TRƯỜNG ĐẠI HỌC SÀI GÒN

**KHOA CÔNG NGHỆ THÔNG TIN**

|  |  |
| --- | --- |
| Trần Hạnh Vy | 3124411355 |

A blue circle with text

Description automatically generated

**Mục lục:**

[I. Câu lý thuyết: 3](#_Toc192881546)

[Câu 1. Trình bày tư tưởng của các thuật toán sắp xếp? 3](#_Toc192881547)

[Câu 2. Trong các thuật toán sắp xếp, bạn thích nhất thuật toán nào? Thuật toán nào bạn không thích nhất vì sao? 5](#_Toc192881548)

[Câu 3. Trình bày và cài đặt tất cả thuật toán sắp xếp nội, ngoại theo thứ tự giảm dần. Cho nhận xét về các thuật toán này. 5](#_Toc192881549)

[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? 10](#_Toc192881550)

[II. Bài tập cơ sở: 11](#_Toc192881551)

[Bài tập 1. 11](#_Toc192881552)

[Bài tập 2. 13](#_Toc192881553)

[III. Bài tập ứng dụng: 16](#_Toc192881554)

[Bài tập 1. Cho dãy n số nguyên: a0, a1,..., an-1 16](#_Toc192881555)

[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. 18](#_Toc192881556)

[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. 20](#_Toc192881557)

[Bài tập 4. Cho ma trận hai chiều m dòng n cột, các phần tử là các số nguyên dương. 22](#_Toc192881558)

[Bài tập 5. Cho ma trận hai chiều m dòng n cột, các phần tử là các số nguyên dương. 25](#_Toc192881559)

[Bài tập 6. Cho mảng một chiều gồm n phần tử là các số nguyên không âm. 26](#_Toc192881560)

[Bài tập 7. Cho mảng một chiều gồm n phần tử là các số nguyên dương. 28](#_Toc192881561)

1. Câu lý thuyết:

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

1. Insertion Sort:

Phương pháp này được những người chơi bài hay dùng.

Giả sử cần sắp xếp tăng dần dãy khoá a1 , a2 ,..., an .

Ý tưởng thuật toán như sau:

– Các phần tử được chia thành dãy đích: a1 ,..., ai-1 (kết quả) và dãy nguồn ai,..., an.

– Bắt đầu với i=2, ở mỗi bước phần tử thứ i của dãy nguồn được lấy ra và chèn vào vị trí thích hợp trong dãy đích sao cho dãy đích vẫn tăng dần. Sau đó i tăng lên 1 và lặp lại.

1. Selection Sort:

Giả sử cần sắp xếp tăng dần một dãy khoá a1 , a2 ,..., an.

Ý tưởng của thuật toán như sau:

– Chọn phần tử có khoá nhỏ nhất .

– Đổi chỗ nó với phần tử a1 .

– Sau đó lặp lại thao tác trên với n-1 phần tử còn lại, rồi lại lặp lại như trên với n-2 phần tử còn lại,..., cho tới khi chỉ còn 1 phần tử.

1. Heap Sort:

Một cây nhị phân có chiều cao h được gọi là đống khi:

– Là cây nhị phân tương đối hoàn chỉnh mà các nút lá ở mức h phải nằm phía bên trái.

– Khoá ở nút cha bao giờ cũng lớn hơn khoá ở nút con.

Thuật toán sắp xếp vun đống chia làm 2 giai đoạn.

• Giai đoạn 1: Tạo đống ban đầu

- Từ dãy khóa ban đầu ánh xạ sang cây nhị phân

- Vun cây nhị phân từ dưới lên trên, từ cây con gốc [n/2] về cây gốc 1 để tạo đống ban đầu.

• Giai đoạn 2: Sắp xếp

- Đổi chỗ nút gốc 1 cho nút n, loại bỏ nút n, hiệu chỉnh lại (vun đống lại) cây gốc 1 với n-1 nút còn lại.

- Cứ tiếp tục như vậy cho tới khi cây chỉ còn 1 nút.

- Đưa khóa trội về vị trí ban đầu của nó bằng cách đổi chỗ cho khóa ở vị trí đó.

