Design and Analysis of Algorithms (DAA)

BSCS 4th semester

Sept – Dec 2023 Morning

Farhan Shafiq (Ph.D.)
farhanshafiq@fuuast.edu.pk

Time Complexity of Recursive Function

Recursion is one of the popular problem solving approaches in data structure and algorithms. Even some problem solving approaches are totally based on recursion like Divide and Conquer , backtracking etc.

- A recurrence relation is a mathematical equation where any term is defined using its previous terms.
- We use recurrence relations to analyze the time complexity of recursive algorithms in terms of input size.
- In recursion , we solve a problem by breaking it down into smaller sub problem.

- If the time complexity funtion of input size n is T(n) then the time complexity of smaller sub-problem will be defined by the same function but in terms of the sub-problem’s input size.
- So here is an approach to write T(n) if we have k number of sub-problems.

- T(n)= T(input size of 1st sub-problem)+ T(input size of 2nd sub-problem)+ T(input size of 3rd sub-problem)+ ………….+ T(input size of kth sub-problem)+ Time Complexity of additional operation other than recursive call.

Basic Concept of finding the time Complexity

Suppose, there are some books kept in one place and you have to move the book and keep it on a shelf or in a rack.

How much time does it take?

Maybe half a second, a quarter of a second, maybe if somebody works very slowly may take one second for keeping one book there.

The time varies from person to person. So, we don't mention seconds or milliseconds, we say one unit of time.

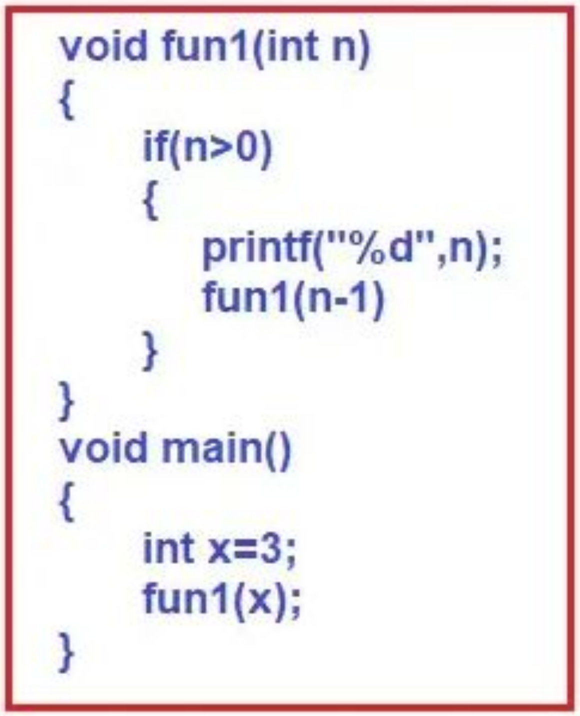
If you take the example of currency, one dollar, one rupee, one pound.

We say one but what is the market value that might be different. So, we say one buck or one unit of currency.

- In the same way, we assume that every statement takes one unit of time. If that statement is repeated multiple times, then we need to count the frequency that how many times it is executed. That is sufficient for analyzing our function.

Example to Find the Time Complexity of a Recursive Function:

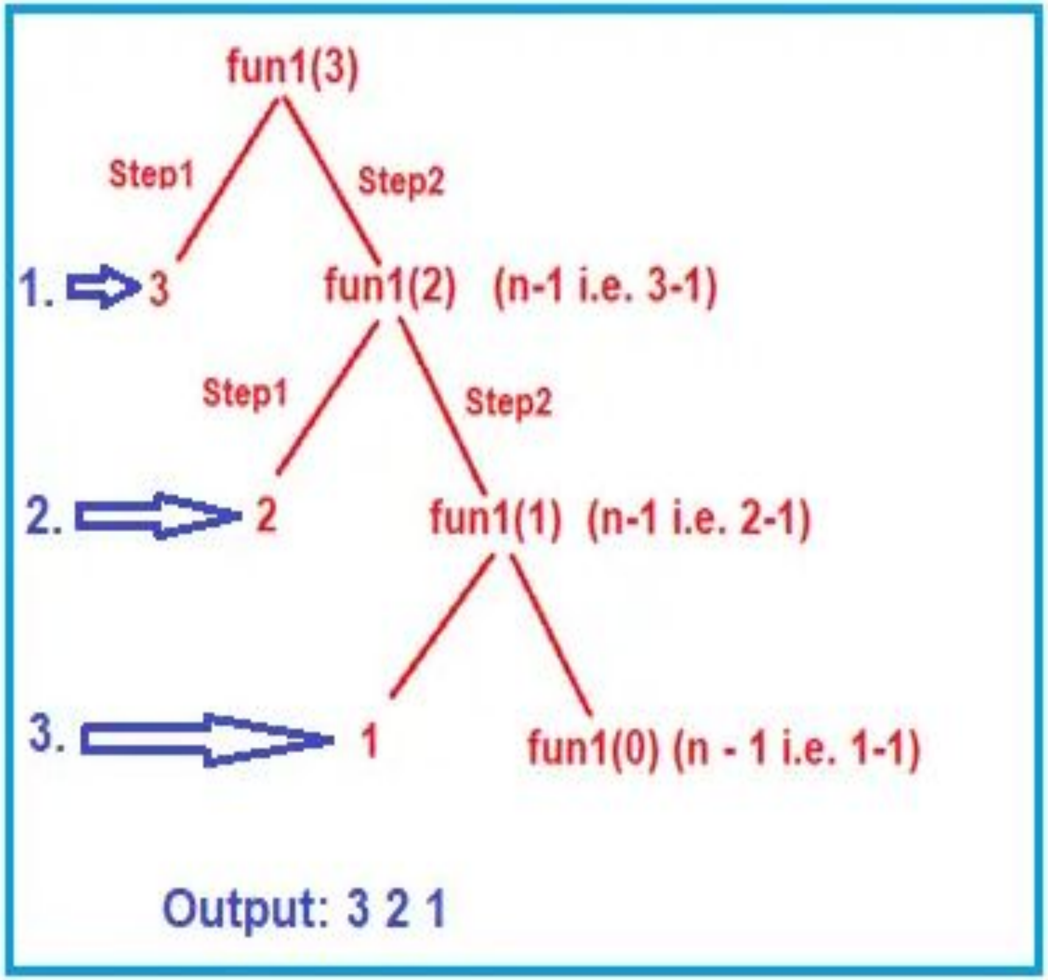
- Consider the Following program we will calculate the time complexity of the following recursive function.



```
void fun1(int n)
{
      if(n>0)
      {
          printf("%d",n);
          fun1(n-1)
      }
}
void main()
{
      int x=3;
      fun1(x);
}
```

- what the above function (fun1) is doing.
- It is doing nothing just printing. It is just printing the value of n.
- **How much time does it take for printing?**
- It takes one unit of time for printing.
- **How many times the printf is written there?**
- Only one-time printf is written there. But this is a recursive function. So, it is calling itself again and again.

- it is a recursive function, let us find out how many times the printf function is executed.



- first it prints the value 3, then print 2 and then print the value 1.
- That means the printf statement executed three times.

