Recursion (computer science)

From Wikipedia, the free encyclopedia

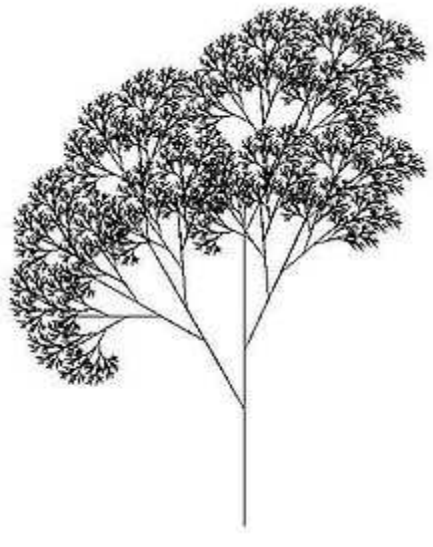
Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem.[1] The approach can be applied to many types of problems, and is one of the central ideas of computer science.[2]

> "The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions." [3]

Most high-level computer programming languages support recursion by allowing a function to call itself within the program text. Some functional programming languages do not define any looping constructs but rely solely on recursion to repeatedly call code. Computability theory has proven that these recursive-only languages are mathematically equivalent to the imperative languages, meaning they can solve the same kinds of problems even without the typical control structures like “while” and “for”.

Contents

- 1 Recursive data types
  - 1.1 Inductively-defined data
  - 1.2 Coinductively-defined data and corecursion
- 2 Recursive algorithms
- 3 Structural versus generative recursion
- 4 Recursive programs
  - 4.1 Recursive procedures
    - 4.1.1 Factorial
    - 4.1.2 Fibonacci
    - 4.1.3 Greatest common divisor
    - 4.1.4 Towers of Hanoi
    - 4.1.5 Binary search
  - 4.2 Recursive data structures (structural recursion)
    - 4.2.1 Linked lists
    - 4.2.2 Binary trees
    - 4.2.3 Filesystem traversal
- 5 Recursion versus iteration
  - 5.1 Expressive power
  - 5.2 Performance issues



Tree created using the Logo programming language and relying heavily on recursion.

- 5.3 Other considerations
- 6 Tail-recursive functions
- 7 Order of function calling
  - 7.1 Function 1
  - 7.2 Function 2 with swapped lines
- 8 Direct and indirect recursion
- 9 See also
- 10 Notes and references
- 11 External links

Recursive data types

Many computer programs must process or generate an arbitrarily large quantity of data. Recursion is one technique for representing data whose exact size the programmer does not know: the programmer can specify this data with a self-referential definition. There are two flavors of self-referential definitions: inductive and coinductive definitions.

Further information: Algebraic data type

Inductively-defined data

Main article: Recursive data type

An inductively-defined recursive data definition is one that specifies how to construct instances of the data. For example, linked lists can be defined inductively (here, using Haskell syntax):

```
data ListOfStrings = EmptyList | Cons String ListOfStrings
```

The code above specifies a list of strings to be either empty, or a structure that contains a string and a list of strings. The self-reference in the definition permits the construction of lists of any (finite) number of strings.

Another example of inductive definition is the natural numbers (or non-negative integers):

```
A natural number is either 1 or n+1, where n is a natural number.
```

Similarly recursive definitions are often used to model the structure of expressions and statements in programming languages. Language designers often express grammars in a syntax such as Backus-Naur form; here is such a grammar, for a simple language of arithmetic expressions with multiplication and addition:

```
<expr> ::= <number>
         | (<expr> * <expr>)
         | (<expr> + <expr>)
```

This says that an expression is either a number, a product of two expressions, or a sum of two expressions. By recursively referring to expressions in the second and third lines, the grammar permits arbitrarily complex arithmetic expressions such as `(5 * ((3 * 6) + 8))`, with more than one product or sum operation in a single expression.

Coinductively-defined data and corecursion

Main article: Coinduction

A coinductive data definition is one that specifies the operations that may be performed on a piece of data; typically, self-referential coinductive definitions are used for data structures of infinite size.

A coinductive definition of infinite streams of strings, given informally, might look like this:

```
A stream of strings is an object s such that
 head(s) is a string, and
 tail(s) is a stream of strings.
```

This is very similar to an inductive definition of lists of strings; the difference is that this definition specifies how to access the contents of the data structure—namely, via the accessor functions `head` and `tail` -- and what those contents may be, whereas the inductive definition specifies how to create the structure and what it may be created from.

