Why are loops faster than recursion?

In practice I understand that any recursion can be written as a loop (and vice versa(?)) and if we measure with actual computers we find that loops are faster than recursion for the same problem. But is there any theory what makes this difference or is it mainly emprical?

time-complexity recursion loops

asked May 1 '16 at 4:43
Dj Dac
243 2 16

8 Looks are only faster than recursion in languages that implement them poorly. In a language with proper Tail Recursion, recursive programs can be translated into loops behind the scenes, in which case there would be no difference because they are identical. – jmite May 1 '16 at 5:37

3 Yes, and if you use a language that supports it, you can use (tail) recursion without having any negative performance effects. – jmite May 1 '16 at 6:10

1 @jmite, tail recursion that can *actually* be optimized into a loop is exceedingly rare, much rarer than you think. Especially in languages that have managed types like reference counted variables. – Johan May 1 '16 at 6:52

1 Since you included the tag time-complexity, I feel I should add that an algorithm with a loop has the same time complexity as an algorithm with recursion, but with the latter, the time taken will be higher by some constant factor, depending on the amount of overhead for recursion. – Lieuwe Vinkhuijzen May 1 '16 at 17:04

2 Hey, since you added bounty with a lot of good answers almost exhausting all possibilities, is there something more you need or feel like something should be clarified? I do not have much to add, I can edit some answer or leave comment, so this is general (not personal) question. – Evil May 5 '16 at 23:44

3 Answers

The reason that loops are faster than recursion is easy.
A loop looks like this in assembly.

```
mov loopcounter,i
dowork:/do work
dec loopcounter
jmp_if_not_zero dowork
```

A single conditional jump and some bookkeeping for the loop counter.

Recursion (when it isn't or cannot be optimized by the compiler) looks like this:

```
start_subroutine:
pop parameter1
pop parameter2
dowork://dowork
test something
jmp_if_true done
push parameter1
push parameter2
call start_subroutine
done:ret
```

It's a lot more complex and you get at least 3 jumps (1 test to see if were done, one call and one return).
Also in recursion the parameters need to be set up and fetched.
None of this stuff is needed in a loop because all the parameters are set up already.

Theoretically the parameters could stay in place with recursion as well, but no compilers that I know of actually go that far in their optimization.

Differences between a call and a jmp
A call-return pair is not very much more expensive then the jmp. The pair takes 2 cycles

and the jmp takes 1; hardly noticeable.
In calling conventions that support register parameters the overhead for parameters in minimal, but even stack parameters are cheap **as long as the CPU's buffers do not overflow**.
It is the overhead of the call setup dictated by the calling convention and parameter handling in use that slows down recursion.
This is very much implementation dependent.

Example of poor recursion handling For example, if a parameter is passed that is reference counted (e.g. a non const managed type parameter) it will add a 100 cycles doing a locked adjustment of the reference count, totally killing performance vs a loop.
In languages that are tuned to recursion this bad behavior does not occur.

CPU optimization
The other reason recursion is slower is that it works against the optimization mechanisms in CPU's.
Returns can only be predicted correctly if there are not too many of them in a row. The CPU has a return stack buffer with a (few) handfuls of entries. Once those run out every additional return will be mispredicted causing huge delays.
On any CPU that uses a stack return buffer call based recursion that exceeds the buffer size is best avoided.

About trivial code examples using recursion
If you use a trivial example of recursion like Fibonacci number generation, then these effects do not occur, because any compiler that is 'knows' about recursion will transform it into a loop, just like any programmer worth his salt would.
If you run these trivial examples in a enviroment that does not optimize properly than the call stack will (needlessly) grow out of bounds.

About tail recursion
Note that sometimes the compiler optimizes away tail recursion by changing it into a loop.
It is best to only rely on this behavior in languages that have a known good track record in this regard.
Many languages insert hidden clean up code before the final return preventing the optimization of tail recursion.

Confusion between true and pseudo recursion
If your programming environment turns your recursive source code into a loop, then it is arguably not true recursion that is being executed.
True recursion requires a store of breadcrumbs, so that the recursive routine can trace back its steps after it exits.
It is the handling of this trail that makes recursion slower than using a loop. This effect is magnified by current CPU implementations as outlined above.

Effect of the programming environment
If your language is tuned towards recursion optimization then by all means go ahead and use recursion at every opportunity. In most cases the language will turn your recursion into some sort of loop.
In those cases where it cannot, the programmer would be hard pressed as well. If your programming language is not tuned towards recursion, then it should be avoided unless the domain is suited towards recursion.
Unfortunately many languages do not handle recursion well.

Misuse of recursion
There is no need to calculate the Fibonacci sequence using recursion, in fact it is a pathological example.
Recursion is best used in languages that explicitly support it or in domains where recursion shines, like the handling of data stored in a tree.

