MÔ TẢ BÀI TOÁN

Bài thực hành

[BTH1: Cài đặt và tìm hiểu luật Horner cho đa thức 1](#_Toc136027789)

[BTH2: Cài đặt QuickSort 2](#_Toc136027790)

[BTH3: Cài đặt QuickSort với vị trí mốc ngẫu nhiên 3](#_Toc136027791)

[BTH4: Cài đặt bài toán Tháp Hà Nội 4](#_Toc136027792)

[BTH5: Cài đặt bài toán 8 con hậu 5](#_Toc136027793)

[BTH6: Cài đặt bài tập balo1 7](#_Toc136027794)

[BTH7: Cài đặt bài tập tìm chuỗi chung dài nhất 8](#_Toc136027795)

[BTH8: Cài đặt bài toán xếp lịch 9](#_Toc136027796)

[BTH9: Cài đặt bài toán tìm đường đi ngắn nhất 11](#_Toc136027797)

[BTH10: Cài đặt bài toán đổi tiền 12](#_Toc136027798)

# Cài đặt và tìm hiểu luật Horner cho đa thức

Đầu vào:

* Nhập bậc của đa thức:

VD: 3

* Nhập hệ số của x3­­:

VD: 2

* Nhập hệ số của x2:

VD: -3

* Nhập hệ số của x1:

VD: 1

* Nhập hệ số của x0:

VD: -1

* Nhập điểm x cần tính giá trị của đa thức:

VD: 2.5

Đầu ra:

* Giá trị của đa thức tại điểm 2.5 là: 14

Cách xử lý:

**Hàm horner:**

Bước 1: Khởi tạo biến result có giá trị ban đầu là 0.

Bước 2: Sử dụng vòng lặp for từ bậc cao nhất đến bậc thấp nhất của đa thức:

Trong mỗi vòng lặp, cập nhật giá trị result bằng cách nhân nó với x và cộng với hệ số tương ứng của bậc hiện tại.

Bước 3: Trả về giá trị result là kết quả của đa thức tại điểm x.

**Hàm main:**

Bước 1: Yêu cầu người dùng nhập vào bậc của đa thức.

Bước 2: Tạo một vector dathuc với kích thước là bac + 1 để lưu trữ các hệ số của đa thức.

Bước 3: Yêu cầu người dùng nhập các hệ số của đa thức từ bậc cao nhất đến bậc thấp nhất.

Bước 4: Yêu cầu người dùng nhập điểm x mà muốn tính giá trị của đa thức tại đó.

Bước 5: Gọi hàm horner với tham số là vector dathuc và điểm x, và lưu kết quả vào biến result.

Bước 6: In ra giá trị của đa thức tại điểm x mà người dùng nhập vào.

# Cài đặt QuickSort

Đầu vào:

* Nhập số lượng phần tử của mảng:

VD: 4

* Nhập các phần tử của mảng:

VD: 3 6 1 5

Đầu ra:

* Mảng trước khi sắp xếp: 3 6 1 5
* Mảng sau khi sắp xếp: 1 3 5 6

Cách xử lý:

**Hàm swap:**

Đầu vào: Nhận vào hai biến nguyên a và b.

Thực hiện hoán đổi giá trị của a và b.

**Hàm partition:**

Đầu vào: Nhận vào một vector arr, chỉ số low và chỉ số high.

Đầu ra: Trả về chỉ số của pivot sau khi hoàn thành phân vùng mảng.

Bước 1: Chọn pivot là phần tử đầu tiên của mảng, và khởi tạo chỉ số i là low + 1.

Bước 2: Sử dụng vòng lặp for để duyệt qua các phần tử từ low + 1 đến high.

Trong mỗi vòng lặp, nếu phần tử tại vị trí j nhỏ hơn pivot, hoán đổi nó với phần tử tại vị trí i và tăng i lên 1.

Bước 3: Hoán đổi pivot với phần tử tại vị trí i - 1 và trả về i - 1 là chỉ số của pivot đã được đặt đúng vị trí.

**Hàm quickSort:**

Đầu vào: Nhận vào một vector arr, chỉ số low và chỉ số high.

