sánh từng cặp trong mọi vòng lặp.

• Đổi chỗ: Cao khi mảng ngẫu nhiên (khoảng n²/4), giảm nếu mảng gần sắp xếp.

• Dự đoán: Chậm nhất trong tất cả, đặc biệt với n = 10000.

b. Selection Sort

• Độ phức tạp: O(n²).

• Thời gian: Tương tự Bubble Sort, nhưng thường nhanh hơn một chút vì ít đổi chỗ hơn.

• So sánh: Luôn (n-1)\*(n-2)/2 lần, không phụ thuộc dữ liệu.

• Đổi chỗ: Tối đa n-1 lần (mỗi lần chọn min chỉ đổi chỗ một lần).

• Dự đoán: Nhanh hơn Bubble Sort về đổi chỗ, nhưng vẫn chậm với n lớn.

c. Insertion Sort

• Độ phức tạp: O(n²).

• Thời gian: Nhanh hơn Bubble/Selection với dữ liệu gần sắp xếp, nhưng chậm với dữ liệu ngẫu nhiên.

• So sánh: Khoảng n²/4 lần trung bình với dữ liệu ngẫu nhiên.

• Đổi chỗ: Tương tự số lần so sánh thành công, khoảng n²/4.

• Dự đoán: Tốt hơn Bubble/Selection với n nhỏ (10, 100), nhưng kém với n lớn.

d. Quick Sort

• Độ phức tạp: O(n log n) trung bình, O(n²) tệ nhất.

• Thời gian: Nhanh nhất trong trường hợp trung bình, nhưng có thể chậm nếu pivot không tốt (ở đây chọn arr[high]).

• So sánh: Khoảng n log n lần trung bình.

• Đổi chỗ: Ít hơn Bubble Sort, khoảng n log n/2.

• Dự đoán: Hiệu quả nhất với n lớn (5000, 10000), nhưng có thể biến động.

e. Merge Sort

• Độ phức tạp: O(n log n).

• Thời gian: Ổn định, chậm hơn Quick Sort do cần bộ nhớ phụ và sao chép.

• So sánh: Luôn khoảng n log n lần.

• Đổi chỗ: Không đổi chỗ trực tiếp, nhưng số lần gán tương đương n log n.

• Dự đoán: Ổn định, hiệu quả với mọi n, nhưng tốn bộ nhớ.

f. Heap Sort

• Độ phức tạp: O(n log n).

• Thời gian: Chậm hơn Quick Sort, nhưng ổn định hơn Bubble/Selection/Insertion.

• So sánh: Khoảng 2n log n lần do kiểm tra hai nhánh trong heapify.

• Đổi chỗ: Khoảng n log n/2 lần.

• Dự đoán: Hiệu quả trung bình, không nổi bật với n lớn.

3. Nhận xét dựa trên mã

a. Với n nhỏ (10, 100)

• Thời gian: Sự khác biệt không quá rõ ràng, Bubble/Selection/Insertion có thể nhanh hơn Quick/Merge/Heap do ít chi phí đệ quy.

• So sánh/Đổi chỗ: Bubble Sort có số lần đổi chỗ cao nhất, Selection Sort ít đổi chỗ nhất.

• Nhận xét: Insertion Sort thường tốt nhất với n nhỏ do tận dụng dữ liệu局部 sắp xếp.

b. Với n trung bình (1000)

• Thời gian: Quick Sort bắt đầu vượt trội, Bubble/Selection/Insertion chậm rõ rệt.

• So sánh/Đổi chỗ: Quick/Merge/Heap có số lần so sánh/đổi chỗ ít hơn đáng kể.

• Nhận xét: Quick Sort và Merge Sort là lựa chọn tốt, Heap Sort trung bình.

c. Với n lớn (5000, 10000)

• Thời gian: Quick Sort nhanh nhất, Bubble/Selection/Insertion gần như không khả thi.

• So sánh/Đổi chỗ: Bubble Sort có số lần so sánh/đổi chỗ cao nhất (gần 50 triệu với n=10000), Quick/Merge/Heap chỉ khoảng vài chục nghìn.

• Nhận xét: Quick Sort tối ưu nhất, Merge Sort ổn định, Heap Sort chấp nhận được.

d. Tổng quan

• Bubble Sort: Chỉ dùng cho học thuật hoặc n rất nhỏ, không thực tế với n lớn.

• Selection Sort: Ít đổi chỗ hơn Bubble, nhưng vẫn chậm.

• Insertion Sort: Tốt với n nhỏ hoặc dữ liệu gần sắp xếp.

• Quick Sort: Hiệu quả cao nhất trong thực tế, nhưng không ổn định.

• Merge Sort: Ổn định, phù hợp khi cần đảm bảo O(n log n).

• Heap Sort: Trung bình, tiết kiệm bộ nhớ nhưng không nhanh bằng Quick Sort.

