Đại học Quốc gia Thành phố Hồ Chí Minh

Trường Đại học Công nghệ Thông tin

Bài tập nhóm tuần 3

Đánh giá độ phức tạp của thuật toán không đệ quy

CS112 - Phân tích và Thiết kế Thuật toán

Nhóm: 05
Lớp: CS112.N21.KHTN
MSSV: 21520029, 21521845

Ngày 4 tháng 4 năm 2023

Problem 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 i-th 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.

Answer

1. The problem Tower of Hanoi with n disks requires at least $2^n - 1$ moves to complete.

   So, with 64 disk, we should need $2^{64} - 1 = 18446744073709551615$ steps or minutes, approximate 35096545041304 years to finish.

2. Let u_i means the number of steps the i-th largest disk should move.

   It is clear that $u_1 = 1$.

   For $k \geq 2$:

   To move disk $k-1$, the k-th disk should be move first, and after the movement of disk $k-1$, disk k will move again to the top. So every single move of disk $k-1$ should lead into 2 moves of disk k.

   So we get $u_i = 2u_{i-1}$.

   After some mathematical transformations, we could conclude that $u_i = 2^{i-1}$.

3. C++ implementation source code:

```
#include <bits/stdc++.h>
using namespace std;

char rod[]={'A', 'B', 'C'};
vector<stack<int>> stacks(3); // 3 stacks for 3 rods

class towerOfHanoi {
   private:
   int n;
   public:
   towerOfHanoi(int);
   void moveDisk(int, int);
   void runner();
};

towerOfHanoi::towerOfHanoi(int n = 0){
   this->n = n;
```

```
        }

        void towerOfHanoi::moveDisk(int a, int b)
        {
           if (stacks[b].empty() || (!stacks[a].empty() && stacks[a].top() <
           stacks[b].top()))
           {
              cout << "Move␣disk␣" << stacks[a].top() << "␣from␣rod␣" << rod[a] <<
                 "␣to␣rod
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" << rod[b] << "\n";
              stacks[b].push(stacks[a].top());
              stacks[a].pop();
           }
           else
           moveDisk(b, a);
        }

        void towerOfHanoi::runner(){
           cout << "Tower␣of␣Hanoi␣for␣" << n << "␣disks:\n";

           int src = 0, aux = 1, dest = 2;
           for (int i = n; i > 0; i--)
           stacks[src].push(i);

           int totalMoves = (1 << n) - 1;
           if (n % 2 == 0)
           swap(aux, dest);

           for (int i = 1; i <= totalMoves; i++)
           {
              if (i % 3 == 0)
              this->moveDisk(aux, dest);
              else if (i % 3 == 1)
              this->moveDisk(src, dest);
              else
              this->moveDisk(src, aux);
           }
        }
```

Problem 2: Quick Sort

Quicksort is one of the fastest sort-algorithm. Below is the example quicksort code.

```
def QuickSort(arr):
if len(arr) <= 1:
return arr
else:
```

```
pivot = arr[0]
left = [x for x in arr[1:] if x <= pivot]
right = [x for x in arr[1:] if x > pivot]
return QuickSort(left) + [pivot] + QuickSort(right)
```

Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed for Quicksort algorithm. And solve it for the best case, worst case and average case, then conclude the time complexity for each case.

Answer

Let $T(n)$ be the time complexity to execute the algorithm for the array size n.
For $n \geq 2$, we have:
Let k be the size of the left part. The size of the right one should be $n-1-k$. Morover, it takes $\mathcal{O}(n)$ to merge the array after recursive calls. Hence, we get:

$$T(n) = T(k) + T(n-1-k) + \mathcal{O}(n) \text{ (1)}$$

1. In the best case, $k = n-1-k$. So (1) could be rewritten as:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

   In this case, it is clear that the total complexity is $\mathcal{O}(n \log n)$

2. The worst case is $k = n-1$. In this situation, we get:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \mathcal{O}(n) \\ \Rightarrow T(n) &= \sum_{i=1}^{n-1} [T(i) + \mathcal{O}(i)] \\ \Rightarrow T(n) &= \frac{n(n-1)}{2} \end{aligned}$$

   So the total complexity should be $\mathcal{O}(n^2)$.

3. In general cases, like a binary search tree, if all elements are distributed randomly, the pivot of Quick Sort algorithm seems be the average of the array. As a result, $T(n) \approx 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$.
   So, in general cases, the complexity is $\Theta(n \log n)$.

Problem 3: EXP

1. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{n-1} + 2^{n-1}$
2. Set up a recurrence relation for the number of additions made by the algorithm and solve it.
3. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.
4. Is it a good algorithm for solving this problem?

Answer

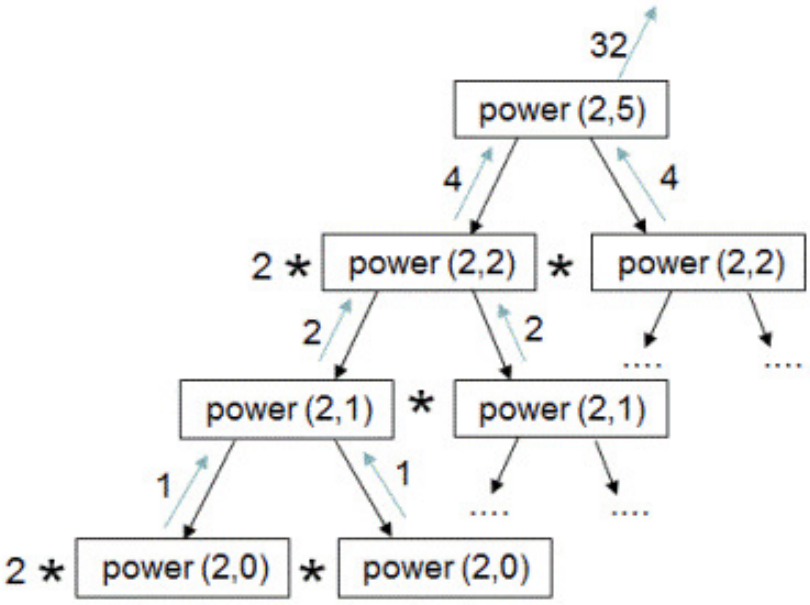
1. The algorithm for this algorithm could be implement in Python as following:

```
def Power(n):
if n = 0:
return 1
else:
return Power(n-1) + Power(n-1)
```

2. Let $T(n)$ be the time complexity of the algorithm while calculating 2^n. We have:

$$\begin{aligned} T(n) &= T(n-1) + T(n-1) \\ &\Rightarrow T(n) = 2T(n-1) \\ &\Rightarrow T(n) = 4T(n-2) \\ &\dots \\ &\Rightarrow T(n) = 2^n \end{aligned}$$

3. With $n = 5$, the recursive tree is:



partial recursion tree for calculating 2^5

Let d be the depth of the recursive calls.

With $d = 1$, there is one node.

With $d = 2$, there are two nodes.

With $d = 3$, there are four nodes.

...

With $d = k$, there are 2^{k-1} nodes.

With $d = n$, there are 2^{n-1} nodes.

Summaries, we have total of $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$ nodes, also the number of recursive calls.

4. This algorithm is not good because of duplication in calls.
   After a tiny optimzation, we could have the new algorithm:

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

This algorithm requires only one recursive call for every n. On the other hand, the time complexity now is $T(n) = T(n-1) + \mathcal{O}(1)$, or $\mathcal{O}(n)$ for total calculation.

However, there is a more optimized algorithm which only takes $\mathcal{O}(\log n)$. It is called **Exponentiation by squaring**.