Can every recursion be converted into iteration?

A reddit thread brought up an apparently interesting question:

Tail recursive functions can trivially be converted into iterative functions. Other ones, can be transformed by using an explicit stack. Can *every* recursion be transformed into iteration?

The (counter?)example in the post is the pair:

```
(define (num-ways x y)
  (case ((= x 0) 1)
        ((= y 0) 1)
        (num-ways2 x y) ))

(define (num-ways2 x y)
  (+ (num-ways (- x 1) y)
     (num-ways x (- y 1))
```

language-agnostic recursion iteration

edited May 31 '09 at 10:41

asked May 31 '09 at 9:48

Tordek
4,408 1 21 44

3 I don't see how this is a counter-example. The stack technique will work. It won't be pretty, and I'm not going to write it, but it is doable. It appears akdas acknowledges that in your link. – Matthew Flaschen May 31 '09 at 10:00

Your (num-ways x y) is just (x+y) `choose` x = (x+y)!/(x!y!), which doesn't need recursion. – ShreevatsaR Jul 7 '09 at 21:18

5 That's already been answered: please see this question – Tim Nov 2 '09 at 16:49

1 Duplicate of: stackoverflow.com/questions/531668 – Henk Holterman Nov 2 '09 at 17:03

1 @Ravi - In all seriousness - if I were you I would take the time to look at those two linked questions and follow those links - there are good answers - you can figure it out for yourself form there. – Tim Nov 2 '09 at 17:07

show **5** more comments

19 Answers

Can you always turn a recursive function into an iterative one? Yes, absolutely, and the Church-Turing thesis proves it if memory serves. In lay terms, it states that what is computable by recursive functions is computable by an iterative model (such as the Turing machine) and vice versa. The thesis does not tell you precisely how to do the conversion, but it does say that it's definitely possible.

In many cases, converting a recursive function is easy. Knuth offers several techniques in "The Art of Computer Programming". And often, a thing computed recursively can be computed by a completely different approach in less time and space. The classic example of this is Fibonacci numbers or sequences thereof. You've surely met this problem in your degree plan.

On the flip side of this coin, we can certainly imagine a programming system so advanced as to treat a recursive definition of a formula as an invitation to memoize prior results, thus offering the speed benefit without the hassle of telling the computer exactly which steps to follow in the computation of a formula with a recursive definition. Dijkstra almost certainly did imagine such a system. He spent a long time trying to separate the implementation from the semantics of a programming language. Then again, his non-deterministic and multiprocessing programming languages are in a league above the practicing professional programmer.

In the final analysis, many functions are just plain easier to understand, read, and write in recursive form. Unless there's a compelling reason, you probably shouldn't (manually) convert these functions to an explicitly iterative algorithm. Your computer will handle that job correctly.

I can see one compelling reason. Suppose you've a prototype system in a super-high level language like

[*donning asbestos underwear*] Scheme, Lisp, Haskell, OCaml, Perl, or Pascal. Suppose conditions are such that you need an implementation in C or Java. (Perhaps it's politics.) Then you could certainly have some functions written recursively but which, translated literally, would explode your runtime system. For example, infinite tail recursion is possible in Scheme, but the same idiom causes a problem for existing C environments. Another example is the use of lexically nested functions and static scope, which Pascal supports but C doesn't.

In these circumstances, you might try to overcome political resistance to the original language. You might find yourself reimplementing Lisp badly, as in Greenspun's (tongue-in-cheek) tenth law. Or you might just find a completely different approach to solution. But in any event, there is surely a way.

answered Jun 1 '09 at 8:32

Ian
2,192 7 13

2 Isn't Church-Turing yet to be proven? – Liran Orevi Jul 8 '09 at 7:57

3 Here's a really short outline: pick two models of computation A and B. Prove that A is at least as powerful as B by writing an interpreter of B using A. Do this in both directions, and you have shown that A and B have equivalent power. Consider that machine code is almost the Turing-machine model, and that lisp interpreters/compilers exist. The debate should be over. But for more information, see: alanturing.net/turing_archive/pages/Reference%20Articles/… – Ian Jul 31 '09 at 5:41