4. Đề xuất cải thiện mã

1. Thực nghiệm nhiều lần: Chạy mỗi thuật toán t lần (ví dụ: t = 10) và lấy trung bình để giảm nhiễu.

2. Chọn pivot tốt hơn cho Quick Sort: Dùng median-of-three thay vì arr[high] để tránh trường hợp tệ nhất.

3. Tối ưu Insertion Sort: Thêm cờ để giảm vòng lặp nếu mảng đã sắp xếp.

4. Lưu kết quả vào file: Ghi dữ liệu vào tệp để dễ phân tích với n lớn.

**Câu 4:**

|  |  |
| --- | --- |
| Thuật toán | Thời gian thực thi |
| Interchange Sort | 20p33s |
| Selection Sort | 15p24s |
| Insertion Sort | 10p15s |
| Bubble Sort | 20p38s |
| Quick Sort | 1p31s |
| Merge Sort | 2p17s |
| Heap Sort | 3p8s |

Cấu hình máy

A screenshot of a computer

AI-generated content may be incorrect.

**Bài tập ứng dụng:**

**Câu 1:**

Cho dãy n số nguyên: a0, a1,..., an-1

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

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Hàm tìm vị trí của k phần tử có giá trị lớn nhất trong dãy

void tim\_k\_phan\_tu\_lon\_nhat(const vector<int>& arr, int k) {

// Tạo một bản sao của dãy để tránh thay đổi dãy gốc

vector<int> arr\_copy = arr;

// Tạo một mảng indices lưu các chỉ số của các phần tử trong dãy

vector<int> indices(arr.size());

// Lưu lại các chỉ số

for (int i = 0; i < arr.size(); ++i) {

indices[i] = i;

}

// Sắp xếp dãy chỉ số theo giá trị giảm dần của các phần tử trong arr\_copy

sort(indices.begin(), indices.end(), [&arr\_copy](int i1, int i2) {

return arr\_copy[i1] > arr\_copy[i2]; // So sánh các phần tử dựa trên giá trị của arr\_copy

});

// In ra vị trí của k phần tử có giá trị lớn nhất

cout << "Vị trí của " << k << " phần tử có giá trị lớn nhất: ";

for (int i = 0; i < k; ++i) {

cout << indices[i] << " "; // In ra vị trí các phần tử có giá trị lớn nhất

}

cout << endl;

}

int main() {

int n, k;

// Nhập số lượng phần tử trong dãy

cout << "Nhập số lượng phần tử trong dãy: ";

cin >> n; // Nhập số phần tử của dãy

vector<int> arr(n); // Khởi tạo vector arr có kích thước n

// Nhập các phần tử của dãy

cout << "Nhập " << n << " phần tử của dãy: ";

for (int i = 0; i < n; ++i) {

cin >> arr[i]; // Nhập các phần tử vào vector arr

}

// Nhập số lượng phần tử lớn nhất cần tìm

cout << "Nhập số lượng phần tử lớn nhất cần tìm: ";

cin >> k; // Nhập k

// Gọi hàm tim\_k\_phan\_tu\_lon\_nhat để tìm và in vị trí của k phần tử lớn nhất

tim\_k\_phan\_tu\_lon\_nhat(arr, k);

return 0; }

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

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Hàm tính tổng các chữ số của một số

int tong\_cac\_chu\_so(int n) {

int sum = 0;

// Vòng lặp để tính tổng các chữ số

while (n != 0) {

sum += n % 10; // Lấy chữ số cuối cùng của n

n /= 10; // Loại bỏ chữ số cuối cùng

}

return sum; // Trả về tổng các chữ số

}

// Hàm sắp xếp dãy theo tổng các chữ số của mỗi phần tử

void sap\_xep\_theo\_tong\_chu\_so(vector<int>& arr) {

// Sử dụng hàm sort để sắp xếp dãy theo tổng chữ số của các phần tử

sort(arr.begin(), arr.end(), [](int a, int b) {

return tong\_cac\_chu\_so(a) < tong\_cac\_chu\_so(b); // So sánh tổng chữ số của hai phần tử

});

}

int main() {

int n;

// Nhập số lượng phần tử trong dãy

cout << "Nhập số lượng phần tử trong dãy: ";

cin >> n;

vector<int> arr(n);

// Nhập các phần tử của dãy

cout << "Nhập " << n << " phần tử của dãy: ";

for (int i = 0; i < n; ++i) {

cin >> arr[i]; // Nhập từng phần tử của dãy

}

// Gọi hàm sắp xếp theo tổng chữ số

sap\_xep\_theo\_tong\_chu\_so(arr);

// In ra dãy sau khi sắp xếp

cout << "Dãy sau khi sắp xếp theo tổng chữ số: ";

for (int num : arr) {

cout << num << " "; // In từng phần tử của dãy đã sắp xếp

}

cout << endl;

return 0; // Kết thúc chương trình

}

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

#include <iostream>

#include <vector>

#include <cmath>

#include <algorithm>

using namespace std;

