Nhập môn phân tích độ phức tạp thuật toán-21TN

TRẦN MINH HOÀNG-21120075

BÀI TẬP LÝ THUYẾT LẦN 2

Câu 1.1 Cho mảng a[7] = 7,4,1,3,2,6,5 hãy thống kê lại số phần tử cực đại bên trái trong từng bước lặp của thuật toán sắp xếp insertion sort (có 6 bước). Hỏi từ đó có thể có nhận xét gì?

Giải:

Bước	Dữ liệu	Số phần tử cực đại bên trái
1	4,7,1,3,2,6,5	2
2	1,4,7,3,2,6,5	3
3	1,3,4,7,2,6,5	4
4	1,2,3,4,7,6,5	5
5	1,2,3,4,6,7,5	6
6	1,2,3,4,5,6,7	7

Bảng 1: Thống kê

Nhận xét dữ liệu:

- Mảng được sắp xếp tăng khi số phần tử cực đại bên trái **đạt đến** số phần tử của mảng. Trong mỗi bước thực hiện của thuật toán insertion sort, số lượng số phần tử cực đại bên trái tăng thêm **một đơn vị**.
- Vì vậy, **số bước cần thực hiện của insertion sort** bằng hiệu giữa **số lượng phần tử trong mảng** và **số lượng các phần tử cực đại bên trái ở bước đầu tiên**.

Câu 1.2 Cho đoạn code sau đây dùng để kiểm tra xem có hai ký tự liên tiếp giống nhau trong một chuỗi ký tự s hay không. Gọi X là biến ngẫu nhiên mô tả số lần lặp cần dùng ứng với phép thử lấy ngẫu nhiên input là một chuỗi s sinh ra bởi hoán vị các ký tự trong từ "pappa". Hãy tính số lần lặp trung bình của đoạn code này dựa trên việc tính kỳ vọng của X:

```
1       //C++
2       bool check(string s){
3           for (int i = 1; i < s.length; i++)
4               if (s[i-1] == s[i]) return true;
5           return false;
6       }
7
```

Giải: Từ "pappa" có 5 ký tự, trong đó có 3 ký tự **p** và 2 ký tự **a**. Vì vậy, khi lấy s là hoán vị của các ký tự trong "pappa" thì:

- Tổng số trường hợp có thể xảy ra: $N = C_5^3.C_2^2 = 10.1 = 10$ (chọn 3 vị trí cho **p** và 2 vị trí cho **a**).

- Tổng số trường hợp mà vòng lặp dừng lại ngay lần đầu tiên $\Leftrightarrow X = 1$ (tức là 2 ký tự đầu giống nhau) :

$$N1 = C_3^1.C_2^2 + C_3^3 = 3.1 + 1 = 4$$

  - $C_3^1.C_2^2$: 2 vị trí đầu là cho **p**; 3 vị trí sau, chọn 1 cho **p** và 2 cho **a**
  - C_3^3: 2 vị trí đầu là cho **a**; 3 vị trí sau, chọn 3 cho **p**
- Tổng số trường hợp mà vòng lặp dùng ở lần thứ 2 $\Leftrightarrow X = 2$:

$$N2 = C_2^1.C_1^1 + C_2^2 = 2.1 + 1 = 3$$

  - $C_2^1.C_1^1$: vị trí đầu là cho **a**, 2 vị trí sau cho **p**; 2 vị trí còn lại chọn 1 cho **p** và 1 cho **a**
  - C_2^2: vị trí đầu là cho **p**, 2 vị trí sau cho **a**; 2 vị trí còn lại cho **p**
- Tổng số trường hợp mà vòng lặp dùng ở lần thứ 3 $\Leftrightarrow X = 3$:

$$N3 = 1$$

  (chỉ có: pappa)
- Tổng số trường hợp mà vòng lặp dùng ở lần thứ 4 $\Leftrightarrow X = 4$:

$$N4 = 2$$

Vì chuỗi s có 5 ký tự nên vòng lặp chỉ lặp tối đa 4 lần, vì vậy với $X > 5$ thì số trường hợp xảy ra là 0.

