**TRƯỜNG ĐẠI HỌC SƯ PHẠM THÀNH PHỐ HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ THÔNG TIN**

**….……o0o……….**



**BÀI THỰC HÀNH**

**MÔN: PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT**

Họ tên sinh viên:  **Kim Trọng Duy**

MSSV:  **47.01.103.038**

Lớp học phần: **COMP140104**

*TP. Hồ Chí Minh, ngày 29 tháng 7 năm 2022*

**MỤC LỤC**

**NỘI DUNG**

**BÀI 1: CÀI ĐẶT VÀ TÌM HIỂU LUẬT HORNER CHO ĐA THỨC**

1. **Tìm hiểu bài toán**

Luật Horner là một phương pháp được sử dụng để chia đa thức thành các nhân tử bậc một và một đa thức bậc nhỏ hơn. Phương pháp này được đặt theo tên của nhà toán học người Anh William Horner.

Phương pháp này thực hiện việc chia đa thức P(x) cho (x-a), trong đó a là một số thực nào đó. Kết quả của phép chia được biểu diễn dưới dạng:

P(x) = (x - a)Q(x) + R

Trong đó Q(x) là đa thức thương, R là phần dư. Để tìm Q(x) và R, ta có thể áp dụng phương pháp Horner như sau:

* Sắp xếp các hệ số của đa thức P(x) theo thứ tự giảm dần của bậc.
* Thực hiện các phép tính Horner để tìm Q(x) và R.

Việc thực hiện các phép tính Horner sẽ giúp ta nhanh chóng tính được giá trị của Q(x) và R mà không cần thực hiện các phép chia và nhân đa thức.

1. **Cài đặt**

#include <iostream>

using namespace std;

// Hàm tính giá trị của đa thức tại x, sử dụng thuật toán Horner

double evaluatePolynomial(double coefficients[], int degree, double x) {

double result = coefficients[degree];

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

result = coefficients[i] + x \* result;

}

return result;

}

int main() {

int degree;

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

cin >> degree;

double coefficients[degree + 1];

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

for (int i = degree; i >= 0; i--) {

cin >> coefficients[i];

}

double x;

cout << "Nhap gia tri cua x: ";

cin >> x;

double result = evaluatePolynomial(coefficients, degree, x);

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

return 0;

}

**3.Hướng dẫn thực thi**

Thuật toán cho phép người dùng nhập bậc của đa thức, các hệ số của đa thức theo thứ tự lần lượt, giá trị của biến. Thuật toán sẽ xuất ra giá trị của biểu thức tại x bằng giá trị đã nhập.

~ ~ ~ o0o ~ ~ ~

**BÀI 3: CÀI ĐẶT GIẢI THUẬT STRASSEN CHO NHÂN ĐA THỨC**

**1.Tìm hiểu bài toán**

Để cài đặt giải thuật Strassen cho việc nhân đa thức, chúng ta cần làm theo các bước sau đây:

- Chia đa thức thành các đa thức con có bậc nhỏ hơn bằng cách sử dụng thuật toán chia đa thức.

- Áp dụng giải thuật Strassen để tính toán tích của các đa thức con này.

- Kết hợp kết quả để tạo ra kết quả cuối cùng.

Bước đầu tiên là chia đa thức thành các đa thức con có bậc nhỏ hơn. Chúng ta có thể sử dụng thuật toán chia đa thức để làm điều này. Cụ thể, chúng ta chia đa thức ban đầu thành hai đa thức con có bậc bằng nhau và tiếp tục chia đa thức con này cho đến khi các đa thức con có bậc thấp hơn một ngưỡng nhất định.

Sau đó, chúng ta sử dụng giải thuật Strassen để tính toán tích của các đa thức con này. Giải thuật Strassen có thể được sử dụng để tính toán tích của hai ma trận vuông có cùng kích thước. Trong trường hợp của đa thức, chúng ta có thể coi đa thức là một ma trận dòng hoặc cột với các hệ số của đa thức là các phần tử của ma trận. Chúng ta sẽ cần thay đổi một chút giải thuật Strassen để làm việc với đa thức.

Cuối cùng, chúng ta kết hợp kết quả của tích các đa thức con để tạo ra kết quả cuối cùng của việc nhân đa thức bằng cách sử dụng phép cộng và phép trừ đa thức.