- Vun lại động với cây gồm các khóa còn lại (sau khi đã loại khóa trội)

Quá trình trên được lặp lại.

1. Bubble Sort:

Giả sử cần sắp xếp tăng dần dãy khoá a1 , a2 ,..., an .

Ý tưởng thuật toán như sau:

– So sánh các cặp khóa liền kề gối nhau từ phải qua trái, nếu khóa đứng sau nhỏ hơn khóa đứng trước thì đổi chỗ khóa đứng sau với khóa đứng trước. Kết quả lần thứ nhất, khóa nhỏ nhất của dãy được đẩy lên vị trí 1 (gọi là khá được sắp).

– Tiếp tục so sánh và đổi chỗ các cặp khóa liền kề của dãy chưa sắp, lần thứ 2 ta được khóa nhỏ nhất của dãy chưa sắp được đưa về vị trí 2.

– Cứ tiếp tục làm tương tự như trên cho đến khi dãy chưa sắp chỉ còn 1 phần tử.

1. Quick Sort:

Sắp xếp nhanh (quick sort) còn được sắp xếp phân đoạn (partition sort).

Ý tưởng thuật toán:

– Chọn ngẫu nhiên một phần tử x làm tiêu chí phân đoạn.

– Duyệt từ trái sang phải cho tới khi gặp phần tử ai>=x

– Sau đó duyệt từ phải sang trái cho tới khi gặp phần tử aj<=x

– Đổi chỗ ai và aj

– Tiếp tục duyệt và đổi chỗ cho tới khi 2 phía gặp nhau.

Kết quả dãy khóa được chia thành 2 đoạn: bên trái là các phần tử < x, bên phải là các phần tử > x.

Áp dụng cách tương tự với đoạn bên trái và đoạn bên phải cho tới khi đoạn con chỉ còn 1 phần tử thì dừng.

1. Merge Sort:

- Phân hoạch dãy ban đầu thành các dãy con liên tiếp mã mỗi dạy con đều có thứ tự

- Làm giảm số dãy con bằng cách trộn từng cặp dãy con của hai dãy phụ thành một dãy con của dãy ban đầu.

1. Radix Sort:

Xếp các phần tử theo thứ tự dựa vào độ lớn hàng từ đơn vị và tạo thành dãy số theo hàng đơn vị. Lấy dãy số xếp theo thứ tự hàng lớn hơn và lặp lại đến hàng lớn nhất của mảng.

1. Interchange Sort:

Duyệt qua từng phần tử có trong mảng, so sánh với các phần tử phía sau phần tử hiện tại lớn hơn phần tử sau thì hoán đổi chúng.

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

* Thuật toán thích nhất: Quick Sort - hữu dụng, dễ dàng triển khai và tối ưu. Ngoài ra còn được ứng dụng rộng rãi.
* Thuật toán không thích nhất: Bubble Sort – kém hiệu quả, thời gian thực hiện chạm vì tốn thời gian cho việc nhiều lần hoán đổi phần tử.

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

* Thuật toán sắp xếp nội:
* Insertion Sort:
* Trình bày: ta có phần đầu của mảng đã được sắp xếp, lấy phần tử chưa được sắp xếp, tìm vị trí thích hợp và chèn vào lặp lại cho đến khi toàn bộ mảng được 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 khi dữ liệu gần như đã được sắp xếp.

Ít ứng dụng thực tế do hiệu suất thấp với dữ liệu lớn.

* Selection Sort:
* Trình bày: tìm phần tử lớn nhất và hoán đổi nó với phần tử đầu, tiếp tục với phần còn lại đến khi toàn bộ được sắp xếp.
* 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]) maxIdx = j;

}

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

}

}

* Nhận xét:

Ít hoán đổi nhưng vẫn chậm

Dễ hiểu nhưng không hiệu quả khi làm việc với dữ liệu lớn.

