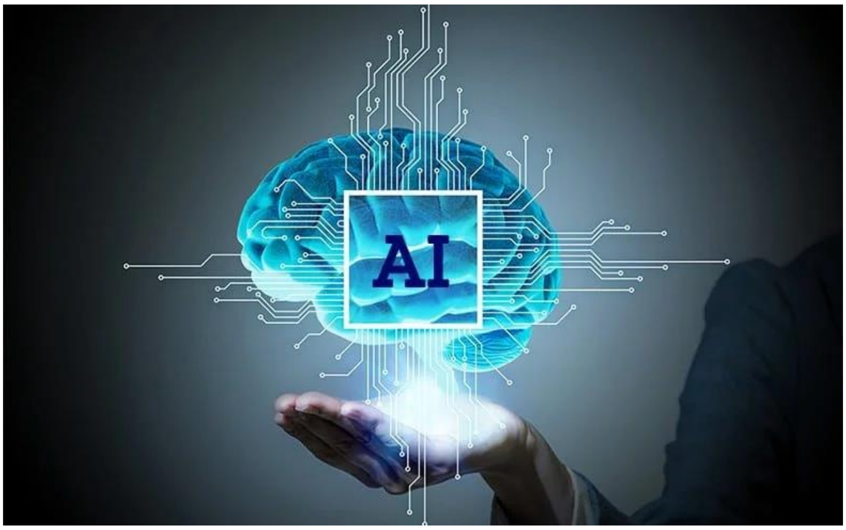
BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

KHOA TOÁN - TIN HỌC

BÀI BÁO CÁO THỰC HÀNH TUẦN 3

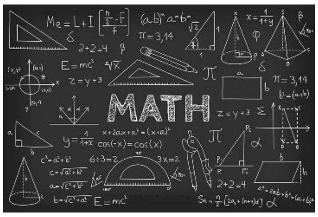


MÔN HỌC: **Phân Tích Thuật Toán**

Sinh Viên: **Trần Công Hiếu - 21110294**

Lớp: **21TTH**

TP.HCM, ngày 21 tháng 04 năm 2024



TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM
KHOA TOÁN – TIN HỌC

BÀI BÁO CÁO THỰC HÀNH TUẦN 3

HK1 - NĂM HỌC: 2024-2025

MÔN: PHÂN TÍCH THUẬT TOÁN

SINH VIÊN: TRẦN CÔNG HIẾU

MSSV: 21110294

LỚP: 21TTH

NHẬN XÉT

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

Tp.HCM, Ngày……Tháng……Năm 2021

Giảng viên Bộ môn

Mục Lục

Bài 1.

1. Ý tưởng.

Như trong phần hướng dẫn có nói đến "Do Output: Chỉ ra độ phức tạp của $f(n) = O(n^{\alpha})$ nên $f(n) \sim n^{\alpha}$. Ta sẽ lấy log cả 2 vế $\log(f(n)) \sim \alpha \log(n)$ thì lúc này ta sẽ xấp xỉ được giá trị của α". Cụ thể vì là xấp xỉ nên ta sẽ cộng thêm một hằng số b để dấu bằng xảy ra, lúc này: $\log\big(f(n)\big) = \alpha \log(n) + b$. Ở đây viết $\log(n)$ được hiểu là $\log_2(n)$, ta sẽ qui ước cho toàn bộ bài này.

Từ đây, với hàm f(n) cần kiểm tra cho ta cặp tương ứng $(n, f(n))$. Và vì n chứa khoảng giá trị từ a đến b với step tự chọn $(n_1, n_2, \dots, n_k)$ với k = ((b-a)/step)+1 ,nên ta có được ma trận tương ứng là:

$$\begin{cases} \log\big(f(n_1)\big) = \alpha * \log(n_1) + b * 1 \\ \log\big(f(n_2)\big) = \alpha * \log(n_2) + b * 1 \\ \dots \\ \log\big(f(n_k)\big) = \alpha * \log(n_k) + b * 1 \end{cases}$$

$$\Rightarrow \begin{pmatrix} \log(f(n_1)) \\ \log(f(n_2)) \\ \dots \\ \log(f(n_k)) \end{pmatrix} = \begin{pmatrix} \log(n_1) & 1 \\ \log(n_2) & 1 \\ \dots & \dots \\ \log(n_k) & 1 \end{pmatrix} * \begin{pmatrix} \alpha \\ b \end{pmatrix}$$

$$\Leftrightarrow \qquad Y = A * X$$

Lúc này, X được tính trực tiếp bằng công thức:

$$X = (A^T A)^{-1} A^T Y$$

Dễ dàng tìm ra được α và b. Vì ta chỉ quan tâm α cho bài toán và trong code thì ta chỉ cần lấy phần tử tại index [0][0] là được.

