BÀI TẬP THỰC HÀNH 1

Mô tả bài toán

Luật Horner là một phương pháp đơn giản để tính giá trị của đa thức tại một điểm xác định. Phương pháp này được đặt tên theo nhà toán học người Anh William George Horner.

Luật Horner cho phép tính giá trị của đa thức bậc n bất kỳ tại một điểm xác định x_0 với độ phức tạp O(n). Phương pháp này cũng được sử dụng để rút gọn đa thức bằng cách tìm thừa số chung lớn nhất của các hệ số.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào:

- A: một mảng chứa các hệ số của đa thức theo thứ tự từ bậc cao nhất đến bậc thấp nhất, A[0] là hệ số của bậc cao nhất, A[n] là hệ số của bậc thấp nhất

- x_0: một số thực, điểm xác định để tính giá trị của đa thức

Dữ liệu đầu ra: Giá trị của đa thức tại điểm x_0

Code Python

```
def horner(A, x0):
    result = A[0]
    for i in range(1, len(A)):
        result = result * x0 + A[i]
    return result
```

Ví dụ: Đa thức $f(x) = 2x^3 - 3x^2 + 4x – 5$

A = [2, -3, 4, -5]

x_0 = 2

```
result = horner(A, x0)
print(result)
```

kết quả: 3

Trong ví dụ này, đa thức f(x) = 2x^3 - 3x^2 + 4x - 5 được tính giá trị tại điểm x = 2 bằng phương pháp Luật Horner, kết quả là 3.

BÀI TẬP THỰC HÀNH 2

Mô tả bài toán

Phép nhân ma trận là một phép tính cơ bản trong đại số tuyến tính. Giải thuật Strassen là một phương pháp hiệu quả để nhân hai ma trận vuông cùng bậc. Độ phức tạp thời gian của phương pháp Strassen là $O(n^{\log_2 7})$, nhanh hơn so với phương pháp thông thường $O(n^3)$.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào:

- A, B: hai ma trận vuông cùng bậc n

- n: kích thước của ma trận vuông

Dữ liệu đầu ra: C: ma trận tích của A và B

Code Python

```
def strassen(A, B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]

  # chia ma trận A và B thành các ma trận con bằng phép chia đôi
    A11, A12, A21, A22 = split_matrix(A)
    B11, B12, B21, B22 = split_matrix(B)

    # tính các ma trận phụ trợ
    P1 = strassen(A11 + A22, B11 + B22)
    P2 = strassen(A21 + A22, B11)
    P3 = strassen(A11, B12 - B22)
    P4 = strassen(A22, B21 - B11)
    P5 = strassen(A11 + A12, B22)
    P6 = strassen(A21 - A11, B11 + B12)
    P7 = strassen(A12 - A22, B21 + B22)
```

```
    # tính ma trận tích C
    C11 = P1 + P4 - P5 + P7
    C12 = P3 + P5
    C21 = P2 + P4
    C22 = P1 - P2 + P3 + P6

    # ghép các ma trận con để tạo ra ma trận C kết quả
    C = [[0 for j in range(n)] for i in range(n)]
    for i in range(n // 2):
        for j in range(n // 2):
            C[i][j] = C11[i][j]
            C[i][j + n // 2] = C12[i][j]
            C[i + n // 2][j] = C21[i][j]
            C[i + n // 2][j + n // 2] = C22[i][j]
    return C

def split_matrix(M):
    n = len(M)
    if n % 2 == 0:
        k = n // 2
        A = [row[:k] for row in M[:k]]
        B = [row[k:] for row in M[:k]]
        C = [row[:k] for row in M[k:]]
        D = [row[k:] for row in M[k:]]
    else:
        k = n // 2
        A = [row[:k+1] for row in M[:k+1]]
        B = [row[k+1:] for row in M[:k
```

BÀI TẬP THỰC HÀNH 3

Mô tả bài toán

Phép nhân đa thức là một phép tính cơ bản trong toán học. Giải thuật Strassen là một phương pháp hiệu quả để nhân hai đa thức có bậc nhỏ hơn hoặc bằng 3. Độ phức tạp thời gian của phương pháp Strassen là $O(n^{\log_2 7})$, nhanh hơn so với phương pháp thông thường $O(n^2)$.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào:

- A, B: hai đa thức cùng bậc n hoặc bậc nhỏ hơn hoặc bằng 3.