Bước 1: Kiểm tra điều kiện thoát của đệ quy: nếu low < high.

Bước 2: Gọi hàm partition để tìm pivot và phân vùng mảng thành hai phần.

Bước 3: Gọi đệ quy quickSort với mảng con bên trái pivot (có chỉ số từ low đến pivotIndex - 1).

Bước 4: Gọi đệ quy quickSort với mảng con bên phải pivot (có chỉ số từ pivotIndex + 1 đến high).

**Hàm printArray:**

Đầu vào: Nhận vào một vector arr.

In ra các phần tử của mảng arr.

**Hàm main:**

Bước 1: Yêu cầu người dùng nhập vào số lượng phần tử của mảng.

Bước 2: Tạo một vector arr với kích thước là n để lưu trữ các phần tử của mảng.

Bước 3: Yêu cầu người dùng nhập các phần tử của mảng.

Bước 4: In ra mảng trước khi sắp xếp.

Bước 5: Gọi hàm quickSort để tiến hành sắp xếp mảng arr.

Bước 6: In ra mảng sau khi sắp xếp.

# Cài đặt QuickSort với vị trí mốc ngẫu nhiên

Đầu vào:

* Nhập số lượng phần tử:

VD: 4

* Nhập các phần tử:

VD: 6 2 3 1

* Nhập vị trí bắt đầu:

VD: 3

Đầu ra:

* Mảng sau khi sắp xếp: 1 3 6

Cách xử lý:

**Hàm swap**

Được sử dụng để hoán đổi giá trị của hai biến.

**Hàm partition**

Là một phần của thuật toán quicksort. Nó chọn một phần tử làm pivot và phân vùng mảng thành hai phần, một phần chứa các phần tử nhỏ hơn pivot và một phần chứa các phần tử lớn hơn pivot. Hàm trả về chỉ số của pivot.

**Hàm quicksort**

Thực hiện thuật toán quicksort bằng cách chia mảng thành các phần nhỏ hơn và lớn hơn pivot, sau đó đệ quy áp dụng quicksort trên các phần đó.

**Hàm shiftArray**

Dịch chuyển các phần tử của mảng bắt đầu từ một chỉ số bắt đầu được nhập từ người dùng. Nó tạo một mảng tạm thời temp bằng cách sao chép các phần tử từ chỉ số bắt đầu đến cuối mảng, sau đó sao chép các phần tử từ đầu mảng đến chỉ số bắt đầu vào mảng temp. Cuối cùng, mảng arr được gán bằng mảng temp.

**Hàm main**

Là hàm chính của chương trình. Nó yêu cầu người dùng nhập số lượng phần tử của mảng, sau đó nhập các phần tử của mảng. Tiếp theo, người dùng nhập chỉ số bắt đầu để dịch chuyển các phần tử của mảng. Sau đó, chương trình gọi hàm shiftArray để dịch chuyển các phần tử và hàm quicksort để sắp xếp mảng. Cuối cùng, mảng được in ra sau khi sắp xếp.

# Cài đặt bài toán Tháp Hà Nội

Mô tả:

Luật chơi:

1. Có 3 cột để đặt đĩa được gọi tên lần lượt là A, B và C.

2. Ban đầu ở cột A có n đĩa được đánh số tăng dần theo kích thước của đĩa (như hình trên).

3. Mỗi lần chỉ được di chuyển 1 đĩa và đĩa đó phải nằm trên cùng của 1 trong 3 cột.

4. Một đĩa chỉ được đặt lên đĩa lớn hơn nó (kích thước không nhất thiết phải liền kề nhau, tức là đĩa nhỏ nhất có thể nằm trên đĩa lớn nhất).

5. Người chơi thắng khi toàn bộ n đĩa ban đầu ở cột A đã được chuyển hết sang cột C.

Đầu vào: Một số tự nhiên n là số đĩa cho trước ở cột A. (1 <= n <= 20)

VD: 2

Đầu ra:

Gồm nhiều dòng là các bước giải bài toán theo cách tối ưu nhất (ít bước chuyển đĩa nhất):

Mỗi dòng có cấu trúc "k Src Dest" (không có ngoặc kép) với:

- k là một số tự nhiên đại diện cho đĩa cần chuyển.