Hơn nữa để f(n) có độ phức tạp là $O(n^{\alpha})$ thì f(n) phải có dạng là 1 đa thức bậc α.

$$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_{\alpha} n^{\alpha}$$

Thế các giá trị của f(n) và n tương ứng vào phương trình trên để tìm hệ số $a_0, a_1, a_2, \dots, a_\alpha$. Với ý tưởng vừa nêu và cách triển khai từ hệ thành ma trận như đã làm ở trên, ta cũng làm tương tự.

$$\begin{cases} f(n_1) = a_0 + a_1 n_1 + a_2 n_1^2 + \cdots + a_\alpha n_1^\alpha \\ f(n_2) = a_0 + a_1 n_2 + a_2 n_2^2 + \cdots + a_\alpha n_2^\alpha \\ \dots \\ f(n_k) = a_0 + a_1 n_k + a_2 n_k^2 + \cdots + a_\alpha n_k^\alpha \end{cases}$$

$$\Rightarrow \begin{pmatrix} f(n_1) \\ f(n_2) \\ \dots \\ f(n_k) \end{pmatrix} = \begin{pmatrix} 1 & n_1 & n_1^2 & \dots & n_1^\alpha \\ 1 & n_2 & n_2^2 & \dots & n_2^\alpha \\ & & \dots & & \\ 1 & n_k & n_k^2 & \dots & n_k^\alpha \end{pmatrix} * \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_\alpha \end{pmatrix}$$

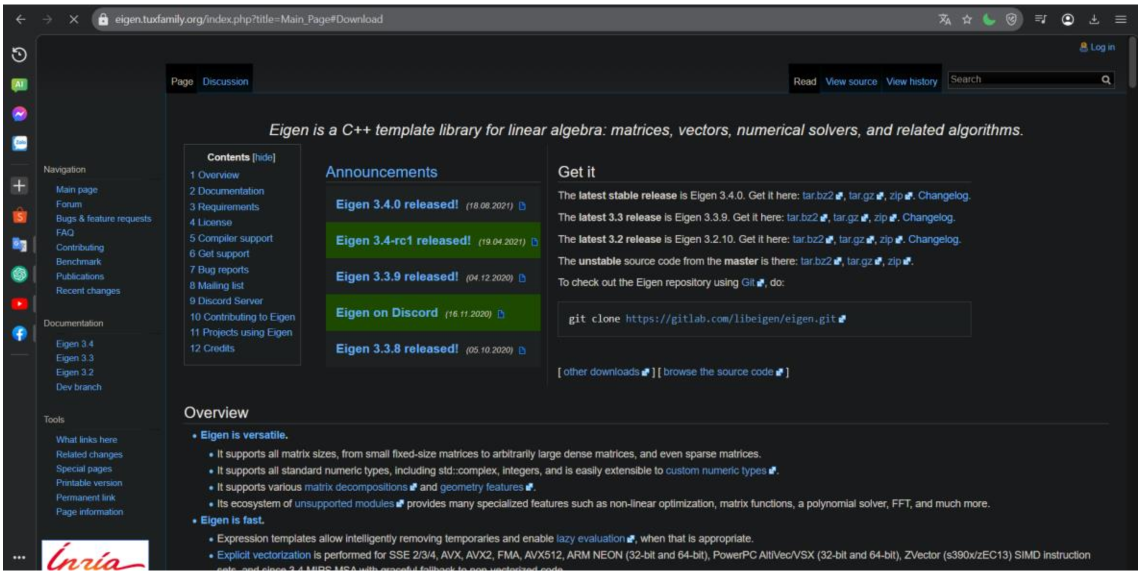
$$\Leftrightarrow \quad Y = N * A$$

Như đã nói, "Để f(n) có độ phức tạp là $O(n^\alpha)$ thì f(n) phải có dạng là 1 đa thức bậc α.", điều này chứng tỏ hệ số trước n^α phải khác 0. Tới đây ta chỉ cần giải hệ bằng việc đưa về $A = Y * N^{-1}$ và xem giá trị cuối cùng trong mảng A chính là hệ số trước n^α. Tuy nhiên vì $k \geq \alpha$ nên việc nghịch đảo sẽ vi phạm, do đó ta chọn $k = \alpha$ (Bởi chỉ cần $k = \alpha$ dòng là đủ để tìm hệ số $a_0, \dots, a_\alpha$).

Cuối cùng ta sẽ in ra thông báo, nếu có thì in ra $f(n) = O(n^\alpha)$, ngược lại thì thông báo không có.

2. Cài đặt thư viện.

Ở phần đầu chương trình vẫn như thường lệ là include các thư viện, tuy nhiên có thư viện ít khi dùng là vector và đặc biệt hơn là Eigen/Dense.



Để cài đặt thư viện này thì ta cần mở trình duyệt và tìm kiếm từ khóa "Eigen" (link tải: https://eigen.tuxfamily.org/index.php?title=Main_Page#Download) ở phía Get it ta chọn tệp muốn tải xuống (ở đây em chọn .zip) và cài đặt cho Visual Studio giống như trong video hướng dẫn (link: https://www.youtube.com/watch?v=6mMjv-tA5Jk) từ đoạn [1:35, 2:17].

3. Trình bày đoạn mã.

```
#include <iostream>
#include <vector>
#include "Eigen/Dense"
#include <cmath>

using namespace std;

double f(int n, int choose_f) {
    switch(choose_f){
        case 1:
            return n*n;
        case 2:
            return pow(n,3) + cos(n)*pow(n,4);
        case 3:
            return pow(n,n);
        case 4:
            return pow(n,3)+ n*n + n + 1;
    }
}

// In các phần tử của ma trận
void cout_matrix(vector<vector<double>> x) {
    for (const auto& row : x) {
        for (double element : row) {
            std::cout << element << " ";
        }
        std::cout << std::endl;
    }
}

```

Đầu chương trình ta include các thư viện cần thiết cho đoạn mã, các thư viện hầu hết chỉ cần gọi, trừ thư viện Eigen/Dense cần để thao tác với ma trận thì được hướng dẫn cài đặt ở mục trên.

