ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

CS112.N21.KHTN

BÀI TẬP TUẦN 2

PHÂN TÍCH ĐỘ PHỨC TẠP CỦA THUẬT TOÁN CÓ SỬ DỤNG ĐỆ QUY

Nhóm : 14
GV hướng dẫn : Nguyễn Thanh Sơn

Bài 1: Tower of Hanoi

1. In the original version of the Tower of Hanoi puzzle, as it was published in the 1890s by Edouard Lucas, French mathematician, the world will end after 64 disks have been moved from a mystical Tower of Brahma. Estimate the number of years it will take if monks could move one disk per minute. (Assume that monks do not eat, sleep, or die.)
2. How many moves are made by the ith largest disk ($1 \leq i \leq n$) in this algorithm?
3. Find a nonrecursive algorithm for the Tower of Hanoi puzzle and implement it in the language of your choice

Trả lời

1.1

Nếu các nhà sư chỉ di chuyển được một đĩa mỗi phút, để hoàn thành trò chơi Tháp Hà Nội ban đầu với 64 đĩa, thì thời gian cần là:

$$(2^{64} - 1) \text{ đĩa } x1x \frac{1}{60} x \frac{1}{24} x \frac{1}{365} = 3.51x10^{13} \text{ năm}$$

Do đó, nếu các nhà sư chỉ di chuyển được một đĩa mỗi phút và không cần nghỉ ngơi, ăn uống hay qua đời, để hoàn thành trò chơi Tháp Hà Nội ban đầu với 64 đĩa, thì sẽ tốn khoảng 351 tỷ năm.

1.2

Giả sử rằng chúng ta đang xét với n đĩa trong trò chơi Tháp Hà Nội. Theo giải thuật cơ bản, đĩa lớn nhất được di chuyển đến cột mới bằng cách thực hiện hai bước sau đây:

1. Di chuyển n-1 đĩa còn lại từ cột ban đầu đến cột trung gian.
2. Di chuyển đĩa lớn nhất từ cột ban đầu đến cột mới.

Để di chuyển n-1 đĩa từ cột ban đầu đến cột trung gian, chúng ta sẽ sử dụng giải thuật đệ quy, vì vậy số bước di chuyển sẽ được tính bằng cách áp dụng các công thức trên cho n-1 đĩa thay vì n đĩa.

Do đó, đĩa i (với $1 \leq i \leq n$) sẽ thực hiện được:

- Nếu i là đĩa lớn nhất ($i = n$): 2^{n-1} lần di chuyển.

1.3

Chúng ta có thể sử dụng cấu trúc dữ liệu Stack để giải quyết bài toán Tháp Hà Nội không dùng đệ quy.

```
void move(stack<int>& A, stack<int>& B, char from, char to) {
   int disk = A.top();
   A.pop();
   B.push(disk);
   cout << "Move disk " << disk << " from " << from << " to " << to << endl;
}

void TowerOfHanoi(int n) {
   stack<int> A, B, C;
   for (int i = n; i >= 1; i--) {
      A.push(i);
   }

   bool left_to_right = true;
   bool top_down = true;
   int total_moves = (1 << n) - 1;

   for (int i = 0; i < total_moves; i++) {
      if (i == 0 || i % 2 == 0) {
         if (left_to_right) {
            move(A, C, 'A', 'C');
         } else {
            move(C, A, 'C', 'A');
         }
      }
      else {
         if (top_down) {
            move(A, B, 'A', 'B');
            move(B, C, 'B', 'C');
         } else {
            move(C, B, 'C', 'B');
            move(B, A, 'B', 'A');
         }
      }

      if (i + 1 == (1 << (i + 1))) {
         if (left_to_right) {
            left_to_right = false;
         } else {
            left_to_right = true;
         }
      } else if (i + 1 == (1 << top_down)) {
         if (top_down) {
            top_down = false;
         } else {
            top_down = true;
         }
      }
   }
}
```

Bài 2: QuickSort

Recurrence Relation:

Giả sử, ta gọi T(n) là độ phức tạp thời gian của thuật toán quick sort với kích thước đầu vào là n. Để tính toán T, ta sẽ tính thời gian cho từng bước của của thuật toán.

Divide Step (Bước chia): Là bước để chia mảng của chúng ta xung quanh phần tử pivot.

Độ phức tạp của bước này sẽ là O(n).

Conquer Step (Bước "trị"): Là bước ta sẽ gọi để quy để sắp xếp nửa bên trái cũng như nửa bên phải.

Chúng ta giải quyết 2 bài toán con với kích thước phụ thuộc vào giá trị của phần tử pivot. Giả sử, những phần tử thứ i là những phần tử năm bên trái pivot và n – i – 1 sẽ là những phần tử năm bên phải phần tử pivot. Ta có:

Kích thước mảng trái: i

Kích thước mảng phải: n – i – 1

Vậy độ phức tạp của bước này sẽ là: T = T(i) + T(n – i – 1)

Combine Step (Bước hợp): Cho vui! Vì sau bước Conquer thì mảng của ta đã được sắp xếp. Độ phức tạp O(1).