* Bubble Sort:
* Trình bày: lặp qua mảng nhiều lần, hoán đổi nếu phần tử lớn hơn phần tử trước.
* Cài đặt:

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

}

}

}

* Nhận xét: Hiếm khi được sử dụng thực tế, vì mất thời gian cho quá trình lặp.
* Heap Sort:
* Trình bày: dùng cấu trúc Heap để luôn lấy phần tử bé để lên đầu.
* Cài đặt:

void heapify(int arr[], int n, int i) {

int smallest = i; // Đổi từ largest -> smallest

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);

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

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

heapify(arr, i, 0);

}

}

* Quick Sort:
* Trình bày: chọn chốt (privot, chia mảng thành 2 phần, giá trị lớn bên trái và giá trị nhỏ bên phải, đệ quy tiếp tục.
* 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) {

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);

}

}

* Interchane Sort:
* Trình bày: duyệt qua từng phần tử trong mảng và so sánh với tất cả các phần tử còn lại. Nếu tìm thấy phần tử lớn hơn, ta sẽ hoán đổi chúng.
* Cài đặt:

void interchangeSort(int arr[], int n) {

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

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

if (arr[i] < arr[j]) { // Sắp xếp giảm dần

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

}

}

}

}

* Nhận xét: thích hợp cho dữ liệu nhỏ.
* Thuật toán sắp xếp ngoại:
* Merge Sort:
* Trình bày: chia mảng, sắp xếp từng phần để xếp theo thứ tự giảm dần cho từng phần nhỏ rồi trộn chúng lại.
* Cài đặt:

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1, n2 = r - m;

int L[n1], R[n2];

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

for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

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

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(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);

}

}

* Nhận xét:

Dùng trong xử lý file lớn và khi cần tính ổn định.

* Radix Sort:
* Trình bày: duyệt qua các phần tử, xét hàng đơn vị giá trị nào lớn nhất xếp đầu, lặp lại quá trình với hàng chục cho đến hết phần tử có trong mảng.
* Cài đặt:

int getMax(int arr[], int n) {

int maxVal = arr[0];

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

if (arr[i] > maxVal) maxVal = arr[i];

return maxVal;

}

void countingSort(int arr[], int n, int exp) {

int output[n]; // Mảng tạm lưu kết quả

int count[10] = {0}; // Lưu số lần xuất hiện của mỗi chữ số

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

count[(arr[i] / exp) % 10]++;

for (int i = 8; i >= 0; i--) // Đảo ngược để sắp xếp giảm dần

count[i] += count[i + 1];

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

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

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

arr[i] = output[i];

}

void radixSort(int arr[], int n) {

int maxVal = getMax(arr, n);

for (int exp = 1; maxVal / exp > 0; exp \*= 10)

countingSort(arr, n, exp);

}

* Nhận xét:

**Bị hạn chế** bởi phạm vi dữ liệu và yêu cầu bộ nhớ. Nếu dữ liệu phù hợp (số nguyên, ít chữ số).

## 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?

|  |  |  |  |
| --- | --- | --- | --- |
| **Tên thuật toán** | **Ưu điểm** | **Nhược điểm** | **Cách khắc phục** |
| Insertion Sort | Ổn định, giữ nguyên thứ tự các phần tử có giá trị bằng nhau. | Chậm khi mảng có nhiều phần tử và không được sắp xếp trước | Khi dữ liệu lớn, sử dụng Merge Sort hoặc Quick Sort |
| Selection Sort | Không bị ảnh hưởng bởi thứ tự ban đầu | Chậm, không tối ưu dữ liệu lớn  Luôn thực hiện phép so sánh, ngay cả khi mảng đã gần như được sắp xếp | Nếu mảng nhỏ có thể dùng Insertion Sort thay cho Selection Sort |
| Heap Sort | Không cần bộ nhớ phụ như Merge Sort  Hoạt động tốt với dữ liệu có kích thước lớn | Không ổn định (có thể làm thay đổi thứ tự của các phần tử bằng nhau)  Hiệu suất thực tế có thể chậm hơn Quick Sort do nhiều phép hoán đổi hơn | Nếu cần sắp xếp ổn định, dùng Merge Sort thay thế |
| Bubble Sort | Dễ hiểu, dễ triển khai  Có thể dừng sớm nếu phát hiện mảng đã được sắp xếp | Chậm khi dữ liệu lớn | Sử dụng kĩ thuật “cầm cờ” để giảm số lần so sánh |
| Quick Sort | Hiệu suất cao hơn Merge Sort trong thực tế do có ít phép gán hơn | Trường hợp xấu nếu chọn pivot không tốt (khi mảng đã sắp xếp hoặc có nhiều phần tử giống nhau)  Không ổn định, có thể thay đổi thứ tự các phần tử bằng nhau | Đặt những câu lệnh điều kiện để tranh trường hợp chọn phần tử pivot trùng với bất kì phần tử nào đó có trong mảng |
| Merge Sort | Phù hợp với dữ liệu lớn  Không bị ảnh hưởng bởi thứ tự ban đầu của dữ liệu | Tốn bộ nhớ phụ vì chia mảng thành các phần nhỏ và ghép lại | Nếu cần tối ưu bộ nhớ, có thể dùng Merge Sort kết hợp với Heap Sort trong một số trường hợp |
| Radix Sort | Ổn định, giữ nguyên thứ tự các phần tử bằng nhau | Chỉ hoạt động tốt với số nguyên, không áp dụng được cho dữ liệu tổng quát  Tốn bộ nhớ phụ, không thích hợp cho dữ liệu lớn | Khi cần tối ưu bộ nhớ  Nếu cần sắp xếp số thực hoặc chuỗi, dùng Merge Sort thay vì Radix Sort |
| Interchange Sort | Dễ cài đặt, dễ hiểu  Không yêu cầu bộ nhớ phụ | Rất chậm kém hiệu quả khi dữ liệu lớn  Nhiều lần hoán đổi không cần thiết | Dùng Quick Sort, Merge Sort hoặc Heap Sort thay thế. |

