Luyện tập thêm 1: về độ phức tạp của thuật toán

Đỗ Mạnh Hùng – 20002053 – K65A5

Phần 1: Reinforcement

R – 4.2: The number of operations executed by algorithms A and B is 8nlogn and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

- Xét biểu thức: $8n\log n \leq 2n^2 \Leftrightarrow 4\log n \leq n$
- Bằng cách sử dụng lập trình:

```
import numpy as np
n = 9
while n < 100:
    if (n >= 4 * np.log(n)):
        print(n)
    n = n + 1
```

- Nhận thấy rằng biểu thức luôn đúng khi $n \geq 9 \Rightarrow n_0 = 9$

R – 4.3: The number of operations executed by algorithms A and B is $40n^2$ and $2n^3$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

- Xét biểu thức: $40n^2 \leq 2n^3 \Leftrightarrow 20 \leq n$
  $\Rightarrow n_0 = 20$

R – 4.6:

What is the sum of all the even numbers from 0 to 2n, for any integer $n \geq 1$?

- Áp dụng công thức tổng các số hạng của cấp số cộng với số số hạng là n, công sai d = 2, $u_1 = 2$:

$$S_n = \frac{n[2u_1 + (n-1)d]}{2} = \frac{n(2 * 2 + (n-1) * 2)}{2} = \frac{n(4 + 2n - 2)}{2} = n(n+1)$$

$\Rightarrow$ Tổng của số chẵn từ 0 đến 2n là $n(n+1)$

R – 4.8: Order the following functions by asymptotic growth rate.

$4n\log n + 2n$	2^{10}	$2^{\log n}$
$3n + 100\log n$	$4n$	2^n
$n^2 + 10n$	n^3	$n\log n$

- Sắp xếp: $2^{10}, 2^{\log n}, 4n, 3n + 100\log n, n\log n, 4n\log n + 2n, n^2 + 10n, n^3, 2^n$

R – 4.9: Give a big-Oh characterization, in terms of n, of the running time of the example1 method shown in Code Fragment 4.12.

```
// Returns the sum of the integers in given array.
 public static int example1(int[ ] arr) {
    int n = arr.length, total = 0;
    for (int j=0; j < n; j++) // loop from 0 to n-1
        total += arr[j];
    return total;
 }
```

Thời gian chạy: $O(n)$

R – 4.10: Give a big-Oh characterization, in terms of n, of the running time of the example2 method shown in Code Fragment 4.12.

```
// Returns the sum of the integers with even index in given array.
 public static int example2(int[ ] arr) {
    int n = arr.length, total = 0;
    for (int j=0; j < n; j += 2) // note the increment of 2
        total += arr[j];
    return total;
 }
```

Thời gian chạy: $O\left(\frac{n}{2}\right) = O(n)$

R – 4.11: Give a big-Oh characterization, in terms of n, of the running time of the example3 method shown in Code Fragment 4.12.

```
// Returns the sum of the prefix sums of given array.
public static int example3(int[ ] arr) {
   int n = arr.length, total = 0;
   for (int j=0; j < n; j++) // loop from 0 to n-1
       for (int k=0; k <= j; k++) // loop from 0 to j
           total += arr[j];
   return total;
}
```

Thời gian chạy: $O\left(\frac{n(n-1)}{2}\right) = O(n^2)$

R – 4.12: Give a big-Oh characterization, in terms of n, of the running time of the example4 method shown in Code Fragment 4.12.

```
// Returns the sum of the prefix sums of given array.
public static int example4(int[ ] arr) {
   int n = arr.length, prefix = 0, total = 0;
   for (int j=0; j < n; j++) { // loop from 0 to n-1
        prefix += arr[j];
        total += prefix;
   }
   return total;
}
```

Thời gian chạy: $O(2n) = O(n)$

R – 4.13: Give a big-Oh characterization, in terms of n, of the running time of the example5 method shown in Code Fragment 4.12.

```
// Returns the number of times second array stores sum of prefix sums from
 ↪ first.
public static int example5(int[ ] first, int[ ] second) {
// assume equal-length arrays
   int n = first.length, count = 0;
   for (int i=0; i < n; i++) { // loop from 0 to n-1
       int total = 0;
       for (int j=0; j < n; j++) // loop from 0 to n-1
           for (int k=0; k <= j; k++) // loop from 0 to j
               total += first[k];
       if (second[i] == total) count++;
   }
   return count;
}
```

$$\text{Thời gian chạy: } O\left(2n * \frac{n(n-1)}{2}\right) = O(n^3)$$

Phần 2: Creativity

C – 4.2: Describe an efficient algorithm for finding the ten largest elements in an array of size n. What is the running time of your algorithm?

- Tạo một mảng ten_larges_element chứa 10 phần tử của mảng có kích thước n ban đầu.
- Bắt đầu từ vị trí thứ 10 trong mảng ban đầu, lặp từng phần tử và so sánh với phần tử nhỏ nhất trong mảng ten_larges_element
  + Nếu phần tử trong mảng ban đầu > phần tử nhỏ nhất trong ten_larges_element thì thay thế phần tử nhỏ nhất trong ten_larges_element = phần tử trong mảng ban đầu
  + Ngược lại lấy phần tử tiếp theo mảng ban đầu so sánh tiếp.
- Lặp đến khi kết thúc ta được mảng gồm 10 phần tử lớn nhất từ mảng ban đầu

- Thời gian chạy của thuật toán này là: O(n)

C – 4.4: show that $\sum_{i=1}^{n} i^2$ is $O(n^3)$

- Áp dụng công thức tổng n số tự nhiên đầu tiên:

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\Rightarrow \text{thời gian chạy: } O\left(\frac{n(n+1)(2n+1)}{6}\right) = O(n^3)$$

C – 4.6: Determine the total number of grains of rice requested by the inventor of chess.