- Src là một chữ cái in hoa đại diện cho tên cột hiện tại của đĩa k.

- Dest là mọt chữ cái in hoa đại diện cho tên cột cần chuyển đến của đĩa k.

VD:

1 A B

2 A C

1 B C

Cách xử lý:

**Hàm chuyen\_dia**

Được sử dụng để giải bài toán tháp Hà Nội. Nó nhận vào số đĩa n cần chuyển, và các tham số nguon, trung\_gian, dich đại diện cho ba cột A, B, C trong bài toán.

Trường hợp cơ sở (base case) của đệ quy là khi chỉ còn một đĩa cần chuyển (n == 1). Khi đó, in ra thông báo chuyển đĩa từ cột nguon sang cột dich.

Trường hợp đệ quy: Nếu có nhiều hơn một đĩa cần chuyển (n > 1), ta thực hiện các bước sau:

Bước 1: Gọi đệ quy chuyen\_dia để chuyển n-1 đĩa từ cột nguon sang cột trung\_gian (sử dụng cột dich làm cột trung gian).

Bước 2: In ra thông báo chuyển đĩa thứ n từ cột nguon sang cột dich.

Bước 3: Gọi đệ quy chuyen\_dia để chuyển n-1 đĩa từ cột trung\_gian sang cột dich (sử dụng cột nguon làm cột trung gian).

**Hàm main**

Là hàm chính của chương trình. Nó yêu cầu người dùng nhập số lượng đĩa n. Sau đó, chương trình gọi hàm chuyen\_dia với các tham số đầu vào tương ứng. Khi chạy, chương trình sẽ in ra các bước chuyển đĩa từ cột A sang cột C theo quy tắc của bài toán tháp Hà Nội.

# Cài đặt bài toán 8 con hậu

Mô tả:

Bài toán tám quân hậu là bài toán đặt tám quân hậu trên bàn cờ vua kích thước 8×8 sao cho không có quân hậu nào có thể "ăn" được quân hậu khác, hay nói khác đi không quân hậu nào có để di chuyển theo quy tắc cờ vua. Màu của các quân hậu không có ý nghĩa trong bài toán này. Như vậy, lời giải của bài toán là một cách xếp tám quân hậu trên bàn cờ sao cho không có hai quân nào đứng trên cùng hàng, hoặc cùng cột hoặc cùng đường chéo. Bài toán tám quân hậu có thể tổng quát hóa thành bài toán đặt n quân hậu trên bàn cờ n×n(n ≥ 1).

Đầu vào: số n (số dòng và số cột của bàn cờ vua)

VD: 8

Đầu ra: số cách xếp quân hậu vào bàn cờ.

VD: 92

Cách xử lý:

Để giải quyết bài toán N-queens, chương trình sử dụng phương pháp backtracking. Khi đặt một quân hậu vào một vị trí trên bàn cờ, nó kiểm tra tính hợp lệ của việc đặt quân hậu đó thông qua hàm isSafe. Nếu quân hậu đó không tạo ra xung đột với các quân hậu đã đặt trước đó, chương trình tiếp tục đặt quân hậu ở dòng tiếp theo. Nếu không, nó thử các vị trí khác trên cùng dòng hoặc quay lại đặt lại vị trí của quân hậu trước đó. Quá trình này được lặp lại cho đến khi tất cả các cách đặt quân hậu trên bàn cờ được thử.

**Hàm isSafe**

Được sử dụng để kiểm tra xem có thể đặt một quân hậu vào vị trí (row, col) trên bàn cờ hay không.

Vòng lặp đầu tiên (for (int i = 0; i < row; i++)) kiểm tra các quân hậu đã được đặt trên các dòng trước đó trong cùng cột col. Nếu tìm thấy quân hậu tại vị trí (i, col) với i từ 0 đến row-1, tức là có quân hậu đang nằm trên cùng cột col trong các dòng từ 0 đến row-1, thì việc đặt quân hậu tại (row, col) sẽ tạo ra xung đột. Trong trường hợp này, hàm isSafe trả về false.