"double f(int n, int choose_f){}": Định nghĩa hàm f(n) để thuận tiện cho việc kiểm tra các trường hợp của hàm f(n) trong bài. Do đó, ở hàm main ta sẽ khởi tạo biến choose_f để người dùng chọn hàm cần kiểm tra, trong hàm có sử dụng cấu trúc swith case mà không có default bởi tí nữa ở hàm main, ta sẽ dùng vòng lặp để yêu cầu người dùng nhập chính xác giá trị cho choose_f trong đoạn từ 1 đến 4. Và trong đoạn có sử dụng hàm pow() của thư viện cmath.

"void cout_matrix(vector<vector<double>> x){}": Định nghĩa hàm cout_matrix() để in ra các giá trị trong ma trận x, ở đây ta biểu diễn ma trận x theo kiểu dữ liệu vector<vector<>> với giá trị trong x thuộc kiểu double.

```cpp
vector<vector<double>> chuyenVi(const vector<vector<double>>& matrix) {
    // Lấy số hàng và số cột của ma trận ban đầu
    int rows = matrix.size();
    int cols = matrix[0].size();

    // Khởi tạo ma trận chuyển vị với số hàng và số cột ngược lại
    vector<vector<double>> transposed(cols, vector<double>(rows));

    // Lặp qua từng phần tử của ma trận ban đầu và gán vào vị trí tương ứng trong ma trận chuyển vị
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            transposed[j][i] = matrix[i][j];
        }
    }

    // Trả về ma trận chuyển vị
    return transposed;
}
```

```cpp
// Hàm nhân hai ma trận
vector<vector<double>> multiplyMatrices(const vector<vector<double>>& matrix1, const vector<vector<double>>& matrix2) {
    int rows1 = matrix1.size();
    int cols1 = matrix1[0].size();
    int rows2 = matrix2.size();
    int cols2 = matrix2[0].size();

    // Kiểm tra tính hợp lệ của phép nhân
    if (cols1 != rows2) {
        cout << "Không thể nhân hai ma trận này." << endl;
        return {};
    }

    // Khởi tạo ma trận kết quả với kích thước phù hợp
    vector<vector<double>> result(rows1, vector<double>(cols2, 0));

    // Thực hiện phép nhân ma trận
    for (int i = 0; i < rows1; ++i) {
        for (int j = 0; j < cols2; ++j) {
            for (int k = 0; k < cols1; ++k) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    return result;
}
```

```cpp
// Tính ma trận nghịch đảo của một ma trận vuông
vector<vector<double>> inverse(const vector<vector<double>>& A) {
    // Chuyển đổi ma trận vector sang Eigen::MatrixXd
    Eigen::MatrixXd eigA(A.size(), A[0].size());
    for (int i = 0; i < A.size(); i++) {
        for (int j = 0; j < A[0].size(); j++) {
            eigA(i, j) = A[i][j];
        }
    }

    // Tính ma trận nghịch đảo
    Eigen::MatrixXd invA = eigA.inverse();

    // Chuyển đổi ma trận Eigen::MatrixXd sang vector
    vector<vector<double>> invAVec(invA.rows(), vector<double>(invA.cols()));
    for (int i = 0; i < invA.rows(); i++) {
        for (int j = 0; j < invA.cols(); j++) {
            invAVec[i][j] = invA(i, j);
        }
    }

    return invAVec;
}
```

Định nghĩa 3 hàm có chức năng là chuyển vị ma trận, nhân hai ma trận và nghịch đảo ma trận để phục vụ cho công thức đã nêu ở phần ý tưởng là $X = (A^T A)^{-1} A^T Y$. Cả 3 có kiểu trả về là ma trận và đối số truyền vào là 1 hoặc 2 ma trận tùy vào chức năng của hàm.

```cpp
int check(double (*function)(int, int), int a, int b, int step, int choose_f, int &luythua) {
    int rows;
    int cols = 2;
    int cols_y = 1;
    rows = int((b - a) / step) + 1;

    //cout << "columns of matrix: " << rows << "\n";

    // Khởi tạo ma trận là một vector 2 chiều
    vector<vector<double>> matrixX(rows, vector<double>(cols));
    vector<vector<double>> matrixY(rows, vector<double>(cols_y));

    // Nhập dữ liệu cho ma trận
    for (int i = 0; i < rows; ++i) {
        matrixY[i][0] = log2(function(a + i * step, choose_f));
        //cout << "f("<< a+i*1 <<") = n^2 = " << function(a+i*step, choose_f) << "\n";
        matrixX[i][0] = log2(a + i * step);
        matrixX[i][1] = 1;
    }

    //cout_matrix(matrixX);
    //cout_matrix(matrixY);

    vector<vector<double>> result(2, vector<double>(1));
    result = multiplyMatrices(multiplyMatrices((inverse(multiplyMatrices(chuyenVi(matrixX), matrixX))), chuyenVi(matrixX)), matrixY);
    double alpha = result[0][0];
    //cout << "alpha = " << alpha;

```

```cpp

    if(alpha>0){
        int alpha_temp = ceil(alpha);
        //cout<<"\nalpha_temp = "<<alpha_temp;
        // f(n) = a0 + a1*n + a2*n^2 + ... + a_alpha * n^alpha
        // Y = N*A
        // (alpha_temp x 1) = (alpha_temp x alpha_temp) * (alphatemp x 1)
        vector<vector<double>> N(alpha_temp, vector<double>(alpha_temp));
        vector<vector<double>> Y(alpha_temp, vector<double>(1));
        vector<vector<double>> A(alpha_temp, vector<double>(1));
        A = multiplyMatrices(inverse(N), Y);
        if (A[alpha_temp-1][0] != 0){ // he so a_alpha khac 0 => deg{f(n)} = n^alpha => f(n) = O(n^alpha)
            luythua = alpha_temp;
            return 1;
        }
        else {return 0;}
    }
    else {
        return 0;
    }
}

```

Khởi tạo hàm check() để kiểm tra có hay không như đề bài hỏi. Giá trị trả về là 0 hoặc 1. Trong hàm ta tạo các ma trận matrixX và matrixY để lưu ma trận Y và A tương ứng trong công thức $X = (A^T A)^{-1} A^T Y$. Sau đó, ta tạo vector result để lưu ma trận kết quả phép nhân ma trận chính là X trong công thức vừa rồi. Lúc này, X là ma trận 2x1, với X[0][0] là α và X[1][0] là b. Vì chỉ quan tâm α để xét nếu $f(n) = O(n^{\alpha})$ thì phải có dạng đa thức có số mũ của n lớn nhất là α. Ta xét tiếp α nếu lớn hơn 0 để đảm bảo tồn tại đa thức làm tiền đề cần để tính tiếp bước sau. Lúc này, lưu biến alpha_temp để lấy nguyên trên của alpha, vì theo quan sát, đôi khi lấy log(n) sẽ tạo ra các số bé và sót mất các phần sau dấu phẩy sau khi tính toán như nhân 2 số nhỏ đó với nhau trong bước nhân ma trận hoặc nghịch đảo ma trận, dẫn đến trường hợp α vô cùng gần với giá trị nguyên cận trên nó. Do đó, ta xét cận trên của α. Cuối cùng ta khởi tạo các ma trận N, Y, A tương ứng cho công thức $Y = N * A$. Kết quả thu được ma trận A chứa các giá trị $\alpha_i, i = \overline{0, \alpha}$, xét giá trị cuối cùng, tức hệ số trước n^{α} và nếu khác 0 thì $f(n) = O(n^{\alpha})$ như đã thảo luận. Lưu trữ biến luythua là hệ số mũ cũng chỉ nhằm mục đích in thông báo cho đúng với từng hàm f(n) ta xét.

```cpp
int main() {
    int choose_f;
    int luythua;
    cout<<"Nhap ham f(n) muon kiem tra [1,4]:"; cin>>choose_f;
    while((4<choose_f) || (choose_f<1)){
        cout<<"Nhap choose_f de chon f(n) gia tri [1,4]!!!";
        cout<<"\nNhap lai choose_f: ";
        cin>>choose_f;
    }

    int a,b,step;
    cout<<"Nhap a,b nguyen duong (a<b).\n";
    cout<<"a = "; cin>>a;
    cout<<"b = "; cin>>b;
    cout<<"Nhap step:"; cin>>step;
    while((b<=a) || ((b-a)%step)!=0){
        cout<<"Nhap lai a, b va step!";
        cout<<"\na = "; cin>>a;
        cout<<"b = "; cin>>b;
        cout<<"step = ";cin>>step;
    }

    int result = check(f, a, b, step, choose_f, luythua);
    if (result == 1) {
        cout << "=> f(n) = O(n^"<< luythua<<")";
    }
    else {
        cout << "=> Khong phai!";
    }
    return 0;
}
```

Và cuối cùng hàm main() sẽ chỉ việc chỉnh lại các tham số và gọi hàm check(), bởi hàm check() đã giải quyết bài toán nên chủ yếu main() chỉ chỉnh các tham số sao cho phù hợp với việc nhập từ bên ngoài vào bài toán cũng như từng trường hợp phải xét.

Khởi tạo các biến choose_f để chọn hàm f(n) cần xét thay vì với từng trường hợp f(n) ta phải sử lại return của hàm f(n); luythua dùng để lưu lại số mũ sau khi giá trị α được tìm thấy và kết quả là đúng, hơn nữa vì là hàm check() cần trả về giá trị cho việc đưa ra thông báo nên không tiện để return luythua, và vì phải thay đổi giá trị khi đi qua check() nên ta thấy biến luythua này phải tham chiếu vào thay vì tham trị như a,b hay step. Tạo vòng lặp while để nếu người dùng nhập ngoài phạm vi hàm f(n) thì sẽ phải yêu cầu nhập lại.

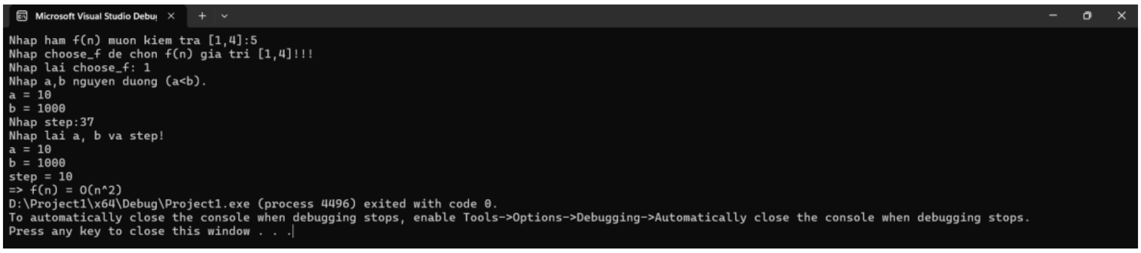
Tương tự, ta cũng khởi tạo biến a, b và step. Bởi ví dụ cho a và b cũng như bước nhảy step được ẩn thông qua giá trị của n, chứ không phải là hằng số nên ta vẫn phải nhập và tạo vòng lặp while kiểm tra. Chủ yếu là kiểm tra các cái đôi khi khó thấy như việc nhập a và b mà không biết liệu với step đưa vào thì b-a có chia hết cho step hay không thay vì những cái cơ bản như a, b phải nguyên dương.

Sau cùng là gọi hàm check() và lưu kết quả vào biến result để kiểm tra và đưa ra thông báo. Cũng có thể đưa check() vào if luôn thay vì phải lưu trung gian, tuy nhiên nếu như thế thì điều kiện quá dài, đôi khi sẽ ảnh hưởng đến việc quan sát, fix bug trong những bài nào đó sau này.

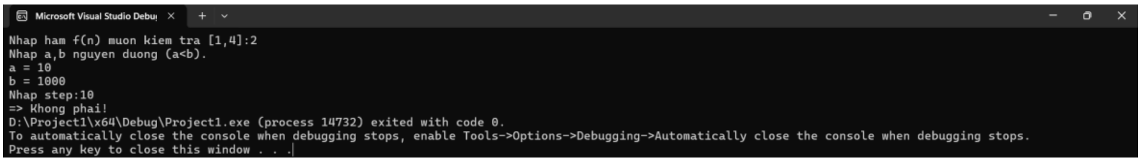
4. Kết quả.

Kết quả chạy ta sẽ thử với mọi trường hợp f(n) với a = 10, b = 1000 và step = 10. Cũng như thử nhập sai các điều kiện ràng buộc cho các biến choose_f hoặc a,b, step ở trường hợp đầu tiên, các trường hợp còn lại là nhập đúng.

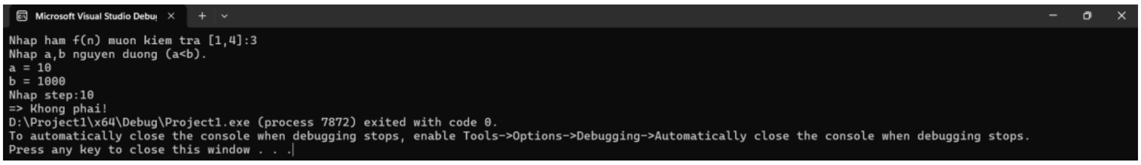
- $f(n) = n^2$.



- $f(n) = n^3 - \cos(n) \, . \, n^4$.



- $f(n) = n^n$.



- $f(n) = n^3 + n^2 + n + 1.$

4. Nhận xét.

Ở trường hợp $f(n) = n^3 + n^2 + n + 1$ và $f(n) = n^2$ thì có $f(n) = O(n^{\alpha})$ đúng với từng α tương ứng. Tuy nhiên ở 2 trường hợp còn lại rất đặc biệt. Với $f(n) = n^3 + \cos(n).n^4$ thì bình thường ta dễ nhận thấy đa thức này sẽ có độ phức tạp là $O(n^4)$ bằng việc xem biến số có bậc lớn nhất, nhưng bởi hàm chứa cos(n) trước n^4 làm thay đổi hệ số, tệ nhất là cos(n) = 0 nên lúc này $f(n) = O(n^3)$, do đó ta có thể có ý tưởng chương trình phân chia trường hợp cho hàm f(n) này. Với $f(n) = n^n$ thì dễ thấy là một dạng của hàm mũ và giá trị của n tăng lên theo cách mũ n. Trong trường hợp này, độ phức tạp của hàm là khó xác định bằng một hàm đa thức đơn giản. Theo khái niệm độ phức tạp trong lý thuyết thuật toán, nếu một hàm không thể được giới hạn bởi một hàm đa thức đơn giản thì ta thường không xác định được độ phức tạp của nó bằng cách đó. Ta chỉ có thể mô tả được tốc độ của hàm này rất nhanh, vượt xa khả năng của bất kỳ hàm đa thức nào để xác định.

Bài 2.

1. Ý tưởng.

Cả 2 phương pháp cho bài toán thì ở phương pháp truyền thống, ta đã tiếp cận từ khá sớm nên không có gì để phân tích. Với phương pháp cải tiến (Phương pháp nhân nhanh của Karatsuba) ta có:

procedure $KO(A, B)$

Input: 2 số A, B có n bit.

Output: Tích $C = A \times B$.

Giả sử $n = 2^k$, nếu cần thêm các chữ số 0 vào đằng trước A, B .

1. If $n = 1$ return $C = A \times B$ (tính bằng bảng cửu chương).

2. else chia A, B thành các phần có $n/2$ -bit: A_1, A_2, B_1, B_2 .