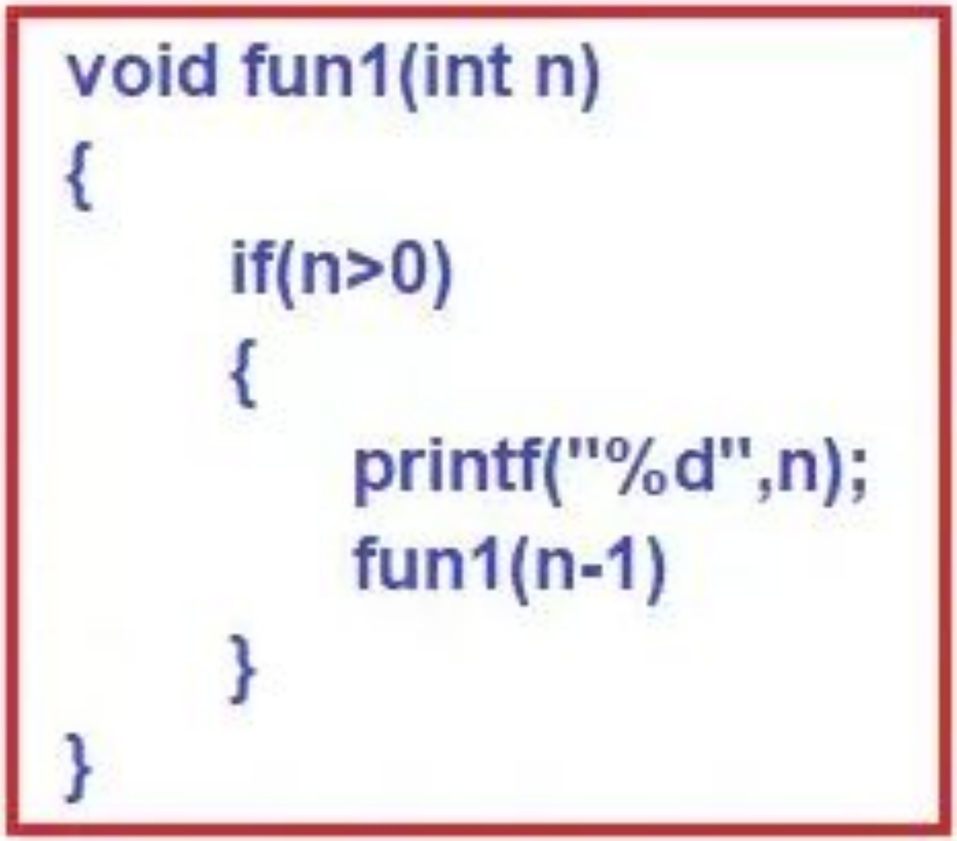
- So, this recursive function will take 3 units of time to execute when the n value is 3.
- If we make the n value is 5 then it will take 5 units of time to execute this recursive function.

- we can say for n it will take n units of time. Coming back to the example, if we have to keep one book on a shelf.
- You will take one unit of time, for 10 books you will take 10 units of time.
- So, for n number books, you will n unit of time.
- The most important point that you need to remember is time depends on **the number of books.**

The time can be represented as the order of **n** i.e. **O(n).** The time taken is in order of **n**.

Time Complexity using Recurrence Relation:

- There is one more method to find the time complexity i.e. using recurrence relation.
- Let us see how to write a recurrence relation and how to solve it to find the time complexity of the recursive function.
- Now, let us find the time complexity of the following recursive function using recurrence relation.



```
void fun1(int n)
{
        if(n>0)
        {
                printf("%d",n);
                fun1(n-1)
        }
}
```

assume that the time taken by the above function is **T(n)** where **T** is for **time**. If the time is taken for **fun1()** is **T(n)**, then the total time should be the sum of all the times taken by the statements inside that function.