Vòng lặp thứ hai (for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) kiểm tra các quân hậu đã được đặt trên các dòng trước đó trong đường chéo chính (đường chéo từ trên xuống dưới và từ trái qua phải). Vòng lặp này duyệt qua các ô (i, j) trên đường chéo chính, bắt đầu từ vị trí (row, col) và đi lên trên và sang trái cho đến khi đạt đến vị trí (0, 0) hoặc vượt ra ngoài bàn cờ. Nếu tìm thấy quân hậu tại vị trí (i, j) trên đường chéo chính, tức là có quân hậu nằm trên đường chéo chính và nằm trước vị trí (row, col), thì việc đặt quân hậu tại (row, col) sẽ gây xung đột. Trong trường hợp này, hàm isSafe trả về false.

Vòng lặp cuối cùng (for (int i = row, j = col; i >= 0 && j < N; i--, j++) kiểm tra các quân hậu đã được đặt trên các dòng trước đó trong đường chéo phụ (đường chéo từ trên xuống dưới và từ phải qua trái). Vòng lặp này duyệt qua các ô (i, j) trên đường chéo phụ, bắt đầu từ vị trí (row, col) và đi lên trên và sang phải cho đến khi đạt đến vị trí (0, N-1) hoặc vượt ra ngoài bàn cờ. Nếu tìm thấy quân hậu tại vị trí (i, j) trên đường chéo phụ, tức là có quân hậu nằm trên đường chéo phụ và nằm trước vị trí (row, col), thì việc đặt quân hậu tại (row, col) sẽ gây xung đột. Trong trường hợp này, hàm isSafe trả về false.

Nếu không tìm thấy xung đột trong các trường hợp trên, tức là việc đặt quân hậu tại (row, col) là hợp lệ và không gây xung đột với các quân hậu đã được đặt trước đó. Trong trường hợp này, hàm isSafe trả về true.

**Hàm solveNQueens**

Được sử dụng để giải quyết bài toán N-queens bằng phương pháp backtracking.

Bước 1: Kiểm tra điều kiện thoát của đệ quy. Nếu row bằng N, tức là đã đặt được N quân hậu trên N dòng, chương trình tăng giá trị của biến count lên 1 (biểu thị một cách đếm số lượng cách đặt hậu hợp lệ) và kết thúc đệ quy.

Bước 2: Dùng một vòng lặp để duyệt qua các cột từ 0 đến N-1. Với mỗi cột, kiểm tra xem việc đặt quân hậu tại vị trí (row, col) có hợp lệ hay không bằng cách sử dụng hàm isSafe. Nếu hợp lệ, đặt quân hậu tại (row, col) bằng cách gán board[row][col] = 1, sau đó gọi đệ quy solveNQueens với row tăng lên 1 để tiếp tục đặt quân hậu trên dòng tiếp theo.

Bước 3: Sau khi kết thúc vòng lặp, trả về kết quả của biến count, đại diện cho số lượng cách đặt quân hậu hợp lệ trên bàn cờ N-queens.

**Hàm main**

Chương trình nhận input từ người dùng là giá trị N, tạo một bàn cờ board kích thước NxN với các giá trị ban đầu đều là 0, và gọi hàm solveNQueens để giải quyết bài toán N-queens trên bàn cờ này. Cuối cùng, chương trình in ra giá trị của biến count, đại diện cho số lượng cách đặt N quân hậu trên bàn cờ N-queens một cách hợp lệ.

# Cài đặt bài tập balo1

Mô tả: Cho n món hàng có khối lượng lần lượt là a[0], a[1], ... ,a[n-1] (đơn vị là kg) và 1 balô có khả năng chứa là w (kg).

Yêu cầu chọn những món hàng nào bỏ vào balô sao cho tổng khối lượng là lớn nhất và không vượt quá w.