// Hàm kiểm tra số nguyên tố

bool la\_so\_nguyen\_to(int n) {

if (n <= 1) return false; // Các số <= 1 không phải là số nguyên tố

for (int i = 2; i <= sqrt(n); ++i) {

if (n % i == 0) return false; // Nếu chia hết cho bất kỳ số nào, n không phải là số nguyên tố

}

return true; // Nếu không chia hết cho số nào thì n là số nguyên tố

}

// Hàm xóa các số nguyên tố khỏi dãy

void xoa\_so\_nguyen\_to(vector<int>& arr) {

// Sử dụng remove\_if để xóa các số nguyên tố

arr.erase(remove\_if(arr.begin(), arr.end(), la\_so\_nguyen\_to), arr.end());

}

int main() {

int n;

// Nhập số lượng phần tử trong dãy

cout << "Nhập số lượng phần tử trong dãy: ";

cin >> n;

vector<int> arr(n);

// Nhập các phần tử của dãy

cout << "Nhập " << n << " phần tử của dãy: ";

for (int i = 0; i < n; ++i) {

cin >> arr[i]; // Nhập từng phần tử của dãy

}

// Gọi hàm xóa các số nguyên tố

xoa\_so\_nguyen\_to(arr);

// In ra dãy sau khi xóa các số nguyên tố

cout << "Dãy sau khi xóa các số nguyên tố: ";

for (int num : arr) {

cout << num << " "; // In từng phần tử của dãy sau khi xóa

}

cout << endl;

return 0;

}

**Câu 2:**

#include <iostream>

using namespace std;

struct SoHang { //Khai bao cau truc so hang

double heSo;

int bac;

};

void insertionSort(SoHang dathuc[], int soLuong) { //Thuat toan sap xep chen

for (int i = 1; i < soLuong; i++) {

SoHang temp = dathuc[i];

int j = i - 1;

while (j >= 0 && dathuc[j].bac > temp.bac) {

dathuc[j + 1] = dathuc[j];

j--;

}

dathuc[j + 1] = temp;

}

}

int main() {

int soLuong;

cout << "Nhap so luong so hang: "; //Nhap so luong

cin >> soLuong;

SoHang\* dathuc = new SoHang[soLuong];

cout << "Nhap cac so hang va bac tuong ung: ";

for (int i = 0; i < soLuong; i++) { //Nhap so hang va bac

cout << "Nhap so hang thu " << i + 1 << ": ";

cin >> dathuc[i].heSo >> dathuc[i].bac;

}

insertionSort(dathuc, soLuong); //Sap xep

cout << "Day thuc sau khi sap xep: \n";

for (int i = 0; i < soLuong; i++) {

cout << dathuc[i].heSo << "x^" << dathuc[i].bac << " ";

}

delete[] dathuc;

cout << endl;

return 0;

}

Code trên đã xây dựng cấu trúc số hạng đi kèm bậc của nó, cùng với thuật toán sắp xếp chèn ( InsertionSort )

Chạy chương trình

->

A black screen with white text

AI-generated content may be incorrect.

Các số đã được sắp xếp tang dần của bậc: 1->3->8

**Câu 3:**

#include <iostream>

using namespace std;

struct Room {

int roomNumber; // Số phòng

char house; // Nhà

int capacity; // Sức chứa

};

// Hàm hoán đổi

void swap(Room& a, Room& b) {

Room temp = a;

a = b;

b = temp;

}

void display(Room a[], int n) {

for (int i = 0; i < n; i++) {

cout << "Phong " << a[i].roomNumber << " - Nha " << a[i].house

<< " - Suc chua " << a[i].capacity << endl;

}

cout << endl;

}

// Sắp xếp theo khả năng chứa giảm dần

void sortCapacityDesc(Room a[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j =i+1 ; j<n; j++) {

if (a[i].capacity < a[j].capacity) {

swap(a[i], a[j]);

}

}

}

}

// Sắp xếp theo Nhà tăng dần, cùng Nhà thì theo Số phòng tăng dần

void sortHouseAndRoomAsc(Room a[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = i+1; j < n ; j++) {

if (a[i].house > a[j].house ||

(a[i].house == a[j].house && a[i].roomNumber > a[j].roomNumber)) {

swap(a[i], a[j]);

}

}

}

}

// Sắp xếp theo Nhà tăng dần, cùng Nhà thì theo Khả năng chứa giảm dần

void sortHouseAscCapacityDesc(Room a[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j =i+1; j < n; j++) {

if (a[i].house > a[j].house ||

(a[i].house == a[j].house && a[i].capacity < a[j].capacity)) {

swap(a[i], a[j]);