Corecursion is related to coinduction, and can be used to compute particular instances of (possibly) infinite objects. As a programming technique, it is used most often in the context of lazy programming languages, and can be preferable to recursion when the desired size or precision of a program's output is unknown. In such cases the program requires both a definition for an infinitely large (or infinitely precise) result, and a mechanism for taking a finite portion of that result. The problem of computing the first n prime numbers is one that can be solved with a corecursive program.

Recursive algorithms

A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those problems, and combine the results. This is often referred to as the divide-and-conquer method; when combined with a lookup table that stores the results of solving sub-problems (to avoid solving them repeatedly and incurring extra computation time), it can be referred to as dynamic programming or memoization.

A recursive function definition has one or more *base cases*, meaning input(s) for which the function produces a result trivially (without recurring), and one or more *recursive cases*,

meaning input(s) for which the program recurs (calls itself). For example, the factorial function can be defined recursively by the equations $0! = 1$ and, for all $n > 0$, $n! = n(n - 1)!$. Neither equation by itself constitutes a complete definition; the first is the base case, and the second is the recursive case.

The job of the recursive cases can be seen as breaking down complex inputs into simpler ones. In a properly-designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached. (Functions that are not intended to terminate under normal circumstances—for example, some system and server processes -- are an exception to this.) Neglecting to write a base case, or testing for it incorrectly, can cause an infinite loop.

For some functions (such as one that computes the series for e = 1+1/1!+1/2!+1/3!...) there is not an obvious base case implied by the input data; for these one may add a parameter (such as the number of terms to be added, in our series example) to provide a 'stopping criterion' that establishes the base case.

Structural versus generative recursion

Some authors classify recursion as either "generative" or "structural". The distinction is related to where a recursive procedure gets the data that it works on, and how it processes that data:

> [Functions that consume structured data] typically decompose their arguments into their immediate structural components and then process those components. If one of the immediate components belongs to the same class of data as the input, the function is recursive. For that reason, we refer to these functions as (STRUCTURALLY) RECURSIVE FUNCTIONS.[4]

Thus the defining characteristic of a structurally recursive function is that the argument to each recursive call is the contents of a field of the original input. Generative recursion is the alternative:

> Many well-known recursive algorithms generate an entirely new piece of data from the given data and recur on it. HTDP (How To Design Programs) refers to this kind as generative recursion. Examples of generative recursion include: gcd, quicksort, binary search, mergesort, Newton's method, fractals, and adaptive integration.[5]

This distinction is important in proving termination of a function. All structurally-recursive functions on finite (inductively-defined) data structures can easily be shown to terminate, via structural induction: intuitively, each recursive call receives a smaller piece of input data, until a base case is reached. Generatively-recursive functions, in contrast, do not necessarily feed smaller input to their recursive calls, so proof of their termination is not necessarily as simple, and avoiding infinite loops requires greater care.

Recursive programs

Recursive procedures

Factorial

A classic example of a recursive procedure is the function used to calculate the factorial of a natural number:

$$\operatorname{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \operatorname{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Pseudocode (recursive):

```
function factorial is:
input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × … × 1]

    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]

end factorial
```

The function can also be written as a recurrence relation:

$$b_n = nb_{n-1}$$
$$b_0 = 1$$

This evaluation of the recurrence relation demonstrates the computation that would be performed in evaluating the pseudocode above:

Computing the recurrence relation for n = 4:

$$\begin{aligned} b_4 &= 4 * b_3 \\ &= 4 * 3 * b_2 \\ &= 4 * 3 * 2 * b_1 \\ &= 4 * 3 * 2 * 1 * b_0 \\ &= 4 * 3 * 2 * 1 * 1 \\ &= 4 * 3 * 2 * 1 \\ &= 4 * 3 * 2 \\ &= 4 * 6 \\ &= 24 \end{aligned}$$

This factorial function can also be described without using recursion by making use of the typical looping constructs found in imperative programming languages:

Pseudocode (iterative):

```
function factorial is:
input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × … × 1]

    1. create new variable called running_total with a value of 1

    2. begin loop
          1. if n is 0, exit loop
          2. set running_total to (running_total × n)
          3. decrement n
          4. repeat loop

    3. return running_total

end factorial
```

The imperative code above is equivalent to this mathematical definition using an accumulator variable *t*:

$$\begin{aligned} \operatorname{fact}(n) &= \operatorname{fact}_{\text{acc}}(n, 1) \\ \operatorname{fact}_{\text{acc}}(n, t) &= \begin{cases} t & \text{if } n = 0 \\ \operatorname{fact}_{\text{acc}}(n-1, nt) & \text{if } n > 0 \end{cases} \end{aligned}$$

The definition above translates straightforwardly to functional programming languages such as Scheme; this is an example of iteration implemented recursively.

Fibonacci

Another well known mathematical recursive function is one that computes the Fibonacci numbers: $\operatorname{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \operatorname{fib}(n-1) + \operatorname{fib}(n-2) & \text{if } n >= 2 \end{cases}$

Pseudocode

```
function fib is:
input: integer n such that n >= 0

    1. if n is 0, return 0
    2. if n is 1, return 1
    3. otherwise, return [ fib(n-1) + fib(n-2) ]

end fib
```

Java language implementation:

```
    /**
     * Recursively calculate the kth Fibonacci number.
     *
     * @param k indicates which (positive) Fibonacci number to compute.
     * @return the kth Fibonacci number.
     */
    private static int fib(int k) {

        // Base Cases:
        //   If k == 0 then fib(k) = 0.
        //   If k == 1 then fib(k) = 1.
        if (k < 2) {
            return k;
        }
        // Recursive Case:
        //   If k >= 2 then fib(k) = fib(k-1) + fib(k-2).
        else {
            return fib(k-1) + fib(k-2);
        }
    }
```

Python language implementation:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Scheme language implementation:

```
(define (fib n)
    (if (or (equal? n 0) (equal? n 1))
        n
        (+ (fib (- n 1)) (fib ( - n 2)))))
```

Recurrence relation for Fibonacci:
$b_n = b_{n-1} + b_{n-2}$
$b_1 = 1, b_0 = 0$

Computing the recurrence relation for n = 4:

```
b4            = b3 + b2
              = b2 + b1 + b1 + b0
              = b1 + b0 + 1 + 1 + 0
              = 1 + 0 + 1 + 1 + 0
              = 3
```

This Fibonacci algorithm is a particularly poor example of recursion, because each time the function is executed on a number greater than one, it makes two function calls to itself, leading to an exponential number of calls (and thus exponential time complexity) in total.[6] The following alternative approach uses two accumulator variables `TwoBack` and `OneBack` to "remember" the previous two Fibonacci numbers constructed, and so avoids the exponential time cost:

Pseudocode

```
function fib is:
input: integer Times such that Times >= 0, relative to TwoBack and OneBack
             long TwoBack such that TwoBack = fib(x)
             long OneBack such that OneBack = fib(x)

    1. if Times is 0, return TwoBack
    2. if Times is 1, return OneBack
    3. if Times is 2, return TwoBack + OneBack
    4. otherwise, return [ fib(Times-1, OneBack, TwoBack + OneBack) ]

end fib
```

To obtain the tenth number in the Fib. sequence, one must perform Fib(10,0,1). Where 0 is considered TwoNumbers back and 1 is considered OneNumber back. As can be seen in this approach, no trees are being created, therefore the efficiency is much greater, being a linear recursion. The recursion in condition 4, shows that OneNumber back becomes TwoNumbers back, and the new OneNumber back is calculated, simply decrementing the Times on each recursion.

Implemented in the Java or the C# programming language:

```
public static long fibonacciOf(int times, long twoNumbersBack, long oneNumberBack) {

    if (times == 0) {                           // Used only for fibonacciOf(0, 0, 1)
        return twoNumbersBack;
    } else if (times == 1) {                    // Used only for fibonacciOf(1, 0, 1)
        return oneNumberBack;
    } else if (times == 2) {                    // When the 0 and 1 clauses are included,
        return oneNumberBack + twoNumbersBack;  // this clause merely stops one additional
    } else {                                    // recursion from occurring
        return fibonacciOf(times - 1, oneNumberBack, oneNumberBack + twoNumbersBack);
    }
}
```

Greatest common divisor

Another famous recursive function is the Euclidean algorithm, used to compute the greatest common divisor of two integers. Function definition*:*

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } x \geq y \text{ and } y > 0 \end{cases}$$

Pseudocode (recursive):

```
function gcd is:
input: integer x, integer y such that x >= y and y >= 0

    1. if y is 0, return x
    2. otherwise, return [ gcd( y, (remainder of x/y) ) ]

end gcd
```

Recurrence relation for greatest common divisor, where *x*%*y* expresses the remainder of *x* / *y*:

$$\gcd(x,y) = \gcd(y,x\%y)$$
$$\gcd(x,0) = x$$

Computing the recurrence relation for x = 27 and y = 9:

```
gcd(27, 9)   = gcd(9, 27 % 9)
             = gcd(9, 0)
             = 9
```

Computing the recurrence relation for x = 259 and y = 111:

```
gcd(259, 111)   = gcd(111, 259 % 111)
                = gcd(111, 37)
                = gcd(37, 0)
                = 37
```

The recursive program above is tail-recursive; it is equivalent to an iterative algorithm, and the computation shown above shows the steps of evaluation that would be performed by a language that eliminates tail calls. Below is a version of the same algorithm using explicit iteration, suitable for a language that does not eliminate tail calls. By maintaining its state entirely in the variables *x* and *y* and using a looping construct, the program avoids making recursive calls and growing the call stack.

Pseudocode (iterative):

```
function gcd is:
input: integer x, integer y such that x >= y and y >= 0

    1. create new variable called remainder

    2. begin loop
          1. if y is zero, exit loop
          2. set remainder to the remainder of x/y
          3. set x to y
          4. set y to remainder
          5. repeat loop

    3. return x

end gcd
```

The iterative algorithm requires a temporary variable, and even given knowledge of the Euclidean algorithm it is more difficult to understand the process by simple inspection, although the two algorithms are very similar in their steps.

Towers of Hanoi

Main article: Towers of Hanoi

For a full discussion of this problem's description, history and solution see the main article or one of the many references.[7][8] Simply put the problem is this: given three pegs, one with a set of N disks of increasing size, determine the minimum (optimal) number of steps it takes to move all the disks from their initial position to another peg without placing a larger disk on top of a smaller one.

Function definition:

$$\text{hanoi}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot \text{hanoi}(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Recurrence relation for hanoi:

$h_n = 2h_{n-1} + 1$
$h_1 = 1$

Computing the recurrence relation for n = 4:

```
hanoi(4)     = 2*hanoi(3) + 1
             = 2*(2*hanoi(2) + 1) + 1
             = 2*(2*(2*hanoi(1) + 1) + 1) + 1
             = 2*(2*(2*1 + 1) + 1) + 1
             = 2*(2*(3) + 1) + 1
             = 2*(7) + 1
             = 15
```

Example Implementations:

Pseudocode (recursive):

```
function hanoi is:
input: integer n, such that n >= 1

    1. if n is 1 then return 1

    2. return [2 * [call hanoi(n-1)] + 1]

end hanoi
```

Although not all recursive functions have an explicit solution, the Tower of Hanoi sequence can be reduced to an explicit formula.[9]

An explicit formula for Towers of Hanoi:

$h_1 = 1 = 2^1 - 1$
$h_2 = 3 = 2^2 - 1$
$h_3 = 7 = 2^3 - 1$
$h_4 = 15 = 2^4 - 1$
$h_5 = 31 = 2^5 - 1$
$h_6 = 63 = 2^6 - 1$
$h_7 = 127 = 2^7 - 1$

In general:
$h_n = 2^n - 1$, for all $n >= 1$

Binary search

The binary search algorithm is a method of searching an ordered array for a single element by cutting the array in half with each pass. The trick is to pick a midpoint near the center of the array, compare the data at that point with the data being searched and then responding to one of three possible conditions: the data is found at the midpoint, the data at the midpoint is greater than the data being searched for, or the data at the midpoint is less than the data being searched for.

Recursion is used in this algorithm because with each pass a new array is created by cutting the old one in half. The binary search procedure is then called recursively, this time on the new (and smaller) array. Typically the array's size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because it essentially divides the problem domain in half with each pass.

Example implementation of binary search in C:

```
/*
 Call binary_search with proper initial conditions.

 INPUT:
   data is an array of integers SORTED in ASCENDING order,
   toFind is the integer to search for,
   count is the total number of elements in the array

 OUTPUT:
   result of binary_search

*/
int search(int *data, int toFind, int count)
{
   //  Start = 0 (beginning index)
   //  End = count - 1 (top index)
   return binary_search(data, toFind, 0, count-1);
}

/*
  Binary Search Algorithm.

  INPUT:
       data is a array of integers SORTED in ASCENDING order,
       toFind is the integer to search for,
       start is the minimum array index,
       end is the maximum array index
  OUTPUT:
       position of the integer toFind within array data,
       -1 if not found
*/
int binary_search(int *data, int toFind, int start, int end)
{
   //Get the midpoint.
   int mid = start + (end - start)/2;   //Integer division

   //Stop condition.
   if (start > end)
      return -1;
   else if (data[mid] == toFind)        //Found?
      return mid;
   else if (data[mid] > toFind)         //Data is greater than toFind, search lower half
      return binary_search(data, toFind, start, mid-1);
   else                                 //Data is less than toFind, search upper half
      return binary_search(data, toFind, mid+1, end);
}
```

Recursive data structures (structural recursion)

Main article: Recursive data type

An important application of recursion in computer science is in defining dynamic data structures such as Lists and Trees. Recursive data structures can dynamically grow to a theoretically infinite size in response to runtime requirements; in contrast, a static array's size requirements must be set at compile time.

"Recursive algorithms are particularly appropriate when the underlying problem

> or the data to be treated are defined in recursive terms." [10]

The examples in this section illustrate what is known as "structural recursion". This term refers to the fact that the recursive procedures are acting on data that is defined recursively.

> As long as a programmer derives the template from a data definition, functions employ structural recursion. That is, the recursions in a function's body consume some immediate piece of a given compound value.[5]

Linked lists

Below is a simple definition of a linked list node. Notice especially how the node is defined in terms of itself. The "next" element of struct node is a pointer to a struct node.

```
struct node
{
  int n;              // some data
  struct node *next;  // pointer to another struct node
};

// LIST is simply a synonym for struct node * (aka syntactic sugar).
typedef struct node *LIST;
```

Procedures that operate on the LIST data structure can be implemented naturally as a recursive procedure because the data structure it operates on (LIST) is defined recursively. The printList procedure defined below walks down the list until the list is empty (NULL), for each node it prints the data element (an integer). In the C implementation, the list remains unchanged by the printList procedure.

```
void printList(LIST lst)
{
    if (!isEmpty(lst))         // base case
    {
       printf("%d ", lst->n);  // print integer followed by a space
       printList(lst->next);   // recursive call
    }
}
```

Binary trees

> *Main article: Binary tree*

Below is a simple definition for a binary tree node. Like the node for Linked Lists, it is defined in terms of itself (recursively). There are two self-referential pointers - left (pointing to the left sub-tree) and right (pointing to the right sub-tree).

```
struct node
{
  int n;               // some data
  struct node *left;   // pointer to the left subtree
  struct node *right;  // point to the right subtree
};

typedef struct node *TREE; // TREE is a synonym for struct node *
```

Operations on the tree can be implemented using recursion. Note that because there are two self-referencing pointers (left and right), tree operations may require two recursive calls:

```
// Test if t contains i; return 1 if so, 0 if not.
int contains(TREE t, int i) {
        if (isEmpty(t))
                return 0;  // base case
        else if (t->n == i)
                return 1;
        else
                return contains(t->left, i) || contains(t->right, i);
}
```

At most two recursive calls will be made for any given call to *contains* as defined above.

```
// Inorder traversal:
void printTree(TREE t) {
        if (!isEmpty(t)) {            // base case
                printTree(t->left);   // go left
                printf("%d ", t->n);  // print the integer followed by a space
                printTree(t->right);  // go right
        }
}
```

The above example illustrates an in-order traversal of the binary tree. A Binary search tree is a special case of the binary tree where the data elements of each node are in order.