**2.Cài đặt**

**3.Hướng dẫn thực thi**

~ ~ ~ o0o ~ ~ ~

**BÀI 4: CÀI ĐẶT QUICKSORT**

**1.Tìm hiểu bài toán**

- Giải thuật Quicksort sắp xếp dãy a1, a2,…,aN dựa trên việc phân hoạch dãy ban đầu thành 3 thành phần:

+ Phần 1: Gồm các phần tử có giá trị không lớn hơn x

+ Phần 2: Gồm các phần tử có giá trị bằng x

+ Phần 3: Gồm các phần tử có giá trị không bé hơn x với x là giá trị của một phần tử tùy ý trong dãy ban đầu.

**2.Cài đặt**

#include <iostream>

using namespace std;

template <class DataType>

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;

}

**3.Hướng dẫn thực thi**

Thuật toán cho phép người dùng nhập số lượng phần tử, sau đó nhập các phần tử của mảng arr. Thuật toán sẽ xuất ra mảng đã được sắp xếp theo thứ tự tăng dần.

~ ~ ~ o0o ~ ~ ~

**BÀI 5: CÀI ĐẶT QUICKSORT VỚI VỊ TRÍ MỐC NGẪU NHIÊN**

**1.Tìm hiểu bài toán**

Để cài đặt thuật toán quicksort sử dụng vị trí mốc ngẫu nhiên, ta cần sử dụng hàm rand() để sinh ra một số ngẫu nhiên trong khoảng từ left đến right, sau đó chọn phần tử ở vị trí này làm mốc

**2.Cài đặt**

#include <iostream>

#include <cstdlib> // thư viện sử dụng hàm rand()

using namespace std;

template <class DataType>

int partition(DataType a[], int left, int right)

{

// sinh số ngẫu nhiên trong khoảng từ left đến right

int randIndex = rand() % (right - left + 1) + left;

DataType x = a[randIndex];

swap(a[randIndex], a[right]); // đưa phần tử mốc lên cuối mảng

int i = left - 1;

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

if (a[j] < x) {

i++;

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

}

}

swap(a[i + 1], a[right]);

return i + 1;

}

template <class DataType>

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

{

if (left >= right) return;

int pivotIndex = partition(a, left, right);

quickSort(a, left, pivotIndex - 1);

quickSort(a, pivotIndex + 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 << "Mang arr da duoc sap xep: ";

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

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

cout << endl;

return 0;

}

**3.Hướng dẫn thực thi**

Thuật toán cho phép người dùng nhập số lượng phần tử, sau đó nhập các phần tử của mảng arr. Thuật toán sẽ xuất ra mảng đã được sắp xếp theo thứ tự tăng dần.

~ ~ ~ o0o ~ ~ ~

**BÀI 6: CÀI ĐẶT BÀI TOÁN THÁP HÀ NỘI**

**1.Tìm hiểu bài toán**

Bài toán Tháp Hà Nội là một bài toán cổ điển trong lĩnh vực toán học và lập trình. Bài toán yêu cầu di chuyển tất cả đĩa từ một tháp ban đầu đến tháp khác, theo quy tắc chỉ được di chuyển một đĩa mỗi lần và không được đặt đĩa lớn hơn lên đĩa nhỏ hơn.

**2.Cài đặt**

#include <iostream>

using namespace std;

void hanoi(int n, char A, char B, char C) {

if (n == 1) {

cout << "Move disk 1 from " << A << " to " << C << endl;

return;

}

hanoi(n - 1, A, C, B);

cout << "Move disk " << n << " from " << A << " to " << C << endl;

hanoi(n - 1, B, A, C);

}

int main() {

int n;

cout << "Enter the number of disks: ";

cin >> n;

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

return 0;

}

**3.Hướng dẫn thực thi**

Thuật toán cho phép người dùng nhập số lượng đĩa của tháp. Thuật toán sẽ xuất ra thứ tự các bước thực hiện để sắp xếp tháp Hà Nội

~ ~ ~ o0o ~ ~ ~

**BÀI 7: CÀI ĐẶT BÀI TOÁN 8 CON HẬU**

**1.Tìm hiểu bài toán**

Bài toán 8 quân hậu là một bài toán cổ điển trong lĩnh vực thủ tục tìm kiếm và tối ưu hóa. Bài toán yêu cầu đặt 8 quân hậu trên một bàn cờ vua kích thước 8x8 sao cho không có quân hậu nào đang ăn được nhau.

