Phân Tích Thuật Toán

Bùi Thị Thanh Phương

Ngày 17 tháng 4 năm 2023

1 Bài toán 1. Viết một hàm mà trong đó:

- Input: f(n), hai số nguyên dương a, b chứa khoảng giá trị của n. Ví dụ $a = 10, b = 1000$. Khi đó n chạy từ $10 \rightarrow 1000$.
- Output: Chỉ ra độ phức tạp của $f(n) = O(n^{\alpha})$, nếu không có dạng này hiện thông báo.

Kiểm tra lại hàm đã có với các trường hợp sau:
$(a) f(n) = n^2$
$(b) f(n) = n^3 + \cos(n) n^4$
$(c) f(n) = n^n$
$(d) f(n) = n^3 + n^2 + n + 1$
Bài Làm
Ý tưởng: Input: $f(n)$, hai số nguyên dương a, b chứa khoảng giá trị của n" nên ta có giá trị của $f(n)$. Do "Output: Chỉ ra độ phức tạp của $f(n) = O(n^{\alpha})$ " nên $f(n) \sim n$ 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. Để $f(n)$ có độ phức tạp $O(n^{\alpha})$ thì $f(n)$ phải có dạng 1 đa thức bậc α:
$f(n) = a_0 + a_1 n + a_2 n^2 + ... + a_{\alpha} n^{\alpha}$
Khi đó ta đi xây dựng thuật toán tìm xấp xỉ giá trị α cho mỗi hàm số $f(n)$ với $n$chạy từ $10 \rightarrow 1000$.

Entrée [2]:
```
def complexity(f, a, b, log = False):
    if log:
        log_fn = f
    else:
        log_fn = lambda n: np.log(np.abs(f(n)))
    max = round(log_fn(a)/np.log(a))
    for n in range(a+1, b+1):
        alpha = round(log_fn(n)/np.log(n))
        if alpha > max:
            max = alpha

    return max
```

Ngoài các tham số f là hàm số cần kiểm tra, a, b là giới hạn củan, thì ta có thêm một tham số là log. Tham số này có chức năng thông báo cho hàm biết rằng hàm f truyền vào có dạng $\log(f(n))$ nếu $\log = True$ và ngược lại. Có một số hàm f, ví dụ như $f(n) = n^2$, với nlớn thì hàm tính toán của thư viện numpy không đáp ứng được. Do đó ta cần lấy $\log(f)$ để dễ dàng cho việc tính toán hơn.
Ta thu được các giá trị α ứng với từng hàm số$f(n)$ như sau:

```
     Exercise 1
********************
>> For n in range [10, 1000] then
@ alpha for f(n) = n^2:                 2
@ alpha for f(n) = n^3 + cos(n).n^4:    4
@ alpha for f(n) = n^n:                 1000
@ alpha for f(n) = n^3 + n^2 + n + 1:   3
```

Từ kết quả tìm được ở trên, ta thấy rằng các hàm số đều có độ phức tạp là $O(n^\alpha)$ với giá trị α tương ứng với bậc của hàm f.
Bài 2 Viết chương trình nhân 2 số nguyên lớn A, Bcó N chữ số.

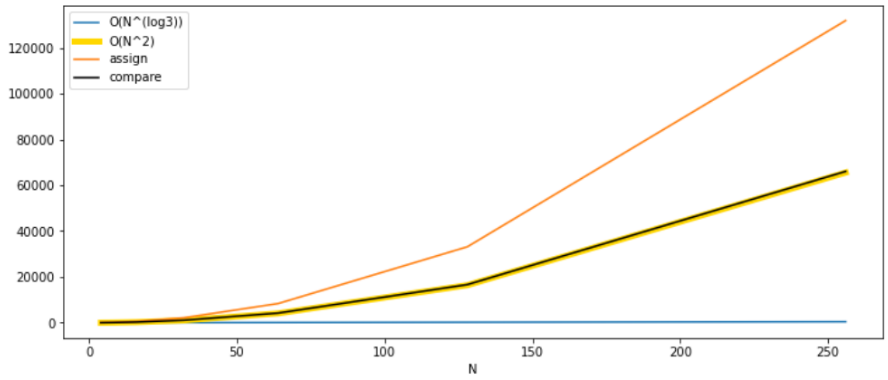
- Input:A, BvN.
- Output: $C = A.B$

Bằng 2 phương pháp
(a) Phương pháp truyền thống có độ phức tạp $O(N^2)$
(b) Phương pháp cải tiến $O(N^{\log 3}$
Kiểm tra lại chương trình với điều kiện sau:

- $LyN = 2^k, k = 10, 11, ..., 32.A, B$ là 2 số nguyên có N chữ số lấy ngẫu nhiên. Ứng với mỗi giá trị củaN, tính thời gian trung bình để$A.B$.
- So sánh 2 phương pháp để xác định hai phương pháp có độ phức tạp là $O(N^2)$ và $O(N^{\log 3})$.

Bài Làm
(a) Phương pháp truyền thống có độ phức tạp $O(N^2)$
Ý tưởng: Với phương pháp truyền thống, ta sẽ xem các số A, B có dạng *string*, Lúc này, ta sẽ từng giá trị củaB theo thứ tự từ phải sang trái nhân với A, kết quả sẽ được dịch trái ứng với vị trí $N - i$ tương ứng, với i là vị trí chữ số đang xem xét của B, ta dịch trái bằng cách thêm vào chuỗi kết quả tìm được 1 ký tự 0. Kết quả của phép nhân sẽ bằng tổng tất cả các giá trị được tính ở trên sau khi chuyển từ chuỗi sang số.

```python
def grade_school_multiply(A, B, N):
    assign = compare = 0
    A_str = str(A)
    B_str = str(B)
    result = 0
    assign += 3
    for i in range(N-1, -1, -1):
        value = 0
        for j in range(N-1, -1, -1):
            mul1 = str(int(B_str[i])*int(A_str[j])) + '0'*(N-1-j)
            value += int(mul1)
            assign += 2
            compare += 1

        mul2 = str(value) + '0'*(N-1-i)
        result += int(mul2)
        compare += 2
        assign += 3
    compare += 1
    return result, assign, compare
```

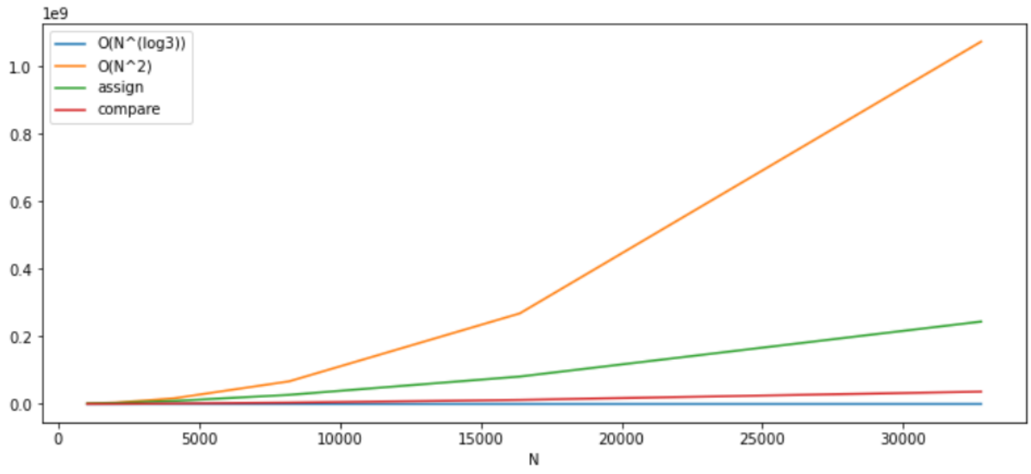


Ta sẽ đi kiểm định về độ phức tạp của phương pháp trên bằng cách vẽ ra đường thời gian tương ứng với mỗi giá trị $N = 2^k, k = 2, ..., 8$.
Từ đồ thị trên, ta có thể thấy rằng các đường biểu thị số phép so sánh và số phép gán của thuật toán trên có nét tương đồng với đường $O(N^2)$, thậm chí đường biểu thị số phép so sánh trùng với đường$O(N2)$.
(b) Phương pháp cải tiến (Karatsuba)