Như vậy: $A = 2^{n/2}A_1 + A_2, B = 2^{n/2}B_1 + B_2$ còn

$$C = 2^n A_1B_1 + 2^{n/2}(A_1B_2 + A_2B_1) + A_2B_2$$

3. $D = KO(A_1, B_1) = A_1 \times B_1$

4. $E = KO(A_2, B_2) = A_2 \times B_2$

5. $F = KO(A_1 - A_2, B_1 - B_2) = (A_1 - A_2) \times (B_1 - B_2)$

6. $G = D + E - F = A_1 \times B_2 + A_2 \times B_1$

7. return $C = 2^n D + 2^{n/2}G + E = A \times B$

Qua các bước của thuật toán Karatsuba-Ofman, ta có thể dễ dàng thấy được tính đúng đắn của thuật toán, nghĩa là thuật toán luôn trả lại $C = A \times B$. Bây giờ ta tìm hiểu độ phức tạp của thuật toán trên. Gọi $T(n)$ là số tính toán cần thiết để nhân 2 số n-bit bằng thủ tục $KO(A, B)$ ở trên. Ta có

$$T(n) = 3T(n/2) + O(n)$$

trong đó $3T(n/2)$ là số tính toán của các bước 3,4,5 (nhân các số có $n/2$ bit) còn $O(n)$ là số tính toán của các bước còn lại. Để ý là bước 7 thực chất chỉ gồm phép dịch trái (shift left) các số D, G, E và phép cộng để tính C. Áp dụng ***định lý tổng quát*** ta tính được:

$$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1,58})$$

2. Trình bày đoạn mã.

a) Phương pháp truyền thống.

Ở đây ta sử dụng 1 hàm bao "đóng gói" cho cả function tính phép nhân a và b multidata(a,b). Trong hàm này ta tạo các hàm con thực hiện từng function nhỏ:

```
def maxidx(data):
    maxi=0
    assert data>=0, "a has to be more than zero!!\n"
    stepdata = []
    while(data>=10**maxi):
        stepdata.append(data%(10**(maxi+1))//10**(maxi))
        maxi+=1

    return stepdata, maxi
```

"def maxidx(data):": Hàm maxidx có nhiệm vụ phân tách số thành từng chữ số. Ví dụ 1234 sẽ thành mảng [4,3,2,1] được lưu vào stepdata. Bên cạnh đó, maxi sẽ biểu thị cơ số 10 có số mũ lớn nhất. Ví dụ 21=2*10^1 + 1*10^0 thì maxi=1.

```
def sep(a,b):
    stepdata_a, maxi_a = maxidx(a)
    stepdata_b, maxi_b = maxidx(b)
    n_a=maxi_a//2
    if n_a*2<maxi_a:
        n_a+=1
        maxi_a+=1
        stepdata_a.append(0)
    n_b=maxi_b//2
    if n_b*2<maxi_b:
        n_b+=1
        maxi_b+=1
        stepdata_b.append(0)
    if maxi_a>=maxi_b:
        n=n_a
        for i in range(maxi_a-maxi_b):
          stepdata_b.append(0)
        maxi_b=maxi_a
    else:
        n=n_b

        for i in range(maxi_b-maxi_a):
            stepdata_a.append(0)
        maxi_a=maxi_b
    return maxi_a, maxi_b, n_a, n_b, stepdata_a, stepdata_b, n
```

"def sep(a,b):": Định nghĩa hàm sep() với 2 tham số đầu vào là 2 số cần xét, theo thiên hướng mở rộng khi giả sử ngay việc số có số chữ số không thuộc dạng 2^k và cả việc 2 số không cùng mức maxi. Kết quả sẽ trả về stepdata cả a và b cùng form, nếu a là 123 và b là 6543 thì stepdata_a=[3,2,1,0] và stepdata_b=[3,4,5,6].

```python
    maxi_a, maxi_b, n_a, n_b, stepdata_a, stepdata_b, n = sep(a,b)
    s=0
    for i in range(maxi_a):
        m=0
        print("step "+str(i+1))
        for j in range(maxi_a):
            print(int(stepdata_a[i])*int(stepdata_b[j])*10**(i+j))
            m+=int(stepdata_a[i])*int(stepdata_b[j])*10**(i+j)
        print("="+str(m))
        print()
        s+=m
    print("ket qua phuong phap 1 voi "+str(a)+"*"+str(b)+"="+str(s))

multidata(a,b)
```

Sau cùng, trong hàm multidata, gọi các tham số bằng việc sử dụng hàm sep vừa nãy, chạy 2 vòng lệnh for lấy từng chữ số (từ nhỏ đến lớn) của a nhân cho từng chữ số của b đồng thời mỗi idx i j sẽ “shift 10” qua bên trái bằng việc nhân 10**(i+j). Và cộng lại các giá trị qua các step để cho ra kết quả a*b.

b) Phương pháp cải tiến (Thuật toán nhân nhanh của Karatsuba).

Tương tự như trên, tuy nhiên sẽ có 1 chút thay đổi:

```python
    def sep_up(stepdata,n,maxi):
        up=""
        for i in range(0,n):
            up+=str(stepdata[maxi-i-1])
        return up


    def sep_down(stepdata,n,maxi):
        down=""
        for i in range(n,maxi):
            down+=str(stepdata[maxi-i-1])
        return down
    maxi_a, maxi_b, n_a, n_b, stepdata_a, stepdata_b, n = sep(a,b)
    return n,sep_up(stepdata_a,n,maxi_a),sep_down(stepdata_a,n,maxi_a),sep_up(stepdata_b,n,maxi_b),sep_down(s
```

Như đã nói ở trên, trong báo cáo này có mở rộng thêm cả những trường hợp số có các chữ số không bằng nhau giữa a và b đồng thời không theo dạng 2^k.