Đầu vào: gồm 2 dòng

- Dòng 1: chứa 2 số n và w

- Dòng 2: chứa n số nguyên a[0], a[1], ... ,a[n-1]

VD:

3 10

3 5 7

Đầu ra: gồm 2 dòng

- Dòng 1: khối lượng tối ưu của balo sau khi chọn các món hàng

- Dòng 2: thứ tự các món hàng được chọn (số nhỏ ghi trước số lớn ghi sau)

VD:

10

0 2

Cách xử lý:

Giải quyết bài toán Balo bằng phương pháp quy hoạch động (Dynamic Programming).

**Trong hàm solveBalo**

Sử dụng một bảng tab có kích thước (n+1) x (w+1) để lưu trữ giá trị tối ưu cho từng trọng lượng và số lượng vật phẩm. Ý tưởng chính của thuật toán là xây dựng dần bảng tab bằng cách tính toán giá trị tối ưu cho từng trọng lượng và số lượng vật phẩm từ các giá trị đã tính toán trước đó.

Vòng lặp bên ngoài (for i) trong hàm solveBalo lặp qua từng vật phẩm và vòng lặp bên trong (for j) lặp qua từng trọng lượng. Trong mỗi lần lặp, chúng ta kiểm tra xem liệu vật phẩm hiện tại có thể được chọn để đưa vào balo hay không. Nếu có (if (items[i - 1] <= j)), chúng ta so sánh và chọn giá trị tối ưu giữa việc chọn vật phẩm hiện tại và không chọn vật phẩm hiện tại.

Sau khi xây dựng xong bảng tab, giá trị tối ưu của balo sẽ nằm ở phần tử tab[n][w].

Sau đó, sử dụng hai biến i và j để truy vết ngược từ phần tử cuối cùng của bảng tab để xác định các vật phẩm đã được chọn. Nếu giá trị của phần tử tab[i][j] khác với giá trị của phần tử tab[i-1][j], có nghĩa là vật phẩm thứ i đã được chọn. Chúng ta thêm i-1 vào vector selectedItems và giảm giá trị của j đi items[i-1]. Tiếp tục lặp lại cho đến khi i hoặc j trở thành 0.

Cuối cùng, in ra giá trị tối ưu của balo và danh sách các vật phẩm đã được chọn theo thứ tự ngược.

**Hàm main**

Chương trình đầu tiên nhận đầu vào từ người dùng là số lượng vật phẩm n và trọng lượng tối đa của balo w.

Tiếp theo, khai báo một vector items có kích thước n để lưu trữ trọng lượng của từng vật phẩm.

Sau đó, sử dụng một vòng lặp để nhập trọng lượng của từng vật phẩm từ người dùng vào vector items.

Cuối cùng, gọi hàm solveBalo để giải quyết bài toán balo với các đầu vào đã được nhập. Kết quả sẽ được in ra màn hình.

# Cài đặt bài tập tìm chuỗi chung dài nhất

Đầu vào:

* Nhập chuỗi đầu tiên:

VD:

ABCDE

* Nhập chuỗi thứ hai

ABDCE

Đầu ra: Chuỗi chung dài nhất là: ABDE

Cách xử lý:

Đây là một thuật toán quy hoạch động (dynamic programming) hiệu quả bằng cách sử dụng một bảng 2D để lưu trữ thông tin về độ dài của LCS cho các đoạn con để tìm chuỗi con chung dài nhất của hai chuỗi.

Thuật toán này tìm LCS dựa trên độ dài của chuỗi chứ không xác định các vị trí cụ thể của LCS trong chuỗi.

**Hàm findLCS** nhận vào hai chuỗi str1 và str2 và trả về chuỗi con chung dài nhất (LCS) của hai chuỗi đó.

Đầu tiên, khai báo một bảng 2D table có kích thước (m+1) x (n+1), với m và n lần lượt là độ dài của str1 và str2. Bảng này được sử dụng để lưu trữ độ dài của LCS tương ứng của các đoạn con của str1 và str2.