Một cách để giải quyết bài toán này là sử dụng thuật toán quay lui (backtracking). Thuật toán này sẽ thử tất cả các cách đặt quân hậu trên bàn cờ và loại bỏ những trường hợp sai, cho đến khi tìm được một cách đặt thỏa mãn yêu cầu bài toán hoặc không còn cách đặt nào khả dĩ nữa.

**2.Cài đặt bài toán**

#include <iostream>

#include <vector>

using namespace std;

void printBoard(const vector<int>& board) {

for (int row = 0; row < 8; row++) {

for (int col = 0; col < 8; col++) {

if (board[row] == col) {

cout << "1 ";

} else {

cout << "0 ";

}

}

cout << endl;

}

cout << endl;

}

bool isSafe(const vector<int>& board, int row, int col) {

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

// Check for conflicts with previous rows

if (board[i] == col || abs(board[i] - col) == abs(i - row)) {

return false;

}

}

return true;

}

void solve(vector<int>& board, int row) {

if (row == 8) {

printBoard(board);

return;

}

for (int col = 0; col < 8; col++) {

if (isSafe(board, row, col)) {

board[row] = col;

solve(board, row + 1);

board[row] = -1;

}

}

}

int main() {

vector<int> board(8, -1);

solve(board, 0);

return 0;

}

**3.Hướng dẫn thực thi**

Bài toán được cài đặt với dữ liệu tự động không cần nhập thêm, do đó người dùng chỉ cần cài đặt code và chạy**.**

~ ~ ~ o0o ~ ~ ~

**BÀI 12: CÀI ĐẶT BÀI TOÁN TÌM ĐƯỜNG ĐI NGẮN NHẤT**

**1.Tìm hiểu bài toán**

Thuật toán tìm đường đi ngắn nhất (Shortest Path Algorithm) là một thuật toán tính toán đường đi ngắn nhất giữa hai điểm trên một đồ thị có hướng hoặc vô hướng. Mục tiêu của thuật toán là tìm đường đi từ một đỉnh xuất phát tới một đỉnh đích sao cho tổng trọng số của các cạnh trên đường đi là nhỏ nhất.

Có nhiều thuật toán tìm đường đi ngắn nhất như Dijkstra, Bellman-Ford, Floyd-Warshall, và A\*. Mỗi thuật toán có ưu và nhược điểm riêng, và được sử dụng tùy vào bối cảnh và yêu cầu của bài toán cụ thể.

Dưới đây là mô tả chung của thuật toán Dijkstra để tìm đường đi ngắn nhất giữa hai điểm trên đồ thị vô hướng.

- Khởi tạo một mảng dist[] có kích thước bằng số đỉnh của đồ thị, ban đầu giá trị của mỗi phần tử trong mảng bằng vô cực và dist[start] = 0 (với start là đỉnh xuất phát).

- Khởi tạo một hàng đợi ưu tiên (priority queue) pq để lưu trữ các đỉnh và khoảng cách tương ứng của chúng đến đỉnh xuất phát.

- Thêm đỉnh xuất phát vào pq.

- Với mỗi đỉnh trong pq:

+ Lấy ra đỉnh u có khoảng cách nhỏ nhất tới đỉnh xuất phát (đỉnh đầu tiên trong pq).

+ Nếu u là đỉnh đích, dừng thuật toán.

+ Đối với mỗi đỉnh kề v của u: Nếu khoảng cách từ đỉnh xuất phát tới v thông qua u là nhỏ hơn dist[v], cập nhật dist[v] và thêm đỉnh v vào pq.

- Nếu pq đã trống mà vẫn chưa tìm thấy đường đi tới đỉnh đích, tức là không có đường đi từ đỉnh xuất phát tới đỉnh đích.

Khi thuật toán kết thúc, giá trị của dist[end] sẽ là khoảng cách ngắn nhất từ đỉnh xuất phát tới đỉnh đích, và đường đi ngắn nhất có thể được xác định bằng cách lưu lại đỉnh kề tiếp theo của mỗi đỉnh trên đường đi.

Lưu ý rằng, thuật toán Dijkstra chỉ áp dụng cho đồ thị không có trọng số âm. Nếu có trọng số âm, ta cần sử dụng thuật toán Bellman-Ford hoặc các thuật toán khác.

Để cài đặt thuật toán Dijkstra trong C++, ta có thể sử dụng một mảng hai chiều graph để lưu trữ trọng số của các cạnh trong đồ thị, và một mảng dist để lưu khoảng cách ngắn nhất tới đỉnh xuất phát. Ta cũng có thể sử dụng một cấu trúc edge để lưu trữ thông tin về các cạnh trong đồ thị.