}

}

}

}

void test() {

Room sample[] = {

{101, 'B', 50}, // Phòng 101, Nhà B, Sức chứa 50

{102, 'A', 100}, // Phòng 102, Nhà A, Sức chứa 100

{103, 'B', 75}, // Phòng 103, Nhà B, Sức chứa 75

{104, 'C', 200}, // Phòng 104, Nhà C, Sức chứa 200

{105, 'A', 30} // Phòng 105, Nhà A, Sức chứa 30

};

int n = 5;

Room\* roomsCopy1 = new Room[n];

Room\* roomsCopy2 = new Room[n];

Room\* roomsCopy3 = new Room[n];

for (int i = 0; i < n; i++) {

roomsCopy1[i] = roomsCopy2[i] = roomsCopy3[i] = sample[i];

}

cout << "Danh sach phong thi ban dau:\n";

display(sample, n);

cout << "Danh sach phong thi theo suc chua giam dan:\n";

sortCapacityDesc(roomsCopy1, n);

display(roomsCopy1, n);

cout << "Danh sach phong thi theo nha tang dan, so phong tang dan:\n";

sortHouseAndRoomAsc(roomsCopy2, n);

display(roomsCopy2, n);

cout << "Danh sach phong thi theo nha tang dan, suc chua giam dan:\n";

sortHouseAscCapacityDesc(roomsCopy3, n);

display(roomsCopy3, n);

// Xóa con trỏ

delete[] roomsCopy1;

delete[] roomsCopy2;

delete[] roomsCopy3;

}

int main() {

test();

return 0;

}

A screenshot of a computer

AI-generated content may be incorrect.

**Câu 4:**

Bước 1: Xác định và kiểm tra số nguyên tố

• Định nghĩa số nguyên tố: Một số nguyên dương lớn hơn 1, chỉ chia hết cho 1 và chính nó.

• Cách kiểm tra:

1. Nếu số x<2 x < 2 x<2, nó không phải số nguyên tố.

2. Với x≥2 x \geq 2 x≥2, kiểm tra xem x x x có chia hết cho bất kỳ số nguyên nào từ 2 đến x \sqrt{x} x không:

 Nếu có, x x x không phải số nguyên tố.

 Nếu không, x x x là số nguyên tố.

Bước 2: Nhập ma trận

1. Xác định kích thước:

o Nhập số dòng m m m và số cột n n n từ người dùng (giả sử m,n≤100 m, n \leq 100 m,n≤100 để giới hạn).

2. Nhập phần tử:

o Dùng hai vòng lặp lồng nhau:

 Vòng ngoài lặp từ dòng 0 đến m−1 m-1 m−1.

 Vòng trong lặp từ cột 0 đến n−1 n-1 n−1.

o Nhập từng phần tử matrix[i][j] matrix[i][j] matrix[i][j] (giả sử tất cả là số nguyên dương).

3. Lưu trữ: Ma trận được lưu trong một mảng hai chiều kích thước m×n m \times n m×n.

Bước 3: Giải quyết yêu cầu a - Tìm số nguyên tố lớn nhất

1. Khởi tạo biến:

o Tạo biến maxPrime maxPrime maxPrime để lưu số nguyên tố lớn nhất, ban đầu gán giá trị không hợp lệ (ví dụ: -1) để biểu thị chưa tìm thấy.

2. Duyệt ma trận:

o Dùng hai vòng lặp:

 Lặp qua từng dòng i i i từ 0 đến m−1 m-1 m−1.

 Lặp qua từng cột j j j từ 0 đến n−1 n-1 n−1.

o Với mỗi phần tử matrix[i][j] matrix[i][j] matrix[i][j]:

 Kiểm tra xem nó có phải số nguyên tố không (dùng cách ở Bước 1).

 Nếu là số nguyên tố và lớn hơn maxPrime maxPrime maxPrime, cập nhật maxPrime=matrix[i][j] maxPrime = matrix[i][j] maxPrime=matrix[i][j].

3. Kết quả:

o Nếu maxPrime≠−1 maxPrime \neq -1 maxPrime=−1, đó là số nguyên tố lớn nhất.

o Nếu maxPrime=−1 maxPrime = -1 maxPrime=−1, ma trận không chứa số nguyên tố nào.

Bước 4: Giải quyết yêu cầu b - Tìm các dòng chứa ít nhất một số nguyên tố

1. Duyệt từng dòng:

o Lặp qua từng dòng i i i từ 0 đến m−1 m-1 m−1.

2. Kiểm tra dòng:

o Với mỗi dòng i i i, lặp qua các cột j j j từ 0 đến n−1 n-1 n−1:

 Kiểm tra matrix[i][j] matrix[i][j] matrix[i][j] có phải số nguyên tố không.

 Nếu tìm thấy ít nhất một số nguyên tố, ghi nhận dòng i i i và dừng kiểm tra dòng đó.