1. Bài tập cơ sở:

## Bài tập 1.

1. Mô phỏng các bước sắp xếp
2. Interchange Sort

Ý tưởng: Duyệt từng cặp phần tử, nếu phần tử trước lớn hơn phần tử sau thì đổi chỗ.

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

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

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

...

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

Độ phức tạp: O(n2)

1. Selection Sort

Ý tưởng: Chọn phần tử nhỏ nhất trong dãy chưa được sắp xếp và đổi chỗ với phần tử đầu tiên của vùng chưa sắp xếp.

Tìm min trong (39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10) → 1

Đổi chỗ 1 ↔ 39 → (1, 8, 5, 39, 3, 6, 9, 12, 4, 7, 10)

...

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

Độ phức tạp: O(n2)

1. Insertion Sort

Ý tưởng: Lấy từng phần tử và chèn vào đúng vị trí trong phần đã sắp xếp.

Lấy 8, chèn vào vị trí đúng → (8, 39, 5, 1, 3, 6, 9, 12, 4, 7, 10)

Lấy 5, chèn vào vị trí đúng → (5, 8, 39, 1, 3, 6, 9, 12, 4, 7, 10)

...

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

Độ phức tạp: O(n2)

1. Bubble Sort

Ý tưởng: So sánh từng cặp liên tiếp, nếu sai thứ tự thì đổi chỗ.

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

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

...

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

Độ phức tạp: O(n2)

1. Cài đặt C++

#include <iostream>

using namespace std;

void interchangeSort(int arr[], int n) {

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

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

if (arr[i] > arr[j])

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

}

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

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

int minIndex = i;

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

if (arr[j] < arr[minIndex])

minIndex = j;

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

}

}

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

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])

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

}

void printArray(int arr[], int n) {

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

cout << arr[i] << " ";

cout << endl;

}

int main() {

int arr[] = {39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Mang ban dau: ";

printArray(arr, n);

int arr1[] = {39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

interchangeSort(arr1, n);

cout << "Interchange Sort: ";

printArray(arr1, n);

int arr2[] = {39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

selectionSort(arr2, n);

cout << "Selection Sort: ";

printArray(arr2, n);

int arr3[] = {39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

insertionSort(arr3, n);

cout << "Insertion Sort: ";

printArray(arr3, n);

int arr4[] = {39, 8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

bubbleSort(arr4, n);

cout << "Bubble Sort: ";

printArray(arr4, n);

return 0;

}

1. Chứng minh độ phức tạp

Tất cả các thuật toán trên có độ phức tạp **O(n²)** trong trường hợp trung bình và xấu nhất. Chỉ có **Insertion Sort** có thể đạt **O(n)** trong trường hợp mảng đã gần như sắp xếp.

