Analysis of Algorithms

Amypo Technologies Pvt Ltd

Agenda

- Space Complexity
- Mathematical analysis for Recursive and Non – Recursive Algorithm

Space Complexity

- The space complexity can be defined as amount of memory required by an algorithm to run.
- To compute the space complexity we use two factors: Constant and instance characteristics. The space requirement S(p) can be given as :

  S(p) = C + Sp

- where C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers.
- And Sp is a space dependent upon instance characteristics.

Analysis of Recursive Algorithms:

Understanding Recursion:

- **Definition**: Recursion is a programming concept where a function calls itself directly or indirectly to solve a smaller instance of the same problem. Recursive algorithms express problems in terms of smaller instances, making them elegant and concise.

- **Essence of Recursion**: The fundamental idea is breaking down a complex problem into simpler, more manageable sub-problems until reaching a base case, which is a problem small enough to be solved directly.

Definition of Recursive Algorithms:

Recursive Algorithm Characteristics:

- A recursive algorithm contains a base case and a set of rules that reduce the original problem towards the base case.
- The base case ensures the recursion stops, preventing an infinite loop.
- Recursive calls operate on smaller instances of the problem.
- Example: The factorial function n! can be defined recursively as n! = n * (n-1)! with a base case of 0! = 1.

Advantages and Challenges:

Advantages :

- **Elegance and Simplicity**: Recursive algorithms often provide a clear and concise representation of a problem, making the code more readable and maintainable.
- **Divide and Conquer**: Recursion is particularly powerful for problems that naturally exhibit a divide-and-conquer structure, where breaking a problem into smaller sub-problems simplifies the overall solution.

Challenges :

- **Stack Overflow**: Recursion can lead to a stack overflow if not carefully managed, especially for problems with deep recursion.
- **Performance Overhead**: Recursive calls may introduce additional function call overhead and memory consumption compared to iterative solutions.
- **Understanding and Debugging**: Recursive code can be challenging to understand and debug, especially for complex algorithms with many recursive calls.

Mathematical Analysis of Recursive Algorithms

EXAMPLE Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n. Since

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \qquad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM $F(n)$

//Computes $n!$ recursively //Input: A nonnegative integer n //Output: The value of $n!$

if $n = 0$ **return** 1

else return $F(n-1) * n$

For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,[5] whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula

- $F(n) = F(n-1) \cdot n$ for $n > 0$,
- the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1)\text{ by } n}} \quad \text{for } n > 0.$$

- Indeed, $M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by n.
- The last equation defines the sequence $M(n)$ that we need to find. This equa-tion defines $M(n)$ not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely $n - 1$. Such equations are called ***recurrence relations*** or, for brevity, ***recurrences***. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics.

- **if *n*** = 0 **return** 1.

This tells us two things. First, since the calls stop when ***n*** = 0***,*** the smallest value of ***n*** for which this algorithm is executed and hence ***M(n)*** defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when ***n*** = 0***,*** the algorithm performs no multiplications. Therefore, the initial condition we are after is

$$M(0) = 0.$$

the calls stop when $n = 0$ no multiplications when $n = 0$

Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0, \qquad \textbf{(2.2)}$$
$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function ***F (n)*** itself; it is defined by the recurrence.

- The second is the number of multiplications ***M(n)*** needed to compute ***F (n)*** by the recursive algorithm whose pseudocode was given at the beginning of the section.
- As we just showed, ***M(n)*** is defined by recurrence (2.2). And it is recurrence (2.2) that we need to solve now.
- After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: ***M(n)*** = ***M(n − i)*** + ***i.***
-

$$\begin{aligned} M(n) &= M(n-1)+1 && \text{substitute } M(n-1)=M(n-2)+1 \\ &= [M(n-2)+1]+1 = M(n-2)+2 && \text{substitute } M(n-2)=M(n-3)+1 \\ &= [M(n-3)+1]+2 = M(n-3)+3. \end{aligned}$$

- What remains to be done is to take advantage of the initial condition given. Since it is specified for ***n*** = 0***,*** we have to substitute ***i*** = ***n*** in the pattern's formula to get the ultimate result of our backward substitutions:

- **General Plan for Analyzing the Time Efficiency of Recursive Algorithms**
- Decide on a parameter (or parameters) indicating an input’s size.
- Identify the algorithm’s basic operation.
- Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
- Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
- Solve the recurrence or, at least, ascertain the order of growth of its solution.

Mathematical Analysis of Non recursive Algorithms

- **EXAMPLE 1** Consider the problem of finding the value of the largest element in a list of ***n*** numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.
- **ALGORITHM** *MaxElement(**A**[0..**n** − 1])*
- //Determines the value of the largest element in a given array
- //Input: An array ***A***[0..***n*** − 1] of real numbers
- //Output: The value of the largest element in ***A maxval*** ← ***A***[0]
- **for *i* ← 1 to *n* − 1 do**
- **if *A*[*i*] > *maxval maxval* ← *A*[*i*]**
- **return *maxval***

- Let us denote ***C(n)*** the number of times this comparison is executed and try to find a formula expressing it as a function of size ***n***. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable ***i*** within the bounds 1 and ***n*** − 1, inclusive. Therefore, we get the following sum for ***C(n)***:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

- This is an easy sum to compute because it is nothing other than 1 repeated ***n*** − 1 times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

- Here is a general plan to follow in analyzing nonrecursive algorithms.

- **General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms**
  Decide on a parameter (or parameters) indicating an input's size.
- Identify the algorithm's basic operation. (As a rule, it is located in the inner-most loop.)
- Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
- Set up a sum expressing the number of times the algorithm's basic operation is executed.
- Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

- Before proceeding with further examples, you may want to review Appen-dix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

and two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.