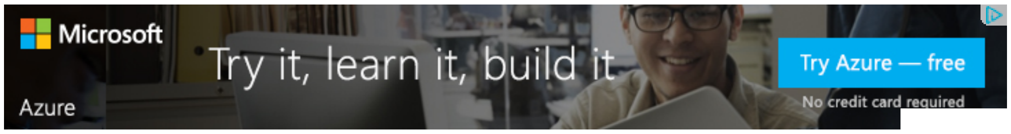
Is recursion ever faster than looping?



I know that recursion is sometimes a lot cleaner than looping, and I'm not asking anything about when I should use recursion over iteration, I know there are lots of questions about that already.

What I'm asking is, is recursion **ever** faster than a loop? To me it seems like, you would always be able to refine a loop and get it to perform more quickly than a recursive function because the loop is absent constantly setting up new stack frames.

I'm specifically looking for whether recursion is faster in applications where recursion is the right way to handle the data, such as in some sorting functions, in binary trees, etc.

performance loops recursion iteration

edited Oct 25 '10 at 0:32
Roger Pate

asked Apr 16 '10 at 6:42
Carson Myers
17k 25 102 147

2 Sometimes iterative procedure or closed-form formulas for some recurrences take centuries to turn up. I think only at those times recursion is faster :) lol – Pratik Deoghare Apr 16 '10 at 6:52

9 Speaking for myself, I much prefer iteration. ;-) – Iterator Oct 27 '11 at 13:33

possible duplicate of Recursion or Iteration? – nawfal Dec 14 '13 at 7:29

3 see stackoverflow.com/questions/2651112/… – richard Jun 22 '15 at 20:56

12 Answers

This depends on the language being used. You wrote 'language-agnostic', so I'll give some examples.

In Java, C, and Python, recursion is fairly expensive compared to iteration (in general) because it requires the allocation of a new stack frame. In some C compilers, one can use a compiler flag to eliminate this overhead, which transforms certain types of recursion (actually, certain types of tail calls) into jumps instead of function calls.

In functional programming language implementations, sometimes, iteration can be very expensive and recursion can be very cheap. In many, recursion is transformed into a simple jump, but changing the loop variable (which is mutable) *sometimes* requires some relatively heavy operations, especially on implementations which support multiple threads of execution. Mutation is expensive in some of these environments because of the interaction between the mutator and the garbage collector, if both might be running at the same time.

I know that in some Scheme implementations, recursion will generally be faster than looping.

In short, the answer depends on the code and the implementation. Use whatever style you prefer. If you're using a functional language, recursion *might* be faster. If you're using an imperative language, iteration is *probably* faster. In some environments, both methods will result in the same assembly being generated (put that in your pipe and smoke it).

Addendum: In some environments, the best alternative is neither recursion nor iteration but instead higher order functions. These include "map", "filter", and "reduce" (which is also called "fold"). Not only are these the preferred style, not only are they often cleaner, but in some environments these functions are the first (or only) to get a boost from automatic parallelization — so they can be significantly faster than either iteration or recursion. Data Parallel Haskell is an example of such an environment.

List comprehensions are another alternative, but these are usually just syntactic sugar for iteration, recursion, or higher order functions.

edited Apr 16 '10 at 7:45 answered Apr 16 '10 at 7:00

Dietrich Epp
125k 23 220 293

31 I +1 that, and would like to comment that "recursion" and "loops" are just what humans name their code. What matters for performance is not how you *name* things, but rather how they are compiled/interpreted. Recursion, by definition, is a mathematical concept, and has little to do with stack frames and assembly stuff. – Pavel Shved Apr 16 '10 at 7:05

1 Also, recursion is, in general, the more natural approach in functional languages, and iteration is normally more intuitive in imperative languages. The performance difference is unlikely to be noticeable, so just use whatever feels more natural for that particular language. For example, you probably wouldn't want to use iteration in Haskell when recursion is much more simple. – Sasha Chedygov Apr 16 '10 at 7:43

1 Generally recursion is *compiled* to loops, with loops being a lower level construct. Why? Because recursion is typically well founded over some data structure, inducing an Initial F-algebra and allowing you to prove some properties about termination along with inductive arguments about the structure of the (recursive) computation. The process by which recursion is compiled to loops is tail call optimization. – Kristopher Micinski Sep 16 '12 at 9:40

> is recursion ever faster than a loop?

