Họ và tên sinh viên: Đặng Quốc Hưng

MSSV: 47.01.104.094 Sáng thứ 6

**BÀI TẬP THỰC HÀNH PHÂN TÍCH THIẾT KẾ VÀ GIẢI THUẬT**

# Bài thực hành 1: Cài đặt và tìm hiểu luật Horner cho đa thức

- Mô tả thuật toán Horner cho đa thức:

+ Phân tích đa thức thành nhân tử là kiến thức cơ bản cho các bài học về nhân chia đơn thức, đa thức. Có rất nhiều cách để phân tích đa thức thành nhân tử. Tuy nhiên, có những bài toán đa thức các bạn học sinh sẽ gặp khó khăn trong việc phân tích chúng thành nhân tử.

+ Luật Horner (Horner's rule) là một phương pháp tính giá trị của một đa thức tại một điểm xác định, sử dụng các phép tính cộng, nhân và lũy thừa. Phương pháp này giúp giảm số lượng phép tính cần thiết để tính giá trị của một đa thức.

+ Công thức của luật Horner được viết dưới dạng sau đây:

P(x) = a\_n \* x^n + a\_(n-1) \* x^(n-1) + ... + a\_1 \* x + a\_0

= (((a\_n \* x + a\_(n-1)) \* x + a\_(n-2)) \* x + ... + a\_1) \* x + a\_0

+ Trong đó, `P(x)` là giá trị của đa thức tại điểm `x`, `a\_i` là hệ số của đa thức tại bậc `i`, và `n` là bậc của đa thức.

+ Công thức này cho phép tính giá trị của đa thức với số lượng phép tính tối đa là `2n`. Trong khi đó, nếu sử dụng công thức định nghĩa của đa thức, số lượng phép tính có thể lên đến `n^2`.

+ Do đó, phương pháp tính giá trị của đa thức bằng luật Horner rất hữu ích trong các ứng dụng yêu cầu tính toán đa thức với độ chính xác cao và hiệu quả tính toán cao.

- Cài đặt thuật toán Horner cho đa thức trong C++:

1. Khai báo hàm tính giá trị của đa thức theo thuật toán Horner:

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

double result = poly[0];

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

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

}

return result;

}

2. Sử dụng hàm Horner để tính giá trị của đa thức tại điểm x:

int main() {

double poly[] = {2, -6, 2, -1};

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

double x = 3;

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

cout << "The value of polynomial at x = " << x << " is: " << result << endl;

return 0;

}

+ Ở đây, đa thức được biểu diễn dưới dạng mảng poly với n là số lượng các hệ số của đa thức. Trong hàm main, chúng ta sử dụng hàm horner để tính giá trị của đa thức tại điểm x và hiển thị kết quả trên màn hình.

+ Lưu ý rằng trong thuật toán Horner, chúng ta tính giá trị của đa thức bằng cách sử dụng các phép tính cộng, nhân và lũy thừa với x. Việc sử dụng thuật toán Horner giúp giảm thiểu số lượng phép tính so với cách tính giá trị của đa thức bằng cách sử dụng công thức định nghĩa.

- Độ phức tạp của thuật toán Horner cho đa thức:

+ Độ phức tạp của hàm horner() trong đa thức là O(n), với n là số lượng hệ số đa thức

+ Trong hàm horner(), chúng ta sử dụng một vòng lặp for với i từ 1 đến n - 1 để tính giá trị của đa thức theo thuật toán Horner. Trong mỗi lần lặp, chúng ta thực hiện một phép nhân và một phép cộng. Vì vậy, số lượng phép tính cần thực hiện trong hàm horner() là 2(n - 1).

+ Vì độ phức tạp của hàm horner() phụ thuộc vào số lượng hệ số của đa thức, do đó độ phức tạp của hàm horner() là **O(n).**

# Bài thực hành 2: Cài đặt chứng minh độ phức tạp, giải thuật Strassen cho nhân ma trận

- Mô tả giải thuật Strassens:

+ Giải thuật Strassen là một thuật toán hiệu quả để nhân hai ma trận vuông cùng kích thước. Nó được đặt theo tên của nhà toán học người Đức Volker Strassen, người đầu tiên giới thiệu nó vào năm 1969.

+ Thuật toán Strassen sử dụng phương pháp chia để trị, tách mỗi ma trận thành 4 ma trận con bằng cách sử dụng các phép tính toán đại số tuyến tính. Tiếp đó, thuật toán kết hợp các ma trận con này để tạo ra ma trận kết quả. Thay vì nhân từng phần tử của hai ma trận, Strassen sử dụng một số phép tính đại số tuyến tính để giảm số lượng phép nhân cần thiết.

+ Với kích thước ma trận lớn, Strassen có thể giảm số lượng phép nhân so với thuật toán nhân ma trận thông thường. Tuy nhiên, vì phương pháp chia để trị, thuật toán Strassen sẽ có thời gian chạy chậm hơn trong trường hợp các ma trận nhỏ.

- Cài đặt giải thuật Strassens nhân hai ma trận trong C++:

vector<vector<int>> strassen(vector<vector<int>>& A, vector<vector<int>>& B) {

int n = A.size();

if (n == 1) {

vector<vector<int>> C(1, vector<int>(1));

C[0][0] = A[0][0] \* B[0][0];

return C;

}

int m = n / 2;

vector<vector<int>> A11(m, vector<int>(m)), A12(m, vector<int>(m)), A21(m, vector<int>(m)), A22(m, vector<int>(m));

vector<vector<int>> B11(m, vector<int>(m)), B12(m, vector<int>(m)), B21(m, vector<int>(m)), B22(m, vector<int>(m));

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

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

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + m];

A21[i][j] = A[i + m][j];

A22[i][j] = A[i + m][j + m];

B11[i][j] = B[i][j];

B12[i][j] = B[i][j + m];

B21[i][j] = B[i + m][j];

B22[i][j] = B[i + m][j + m];