3. Lưu và hiển thị:

o Ghi nhận các dòng có ít nhất một số nguyên tố (có thể dùng mảng hoặc in trực tiếp).

o Ví dụ: Nếu dòng 0 và 2 chứa số nguyên tố, kết quả là "0 2".

Bước 5: Giải quyết yêu cầu c - Tìm các dòng chỉ chứa số nguyên tố

1. Duyệt từng dòng:

o Lặp qua từng dòng i i i từ 0 đến m−1 m-1 m−1.

2. Kiểm tra dòng:

o Với mỗi dòng i i i, lặp qua các cột j j j từ 0 đến n−1 n-1 n−1:

 Kiểm tra từng matrix[i][j] matrix[i][j] matrix[i][j] có phải số nguyên tố không.

 Nếu bất kỳ phần tử nào không phải số nguyên tố, dòng đó không thỏa mãn → bỏ qua.

 Nếu tất cả phần tử đều là số nguyên tố, dòng đó thỏa mãn.

3. Lưu và hiển thị:

o Ghi nhận các dòng chỉ chứa số nguyên tố (in trực tiếp hoặc lưu vào mảng).

o Ví dụ: Nếu dòng 1 chỉ chứa số nguyên tố, kết quả là "1".

Bước 6: Hiển thị kết quả

1. Yêu cầu a: In số nguyên tố lớn nhất (hoặc thông báo "Không có" nếu không tìm thấy).

2. Yêu cầu b: In danh sách các dòng chứa ít nhất một số nguyên tố.

3. Yêu cầu c: In danh sách các dòng chỉ chứa số nguyên tố.

Ví dụ minh họa

Giả sử ma trận 3×3 3 \times 3 3×3:

text

Thu gọnBọc lạiSao chép

2 4 5

6 7 3

11 13 17

• Số nguyên tố: 2, 3, 5, 7, 11, 13, 17.

• Yêu cầu a: Số nguyên tố lớn nhất = 17.

• Yêu cầu b: Dòng có số nguyên tố: 0 (2, 5), 1 (7, 3), 2 (11, 13, 17) → Kết quả: "0 1 2".

• Yêu cầu c: Dòng chỉ chứa số nguyên tố: Dòng 2 (11, 13, 17) → Kết quả: "2".

Nhận xét

• Hiệu quả: Các bước trên đều duyệt toàn bộ ma trận, độ phức tạp là O(m⋅n⋅k) O(m \cdot n \cdot \sqrt{k}) O(m⋅n⋅k), với k k k là giá trị lớn nhất trong ma trận (do kiểm tra số nguyên tố).

• Đơn giản: Logic dễ hiểu, không cần cấu trúc dữ liệu phức tạp.

• Mở rộng: Có thể thêm kiểm tra đầu vào (số nguyên dương) hoặc tối ưu kiểm tra số nguyên tố (ví dụ: dùng sàng Eratosthenes nếu cần).

Code

#include <iostream>

#include <cmath>

#include <string>

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 maxPrimeInMatrix(int matrix[][100], int m, int n) {

int maxPrime = -1;

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (isPrime(matrix[i][j]) && matrix[i][j] > maxPrime) {

maxPrime = matrix[i][j];

}

}

}

return maxPrime;

}

// b. Tìm các dòng chứa ít nhất một số nguyên tố

void rowsWithPrime(int matrix[][100], int m, int n) {

cout << "Cac dong co chua so nguyen to: ";

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (isPrime(matrix[i][j])) {

cout << i << " ";

break;

}

}

}

cout << endl;

}

// c. Tìm các dòng chỉ chứa số nguyên tố

void rowsOnlyPrime(int matrix[][100], int m, int n) {

cout << "Cac dong chi chua so nguyen to: ";

for (int i = 0; i < m; i++) {

bool allPrime = true;

for (int j = 0; j < n; j++) {

if (!isPrime(matrix[i][j])) {

allPrime = false;

break;

}

}

if (allPrime) {

cout << i << " ";

}

}

cout << endl;

}

int main() {

int m, n;

cout << "Nhap so dong m: ";

cin >> m;

cout << "Nhap so cot n: ";

cin >> n;

int matrix[100][100]; // Mảng tĩnh có thể lưu tối đa 100x100 phần tử

// Nhập ma trận

cout << "Nhap cac phan tu cua ma tran:\n";

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

cin >> matrix[i][j];

}

}

// a. Tìm số nguyên tố lớn nhất

int maxPrime = maxPrimeInMatrix(matrix, m, n);

cout << "So nguyen to lon nhat: " << (maxPrime != -1 ? to\_string(maxPrime) : "Khong co") << endl;

// b. Tìm các dòng có chứa số nguyên tố

rowsWithPrime(matrix, m, n);

// c. Tìm các dòng chỉ chứa số nguyên tố

rowsOnlyPrime(matrix, m, n);

return 0;

}

**Câu 5:**

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.