No, Iteration will always be faster than Recursion. (in a Von Neumann Architecture)

Explanation:

If you build the minimum operations of a generic computer from scratch, "Iteration" comes first as a building block and is less resource intensive than "recursion", ergo is faster.

Building a pseudo-computing-machine from scratch:

Question yourself: What do you need to ***compute*** a value, i.e. to follow an algorithm and reach a result?

We will establish a hierarchy of concepts, starting from scratch and defining in first place the basic, core concepts, then build second level concepts with those, and so on.

1. First Concept: **Memory cells, storage, State**. To do something you need *places* to store final and intermediate result values. Let's assume we have an infinite array of "integer" cells, called **Memory**, M[0..Infinite].
2. **Instructions:** do something - transform a cell, change its value. **alter state**. Every interesting instruction performs a transformation. Basic instructions are:

   a) **Set & move memory cells**

   - store a value into memory, e.g.: ***store 5 m[4]***
   - copy a value to another position: e.g.: ***store m[4] m[8]***

   b) **Logic and arithmetic**

   - and, or, xor, not
   - add, sub, mul, div. e.g. ***add m[7] m[8]***
3. **An Executing Agent**: a *core* in a modern CPU. An "agent" is something that can execute instructions. An *Agent* can also be a person following the algorithm on paper.
4. **Order of steps: a sequence of instructions**: i.e.: do this first, do this after, etc. An imperative sequence of instructions. Even one line *expressions* are "an imperative

sequence of instructions". If you have an expression with a specific "order of evaluation" then you have ***steps***. It means than even a single composed expression has implicit “steps” and also has an implicit local variable (let’s call it “result”). e.g.:

```
4 + 3 * 2 - 5
(- (+ (* 3 2) 4 ) 5)
(sub (add (mul 3 2) 4 ) 5)
```

The expression above implies 3 steps with an implicit "result" variable.

```
// pseudocode

       1. result = (mul 3 2)
       2. result = (add 4 result)
       3. result = (sub result 5)
```

So even infix expressions, since you have a specific order of evaluation, are *an imperative sequence of instructions*. The expression *implies* a sequence of operations to be made in a specific order, and because there are *steps*, there is also an implicit "result" intermediate variable.

5. **Instruction Pointer**: If you have a sequence of steps, you have also an implicit "instruction pointer". The instruction pointer marks the next instruction, and advances after the instruction is read but before the instruction is executed.

   In this pseudo-computing-machine, the Instruction Pointer is part of *Memory*. (Note: Normally the **Instruction Pointer** will be a “special register” in a CPU core, but here we will simplify the concepts and assume all data (registers included) are part of “Memory”)

6. **Jump** - Once you have an ordered number of steps and an *Instruction Pointer*, you can apply the "**store**" instruction to alter the value of the Instruction Pointer itself. We will call this specific use of the *store instruction* with a new name: **Jump**. We use a new name because is easier to think about it as a new concept. By altering the instruction pointer we're instructing the agent to “go to step x“.

7. **Infinite Iteration**: By ***jumping back,*** now you can make the agent "repeat" a certain number of steps. At this point we have **infinite Iteration.**

```
                   1. mov 1000 m[30]
                   2. sub m[30] 1
                   3. jmp-to 2  // infinite loop
```

8. **Conditional** - Conditional execution of instructions. With the "conditional" clause, you can conditionally execute one of several instructions based on the current state (which can be set with a previous instruction).