Vậy, **Kỳ vọng của X là:**

$$E(X) = 1 \cdot \frac{N1}{N} + 2 \cdot \frac{N2}{N} + 3 \cdot \frac{N3}{N} + 4 \cdot \frac{N4}{N} = 1 \cdot \frac{4}{10} + 2 \cdot \frac{3}{10} + 3 \cdot \frac{1}{10} + 4 \cdot \frac{2}{10} = 2.1$$

Câu 2: Xét bài toán sau đây: cho số nguyên $n \geq 2$ và mảng $a_0, a_1, .., a_{n-1}$ gồm các số nguyên có giá trị tuyệt đối không quá 10^9. Tìm giá trị lớn nhất của hiệu $a_j - a_i$ với $0 \leq i < j < n$.

Câu 2.1 Hãy mô tả ngắn gọn cách giải vét cạn brute-force cho bài toán trên (dùng mã giả hoặc code C++/Python) và đánh giá độ phức tạp tương ứng của thuật toán đó.

Giải:Cách vét cạn brute-force cho bài toán trên:

- Duyệt tất cả cặp (i, j) với $0 \leq i < j \leq n$:
- Với mỗi cặp (i, j) ta tính hiệu $sub = a_j - a_i$ và dùng một biến max để lưu trữ giá trị lớn nhất của hiệu hai số.

```
def brute_force_solution(arr, n):
    max = arr[1] - arr[0]
    for i in range(0, n-1):
        for j in range(i, n):
            sub = arr[j] - arr[i]
            if sub > max:
                max = sub
    return max


```

Đánh giá độ phức tạp:

- Ở bên ngoài vòng for có 01 phép gán cố định.
- Ở bên trong 2 vòng for có 01 phép gán cố định và 01 phép gán phụ thuộc vào điều kiện của mệnh đề if.

Gọi $T(n)$ là số phép gán của thuật toán trên theo dữ liệu nhập có kích thước n.

Ta có bất phương trình:

$$1 + \sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 1 \le T(n) \le 1 + \sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 2 (*)$$

Mà ta có:

$$\sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 1 = \sum_{0 \le i \le n-2} (n-i) = \sum_{2 \le t \le n} t = 2 + 3 + \cdots + n = \frac{(n-1)(n-2)}{2} \in \Theta(n^2)$$

Suy ra:

$$\sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 2 = 2 \cdot \sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 1 = (n-1)(n-2) \in \Theta(n^2)$$

Từ đó:

$$1 + \sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 1 \in \Theta(n^2)$$

$$1 + \sum_{0 \le i \le n-2} \sum_{i \le j \le n-1} 2 \in \Theta(n^2)$$

Kết hợp với (*), suy ra:

$$T(n) \in \Theta(n^2)$$

Vậy, độ phức tạp của thuật toán brute-force là $\Theta(n^2)$.

Câu 2.2 Cho ý tưởng về thuật toán "chia để trị" để giải bài toán trên như sau: chia mảng ban đầu thành 2 mảng con từ vị trí trung vị mid để tính giá trị lớn nhất của $a_j - a_i$ trong các trường hợp sau đây: (I) $0 \le i < j < mid$, (II) $mid \le i < j < n$ và (III) $0 \le i < mid \le j < n$.

Đối với trường hợp (I) và (II), ta thực hiện gọi đệ quy giải hai bài toán riêng biệt cho hai mảng con tương ứng. Còn với trường hợp (III), ta chỉ cần lấy giá trị lớn nhất trong các số từ $mid \Rightarrow n$ (đặt là max_R) rồi trừ cho giá trị nhỏ nhất trong các số từ $0 \Rightarrow mid$ (đặt là min_L) là được; chú ý rằng để tìm giá trị lớn nhất (max_R) và nhỏ nhất (min_L), ta duyệt trực tiếp các mảng con qua một vòng lặp.

