Họ và tên: Lưu Quang Huy

MSV: 20001554

1)

1.3) + Trường hợp xấu nhất: khóa tìm kiếm không có trong danh sách hoặc khi nó nằm ở cuối danh sách. Nó duyệt qua toàn bộ các phần tử của danh sách để xác định rằng khóa không có trong danh sách, nên số lần so sánh sẽ là n => O(n)

+ Trường hợp trung bình sẽ phụ thuộc vào phân bố của các phần tử trong danh sách. Nếu khóa tìm kiếm xuất hiện ngẫu nhiên trong danh sách, thì trung bình số lần so sánh là n/2. Tuy nhiên, nếu khóa xuất hiện ở đầu danh sách, số lần so sánh sẽ là 1, trong khi nếu khóa xuất hiện ở cuối danh sách, số lần so sánh sẽ là n. Vì vậy, trường hợp trung bình của tìm kiếm tuần tự là O(n/2) = O(n).

+ Trường hợp tốt nhất khi khóa tìm kiếm là phần tử đầu tiên của danh sách. Trong trường hợp này, chỉ cần một lần so sánh để tìm thấy khóa, vì vậy số lần so sánh là 1. Do đó, trường hợp tốt nhất của tìm kiếm tuần tự là O(1).

1.4) a) Best case: Lấy lần đầu tiên 1 chiếc, lần thứ 2 lấy chiếc bên còn lại => 2 chiếc

Worst- case: đối với mỗi nhóm màu của găng tay, bạn lấy găng tay bên phải hoặc bên trái, chẳng hạn, bạn lấy 5 chiếc găng tay màu đỏ bên trái, sau đó là 4 chiếc găng tay màu vàng bên trái, sau đó là 2 chiếc găng tay màu xanh lá cây bên trái, bởi vì bạn đã lấy tất cả các găng tay bên trái, chiếc tiếp theo sẽ là một chiếc găng tay bên phải sẽ kết hợp với một trong những chiếc găng tay đã chọn, do đó, câu trả lời là 12 -> 5+4+2+1 = 12

b) Mất 2 chiếc trong 5 đôi: $C_{10}^{2} = 45$

Trường hợp còn 4 đôi hoàn chỉnh: $\frac{5}{45} = \frac{1}{9}$

Trường hợp còn 3 đôi hoàn chỉnh: 1 - $\frac{1}{9}$ = $\frac{8}{9}$

$$\Rightarrow \quad 3\,\frac{8}{9} + 4\,\frac{1}{9} = \frac{28}{9}$$

1.8)

a) Ta có 3nlog(23n) = 3nlog(6n). Do đó, giá trị của hàm số tăng lên khoảng 3nlog(6n) - nlog(2n) = nlog(6n/2n) = n*log(3) ≈ 1.585n.b) Ta có 3n. Do đó, giá trị của hàm số tăng lên gấp ba lần.

c) Ta có (3n)^2 = 9n^2. Do đó, giá trị của hàm số tăng lên 9 lần.

d) Ta có (3n)!. Do đó, giá trị hàm số tăng lên: P_{3n}^{27}

e) Ta có $\frac{(2^n)^3}{2^n} = (2^n)^2$ = 4^n

2)

2.1)

a) Ta áp dụng công thức $s = \frac{n*(n+1)}{2}$

Mà n = 100 nên thay vào ta có S = 5050

b) S = 10 + 20 + 30 + 40 + ... + 1000

Số số hạng của tổng trên là:

$$\frac{1000-10}{10}+1 = 100$$

⇨ Sum S= $\frac{1000+10}{2}*100 = 50500$

2.4)

a) Thuật toán trên tính tổng các số hạng của dãy:

$$S = \frac{1}{1!}+\frac{2}{2!}+\frac{3}{3!}+...+\frac{n}{n!}$$

b) Thuật toán có các hoạt động thực hiện như sau:

1. Khởi tạo biến Sum = 0 để lưu trữ tổng các số hạng của chuỗi.
2. Sử dụng vòng lặp for để lặp từ 1 đến n. Trong mỗi vòng lặp, tính giá trị của số hạng $\frac{i}{i!}$ bằng cách chia i cho giai thừa của i. Sau đó, cộng giá trị này vào biến Sum.
3. Sau khi vòng lặp kết thúc, trả về kết quả tổng Sum.

c) Vòng lặp for được sử dụng để duyệt qua từng số trong đoạn [1, n], do đó hoạt động cơ bản sẽ được thực hiện n lần.