9. **Proper Iteration**: Now with the **conditional** clause, we can escape the infinite loop of the **jump back** instruction. We have now a **conditional loop** and then **proper Iteration**

```
1. mov 1000 m[30]
2. sub m[30] 1
3. (if not-zero) jump 2  // jump only if the previous
                        // sub instruction did not result in 0

// this loop will be repeated 1000 times
// here we have proper ***iteration***, a conditional loop.
```

10. **Naming**: giving names to a specific memory location holding data or holding a *step*. This is just a "convenience" to have. We do not add any new instructions by having the capacity to define “names” for memory locations. “Naming” is not a instruction for the agent, it’s just a convenience to us. *Naming* makes code (at this point) easier to read and easier to change.

```
   #define counter m[30]   // name a memory location
   mov 1000 counter
loop:                      // name a instruction pointer location
    sub counter 1
    (if not-zero) jmp-to loop
```

11. **One-level subroutine**: Suppose there’s a series of steps you need to execute frequently. You can store the steps in a named position in memory and then *jump to* that position when you need to execute them (call). At the end of the sequence you'll need to *return* to the point of *calling* to continue execution. With this mechanism, you’re *creating new instructions* (subroutines) by composing core instructions.

    Implementation: (no new concepts required)

    - Store the current Instruction Pointer in a predefined memory position
    - **jump** to the subroutine
    - at the end of the subroutine, you retrieve the Instruction Pointer from the predefined memory location, effectively jumping back to the following instruction of the original *call*

Problem with the **one-level** implementation: You cannot call another subroutine from a subroutine. If you do, you'll overwrite the returning address (global variable), so you cannot nest calls.

To have a **better Implementation for subroutines: You need a STACK**

12. **Stack**: You define a memory space to work as a "stack", you can “push” values on the stack, and also “pop” the last “pushed” value. To implement a stack you'll need a ***Stack Pointer*** (similar to the Instruction Pointer) which points to the actual “head” of the stack. When you “push” a value, the stack pointer decrements and you store the value. When you “pop”, you get the value at the actual Stack Pointer and then the Stack Pointer is incremented.
13. **Subroutines** Now that we have a ***stack*** we can implement proper subroutines *allowing nested calls*. The implementation is similar, but instead of storing the Instruction Pointer in a predefined memory position, we "push" the value of the IP in the *stack. *At the end of the subroutine, we just “pop” the value from the stack, effectively jumping back to the instruction after the original *call. This implementation, having a “stack” allows calling a subroutine from another subroutine. With this implementation we can create several levels of abstraction when defining *new instructions* as subroutines, by using core instructions or other subroutines as building blocks.
14. **Recursion**: What happens when a subroutine calls itself?. This is called "recursion".

    Problem: Overwriting the local intermediate results a subroutine can be storing in memory. Since you are calling/reusing the same steps, ***if*** the intermediate result are stored in predefined memory locations (global variables) they will be overwritten on the nested calls.

    Solution: To allow recursion, subroutines should store local intermediate results ***in the stack***, therefore, on each *recursive call* (direct or indirect) the intermediate results are stored in different memory locations.

...

having reached ***recursion*** we stop here.

Conclusion:

In a Von Neumann Architecture, clearly ***"Iteration"*** is a simpler/basic concept than ***“Recursion"***. We have a form of **"Iteration"** at level 7, while **"Recursion"** is at level 14 of the concepts hierarchy.

Iteration will always be faster in machine code because it implies less instructions therefore less CPU cycles.

Which one is "better"?

- You should use "iteration" when you are processing simple, sequential data structures, and everywhere a “simple loop” will do.
- You should use "recursion" when you need to process a recursive data structure (I like to call them “Fractal Data Structures”), or when the recursive solution is clearly more “elegant”.

Advice: use the best tool for the job, but understand the inner workings of each tool in order to choose wisely.

Finally, note that you have plenty of opportunities to use recursion. You have *Recursive Data Structures* everywhere, you’re looking at one now: parts of the DOM supporting what you are reading are a RDS, a JSON expression is a RDS, the hierarchical file system in your computer is a RDS, i.e: you have a root directory, containing files and directories, every directory containing files and directories, every one of those directories containing files and directories...

edited Feb 28 '16 at 17:16 answered Jan 19 '15 at 20:48
Lucio M. Tato
2,131 17 20

Recursion may well be faster where the alternative is to explicitly manage a stack, like in the sorting or binary tree algorithms you mention.

