What is Tail Call Optimization? What is tail call elimination? Provide an example of tail call optimization.

If you haven't already done so, then you may want to read our article on Tail Recursion before you proceed with this tutorial on tail call optimization. In any case, we reproduce a large part of that article in our tutorial here since it's necessary to understand tail call optimization and tail call elimination.

Let's go through a normal recursive function that calculates the Factorial of a number in Java, and then contrast that with a tail recursive function. Then, we will explain in detail what the meaning of tail call optimization is.

A normal recursive Factorial call in Java

Here's what a normal recursive Factorial call looks like:

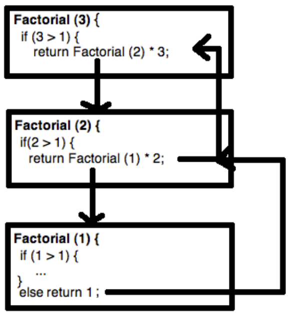
```
public int factorial (int x)
{
  if (x == 1)
  {
    return 1;  //base case
  }

  else  /*recursive case*/
   return factorial(x-1) * x;
}
```

Now, here is what the sequence of function calls would look like if we want to make a call to the factorial and we pass in the value of 3:

```
factorial (3) ;
return factorial(2) * 3;
return factorial(1) * 2;
return 1;
```

What you should pay special attention to is the fact that in order to reach the final value of the factorial (which is 3), ***each and every function call must be executed to completion***. You should be able to see that – in order to know the factorial of 3 we must find the factorial of 2 multiplied by 3, and in order to get the factorial of 2, we must get the factorial of 1 multiplied by 2.

Each recursive call results in a **new** stack frame as you can see in this image:



```
Factorial (3) {
 if (3 > 1) {
   return Factorial (2) * 3;

Factorial (2) {
 if(2 > 1) {
   return Factorial (1) * 2;

Factorial (1) {
 if (1 > 1) {
   ...
 }
 else return 1 ;
```

A tail recursive factorial call

Now, here is what the factorial of a function would look like in a tail recursive Python function:

```
def factorial(i, current_factorial=1):
  if i == 1:
    return current_factorial
  else:
    return factorial(i - 1, current_factorial * i)
```

If we pass in a value of 3 into our tail recursive function, the sequence of function calls would look like this:

```
factorial(3, 1)
return factorial (3,  1 * 3 ) //i == 3
return factorial (2,  3 *2 ) //i == 2
return 6  //i == 1
```

So, you can see that each time the factorial function is called in our example, a new value for the current_factorial variable is passed into the factorial function. The function is basically updating the current_factorial variable with each call to the function. We are able to maintain the current factorial value because this function accepts 2 arguments/parameters – not just 1 like our normal, non-tail recursive factorial function above.

Tail recursion versus normal recursion

You can see that once a final call to the tail recursive factorial is made – the "factorial (2, 3 *2)", it will return the value of 6 – which is actually the answer that we want. The most important thing to notice about that is the fact that **all** of the recursive calls to factorial (like factorial (3, 1 * 3), factorial (2, 3 *2), etc) do not actually need to return in order to get the final value of 6 – you can see that we actually arrive at the value of 6 before any of the recursive calls actually return. So, we can say that a function is tail recursive if the final result of the recursive call – in this case 6 – is also the final result of the function itself, and that is why it's called "tail" recursion – because the final function call (the tail-end call) actually holds the final result. If we compare that with our earlier example of "normal" recursion, you should see the difference – the "normal" recursive call is certainly not in it's final state in the last function call, because all of the recursive calls leading up to the last function call must also return in order to actually come up with the final answer.