Filesystem traversal

Since the number of files in a filesystem may vary, recursion is the only practical way to traverse and thus enumerate its contents. Traversing a filesystem is very similar to that of tree traversal, therefore the concepts behind tree traversal are applicable to traversing a filesystem. More specifically, the code below would be an example of a preorder traversal of a filesystem.

```
import java.io.*;

public class FileSystem {

        public static void main (String [] args) {
                traverse ();
        }

        /**
         * Obtains the filesystem roots
         * Proceeds with the recurisve filesystem traversal
         */
        private static void traverse () {
                File [] fs = File.listRoots ();
                for (int i = 0; i < fs.length; i++) {
                        if (fs[i].isDirectory () && fs[i].canRead ()) {
                                rtraverse (fs[i]);
                        }
                }
        }

        /**
         * Recursively traverse a given directory
         *
         * @param fd indicates the starting point of traversal
         */
        private static void rtraverse (File fd) {
                File [] fss = fd.listFiles ();

                for (int i = 0; i < fss.length; i++) {
                        System.out.println (fss[i]);
                        if (fss[i].isDirectory () && fss[i].canRead ()) {
                                rtraverse (fss[i]);
                        }
                }
        }

}
```

This code blends the lines, at least somewhat, between recursion and iteration. It is, essentially, a recursive implementation, which is the best way to traverse a filesystem. It is also an example of direct and indirect recursion. "rtraverse" is purely a direct example; "traverse" is the indirect, which calls "rtraverse." This example needs no "base case" scenario due to the fact that there will always be some fixed number of files and/or directories in a given filesystem.

Recursion versus iteration

Expressive power

Most programming languages in use today allow the direct specification of recursive functions and procedures. When such a function is called, the program's runtime environment keeps track of the various instances of the function (often using a call stack, although other

methods may be used). Every recursive function can be transformed into an iterative function by replacing recursive calls with iterative control constructs and simulating the call stack with a stack explicitly managed by the program.[11][12]

Conversely, all iterative functions and procedures that can be evaluated by a computer (see Turing completeness) can be expressed in terms of recursive functions; iterative control constructs such as while loops and do loops routinely are rewritten in recursive form in functional languages.[13][14] However, in practice this rewriting depends on tail call elimination, which is not a feature of all languages. C, Java, and Python are notable mainstream languages in which all function calls, including tail calls, cause stack allocation that would not occur with the use of looping constructs; in these languages, a working iterative program rewritten in recursive form may overflow the call stack.

Performance issues

In languages (such as C and Java) that favor iterative looping constructs, there is usually significant time and space cost associated with recursive programs, due to the overhead required to manage the stack and the relative slowness of function calls; in functional languages, a function call (particularly a tail call) is typically a very fast operation, and the difference is usually less noticeable.

As a concrete example, the difference in performance between recursive and iterative implementations of the "factorial" example above depends highly on the language used. In languages where looping constructs are preferred, the iterative version may be as much as several orders of magnitude faster than the recursive one. In functional languages, the overall time difference of the two implementations may be negligible; in fact, the cost of multiplying the larger numbers first rather than the smaller numbers (which the iterative version given here happens to do) may overwhelm any time saved by choosing iteration.

Other considerations

In some programming languages, the stack space available to a thread is much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms. Consequently, these languages sometimes place a limit on the depth of recursion to avoid stack overflows. (Python is one such language.[15]) Note the caveat below regarding the special case of tail recursion.

There are some types of problems whose solutions are inherently recursive, because of prior state they need to track. One example is tree traversal; others include the Ackermann function, depth-first search, and divide-and-conquer algorithms such as Quicksort. All of these algorithms can be implemented iteratively with the help of an explicit stack, but the programmer effort involved in managing the stack, and the complexity of the resulting program, arguably outweigh any advantages of the iterative solution.

Tail-recursive functions

Main article: Tail recursion

Tail-recursive functions are functions in which all recursive calls are tail calls and hence do not build up any deferred operations. For example, the gcd function (shown again below) is tail-recursive. In contrast, the factorial function (also below) is **not** tail-recursive; because its recursive call is not in tail position, it builds up deferred multiplication operations that must be performed after the final recursive call completes. With a compiler or interpreter that treats tail-recursive calls as jumps rather than function calls, a tail-recursive function such as gcd will execute using constant space. Thus the program is essentially iterative, equivalent to using imperative language control structures like the "for" and "while" loops.

Tail recursion:	Augmenting recursion:

`//INPUT: Integers x, y such that x >= y and y > 0`
`int gcd(int x, int y)`
`{`
`  if (y == 0)`
`  return x;`
`  else`
`  return gcd(y, x % y);`
`}`
	`//INPUT: n is an Integer such that n >= 1`
	`int fact(int n)`
	`{`
	`  if (n == 1)`
	`  return 1;`
	`  else`
	`  return n * fact(n - 1);`
	`}`

The significance of tail recursion is that when making a tail-recursive call, the caller's return position need not be saved on the call stack; when the recursive call returns, it will branch directly on the previously saved return position. Therefore, on compilers that support tail-recursion optimization, tail recursion saves both space and time.