#include <iostream>

#include <vector>

using namespace std;

// Hàm tính tổng các phần tử trong một dòng

int tong\_dong(const vector<int>& row) {

int sum = 0;

for (int val : row) {

sum += val; // Cộng tất cả các phần tử trong dòng

}

return sum;

}

// Hàm tìm dòng có tổng lớn nhất

void tim\_dong\_co\_tong\_lon\_nhat(const vector<vector<int>>& matrix) {

int max\_sum = -1; // Khởi tạo tổng lớn nhất ban đầu là -1 (tổng luôn là số dương)

int row\_index = -1; // Chỉ số của dòng có tổng lớn nhất

// Duyệt qua từng dòng của ma trận

for (int i = 0; i < matrix.size(); ++i) {

int current\_sum = tong\_dong(matrix[i]); // Tính tổng của dòng i

if (current\_sum > max\_sum) {

max\_sum = current\_sum; // Cập nhật tổng lớn nhất

row\_index = i; // Cập nhật chỉ số của dòng có tổng lớn nhất

}

}

// In ra dòng có tổng lớn nhất

cout << "Dòng có tổng lớn nhất là dòng thứ " << row\_index + 1 << " với tổng là: " << max\_sum << endl;

}

int main() {

int m, n;

// Nhập số lượng dòng và cột của ma trận

cout << "Nhập số lượng dòng m: ";

cin >> m;

cout << "Nhập số lượng cột n: ";

cin >> n;

vector<vector<int>> matrix(m, vector<int>(n));

// Nhập các phần tử của ma trận

cout << "Nhập các phần tử của ma trận (m dòng, n cột):" << endl;

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

cin >> matrix[i][j]; // Nhập từng phần tử

}

}

// Gọi hàm tìm dòng có tổng lớn nhất

tim\_dong\_co\_tong\_lon\_nhat(matrix);

return 0;

}

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Hàm tính tổng các phần tử trong một dòng

int tong\_dong(const vector<int>& row) {

int sum = 0;

for (int val : row) {

sum += val; // Cộng tất cả các phần tử trong dòng

}

return sum;

}

// Hàm sắp xếp các dòng sao cho dòng có tổng các phần tử lớn hơn sẽ nằm phía trên

void sap\_xep\_theo\_tong\_dong(vector<vector<int>>& matrix) {

// Sắp xếp các dòng của ma trận theo tổng các phần tử trong mỗi dòng

sort(matrix.begin(), matrix.end(), [](const vector<int>& a, const vector<int>& b) {

return tong\_dong(a) > tong\_dong(b); // Sắp xếp theo tổng giảm dần

});

}

// Hàm in ma trận

void in\_ma\_tran(const vector<vector<int>>& matrix) {

for (const auto& row : matrix) {

for (int val : row) {

cout << val << " "; // In từng phần tử của dòng

}

cout << endl; // Xuống dòng sau mỗi dòng ma trận

}

}

int main() {

int m, n;

// Nhập số lượng dòng và cột của ma trận

cout << "Nhập số lượng dòng m: ";

cin >> m;

cout << "Nhập số lượng cột n: ";

cin >> n;

vector<vector<int>> matrix(m, vector<int>(n));

// Nhập các phần tử của ma trận

cout << "Nhập các phần tử của ma trận (m dòng, n cột):" << endl;

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

cin >> matrix[i][j]; // Nhập từng phần tử

}

}

// Sắp xếp ma trận theo tổng các phần tử trong từng dòng

sap\_xep\_theo\_tong\_dong(matrix);

// In ra ma trận sau khi sắp xếp

cout << "Ma trận sau khi sắp xếp các dòng theo tổng các phần tử lớn hơn: " << endl;

in\_ma\_tran(matrix);

return 0;

}

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

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Hàm tính tổng các phần tử trong một dòng

int tong\_dong(const vector<int>& row) {

int sum = 0;

for (int val : row) {

sum += val; // Cộng tất cả các phần tử trong dòng

}

return sum;

}

// Hàm sắp xếp các dòng sao cho dòng có tổng các phần tử lớn hơn sẽ nằm phía trên

void sap\_xep\_theo\_tong\_dong(vector<vector<int>>& matrix) {

// Sắp xếp các dòng của ma trận theo tổng các phần tử trong mỗi dòng

sort(matrix.begin(), matrix.end(), [](const vector<int>& a, const vector<int>& b) {

return tong\_dong(a) > tong\_dong(b); // Sắp xếp theo tổng giảm dần

});

}

// Hàm in ma trận

void in\_ma\_tran(const vector<vector<int>>& matrix) {

for (const auto& row : matrix) {

for (int val : row) {

cout << val << " "; // In từng phần tử của dòng

}

cout << endl; // Xuống dòng sau mỗi dòng ma trận

}

}

int main() {

int m, n;

// Nhập số lượng dòng và cột của ma trận

cout << "Nhập số lượng dòng m: ";

cin >> m;

cout << "Nhập số lượng cột n: ";

cin >> n;

vector<vector<int>> matrix(m, vector<int>(n));