> I understand any recursion can be written as a loop

Yes, if you are willing to put the cart before the horse.
All instances of recursion can be written as a loop, some of those instances require you to use an explicit stack like storage.
If you need to roll your own stack just to turn recursive code into a loop you might as well use plain recursion.
Unless of course you've got special needs like using enumerators in a tree structure and you don't have proper language support.

edited May 2 '16 at 18:57   answered May 1 '16 at 6:45
Johan
771 4 22

These other answers are somewhat misleading. I agree that they state implementation details that can explain this disparity, but they overstate the case. As correctly suggested by jmite, they are implementation-oriented toward *broken* implementations of function calls/recursion. Many languages implement loops via recursion, so loops are clearly not going to be faster in those languages. Recursion is in no way less efficient than looping (when both are applicable) in theory. Let me quote the abstract to Guy Steele's 1977 paper Debunking the "Expensive Procedure Call" Myth or, Procedure Implementations Considered Harmful or, Lambda: the Ultimate GOTO

> Folklore states that GOTO statements are "cheap", while procedure calls are "expensive". This myth is largely a result of poorly designed language implementations. The historical growth of this myth is considered. Both theoretical ideas and an existing implementation are discussed which debunk this myth. It is shown that the unrestricted use of procedure calls permits great stylistic freedom. In particular, any flowchart can be written as a "structured" program without introducing extra variables. The difficulty with the GOTO statement and the procedure call is characterized as a conflict between abstract programming concepts and concrete language constructs.

The "conflict between abstract programming concepts and concrete language constructs" can be seen from the fact that most theoretical models of, for example, the untyped lambda calculus, *don't have a stack*. Of course, this conflict is not necessary as the above paper illustrates and as is also demonstrated by languages who have no iteration mechanism other than recursion such as Haskell.

Let me demonstrate. For simplicity, I'll use an "applied" lambda calculus with numbers and and booleans, and I'll assume we have a fixed-point combinator, `fix`, which satisfies `fix f x = f (fix f) x`. All of this can be reduced to just the untyped lambda calculus without changing my argument. The archetypal way of understanding the evaluation of the lambda calculus is through term rewriting with the central rewrite rule of beta reduction, namely $(\lambda x.\, M)N \rightsquigarrow M[N/x]$ where $[N/x]$ means "replace all free occurrences of x in M with N" and $\rightsquigarrow$ means "rewrites to". This is just the formalization of substituting the arguments of a function call into the functions body.

Now for an example. Define `fact` as

```
fact = fix (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) 1
```

Here's the evaluation of `fact 3`, where, for compactness, I'll use `g` as synonym for `fix (λf.λa.λn.if n == 0 then a else f (a*n) (n-1))`, i.e. `fact = g 1`. This doesn't affect my argument.

```
fact 3
~> g 1 3
~> fix (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) 1 3
~> (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) g 1 3
~> (λa.λn.if n == 0 then a else g (a*n) (n-1)) 1 3
~> (λn.if n == 0 then 1 else g (1*n) (n-1)) 3
~> if 3 == 0 then 1 else g (1*3) (3-1)
~> g (1*3) (3-1)
~> g 3 2
~> fix (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) 3 2
~> (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) g 3 2
~> (λa.λn.if n == 0 then a else g (a*n) (n-1)) 3 2
~> (λn.if n == 0 then 3 else g (3*n) (n-1)) 2
~> if 2 == 0 then 3 else g (3*2) (2-1)
~> g (3*2) (2-1)
~> g 6 1
~> fix (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) 6 1
~> (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) g 6 1
~> (λa.λn.if n == 0 then a else g (a*n) (n-1)) 6 1
~> (λn.if n == 0 then 6 else g (6*n) (n-1)) 1
~> if 1 == 0 then 6 else g (6*1) (1-1)
~> g (6*1) (1-1)
~> g 6 0
~> fix (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) 6 0
~> (λf.λa.λn.if n == 0 then a else f (a*n) (n-1)) g 6 0
~> (λa.λn.if n == 0 then a else g (a*n) (n-1)) 6 0
~> (λn.if n == 0 then 6 else g (6*n) (n-1)) 0
~> if 0 == 0 then 6 else g (6*0) (0-1)
~> 6
```

You can see from the shape without even looking at the details that there is no growth and each iteration needs the same amount of space. (Technically, the numeric result grows which is unavoidable and just as true for a `while` loop.) I defy you to point out the boundlessly growing "stack" here.

It seems the archetypal semantics of the lambda calculus already does what is commonly misnamed "tail call optimization". Of course, no "optimization" is happening here. There are no special rules here for "tail" calls as opposed to "normal" calls. For this reason, it's hard to give an "abstract" characterization of what tail call "optimization" is doing, as in many abstract characterizations of function call semantics, there is nothing for tail call "optimization" to do!