Ian, I'm not sure that proves each has equivalent power, only that each uses equivalent power. Use demonstrates certain capabilities, but not necessarily the extent of them. – eyelidlessness Nov 2 '09 at 17:20

9 @eyelidlessness: If you can implement A in B, it means B has at least as much power as A. If you cannot execute some statement of A in the A-implementation-of-B, then it's not an implementation. If A can be implemented in B and B can be implemented in A, power(A) >= power(B), and power(B) >= power(A). The only solution is power(A) == power(B). – Tordek Jan 1 '10 at 7:31

2 re: 1st paragraph: You are speaking about equivalence of models of computation, not the Church-Turing thesis. The equivalence was AFAIR proved by Church and/or Turing, but it is not the thesis. The thesis is an experimental fact that everything intuitively computable is computable in strict mathematical sense (by Turing machines/recursive functions etc.). It could be disproven if using laws of physics we could build some nonclassical computers computing something Turing machines cannot do (e.g. halting problem). Whereas the equivalence is a mathematical theorem, and it will not be disproven. – sdcvvc Apr 14 '12 at 23:20

show **6** more comments

Is it always possible to write a non-recursive form for every recursive function?

Yes. A simple formal proof is to show that both μ recursion and a non-recursive calculus such as GOTO are both Turing complete. Since all Turing complete calculi are strictly equivalent in their expressive power, all recursive functions can be implemented by the non-recursive Turing-complete calculus.

Unfortunately, I'm unable to find a good, formal definition of GOTO online so here's one:

A GOTO program is a sequence of commands *P* executed on a register machine such that *P* is one of the following:

- `HALT` , which halts execution
- $r = r + 1$ where r is any register
- $r = r - 1$ where r is any register
- `GOTO` x where x is a label
- `IF` $r \neq 0$ `GOTO` x where r is any register and x is a label
- A label, followed by any of the above commands.

However, the conversions between recursive and non-recursive functions isn't always trivial (except by mindless manual re-implementation of the call stack).

edited May 24 '12 at 15:05

answered Nov 2 '09 at 17:08

Konrad Rudolph
241k 51 516 772

Aha! Konrad, you are back. – bludger Nov 2 '09 at 17:18

Great answer! However in practice I have great difficulty turing recursive algos into iterative ones. For example I was unable so far to turn the monomorphic typer presented here community.topcoder.com/… into an iterative algorithm – Nils Aug 21 '11 at 12:23

add a comment

Recursion is implemented as stacks or similar constructs in the actual interpreters or compilers. So you certainly can convert a recursive function to an iterative counterpart **because that's how it's always done (if automatically)**. You'll just be duplicating the compiler's work in an ad-hoc and probably in a very ugly and inefficient manner.

answered May 31 '09 at 10:10

Vinko Vrsalovic
121k 26 241 298

add a comment

Basically yes, in essence what you end up having to do is replace method calls (which implicitly push state onto the stack) into explicit stack pushes to remember where the 'previous call' had gotten up to, and then execute the 'called method' instead.

I'd imagine that the combination of a loop, a stack and a state-machine could be used for all scenarios by basically simulating the method calls. Whether or not this is going to be 'better' (either faster, or more efficient in some sense) is not really possible to say in general.

answered May 31 '09 at 10:01

jerryjvl
9,425 6 18 41

add a comment

---

yes, using explicitly a stack but recursion is far more pleasant to read imho

answered May 31 '09 at 9:52

dfa
57k 14 124 185

9 I wouldn't say it's always more pleasant to read. Both iteration and recursion have their place. – Matthew Flaschen May 31 '09 at 10:02

add a comment

Yes, it's always possible to write a non-recursive version. The trivial solution is to use a stack data structure and simulate the recursive execution.

answered Nov 2 '09 at 16:52

Heinzi
69.3k 13 128 234

Which either defeats the purpose if your stack data structure is allocated on the stack, or takes way longer if it's allocated on the heap, no? That sounds trivial but inefficient to me. – conradk Sep 18 at 22:11

add a comment

In principle it is always possible to remove recursion and replace it with iteration in a language that has infinite state both for data structures and for the call stack. This is a basic consequence of the Church-Turing thesis.

