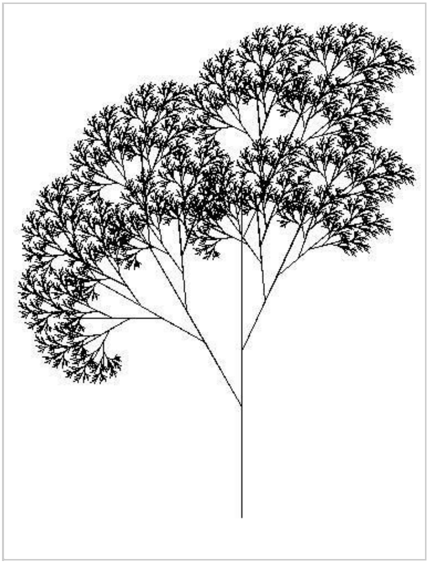
Recursion (computer science)

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 (as opposed to iteration).[1] The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.[2]


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

> "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 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 proves that these recursive-only languages are Turing complete; they are as computationally powerful as Turing complete imperative languages, meaning they can solve the same kinds of problems as imperative languages even without iterative control structures such as “while” and “for”.

Contents

Recursive functions and algorithms

A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-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. Because the base case breaks the chain of recursion, it is sometimes also called the "terminating 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 $e = 1/0! + 1/1! + 1/2! + 1/3! + ...$) 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. Such an example is more naturally treated by co-recursion, where successive terms in the output are the partial sums; this can be converted to a recursion by using the indexing parameter to say "compute the *n*th term (*n*th partial sum)".

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 types of self-referential definitions: inductive and coinductive definitions.

Inductively defined data

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):

```haskell
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 positive 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

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 (e.g. here).

Types of recursion

Single recursion and multiple recursion

Recursion that only contains a single self-reference is known as **single recursion**, while recursion that contains multiple self-references is known as **multiple recursion**. Standard examples of single recursion include list traversal, such as in a linear search, or computing the factorial function, while standard examples of multiple recursion include tree traversal, such as in a depth-first search.

Single recursion is often much more efficient than multiple recursion, and can generally be replaced by an iterative computation, running in linear time and requiring constant space. Multiple recursion, by contrast, may require exponential time and space, and is more fundamentally recursive, not being able to be replaced by iteration without an explicit stack.

Multiple recursion can sometimes be converted to single recursion (and, if desired, thence to iteration). For example, while computing the Fibonacci sequence naively is multiple iteration, as each value requires two previous values, it can be computed by single recursion by passing two successive values as parameters. This is more naturally framed as corecursion, building up from the initial values, tracking at each step two successive values – see corecursion: examples. A more sophisticated example is using a threaded binary tree, which allows iterative tree traversal, rather than multiple recursion.

Indirect recursion

Most basic examples of recursion, and most of the examples presented here, demonstrate ***direct* recursion**, in which a function calls itself. *Indirect* recursion occurs when a function is called not by itself but by another function that it called (either directly or indirectly). For example, if *f* calls *f,* that is direct recursion, but if *f* calls *g* which calls *f,* then that is indirect recursion of *f.* Chains of three or more functions are possible; for example, function 1 calls function 2, function 2 calls function 3, and function 3 calls function 1 again.

Indirect recursion is also called mutual recursion, which is a more symmetric term, though this is simply a difference of emphasis, not a different notion. That is, if *f* calls *g* and then *g* calls *f,* which in turn calls *g* again, from the point of view of *f* alone, *f* is indirectly recursing, while from the point of view of *g* alone, it is indirectly recursing, while from the point of view of both, *f* and *g* are mutually recursing on each other. Similarly a set of three or more functions that call each other can be called a set of mutually recursive functions.

Anonymous recursion

Recursion is usually done by explicitly calling a function by name. However, recursion can also be done via implicitly calling a function based on the current context, which is particularly useful for anonymous functions, and is known as anonymous recursion.

Structural versus generative recursion

Some authors classify recursion as either "structural" or "generative". 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 content of a field of the original input. Structural recursion includes nearly all tree traversals, including XML processing, binary tree creation and search, etc. By considering the algebraic structure of the natural numbers (that is, a natural number is either zero or the successor of a natural number), functions such as factorial may also be regarded as structural recursion.

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. These generatively recursive functions can often be interpreted as corecursive functions – each step generates the new data, such as successive approximation in Newton's method – and terminating this corecursion requires that the data eventually satisfy some condition, which is not necessarily guaranteed.
- In terms of loop variants, structural recursion is when there is an obvious loop variant, namely size or complexity, which starts off finite and decreases at each recursive step.
- By contrast, generative recursion is when there is not such an obvious loop variant, and termination depends on a function, such as "error of approximation" that does not necessarily decrease to zero, and thus termination is not guaranteed without further analysis.

Recursive programs

Recursive procedures

Factorial

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

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

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:

```
b4           = 4 * b3
             = 4 * (3 * b2)
             = 4 * (3 * (2 * b1))
             = 4 * (3 * (2 * (1 * b0)))
             = 4 * (3 * (2 * (1 * 1)))
             = 4 * (3 * (2 * 1))
             = 4 * (3 * 2)
             = 4 * 6
             = 24
```

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:*

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

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

Greatest common divisor

The Euclidean algorithm, which computes the greatest common divisor of two integers, can be written recursively.

Function definition*:*

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

Pseudocode (recursive):

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

$$\operatorname{gcd}(x,y) = \operatorname{gcd}(y, x\%y) \text{ if } y \neq 0$$
$$\operatorname{gcd}(x,0) = x$$

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 = 111 and y = 259:

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

The Towers of Hanoi is a mathematical puzzle whose solution illustrates recursion.[6][7] There are three pegs which can hold stacks of disks of different diameters. A larger disk may never be stacked on top of a smaller. Starting with *n* disks on one peg, they must be moved to another peg one at a time. What is the smallest number of steps to move the stack?

Towers of Hanoi

Function definition:

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

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.[8]

An explicit formula for Towers of Hanoi:

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

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

Binary search

The binary search algorithm is a method of searching a sorted array for a single element by cutting the array in half with each recursive 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)

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, the size of a static array 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."[9]

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 C definition of a linked list node structure. Notice especially how the node is defined in terms of itself. The "next" element of *struct node* is a pointer to another *struct node*, effectively creating a list type.

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

Because the *struct node* data structure is defined recursively, procedures that operate on it can be implemented naturally as recursive procedures. The *list_print* procedure defined below walks down the list until the list is empty (i.e., the list pointer has a value of NULL). For each node it prints the data element (an integer). In the C implementation, the list remains unchanged by the *list_print* procedure.

```
void list_print(struct node *list)
{
    if (list != NULL)               // base case
    {
       printf ("%d ", list->data);  // print integer data followed by a space
       list_print (list->next);     // recursive call on the next node
    }
}
```

Binary trees

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

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 tree_node contains i; return 1 if so, 0 if not.
int tree_contains(struct node *tree_node, int i) {
    if (tree_node == NULL)
        return 0;  // base case
    else if (tree_node->data == i)
        return 1;
    else
        return tree_contains(tree_node->left, i) || tree_contains(tree_node->right, i);
}
```