Đầu tiên với hàm sep_down và sep_up ta sẽ tách chữ số ra thành 2 phần lấy N/2=n=maxi/2 làm trung tâm. Ví dụ a=012345 với n=3 ta sẽ tách ra up là 012 và down là 345. Việc tách ra này để áp dụng thuật toán khi lấy 012*(3 chữ số đầu của b) và 345*(3 chữ số sau của b).

Tiếp tục khi áp dụng công thức như tài liệu hướng dẫn, sau khi nhân sẽ vẫn có tình trạng 2 chữ số có trên 2 chữ số nhân với nhau buộc ta phải thực hiện multidata thêm nhiều lần nữa vì thế trong báo cáo này đã sử dụng đệ quy thực hiện phương pháp ấy:

```
def dequy(a,b):
    n,x1,y1,x2,y2=multidata(a,b)
    # print(n)
    print("step : search "+str(a)+"*"+str(b)+"??")
    print("C = "+x1+"*"+x2)
    print("D = "+y1+"*"+y2)
    print("E = ("+x1+"+"+y1+")*("+x2+"+"+y2+")="+str(int(x1)+int(y1))+"*"+str(int(x2)+int(y2))+"-C1-D1")
    print("\n")
    search=0
    if (max(int(x1),int(x2))>=10):
        dequy(int(x1),int(x2))
    if (max(int(y1),int(y2))>=10):
        dequy(int(y1),int(y2))
    if max(int(x1)+int(y1),int(x2)+int(y2))>=10:
        dequy(int(x1)+int(y1),int(x2)+int(y2))
    else:

        C=int(x1)*int(x2)
        D=int(y1)*int(y2)
        E=(int(x1)+int(y1))*(int(x2)+int(y2))-int(x1)*int(x2)-int(y1)*int(y2)
        search=C*10**(n*2)+E*10**n+D

dequy(a,b)

# def multidata(a,b):
```

Đệ quy này sẽ thực hiện với 3 biến số C, D, E cho đến khi nào 2 chữ số ai và bi có số chữ số <=1 tức là không lớn hơn 10. Ta dừng vòng lặp đệ quy và cho ra các steps như tài liệu hướng dẫn.

3. Kết quả.

a) Phương pháp truyền thống.

Với a = 123456 và b = 926182, ta có kết quả:

Run: alo

```
C:\Users\PC-LENOVO\Downloads\Pycharm_code\venv\Scripts\python.exe C:\Users\PC-LENOVO\Downloads\Pycharm_code\alo.py
step 1
12
480
600
36000
120000
5400000
=5557092

step 2
100
4000
5000
300000
1000000
45000000
=46309100

step 3
800
32000
40000
2400000
8000000
360000000
=370472800
```

```
Run:  alo
step 4
6000
240000
300000
18000000
60000000
2700000000
=2778546000

step 5
40000
1600000
2000000
120000000
400000000
18000000000
=18523640000

step 6
200000
8000000
10000000
600000000
2000000000
90000000000
=92618200000

ket qua phuong phap 1 voi 123456*926182=114342724992

Process finished with exit code 0
```

Lấy N $= 2^k, k = 10, 11, \ldots, 32$. Ta sẽ tạo hàm tạo ngẫu nhiên 2 số có N $= 2^k$.

```python
import random

2 usages
def random_number_with_n_digits(n):
    lower_bound = 10 ** (n - 1)  # Lower bound inclusive
    upper_bound = 10 ** n - 1  # Upper bound inclusive
    return random.randint(lower_bound, upper_bound)
for i in range(10,33):
    print("- Với N = 2^",i)
    a = random_number_with_n_digits(i)
    b = random_number_with_n_digits(i)
    multidata(a,b)