Given an actual programming language, the answer is not as obvious. The problem is that it is quite possible to have a language where the amount of memory that can be allocated in the program is limited but where the amount of call stack that can be used is unbounded (32-bit C where the address of stack variables is not accessible). In this case, recursion is more powerful simply because it has more memory it can use; there is not enough explicitly allocatable memory to emulate the call stack. For a detailed discussion on this, see this discussion.

edited Aug 24 '09 at 13:39

answered Jul 8 '09 at 7:47

Zayenz
801 5 9

add a comment

- Recursive function execution flow, can be represented as a tree
- The same logic can be done by a loop, that use use a data-structure to traverse that tree
- Depth-First Traversal can be done using a Stack, Breadth-First Traversal can be done using a Queue

So the answer is Yes,

Why: http://stackoverflow.com/a/531721/2128327

Can any recursion be done in a single loop, yes .. why:

A Turing machine does everything it does by executing a single loop: 1. fetch an instruction, 2. evaluate it, 3. goto 1. -mokus

edited Mar 23 '13 at 16:29

answered Mar 23 '13 at 16:17

Khaled A Khunaifer
2,385 8 23

add a comment

Removing recursion is a complex problem and is feasible under well defined circumstances.

The below cases are among the easy:

- tail recursion
- direct linear recursion

answered May 31 '09 at 10:01


Nick Dandoulakis
27.3k 8 59 102

add a comment

I'd say yes - a function call is nothing but a goto and a stack operation (roughly speaking). All you need to do is imitate the stack that's built while invoking functions and do something similar as a goto (you may imitate gotos with languages that don't explicitly have this keyword too).

answered Nov 2 '09 at 16:52


sfussenegger
16.7k 3 53 84

1 I think the OP is looking for a proof or something else substantive – Tim Nov 2 '09 at 16:56

add a comment

Have a look at the following entries on wikipedia, you can use them as a starting point to find a complete answer to your question.

- Recursion in computer science
- Recurrence relation

Follows a paragraph that may give you some hint on where to start:

> Solving a recurrence relation means obtaining a closed-form solution: a non-recursive function of n.

Also have a look at the last paragraph of this entry.

answered Nov 2 '09 at 17:05


Alberto Zaccagni
12.1k 1 30 67

add a comment

Appart from the explicit stack, another pattern for converting recursion into iteration is with the use of a trampoline.

Here, the functions either return the final result, or a closure of the function call that it would otherwise have performed. Then, the initiating (trampolining) function keep invoking the closures returned until the final result is reached.

This approach works for mutually recursive functions, but I'm afraid it only works for tail-calls.

http://en.wikipedia.org/wiki/Trampoline_(computers)

answered May 31 '09 at 10:17


Chris Vest
5,742 1 19 38

add a comment

Sometimes replacing recursion is much easier than that. Recursion used to be the fashionable thing taught in CS in the 1990's, and so a lot of average developers from that time figured if you solved something with recursion, it was a better solution. So they would use recursion instead of looping backwards to reverse order, or silly things like that. So sometimes removing recursion is a simple "duh, that was obvious" type of exercise.

This is less of a problem now, as the fashion has shifted towards other technologies.

answered May 31 '09 at 11:23


Matthias Wandel
3,142 4 21 28

add a comment

Homework question, huh?

What I remember from CS coursework was that tail-end recursion is always writeable as non-recursive, ~~other types of recursion may be, but it's not an absolute rule...~~

As to justifying your answer, you gotta do that, it's your homework assignment ;-)

edited Nov 2 '09 at 17:27

answered Nov 2 '09 at 16:53

Gabriel Magana
3,094 8 16

I stand corrected, see Konrad's answer. – Gabriel Magana Nov 2 '09 at 17:26

add a comment

No - not every recursive function is expressable as a non-recursive relation.