}

}

vector<vector<int>> P1 = strassen(A11, B12 - B22);

vector<vector<int>> P2 = strassen(A11 + A12, B22);

vector<vector<int>> P3 = strassen(A21 + A22, B11);

vector<vector<int>> P4 = strassen(A22, B21 - B11);

vector<vector<int>> P5 = strassen(A11 + A22, B11 + B22);

vector<vector<int>> P6 = strassen(A12 - A22, B21 + B22);

vector<vector<int>> P7 = strassen(A11 - A21, B11 + B12);

vector<vector<int>> C11(m, vector<int>(m)), C12(m, vector<int>(m)), C21(m, vector<int>(m)), C22(m, vector<int>(m));

C11 = P5 + P4 - P2 + P6;

C12 = P1 + P2;

C21 = P3 + P4;

C22 = P5 + P1 - P3 - P7;

vector<vector<int>> C(n, vector<int>(n));

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

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

C[i][j] = C11[i][j];

C[i][j + m] = C12[i][j];

C[i + m][j] = C21[i][j];

C[i + m][j + m] = C22[i][j];

}

}

return C;

}

- Độ phức tạp của thuật toán Strassen cho nhân ma trận:

+ Độ phức tạp của thuật toán Strassen được tính bằng đệ quy, với công thức như sau:

T(n) = 7T(n/2) + O(n^2)

trong đó n là kích thước của ma trận đầu vào. T(n) là độ phức tạp của thuật toán Strassen.

+ Theo công thức trên, độ phức tạp của thuật toán Strassen là **O(n^log2(7))**, tức là khoảng O(n^2.81). Vì vậy, thuật toán Strassen nhanh hơn phép nhân ma trận truyền thống (độ phức tạp **O(n^3)**) đối với các ma trận lớn. Tuy nhiên, do hệ số ở độ phức tạp của Strassen cao hơn, nên chỉ hiệu quả trên các ma trận có kích thước lớn.

# Bài thực hành 3: Cài đặt giải thuật Strassen cho nhân đa thức

- Mô tả giải thuật Strassens cho nhân đa thức:

+ Giải thuật Strassen cũng có thể được áp dụng để nhân đa thức. Để làm điều này, chúng ta có thể biểu diễn đa thức dưới dạng các ma trận, với mỗi hệ số đa thức được lưu trữ trong một phần tử của ma trận.

+ Giả sử chúng ta muốn nhân hai đa thức A(x) và B(x) có bậc là n - 1. Ta có thể biểu diễn hai đa thức này dưới dạng ma trận như sau:

A = [a0 a1 a2 ... an-1 0 0 ... 0]

[0 a0 a1 ... an-2 an-1 0 ... 0]

[0 0 a0 ... an-3 an-2 ... 0]

[... ]

[0 0 0 ... a1 a0 0 ... 0]

[0 0 0 ... 0 a1 a0 ... 0]

[0 0 0 ... 0 0 a1 ... an-1]

B = [b0 b1 b2 ... bn-1 0 0 ... 0]

[0 b0 b1 ... bn-2 bn-1 0 ... 0]

[0 0 b0 ... bn-3 bn-2 ... 0]

[... ]

[0 0 0 ... b1 b0 0 ... 0]

[0 0 0 ... 0 b1 b0 ... 0]

[0 0 0 ... 0 0 b1 ... bn-1]

+ Từ đó, chúng ta có thể áp dụng giải thuật Strassen trên hai ma trận này để tính toán đa thức sản phẩm C(x) = A(x) \* B(x).

+ Tuy nhiên, khi sử dụng giải thuật Strassen cho nhân đa thức, chúng ta cần đảm bảo rằng độ dài của hai đa thức A(x) và B(x) phải là cùng một lũy thừa của 2 để có thể chia chúng thành các phần bằng nhau. Nếu không, ta có thể sử dụng các phép bù và thêm phần tử 0 vào cuối của một trong hai đa thức để đạt được độ dài tương đương.

+ Sau khi tính được ma trận sản phẩm C, ta có thể chuyển đổi lại thành đa thức bằng cách gộp các phần tử của cùng một bậc lại với nhau.

- Cài đặt giải thuật Strassens cho nhân đa thức bằng C++:

// Hàm tính tổng của hai đa thức A và B

vector<int> add\_polynomial(vector<int> A, vector<int> B) {

int n = A.size(), m = B.size();

vector<int> C(max(n, m));

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

C[i] = ((i < n) ? A[i] : 0) + ((i < m) ? B[i] : 0);

}

return C;

}

// Hàm tính hiệu của hai đa thức A và B

vector<int> sub\_polynomial(vector<int> A, vector<int> B) {

int n = A.size(), m = B.size();

vector<int> C(max(n, m));

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

C[i] = ((i < n) ? A[i] : 0) - ((i < m) ? B[i] : 0);

}

return C;

}

// Hàm nhân đa thức A và B bằng giải thuật Strassen

vector<int> multiply\_polynomial(vector<int> A, vector<int> B) {

int n = A.size(), m = B.size();

if (n < m) {

swap(A, B);

swap(n, m);

}

if (n == 1) {

vector<int> C(1);

C[0] = A[0] \* B[0];

return C;

}

int k = (n + 1) / 2;

vector<int> A1(A.begin(), A.begin() + k);

vector<int> A2(A.begin() + k, A.end());

vector<int> B1(B.begin(), B.begin() + min(m, k));

vector<int> B2(B.begin() + min(m, k), B.end());

vector<int> C1 = multiply\_polynomial(A1, B1);

vector<int> C2 = multiply\_polynomial(A2, B2);

vector<int> A3 = add\_polynomial(A1, A2);

vector<int> B3 = add\_polynomial(B1, B2);

vector<int> C3 = multiply\_polynomial(A3, B3);

vector<int> C4 = sub\_polynomial(sub\_polynomial(C3, C1), C2);

vector<int> C(n + m - 1);

for (int i = 0; i < C1.size(); i++) {

C[i] += C1[i];

}

for (int i = 0; i < C2.size(); i++) {

C[i + 2 \* k] += C2[i];

}

for (int i = 0; i < C4.size(); i++) {

C[i + k] += C4[i];

}

return C;

}

+ Chúng ta đã cài đặt các hàm tính tổng, tính hiệu và nhân đa thức, sử dụng giải thuật Strassen. Trong hàm `multiply\_polynomial`, ta sử dụng đệ quy để nhân đa thức `A` và `B` dựa trên giải thuật Strassen.

+ Trong hàm `multiply\_polynomial`, ta bổ sung các bước sau:

1. Nếu `l` = 1, tức là `A` và `B` chỉ có một phần tử, ta trả về một đa thức có một phần tử là tích của hai phần tử đó.

2. Chia đa thức `A` thành hai nửa `A1` và `A2`, chia đa thức `B` thành hai nửa `B1` và `B2`.

3. Tính tích `C1` của `A1` và `B1`, tính tích `C2` của `A2` và `B2`.

4. Tính tổng `A1` và `A2`, tính tổng `B1` và `B2`.

5. Tính tích `C3` của `A1 + A2` và `B1 + B2`.

6. Tính `C4` bằng cách trừ `C3`, `C1` và `C2`.

7. Tạo đa thức `C` bằng cách ghép `C1`, `C4` và `C2` theo thứ tự.

- Độ phức tạp của thuật toán Strassens cho nhân đa thức:

+ Độ phức tạp của giải thuật Strassen cho nhân đa thức là **O(n^log2(7))**, trong đó n là số lượng hệ số của đa thức.

+ Thuật toán Strassen cho phép giảm số lượng phép nhân cần thiết để nhân hai ma trận từ 8 phép nhân xuống còn 7 phép nhân. Tuy nhiên, trong trường hợp nhân đa thức, không có ma trận nào được sử dụng, do đó sẽ không có giảm số lượng phép nhân nào.

+ Tuy nhiên, giải thuật Strassen vẫn giảm số lượng phép tính so với giải thuật nhân đa thức truyền thống bằng cách tối ưu việc tính toán các đa thức con và thực hiện phép cộng và trừ một cách hiệu quả. Vì vậy, giải thuật Strassen vẫn là một giải thuật hiệu quả cho việc nhân đa thức.

# Bài thực hành 4: Cài đặt QuickSort

- Mô tả thuật toán QuickSort:

+ Quick Sort là một thuật toán sắp xếp hiệu quả dựa trên việc phân chia mảng dữ liệu thành các nhóm phần tử nhỏ hơn. Giải thuật sắp xếp nhanh chia mảng thành hai phần bằng cách so sánh từng phần tử của mảng với một phần tử được gọi là phần tử chốt. Một mảng bao gồm các phần tử nhỏ hơn hoặc bằng phần tử chốt và một mảng gồm các phần tử lớn hơn phần tử chốt.

+ Quá trình phân chia này diễn ra cho đến khi độ dài của các mảng con đều bằng 1. Với phương pháp đệ quy ta có thể sắp xếp nhanh các mảng con sau khi kết thúc chương trình ta được một mảng đã sắp xếp hoàn chỉnh.

- Cài đặt thuật toán QuickSort bằng C++:

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

int i = left, j = right;

int pivot = arr[(left + right) / 2];

while (i <= j) {

while (arr[i] < pivot) {

i++;

}

while (arr[j] > pivot) {

j--;

}

if (i <= j) {

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

i++;

j--;

}

}

if (left < j) {

quickSort(arr, left, j);

}

if (i < right) {

quickSort(arr, i, right);

}

}

+ Giải thích:

• Hàm quickSort nhận vào một mảng arr, giới hạn trái left và giới hạn phải right. Ban đầu, giới hạn trái và phải là đầu và cuối mảng, tương ứng với left=0 và right=n-1.

• Chọn một phần tử pivot trong mảng, thường là phần tử ở giữa (arr[(left + right) / 2]).

• Chia mảng thành 2 phần: các phần tử nhỏ hơn pivot được đưa về bên trái pivot, các phần tử lớn hơn pivot được đưa về bên phải pivot.

• Lặp lại việc chia mảng cho đến khi mỗi phần tử được đặt ở đúng vị trí của nó.

• Cuối cùng, thực hiện đệ quy với 2 mảng con trái và phải của pivot.

Kết quả sẽ là mảng đã được sắp xếp theo thứ tự tăng dần.

- Độ phức tạp của thuật toán QuickSort:

+ Độ phức tạp của thuật toán QuickSort phụ thuộc vào cách chọn pivot và thứ tự đầu vào của mảng. Trong trường hợp xấu nhất, khi mảng đã sắp xếp hoặc ngược lại sắp xếp theo thứ tự đảo ngược, độ phức tạp của QuickSort sẽ là O(n^2).

+ Tuy nhiên, trong trường hợp trung bình và tốt nhất, khi pivot được chọn một cách tối ưu, độ phức tạp của QuickSort là O(n\*log n).

+ Trong đoạn code trên, pivot được chọn ở giữa mảng (`arr[(left + right) / 2]`), do đó độ phức tạp trung bình của QuickSort là **O(n\*log n)**. Tuy nhiên, độ phức tạp cụ thể của đoạn code trên phụ thuộc vào đầu vào của mảng, và có thể dao động từ **O(n\*log n)** đến **O(n^2)**.

# Bài thực hành 5: Cài đặt QuickSort với vị trí mốc ngẫu nhiên

- *Mô tả thuật toán QuickSort với vị trí mốc ngẫu nhiên*:

+ Thuật toán QuickSort với vị trí mốc ngẫu nhiên (Randomized QuickSort) là một biến thể của thuật toán QuickSort, trong đó mốc được chọn ngẫu nhiên trong mảng.

+ Các bước của thuật toán Randomized QuickSort như sau:

1. Chọn một phần tử ngẫu nhiên trong mảng làm mốc.

2. Chia mảng thành 2 phần: các phần tử nhỏ hơn mốc được đưa về bên trái mốc, các phần tử lớn hơn mốc được đưa về bên phải mốc.

3. Lặp lại việc chia mảng cho đến khi mỗi phần tử được đặt ở đúng vị trí của nó.

4. Thực hiện đệ quy với 2 mảng con trái và phải của mốc.

+ Đối với việc chọn mốc ngẫu nhiên, ta có thể sử dụng hàm `rand()` trong thư viện `<cstdlib>` để chọn một vị trí ngẫu nhiên trong mảng làm mốc.

- *Cài đặt giải thuật toán QuickSort với vị trí mốc ngẫu nhiên bằng C++:*

// Hàm hoán đổi giá trị của 2 phần tử

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

// Hàm chọn vị trí mốc ngẫu nhiên

int getRandomPivot(int arr[], int left, int right) {

srand(time(NULL));

return left + rand() % (right - left + 1);

}

// Hàm chia mảng và trả về vị trí của phần tử mốc

int partition(int arr[], int left, int right) {

int pivotIndex = getRandomPivot(arr, left, right);

int pivotValue = arr[pivotIndex];

swap(arr[pivotIndex], arr[right]); // Di chuyển phần tử mốc vào cuối mảng

int storeIndex = left;

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

if (arr[i] < pivotValue) {

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

storeIndex++;

}

}

swap(arr[storeIndex], arr[right]); // Đưa phần tử mốc về vị trí đúng của nó

return storeIndex;

}

// Hàm sắp xếp QuickSort với vị trí mốc ngẫu nhiên

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

if (left < right) {

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

quickSort(arr, left, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, right);

}

}

Trong code trên, hàm getRandomPivot() được sử dụng để chọn một vị trí ngẫu nhiên trong mảng làm mốc cho QuickSort. Hàm partition() thực hiện việc chia mảng và đưa phần tử mốc về vị trí đúng của nó. Hàm quickSort() sử dụng đệ quy để sắp xếp mảng.

- Độ phức tạp của thuật toán QuickSort với vị trí mốc ngẫu nhiên:

+ Độ phức tạp của thuật toán QuickSort với vị trí mốc ngẫu nhiên được tính theo trường hợp trung bình (expected case), và là O(n log n). Tuy nhiên, trong trường hợp xấu nhất (worst case), khi mảng đã được sắp xếp hoặc gần như đã được sắp xếp, độ phức tạp sẽ là O(n^2).

+ Trong đoạn code trên, hàm `partition()` và hàm `getRandomPivot()` đều có độ phức tạp là O(n). Hàm `quickSort()` được gọi đệ quy hai lần với mỗi lần gọi sẽ xử lý một nửa của mảng, do đó độ phức tạp trung bình của hàm này là O(n log n). Hàm `printArray()` có độ phức tạp là O(n).

+ Tổng thể, độ phức tạp của đoạn code trên trong trường hợp trung bình là **O(n log n).**

# Bài thực hành 6: Cài đặt bài toán tháp Hà Nội

-Mô tả bài toán tháp Hà Nội:

+ Bài toán tháp Hà Nội là một bài toán kinh điển trong lý thuyết đồng thời cũng là một bài toán được ứng dụng rộng rãi trong thực tế. Bài toán được đặt tên theo tên thành phố Hà Nội ở Việt Nam, nơi nó được phát triển và trình bày lần đầu tiên.

+ Bài toán tháp Hà Nội đặt ra câu hỏi: "Làm thế nào để chuyển một đống đĩa có kích thước khác nhau xếp theo thứ tự đường kính giảm dần, từ một cọc sang cọc khác, sao cho trên mỗi cọc luôn đặt đĩa nhỏ hơn đĩa lớn?".

+ Bài toán tháp Hà Nội có thể được mô hình hóa như sau: Cho ba cọc (được đánh số từ 1 đến 3) và một số đĩa được xếp theo thứ tự đường kính giảm dần trên cọc 1. Mục tiêu là di chuyển tất cả các đĩa sang cọc 3 sao cho không có đĩa nào được đặt lên một đĩa có đường kính nhỏ hơn.

+ Theo quy tắc của bài toán, một bước di chuyển bao gồm việc chọn đĩa lớn nhất trên cọc hiện tại, di chuyển đĩa đó sang cọc khác và xếp lên trên cùng của cọc đó. Bài toán yêu cầu tìm cách di chuyển tất cả các đĩa từ cọc 1 sang cọc 3 bằng cách sử dụng cọc 2 như một bộ trung gian.

+ Bài toán tháp Hà Nội là một bài toán về đệ quy, và có thể được giải quyết bằng cách áp dụng thuật toán đệ quy để tìm lời giải cho từng trường hợp nhỏ hơn của bài toán, cho đến khi đạt được trường hợp cơ sở.

- Cài đặt thuật toán chia để trị giải bài toán tháp Hà Nội bằng C++:

void move(int n, char source, char target, char auxiliary) {

if (n == 1) {

cout << "Move disk 1 from rod " << source << " to rod " << target << endl;

return;

}

move(n-1, source, auxiliary, target);

cout << "Move disk " << n << " from rod " << source << " to rod " << target << endl;

move(n-1, auxiliary, target, source);

}

+ Trong đó, hàm move được gọi đệ quy để di chuyển các đĩa từ cọc nguồn (source) sang cọc đích (target) thông qua cọc phụ (auxiliary). Nếu chỉ có một đĩa cần di chuyển, thì đĩa đó được chuyển trực tiếp từ cọc nguồn sang cọc đích.

- Độ phức tạp của thuật toán chia để trị giải bài toán tháp Hà Nội:

+ Độ phức tạp của thuật toán giải bài toán tháp Hà Nội với phương pháp chia để trị là O(2^n), trong đó n là số lượng đĩa. Vì mỗi lần đệ quy, chúng ta giảm số lượng đĩa cần di chuyển đi 1, cho đến khi chỉ còn 1 đĩa cần di chuyển.

+ Do đó, độ phức tạp của mã nguồn trên là **O(2^n).**

# Bài thực hành 7: Cài đặt bài toán 8 con hậu

- Mô tả bài toán 8 con hậu:

+ Bài toán 8 con hậu (hay còn gọi là bài toán n quân hậu) là một bài toán đặt ra câu hỏi: Làm thế nào để đặt n quân hậu lên một bàn cờ vua kích thước nxn sao cho không có quân hậu nào ăn được quân hậu khác?

+ Trong bàn cờ vua, mỗi con hậu có thể ăn được các quân cờ nằm trên cùng hàng, cùng cột hoặc cùng đường chéo với nó. Bài toán 8 con hậu là bài toán đặt 8 quân hậu lên bàn cờ vua 8x8 sao cho không có quân hậu nào ăn được quân hậu khác. Đây là một bài toán rất nổi tiếng trong lĩnh vực trí tuệ nhân tạo và tối ưu hóa.

+ Bài toán 8 con hậu có thể được giải quyết bằng nhiều cách khác nhau, trong đó phương pháp sử dụng thuật toán quay lui là phổ biến nhất. Thuật toán quay lui sử dụng đệ quy để thử từng vị trí cho mỗi con hậu trên bàn cờ. Nếu vị trí đó là hợp lệ (tức là không bị ăn), tiếp tục đệ quy để đặt các quân hậu khác. Nếu vị trí đó không hợp lệ, ta quay lại vị trí trước đó và thử một vị trí khác cho quân hậu đó. Khi đã đặt được n quân hậu, ta sẽ có một cách đặt hợp lệ và xuất kết quả.

+ Tuy nhiên, vì số lượng trường hợp cần kiểm tra lên đến hàng tỷ, nên thuật toán quay lui có độ phức tạp rất cao và không thực tế để giải bài toán n quân hậu với n lớn.

- Cài đặt thuật toán 8 con hậu bằng C++:

const int n = 8;

vector<int> col(n);

vector<int> diag1(2\*n-1);

vector<int> diag2(2\*n-1);

int count = 0;

void search(int y) {

if (y == n) { // Tìm được 1 giải pháp

count++;

return;

}

for (int x = 0; x < n; x++) { // Thử đặt 1 quân hậu vào mỗi cột của hàng y

if (col[x] || diag1[x+y] || diag2[x-y+n-1]) {

continue; // Ô (x, y) đã bị chiếm

}

col[x] = diag1[x+y] = diag2[x-y+n-1] = 1; // Đánh dấu ô (x, y) là đã chiếm

search(y+1); // Đệ quy tìm nước đi tiếp theo trên hàng y+1

col[x] = diag1[x+y] = diag2[x-y+n-1] = 0; // Hủy đánh dấu ô (x, y) là đã chiếm

}

}

+ Trong đó, col là một mảng lưu trữ các cột đã được sử dụng, diag1 và diag2 lần lượt lưu trữ các đường chéo chính và đường chéo phụ đã được sử dụng. Biến count là biến đếm số lượng giải pháp tìm được. Hàm search sẽ đệ quy để thử tất cả các nước đi có thể trên mỗi hàng, và đánh dấu các ô đã sử dụng bằng cách gán giá trị 1 cho các phần tử tương ứng của col, diag1, và diag2. Nếu đến khi tìm được một giải pháp, biến count được tăng lên 1. Khi hoàn thành thử các nước đi trên một hàng, các đánh dấu sẽ được hủy bỏ bằng cách gán lại giá trị 0 cho các phần tử tương ứng của col, diag1, và diag2. Cuối cùng, hàm main sẽ gọi hàm search để bắt đầu quá trình tìm kiếm giải pháp và in ra số lượng giải pháp tìm được.

- Độ phức tạp của thuật toán 8 con hậu:

+ Độ phức tạp của thuật toán 8 con hậu trong trường hợp xấu nhất là **O(n!),** với n là số lượng ô trên một dòng hoặc cột của bàn cờ. Tuy nhiên, trong thực tế, việc sử dụng thuật toán quay lui như trên sẽ có hiệu quả tốt hơn khi số n lớn hơn khoảng 12.

# Bài thực hành 8: Cài đặt bài toán balo 1, balo 2

- Mô tả bài toán balo:

+ Bài toán balo (hay còn gọi là bài toán cái túi) là một bài toán tối ưu hóa trong lĩnh vực khoa học máy tính. Bài toán đặt ra là cho trước một túi có khả năng chứa được một trọng lượng tối đa và một tập hợp các vật phẩm với trọng lượng và giá trị riêng. Nhiệm vụ là chọn một tập con các vật phẩm để đưa vào túi sao cho tổng trọng lượng của chúng không vượt quá trọng lượng tối đa của túi và tổng giá trị của chúng là lớn nhất có thể.

+ Bài toán balo được ứng dụng rộng rãi trong đời sống, chẳng hạn như trong việc đóng gói hàng hóa, sắp xếp thư viện, quản lý dữ liệu, tối ưu hóa chi phí và tài nguyên trong các công ty sản xuất và đa ngành khác.

+ Về mặt thuật toán, bài toán balo có thể giải quyết được bằng nhiều phương pháp khác nhau, bao gồm cả thuật toán tham lam, quy hoạch động và tìm kiếm nhánh cận.

- Cài đặt thuật toán Quy hoạch động giải bài toán balo bằng C++:

int knapsack(int W, vector<int> wt, vector<int> val, int n) {

int K[n + 1][W + 1];

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

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

if (i == 0 || w == 0)

K[i][w] = 0;

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

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

K[i][w] = K[i - 1][w];

}

}

return K[n][W];

}

- Trong đoạn code trên, ta sử dụng ma trận K để lưu trữ giá trị tối đa của các vật phẩm có thể chứa được trong balo, dựa trên trọng lượng của chúng và khả năng chứa của balo. Ta sử dụng hai vòng lặp để xác định giá trị tối đa có thể chứa được trong balo với số lượng các vật phẩm tương ứng. Cuối cùng, ta trả về giá trị tối đa của vật phẩm có thể chứa được trong balo.

