ANALYSIS OF RECURSIVE ALGORITHMS

GROUP 7

Nguyễn Quốc Trường 21521604

Trần Xuân Minh 21520352

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 ($1 \le i \le n$) 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 number of moves required to complete the Tower of Hanoi puzzle with n disks can be calculated using the formula 2^n - 1. Therefore, for 64 disks, the total number of moves required would be: 2^64 - 1 = 18446744073709551615

Approximate 35,072,522,765,438 years to finish.

Therefore, it would take approximately 35 trillion years for the monks to complete the puzzle.

2.The ith largest disk moves 2^(i-1) times in this algorithm. This can be derived from the observation that the ith largest disk must be moved every time the smaller disks are moved, and there are 2^(i-1) such moves before the ith disk can be moved.

3.Here is a non-recursive algorithm for the Tower of Hanoi puzzle:

```
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print("Move disk 1 from source", source, "to target", target)
        return
    tower_of_hanoi(n-1, source, target, auxiliary)
```

```
    print("Move disk", n, "from source", source, "to target", target)
    tower_of_hanoi(n-1, auxiliary, source, target)
```

This algorithm uses recursion to solve the puzzle by breaking it down into smaller sub-problems. The base case is when there is only one disk, in which case it can be moved directly from the source pole to the target pole. For larger values of n, we first move the top n-1 disks from the source pole to the auxiliary pole, using the target pole as a temporary holding place. We then move the largest disk from the source pole to the target pole, and finally move the n-1 disks from the auxiliary pole to the target pole, using the source pole as a temporary holding place. This process is repeated recursively until all the disks have been moved to the target pole.

2. QuickSort

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

```python
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:

To set up a recurrence relation for the Quicksort algorithm, we need to determine the number of times the basic operation (i.e., the comparison operation) is executed for an input of size n. Let T(n) be the number of times the basic operation is executed for an input of size n.

In the Quicksort algorithm, the input array is partitioned into two subarrays around a pivot element, and the basic operation is executed once for each pair of elements that are compared during the partitioning process. Let k be the index of the pivot element in the input array, and let m and n-m-1 be the sizes of the left and right subarrays, respectively. Then, the number of times the basic operation is executed for an input of size n is given by:

$$T(n) = T(m) + T(n-m-1) + n-1$$

The first two terms on the right-hand side represent the number of times the basic operation is executed during the partitioning process for the left and right subarrays, respectively. The last term represents the number of times the basic operation is executed during the merging process, where the pivot element is inserted between the left and right subarrays.

The initial condition for the recurrence relation is $T(0) = 0$, since there are no comparisons to be made for an empty array.

Now, let's solve the recurrence relation for the best, worst, and average cases of the Quicksort algorithm:

1.Best case: In the best case, the pivot element is the median of the input array, and the left and right subarrays have equal size. Therefore, we have $m = \text{floor}((n-1)/2)$ and $n-m-1 = \text{ceil}((n-1)/2)$. Substituting these values into the recurrence relation, we get:

$$T(n) = 2T(\text{floor}((n-1)/2)) + n-1$$

Using the master theorem, we can show that the solution to this recurrence relation is $T(n) = O(n\log n)$, which gives us the time complexity of the Quicksort algorithm in the best case.

1.Worst case: In the worst case, the pivot element is either the smallest or largest element in the input array, and one of the subarrays is empty. Therefore, we have $m = 0$ or $m = n-1$. Substituting these values into the recurrence relation, we get:

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

Solving this recurrence relation using mathematical induction, we get:

T(n) = 1 + 2 + 3 + ... + (n-1) = n(n-1)/2

Therefore, the time complexity of the Quicksort algorithm in the worst case is O(n^2).

3.Average case: In the average case, the pivot element is chosen randomly, and the left and right subarrays have sizes that are roughly proportional to each other. Let T(n) be the expected number of times the basic operation is executed for an input of size n. Then, we have:

T(n) = (1/n) * sum(T(i) + T(n-i-1) + n-1) for i=0 to n-1

Using the substitution method, we can show that the solution to this recurrence relation is T(n) = O(nlogn), which gives us the time complexity of the Quicksort algorithm in the average case.

Therefore, the time complexity of the Quicksort algorithm is O(nlogn) in the best and average cases, and O(n^2) in the worst case.

3. EXP

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?

Answer:

a.Here is a recursive algorithm for computing 2^n based on the formula 2^n = 2^(n-1) + 2^(n-1):

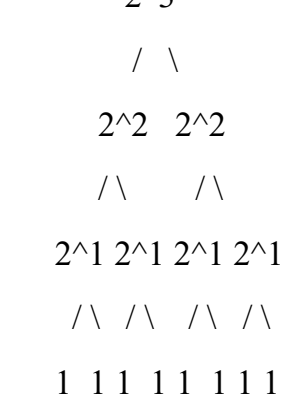
```
def compute_power_of_two(n):
    if n == 0:
        return 1
    else:
        return 2 * compute_power_of_two(n-1)
```

b.To set up a recurrence relation for the number of additions made by the algorithm, let A(n) be the number of additions made by the algorithm when computing 2^n. Each recursive call to compute_power_of_two(n-1) involves two additions, so we have:

A(n) = 2A(n-1) + 2 (for n > 0)

A(0) = 0

c.Here is a tree of recursive calls for the algorithm when computing 2^3:



The number of calls made by the algorithm is equal to the number of nodes in the tree, which is 2^(n+1) - 1 for an input of size n.

d.This algorithm is not a good algorithm for computing 2^n, because it requires an exponential number of additions and recursive calls to compute the result. The time complexity of the algorithm is O(2^n), which is very slow for large values of n. A more efficient algorithm for computing 2^n is the bitwise shift operator, which can compute 2^n in O(logn) time.