Suy ra:

T(n) = O(n) + T(i) + T(n – i – 1) + O(1)

= T(i) + T(n – i – 1) + O(n) = T(i) + T(n – i – 1) + cn

Vậy **Recurrence Relation** của **quicksort**:

T(n) = c, if n = 1

T(n) = T(i) + T(n – i – 1) + cn, if n>1

Best Case:

Trường hợp tốt nhất xảy ra khi pivot được chọn là phần tử trung vị của mảng. Khi đó, số lượng phần tử trong mỗi mảng con sau khi phân chia là xấp xỉ bằng nhau hoặc chênh lệch không quá 1.

Do đó, ta có:

T(n) = T(n/2) + T(n - 1 - n/2) + cn = T(n/2) + T(n/2 - 1) + cn ~ 2T(n/2)+ cn

Sử dụng định lý thợ với:

a = 2, b = 2

f(n) = cn

Nên ta có trường hợp 2. Vì vậy, độ phức tạp thời gian của giải thuật QuickSort trong trường hợp tốt nhất là O(nlog n).

Worst Case:

Trong trường hợp xấu nhất, giải thuật QuickSort sẽ phân chia mảng thành hai phần mất cân bằng, do đó, kích thước của một mảng con là n - 1 và mảng con còn lại có kích thước là 0. Khi đó, ta có phương trình đệ quy sau:

T(n) = T(n - 1) + T(0) + cn = T(n - 1) + cn

Ta có thể giải phương trình này bằng cách triệt tiêu đệ quy. Khi triệt tiêu đệ quy, ta thu được:

T(n) = n - 1 + (n - 2) + ... + 2 + 1

Khi đó, ta có:

T(n) = O(n^2)

Vì vậy, độ phức tạp thời gian của giải thuật QuickSort trong trường hợp xấu nhất là O(n^2).

Average Case:

Giả sử, thuật toán chia cho ra giá trị mất cân bằng theo tỉ lệ 1, 9

Recurrence Relation: T(n) = T(9n/10) + T(n/10) + cn

Sử dụng phương pháp Recursion Tree cho ta kết quả:

T(n) = O(nlogn)

Bài 3:

a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{n-1} + 2^{n-1}$

b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.

c. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.

d. Is it a good algorithm for solving this problem?

Trả lời

3.a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{n-1} + 2^{n-1}$

```
def power_of_two(n):
    if n == 0:
        return 1
    else:
        return 2 * power_of_two(n - 1)
```

3.b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.

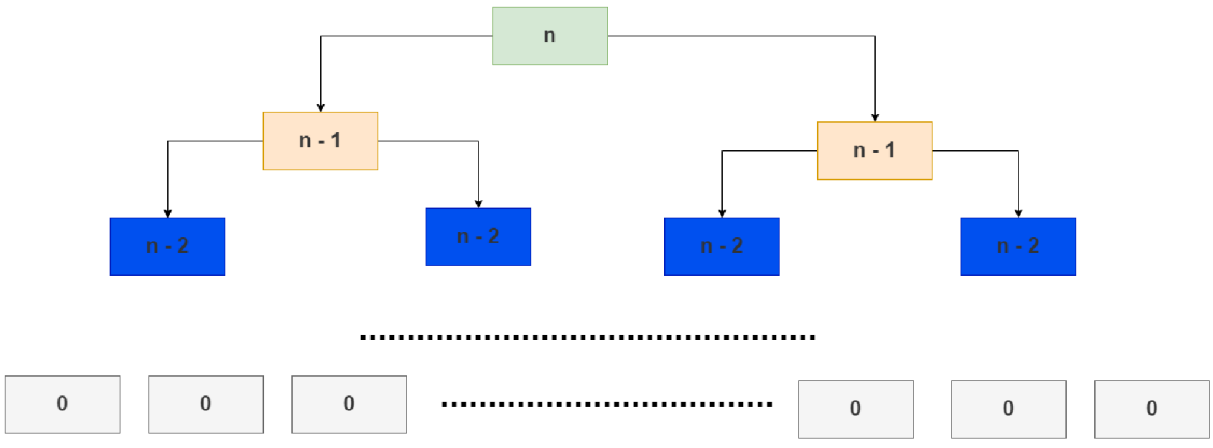
Giả sử, T(n) là độ phức tạp thời gian của thuật toán. Ta có recurrence relation cho các bước của thuật toán như sau:

T(n) = 0, if n = 0

T(n) = T(n – 1) + 1, otherwise

Dễ dàng thấy số bước thực hiện để tính toán 2^n sẽ bằng số bước thực hiện tính toán 2^(n-1) cộng thêm 1 bước combine.

3.c Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.



3.d Is it a good algorithm for solving this problem?

Với n lớn, thì thuật toán trên khi chạy sẽ tốn rất nhiều thời gian. Vì vậy, đây không phải là thuật toán tốt khi xử lý n lớn.