// Nhập các phần tử của ma trận

cout << "Nhập các phần tử của ma trận (m dòng, n cột):" << endl;

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

cin >> matrix[i][j]; // Nhập từng phần tử

}

}

// Sắp xếp ma trận theo tổng các phần tử trong từng dòng

sap\_xep\_theo\_tong\_dong(matrix);

// In ra ma trận sau khi sắp xếp

cout << "Ma trận sau khi sắp xếp các dòng theo tổng các phần tử lớn hơn: " << endl;

in\_ma\_tran(matrix);

return 0;

}

**Câu 6:**

Bước 1: Nhập mảng

1. Xác định kích thước mảng:

o Nhập số nguyên n n n từ người dùng, đây là số phần tử của mảng.

2. Khởi tạo mảng:

o Tạo một mảng một chiều arr arr arr với kích thước n n n để lưu các phần tử.

3. Nhập phần tử:

o Dùng vòng lặp từ 0 đến n−1 n-1 n−1, nhập từng phần tử arr[i] arr[i] arr[i] từ người dùng.

o Giả sử tất cả các phần tử đều là số nguyên không âm (bao gồm 0).

4. Hiển thị mảng ban đầu:

o In mảng arr arr arr để kiểm tra dữ liệu đầu vào.

Bước 2: Phân loại các phần tử

1. Tạo hai danh sách tạm:

o Tạo một danh sách evens evens evens để lưu các số chẵn (không bao gồm 0).

o Tạo một danh sách odds odds odds để lưu các số lẻ.

o Kích thước tối đa của mỗi danh sách là n n n (trường hợp tất cả phần tử đều chẵn hoặc lẻ).

2. Duyệt mảng và phân loại:

o Lặp qua từng phần tử arr[i] arr[i] arr[i] từ 0 đến n−1 n-1 n−1:

 Nếu arr[i]=0 arr[i] = 0 arr[i]=0: Bỏ qua (giữ nguyên vị trí).

 Nếu arr[i]≠0 arr[i] \neq 0 arr[i]=0 và arr[i]%2=0 arr[i] \% 2 = 0 arr[i]%2=0: Thêm arr[i] arr[i] arr[i] vào danh sách evens evens evens.

 Nếu arr[i]%2=1 arr[i] \% 2 = 1 arr[i]%2=1: Thêm arr[i] arr[i] arr[i] vào danh sách odds odds odds.

3. Đếm số lượng:

o Ghi nhận số lượng phần tử trong evens evens evens (gọi là evenCount evenCount evenCount).

o Ghi nhận số lượng phần tử trong odds odds odds (gọi là oddCount oddCount oddCount).

Bước 3: Sắp xếp các danh sách

1. Sắp xếp số chẵn:

o Sắp xếp danh sách evens evens evens theo thứ tự tăng dần (từ nhỏ đến lớn).

o Ví dụ: Nếu evens=[4,2,6] evens = [4, 2, 6] evens=[4,2,6], sau khi sắp xếp sẽ là [2,4,6] [2, 4, 6] [2,4,6].

2. Sắp xếp số lẻ:

o Sắp xếp danh sách odds odds odds theo thứ tự giảm dần (từ lớn đến nhỏ).

o Ví dụ: Nếu odds=[3,1,5] odds = [3, 1, 5] odds=[3,1,5], sau khi sắp xếp sẽ là [5,3,1] [5, 3, 1] [5,3,1].

3. Giữ nguyên số 0:

o Không cần xử lý danh sách riêng cho số 0, vì chúng sẽ giữ nguyên vị trí trong mảng gốc.

Bước 4: Ghép lại mảng gốc

1. Khởi tạo chỉ số:

o Tạo biến evenIndex evenIndex evenIndex để theo dõi vị trí trong danh sách evens evens evens, ban đầu là 0.

o Tạo biến oddIndex oddIndex oddIndex để theo dõi vị trí trong danh sách odds odds odds, ban đầu là 0.

2. Duyệt mảng gốc:

o Lặp qua từng phần tử arr[i] arr[i] arr[i] từ 0 đến n−1 n-1 n−1:

 Nếu arr[i]=0 arr[i] = 0 arr[i]=0: Giữ nguyên arr[i] arr[i] arr[i] (bỏ qua bước thay thế).

 Nếu arr[i]≠0 arr[i] \neq 0 arr[i]=0 và arr[i]%2=0 arr[i] \% 2 = 0 arr[i]%2=0: Thay arr[i] arr[i] arr[i] bằng evens[evenIndex] evens[evenIndex] evens[evenIndex], tăng evenIndex evenIndex evenIndex lên 1.

 Nếu arr[i]%2=1 arr[i] \% 2 = 1 arr[i]%2=1: Thay arr[i] arr[i] arr[i] bằng odds[oddIndex] odds[oddIndex] odds[oddIndex], tăng oddIndex oddIndex oddIndex lên 1.