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

```
// Inorder traversal:
void tree_print(struct node *tree_node) {
        if (tree_node != NULL) {                  // base case
                tree_print(tree_node->left);      // go left
                printf("%d ", tree_node->data);   // print the integer followed by a space
                tree_print(tree_node->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 recursive 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. The method "rtraverse" is purely a direct example; the method "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 or directories in a given filesystem.

Implementation issues

In actual implementation, rather than a pure recursive function (single check for base case, otherwise recursive step), a number of modifications may be made, for purposes of clarity or efficiency. These include:

- Wrapper function (at top)
- Short-circuiting the base case, aka "Arm's-length recursion" (at bottom)
- Hybrid algorithm (at bottom) – switching to a different algorithm once data is small enough

On the basis of elegance, wrapper functions are generally approved, while short-circuiting the base case is frowned upon, particularly in academia. Hybrid algorithms are often used for efficiency, to reduce the overhead of recursion in small cases, and arm's-length recursion is a special case of this.

Wrapper function

A wrapper function is a function that is directly called but does not recurse itself, instead calling a separate auxiliary function which actually does the recursion.

Wrapper functions can be used to validate parameters (so the recursive function can skip these), perform initialization (allocate memory, initialize variables), particularly for auxiliary variables such as "level of recursion" or partial computations for memoization, and handle exceptions and errors. In languages that support nested functions, the auxiliary function can be nested inside the wrapper function and use a shared scope. In the absence of nested functions, auxiliary functions are instead a separate function, if possible private (as they are not called directly), and information is shared with the wrapper function by using pass-by-reference.

Short-circuiting the base case

Short-circuiting the base case, also known as **arm's-length recursion**, consists of checking the base case *before* making a recursive call – i.e., checking if the next call will be the base case, instead of calling and then checking for the base case. Short-circuiting is particularly done for efficiency reasons, to avoid the overhead of a function call that immediately returns. Note that since the base case has already been checked for (immediately before the recursive step), it does not need to be checked for separately, but one does need to use a wrapper function for the case when the overall recursion starts with the base case itself. For example, in the factorial function, properly the base case is 0! = 1, while immediately returning 1 for 1! is a short-circuit, and may miss 0; this can be mitigated by a wrapper function.

Short-circuiting is primarily a concern when many base cases are encountered, such as Null pointers in a tree, which can be linear in the number of function calls, hence significant savings for $O(n)$ algorithms; this is illustrated below for a depth-first search. Short-circuiting on a tree corresponds to considering a leaf (non-empty node with no children) as the base case, rather than considering an empty node as the base case. If there is only a single base case, such as in computing the factorial, short-circuiting provides only $O(1)$ savings.

Conceptually, short-circuiting can be considered to either have the same base case and recursive step, only checking the base case before the recursion, or it can be considered to have a different base case (one step removed from standard base case) and a more complex recursive step, namely "check valid then recurse", as in considering leaf nodes rather than Null nodes as base cases in a tree. Because short-circuiting has a more complicated flow, compared with the clear separation of base case and recursive step in standard recursion, it is often considered poor style, particularly in academia.

Depth-first search

A basic example of short-circuiting is given in depth-first search (DFS) of a binary tree; see binary trees section for standard recursive discussion.

The standard recursive algorithm for a DFS is:

- base case: If current node is Null, return false
- recursive step: otherwise, check value of current node, return true if match, otherwise recurse on children

In short-circuiting, this is instead:

- check value of current node, return true if match,
- otherwise, on children, if not Null, then recurse.

In terms of the standard steps, this moves the base case check *before* the recursive step. Alternatively, these can be considered a different form of base case and recursive step, respectively. Note that this requires a wrapper function to handle the case when the tree itself is empty (root node is Null).

In the case of a perfect binary tree of height *h,* there are $2^{h+1}-1$ nodes and 2^{h+1} Null pointers as children (2 for each of the 2^h leaves), so short-circuiting cuts the number of function calls in half in the worst case.

In C, the standard recursive algorithm may be implemented as:

```
bool tree_contains(struct node *tree_node, int i) {
    if (tree_node == NULL)
        return false;  // base case
    else if (tree_node->data == i)
        return true;
    else
        return tree_contains(tree_node->left, i) ||
               tree_contains(tree_node->right, i);
}
```

The short-circuited algorithm may be implemented as:

```
// Wrapper function to handle empty tree
bool tree_contains(struct node *tree_node, int i) {
    if (tree_node == NULL)
        return false;  // empty tree
    else
        return tree_contains_do(tree_node, i);  // call auxiliary function
}