Yêu cầu: Mô tả công thức đệ quy cho $T(n)$ thích hợp để xác định chi phí tính toán trong thuật toán nêu trên. Từ đó, sử dụng định lý Master để đánh giá độ phức tạp của $T(n)$.

Giải: Để tìm giá trị lớn nhất của R trong trường hợp (III), chúng ta cần duyệt qua tất cả các số từ mid đến n. Tương tự, để tìm giá trị nhỏ nhất của L, ta cần duyệt qua tất cả các số từ 0 đến mid. Do đó, ta cần duyệt qua tất cả các phần tử trong mảng một lần (ngoại trừ phần tử mid sẽ được duyệt hai lần). Do đó, tổng chi phí để thực hiện việc tìm hiệu lớn nhất trong trường hợp (III) này sẽ là $\Theta(n)$. Gọi $f(n)$ là chi phí tính toán trong trường hợp (III). Ta có: $f(n) \in \Theta(n)$.

Để đơn giản, ta giả sử:

- Số phần tử cần mà thuật toán thực hiện trong trường hợp (I) và (II) là bằng nhau và bằng $\frac{n}{2}$
- Lấy $f(n) = n$

Vì thuật toán thực hiện cả 3 trường hợp nên **Chi phí tính toán của thuật toán là**:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + f(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

Theo định lý Master: ($a = 2$, $b = 2$, $f(n) = n$)

Xét $c = \lim\limits_{n\to\infty} \frac{a \cdot f(\frac{n}{b})}{f(n)} = \lim\limits_{n\to\infty} \frac{2 \cdot (\frac{n}{2})}{n} = \lim\limits_{n\to\infty} (1) = 1.$

Vì $c = 1$, nên $T(n) \in \Theta(f(n) \cdot \log_b n)$ hay $T(n) \in \Theta(n \log n)$.

Vậy độ phức tạp của thuật toán là $\Theta(n \log n)$.

Câu 2.3 Biết rằng có thể sử dụng cấu trúc dữ liệu segment tree (cây phân đoạn) để giải bài toán range minimum/maximum queries (truy vấn min/max trên nhiều mảng con), trong đó tốn chi phí $O(n)$ cho việc dựng cây một lần duy nhất và chi phí $O(logn)$ để tìm giá trị nhỏ nhất & lớn nhất cho các phần tử trên mỗi mảng con. Hỏi nếu áp dụng cấu trúc dữ liệu này thì có thể cải tiến độ phức tạp của thuật toán thế nào (mô tả lại công thức cho $T(n)$ và đánh giá)?

Giải: Trong giải pháp này, chúng ta xem xét chi phí của việc xây dựng cây và việc tính toán trong công thức đệ quy $T(n)$. Chi phí xây dựng cây $O(n)$ chỉ được tính một lần và không được xem xét trong công thức đệ quy. Thay vào đó, chúng ta tập trung vào việc tính toán chi phí của việc tìm kiếm giá trị lớn nhất và nhỏ nhất trên một phân đoạn trong segment tree, mỗi phép tính này đều mất $O(\log n)$ thời gian. Vậy, tổng chi phí tính toán trong trường hợp (III) là $O(\log n)$ (hay $f(n) \in O(\log n)$).

Để đơn giản ta giả sử:

- Số phần tử cần mà thuật toán thực hiện trong trường hợp (I) và (II) là bằng nhau và bằng $\frac{n}{2}$
- Lấy $f(n) = \log_2 n$

Vì thuật toán thực hiện cả 3 trường hợp nên chi phí tính toán của thuật toán là:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + f(n) = 2.T(\frac{n}{2}) + \log_2 n$$

Theo định lý Master: $(a = 2, b = 2, f(n) = \log_2 n)$

Xét $c = \lim\limits_{n\to\infty} \frac{a.f(\frac{n}{b})}{f(n)} = \lim\limits_{n\to\infty} \frac{2.\log_2 \frac{n}{2}}{\log_2 n} = \lim\limits_{n\to\infty} \frac{2.log_2 n - 2.log_2 2}{log_2 n} = lim_{n\to\infty}(2 - \frac{2}{log_2 n}) = 2$