- n: bậc của đa thức

Dữ liệu đầu ra: C: đa thức tích của A và B

Code Python

```
def strassen_poly(A, B):
    n = len(A)
    if n <= 3:
        return naive_poly_mult(A, B)

    # chia đa thức A và B thành các đa thức con bằng phép chia đôi
    A0, A1 = split_poly(A)
    B0, B1 = split_poly(B)

    # tính các đa thức phụ trợ
    P0 = strassen_poly(A0, B0)
    P1 = strassen_poly(A1, B1)
    P2 = strassen_poly(add_poly(A0, A1), add_poly(B0, B1))
    P3 = strassen_poly(subtract_poly(A0, A1), add_poly(B0, B1))
    P4 = strassen_poly(add_poly(A0, A1), B1)
    P5 = strassen_poly(A0, subtract_poly(B1, B0))
    P6 = strassen_poly(A1, subtract_poly(B0, B1))
```

```
    # tính đa thức tích C
    C0 = add_poly(add_poly(P0, P1), subtract_poly(P4, P5))
    C1 = add_poly(P2, P3)
    C2 = add_poly(P4, P6)
    C = [0] * (2 * n - 1)
    for i in range(n):
        C[i] = C0[i]
        C[i + n] = C1[i]
        C[i + 2 * n] = C2[i]
    return C

def add_poly(A, B):
    return [a + b for a, b in zip(A, B)]

def subtract_poly(A, B):
    return [a - b for a, b in zip(A, B)]

def split_poly(P):
    n = len(P) // 2
    return P[:n], P[n:]

def naive_poly_mult(A, B):
    n = len(A)
    C = [0] * (2 * n - 1)
    for i in range(n):
        for j in range(n):
            C[i + j] += A[i] * B[j]
    return C
```

BÀI TẬP THỰC HÀNH 4

Mô tả bài toán

Quicksort là một thuật toán sắp xếp nhanh và hiệu quả trong việc sắp xếp một mảng các phần tử. Thuật toán QuickSort được phát minh bởi Tony Hoare vào năm 1960. Quicksort là một phương pháp chia để trị đệ quy. Quicksort chọn một phần tử trong mảng làm pivot và phân chia các phần tử còn lại thành hai mảng: các phần tử bé hơn pivot và các phần tử lớn hơn pivot. Tiếp theo, thuật toán đệ quy được áp dụng đối với hai mảng này.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào:

- arr: một mảng các số nguyên

Dữ liệu đầu ra:

- arr: mảng đầu vào đã được sắp xếp tăng dần

Code Python

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

BÀI TẬP THỰC HÀNH 5

Mô tả bài toán

Quicksort là một thuật toán sắp xếp nhanh và hiệu quả trong việc sắp xếp một mảng các phần tử. Thuật toán QuickSort được phát minh bởi Tony Hoare vào năm 1960. Quicksort là một phương pháp chia để trị đệ quy. Quicksort chọn một phần tử trong mảng làm pivot và phân chia các phần tử còn lại thành hai mảng: các phần tử bé hơn pivot và các phần tử lớn hơn pivot. Tiếp theo, thuật toán đệ quy được áp dụng đối với hai mảng này. Trong thuật toán quicksort, việc chọn pivot là rất quan trọng. Khi chọn pivot tốt, thuật toán sẽ đạt hiệu quả cao.

Trong bài toán này, chúng ta sẽ cài đặt thuật toán quicksort với việc chọn vị trí mốc ngẫu nhiên làm pivot.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào

- arr: một mảng các số nguyên

Dữ liệu đầu ra:

- arr: mảng đầu vào đã được sắp xếp tăng dần

Code Python

```
import random

def quicksort_random(arr):
    if len(arr) <= 1:
        return arr
    pivot = random.choice(arr)
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort_random(left) + middle + quicksort_random(right)
```

Trong hàm `*quicksort_random(arr)*`, chúng ta chọn ngẫu nhiên một phần tử trong mảng `*arr*` làm pivot bằng hàm `*random.choice(arr)*`. Sau đó, chúng ta sử dụng pivot để phân chia mảng `*arr*` thành 3 mảng: `*left*`, `*middle*` và `*right*`. Tiếp theo, chúng ta đệ quy áp dụng thuật toán quicksort cho hai mảng `*left*` và `*right*`.

BÀI TẬP THỰC HÀNH 8

Mô tả bài toán

Bài toán balo là một bài toán tối ưu hóa về cách sắp xếp các vật phẩm vào trong một balo với dung tích giới hạn nhất định. Bài toán được sử dụng rộng rãi trong lĩnh vực quản lý chuỗi cung ứng và vận tải.

Trong bài toán này, chúng ta sẽ giải quyết hai bài toán con của bài toán balo:

- Balo 1: Cho trước danh sách các vật phẩm có giá trị và trọng lượng khác nhau. Hãy chọn các vật phẩm để đặt vào một balo có dung tích giới hạn, sao cho giá trị của các vật phẩm được chọn là lớn nhất có thể.

- Balo 2: Tương tự như balo 1, nhưng mỗi vật phẩm có một số lượng giới hạn, tức là chỉ có thể chọn số lượng nhất định của vật phẩm đó.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào:

- weights: một mảng chứa trọng lượng của các vật phẩm

- values: một mảng chứa giá trị của các vật phẩm

- capacity: dung tích giới hạn của balo (trong trường hợp balo 1), hoặc một mảng chứa số lượng giới hạn của mỗi vật phẩm (trong trường hợp balo 2)

Dữ liệu đầu ra:

- Balo 1: Một mảng chứa chỉ số của các vật phẩm được chọn để đặt vào balo, sao cho tổng trọng lượng của các vật phẩm này không vượt quá dung tích giới hạn của balo, và tổng giá trị của các vật phẩm được chọn là lớn nhất có thể.

- Balo 2: Một mảng chứa số lượng của mỗi vật phẩm được chọn để đặt vào balo, sao cho tổng trọng lượng của các vật phẩm này không vượt quá dung tích giới hạn của balo, và tổng giá trị của các vật phẩm được chọn là lớn nhất có thể.

Code Python

Balo 1

```
def knapsack1(W, wt, val):
    n = len(wt)
    dp = [0] * (W + 1)
    for i in range(n):
        for w in range(W, wt[i] - 1, -1):
            dp[w] = max(dp[w], dp[w - wt[i]] + val[i])
    return dp[W]
```

Balo 2

```
def knapsack2(W, wt, val):
    n = len(wt)
    dp = [[0] * (W + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif wt[i-1] <= w:
                dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
     return dp[n][W]
```

BÀI TẬP THỰC HÀNH 10

Mô tả bài toán:

Cho hai chuỗi X và Y, tìm chuỗi chung dài nhất (LCS - Longest Common Subsequence) của chúng.

Ví dụ:

- X = "ABCDEF"
- Y = "ACDFG"
- Kết quả: chuỗi chung dài nhất là "ACDF"

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào: hai chuỗi X và Y.

Dữ liệu đầu ra: chuỗi chung dài nhất của X và Y.

Code Python

```
def LCS(X, Y):
    m = len(X)
    n = len(Y)
    # Tạo bảng đường chéo để lưu LCS của các cặp chuỗi con
    c = [[0] * (n + 1) for i in range(m + 1)]
    # Duyệt các phần tử của X và Y
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # Nếu ký tự tại vị trí i của X và ký tự tại vị trí j của Y giống nhau
            if X[i - 1] == Y[j - 1]:
                c[i][j] = c[i - 1][j - 1] + 1
            # Nếu khác nhau, lấy giá trị lớn hơn của LCS(X[:i], Y[:j-1]) và LCS(X[:i-1], Y[:j])
            else:
                c[i][j] = max(c[i - 1][j], c[i][j - 1])
    # Truy vết LCS từ bảng đường chéo
    lcs = ""
```

```
    i = m
    j = n
    while i > 0 and j > 0:
        # Nếu ký tự tại vị trí i của X và ký tự tại vị trí j của Y giống nhau
        if X[i - 1] == Y[j - 1]:
            lcs = X[i - 1] + lcs
            i -= 1
            j -= 1
        # Nếu khác nhau, đi theo hướng có giá trị lớn hơn
        elif c[i - 1][j] > c[i][j - 1]:
            i -= 1
        else:
            j -= 1
    return lcs

# Kiểm tra kết quả
X = "ABCDEF"
Y = "ACDFG"
print("Chuỗi chung dài nhất của", X, "và", Y, "là:", LCS(X, Y)) # Kết quả: ACDF
```

BÀI TẬP THỰC HÀNH 11

Mô tả bài toán:

Bài toán xếp lịch (Scheduling) là bài toán tìm cách xếp các công việc (jobs) vào các thời điểm trong một khoảng thời gian nhất định sao cho thỏa mãn các ràng buộc về thời gian hoàn thành và tối ưu hóa mục tiêu nào đó, ví dụ như tối ưu số lượng công việc hoàn thành, tối ưu thời gian hoàn thành, hoặc tối ưu tổng giá trị của các công việc.

Dữ liệu đầu vào và dữ liệu đầu ra

Dữ liệu đầu vào:

- Một danh sách các công việc (jobs), mỗi công việc có thời gian thực hiện và thời gian kết thúc.

- Khoảng thời gian tổng thể (tổng thời gian) mà các công việc phải được hoàn thành.

Dữ liệu đầu ra:

- Một lịch trình thực hiện các công việc sao cho tối ưu hóa mục tiêu được đề ra.

Code Python

```
def schedule(jobs, t):
    # Sắp xếp các công việc theo thời gian kết thúc
    jobs = sorted(jobs, key=lambda x: x[1])

    # Tạo danh sách lịch trình ban đầu với công việc đầu tiên
    schedule = [jobs[0]]

    # Thêm các công việc có thể được thực hiện vào lịch trình
    for i in range(1, len(jobs)):
        # Nếu thời gian bắt đầu công việc tiếp theo lớn hơn thời gian kết thúc công việc trước đó
        # thì thêm công việc đó vào lịch trình
        if jobs[i][0] >= schedule[-1][1]:
            schedule.append(jobs[i])

    # Trả về lịch trình hoàn chỉnh
    return schedule
```

BÀI TẬP THỰC HÀNH 12&13

Mô tả bài toán:

Bài toán tìm đường đi ngắn nhất là một trong những bài toán quan trọng trong lý thuyết đồ thị. Đây là bài toán về việc tìm đường đi từ một đỉnh xuất phát đến một đỉnh đích sao cho tổng trọng số của các cạnh trên đường đi là nhỏ nhất.

Dữ liệu đầu vào và dữ liệu đầu ra:

Dữ liệu đầu vào: đồ thị có trọng số, với số đỉnh và số cạnh được biểu diễn bằng các số nguyên dương. Các cạnh được biểu diễn bằng cặp (u, v, w) trong đó u và v là các đỉnh được nối với nhau bởi cạnh có trọng số w.

Dữ liệu đầu ra: đường đi ngắn nhất từ đỉnh xuất phát đến đỉnh đích, với tổng trọng số nhỏ nhất.

***Code Python*:**

```
import heapq

def dijkstra(graph, start, end):
    # Khởi tạo đường đi ngắn nhất tới tất cả các đỉnh là vô cùng
    distances = {vertex: float('infinity') for vertex in graph}
    # Khởi tạo đỉnh bắt đầu với khoảng cách là 0
    distances[start] = 0
    # Khởi tạo heap
    heap = [(0, start)]
    while heap:
        # Lấy đỉnh có khoảng cách ngắn nhất ra khỏi heap
        (current_distance, current_vertex) = heapq.heappop(heap)
        # Nếu đỉnh này đã được duyệt rồi thì bỏ qua
        if current_distance > distances[current_vertex]:
            continue
        # Duyệt qua tất cả các đỉnh kề với đỉnh hiện tại
        for neighbor, weight in graph[current_vertex].items():
            # Tính toán khoảng cách mới từ đỉnh bắt đầu tới đỉnh kề
```

```
        distance = current_distance + weight
        # Nếu khoảng cách mới nhỏ hơn khoảng cách đang lưu trữ thì cập nhật lại khoảng cách
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            # Thêm đỉnh kề vào heap
            heapq.heappush(heap, (distance, neighbor))
    # Trả về khoảng cách tối ưu tới đỉnh đích
    return distances[end]
```

BÀI TẬP THỰC HÀNH 14

Mô tả bài toán:

Bài toán đổi tiền là một bài toán trong lĩnh vực tiền tệ và tài chính, đặc biệt là trong các hoạt động mua bán. Bài toán đặt ra là cho trước số tiền cần đổi và danh sách các loại tiền có sẵn, mục tiêu là tìm ra cách đổi ít tiền nhất để được số tiền cần đổi.

Dữ liệu đầu vào và dữ liệu đầu ra:

Dữ liệu đầu vào:

- Số tiền cần đổi (total_amount) - số nguyên dương.

- Danh sách các loại tiền có sẵn (coins) - một mảng các số nguyên dương biểu thị giá trị của từng đồng tiền.

Dữ liệu đầu ra:

- Số lượng đồng tiền cần đổi của mỗi loại tiền (coin_counts) - một mảng các số nguyên không âm biểu thị số lượng đồng tiền cần đổi của từng loại tiền.

- Tổng số đồng tiền cần đổi (total_coin_count) - một số nguyên không âm biểu thị tổng số đồng tiền cần đổi.

Code Python

```
def exchange_coin(total_amount, coins):
    coin_counts = [0] * len(coins)
    total_coin_count = 0

    # Sắp xếp danh sách các đồng tiền theo giá trị giảm dần
    coins.sort(reverse=True)

    # Với mỗi đồng tiền, thực hiện đổi đến khi số tiền cần đổi bằng 0
    for i in range(len(coins)):
        while total_amount >= coins[i]:
            total_amount -= coins[i]
            coin_counts[i] += 1
            total_coin_count += 1
return coin_counts, total_coin_count
```

Giải thích code:

- Tạo một mảng coin_counts với độ dài bằng với số lượng loại tiền và gán giá trị ban đầu là 0 để lưu số lượng đồng tiền cần đổi của mỗi loại tiền.

- Tạo biến total_coin_count với giá trị ban đầu là 0 để lưu tổng số đồng tiền cần đổi.

- Sắp xếp danh sách các đồng tiền theo giá trị giảm dần để đảm bảo việc đổi ít tiền nhất có thể.

- Sử dụng vòng lặp for để duyệt qua từng loại tiền. Với mỗi loại tiền, thực hiện đổi đến khi số tiền cần đổi bằng 0. Trong quá trình đổi, tăng số lượng đồng tiền cần đổi của loại tiền đó lên 1 và tăng tổng số đồng tiền cần đổi lên 1