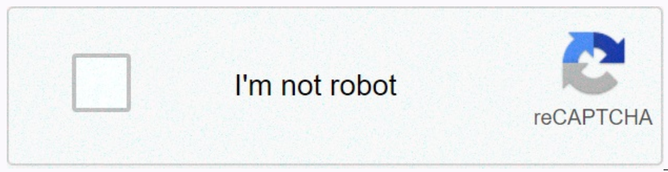

Continue

Recursive vs iterative performance

Using features that are called (on smaller inputs) This article is about recursive approaches to problem solving. For evidence through recursive, see Mathematical induction. For recursions in computer science acronyms, see Recursive acronym § Computer examples. Wood created using logo programming language and relying heavily on recursive. Each branch can be seen as a smaller version of wood. Programming Paradigms Action Agent-Oriented Array Oriented Automated Based Automated Parallel Calculations Reflotitic Programming Relatistic ProgrammingDeclarator(Contrast: Imperative) Functional Functional Logic Pure Functional Logic Response Simultaneous Logic Logic Inductive Restriction Logic Logic Flow-Based Reactive Functional Ontology Differentiated Dynamic/Scripted Based Function-Drive Level (Contrast: Value Level) Style Without Point Concatenative Total Imperative (Contrast: Declarative) Procedural Object Oriented Multimorphic, Language-Oriented, Domain-specific Literate Natural Language Programming Meta Programming Auto Inductive Programming Reflective Attribute Oriented Template Macro Non-Structured (Contrast: Structured) Array Nondeterministic Parallel Process Oriented Process of Probabilistic Quantum Set-theoretic Stack-Based Structures Structured Structured Reconciliation Object Oriented Actor-based Class Based Parallel Prototype-Based By Splitting Worries: Aspect-Oriented Role-Oriented Home Recursive Symbol Value Level (Contrast : Function level) vte In the field of computer science, recursion is a method of solving a problem in which the solution depends on the solutions of smaller cases of the same problem. [1] Such problems can usually be solved by iteration, but this must identify and index smaller instances during programming. Recursion solves such recursive problems using features that are called by their own code. The approach can be applied to many types of problems, and recursion is one of the main ideas of computer science. [2] The power of vomit apparently lies in the ability to determine an infinite set of objects through a final statement. Similarly, an infinite number of calculations can be described by an end-user recursive program, even if this program does not contain explicit repetitions.- Niklaus Wirth, Algorithms + Data Structures = Programs, 1976[3] Most computer programming languages support recursion, allowing a function to be called by its own code. Some functional programming languages (e.g. Clojure)[4] do not define any cyclic structures, but rely solely on recursions to repeat code repeatedly. In the compilation theory, these recursive only have been completed; This means that they are as powerful (they can be used to solve the same problems) as imperative languages based on control structures as time and for. For. calling a function by itself may cause the stack to have a size equal to the sum of the input dimensions of all the calls included. It follows that for problems that can be easily solved by iteration, recursive is generally less effective, and for major problems it is essential to use optimization techniques such as tail optimization. [Reference required] Recursive functions and algorithms Common computer programming tactics is to divide a problem into subproblems of the same type as the original, solve these subproblems, and combine the results. This is often called a method of separation and conquerion; when combined with a search table that stores the results of solving sub-troubled (in order to avoid them being repeatedly solved and to prove to be additional computational time), it can be called dynamic programming or notation. The definition of a recursive function has one or more main cases, which means input(s) for which the function produces a result trivially (without repetitive), and one or more recursive cases, which means

input(s) for which the program is repeated (called). For example, the factorial function can be recursively defined by equations 0! = 1 and for all n > 0, n! = n(n− 1)!. None of the equation in itself constitutes a complete definition; the first is the main case, and the second is a recursive case. Since the main case interrupts the chain of recursion, it is sometimes called the end of the case. The work of recursive cases can be seen as breaking down complex inputs into simpler ones. With a properly designed recursive function, every time a recursive call is made, the input problem must be simplified in such a way that the base base is eventually reached. (Features that are not intended to be terminated under normal circumstances— such as some system and server processes — are an exception.) Ignoring a basic case or improper testing for it can result in infinitely cyclical. For some functions (such as one that calculates the series for e = 1/0! + 1/1! + 1/2! + 1/3! + ...) there is no obvious baseline case implied by the input data; for these, a parameter (such as the number of terms that will be added to our example series) can be added to provide a stop criterion that determines the main case. Such an example is treated more naturally than nuclear uncertainty,[how?] where successive terms in production are partial amounts; this can be converted to recursion using the indexing parameter to say the calculation of the nth term (nth partial amount). Types of recursive data Many computer programs have to process or generate any amount of data. Recursion is a data presentation technique the exact size of which is unknown to the programmer: the programmer can determine this data with the self-relevant There are two types of self-relevant definitions: inductive and resinous definitions. More information: Algebraic data type Inductively defined data Main article: Recursive data type the definition of data is the one that determines how to compile copies 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 blank or a structure that contains a string and a list of strings. Self-sending in the definition makes it easy to snatch lists of each (final) number of strings. Another example of inductive definition is natural numbers (or positive integers): A natural number is 1 or n +1, where n is a natural number. Similarly, recursive definitions are often used to model the structure of expressions and expressions in programming languages. Language designers often express grammar in syntax as a backus-Naur form; here is such a grammar, for a simple language of arithmetic expressions with multiplication and addition: <expr> ::= <number> | (<expr> * <expr>) | (<expr> +) This <expr>says that the expression is either a number, a work of two expressions, or an amount of two expressions. By recursive reference to expressions in the second and third rows, grammar allows arbitrarily complex arithmetic expressions such as (5 * ((3 * 6) + 8)), with more than one product or operation with a total amount in a single expression. Coinductively defined data and corecursion Basic products: Coinduction and Corecursion the definition of cohesive data is one that determines the operations that can be performed on some of the data; usually self-relevant succular definitions are used for data structures of infinite size. It may seem that strings are strings that are strings, such as: the head(s) are a string, and the tail(s) are a stream of strings. This is very similar to an inductive definition of string lists; the difference is that this definition determines how to access the content of the data structure - namely through the functions of accessor head and tail - and what that content can be, while inductive definitions determine how to create the structure and what can be created by it. Corecursion is connected to the coin and can be used to calculate specific cases of (possibly) endless objects. As a programming technique, it is most commonly used in the context of lazy programming languages, and may be preferable to recursion when the desired size or precision of the output of the program is unknown. In such cases, the program requires both a definition of an infinite (or infinitely accurate) result and a mechanism for making the final part of that result. The problem with calculating the first n simple numbers is one that can be solved with a corecursive program (for example here). Types of recursion One recursive and multiple recursive recursion recursion, which contains only one self-sending is known as single recursion, while recursion, which contains multiple self-sendings is known as multiple recursion. Examples of a recursion include crossing a list, such as when searching linearly or calculating the factorial function, while standard examples of multiple re-travel include a tree impediment, such as a first search in depth. A recursive is often much more effective than multiple recursions and can usually be replaced by an iterative calculation that works in linear time and requires constant space. In contrast, multiple recursives can require exponential time and space and are more fundamentally recursive, and cannot be replaced by iteration without an explicit bunch. Multiple recursion can sometimes be converted into a recursive (and, if necessary, hence and afterwards). For example, while the calculation of the Fibonacci sequence is a multiple iteration, since each value requires two previous values, it can be calculated by one recursion by passing two consecutive values as parameters. This is more naturally shaped as corecursion, building the original values, tracking each step two consecutive values – see corecursion: examples. The more complex example is the use of a binary tree with a thread, which allows iterative tree sleeper, not repeated recursion. Indirect repetition Main article: Mutual repetition Initiality The main examples of recursions, and most of the examples presented here, demonstrate a direct repetition in which it is called a function. Indirect recursive occurs when a function is called not in itself, but by another function it calls (directly or indirectly). For example, if f calls f, it is a direct repeat, but if f calls g that call f, then this is an indirect recursive 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 symmetrical term, although this is simply a difference in emphasis, not a different concept. That is, if f calls g and then g called f, which in turn calls g again, in terms of only f, f is indirectly repeated, whereas from the point of view of g only it is repeated indirectly, whereas from the point of view of both, f and g repeat to each other. Similarly, a set of three or more features that call each other can be called a set of mutual recursive features. Anonymous vomiting Main article: Anonymous recursive recursive as a rule is carried out by explicitly calling a function by name. However, recursion can be performed by implicitly calling a feature based on the current context, which is especially useful for anonymous features, and is known as anonymous repeats. Structural vs. generative recursive See also: Structural recursion Some authors classify vomit as structural or generative. The difference is related to where recursive receives the data it works and how it processes that data: [Functions that consume structured data] usually decompose decomposing structural components and then processes these components. If one of the immediate components belongs to the same data class as the input, the function is recursive. For this reason, we call these functions (STRUCTURAL) RECURSIVE FUNCTIONS.