

Lesson:



Problems based on recursion-1



Pre-Requisites

- Functions
- Basics of Recursion concept

List of Concepts Involved

- Given an integer, find out the sum of its digits using recursion.
- Given two numbers p & q, find the value p^q using a recursive function.

Problem 1 Given an integer, find out the sum of its digits using recursion.

Input n= 1234

Output 10

Explanation 1+2+3+4=10

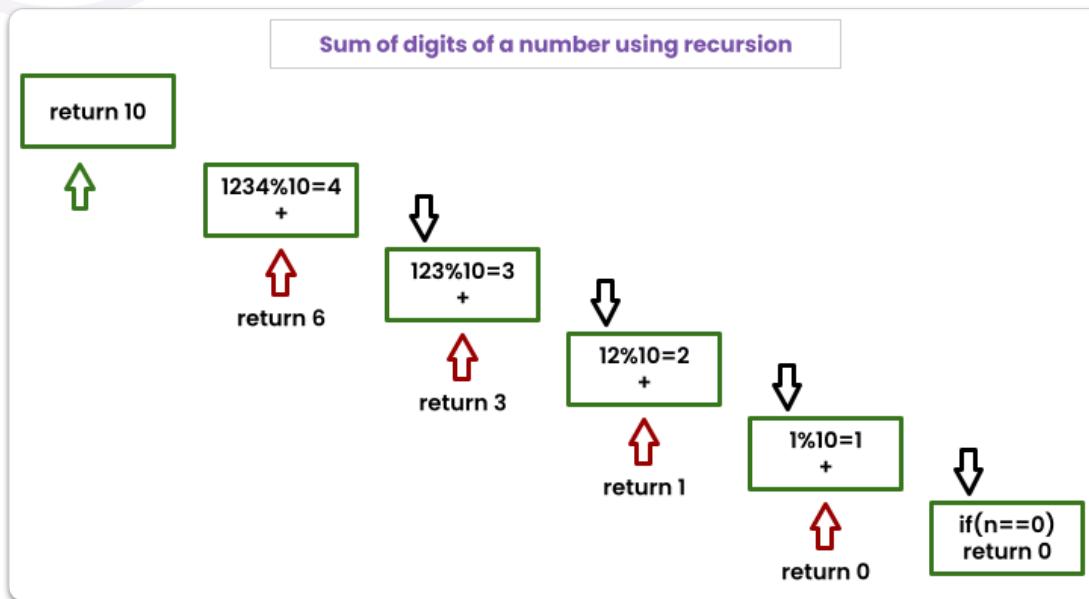
Approach

let's define a function func which will take a number as input and will return the sum of all of its digits. now this function can we written in this way first digit + sum of remaining digit (the same way human do sum in left to right manner) but as extracting first digit is relatively tough thus we will start with last digit so the function $func(num)$ can be written as $func(num) = last_digit + func(num \text{ after removing last digit})$. Now to get the last digit we can use % operator and to remove the last digit we can use / operator.

Let's see the step-by-step approach for a better understanding of how the algorithm works.

Let the number be 1234.

- Step 1-> $1234 \% 10$ which is equal-to 4 + (send $1234/10$ to next step)
- Step 2-> $123 \% 10$ which is equal-to 3 + (send $123/10$ to next step)
- Step 3-> $12 \% 10$ which is equal-to 2 + (send $12/10$ to next step)
- Step 4-> $1 \% 10$ which is equal-to 1 + (send $1/10$ to next step)
- Step 5-> 0 algorithm stops



Code link <https://pastebin.com/MApb46vE>

```

    ↴ ↵ ⏪
Enter the number
1234
Sum of digits in 1234 is 10
  
```

Problem 2 Given two numbers a & b, find the value a^b using a recursive function.