I forget exact examples off the top of my head, but it's generally covered in-depth in a discrete mathematics course at the collegiate level.

EDIT
I was thinking of *recurrence relations* not *recursive functions*. Not all of those have [known] closed-forms.

edited Nov 4 '09 at 13:21

answered Nov 2 '09 at 16:53

warren
12.5k 10 51 85

1 Please provide an example. – Lucas B Nov 2 '09 at 17:02

2 An example would be really interesting, since your claim is incorrect. – Konrad Rudolph Nov 2 '09 at 17:03

I guess you remembered something about replacing recursion with iteration without using additional memory (e.g. stack). Just like tail recursion would be. Right? – Grzegorz Oledzki Nov 2 '09 at 17:09

His claim is correct (not every recursive function is rewriteable as non-recursive), just not enough detail. – Gabriel Magana Nov 2 '09 at 17:12

@gmagana: check your computer science basics. The claim is simply wrong and the formal proof to the contrary is trivial. See my answer for details. – Konrad Rudolph Nov 2 '09 at 17:14

show **1** more comment

---

tazzego, recursion means that a function will call itself whether you like it or not. When people are talking about whether or not things can be done without recursion, they mean this and you cannot say "no, that is not true, because I do not agree with the definition of recursion" as a valid statement.

With that in mind, just about everything else you say is nonsense. The only other thing that you say that is not nonsense is the idea that you cannot imagine programming without a callstack. That is something that had been done for decades until using a callstack became popular. Old versions of FORTRAN lacked a callstack and they worked just fine.

By the way, there exist Turing-complete languages that only implement recursion (e.g. SML) as a means of looping. There also exist Turing-complete languages that only implement iteration as a means of looping (e.g. FORTRAN IV). The Church-Turing thesis proves that anything possible in a recursion-only languages can be done in a non-recursive language and vica-versa by the fact that they both have the property of turing-completeness.

answered May 6 '10 at 10:59

Richard
11

add a comment

All computable functions can be computed by Turing Machines and hence the recursive systems and Turing machines (iterative systems) are equivalent.

answered May 21 '13 at 11:10

JOBBINE
1

add a comment

Here is an iterative algorithm:

```
def howmany(x,y)
  a = {}
  for n in (0..x+y)
    for m in (0..n)
      a[[m,n-m]] = if m==0 or n-m==0 then 1 else a[[m-1,n-m]] + a[[m,n-m-1]] end
    end
  end
  return a[[x,y]]
end
```

answered May 31 '09 at 10:43

Jules
3,744 12 28

add a comment

A question: if as first thing the function makes a copy of itself in a random void memory space, and then instead of calling itself call the copy, is this still recursion?(1) I would say yes.

Is the explicit use of stack a real way to remove recursion? (2) I would say no. Basically, aren't we imitating what happens when we use explicitly recursion? I believe we can't define recursion simply as "a function that call itself", since I see recursion also in the "copy code" (1) and in the "explicit use of stack" (2).

Moreover, I don't see how the CT demonstrates that all recursive algorithms can become iterative. It only seems to say to me that "everything" having the "power" of the Turing machine can express all algorithms this can express. **If** Turing-machine can't recursion, we are sure every recursive algo has its interative translation... The Turing-machine can recursion? According to me, if it can be "implemented" (by any mean), then we can say it has it. Has it? I don't know.

All real CPUs I know can recursion. Honestly, I can't see how to program for real without having a call stack, and I think this is what makes recursion possible first.

Avoiding copying(1) and "imitated stack"(2), have we demonstrated that **every** recursive algo can be, on real machines, be expressed iteratively?! I can't see where we demonstrated it.

answered May 6 '10 at 9:18

tazzego
1

add a comment

Not the answer you're looking for? Browse other questions tagged language-agnostic recursion iteration **or ask your own question.**