Sử dụng hai vòng lặp lồng nhau, ta duyệt qua từng ký tự của str1 và str2. Nếu ký tự tại vị trí i trong str1 và ký tự tại vị trí j trong str2 giống nhau, chúng ta cập nhật giá trị của table[i][j] bằng giá trị của table[i-1][j-1] + 1, tức là độ dài của LCS tăng thêm một.

Nếu ký tự tại vị trí i trong str1 và ký tự tại vị trí j trong str2 khác nhau, ta so sánh giá trị của table[i-1][j] và table[i][j-1] và lấy giá trị lớn nhất để cập nhật table[i][j].

Sau khi xử lý xong, giá trị của table[m][n] sẽ là độ dài của LCS của str1 và str2.

Tiếp theo, khởi tạo chuỗi lcs rỗng và sử dụng hai biến i và j để lần ngược từ m và n đến 0. Kiểm tra xem ký tự tại vị trí i trong str1 và ký tự tại vị trí j trong str2 có giống nhau hay không. Nếu giống nhau, thêm ký tự đó vào đầu chuỗi lcs và di chuyển sang vị trí i-1 và j-1. Nếu không giống nhau, thì di chuyển sang vị trí i-1 hoặc j-1 tùy thuộc vào giá trị của table[i-1][j] và table[i][j-1].

Sau khi kết thúc vòng lặp, chuỗi lcs sẽ chứa LCS của str1 và str2.

**Trong hàm main**

Yêu cầu người dùng nhập vào hai chuỗi str1 và str2.

Tiếp theo, gọi hàm findLCS với hai chuỗi str1 và str2 để tìm LCS.

Cuối cùng, in ra màn hình chuỗi LCS tìm được.

# Cài đặt bài toán xếp lịch

Đầu vào:

* Nhập số lượng công việc:

VD: 3

* Nhập tên công việc:

VD: A

* Nhập thời gian bắt đầu công việc 1:

VD: 9

* Nhập thời gian kết thúc công việc 1:

VD: 11

* Nhập tên công việc:

VD: B

* Nhập thời gian bắt đầu công việc 2:

VD: 10

* Nhập thời gian kết thúc công việc 2:

VD: 12

* Nhập tên công việc:

VD: C

* Nhập thời gian bắt đầu công việc 3:

VD: 11

* Nhập thời gian kết thúc công việc 3:

VD: 14

Đầu ra: Các công việc đã sắp xếp:

* Tên công việc: A, Bắt đầu: 9, Kết thúc: 11
* Tên công việc: B, Bắt đầu: 11, Kết thúc: 14

Cách xử lý:

**Khai báo struct Task** để biểu diễn thông tin về công việc, bao gồm tên công việc (name), thời gian bắt đầu (start) và thời gian kết thúc (end).

Định nghĩa **hàm compareTasks** để so sánh hai công việc dựa trên thời gian kết thúc (end).

Định nghĩa **hàm schedule** để lên lịch các công việc.

Trong hàm schedule:

Sắp xếp danh sách các công việc theo thứ tự tăng dần của thời gian kết thúc bằng cách sử dụng hàm sort và hàm so sánh compareTasks.

Nếu danh sách công việc không rỗng:

Thêm công việc đầu tiên vào danh sách scheduledTasks.

Gán giá trị của thời gian kết thúc của công việc cuối cùng cho biến lastEndTime.

Duyệt qua danh sách các công việc từ vị trí thứ 2:

Nếu thời gian bắt đầu của công việc hiện tại lớn hơn hoặc bằng lastEndTime, thêm công việc này vào danh sách scheduledTasks và cập nhật lastEndTime với thời gian kết thúc của công việc này.

**Trong hàm main:**

Yêu cầu người dùng nhập số lượng công việc (numTasks).

Tạo một vector tasks để lưu trữ danh sách các công việc.

Duyệt qua từng công việc và yêu cầu người dùng nhập tên công việc, thời gian bắt đầu và thời gian kết thúc.

Gọi hàm schedule để lên lịch các công việc và lưu kết quả vào vector scheduledTasks.

Nếu scheduledTasks rỗng, in ra thông báo không có công việc nào được sắp xếp.