Input a = 5, b = 3

Output 125

Explanation $(5 \times 5 \times 5) = 125$

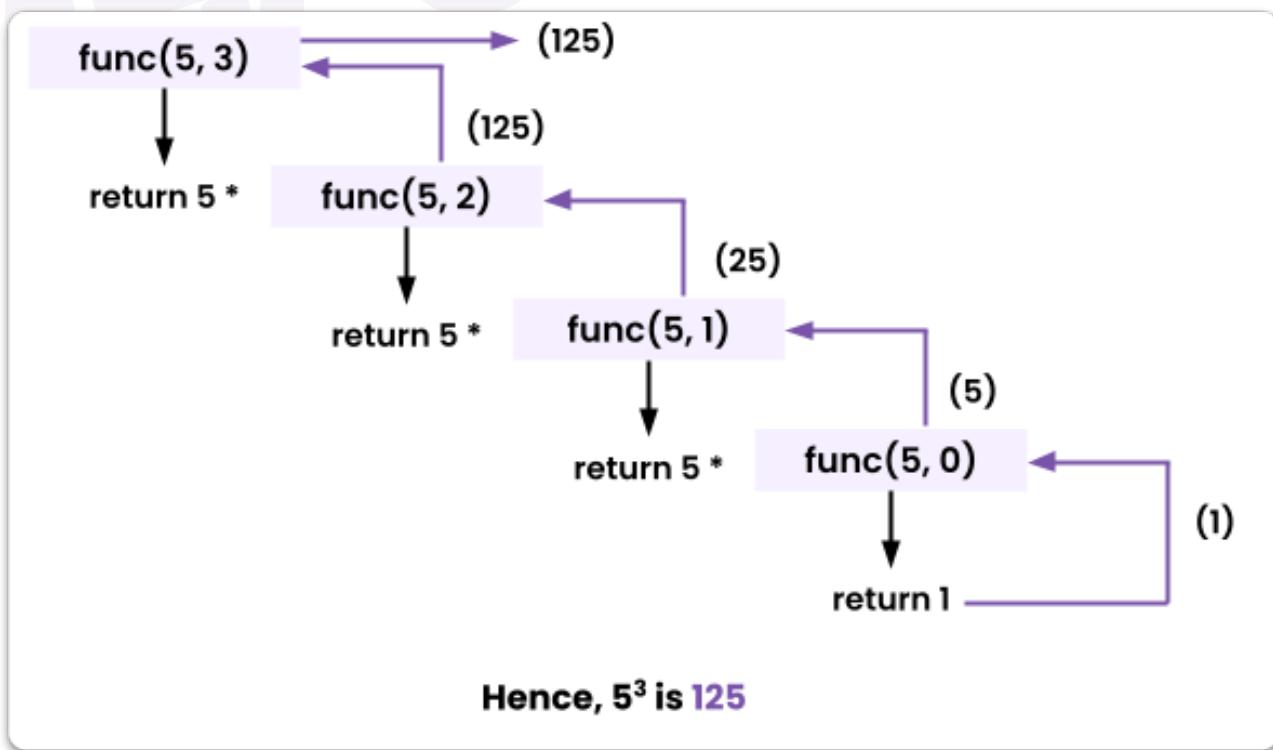
Approach

- Create a recursive function say func(int a, int b)
- Base condition : if($b == 0$) return 1.
- Otherwise, return $(a * \text{func}(a, b-1))$

Code link <https://pastebin.com/1dVB6xr8>

```

    ↴ ↵ ⏪
Required Power is 125
  
```



In the above program, the function `func()` is a recursive function. If the power is zero, then the function returns 1 because any number raised to power 0 is 1. If the power is not 0, then the function recursively calls itself.

- Time Complexity : $O(b)$, b denotes the power.
- Space Complexity : $O(b)$, b is the size of the recursion stack.