- Mô tả bài toán: Trên một bàn cờ 64 ô, ta thực hiện việc đặt gạo với ô đầu tiên 1 hạt và các ô tiếp theo số lượng hạt gạo gấp đôi ô trước. Ví dụ: 1, 2, 4, 8, 16, 32, … 2^{64}.

  ⇒Nhận thấy đây là một cấp số nhân với công bội q = 2 và số hạng đầu tiên $u_1 = 1$, ta thực hiện tính tổng của một cấp số nhân:

$$S_n = \frac{u_1(q^n - 1)}{q - 1}, (q \neq 1)$$

$$\Rightarrow S_{64} = \frac{1(2^{64} - 1)}{2 - 1} = 2^{64} - 1$$

$$= 18\,446\,744\,073\,709\,551\,615 \text{ hạt gạo}$$

C – 4.11: An array A contains n − 1 unique integers in the range [0, n − 1], that is, there is one number from this range that is not in A. Design an O(n)-time algorithm for finding that number. You are only allowed to use O(1) additional space besides the array A itself.

- Khởi tạo mảng A có kích thước n - 1.
- Sử dụng phép xor và ứng dụng tính chất giao hoán của nó:
  + Vòng lặp đầu: Xor từng phần tử trong mảng A
  + Vòng lặp tiếp theo: Xor từng số trong phạm vi [0, n - 1] với kết quả xor phía trước. Các giá trị trùng nhau sẽ triệt tiêu lẫn nhau chỉ còn giá trị không xuất hiện cần tìm.
- Ví dụ minh họa: ta có mảng A{1, 4, 2, 0} (chứa 4 giá trị không trùng nhau trong phạm vi [0, 4])
  + Xor từng phần tử trong mảng A: 1 xor 4 xor 2 xor 0 = xorResult.

+ Xor từng số trong phạm vi [0, 4] với xorResult:
xorResulr xor 0 xor 1 xor 2 xor 3 xor 4 = 1 xor 4 xor 2 xor 0 xor 0 xor 1 xor 2 xor 3 xor 4 = 0 xor 0 xor 1 xor 1 xor 2 xor 2 xor 4 xor 4 xor 3 = 3.

- Code thực hiện:

```
public class test {
    static int findMissingNumber(int[] A, int n) {
        int xorResult = 0;

        // XOR all elements in the array A
        for (int i = 0; i < A.length; i++) {
            xorResult ^= A[i];
        }

        // XOR all numbers in the range [0, n-1]
        for (int i = 0; i <= n; i++) {
            xorResult ^= i;
        }

        return xorResult;
    }

    Run | Debug
    public static void main(String[] args) {
        int[] A = {0, 4, 1, 2}; // Example array
        int n = A.length; // n is one more than the length of A
        int missingNumber = findMissingNumber(A, n);
        System.out.println("The missing number is: " + missingNumber);
    }
}
```

C – 4.22: An array A contains n integers taken from the interval [0, 4n], with repetitions allowed. Describe an efficient algorithm for determining an integer value k that occurs the most often in A. What is the running time of your algorithm?

- Tạo mảng count_accurs có kích thước 4n + 1 có các giá trị ban đầu là 0 nhằm lưu số lần xuất hiện của số nào đó trong mảng A (chỉ số vị trí của mảng coutn_accurs là các số có khả năng xuất hiện trong A)
- Duyệt qua các giá trị trong mảng A và đếm các giá trị trương ứng. Mỗi khi đếm một giá trị trong mảng A ta sẽ lưu vào mảng count_accurs với vị trí là giá trị mảng A và giá trị là số lần xuất hiện giá trị đó trong mảng A.
- Sau khi đếm lưu vào vị trí tương ứng trong count_accurs.
- Sau khi duyệt hết các 4n + 1 vị trí ta được mảng count_accurs với các giá trị là số lần xuất hiện tương ứng với các chỉ số. lấy ra giá trị lớn nhất và chỉ số mảng tại giá trị lớn nhất đó là số k cần tìm.

- Thời gian thực hiện:
+ Khi duyệt n giá trị trong mảng A ta có O(n)

+ Khi duyệt 4n + 1 giá trị trong mảng count_accurs ta có O(4n + 1)
⇒ thời gian chạy: O(n)

- Code thực hiện:

```java
import java.util.Random;
public class test {

    static void inputArray(int[] A, Random rand, int n) {
        for (int i = 0; i < n; i++) {
            A[i] = rand.nextInt(4 * n);
        }
    }

    static int count(int arr[]) {

        int count_accurs[] = new int[arr.length * 4];

        for (int i = 0; i < arr.length; i++) {
            count_accurs[arr[i]]++;
        }

        int maxCount = -1;
        int frenquent = -1;
        for (int i = 0; i < count_accurs.length; i++) {
            if (count_accurs[i] > maxCount) {
                maxCount = count_accurs[i];
                frenquent = i;
            }
        }

        return frenquent;
    }

    Run | Debug
    public static void main(String[] args) {
        Random rand = new Random();
        int n = 10;
        int A[] = new int[n];
        inputArray(A, rand, n);

        for (int i : A) {
            System.out.println(i);
        }

        System.out.printf(format:"value most frenquent: k = %s", count(A));
    }
}
```