I've had a case where rewriting a recursive algorithm in Java made it slower.

So the right approach is to first write it in the most natural way, only optimize if profiling shows it is critical, and then measure the supposed improvement.

answered Apr 16 '10 at 18:13

starblue
40k 11 67 122

1 +1 for "*first write it in the most natural way*" and especially "*only optimize if profiling shows it is critical*" – TripeHound Apr 16 '14 at 14:39

2 +1 for acknowledging that the hardware stack may be faster than a software, manually implemented, in-heap stack. Effectively showing that all the "no" answers are incorrect. – sh1 May 7 '15 at 18:46

Consider what absolutely must be done for each, iteration and recursion.

- iteration: a jump to beginning of loop
- recursion: a jump to beginning of called function

You see that there is not much room for differences here.

(I assume recursion being a tail-call and compiler being aware of that optimization).

answered Apr 16 '10 at 7:38
Pasi Savolainen
1,785 1 14 28

Tail recursion is as fast as looping. Many functional languages have tail recursion implemented in them.

answered Apr 16 '10 at 6:54
mkorpela
2,396 2 11 20

32 Tail recursion *can be* as fast as looping, when a tail call optimization is implemented: c2.com/cgi/wiki?TailCallOptimization – Joachim Sauer Apr 16 '10 at 7:13

Most answers here forget the obvious culprit why recursion is often slower than iterative solutions. It's linked with the build up and tear down of stack frames but is not exactly that. It's generally a big difference in the storage of the auto variable for each recursion. In an iterative algorithm with a loop, the variables are often held in registers and even if they spill, they will reside in the Level 1 cache. In a recursive algorithm, all intermediary states of the variable are stored on the stack, meaning they will engender many more spills to memory. This means that even if it makes the same amount of operations, it will have a lot memory accesses in the hot loop and what makes it worse, these memory operations have a lousy reuse rate making the caches less effective.

TL;DR recursive algorithms have generally a worse cache behavior than iterative ones.

edited Sep 5 '16 at 12:45
answered Nov 19 '14 at 17:42
Patrick Schlüter
8,046 27 38

Most of the answers here are **wrong**. The right answer is **it depends**. For example, here are two C functions which walks through a tree. First the recursive one:

```
static
void mm_scan_black(mm_rc *m, ptr p) {
    SET_COL(p, COL_BLACK);
    P_FOR_EACH_CHILD(p, {
        INC_RC(p_child);
        if (GET_COL(p_child) != COL_BLACK) {
            mm_scan_black(m, p_child);
        }
    });
}
```

And here is the same function implemented using iteration:

```
static
void mm_scan_black(mm_rc *m, ptr p) {
    stack *st = m->black_stack;
    SET_COL(p, COL_BLACK);
    st_push(st, p);
    while (st->used != 0) {
        p = st_pop(st);
        P_FOR_EACH_CHILD(p, {
            INC_RC(p_child);
            if (GET_COL(p_child) != COL_BLACK) {
                SET_COL(p_child, COL_BLACK);
                st_push(st, p_child);
            }
        });
    }
}
```

It's not important to understand the details of the code. Just that `p` are nodes and that `P_FOR_EACH_CHILD` does the walking. In the iterative version we need an explicit stack `st` onto which nodes are pushed and then popped and manipulated.

The recursive function runs much faster than the iterative one. The reason is because in the latter, for each item, a `CALL` to the function `st_push` is needed and then another to `st_pop` .

In the former, you only have the recursive `CALL` for each node.

Plus, accessing variables on the callstack is incredibly fast. It means you are reading from memory which is likely to always be in the innermost cache. An explicit stack, on the other hand, has to be backed by `malloc` :ed memory from the heap which is much slower to access.

With careful optimization, such as inlining `st_push` and `st_pop` , I can reach roughly parity with the recursive approach. But at least on my computer, the cost of accessing heap memory is bigger than the cost of the recursive call.