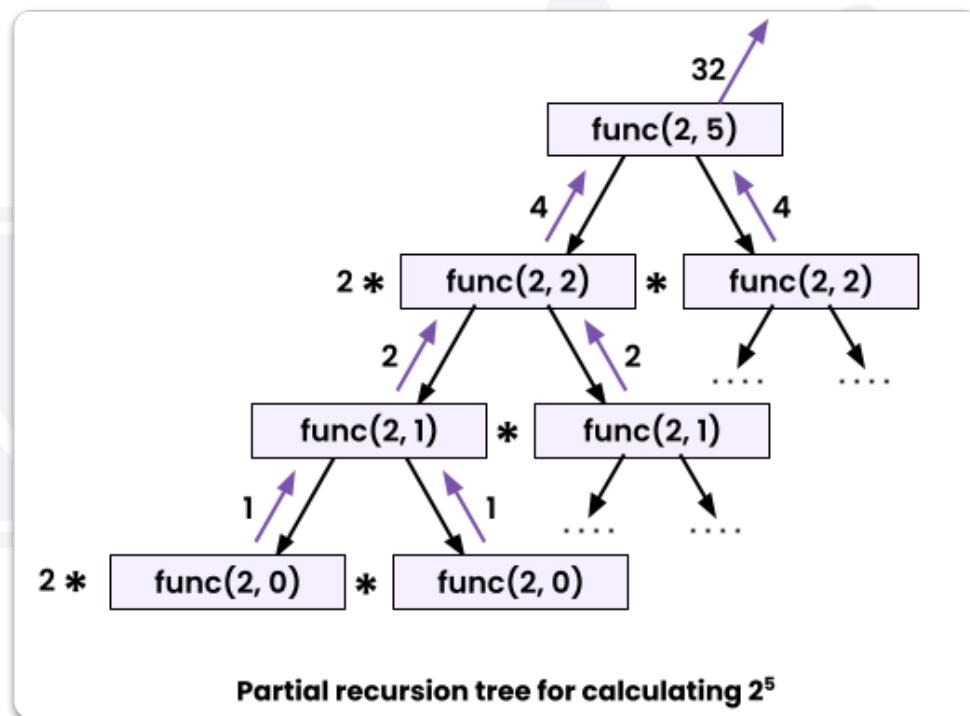
Efficient approach:

Lets see a better approach on how we can improve the time complexity of our solution, lets define a function `func` which will take two numbers a and b as input and will return a^b . Now this function can be written in this way $\text{func}(a, b) = \text{func}(a, b / 2) \times \text{func}(a, b / 2)$; Like humans do in real life for example $a=2, b=4$, so for calculating 2^4 = we can find (2^2) , take square of it in case b is even. In case b is odd, for example $a=2, b=5$, so for calculating 2^5 = we can find (2^2) , take square of it and multiply it with 2 like $(2^5) = (2) * (2^2) * (2^2)$.

`func(a, b) = func(a, b / 2) × func(a, b / 2); // if b is even`

`func(a, b) = a × func(a, (b-1) / 2) × func(a, (b-1) / 2); // if b is odd`

The generated recursion tree when above recurrence relation is implemented to calculate 2^5 is shown below.



As you can observe, `func(2,2)`, `func(2,1)` are being computed multiple times. To avoid these redundant computations, we store the result in a result variable while making the recursive call and use this variable value subsequently.

Code

<https://pastebin.com/BWplrapM>

- This function has a base case: $b = 0$.
- This function has two recursive calls. Only one of them is made in any given call.
- Let's first consider the case when b is even: In that case, the recursive call is made with $b / 2$. I will store the value return by $\text{func}(a, b/2)$ in result variable and will return square of result variable i.e ($\text{result} * \text{result}$).
- If all calls would use this case, then you half b in each call down until you reach 1. This is indeed $\log(b)$ calls (plus 1 for the base case).
- In that case when b is odd, the recursive call is made with $(b-1) / 2$. I will store the value return by $\text{func}(a, (b-1)/2)$ in result variable and will return square of result variable multiplied by a i.e ($a * \text{result} * \text{result}$).

Time Complexity

- To more easily observe that the algorithm is $O(\log b)$, you can rewrite the function to always reduce the power by half at each iteration, so that in the worst case, there are only $\log b$ call.
- The reason why the algorithm remains $O(\log b)$ even with the extra calls for the odd number case is because the number of extra calls is bounded by a constant. In the worst case, $b/2$ is odd at each iteration, but this would only double the number of extra calls (the constant is 2). That is, at worst, there will be $2\log b$ calls to complete the algorithm.
- The time complexity of this algorithm is $O(\log(b))$ while computing $\text{power}(a, b)$. This is because at every level in the recursion sub-tree, we are doing only one computation (and using that value subsequently) and there are $\log(b)$ levels overall and Space Complexity is $O(1)$.

Upcoming Class Teasers

- Recursion on arrays
- Recursion on strings