```

Kết quả thu được:

```
alo
C:\Users\PC-LENOVO\Downloads\Pycharm_code\venv\Scripts\python.exe C:\Users\PC-LENOVO\Downloads\Pycharm_code\alo.py
- Với N = 2^ 10
ket qua phuong phap 1 voi:5511711729*6032691788=33250458085361581452
- Với N = 2^ 11
ket qua phuong phap 1 voi:51834805557*25787831688=1336707241284123090216
- Với N = 2^ 12
ket qua phuong phap 1 voi:671435075724*718154609178=482194194394970055394872
- Với N = 2^ 13
ket qua phuong phap 1 voi:2278332157162*4074233809519=9282457904023776279625078
- Với N = 2^ 14
ket qua phuong phap 1 voi:17679264357203*76340024125412=1349635467548413494437542636
- Với N = 2^ 15
ket qua phuong phap 1 voi:891125896991495*830808682919588=740355132795040407522204904060
- Với N = 2^ 16
ket qua phuong phap 1 voi:4523678618516319*2392488100213327=10822847263989755651956130783313
- Với N = 2^ 17
ket qua phuong phap 1 voi:98540790990765657*21584687030503542=2126972133273939753394400830457094
- Với N = 2^ 18
ket qua phuong phap 1 voi:174823627228872020*737760070058680362=128977891472285241228184666985271240
- Với N = 2^ 19
ket qua phuong phap 1 voi:5140692896572456989*8465855523635928116=43520363353763913918657505677505802724
- Với N = 2^ 20
ket qua phuong phap 1 voi:48134971309838662000*24005251380738294186=1155492086497302718014184248752619132000
- Với N = 2^ 21
ket qua phuong phap 1 voi:852695151204622447583*466273099494507890838=397588811076117372509477811227378540944554
- Với N = 2^ 22
ket qua phuong phap 1 voi:4965535840078425844430*7183409484264308507582=35669477248073704881905231930897151407468260
- Với N = 2^ 23
ket qua phuong phap 1 voi:12226721017851007686444*74638158072026947880798=912579936032937729665833733761226474672502312
```

```
- Với N = 2^ 24
ket qua phuong phap 1 voi:814293388333422478363883*346575501195635855785760=282214139181948434370919715478190298424169706080
- Với N = 2^ 25
ket qua phuong phap 1 voi:6484251080243587920255787*7965170152717146177005959=51648163167080139264752599775631614049593103234733
- Với N = 2^ 26
ket qua phuong phap 1 voi:99613425365614273301019063*12921526832628454325984336=1287157548751816730718437112506443164373829375397168
- Với N = 2^ 27
ket qua phuong phap 1 voi:136518062633794303000084126*443742626108798224138890745=60578883624405283134535686265046253679829643322813870
- Với N = 2^ 28
ket qua phuong phap 1 voi:9041775284389926535136635866*8375807432549275872795664628=75732168630433489269886989475637375519970445457492347848
- Với N = 2^ 29
ket qua phuong phap 1 voi:13431244930239461449008971981*62989582957995781269745409827=846028516762478816948249515389903284755904571014805057287
- Với N = 2^ 30
ket qua phuong phap 1 voi:690566395283688045649138551087*976754374418006017868366859080=674513747419416178405554745797254203517913993794160309819960
- Với N = 2^ 31
ket qua phuong phap 1 voi:9149658645670814317849592946206*3854541305749310319318794819692=35267737183244446862582224412275292090217440660175336825488552
- Với N = 2^ 32
ket qua phuong phap 1 voi:62223384258070333372662361848161*88620490840884568450040604174921=5514266854731163017297213348538747874760791277540405726086170281

Process finished with exit code 0
```

b) Phương pháp cải tiến (Thuật toán nhân nhanh của Karatsuba).

Với a = 123456 và b = 926182 thì ta có kết quả:

```
test
C:\Users\PC-LENOVO\Downloads\Pycharm_code\venv\Scripts\python.exe C:\Users\PC-LENOVO\Downloads\Pycharm_code\test.py
step : search 123456*926182??
C = 123*926
D = 456*182
E = (123+456)*(926+182)=579*1108-C1-D1


step : search 123*926??
C = 01*09
D = 23*26
E = (01+23)*(09+26)=24*35-C1-D1


step : search 23*26??
C = 2*2
D = 3*6
E = (2+3)*(2+6)=5*8-C1-D1


step : search 24*35??
C = 2*3
D = 4*5
E = (2+4)*(3+5)=6*8-C1-D1


step : search 456*182??
C = 04*01
D = 56*82
E = (04+56)*(01+82)=60*83-C1-D1
```

test

```
step : search 56*82??
C = 5*8
D = 6*2
E = (5+6)*(8+2)=11*10-C1-D1


step : search 11*10??
C = 1*1
D = 1*0
E = (1+1)*(1+0)=2*1-C1-D1


step : search 60*83??
C = 6*8
D = 0*3
E = (6+0)*(8+3)=6*11-C1-D1


step : search 6*11??
C = 0*1
D = 6*1
E = (0+6)*(1+1)=6*2-C1-D1


step : search 579*1108??
C = 05*11
D = 79*08
E = (05+79)*(11+08)=84*19-C1-D1
```

test

```
step : search 5*11??
C = 0*1
D = 5*1
E = (0+5)*(1+1)=5*2-C1-D1


step : search 79*8??
C = 7*0
D = 9*8
E = (7+9)*(0+8)=16*8-C1-D1


step : search 16*8??
C = 1*0
D = 6*8
E = (1+6)*(0+8)=7*8-C1-D1


step : search 84*19??
C = 8*1
D = 4*9
E = (8+4)*(1+9)=12*10-C1-D1


step : search 12*10??
C = 1*1
D = 2*0
E = (1+2)*(1+0)=3*1-C1-D1
```

```
step : search 12*10??
C = 1*1
D = 2*0
E = (1+2)*(1+0)=3*1-C1-D1



Process finished with exit code 0
```

4. Đánh giá.

Nhìn chung đoạn mã chạy tốt và đưa ra kết quả chính xác, ở đoạn mã a với phương pháp cổ điển thì nhân được từng chữ số của A với B (kết quả được dịch trái 1 ví trí sau mỗi lần nhân) và cộng lại kết quả với độ phức tạp $O(N^2)$, tuy nhiên ở đoạn mã phần b với độ phức tạp $O(N^{\log 3})$ thì đưa ra được từng quá trình như ví dụ nhưng chưa thể in ra kết quả nên cần cải thiện thêm. Hơn nữa, không chỉ dừng lại ở việc 2 số đầu vào có cùng số chữ số N mà đoạn mã trên đã phát triển hơn thông qua việc xét tồn tại 1 số có ít hơn số còn lại 1 chữ số, từ đó dùng các kĩ thuật để cân bằng và tính toán như bình thường.