3. Kết quả:

o Mảng arr arr arr giờ đây đã được cập nhật với số chẵn tăng dần, số lẻ giảm dần, và số 0 giữ nguyên vị trí.

Bước 5: Hiển thị kết quả

1. In mảng sau khi sắp xếp:

Dùng vòng lặp từ 0 đến n−1 n-1 n−1, in từng phần tử arr[i] arr[i] arr[i] để kiểm tra kết quả.

Nhận xét

• Hiệu quả: Độ phức tạp là O(nlog⁡n) O(n \log n) O(nlogn) do sử dụng sắp xếp trên hai danh sách evens evens evens và odds odds odds, với kích thước tối đa là n n n.

• Ưu điểm: Đơn giản, giữ nguyên vị trí số 0 mà không cần mảng phụ phức tạp.

• Nhược điểm: Cần bộ nhớ phụ để lưu evens evens evens và odds odds odds, không tối ưu nếu n n n rất lớn.

• Mở rộng: Có thể tối ưu bằng cách sắp xếp tại chỗ (in-place) thay vì dùng mảng phụ, nhưng sẽ phức tạp hơn.

Code

#include <iostream>

#include <algorithm>

using namespace std;

void sortArray(int\* arr, int n) {

int evenCount = 0, oddCount = 0;

int\* evens = new int[n];

int\* odds = new int[n];

// Phân loại số chẵn và lẻ (bỏ qua số 0)

for (int i = 0; i < n; i++) {

if (arr[i] != 0) {

if (arr[i] % 2 == 0) evens[evenCount++] = arr[i];

else odds[oddCount++] = arr[i];

}

}

// Sắp xếp

sort(evens, evens + evenCount); // Số chẵn tăng dần

sort(odds, odds + oddCount, greater<int>()); // Số lẻ giảm dần

int evenIndex = 0, oddIndex = 0;

// Ghép lại vào mảng chính

for (int i = 0; i < n; i++) {

if (arr[i] == 0) continue; // Giữ nguyên số 0

if (arr[i] % 2 == 0) arr[i] = evens[evenIndex++];

else arr[i] = odds[oddIndex++];

}

delete[] evens;

delete[] odds;

}

void printArray(int\* arr, int n) {

for (int i = 0; i < n; i++) cout << arr[i] << " ";

cout << endl;

}

int main() {

int n;

cout << "Nhap so phan tu: ";

cin >> n;

int\* arr = new int[n];

cout << "Nhap phan tu mang: ";

for (int i = 0; i < n; i++) cin >> arr[i];

cout << "Mang ban dau: ";

printArray(arr, n);

sortArray(arr, n);

cout << " Mang sau khi sap xep: ";

printArray(arr, n);

delete[] arr;

return 0;

}

**Câu 7:**

#include<iostream>

using namespace std;

void sort(int a[], int n) {

int\* temp=new int [n];

int left = 0;

int right = n - 1;

// Duyệt qua mảng một lần

for (int i = 0; i < n; i++) {

if (a[i] % 2 == 0) {

temp[left] = a[i];

left++;

}

else {

temp[right] = a[i];

right--;

}

}

// Sao chép lại vào mảng gốc

for (int i = 0; i < n; i++) {

a[i] = temp[i];

}

}

void test() {

int n = 9;

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

sort(a, n);

for (int i = 0; i < n; i++) {

cout << a[i] << " ";

}

}

int main() {

test();

return 0;

}



**Câu 8:**

#include <iostream>

using namespace std;

// Hàm tính mảng nghịch thế

void tinhMangNghichThe(int\* A, int\* B, int N) {

for (int i = 0; i < N; i++) {

int count = 0; // Đếm số phần tử lớn hơn A[i] đứng trước nó

for (int j = 0; j < i; j++) {

if (A[j] > A[i]) {

count++;

}

}

B[A[i] - 1] = count; // Gán giá trị vào mảng B tại vị trí A[i] - 1

}

}

int main() {

// Khai báo mảng A (hoán vị của 1 đến N)

int A[] = { 5, 9, 1, 8, 2, 6, 4, 7, 3 };

int N = sizeof(A) / sizeof(A[0]); // Số phần tử trong A

// Cấp phát động mảng B

int\* B = new int[N];

// Khởi tạo mảng B với giá trị 0

for (int i = 0; i < N; i++) {

B[i] = 0;

}

// Tính mảng nghịch thế B

tinhMangNghichThe(A, B, N);

// In mảng B

cout << "Mang nghich the B: ";

for (int i = 0; i < N; i++) {

cout << B[i] << " ";

}

cout << endl;

// Giải phóng bộ nhớ đã cấp phát cho B

delete[] B;

return 0;

}

Code trên đã tạo hàm tạo mảng B nghịch thế cho A

Chạy chương trình

->