That the analogous definition of `fact` in many languages "stack overflows", is a failure by those languages to correctly implement function call semantics. (Some languages have an excuse.) The situation is roughly analogous to having a language implementation that implemented arrays with linked lists. Indexing into such "arrays" would then be an O(n) operation which doesn't meet the expectation of arrays. If I made a separate implementation of the language, that used real arrays instead of linked lists, you wouldn't say I've implemented "array access optimization", you would say I fixed a broken implementation of arrays.

So, responding to Veedrac's answer. Stacks are *not* "fundamental" to recursion. To the extent that "stack-like" behavior occurs during the course of evaluation, this can only happen in cases where loops (without an auxiliary data structure) would not be applicable in the first place! To put it another way, I can implement loops with recursion with exactly the same performance characteristics. Indeed, Scheme and SML both contain looping constructs, but both of them define those in terms of recursion (and, at least in Scheme, `do` is often *implemented* as a macro that expands into recursive calls.) Similarly, for Johan's answer, nothing says a compiler must emit the assembly Johan described for recursion. Indeed, nothing says a compiler can't emit *exactly the same* assembly whether you use loops or recursion. The only time the compiler would be (somewhat) *obligated* to emit assembly like what Johan describes is when you are doing something that isn't expressible by a loop anyway. As outlined in Steele's paper and demonstrated by the actual practice of languages like Haskell, Scheme, and SML, it is not "exceedingly rare" that tail calls can be "optimized", they can *always* be "optimized". Whether a particular use of recursion will run in constant space depends on how it is written, but the restrictions you need to apply to make that possible are the restrictions you'd need to fit your problem into the shape of a loop. (Actually, they are less stringent. There are problems, such as encoding state machines, that are more cleanly and efficiently handled via tails calls as opposed to loops which would require auxiliary variables.) Again, the only time recursion *requires* doing more work is when your code isn't a loop anyway.

My guess is Johan is referring to C compilers which have arbitrary restrictions on when it will perform tail call "optimization". Johan also is presumably referring to languages like C++ and Rust when he talks about "languages with managed types". The RAII idiom from C++ and present in Rust as well makes things which superficially look like tail calls, not tail calls (because the "destructors" still need to be called). There have been proposals to use a different syntax to opt-in to a slightly different semantics that would allow tail recursion (namely call destructors *before* the final tail call and obviously disallow accessing "destroyed" objects). (Garbage collection has no such issue, and all of Haskell, SML, and Scheme are garbage collected languages.) In a quite different vein, some languages, such as Smalltalk, expose the "stack" as a first-class object, in these cases the "stack" is no longer an implementation detail, though this doesn't preclude having separate types of calls with different semantics. (Java says it can't due to the way it handles some aspects of security, but this is actually false.)

In practice, the prevalence of broken implementations of function calls come from three main factors. First, many languages inherit the broken implementation from their implementation language (usually C). Second, deterministic resource management is nice and does make the issue more complicated, though only a handful of languages offer this. Third, and, in my experience, the reason most people care about, is that they want stack traces when errors occur for debugging purposes. Only the second reason is one that can be potentially theoretically motivated.

answered May 1 '16 at 21:08
Derek Elkins
2,989 4 11

I used "fundamental" to refer to the most basic reason that the claim is true, not on whether it logically has to be this way (which clearly it does not, as the two programs are provably identical). But I disagree with your comment as a whole. Your use of lambda calculus doesn't remove the stack as much as obscure it. – Veedrac May 2 '16 at 6:27

Your claim "The only time the compiler would be (somewhat) obligated to emit assembly like what Johan describes is when you are doing something that isn't expressible by a loop anyway." is also quite strange; a compiler is (normally) able to produce any code that produces the same output, so your comment is basically a tautology. But in practice compilers do produce different code for different equivalent programs, and the question was about why. – Veedrac May 2 '16 at 6:30

In practice a stack refers to those variables captured from outer function's frames, and the difference is in labelling. It's true that generalized reduction on these "stacks" has some desirable properties, but that hardly makes a typical language's stack *broken* any more than a vector is broken for not having $\mathcal{O}(1)$ prepends. This is especially true given that tail recursion doesn't optimize all the cases that are optimized with more general reductions. – Veedrac May 2 '16 at 6:35

To give an analogy, responding to a question of why adding immutable strings in loops takes quadratic time with "it doesn't have to be" would be entirely reasonable, but going on to claim that the implementation was thus broken would not. – Veedrac May 2 '16 at 6:43

Very interesting answer. Even though it sounds a bit like a rant :-). Upvoted because I learned something new. – Johan May 2 '16 at 18:51

Fundamentally the difference is that recursion includes a stack, an auxiliary data structure you probably don't want, whereas loops do not automatically do so. Only in rare cases is a typical compiler able to deduce that you don't actually need the stack after all.

If you compare instead loops operating manually on an allocated stack (eg. through a pointer to heap memory), you'll normally find them no faster or even slower than using the hardware stack.

answered May 1 '16 at 16:39
Veedrac
353 1 10