## Bài tập 2.

1. Mô phỏng các bước sắp xếp
2. Quick Sort

Ý tưởng:

* Chọn một phần tử làm **pivot**.
* Chia mảng thành hai phần: phần nhỏ hơn pivot và phần lớn hơn pivot.
* Áp dụng đệ quy cho từng phần.

Chia mảng: (5, 1, 3, 4) | 6 | (8, 9, 12, 7, 10)

Sắp xếp đệ quy hai phần:

(1, 3, 4, 5) | 6 | (7, 8, 9, 10, 12)

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

Độ phức tạp trung bình: O(n log n).

1. Merge Sort

Ý tưởng:

* Chia đôi mảng cho đến khi mỗi phần chỉ có một phần tử.
* Trộn (merge) các phần tử đã sắp xếp.

Chia: (8, 5, 1, 3, 6) | (9, 12, 4, 7, 10)

Chia tiếp: (8, 5) | (1, 3, 6) và (9, 12) | (4, 7, 10)

Trộn dần lại: (5, 8) | (1, 3, 6) và (9, 12) | (4, 7, 10)

...

Kết quả cuối cùng:

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

Độ phức tạp: O(n log n)

1. Heap Sort

Ý tưởng:

* Xây dựng cây **Max Heap**.
* Đưa phần tử lớn nhất về cuối, sau đó tiếp tục xây dựng lại heap.

Xây dựng max heap: (12, 10, 9, 7, 8, 1, 5, 3, 4, 6)

Đưa phần tử lớn nhất (12) về cuối và heapify lại.

Lặp lại quá trình trên đến khi mảng được sắp xếp.

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

Độ phức tạp: O(n log n).

1. Cài đặt C++

#include <iostream>

using namespace std;

void quickSort(int arr[], int left, int right) {

if (left >= right) return;

int pivot = arr[right], i = left - 1;

for (int j = left; j < right; j++) {

if (arr[j] < pivot)

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

}

swap(arr[++i], arr[right]);

quickSort(arr, left, i - 1);

quickSort(arr, i + 1, right);

}

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

int n1 = mid - left + 1, n2 = right - mid;

int L[n1], R[n2];

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

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

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

while (i < n1 && j < n2) {

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(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);

}

void heapify(int arr[], int n, int i) {

int largest = i, left = 2 \* i + 1, 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(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);

}

}

void printArray(int arr[], int n) {

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

cout << arr[i] << " ";

cout << endl;

}

int main() {

int arr[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Mang ban dau: ";

printArray(arr, n);

int arr1[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

quickSort(arr1, 0, n - 1);

cout << "Quick Sort: ";

printArray(arr1, n);

int arr2[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

mergeSort(arr2, 0, n - 1);

cout << "Merge Sort: ";

printArray(arr2, n);

int arr3[] = {8, 5, 1, 3, 6, 9, 12, 4, 7, 10};

heapSort(arr3, n);

cout << "Heap Sort: ";

printArray(arr3, n);

return 0;

}

1. Chứng minh độ phức tạp

* Quick Sort nhanh nhất trong thực tế nhưng có thể bị chậm nếu chọn pivot kém.
* Merge Sort ổn định và không phụ thuộc vào dữ liệu ban đầu.
* Heap Sort tốt nhưng có chi phí heapify cao.

1. Bài tập ứng dụng:

## Bài tập 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.

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.

#include <iostream>

#include <vector>

#include <algorithm>

#include <cmath>

using namespace std;

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

vector<int> findLargestElementsPositions(vector<int>& arr, int k) {

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

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

indices[i] = i;

}

sort(indices.begin(), indices.end(), [&](int i, int j) {

return arr[i] > arr[j]; // Sắp xếp giảm dần theo giá trị

});

indices.resize(k); // Giữ lại k phần tử lớn nhất

return indices;

}

// Hàm tính tổng các chữ số của một số nguyên

int digitSum(int num) {

int sum = 0;

num = abs(num); // Đảm bảo xử lý số âm đúng

while (num > 0) {

sum += num % 10;

num /= 10;

}

return sum;

}

// Hàm sắp xếp theo tổng các chữ số

void sortByDigitSum(vector<int>& arr) {

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

return digitSum(a) < digitSum(b);

});

}

// Hàm kiểm tra số nguyên tố

bool isPrime(int num) {

if (num < 2) return false;

for (int i = 2; i \* i <= num; i++) {

if (num % i == 0) return false;

}

return true;

}

// Hàm xóa tất cả số nguyên tố trong dãy

void removePrimes(vector<int>& arr) {

arr.erase(remove\_if(arr.begin(), arr.end(), isPrime), arr.end());

}

// Hàm in mảng

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

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

int main() {

vector<int> arr = {23, 5, 17, 40, 12, 8, 31, 15, 7, 9};

int k = 3;

// a) Tìm vị trí k phần tử lớn nhất

vector<int> largestPositions = findLargestElementsPositions(arr, k);

cout << "Vi tri " << k << " phan tu lon nhat: ";

for (int pos : largestPositions) {

cout << pos << " ";

}

cout << endl;

// b) Sắp xếp theo tổng các chữ số

sortByDigitSum(arr);

cout << "Day sau khi sap xep theo tong chu so: ";

printArray(arr);

// c) Xóa số nguyên tố

removePrimes(arr);

cout << "Day sau khi xoa so nguyen to: ";

printArray(arr);

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.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Định nghĩa cấu trúc dữ liệu cho số hạng của đa thức

struct Term {

double coefficient; // Hệ số

int exponent; // Bậc

// Hàm tạo để khởi tạo giá trị

Term(double c, int e) : coefficient(c), exponent(e) {}

};

// Hàm so sánh để sắp xếp theo bậc tăng dần

bool compareByExponent(const Term& a, const Term& b) {

return a.exponent < b.exponent;

}

// Hàm nhập dữ liệu

void inputPolynomial(vector<Term>& polynomial, int n) {

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

double coefficient;

int exponent;

cout << "Nhap he so va bac cua so hang thu " << i + 1 << ": ";

cin >> coefficient >> exponent;

polynomial.push\_back(Term(coefficient, exponent));

}

}

// Hàm hiển thị đa thức

void displayPolynomial(const vector<Term>& polynomial) {

for (const auto& term : polynomial) {

cout << term.coefficient << "x^" << term.exponent << " ";

}

cout << endl;

}

int main() {

int n;

cout << "Nhap so luong so hang cua da thuc: ";

cin >> n;

vector<Term> polynomial;

inputPolynomial(polynomial, n);

// Sắp xếp theo bậc tăng dần

sort(polynomial.begin(), polynomial.end(), compareByExponent);

// Xuất kết quả sau khi sắp xếp

cout << "Da thuc sau khi sap xep theo bac tang dan: ";

displayPolynomial(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.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Định nghĩa cấu trúc dữ liệu cho phòng thi

struct ExamRoom {

int roomNumber; // Số phòng

char building; // Nhà (A-Z)

int capacity; // Khả năng chứa

// Hàm khởi tạo

ExamRoom(int num, char b, int cap) : roomNumber(num), building(b), capacity(cap) {}

};

// Hàm so sánh để sắp xếp theo khả năng chứa giảm dần

bool compareByCapacity(const ExamRoom& a, const ExamRoom& b) {

return a.capacity > b.capacity;

}

// Hàm so sánh theo nhà (tăng dần), nếu cùng nhà thì sắp xếp theo số phòng (tăng dần)

bool compareByBuildingAndRoom(const ExamRoom& a, const ExamRoom& b) {

if (a.building == b.building)

return a.roomNumber < b.roomNumber;

return a.building < b.building;

}

// Hàm so sánh theo nhà (tăng dần), nếu cùng nhà thì sắp xếp theo khả năng chứa (giảm dần)

bool compareByBuildingAndCapacity(const ExamRoom& a, const ExamRoom& b) {

if (a.building == b.building)

return a.capacity > b.capacity;

return a.building < b.building;

}

// Hàm nhập danh sách phòng thi

void inputExamRooms(vector<ExamRoom>& rooms, int n) {

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

int roomNumber, capacity;

char building;

cout << "Nhap so phong, nha (A-Z) va kha nang chua: ";

cin >> roomNumber >> building >> capacity;

rooms.push\_back(ExamRoom(roomNumber, building, capacity));

}

}

// Hàm hiển thị danh sách phòng thi

void displayExamRooms(const vector<ExamRoom>& rooms) {

for (const auto& room : rooms) {

cout << "Phong " << room.roomNumber << " - Nha " << room.building

<< " - Kha nang chua: " << room.capacity << endl;

}

}

int main() {

int n;

cout << "Nhap so luong phong thi: ";

cin >> n;

vector<ExamRoom> rooms;

inputExamRooms(rooms, n);

// Sắp xếp theo khả năng chứa giảm dần

sort(rooms.begin(), rooms.end(), compareByCapacity);

cout << "\nDanh sach phong theo kha nang chua giam dan:\n";

displayExamRooms(rooms);

// Sắp xếp theo nhà (tăng dần), cùng nhà thì theo số phòng (tăng dần)

sort(rooms.begin(), rooms.end(), compareByBuildingAndRoom);

cout << "\nDanh sach phong theo nha (A-Z), cung nha theo so phong tang dan:\n";

displayExamRooms(rooms);

// Sắp xếp theo nhà (tăng dần), cùng nhà thì theo khả năng chứa (giảm dần)

sort(rooms.begin(), rooms.end(), compareByBuildingAndCapacity);

cout << "\nDanh sach phong theo nha (A-Z), cung nha theo kha nang chua giam dan:\n";

displayExamRooms(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.

#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;

}

// Hàm tìm số nguyên tố lớn nhất trong ma trận

int findMaxPrime(const vector<vector<int>>& matrix, int m, int n) {

int maxPrime = -1; // Nếu không có số nguyên tố nào thì trả về -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;

}

// Hàm tìm các dòng chứa ít nhất một số nguyên tố

vector<int> findRowsWithPrimes(const vector<vector<int>>& matrix, int m, int n) {

vector<int> rows;

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

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

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

rows.push\_back(i);

break;

}

}

}

return rows;

}

// Hàm tìm các dòng chỉ chứa toàn số nguyên tố

vector<int> findRowsWithOnlyPrimes(const vector<vector<int>>& matrix, int m, int n) {

vector<int> rows;

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) rows.push\_back(i);

}

return rows;

}

// Hàm nhập ma trận

void inputMatrix(vector<vector<int>>& matrix, int m, int n) {

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];

}

}

}

// Hàm hiển thị danh sách các dòng

void displayRows(const vector<int>& rows) {

if (rows.empty()) {

cout << "Khong co dong nao thoa man.\n";

} else {

cout << "Cac dong: ";

for (int row : rows) {

cout << row << " ";

}

cout << endl;

}

}

int main() {

int m, n;

cout << "Nhap so dong (m) va so cot (n): ";

cin >> m >> n;

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

inputMatrix(matrix, m, n);

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

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

if (maxPrime == -1)

cout << "Ma tran khong co so nguyen to.\n";

else

cout << "So nguyen to lon nhat trong ma tran: " << maxPrime << endl;

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

cout << "\nNhung dong chua it nhat mot so nguyen to:\n";

displayRows(findRowsWithPrimes(matrix, m, n));

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

cout << "\nNhung dong chi chua cac so nguyen to:\n";

displayRows(findRowsWithOnlyPrimes(matrix, m, n));

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.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Hàm tính tổng của một dòng trong ma trận

int rowSum(const vector<int>& row) {

int sum = 0;

for (int num : row) {

sum += num;

}

return sum;

}

// Hàm tìm dòng có tổng lớn nhất

int findMaxSumRow(const vector<vector<int>>& matrix, int m, int n) {

int maxSum = -1, maxRowIndex = -1;

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

int sum = rowSum(matrix[i]);

if (sum > maxSum) {

maxSum = sum;

maxRowIndex = i;

}

}

return maxRowIndex;

}

// Hàm sắp xếp các dòng theo tổng giảm dần

void sortRowsBySum(vector<vector<int>>& matrix, int m, int n) {

sort(matrix.begin(), matrix.end(), [](const vector<int>& row1, const vector<int>& row2) {

return rowSum(row1) > rowSum(row2); // Sắp xếp giảm dần theo tổng dòng

});

}

// Hàm nhập ma trận

void inputMatrix(vector<vector<int>>& matrix, int m, int n) {

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];

}

}

}

// Hàm in ma trận

void printMatrix(const vector<vector<int>>& matrix) {

for (const auto& row : matrix) {

for (int num : row) {

cout << num << " ";

}

cout << endl;

}

}

int main() {

int m, n;

cout << "Nhap so dong (m) va so cot (n): ";

cin >> m >> n;

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

inputMatrix(matrix, m, n);

// a. Tìm dòng có tổng lớn nhất

int maxRowIndex = findMaxSumRow(matrix, m, n);

cout << "Dong co tong lon nhat la dong: " << maxRowIndex << endl;

// b. Sắp xếp các dòng theo tổng giảm dần

sortRowsBySum(matrix, m, n);

cout << "\nMa tran sau khi sap xep theo tong giam dan:\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.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Hàm sắp xếp mảng theo yêu cầu

void sortArray(vector<int>& arr) {

vector<int> even, odd; // Danh sách chứa số chẵn và số lẻ

vector<int> zeroPositions; // Lưu vị trí số 0

// Phân loại phần tử

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

if (arr[i] == 0) {

zeroPositions.push\_back(i); // Ghi nhớ vị trí số 0

} else if (arr[i] % 2 == 0) {

even.push\_back(arr[i]); // Số chẵn

} else {

odd.push\_back(arr[i]); // Số lẻ

}

}

// Sắp xếp số chẵn tăng dần và số lẻ giảm dần

sort(even.begin(), even.end());

sort(odd.rbegin(), odd.rend());

// Ghép lại thành mảng ban đầu

int evenIndex = 0, oddIndex = 0;

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

if (find(zeroPositions.begin(), zeroPositions.end(), i) != zeroPositions.end()) {

arr[i] = 0; // Giữ nguyên số 0

} else if (evenIndex < even.size()) {

arr[i] = even[evenIndex++]; // Lấy số chẵn đã sắp xếp

} else {

arr[i] = odd[oddIndex++]; // Lấy số lẻ đã sắp xếp

}

}

}

// Hàm main để chạy thử chương trình

int main() {

vector<int> arr = {3, 0, 2, 8, 5, 0, 1, 4, 7, 0, 6}; // Ví dụ mảng đầu vào

cout << "Mảng ban đầu: ";

for (int num : arr) cout << num << " ";

cout << endl;

sortArray(arr);

cout << "Mảng sau khi sắp xếp: ";

for (int num : arr) cout << num << " ";

cout << endl;

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).

#include <iostream>

#include <vector>

using namespace std;

// Hàm sắp xếp chẵn trước, lẻ sau với độ phức tạp O(n)

void separateEvenOdd(vector<int>& arr) {

int left = 0, right = arr.size() - 1;

while (left < right) {

// Nếu phần tử đầu là số chẵn, tiếp tục duyệt sang phải

if (arr[left] % 2 == 0) {

left++;

}

// Nếu phần tử cuối là số lẻ, tiếp tục duyệt sang trái

else if (arr[right] % 2 == 1) {

right--;

}

// Nếu arr[left] là số lẻ và arr[right] là số chẵn -> hoán đổi vị trí

else {

swap(arr[left], arr[right]);

left++;

right--;

}

}

}

// Hàm main để chạy thử chương trình

int main() {

vector<int> arr = {3, 8, 5, 2, 7, 4, 6, 9, 10}; // Ví dụ mảng đầu vào

cout << "Mảng ban đầu: ";

for (int num : arr) cout << num << " ";

cout << endl;

separateEvenOdd(arr);

cout << "Mảng sau khi sắp xếp (chẵn trước, lẻ sau): ";

for (int num : arr) cout << num << " ";

cout << endl;

return 0;

}

Thưa thầy,

Tuần này em có nhiều bài thi và chưa tìm được nhóm làm chung nên không kịp hoàn thành bài tập. Em mong thầy có thể xem xét và nhẹ tay cho em. Mong thầy thông cảm ạ.

Em cảm ơn thầy!