=> Độ phức tạp sẽ là O(n)

d) Lớp hiệu quả của thuật toán trên là O(n). Độ phức tạp thời gian tăng đều theo kích thước của đầu vào n.

e) Có thể dùng thuật toán tốt hơn đó là thuật toán *chia để trị*.

Tính tổng của chuỗi số bằng cách chia chuỗi thành hai phần đều nhau, tính tổng của từng nửa và sau đó cộng lại. Thuật toán chia để trị có thể giảm đáng kể số lần thực hiện so với thuật toán trên.

Chia chuỗi từ 1 đến n thành hai phần đều nhau. Sau đó, ta tính tổng của từng nửa bằng cách áp dụng cùng thuật toán cho mỗi nửa, đến khi chuỗi chỉ còn một phần tử. Khi đó, tổng của chuỗi ban đầu sẽ bằng tổng của hai tổng con.

Công thức tính tổng của chuỗi số từ 1 đến n sẽ như sau:

+ Nếu n = 1, tổng là 1

+ Nếu n > 1, tổng sẽ là T(n) = T(n/2) + T(n/2) + n, với T(1) = 1

Thuật toán chia để trị có độ phức tạp thời gian là O(nlogn), hiệu quả hơn thuật toán trên

```
public class Demo {
    2 usages
    public static int divideAndConquerSum(int n) {
        if (n == 1) {
            return 1;
        } else {
            int leftSum = divideAndConquerSum( n: n/2);
            int rightSum = divideAndConquerSum( n: n - n/2);
            return leftSum + rightSum + n;
        }
    }
}
```

2.5)

a) Thuật toán thực hiện tính hiệu giữa sum các phần tử lớn hơn giá trị val và tổng các phần tử nhỏ hơn giá trị val trong một mảng chứa n số thực.

b) Thuật toán trên tìm tổng của tất cả các phần tử trong mảng A[0..n-1] mà lớn hơn một giá trị được định sẵn (val), trừ đi tổng của tất cả các phần tử trong mảng A[0..n-1] mà nhỏ hơn giá trị (val). Kết quả trả về sẽ là hiệu giữa tổng các phần tử lớn hơn giá trị val và tổng các phần tử nhỏ hơn giá trị val.

c) Thuật toán sẽ thực hiện n lần lặp, với n là số phần tử của mảng A[0..n-1]. Trong mỗi vòng lặp, thuật toán sẽ kiểm tra xem giá trị của phần tử thứ i có lớn hơn hay nhỏ hơn giá trị ngưỡng val hay không.

=> Số lần thực hiện cơ bản của thuật toán sẽ là 2n, với n là số phần tử của mảng A[0..n-1].

d) Cải tiến cho thuật toán trên có thể là sử dụng phương pháp đếm số lượng phần tử lớn hơn và nhỏ hơn giá trị ngưỡng val trước khi tính tổng. Việc này sẽ giúp giảm số lần thực hiện cơ bản của thuật toán từ 2n xuống còn 1 lần.

```java
public class Demo {
    no usages
    public static double improvedFoo(double[] A) {
        double val = 100;
        int countGreater = 0;
        int countLess = 0;
        double sumGreater = 0;
        double sumLess = 0;

        for (int i = 0; i < A.length; i++) {
            if (A[i] > val) {
                countGreater++;
                sumGreater += A[i];
            } else if (A[i] < val) {
                countLess++;
                sumLess += A[i];
            }
        }

        double avgGreater = countGreater > 0 ? sumGreater/countGreater : 0;
        double avgLess = countLess > 0 ? sumLess/countLess : 0;

        return avgGreater - avgLess;
    }
}

```

2.6)

a) Thuật toán kiểm tra các phần tử nằm trên đường chéo phụ (chéo từ góc trên bên phải xuống góc dưới bên trái) của ma trận, nếu phần tử A[i,j] bằng phần tử A[j,i] thì ma trận là đối xứng, ngược lại thì không đối xứng.

b) Thuật toán sử dụng hai vòng lặp lồng nhau để duyệt qua các phần tử nằm trên đường chéo phụ của ma trận (từ hàng thứ nhất đến hàng thứ n-1 và từ cột thứ hai đến cột thứ n), và so sánh giá trị của các phần tử tương ứng với nhau. Nếu giá trị các phần tử đối xứng bằng nhau thì ma trận là đối xứng và trả về giá trị true, ngược lại trả về giá trị false.