Tail call optimization explanation

In our tail recursive example, the recursive calls to factorial do not actually ***need*** a new stack frame for each and every recursive call that is made. This is because the calculation is made within the function parameters/arguments – and the final function call actually contains the final result, and the final result does not rely on the return value of each and every recursive call. However, an important point that you should make sure you understand is that **it is up to the compiler/interpreter of the particular language to determine whether or not the recursive calls in a tail recursive function actually use an extra stack frame for each recursive call to the function.**

Some language's compilers choose to optimize tail recursive functions because they know that the final result will be in the very last function call. So, the compilers will not create a new stack frame for each recursive call, and will instead just re-use the same stack frame. This saves memory overhead by using a constant amount of stack space, which is why it is called tail recursion optimization. But, keep in mind that some compilers do not perform this optimization on tail recursive functions, which means that the tail recursive function will be run like a normal recursive function and will use a new stack frame for each and every function call.

Python does not use tail recursive optimization

In our example of tail recursion, we used Python because it made it easy to illustrate our example. But, it's interesting to note that Python does not actually utilize tail recursive *optimization*, which means that it treats tail recursive calls just as it would treat normal recursive calls. This of course means that tail recursive calls in Python will be less efficient then they would be if they were optimized, but there are valid reasons the creator of Python decided not to add this feature.

Tail Recursion versus Iteration

Tail recursion can be as efficient as iteration if the compiler uses what is known as tail recursion optimization. If that optimization is not used, then tail recursion is just as efficient as normal recursion.

Is Tail Recursion optimization automatic?

No, tail recursion optimization is a ***feature*** that must be built in as part of the compiler, as we mentioned before.

Tail recursion optimization and stack overflow

Because tail recursion optimization essentially makes your tail recursive call equivalent to an iterative function, there is no risk of having the stack overflow in an optimized tail recursive function.

Tail Recursion optimization in Java

Tail recursion ***optimization*** is not actually supported by Java because the Java standard does not require that tail recursion be supported, although there are some JVM implementations that do support it as an add-on. But, tail recursion itself (note that we left out the "optimization" part) is supported in Java because it is just a special case of normal recursion – so there's really nothing extra that Java JVM has to ***do*** in order to support tail recursion versus normal recursion. Because there is no tail recursion optimization in Java, code that uses tail recursion will behave like a normal recursive call, and will use multiple stack frames.

Tail call optimization versus tail call elimination

Both tail call optimization and tail call elimination mean exactly the same thing and refer to the same exact process in which the same stack frame is reused by the compiler, and unnecessary memory on the stack is not allocated.

What does TRO stand for in computer science?

TRO stands for Tail recursion optimization.

What does TCE stand for in computer science?

TCE stands for Tail recursion elimination.

« Previous

2 Comments Programmer Interview Login

Sort by Best Share Favorite

Join the discussion...

pranjal gupta · 2 years ago

awesome tutorial. splendid explanation.

Reply · Share ›

aresh · 2 years ago

Awsome tutorial... Thanx a lot sir ji:)

Reply · Share ›

ALSO ON PROGRAMMER INTERVIEW WHAT'S THIS?

Difference between object and class

15 comments • a year ago

Robert — Surely Cat, Lion, Zebra etc. are classes, with a common ancestor of Animal related …

Practice Interview Question 2

10 comments • a year ago

vins — Solution in SQL:SELECT USER1.name,USER1.phone_num,max(U FROM USER1,UserHistory …

Java Reflection Example

3 comments • a year ago

rahul — Check out Reflection in this android app. Clear and precise - …

Clustered vs. Non Clustered Index

38 comments • a year ago

Pratik chavan — Very good explanation .....really now i knw exact meaning of this

Subscribe Add Disqus to your site Privacy

Would you like to thank ProgrammerInterview.com for being a helpful free resource? Then why not tell a friend about us, or **simply add a link to this page from your webpage** using the HTML below.

Link to this page: <a href="http://www.programmerinterview.com/index.php/recursion/tail-call-optimization/">Programmer an connect programmerinterview.com

Please bookmark with social media, your votes are noticed and appreciated: Like 23,622 people like this. Be the first of your friends.