- Độ phức tạp của thuật toán Quy hoạch động giải bài toán balo:

+ Độ phức tạp của thuật toán trên là **O(nW)**, với n là số lượng vật phẩm và W là trọng lượng tối đa của balo.

# Bài thực hành 9: Cài đặt bài toán nhân xích ma trận

# Bài thực hành 10: Cài đặt bài toán tìm chuỗi chung dài nhất

- Mô tả bài toán tìm chuỗi chung dài nhất:

+ Bài toán tìm chuỗi chung dài nhất (Longest Common Subsequence - LCS) là một bài toán cơ bản trong lý thuyết xâu ký tự. Cho trước hai xâu ký tự X và Y, ta cần tìm chuỗi con chung dài nhất của X và Y.

Ví dụ:

• X = "AGGTAB"

• Y = "GXTXAYB"

+ Kết quả là chuỗi chung dài nhất của X và Y là "GTAB" với độ dài là 4.

+ Bài toán này được áp dụng rất nhiều trong các lĩnh vực như xử lý ngôn ngữ tự nhiên, truy vấn dữ liệu, mã hóa, giải mã, ...

+ Thuật toán tiêu biểu để giải bài toán LCS là thuật toán đệ quy và quy hoạch động.

- Cài đặt thuật toán đệ quy giải bài toán tìm chuỗi chung dài nhất bằng C++:

int lcs(string X, string Y, int m, int n)

{

if (m == 0 || n == 0) // nếu một trong hai xâu có độ dài bằng 0

return 0;

else if (X[m - 1] == Y[n - 1]) // nếu hai ký tự cuối cùng của hai xâu giống nhau

return 1 + lcs(X, Y, m - 1, n - 1);

else

return max(lcs(X, Y, m, n - 1), lcs(X, Y, m - 1, n));

}

- Độ phức tạp của thuật toán đệ quy tìm chuỗi chung dài nhất:

Độ phức tạp của thuật toán đệ quy giải bài toán tìm chuỗi chung dài nhất là O(2^n), trong đó n là độ dài của hai chuỗi s và t. Bởi vì mỗi lần gọi đệ quy sẽ có hai trường hợp: ký tự cuối của cả hai chuỗi giống nhau hoặc khác nhau. Vì thế số lần gọi đệ quy tăng theo cấp số nhân với số bậc là n, do đó độ phức tạp là **O(2^n).**

# Bài thực hành 11: Cài đặt bài toán xếp lịch

- Mô tả bài toán xếp lịch:

+ Bài toán xếp lịch (hay còn gọi là bài toán lập lịch) là bài toán tìm cách sắp xếp các công việc trong một thời gian nhất định sao cho hoàn thành các công việc đó trong thời gian ngắn nhất hoặc tối ưu nhất có thể.

+ Bài toán xếp lịch là một trong những bài toán kinh điển trong lý thuyết tối ưu và quản lý dự án. Bài toán có ứng dụng rộng rãi trong nhiều lĩnh vực, từ kinh doanh, sản xuất đến giáo dục và y tế.

+ Bài toán xếp lịch thường được mô hình hóa bằng các công việc cần phải thực hiện và các ràng buộc thời gian hoàn thành của chúng. Các ràng buộc này bao gồm:

• Ràng buộc bắt đầu: công việc phải bắt đầu sau một thời gian nhất định.

• Ràng buộc kết thúc: công việc phải kết thúc trước một thời gian nhất định.

• Ràng buộc trùng lặp: các công việc không thể được thực hiện đồng thời.

• Ràng buộc ưu tiên: một công việc có thể ưu tiên hơn các công việc khác.

+ Các thuật toán giải bài toán xếp lịch thường tập trung vào việc tìm ra lịch làm việc tối ưu để đảm bảo hoàn thành tất cả các công việc trong thời gian ngắn nhất có thể và đáp ứng các ràng buộc thời gian. Các thuật toán này có thể dựa trên phương pháp tham lam, quy hoạch động, chia để trị, v.v. Tuy nhiên, do tính khó giải của bài toán, việc tìm ra lịch làm việc tối ưu vẫn là một thách thức đối với các nhà toán học và kỹ sư.

- Cài đặt thuật toán tham lam giải bài toán xếp lịch bằng C++:

struct Event {

int start\_time;

int end\_time;

};

bool compareEvents(Event a, Event b) {

return a.end\_time < b.end\_time;

}

int scheduleEvents(vector<Event>& events) {

sort(events.begin(), events.end(), compareEvents);

int count = 1;

int current\_end\_time = events[0].end\_time;

for (int i = 1; i < events.size(); i++) {

if (events[i].start\_time >= current\_end\_time) {

count++;

current\_end\_time = events[i].end\_time;

}

}

return count;

}

+ Trong đó, chúng ta định nghĩa một kiểu dữ liệu Event để lưu trữ thông tin về một sự kiện bao gồm thời gian bắt đầu và kết thúc của sự kiện. Chúng ta cũng định nghĩa một hàm compareEvents để so sánh hai sự kiện với nhau, dựa trên thời gian kết thúc của chúng.

+ Trong hàm scheduleEvents, chúng ta sắp xếp các sự kiện theo thời gian kết thúc và duyệt qua danh sách sự kiện để đếm số lượng sự kiện có thể được xếp lịch. Nếu sự kiện tiếp theo có thời gian bắt đầu lớn hơn hoặc bằng thời gian kết thúc của sự kiện hiện tại, chúng ta tăng biến đếm và cập nhật thời gian kết thúc của sự kiện hiện tại.

- Độ phức tạp của thuật toán tham lam:

+ Độ phức tạp của thuật toán tham lam giải bài toán xếp lịch là O(nlogn), với n là số lượng công việc cần phải xếp lịch. Điều này bởi vì trong thuật toán, ta sắp xếp các công việc theo thời gian kết thúc tăng dần, sử dụng hàm `sort()`, có độ phức tạp là O(nlogn). Sau đó, ta duyệt qua các công việc và xếp lịch nếu thời gian bắt đầu tiếp theo lớn hơn thời gian kết thúc công việc hiện tại. Quá trình này có độ phức tạp là O(n). Tổng độ phức tạp của thuật toán là O(nlogn) + O(n) = **O(nlogn).**

# Bài thực hành 12: Cài đặt bài toán tìm đường đi ngắn nhất

- Mô tả bài toán tìm đường đi ngắn nhất:

+ Bài toán tìm đường đi ngắn nhất là bài toán trong lĩnh vực giải thuật đồ thị, có nhiều ứng dụng trong đời sống thực tế như tìm đường đi ngắn nhất trên bản đồ, tính khoảng cách giữa các thành phố, tính đường đi ngắn nhất giữa các điểm trong mạng lưới máy tính, ...

+ Bài toán tìm đường đi ngắn nhất có thể được mô hình hoá thành một đồ thị, trong đó đỉnh của đồ thị biểu thị cho các điểm cần được kết nối và cạnh của đồ thị biểu thị cho khoảng cách giữa các điểm. Ví dụ, trong bài toán tìm đường đi ngắn nhất trên bản đồ, các địa điểm được biểu thị bởi các đỉnh và khoảng cách giữa các địa điểm được biểu thị bởi các cạnh.

+ Thuật toán tìm đường đi ngắn nhất có rất nhiều phương pháp, trong đó phương pháp phổ biến nhất là thuật toán Dijkstra và thuật toán Bellman-Ford.

- Cài đặt thuật toán Dijkstra tìm đường đi ngắn nhất bằng C++:

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

const int INF = numeric\_limits<int>::max();

// Định nghĩa cấu trúc đỉnh

struct Vertex {

int id;

int dist;

bool operator>(const Vertex& other) const {

return dist > other.dist;

}

};

// Định nghĩa cấu trúc đồ thị

class Graph {

public:

Graph(int n) : adj\_list(n) {}

void add\_edge(int u, int v, int w) {

adj\_list[u].push\_back({v, w});

}

vector<int> dijkstra(int s) {

vector<int> dist(adj\_list.size(), INF);

priority\_queue<Vertex, vector<Vertex>, greater<Vertex>> pq;

pq.push({s, 0});

dist[s] = 0;

while (!pq.empty()) {

auto u = pq.top();

pq.pop();

if (u.dist != dist[u.id]) continue;

for (const auto& [v, w] : adj\_list[u.id]) {

int new\_dist = u.dist + w;

if (new\_dist < dist[v]) {

dist[v] = new\_dist;

pq.push({v, new\_dist});

}

}

}

return dist;

}

private:

vector<vector<pair<int, int>>> adj\_list;

};

// Hàm main để kiểm tra kết quả

int main() {

Graph g(6);

g.add\_edge(0, 1, 5);

g.add\_edge(0, 2, 1);

g.add\_edge(1, 2, 2);

g.add\_edge(1, 3, 1);

g.add\_edge(2, 3, 4);

g.add\_edge(2, 4, 8);

g.add\_edge(3, 4, 3);

g.add\_edge(3, 5, 6);

g.add\_edge(4, 5, 7);

auto dist = g.dijkstra(0);

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

cout << "Shortest distance from 0 to " << i << " is " << dist[i] << "\n";

}

return 0;

}

- Độ phức tạp của thuật toán Dijkstra tìm đường đi ngắn nhất:

+ Độ phức tạp của thuật toán Dijkstra là O((V+E) log E), trong đó V là số lượng cạnh, E là số lượng đỉnh trong đồ thị.

+ Trong đoạn code trên, việc sử dụng Heap (hàng đợi ưu tiên) giúp giảm độ phức tạp của thuật toán, làm cho độ phức tạp của đoạn code trên là **O((V+E) log E).**

# Bài thực hành 13: Cài đặt bài toán tìm đường đi ngắn nhất

- Mô tả bài toán tìm đường đi ngắn nhất:

+ Bài toán tìm đường đi ngắn nhất là bài toán trong lĩnh vực giải thuật đồ thị, có nhiều ứng dụng trong đời sống thực tế như tìm đường đi ngắn nhất trên bản đồ, tính khoảng cách giữa các thành phố, tính đường đi ngắn nhất giữa các điểm trong mạng lưới máy tính, ...

+ Bài toán tìm đường đi ngắn nhất có thể được mô hình hoá thành một đồ thị, trong đó đỉnh của đồ thị biểu thị cho các điểm cần được kết nối và cạnh của đồ thị biểu thị cho khoảng cách giữa các điểm. Ví dụ, trong bài toán tìm đường đi ngắn nhất trên bản đồ, các địa điểm được biểu thị bởi các đỉnh và khoảng cách giữa các địa điểm được biểu thị bởi các cạnh.

+ Thuật toán tìm đường đi ngắn nhất có rất nhiều phương pháp, trong đó phương pháp phổ biến nhất là thuật toán Dijkstra và thuật toán Bellman-Ford.