Ý tưởng: Viết A, B dưới dạng
$A = A_1 * 10^{\frac{n}{2}} + A_2$ và $B = B_1 * 10^{\frac{n}{2}} + B_2$
Đặt $C = A_1 * B_1$
$D = A_2 * B_2$
$E = (A_1 + A_2) * (B_1 + B_2) - C - D$ Khi đó
$A * B = C * 10^n + E * 10^n + D$
Ta sẽ lặp lại các bước trên đến khi các số tham gia phép nhân có số phần tử là 1

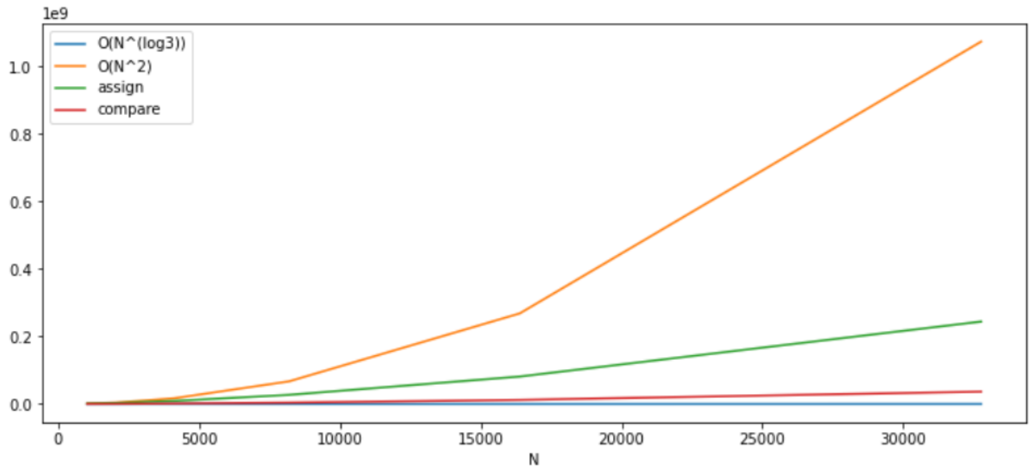
```
: def karatsuba_multiply(A, B, N):
    assign = compare = 0
    def Karatsuba(X, Y, N):
        assign = compare = 0
        m = (N//2)
        A1 = X // (10 ** m)
        A2 = X % (10 ** m)
        B1 = Y // (10 ** m)
        B2 = Y % (10 ** m)
        assign += 5

        compare += 1
        if N == 1:
            return X*Y, assign, compare
        else:
            C, assign1, compare1 = Karatsuba(A1,B1, len(str(A1)))
            D, assign2, compare2 = Karatsuba(A2,B2, len(str(A2)))
            E3, assign3, compare3 = Karatsuba(A1+A2, B1+B2, len(str(A1+A2)))
            E = E3 - C - D
            result = (10**(m*2))*C + (10**m)*E + D
            assign += (5 + assign1 + assign2 + assign3)
            compare += (compare1 + compare2 + compare3)
            return result, assign, compare
    result, assign_k, compare_k = Karatsuba(A, B, N)
    assign += assign_k
    compare += compare_k
    return result, assign, compare
```

Ta sẽ đi kiểm định về độ phức tạp của phương pháp cải tiến này với $O(N^{\log 3})$
Từ độ thị trên ta thấy rằng, các đường biểu thị số lượng phép gán và phép so sánh của



thuật toán trên có nét tương đồng với đường $O(N^{\log 3})$ hơn so với đường $O(N^2)$. Điều đó cho thấy rằng, thuật toán trên có độ phức tạp xấp xỉ $O(N^{\log 3})$.
Một ví dụ kiểm định tính chính xác của hai thuật toán trên:

```
     Exercise 2
********************
Example
N =  8
A =  23927689
B =  65526575
>> Implement grade school multiplication:
A*B =  1567899507835175
>> Implement karatsuba multiplication:
A*B =  1567899507835175
```


1e9
O(N^(log3))
O(N^2)
assign
compare
1.0
0.8
0.6
0.4
0.2
0.0
0
5000
10000
15000
20000
25000
30000
N