**2.Cài đặt**

#include <bits/stdc++.h>

using namespace std;

#define INF INT\_MAX // giá trị vô cùng

int main() {

int n, m; // n là số đỉnh, m là số cạnh

cin >> n >> m;

vector<vector<pair<int, int>>> adj(n); // danh sách kề của đồ thị

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

int u, v, w;

cin >> u >> v >> w;

adj[u].push\_back({v, w}); // thêm cạnh (u, v) với trọng số w vào danh sách kề của u

adj[v].push\_back({u, w}); // thêm cạnh (v, u) với trọng số w vào danh sách kề của v (nếu đồ thị vô hướng)

}

int s, t; // s là đỉnh bắt đầu, t là đỉnh kết thúc

cin >> s >> t;

vector<int> dist(n, INF); // khoảng cách ngắn nhất từ s đến các đỉnh khác

dist[s] = 0;

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({0, s});

while (!pq.empty()) {

int u = pq.top().second;

int d = pq.top().first;

pq.pop();

if (d > dist[u]) continue; // nếu khoảng cách đến u đã được cập nhật, bỏ qua

for (auto e : adj[u]) { // duyệt các đỉnh kề của u

int v = e.first;

int w = e.second;

if (dist[u] + w < dist[v]) { // nếu có đường đi từ s đến v thông qua u ngắn hơn

dist[v] = dist[u] + w; // cập nhật khoảng cách ngắn nhất từ s đến v

pq.push({dist[v], v}); // thêm v vào hàng đợi ưu tiên

}

}

}

if (dist[t] == INF) {

cout << "Khong co duong di tu " << s << " den " << t << endl;

} else {

cout << "Khoang cach ngan nhat tu " << s << " den " << t << " la " << dist[t] << endl;

}

return 0;

}

**3.Hướng dẫn thực thi**

Thuật toán cho phép người dùng nhập ma trận kề của đồ thị. Thuật toán sẽ xuất ra khoảng cách ngắn nhất từ một đỉnh đến đỉnh khác hoặc không có đường đi đến đỉnh đó.

~ ~ ~ o0o ~ ~ ~

**BÀI 14: CÀI ĐẶT BÀI TOÁN ĐỔI TIỀN**

**1.Tìm hiểu bài toán**

Thuật toán đổi tiền Việt là một bài toán cơ bản trong lĩnh vực lập trình. Bài toán yêu cầu tìm số lượng ít nhất các tờ tiền và xu để đổi cho một số tiền cho trước. Trong bài này, ta sẽ cài đặt thuật toán đổi tiền Việt bằng cách sử dụng giải thuật tham lam (greedy algorithm).

Giải thuật tham lam được sử dụng trong bài toán đổi tiền Việt như sau:

- Sắp xếp danh sách các loại tiền theo thứ tự giảm dần của giá trị.

- Với mỗi loại tiền, ta lấy càng nhiều càng tốt để đổi đến khi số tiền đổi được bằng số tiền cần đổi.

- Lặp lại bước 2 cho đến khi số tiền đổi được bằng số tiền cần đổi.

**2.Cài đặt**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

vector<int> V = {500000, 200000, 100000, 50000, 20000, 10000, 5000, 2000, 1000}; // danh sách các loại tiền

void doi\_tien(int n) {

sort(V.begin(), V.end(), greater<int>()); // sắp xếp danh sách theo thứ tự giảm dần

int i = 0;

while (n > 0) { // lặp lại cho đến khi số tiền đổi được bằng số tiền cần đổi

if (n >= V[i]) { // nếu số tiền cần đổi vẫn còn lớn hơn loại tiền đang xét

cout << V[i] << " "; // đưa loại tiền đang xét vào danh sách kết quả

n -= V[i]; // giảm số tiền cần đổi đi bằng giá trị loại tiền đang xét

} else { // nếu không đổi được loại tiền đang xét, chuyển sang loại tiền tiếp theo

i++;

}

}

cout << endl; // in ra kết quả

}

int main() {

int n;

cout << "Nhap so tien can doi: ";

cin >> n;

cout << "Cac loai tien can dung de doi " << n << " la: ";

doi\_tien(n);

return 0;

}

**3.Hướng dẫn thực thi**

Thuật toán cho phép người dùng nhập số tiền cần đổi. Thuật toán sẽ xuất ra số lượng các tờ tiền được đổi.