// Assumes tree_node != NULL
bool tree_contains_do(struct node *tree_node, int i) {
    if (tree_node->data == i)
        return true;  // found
    else  // recurse
        return (tree_node->left  && tree_contains_do(tree_node->left,  i)) ||
               (tree_node->right && tree_contains_do(tree_node->right, i));
}
```

Note the use of short-circuit evaluation of the Boolean && (AND) operators, so that the recursive call is only made if the node is valid (non-Null). Note that while the first term in the AND is a pointer to a node, the second term is a bool, so the overall expression evaluates to a bool. This is a common idiom in recursive short-circuiting. This is in addition to the short-circuit evaluation of the Boolean || (OR) operator, to only check the right child if the left child fails. In fact, the entire control flow of these functions can be replaced with a single Boolean expression in a return statement, but legibility suffers at no benefit to efficiency.

Hybrid algorithm

Recursive algorithms are often inefficient for small data, due to the overhead of repeated function calls and returns. For this reason efficient implementations of recursive algorithms often start with the recursive algorithm, but then switch to a different algorithm when the input becomes small. An important example is merge sort, which is often implemented by switching to the non-recursive insertion sort when the data is sufficiently small, as in the tiled merge sort. Hybrid recursive algorithms can often be further refined, as in Timsort, derived from a hybrid merge sort/insertion sort.

Recursion versus iteration

Recursion and iteration are equally expressive: recursion can be replaced by iteration with an explicit stack, while iteration can be replaced with tail recursion. Which approach is preferable depends on the problem under consideration and the language used. In imperative programming, iteration is preferred, particularly for simple recursion, as it avoids the overhead of function calls and call stack management, but recursion is generally used for multiple recursion. By contrast, in functional languages recursion is preferred, with tail recursion optimization leading to little overhead, and sometimes explicit iteration is not available.

Compare the templates to compute x_n defined by $x_n = f(n, x_{n-1})$ from x_{base}:

```
function recursive(n)
    if n==base
        return x_base
    else
        return f(n, recursive(n-1))
```

```
function iterative(n)
    x = x_base
    for i = n downto base
        x = f(i, x)
    return x
```

For imperative language the overhead is to define the function, for functional language the overhead is to define the accumulator variable x.

For example, the factorial function may be implemented iteratively in C by assigning to an loop index variable and accumulator variable, rather than passing arguments and returning values by recursion:

```c
unsigned int factorial(unsigned int n) {
  unsigned int product = 1; // empty product is 1
  while (n) {
    product *= n;
    --n;
  }
  return product;
}
```

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.[10][11]

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 are routinely rewritten in recursive form in functional languages.[12][13] 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, may 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, although tail call elimination may be a feature that is not covered by a language's specification, and different implementations of the same language may differ in tail call elimination capabilities.

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 compiler 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.

Stack space

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.[14] Note the caveat below regarding the special case of tail recursion.

Multiply recursive problems

Multiply recursive problems are inherently recursive, because of prior state they need to track. One example is tree traversal as in depth-first search; contrast with list traversal and linear search in a list, which is singly recursive and thus naturally iterative. Other examples include divide-and-conquer algorithms such as Quicksort, and functions such as the Ackermann function. 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

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:

```
//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);
}
```

Augmenting recursion:

```
//INPUT: n is an Integer such that n >= 0
int fact(int n)
{
  if (n == 0)
     return 1;
  else
     return n * fact(n - 1);
}
```

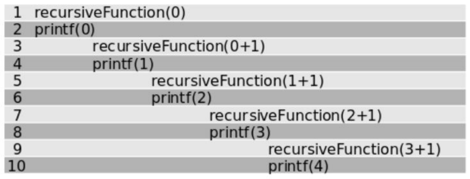
The significance of tail recursion is that when making a tail-recursive call (or any tail call), the caller's return position need not be saved on the call stack; when the recursive call returns, it will branch directly on the previously saved return position. Therefore, in languages that recognize this property of tail calls, tail recursion saves both space and time.

Order of execution

In the simple case of a function calling itself only once, instructions placed before the recursive call are executed once per recursion before any of the instructions placed after the recursive call. The latter are executed repeatedly after the maximum recursion has been reached. Consider this example:

Function 1

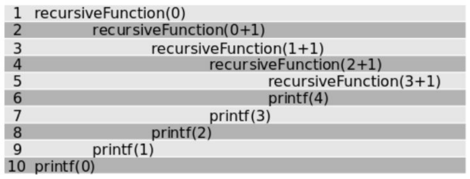
```
void recursiveFunction(int num) {
   printf("%d\n", num);
   if (num < 4)
      recursiveFunction(num + 1);
}
```