Ngược lại, in ra danh sách các công việc đã được sắp xếp, bao gồm tên công việc, thời gian bắt đầu và thời gian kết thúc.

# Cài đặt bài toán tìm đường đi ngắn nhất

Mô tả: Cho đơn đồ thị có hướng với N đỉnh (0 < N < 1000). Hãy tìm đường đi ngắn nhất từ đỉnh u đến đỉnh v bất kỳ của đồ thị.

Yêu cầu: Sử dụng thuật toán Floyd để giải bài tập này.

Dữ liệu vào:

* Dòng 1: Nhập 3 số N u v

VD: 4 2 4

* N dòng tiếp theo: nhập ma trận trọng số của đồ thị

VD:

0 9 0 3

0 0 8 0

5 0 0 0

0 4 1 0

Dữ liệu ra:

- Nếu có đường đi thì xuất:

+ Tổng độ dài đường đi

+ Đường đi từ u đến v theo mẫu: u->...->v

- Nếu không có đường đi xuất “NO”

VD:

16

2->3->1->4

Cách xử lý:

Khai báo hằng số INF với giá trị lớn, được sử dụng để đại diện cho các cạnh không có kết nối trong đồ thị.

**Định nghĩa hàm floyd** để thực hiện thuật toán Floyd-Warshall.

Trong hàm floyd:

Khởi tạo ma trận dist và next với kích thước nxn và giá trị ban đầu là INF và -1.

Sao chép ma trận trọng số graph vào dist và thiết lập các đỉnh kề của mỗi cạnh.

Thực hiện thuật toán Floyd-Warshall để tính toán đường đi ngắn nhất và ma trận next.

Nếu dist[u][v] vẫn bằng INF, tức là không có đường đi từ u đến v, trả về ma trận chứa giá trị -1.

Ngược lại, tạo một vector path để lưu trữ đường đi từ u đến v bằng cách sử dụng ma trận next. Cuối cùng, trả về vector chứa đường đi path và độ dài của đường đi.

**Trong hàm main:**

Nhập số đỉnh n, đỉnh xuất phát u và đỉnh đích v.

Tạo ma trận graph với kích thước nxn và nhập các trọng số của các cạnh.

Gọi hàm floyd để tìm đường đi ngắn nhất và lưu kết quả vào biến result.

Nếu result[0][0] bằng -1, in ra "NO" để cho biết không có đường đi từ u đến v.

Ngược lại, in ra độ dài đường đi và in ra các đỉnh trên đường đi từ u đến v.

# Cài đặt bài toán đổi tiền

Đầu vào:

* Nhập số tiền cần đổi:

VD: 123

* Nhập số lượng đồng tiền:

VD: 4

* Nhập mệnh giá của từng đồng tiền:

VD: 1 10 25 50

Đầu ra:

* Số lượng đồng tiền cần ít nhất để đổi:

50: 2 tờ

25: 0 tờ

10: 2 tờ

1: 3 tờ

Cách xử lý:

**Khai báo hàm coinChange** để thực hiện việc đổi tiền.

Trong hàm coinChange:

Sắp xếp các đồng tiền theo thứ tự giảm dần.

Khởi tạo vector change với kích thước bằng số lượng đồng tiền và giá trị ban đầu là 0.

Lặp qua từng đồng tiền và kiểm tra nếu số tiền cần đổi lớn hơn hoặc bằng giá trị của đồng tiền đó:

Tính số lượng đồng tiền cần thiết để đổi bằng cách chia số tiền cần đổi cho giá trị của đồng tiền.

Giảm số tiền cần đổi bằng số lượng đồng tiền đã tính và giá trị của đồng tiền.

Nếu số tiền cần đổi bằng 0, in ra số lượng đồng tiền cần để đổi cho từng giá trị đồng tiền.

Ngược lại, in ra "Không thể đổi."

**Trong hàm main:**

Nhập số tiền cần đổi và số lượng đồng tiền.

Tạo vector coins với kích thước bằng số lượng đồng tiền và nhập giá trị của từng đồng tiền.

Gọi hàm coinChange để tính toán và in ra kết quả đổi tiền.