c) Ta có hai vòng lặp lồng nhau, vòng lặp bên ngoài chạy từ 0 đến n-2, vòng lặp bên trong chạy từ i+1 đến n-1. Vì thế, số lần so sánh được thực hiện trong thuật toán là $\frac{n*(n-1)}{2}$ lần.

d) $O(n^2)$

e) Ta có thể sử dụng tính chất đối xứng của ma trận để so sánh ma trận A và ma trận chuyển vị của A. Nếu chúng bằng nhau, ma trận A là đối xứng. Độ phức tạp của thuật toán này là O(n^2). Tuy nhiên, cách này sẽ giảm bớt số lần xét phần tử của ma trận, do đó có thể hiệu quả hơn khi xử lý các ma trận lớn.

```java
public class Demo {
    no usages
    public static boolean improvedEnigma(double[][] A) {
        int n = A.length;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (A[i][j] != A[j][i]) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

3)

+ Mã giả của thuật toán trên là:

ALGORITHM SelectionSort(A[0..n − 1])

//Input: An array A[0..n − 1] of n numbers

for i ← 0 to n − 2 do

minIndex ← i

for j ← i + 1 to n − 1 do

if A[j] < A[minIndex]

minIndex ← j

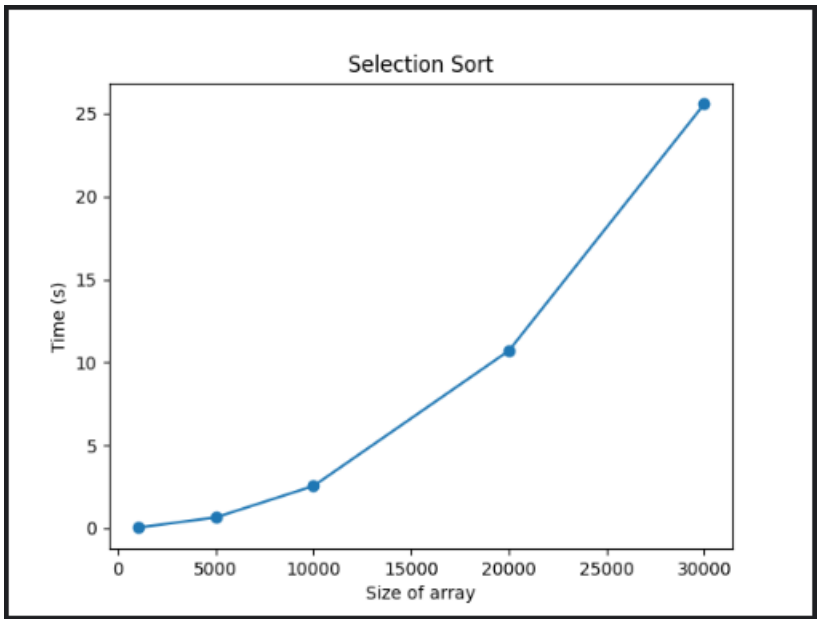
swap(A[i], A[minIndex])

+ Chỉ cần thực hiện với n − 1 phần tử đầu tiên, thay cho cả n phần tử vì khi đã sắp xếp được n-1 phần tử, phần tử cuối cùng đã nằm đúng vị trí của nó trong dãy. Do đó, việc sắp xếp phần tử cuối cùng không ảnh hưởng đến các phần tử trước đó đã được sắp xếp.

+ Trong trường hợp tốt nhất, khi mảng đã được sắp xếp, thuật toán sẽ vẫn phải tìm và so sánh tất cả các phần tử trong mảng để đảm bảo không có phần tử nào nhỏ hơn phần tử tại vị trí đang xét. Do đó, số lần so sánh sẽ là: n-1 + n-2 + ... + 1 = (n-1)*n/ => O(n^2).

Trong trường hợp xấu nhất, khi mảng được sắp xếp ngược lại, mỗi lần tìm phần tử nhỏ nhất và đổi chỗ, cần phải duyệt qua toàn bộ n phần tử của mảng chưa được sắp xếp. Do đó, số lần so sánh sẽ là: (n-1) + (n-2) + ... + 1 = (n-1)*n/2. Nên thời gian thực hiện trong trường hợp xấu nhất cũng là O(n^2).

+ Chạy flie **demo.py** sử dụng thư viện **matplot** để vẽ sơ đồ theo yêu cầu



Ta thấy thời gian thực hiện tăng theo kích thước của mảng