```
1  recursiveFunction(0)
2  printf(0)
3          recursiveFunction(0+1)
4          printf(1)
5                  recursiveFunction(1+1)
6                  printf(2)
7                          recursiveFunction(2+1)
8                          printf(3)
9                                  recursiveFunction(3+1)
10                                 printf(4)
```

Function 2 with swapped lines

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



```
1  recursiveFunction(0)
2          recursiveFunction(0+1)
3                  recursiveFunction(1+1)
4                          recursiveFunction(2+1)
5                                  recursiveFunction(3+1)
6                                  printf(4)
7                          printf(3)
8                  printf(2)
9          printf(1)
10 printf(0)
```

Time-efficiency of recursive algorithms

The time efficiency of recursive algorithms can be expressed in a recurrence relation of Big O notation. They can (usually) then be simplified into a single

Big-Oh term.

Shortcut rule (master theorem)

If the time-complexity of the function is in the form

$T(n) = a \cdot T(n/b) + f(n)$

Then the Big-Oh of the time-complexity is thus:

- If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

where a represents the number of recursive calls at each level of recursion, b represents by what factor smaller the input is for the next level of recursion (i.e. the number of pieces you divide the problem into), and $f(n)$ represents the work the function does independent of any recursion (e.g. partitioning, recombining) at each level of recursion.

See also

- Functional programming
- Hierarchical and recursive queries in SQL
- Kleene–Rosser paradox
- Open recursion
- Recursion
- Sierpiński curve

Recursive functions

- McCarthy 91 function
- µ-recursive functions
- Primitive recursive functions
- Tak (function)

Books

- Structure and Interpretation of Computer Programs
- Walls and Mirrors

Notes and references

1. Graham, Ronald; Donald Knuth; Oren Patashnik (1990). *Concrete Mathematics*. Chapter 1: Recurrent Problems.
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*. Cambridge, MASS: MIT Press. p. art V "Generative Recursion".
5. Felleisen, Matthias (2002). "Developing Interactive Web Programs". In Jeuring, Johan. *Advanced Functional Programming: 4th International School*. Oxford, UK: Springer. p. 108.
6. Graham, Ronald; Donald Knuth; Oren Patashnik (1990). *Concrete Mathematics*. Chapter 1, Section 1.1: The Tower of Hanoi.
7. Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). pp. 427–430: The Tower of Hanoi.
8. Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). pp. 447–448: An Explicit Formula for the Tower of Hanoi Sequence.
9. Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 127.
10. Hetland, Magnus Lie (2010), *Python Algorithms: Mastering Basic Algorithms in the Python Language*, Apress, p. 79, ISBN 9781430232384.
11. Drozdek, Adam (2012), *Data Structures and Algorithms in C++* (4th ed.), Cengage Learning, p. 197, ISBN 9781285415017.
12. Shivers, Olin. "The Anatomy of a Loop - A story of scope and control" (PDF). Georgia Institute of Technology. Retrieved 2012-09-03.
13. Lambda the Ultimate. "The Anatomy of a Loop". Lambda the Ultimate. Retrieved 2012-09-03.
14. "27.1. sys — System-specific parameters and functions — Python v2.7.3 documentation". Docs.python.org. Retrieved 2012-09-03.

Further reading

- Dijkstra, Edsger W. (1960). "Recursive Programming". *Numerische Mathematik*. **2** (1): 312–318. doi:10.1007/BF01386232.

External links

- Jonathan Bartlett: "Mastering Recursive Programming" (http://www.ibm.com/developerworks/linux/library/l-recurs/index.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/)
- Owen L. Astrachan: "Big-Oh for Recursive Functions: Recurrence Relations" (http://www.cs.duke.edu/~ola/ap/recurrence.html)