- Felleisen, Findler, Flatt, and Krishnaurthi, How to Design Programs, 2001[5] Thus, the defining characteristic of a structural recursive function is that the argument for each recursive call is the field content of the original input. Structural recursion includes almost all wood-cleaning, including XML processing, creation of binary wood and search, etc. Generative recursive is an alternative: Many well-known recursive algorithms generate an entirely new piece of data from the given data and repeat themselves. HtDP (How to design programs) refers to this species as a generative recursive. Examples of generative recursiveness include: GCD, quicksort, binary search, mergesort, Newton method, fractals and adaptive integration.— Matthias Felisen, Advanced Functional Programming, 2002[6] This distinction is important for proving termination of function. All structural recursive functions of limited (inductively defined) data structures can be easily proven to be terminated by structural induction: intuitively, each recursive call receives a smaller number of input data until a baseline is reached. Generator recursive functions, by contrast, do not necessarily feed less input for their recursive calls, so the proof of their termination is not necessarily so simple, and avoiding endless cycles requires greater care. These generative recursive functions can often be interpreted as essential functions — each step generates new data, such as the consistent convergence in newton's method — and ending this corecursion requires that the data ultimately meet certain conditions that are not necessarily guaranteed. With regard to cycles, structural recursion is when there is an obvious variant of a line, namely size or complexity, which starts from extreme and decreases at each recursive step. By contrast, generative recursion is when there is no such obvious variant of a cycle and termination depends on a function, such as an approximation error 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 recursive procedure is the function used to calculate the factorial value of a natural number: fact (n) = 1 if n = 0 n ─ fact (n − 1) if n > 0 {\display *operator name {fact} (n)={\begin{cases}1&{\mbox{if}n=0\\cdfrom \name of {fact} (n-1)&{\mbox{if}n>0\\end}} псевдокок (рекурсивно) : функция factorial е: вход: цяло число число such that n > = 0 output: [n × (n-1) × (n-2) × ... × 1] 1. if n is 0, return 1 2. otherwise, × factorial(n-1)] ultimate factorial function The function can also be saved as a recurrence link: b n = n = n n - 1 {\displaystyle b_{n}=nb_{n-1}} b 0 = 1 {\tyle displays b_{0}=1} This recurrence relationship assessment shows the calculations which would have been carried out in the assessment of pseudocode above: Calculation of the recurrence relationship at 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 factor function can also be described without using recursion, using typical cyclical structures found in imperative programming languages: Pseudocode (iterative): factorial function is : input: integer n such that n > = 0 output × × (n-2) × ... × 1. create a new variable called running_total with a value of 1 2. cycle 1. if n is 0, baseline cycle 2. running_total (running_total × n) 3. 4. repeat the loop 3. return running_total the final factorial 1 is equivalent to this mathematical definition using a battery variable t: fact (n) = is a in h (n , 1) is h and h in (n), t = t if n = 0 f a c a c a (n − 1 , n t) if n > 0 {\displaystyle {\begin{array}{rcl}\operator name {fact} (n)>gt;operator name {fact_{fact_{acc}} (n,1)\\\operator name {fact_{acc}} } (n,t)&=&{\begin{cases}t&{\mbox{if}n=0\\\operator name {fact_{acc}} (n-1,nt)>gt;; The largest common divider Euclid's algorithm, which calculates the largest total divisor of two integers can be recorded recursively. Function definition: gcd (x , y) = { x if y = 0 gcd (y , remainder (x , y)) if y > 0 {\displaystyle \gcd(x,y)={\begin{cases}x&{\mbox{if }}y=0\\\gcd(y,\operatorname {remainder} (x,y))&{\mbox{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 {\displaystyle x\%y} expresses the remainder of x / y {\displaystyle x/y} : gcd (x , y) = gcd (y , x % y) {\displaystyle \gcd(x,y)=\gcd(y,x\%y)} if y ≠ 0 {\displaystyle yeq 0} gcd (x , 0) = x {\displaystyle \gcd(x ,0)=x} Calculation of repeatability for x = 27 and y = 9:gcd(27, 9) = GCD (9, 27% 9) = GCD (9, 0) = 9 Calculation of the relapse relationship 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 111% recursive; it is equivalent to an iterative algorithm and the above-shown language that eliminates queues. The following is a version of the same algorithm using an explicit iteration suitable for language that does not eliminate queue calls. By maintaining its status entirely in the x and y variables and using a haul, the program avoids making recursive calls and growing a call stack. Pseudocode (iterative): the gcd function is: input: integer x, integer y so that x > = y and y > = 0 1. create a new variable called remnant 2. cycle 1. if y is zero, output cycle 2. the rest of x/y 3. set x to y 4. 5. repeat the loop 3. returns x end GCD Iterative algorithm requires a temporary variable, and even when knowing the euclide algorithm it is more difficult to understand the process through simple verification, although the two algorithms are very similar in their steps. The towers of Hanoi towers of Hanoi The main article: Towers of Hanoi Towers of Hanoi is a mathematical puzzle, the solution of which illustrates vomit. [7] [8] There are three clips that can hold stacks of discs of different diameters. A larger disk can never be stacked on a smaller disk. Starting with n discs on one stake, they need to be moved to another peg one by one. What is the smallest number of steps to move the stack? Function definition: Hanoi (n) = { 1 if n = 1 2 ─ Hanoi (n − 1) + 1 if n > 1 {\display \operator {hanoi} (n)={\begin{cases}1&{\mbox{if}, {}, {1\\2\cd\nfrom operator name {hanoi} (n-1)+1&<7>_{\mbox{if}n>1\\\end{cases}}} Hanoi appearance relationship: h n = 2 h n – 1 + 1 {\displaystyle h_{n}=2h_{n-1}+1} h 1 = 1 {\} dysptius h_{1}=1} Calculation of the recurrence connection for n = 4: Hanoi (4) = 2* Hanoi (3) + 1 = 2 *(2* hanoi(2) + 1 = 2 * (1)) + 1 = 2 * (2 * **** Hanoi (1) + 1) + 1 = 2 * (1) 2*(2*1 + 1) + 1) + 1 = 2*(2*(3) + 1) + 1 = 2 *(7) + 1 = 15 Sample 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] Explicit formula for towers of Hanoi: h1 = 1 = 1 - 1 h2 = 3 = 32 - 1 h3 = 7 = 23 - 1 h4 = 15 = 15 = 24 - 1 1 h5 = 31 = 25 - 1 h6 = 1 h6 = = 26 - 1 h7 = 127 = 27 - 1 In general: hn = 2n - 1, for all n > = 1 Binary search Binary search algorithm is a method of searching for a sorted array for one item by cutting the array in half with each recursive liability. The trick is to select the environment near the center of the array, compare the data at this point with the data being searched, and then meet one of the three possible conditions: the data is located in the middle, the data in the middle is larger than the data sought, or the data in the environment is less than the data sought. Recursion is used in this algorithm, because with each passage a new array is created by cutting the old in half. After is called the binary search procedure this time on the new (and smaller) array. Typically, the array size is adjusted by manipulating the initial and end index. The algorithm shows a logarithmic order of growth because it essentially divides the domain issue in half with each pass. Example, perform a binary search in C: /* Call binary_search with appropriate initial conditions. INPUT: data is an array of integers SORTED IN ASCENDING ORDER, up toFind is the number to be searched, number is the total number of items in the OUTPUT array: result of binary_search */ int search (int *data, int *data, int toFind, int count, int count) { // Start = 0 (initial index) // End = number - 1 (superscript) returns binary_search (data, toFind, 0, number- 1); } /* Binary search algorithm. INPUT: Data is an array of integers SORTED IN ASCENDING ORDER, toFind is the integer for search, start is the minimum array index, the end is the maximum index output array: integer position toFind within the data set, -1 if not found */ int binary_search(int*data, int toFind, int start, int end) { //Get the midt environment = start + (end - start)/2; /// Integer division //Stop condition. if (start > end) return -1; other if (data[average] == ofFind) //Found? return mid; other if (data[environment] > reflected) // Data is greater than toFind, search in lower half return binary_search(data, toFind, start, mid-1); other // Data is less than tofind, search for top half return binary_search(data, toFind, mid+1 , end); } Recursive data structures (structural recursive) Main article: Recursive data type Important application of computer science recursive is in the definition of dynamic data structures such as lists and trees. Recursive data structures can dynamically increase to a theoretically infinite size in response to the time of operation requirements; In contrast, the size of a static array must be set in compilation time. Recursive algorithms are particularly relevant when the underlying problem or data to be treated is defined in recursive terms. [10] The examples in this section illustrate what is known as structural recursion. This term refers to the fact that recursive procedures act on data that are recursively defined. While the developer retrieves the template from a data definition, the features use structural recursion. This means that recursions in the body of a function consume some immediate part of a given constituent value. [6] Related Lists Main Article: The related list below is the C definition of the structure of the linked list node. Note that the node is defined in relation to itself. The next item on the string node is a pointer to another startling node, effectively creating a list type. node { int data; // some node on the page of integer data *next; // pointer to another node of the structure}; As the structure of the data node are defined recursively, the procedures working on it can be applied naturally as a recursive procedure. The list_print, defined below, is displayed down the list while the list is empty (i.e. the directory of the For each node, it prints the data element (integer). In the implementation of the In list, it remains unchanged list_print procedure. void list_print (struct node *list) { if (list != NULL) // main case { printf (% d , list >data); printing integers followed by space list_print (list->after); recursive call to the next node } binary trees Main article: Binary tree below is a simple definition for binary node tree. There are two self-referring pointers: left (pointing to the sub-tree) and right (pointing to the right sub-tree). node { int data; // some data from the integer node *left; // pointer to the support structure node *right; // point to the right sub-section }; Tree operations can be accomplished using recursive. Note that since there are two self-relative pointers (left and right) , the tree may require two recursive calls: // Test if tree_node contains i; 1, if so, 0 if not. tree_contains (struct node *tree_node, int i) {if (tree_node == NULL) return 0; // base case if (tree_node- >data == i) return 1; tree_contains tree_node-> return, i) || tree_contains(tree_node->, i) } A further two recursive calls will be made for each call to tree_contains as defined above. Invalid translation: invalid 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 an integer followed by space tree_print(tree_node->right); go right } The above example illustrates a more orderly binary tree crossing. Binary tree search is a special case of a binary tree where the data elements on each node are in order. File system sleeper Since the number of files in a file system may vary, recursion is the only practical way to cross and thus count its contents. , the code below will be an example of the file system being rediseded. java.io imports.*; Public class FileSystem { public static non-stavins (String [] args) { traverse (); } /** * Gets file system roots * Proceeds with reversible file system traverse */ private static secret tally sleeper () { File [] fs = File.listRoots (); for (int i = 0; i < length of fs.; i++) { if (fs[i]. * * * * *@param fd indicates the starting point of the traverse */ private static void rtraverse (File fd) { File [] fss = fd.listFiles (for); for (int i = 0; i < fss.length; i++) { System.out.println (fss[i]) if (fss[i].isDirectory () >fss[]]] {rtraverse (fss[i]) } } } } } } } } } between recursion and iteration. It is essentially a recursive recursive which is the best way to go through a file system. It is an example of direct and indirect recursion. The rtraverse method is only a direct example; the traverse method is the indirect one it calls rtraverse. This example needs a basic scenario because there will always be a certain number of files or directories in a file system. Implementation issues In real performance rather than purely recursive function (single background check, otherwise recursive step), a number of amendments can be made for clarity or effectiveness. These include: Wrapper function (at the top) Short circuit of the base case, even more Arm's length recursive (at the bottom) Hybrid algorithm (at the bottom) – switching to a different data algorithm is small enough Based on elegance, shell features are usually approved, while the short round of the main case is frowning, especially in academia. Hybrid algorithms are often used for efficiency, for reducing recursive overhead in small cases, and greater recursion in the same place is a special case of this. Wrapper Function Shell function is a function that is called directly but not repeated, instead calling a separate help function that actually does recursion. Shell functions can be used to validate parameters (so that the recursive function skips these), perform initialization (memory allocation, initialization of variables), especially for additional variables such as recursive level or partial calculations for remembering and manipulations and exceptions and errors. In languages that support nested functions, the auxiliary function can be nested inside the shell 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 the information is shared with the shell function using pass-by-reference. Short circuit of the main case Short circuit main case, also known as recursion at your fingertips, consists in checking the main case before making a recursive call - that is, checking whether the next call will be the main case instead of calling and then checking for the main case. A short circuit is made especially for efficiency purposes to avoid the overhead function call that immediately returns. Note that since the main case has already been checked (immediately before the recursive step), it is not necessary to check separately, but a shell function should be used for the case when the overall recursion begins with the main case itself. For example, in the function of factorial function, correctly the main case is 0! = 1, immediately returning 1 for 1! is a short circuit and may miss 0; this can be iced by the shell function. Short circuit is first and foremost a problem when many major cases occur, such as zero in wood, which can be linear in the number of functional calls, therefore significant savings O(n) algorithms; this is illustrated below for the first search. The short tree lick corresponds to viewing the sheet (not an empty node without children) as the main case, rather than treating an empty node as the main case. If there is only one main case, such as when calculating factorial, a short circuit provides only O(1) savings. Conceptually, short circuit can be considered to have the same base base and recursive step, only checking the main case before recursion, or can be considered to have a different baseline case (one step removed from the standard case base) and a more complex recursive step, namely checking valid then recurse, as when examining leaf nodes rather than zero nodes as base cases in wood. Since short circuit has a more complex flow, compared to the clear separation of the main case and recursive step in standard recursion, it is often considered a bad style, especially in academia. [11] The first search is a basic short-circuit search (DFS) of a binary tree; see the binary trees section for a standard recursive discussion. The standard recursive algorithm for DFS is: base case: If the current node is Null, return untruths recursive step: otherwise, check the value of the current node, returning true if match, otherwise recursion of children In short join, this is instead: check value of the current node, return true if match, otherwise, to children, if not Zero, then again. With regard to standard steps, this moves the check of the main case before a recursive step. Alternatively, they can be considered a different form of basic and recursive step, respectively. Note that this requires a shell function to process the case when the tree is empty (the root node is Null). In the case of the ideal two-component trees with a height of h, there are 2h + 1-1 knots and 2h + 1 Zero pointers as children (2 for each of the leaves 2h), so short encirclement reduces the number of functional calls in half at worst. In C, the standard recursive algorithm can be applied as: bool tree_contains(struct node * tree_node, int i) { if (tree_node == NULL) return false; // main case if (tree_node->date == i) return true; another return tree_contains(tree_node->s, i) || tree_contains(tree_node->right, i) The short-circuit algorithm can be implemented as: // Function shell to work with empty tree bool tree_contains(struct node *tree_node, int i) {if (tree_node == NULL) return false; // empty tree other return tree_contains_do(tree_node, i) // call additional function } // Implies tree_node bool tree_contains_do(struct no*tree_node int i) { if (tree_node->data =i) return true; // found another // return of recursive (tree_node-> tree_contains_do(tree_node->listen , i)) || (tree_node-> right to tree_contains_do(tree_node-> right, i)); } Note the assessment of short chains of booleans " (And) operators, so the recursive call is made only if the node is valid (non-Null). Note Note while the first term in I is a pointer to a node, the second is bool, so the general expression is valued on bool. It's a common idiom in recursive short circuit. This is in addition to the assessment of the short circuit of the boolean || (OR) check only the right child if the left child fails. In fact, the entire control flow of these functions can be replaced by one boolean expression in a return declaration, but legibility does not benefit efficiency. Hybrid algorithm Recursive algorithms are often ineffective for small data due to the overhead of repetitive call functions and returns. For this reason, effective implemented algorithms often start with a recursive algorithm, but then switch to another algorithm when the input becomes small. An important example is merging sorting, which is often done by switching to non-recursive insertion when the data is small enough as in subheading merge sorting. Hybrid recursive algorithms can often be further refined, as in Timort, derived from hybrid sorting of mixed sorting/insertion. Recursion against iteration Recursion and iteration are equally expressive: recursion can be replaced by iteration with an explicit chimney, while iteration can be replaced with the recursion tail. Which approach is preferable depends on the problem under consideration and the language used. In imperative programming, iteration is preferable, especially for simple recursive, since it avoids overhead calls and call stack management, but is generally used for repeated repetition. In contrast, in functional languages, recursion is preferred, with tail recursive optimization leading to a little overhead. Applying an algorithm using iteration may not be easily achievable. Compare xn calculation templates defined by xn = f(n, xn-1) from xbase: recursive function (n) if n == base return xbase otherwise return f (n, recursive (n-1)) function iterativ (n) x = xbase for i = base +1 to n x = f (i, x) return x For mandatory language overhead must determine the function, for functional language the overhead is to define the battery variable x. For example, a function of factorial function in C may be applied, attributable variable to the cycle index and battery variable, rather than by submitting arguments and return values by repeating: unsigned driver factors (unsigned int n) { unsigned product = 1; // empty product is 1, while (n) { product *= n; -- n; } returning product; } Expressive power Most programming languages , which are used today, allow direct identification of recursive functions and procedures. When called such a function, the program execution environment monitors the different instances of the function (often using a chimney, although other methods can be used). Any recursive function can be transformed into a repetitive function by replacing recursive with iterative control structures and chimney simulation which is explicitly managed by the programme. [12] [13] Conversely, all iterative functions and procedures that can be evaluated by a computer (see Turing completeness) can be expressed as a recursive function; iterative control constructors such as noems and noems are routinely rewritten in recursive form in functional languages. [14] [15] In practice, however, this rewrite depends on the elimination of the queue, which is not a feature of all languages. C, Java and Python are notable basic languages in which all call functions, including queue calls, can cause stack distribution that will not occur with the use of cyclic constructions; In these languages, a work iterative program rewritten in recursive form may overflow the call stack, although removing the call queue may be a feature that is not covered by the language specification, and different implementations in the same language may differ the call queue elimination capabilities. Performance issues In languages (such as C and Java) that favor iterative cyclic constructions, there are usually significant time and spatial costs associated with recursive programs due to the costs required to manage the stack and the relative delay of the call function; in functional languages, the call function (especially the call queue) is usually a very fast operation, and the difference is usually less noticeable. As a specific example, the difference in performance between recursive and iterative performance of the factorial example above depends to a large extent on the compiler used. In languages where cyclical constructions are preferred, the iteratitic version can be faster than recursive. In functional languages, the overall time difference of both applications may be negligible; in fact, the cost of multiplying larger numbers rather than smaller numbers (which happens to be done with an iterative version given here) can overload any time saved by selecting an iteration. Stack space In some programming languages, the maximum size of the call stack is much less than the space available in free memory and recursive algorithms tend to require more stack space than iterative algorithms. Therefore, these languages sometimes place a limit on the depth of recursion to avoid stack overflow; A python is one such language. [16] Note the warning below regarding the special tail repeat. Vulnerability Due to recursive algorithms can be subjected to stack overflow, they can be vulnerable to pathological or malicious inputs. [17] Some malware specifically targets the call stack in the program and takes advantage of its inherently recursive nature. [18] Even in the absence of malware, stack overflow caused by unrelated recursive can be fatal for the program, the logic for processing exceptions may not prevent the relevant process from being terminated. [19] Multiplying recursive problems Multiply recursive problems are inherently recursive due to you need to track. One example is the tree that intersects as in depth-first search; although a recursive and iterative method is used,[20] they oppose a list crossing and liner in a list that is one-cento-recursive and thus a natural iterative method. Other examples include divide-and-conquer algorithms such as Quicksort, and features such as the Ackerman feature. All these algorithms can be applied iteratively using an explicit stack, but the efforts of a developer involved in stack management, and the complexity of the resulting program, perhaps outweigh all the advantages of an iterative solution. Recursive recursive recursive algorithms can be replaced by non-recursive counterparts. One method of replacing recursive algorithms is to simulate using free memory instead of stack memory. [22] The alternative is to develop a replacement algorithm entirely based on non-recursive methods that can be challenging. [23] For example, recursive wildcard matching algorithms, such as Rich Salz's Wildmat algorithm,[24] were once typical. Algorithms of non-requisition origin for the same purpose, such as the Kraus wildcard matching algorithm, were developed to avoid the shortcomings of recursion[25] and only gradually improved on the basis of techniques such as test collection and profiling. [26] Tail recursive tail functions tail-recursive functions are functions in which all recursive calls are queues and therefore no deferred operations are built. For example, the GCD function (shown again below) is a recursive tail. Unlike factorial function (also under) is not tail-recursive; since its recursive call is not in a queue position, it builds postponed multiplication operations that must be performed after the last recursive call has ended. With a compiler or interpreter that treats queue recursive calls as jumps rather than function calls the queue recursive function such as GCD will perform using permanent space. Thus, the program is essentially iterative, equivalent to the use of imperative structures for language control such as for and time cycles. Queue recursion: Increase recursive: //INPUT: Integers x, y such that x >= y and y >= 0 int gcd(int x, int y) { if (y == 0) returns x; otherwise return gcd(y, x % y) } //INPUT: n is such that n >= 0 int fact(int) { if (n == 0) return 1; return n * fact(n - 1); The meaning of the tail is repeated, that when making a recursive call (or any call), it is not necessary to keep the caller position in the stack; when the recursive call returns, it will branch directly to the pre-recorded return position. Therefore, in languages that recognize this call queue property, the queue vomits saves both space and time. , is called only once, the instructions placed before the recursive call are executed once on a recursive before each of the instructions placed after the recursive call. 1. 1. repeated after reaching the maximum recursion. Consider this example: Feature 1 cancels recursive Function(int num) { printf (%d, num) if (num < 4) recursiveFunction(num +1); } Function 2 with the smet lines cancels the recursive Function(int num) {if (num < 4) recursive Function(num +1); printf (%d, num); } Time efficiency of recursive algorithms The effectiveness of recursive algorithms can be expressed in repressive relationships of Big O notation. They can (usually) be simplified in one big-o term. Правило за пряк път (главна теорем) Основна статия: Главна теореми (анализ на алгоритми) Ако сложността на време-сложността на функцията е във формата T (n) = · T (n / b) + f (n) {\displaystyle T(n)=a\cdot T(n/b)+f(n)} Тогава Голямото О на сложността на времето е така: Ако f (n) = O (n log b за ε) {\displaystyle f(n)=O(n^\log_en_en_000] за някои постоянни ε > 0 {\displaystyle \epsile >0} , след това T (n) = Θ (n log b a) {\ displaystyle T(n)=\Тета (n^{\log_00 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000) (n^ {\log _{b}a}log n)} Ако f (n) = Ω (n log b + ε) {\displaystyle f(n)=\Омега (n^\log _{b}a+\epsilon)} за някои постоянни ε > 0 {\дисплеи >} и ако · f (n / b) ≤ c · f (n) {\ displaystyle a\cdot f(n/b)\leq c\cdot f(n)} за някои константни c < 1 и всички достатъчно големи n , then T (n) = Φ (f (n) {\ displaystyle T(n)=\Theta (f(n))} where a represents the number of recursive calls at each recursive level, b represents by what coefficient the input to the next level of recursion is (i.e. number of pieces, and f (n) represents the work that the function does independently of any recursion (e.g. recombination) at each recursion level. See also Functional programming Hierarchical and recursive queries in SQL Kleene-Roser paradox Open Recursion Recursion Recursion Sierpiński Curve McCarthy 91 function μ-recursive functions Primitive recursive functions Tak (function) Recursion ^ Graham, Ronald; Knut, Donald; Patašnik, Oren (1990). ISBN 0-201-55802-5.CS1 main channel: ref= harv (connection) ^ EPP, Susannah (1995). Discrete mathematics with applications (2nd and e.). 427. ISBN 978-0-53494446-9.CS1 master: ref=harv (connection) ^ Wirth, Niklaus (1976). Algorithms + Data Structures = Programs. - Prentas. 126. ISBN 978-0-13022418-7.CS1 network: ref=harv (connection) ^ Functional programming | Brave for the brave and faithful. www.braveclojure.com. Retrieved 2020-10-21. [1] 2001, 2001, 2001, 2001, item V. Development of interactive web programs. In Zheur, Johan (id.). Advanced functional programming: 4th International School (PDF). Springer. 108. 9783540448334. [1] 1990 was 1990. 1995, p. 1. Jiguer, Eric; 1999 Programming interviews Revealed: Secrets to unloading your next job (3rd ed.). Wiley. 115. 118-26136-1. At the beginning of 2010 is 1000000000000000000000000000000000000000000000020232384. [197] 1999 Anatomy of a Cycle - History of Scope and Control (PDF). The Georgia Institute of Technology. Retrieved 2012-09-03. They're 10000000000000 Anatomy of the Noose. Lambda The Ultimate. Retrieved 2012-09-03. 27.1. system, system-specific parameters and functions — Python v2.7.3 documentation. Docs.python.org. Retrieved 2012-09-03. Kirk J . Wildcard match: an empirical way to tame an algorithm. Dr. Dobb Journal. ↑ Oliver Oliver (2012). The anatomy of a stack-crushing attack and how GCC protects it. Dr. Dobb Journal. 1999 .NET Class Library. Microsoft Developer Network. 2018. ^ Depth of first search (DFS): iterative and recursive execution. 20, 2014, in New Year's Day. 2018. ^ Mitrovic, Ivan. Replace recursion with iteration. Works. 2015 How to replace recursive functions using the stack and during a cycle to avoid stack overflow. Code project. They are at 1000 m. Tricks of trade: Recursion to Iteration, Part 2: Remove Recursion with Time-Traveling Secret Trick. They're in the top 19. wild mat.k. Github. Kirk J . Wildcard match: algorithm. Dr. Dobb Journal. Kirk J . Wildcard Match: Improved big data algorithm. Development for performance. Further, by quoting Barron, David William (1968) [1967]. It's written in Cambridge, UK. Gil, Stanley (ed.). Recursive programming techniques. McDonald's Computer Monographs (1.ed.). London, United Kingdom: Macdonald "" " " (Publishers) EOOD SBN 356-02201-3. (viii+64 pages) Rubio-Sanchez, Manuel (2017). Introduction to recursive programming. Press THE CPA. 1.1.1.351.64717-5. Pevac, Irena (2016). Practicing recursion in Java. Create an independent space. 1.5327-1227-2. Roberts, Eric (2005). I'm thinking recursively with Java. Wiley. 1000000000000000000000000000000146 Rahl, Jeffrey S. (1984). Your recursions. The University of Cambridge press. 10000000000000000000000012999990011999-12. Hellman, Paul; Veroff, Robert. Walls and mirrors. Abelson, Harold; Sussman, Gerald Jay; 1996 Structure and interpretation of computer programs (2nd id.). MIT press. 0-262-51087-1. Deekstra, Edsger B. (1960). Recursive programming. Numerische Mathematician. 2 (1): 312–318. 10.1007/BF01386232. External relations Bartlett, Jonathan. Mastering recursive programming. IBM Developer works. Touretzky, David S. (1990). General lysp: a gentle introduction to the symbolic calculation. 9780805304923. Felisen, Matthias; Find, Robert B. Matthew; Krishnamurthy, Sriram (2001). How to design programs: Introduction to computers and programming. MIT press. ISBN 026206216.CS1 Big-O for recursive functions: Repetition relationships. Visited by