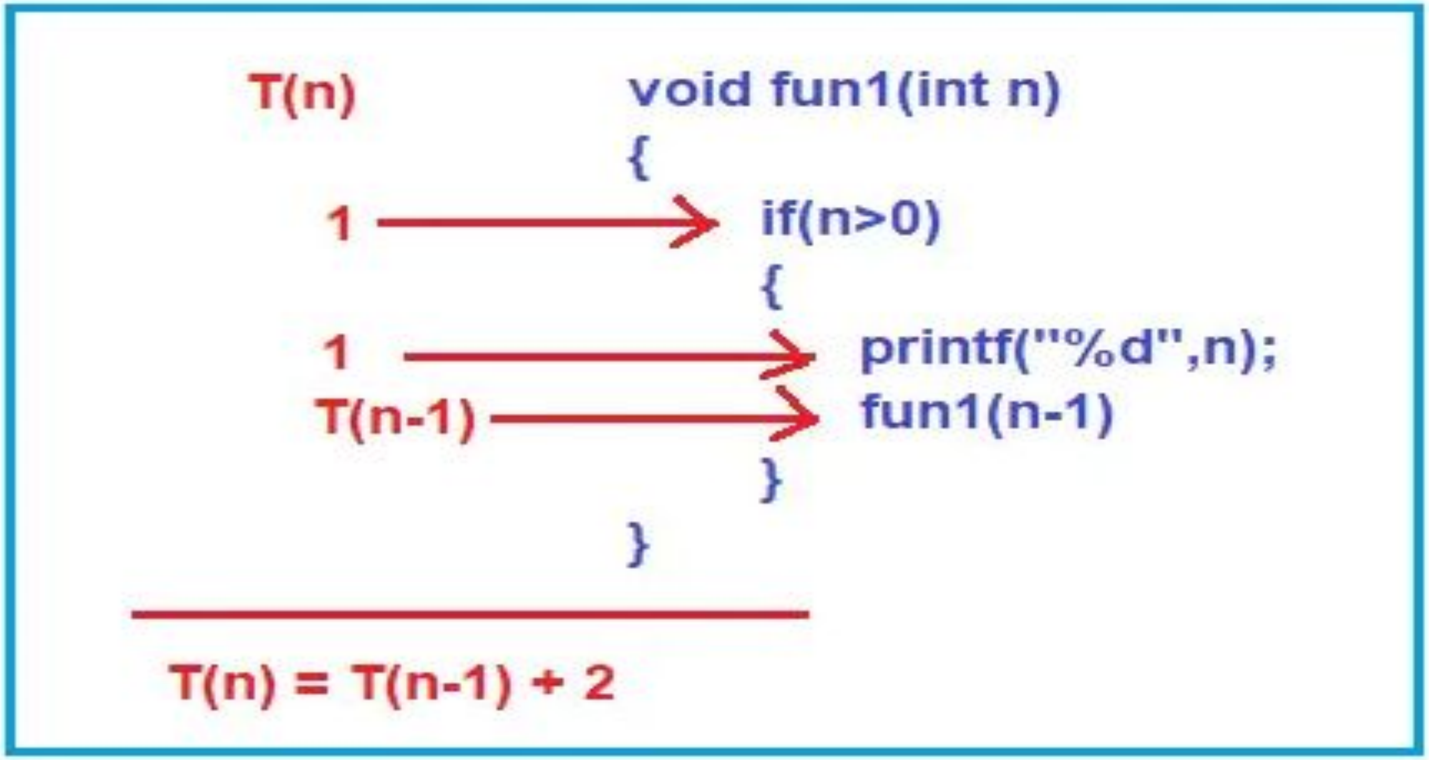
- let us look at the statement.
- Every statement takes one unit of time for execution.
- See there is a conditional statement (if (n > 0)) inside the function.
- How much time it takes for execution, just one unit of time it takes for execution.
- Then there is a printf statement, this also takes one unit of time.

- Then there is one more function call statement (fun1(n-1)) there, how much time it will take, it also takes one unit of time.

- No, that is not correct.
- It will not take one unit of time.
- This is a function call.
- It should be the total time taken by that function. It is not just a normal statement. It will call again itself. So, there is something more behind that one. So, we need to know how much time that function call is taking?

- What we said fun1(int n) function call, the total time is T(n).
- Then this fun1(n-1) is similar to fun1(int n) one, and here it is n-1.
- So, the total time taken by this function will be T(n-1) time.
- Then what is T(n)? As we said sum of all the times taken by the statement. So, let us take the sum that is **T(n) =T(n-1)+2**.

- For better understanding, please have a look at the below image.



```
void fun1(int n)
{
    if(n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}
```

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

- So, the recurrence relation is
- **T(n)=T(n-1)+ 2** when **n>0**.
- What happens when **n=0**, it will just check the condition and it will not enter inside it and will come out.
- Just checking the condition, so, it will take one unit of time.

- For better understanding, please have a look at the below image.

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + 2 & n>0 \end{cases}$$

- So, this is the recurrence formed from that fun1 function. So, the time complexity of the recursive function can be represented in the form of a recurrence relation.

Induction Method or Successive Substitution method: