1. Cài đặt và tìm hiểu luật Horner cho đa thức
   1. Mô tả bài toán
      1. Đầu vào

Một đa thức bậc n với hệ số là một mảng các số thực và giá trị x.

* + 1. Đầu ra

Giá trị của đa thức tại x, được tính bằng phương pháp Horner.

* + 1. Cách xử lý

Phương pháp Horner là một phương pháp hiệu quả để tính giá trị của một đa thức tại một giá trị x cụ thể. Ý tưởng của phương pháp này là chuyển đổi đa thức ban đầu thành một dãy các phép tính nhân và cộng. Sau đó, thực hiện các phép tính này với giá trị x đưa vào để tính giá trị của đa thức tại x đó.

* 1. Hướng dẫn thực thi chương trình
* Nhập bậc của đa thức và các hệ số của nó từ bàn phím.
* Nhập giá trị x cần tính giá trị đa thức tại đó từ bàn phím.
* Thực hiện tính giá trị của đa thức tại x bằng phương pháp Horner.
* In kết quả ra màn hình.
  1. Source code

#include <iostream>

#include <cmath>

using namespace std;

// Ham tinh gia tri cua da thuc tai x bang phuong phap Horner

double horner(double a[], int n, double x) {

double result = a[n]; // Gan result bang he so bac n cua da thuc

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

result = result \* x + a[i]; // Ap dung cong thuc Horner

}

return result;

}

int main() {

int n = 0 ;

double x;

cout << "Nhap bac cua da thuc: ";

cin >> n;

double a[100];

cout << "Nhap cac he so cua da thuc: ";

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

cin >> a[i];

}

cout << "Nhap gia tri x: ";

cin >> x;

double result = horner(a, n, x);

cout << "Gia tri cua da thuc tai x = " << x << " la: " << result << endl;

return 0;

}

1. Cài đặt thuật toán quicksort
   1. Mô tả thuật toán
      1. Đầu vào

Mảng chứa các phần tử cần được sắp xếp.

* + 1. Đầu ra

Mảng các phần tử được sắp sếp.

* + 1. Cách xử lý
* Hàm kiểm tra nếu chỉ số trái (left) lớn hơn hoặc bằng chỉ số phải (right) thì thoát khỏi hàm. Ngược lại, hàm chọn một phần tử chính giữa trong mảng (a) được đánh dấu bởi chỉ số (left + right) / 2. Phần tử này được sử dụng để chia mảng thành hai phần: các phần tử nhỏ hơn phần tử chính giữa và các phần tử lớn hơn.
* Khởi tạo hai biến i và j lần lượt là chỉ số trái (left - 1) và chỉ số phải (right + 1).Tiếp theo, hàm sử dụng vòng lặp while để tìm các phần tử trong mảng cần được đổi chỗ để đảm bảo các phần tử bên trái nhỏ hơn phần tử chính giữa và các phần tử bên phải lớn hơn phần tử chính giữa.
* Trong vòng lặp, biến i được tăng dần đến khi gặp phần tử lớn hơn hoặc bằng phần tử chính giữa, biến j được giảm dần đến khi gặp phần tử nhỏ hơn hoặc bằng phần tử chính giữa.
* Nếu i < j, hai phần tử được đổi chỗ cho nhau.
* Hàm gọi đệ quy quickSort để sắp xếp phân đoạn từ left đến j (phần tử nhỏ hơn phần tử chính giữa).
* Sau đó, hàm gọi đệ quy quickSort để sắp xếp phân đoạn từ j+1 đến right (phần tử lớn hơn phần tử chính giữa).

Quá trình đệ quy tiếp tục cho đến khi tất cả các phân đoạn đều được sắp xếp.

* 1. Hướng dẫn thực thi chương trình
* Nhập vào số lượng phần tử trong mảng.
* Nhập vào các phần tử trong mảng.
* In ra mảng đã được sắp xếp.
  1. Source code

#include <iostream>

using namespace std;

template <class DataType>

// Ham quicksort

void quickSort(DataType a[], int left, int right)

{

if (left >= right) {

return;

}

DataType x = a[(left + right) / 2];

int i = left - 1, j = right + 1;

while (i < j) {

do { i++; } while (a[i] < x);

do { j--; } while (a[j] > x);

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

}

quickSort(a, left, j);

quickSort(a, j + 1, right);

}

int main()

{

int n;

cout << "Nhap so luong phan tu : ";

cin >> n;

int arr[n];

cout << "Nhap mang arr:";

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

cin >> arr[i];

quickSort(arr, 0, n - 1);

cout << "Sorted array: ";

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

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

cout << endl;

return 0;

}

1. Cài đặt bài toán Tháp Hà Nội
   1. Mô tả thuật toán
      1. Đầu vào

Số lượng đĩa n (n > 0) và ba cột A, B, C.