- Cài đặt thuật toán Ford-Bellman tìm đường đi ngắn nhất bằng C++:

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

// Định nghĩa cấu trúc cạnh của đồ thị

struct Edge {

int source, dest, weight;

};

// Hàm để tìm đường đi ngắn nhất giữa hai đỉnh sử dụng thuật toán Ford-Bellman

void BellmanFord(vector<Edge>& edges, int numVertices, int source, int destination) {

// Khởi tạo khoảng cách từ đỉnh nguồn tới các đỉnh khác là vô cùng

vector<int> distance(numVertices, INT\_MAX);

distance[source] = 0;

// Lặp lại numVertices - 1 lần để cập nhật khoảng cách tới các đỉnh

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

// Duyệt qua tất cả các cạnh và cập nhật khoảng cách

for (auto edge : edges) {

int u = edge.source;

int v = edge.dest;

int weight = edge.weight;

if (distance[u] != INT\_MAX && distance[u] + weight < distance[v]) {

distance[v] = distance[u] + weight;

}

}

}

// Kiểm tra xem có chu trình âm hay không

for (auto edge : edges) {

int u = edge.source;

int v = edge.dest;

int weight = edge.weight;

if (distance[u] != INT\_MAX && distance[u] + weight < distance[v]) {

cout << "Đồ thị có chu trình âm" << endl;

return;

}

}

// In ra đường đi ngắn nhất từ đỉnh nguồn tới đỉnh đích

cout << "Độ dài đường đi ngắn nhất từ đỉnh " << source << " tới đỉnh " << destination << " là: " << distance[destination] << endl;

}

int main() {

// Khởi tạo đồ thị có trọng số âm

vector<Edge> edges = {

{0, 1, 5},

{0, 2, 2},

{1, 3, 1},

{2, 1, -2},

{2, 3, 6},

{3, 4, -1}

};

int numVertices = 5;

int source = 0;

int destination = 4;

// Gọi hàm để tìm đường đi ngắn nhất

BellmanFord(edges, numVertices, source, destination);

return 0;

}

- Độ phức tạp của thuật toán Ford-Bellman tìm đường đi ngắn nhất:

+ Độ phức tạp của thuật toán Ford-Bellman là O(V\*E), với V là số lượng đỉnh trong đồ thị và E là số lượng cạnh trong đồ thị. Thuật toán sử dụng hai vòng lặp lồng nhau, vòng lặp bên ngoài lặp V-1 lần, vòng lặp bên trong lặp E lần để cập nhật khoảng cách từ đỉnh nguồn đến các đỉnh khác trong đồ thị. Do đó, độ phức tạp của thuật toán là **O(V\*E).**

# Bài thực hành 14: Cài đặt bài toán đổi tiền

- Mô tả bài toán đổi tiền:

+ Bài toán đổi tiền là bài toán nhằm tìm cách đổi một số tiền nhất định thành các tờ tiền có mệnh giá khác nhau sao cho số tờ tiền đổi được là ít nhất.

+ Ví dụ: giả sử chúng ta cần đổi 82 đồng, với các tờ tiền có mệnh giá là 1, 5, 10 và 25 đồng. Ta có thể đổi 82 đồng bằng cách sử dụng các tờ tiền như sau:

• 8 tờ 10 đồng

• 1 tờ 10 đồng và 6 tờ 5 đồng và 2 tờ 1 đồng

• 3 tờ 25 đồng, 1 tờ 5 đồng, 2 tờ 1 đồng

• ...

+ Tuy nhiên, cách đổi tiền tốt nhất là sử dụng 3 tờ 25 đồng và 1 tờ 5 đồng, vì vậy số tờ tiền đổi được là 4.

+ Bài toán đổi tiền được áp dụng rộng rãi trong thực tế, ví dụ như trong ngân hàng, cửa hàng, trạm xăng dầu...

- Cài đặt thuật toán tham lam giải bài toán đổi tiền bằng C++:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Định nghĩa cấu trúc đồng tiền

struct Coin {

int value;

string name;

};

// Hàm để sắp xếp các đồng tiền theo giá trị giảm dần

bool compareCoins(Coin a, Coin b) {

return a.value > b.value;

}

// Hàm để đổi tiền bằng thuật toán tham lam

vector<string> getChange(int amount, vector<Coin> coins) {

// Sắp xếp các đồng tiền theo giá trị giảm dần

sort(coins.begin(), coins.end(), compareCoins);

vector<string> result;

// Duyệt qua các đồng tiền và đổi tiền cho đến khi đổi đủ số tiền cần

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

int count = amount / coins[i].value;

amount = amount % coins[i].value;

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

result.push\_back(coins[i].name);

}

}

return result;

}

int main() {

// Khởi tạo danh sách đồng tiền và số tiền cần đổi

vector<Coin> coins = { {500, "VND500"}, {200, "VND200"}, {100, "VND100"}, {50, "VND50"}, {20, "VND20"}, {10, "VND10"}, {5, "VND5"}, {2, "VND2"}, {1, "VND1"} };

int amount = 1234;

// Gọi hàm để đổi tiền

vector<string> change = getChange(amount, coins);

// In ra kết quả

cout << "Số tiền cần đổi: " << amount << endl;

cout << "Các đồng tiền cần đổi: ";

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

cout << change[i] << " ";

}

cout << endl;

return 0;

}

+ Trong đó, hàm getChange sử dụng thuật toán tham lam để đổi tiền. Cụ thể, thuật toán này sẽ sắp xếp các đồng tiền theo giá trị giảm dần, sau đó duyệt qua từng đồng tiền và lấy đủ số lượng cần thiết để đổi đủ số tiền cần.

- Độ phức tạp của thuật toán tham lam giải bài toán đổi tiền:

+ Độ phức tạp của thuật toán tham lam giải bài toán đổi tiền là O(nlogn), với n là số lượng loại tiền tệ có trong hệ thống. Cụ thể, thuật toán sử dụng heap để sắp xếp các đồng tiền theo giá trị giảm dần, sau đó lần lượt chọn đồng tiền có mệnh giá lớn nhất và trừ đi từ số tiền cần đổi cho đến khi số tiền cần đổi bằng 0. Do số lượng loại tiền tệ n không thay đổi, nên độ phức tạp của thuật toán sẽ phụ thuộc vào độ phức tạp của heapify, tức là O(logn), được thực hiện n lần, nên độ phức tạp tổng thể là **O(nlogn).**