But this discussion is mostly moot because recursive tree walking is **incorrect**. If you have a large enough tree, you will run out of callstack space which is why an iterative algorithm must be used.

edited Apr 18 '16 at 2:00 answered Apr 18 '16 at 1:42
Björn Lindqvist
7,556 6 43 74

I can confirm that I have run into a similar situation, and that there are situations where recursion can be faster than a manual stack on the heap. Especially when optimization is turned on in the compiler to avoid some of the overhead of calling a function. – while1fork Mar 19 at 23:03

In any realistic system, no, creating a stack frame will always be more expensive than an INC and a JMP. That's why really good compilers automatically transform tail recursion into a call to the same frame, i.e. without the overhead, so you get the more readable source version and the more efficient compiled version. A *really, really* good compiler should even be able to transform normal recursion into tail recursion where that is possible.

answered Apr 16 '10 at 6:54
Kilian Foth
10.5k 3 23 40

Functional programming is more about "**what**" rather than "**how**".

The language implementors will find a way to optimize how the code works underneath, if we don't try to make it more optimized than it needs to be. Recursion can also be optimized within the languages that support tail call optimization.

What matters more from a programmer standpoint is readability and maintainability rather than optimization in the first place. Again, "premature optimization is root of all evil".

answered Dec 1 '16 at 3:53
Ender Aydin Orak
1,468 9 12

This is a guess. Generally recursion probably doesn't beat looping often or ever on problems of decent size if both are using really good algorithms(not counting implementation difficulty) , it may be different if used with a language w/ tail call recursion(and a tail recursive algorithm and with loops also as part of the language)-which would probably have very similar and possibly even prefer recursion some of the time.

answered Apr 16 '10 at 7:04
Roman A. Taycher
4,609 14 57 107

In general, no, recursion will not be faster than a loop in any realistic usage that has viable implementations in both forms. I mean, sure, you could code up loops that take forever, but there would be better ways to implement the same loop that could outperform any implementation of the same problem via recursion.

You hit the nail on the head regarding the reason; creating and destroying stack frames is more expensive than a simple jump.

However, do note that I said "has viable implementations in both forms". For things like many sorting algorithms, there tends to not be a very viable way of implementing them that doesn't effectively set up its own version of a stack, due to the spawning of child "tasks" that are inherently part of the process. Thus, recursion may be just as fast as attempting to implement the algorithm via looping.

Edit: This answer is assuming non-functional languages, where most basic data types are mutable. It does not apply to functional languages.

edited Apr 16 '10 at 7:22

answered Apr 16 '10 at 6:47
Amber
274k 42 446 437

That's also why several cases of recursion are often optimized by compilers in languages where recursion is frequently used. In F#, for instance, in addition to full support to tail recursive functions with the .tail opcode, you often see a recursive function compiled as a loop. – em70 Apr 16 '10 at 6:51

Yep. Tail recursion can sometimes be the best of both worlds - the functionally "appropriate" way to implement a recursive task, and the performance of using a loop. – Amber Apr 16 '10 at 6:53

1 This is not, in general, correct. In some environments, mutation (which interacts with GC) is more expensive than tail recursion, which is transformed into a simpler loop in the output which does not use an extra stack frame. – Dietrich Epp Apr 16 '10 at 7:04

According to theory its the same things. Recursion and loop with the same O() complexity will work with the same theoretical speed, but of course real speed depends on language, compiler and processor. Example with power of number can be coded in iteration way with O(ln(n)):

```
int power(int t, int k) {
int res = 1;
while (k) {
  if (k & 1) res *= t;
  t *= t;
  k >>= 1;
}
return res;
}
```

answered Apr 16 '10 at 8:01
Hydrophis Spiralis
70 7

1 Big O is “proportional to”. So both are `O(n)` , but one **may** take `x` times longer than the other, for all `n` . – richard Jun 22 '15 at 21:02

protected by Srikar Appal Jul 29 '13 at 4:35

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?