* + 1. Đầu ra

Thứ tự các bước di chuyển để chuyển toàn bộ n đĩa từ cột A sang cột C.

* + 1. Cách xử lý
* Bước 1: Nếu n = 1, di chuyển đĩa đó từ cột A sang cột C và kết thúc.
* Bước 2: Di chuyển n-1 đĩa từ cột A sang cột B bằng cách sử dụng cột C như một trung gian.
* Bước 3: Di chuyển đĩa còn lại từ cột A sang cột C.
* Bước 4: Di chuyển n-1 đĩa từ cột B sang cột C bằng cách sử dụng cột A như một trung gian.
  1. Hướng dẫn thực thi chương trình
* Nhập số lượng đĩa n (n > 0).
* Gọi hàm hanoi(n, 'A', 'B', 'C') để tìm thứ tự các bước di chuyển để chuyển n đĩa từ cột A sang cột C.
* Hiển thị thứ tự các bước di chuyển.
  1. Source code

#include <iostream>

using namespace std;

void hanoi(int n, char from, char to, char aux)

{

if (n == 1)

{

cout << "Di chuyển từ " << from << " đến " << to << endl;

return;

}

hanoi(n - 1, from, aux, to);

cout << "Di chuyển đĩa " << n << " từ " << from << " đến " << to << endl;

hanoi(n - 1, aux, to, from);

}

int main()

{

int n;

cout << "Nhập sl đĩa n: ";

cin >> n;

cout << "Các bước di chuyển của " << n << endl;

hanoi(n, 'A', 'C', 'B');

return 0;

}

1. Cài đặt bài toán Balo 1
   1. Mô tả thuật toán
      1. Đầu vào

* W: số nguyên dương thể hiện khối lượng tối đa của balo.
* n: số nguyên dương thể hiện số lượng vật phẩm.
* w: mảng số nguyên dương thể hiện khối lượng của các vật phẩm.
* v: mảng số nguyên dương thể hiện giá trị của các vật phẩm.
  + 1. Đầu ra

Tổng giá trị lớn nhất của các đồ vật có thể chứa trong balo có khối lượng tối đa là W.

* + 1. Cách xử lý
* Sử dụng thuật toán Quy hoạch động để giải quyết bài toán.
* Tạo một mảng F có kích thước (n+1)x(W+1), F[i][j] sẽ là giá trị tối đa có thể chứa với j khối lượng và i vật phẩm.
* Khởi tạo giá trị cho F[0][j] và F[i][0] bằng 0, vì khi khối lượng của balo bằng 0 hoặc không có vật phẩm nào thì giá trị sẽ là 0.
* Sử dụng vòng lặp for lồng nhau để tính giá trị cho các ô còn lại trong mảng F, theo công thức: F[i][j] = max(F[i-1][j], v[i-1] + F[i-1][j-w[i-1]]) nếu j >= w[i-1], ngược lại thì F[i][j] = F[i-1][j].
* Kết quả sẽ là giá trị F[n][W].
* Sắp xếp các đồ vật theo thứ tự giá trị giảm dần
* Dùng một mảng F có kích thước (n+1) x (W+1), trong đó F[i][j] là giá trị lớn nhất mà ta có thể đạt được nếu sử dụng các đồ vật từ đầu đến vật thứ i với khối lượng không vượt quá j.

* 1. Hướng dẫn thực thi chương trình
* Nhập số nguyên dương W thể hiện khối lượng tối đa của balo.
* Nhập số nguyên dương n thể hiện số lượng vật phẩm.
* Nhập mảng số nguyên dương w thể hiện khối lượng của các vật phẩm.
* Nhập mảng số nguyên dương v thể hiện giá trị của các vật phẩm.
* Chạy chương trình và xuất kết quả tối đa của giá trị các vật phẩm có thể chứa trong balo.
  1. Source code

#include<bits/stdc++.h>

using namespace std;

void Balo1(int n, int W, int F[][100], int w[], int v[]) {

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

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

if (i == 0 || j == 0) {

F[i][j] = 0;

}

else if (w[i - 1] <= j) {

F[i][j] = max(F[i - 1][j], v[i - 1] + F[i - 1][j - w[i - 1]]);

}

else {

F[i][j] = F[i - 1][j];

}

}

}

cout << F[n][W] << endl;

}

int main() {

int W, n;

cin >> W >> n;

int w[n], v[n];

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

cin >> w[i] >> v[i];

}

int F[n + 1][100];

Balo1(n, W, F, w, v);

return 0;

}

1. Cài đặt bài toán tìm đường đi ngắn nhất (Dijkstra)
   1. Mô tả thuật toán
      1. Đầu vào

* Đồ thị có trọng số và hướng.
* Một đỉnh nguồn s.
  + 1. Đầu ra
* Đường đi ngắn nhất từ đỉnh nguồn s đến tất cả các đỉnh còn lại trong đồ thị.
* Trọng số của đường đi ngắn nhất từ s đến mỗi đỉnh trong đồ thị.
  + 1. Cách xử lý
* Khởi tạo đường đi ngắn nhất từ s đến tất cả các đỉnh trong đồ thị bằng vô cùng.
* Khởi tạo đường đi ngắn nhất từ s đến chính nó bằng 0.
* Lặp lại cho đến khi tất cả các đỉnh đều được duyệt:
* Tìm một đỉnh chưa được duyệt có đường đi ngắn nhất từ s đến nó là nhỏ nhất.
* Đánh dấu đỉnh này là đã được duyệt.
* Cập nhật đường đi ngắn nhất từ s đến tất cả các đỉnh kề của đỉnh vừa được duyệt.
* Lưu lại đỉnh trước đỉnh vừa được cập nhật vào mảng previous
  1. Hướng dẫn thực thi chương trình
* Hướng dẫn thực thi chương trình:
* Nhập đồ thị có trọng số và hướng.
* Nhập đỉnh nguồn s.
* Thực hiện t.
  1. Source code

#include <iostream>

#include <climits>

using namespace std;

const int MAX = 100;

void dijkstra(int graph[MAX][MAX], int n, int s) {

int dist[MAX]; // Lưu trữ khoảng cách ngắn nhất từ s đến các đỉnh trong đồ thị

bool visited[MAX] = { false }; // Đánh dấu các đỉnh đã được duyệt

int previous[MAX]; // Lưu trữ đỉnh trước đỉnh hiện tại trên đường đi ngắn nhất

// Khởi tạo các giá trị ban đầu cho dist và previous

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

dist[i] = INT\_MAX;

previous[i] = -1;

}

dist[s] = 0;

// Thực hiện thuật toán Dijkstra

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

// Tìm đỉnh chưa được duyệt có đường đi ngắn nhất từ s đến nó là nhỏ nhất

int u = -1;

int minDist = INT\_MAX;

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

if (!visited[v] && dist[v] < minDist) {

u = v;

minDist = dist[v];

}

}

// Đánh dấu đỉnh đã được duyệt

visited[u] = true;

// Cập nhật đường đi ngắn nhất từ s đến các đỉnh kề của đỉnh u

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

if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] +graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

previous[v] = u;

}

}

}

// In ra đường đi ngắn nhất từ s đến các đỉnh trong đồ thị và trọng số của chúng

cout << "Đường đi ngắn nhất đến từ " << s << ":\n";

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

if (i != s) {

cout << "Vertex " << i << ": ";

if (dist[i] != INT\_MAX) {

cout << dist[i] << " (";

int j = i;

while (j != s) {

cout << j << " <- ";

j = previous[j];

}

cout << s << ")\n";

}

else {

cout << "Không có đường đi.\n";

}

}

}

}

int main() {

int n; // Số đỉnh trong đồ thị

int graph[MAX][MAX]; // Ma trận kề của đồ thị

int s; // Đỉnh nguồn

// Nhập thông tin về đồ thị

cout << "Nhập số đỉnh: ";

cin >> n;

cout << "Nhập ma trận kề\n";

// Nhập ma trận kề của đồ thị

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

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

cin >> graph[i][j];

}

}

// Nhập đỉnh nguồn

cout << "Nhập đỉnh dừng: ";

cin >> s;

// Tìm đường đi ngắn nhất từ đỉnh nguồn

dijkstra(graph, n, s);

return 0;

}