Vì $c = 2 > 1$ nên $T(n) \in \Theta(n^{log_b a})$ hay $T(n) \in \Theta(n^{log_2 2})$ hay $T(n) \in \Theta(n)$

Vậy chi phí tính toán của thuật toán sẽ là $\Theta(n)$

Câu 3: Xét đoạn code sau đây(viết bằng C++):

```
1   int compute(int k){
2       int tmp = 0;
3       while(k>0){
4           if (k%2 == 1) tmp = tmp+k;
5           else tmp = tmp-1;
6           k = k/2;
7       }
8       return tmp;
9   }
10
11      int solve(int n){
12      int i = 1, res = 0;
13      while (i <= n*n){
14          res = res + i;
```

```
            res = res - compute(i);
            i = i + 1;
        }
        return res;
    }
```

Câu 3.1 Hãy viết lại đoạn code ứng với hai hàm solve() và compute() ở trên, trong đó có chèn thêm biến đếm count_compare vào các vị trí thích hợp để có thể đếm được số phép so sánh bằng thực nghiệm.

Giải:

```
    int compute(int k, int& count_compare){
    int tmp = 0;
    while(++count_compare && k>0){
        if (++count_compare && k%2 == 1) tmp = tmp+k;
        else tmp = tmp-1;
        k = k/2;
    }
    return tmp;
    }

    int solve(int n){
    int count_compare = 0;
    int i = 1, res = 0;
    while (++count_compare && i <= n*n){
        res = res + i;
        res = res - compute(i, count_compare);
        i = i + 1;
    }
    cout << "The number of calculations is  " << count_compare << endl;
    return res;
    }
```

```

    def compute(k, count_compare):
    tmp = 0;
    count_compare[0] = count_compare[0] + 1
    while(k>0):
        count_compare[0] = count_compare[0] + 1
        if (k%2 == 1):
        tmp = tmp+k
        else:
        tmp = tmp-1
        count_compare[0] = count_compare[0] + 1
        k = k // 2
    return tmp

    def solve(n):
    count_compare = [0]
    i = 1
    res = 0

    count_compare[0] = count_compare[0] + 1
    while (i <= n*n):
        count_compare[0] = count_compare[0] + 1
        res = res + i
        res = res - compute(i, count_compare)
        i = i + 1

    print("The number of comparisons is : ", count_compare[0]);
    return res
```

Câu 3.2 Hãy đánh giá theo n độ phức tạp thời gian của hàm solve() bằng lý thuyết, dựa trên việc đếm và ước lượng số phép gán cần thực hiện (ở đây để đơn giản, cho phép dùng xấp xỉ $\lfloor \log k \rfloor \approx \log k$ trong các tính toán).

Giải: Ta có :

$$Q(i) = 1 + 2\lfloor \log i \rfloor = 1 + 2\log i$$

$$S = 2 + \sum_{n=1}^{n^2} 3 + Q(i) = 2 + 3n^2 + \sum_{n=1}^{n^2} 1 + 2\log i = 2 + 4n^2 + 2\sum_{n=1}^{n^2} \log i$$

Ta tháy:

- $\sum_1^{n^2} \log i \leq \sum_1^{n^2} \log(n^2) < n^2 \log(n^2)(1)$
- $\sum_{i=1}^{n^2} \log(i) \geq \sum_{i=\frac{n^2}{2}}^{n^2} \log(i) \geq \sum_{i=\frac{n^2}{2}}^{n^2} \log(\frac{n^2}{2}) \geq \frac{n^2}{2} \log(\frac{n^2}{2})(2)$

Từ (1) và (2) suy ra:

$$\sum_{i=1}^{n^2} \log(i) \in \Theta(n^2 log(n^2))$$

$$\text{Vậy } T(n) = 2 + 4n^2 + 2\sum_{i=1}^{n^2} \log(i) \in O(n^2 \log(n^2)).$$