Order of function calling

The order of calling a function may change the execution of a function, see this example in C language:

Function 1

```
void recursiveFunction(int num) {
   printf("%d\n", num);
   if (num < 4)
      recursiveFunction(num + 1);
}
```

RecursiveFunction1 execution.png

Function 2 with swapped lines

```
void recursiveFunction(int num) {
   if (num < 4)
      recursiveFunction(num + 1);
   printf("%d\n", num);
}
```

RecursiveFunction2 execution.png

Direct and indirect recursion

Direct recursion is when a function calls itself. Indirect recursion is when (for example) function A calls function B, function B calls function C, and then function C calls function A. Long chains and branches are possible, see Recursive descent parser.

See also

- Ackermann function
- Anonymous recursion
- Corecursion
- Functional programming
- Kleene–Rosser paradox
- McCarthy 91 function
- Memoization
- Mutual recursion
- μ-recursive function
- Primitive recursive function
- Recursion
- Sierpiński curve

Notes and references

1. **^** Graham, Ronald; Donald Knuth, Oren Patashnik (1990). *Concrete Mathematics* (http://www-cs-faculty.stanford.edu/~knuth/gkp.html) . Chapter 1: Recurrent Problems. http://www-cs-faculty.stanford.edu/~knuth/gkp.html.
2. **^** Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). p. 427.
3. **^** Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 126.
4. **^** Felleisen, Matthias; Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi (2001). *How to Design Programs: An Introduction to Computing and Programming* (http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-31.html) . Cambridge, MASS: MIT Press. p. art V "Generative Recursion". http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-31.html.
5. ^ ***a b*** Felleisen, Matthias (2002). "Developing Interactive Web Programs". In Jeuring, Johan. *Advanced Functional Programming: 4th International School*. Oxford, UK: Springer. p. 108
6. **^** Abelson, Harold; Gerald Jay Sussman (1996). *Structure and Interpretation of Computer Programs* (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#%_sec_1.2.2) . Section 1.2.2. http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#%_sec_1.2.2.
7. **^** Graham, Ronald; Donald Knuth, Oren Patashnik (1990). *Concrete Mathematics* (http://www-cs-faculty.stanford.edu/~knuth/gkp.html) . Chapter 1, Section 1.1: The Tower of Hanoi. http://www-cs-faculty.stanford.edu/~knuth/gkp.html.
8. **^** Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). pp. 427–430: The Tower of Hanoi.
9. **^** Epp, Susanna (1995). *Discrete*

Mathematics with Applications (2nd ed.). pp. 447–448: An Explicit Formula for the Tower of Hanoi Sequence.
10. **^** Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 127.
11. **^** http://www.saasblogs.com/2006/09/15/how-to-rewrite-standard-recursion-through-a-state-stack-amp-iteration/
12. **^** http://www.refactoring.com/catalog/replaceRecursionWithIteration.html
13. **^** http://www.ccs.neu.edu/home/shivers/papers/loop.pdf
14. **^** http://lambda-the-ultimate.org/node/1014
15. **^** http://docs.python.org/library/sys.html

External links

- Harold Abelson and Gerald Sussman: "Structure and Interpretation Of Computer Programs" (http://mitpress.mit.edu/sicp/full-text/book/book.html)
- IBM DeveloperWorks: "Mastering Recursive Programming" (http://www-128.ibm.com/developerworks/linux/library/l-recurs.html)
- David S. Touretzky: "Common Lisp: A Gentle Introduction to Symbolic Computation" (http://www.cs.cmu.edu/~dst/LispBook/)
- Matthias Felleisen: "How To Design Programs: An Introduction to Computing and Programming" (http://www.htdp.org/2003-09-26/Book/)
- Duke University: "Big-Oh for Recursive Functions: Recurrence Relations" (http://www.cs.duke.edu/~ola/ap/recurrence.html)

Retrieved from "http://en.wikipedia.org/wiki/Recursion_(computer_science)"
Categories: Theoretical computer science | Recursion | Computability theory | Articles with example pseudocode | Programming idioms | Subroutines

- This page was last modified on 18 February 2011 at 02:23.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of Use for details.
  Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.