Algorithms
Divide and Conquer

Contents

Chapter 1

Recursion (computer science)

This article is about recursive approaches to solving problems. For proofs by recursion, see Mathematical induction. For recursion in computer science acronyms, see Recursive acronym § Computer-related examples.

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]

> "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".

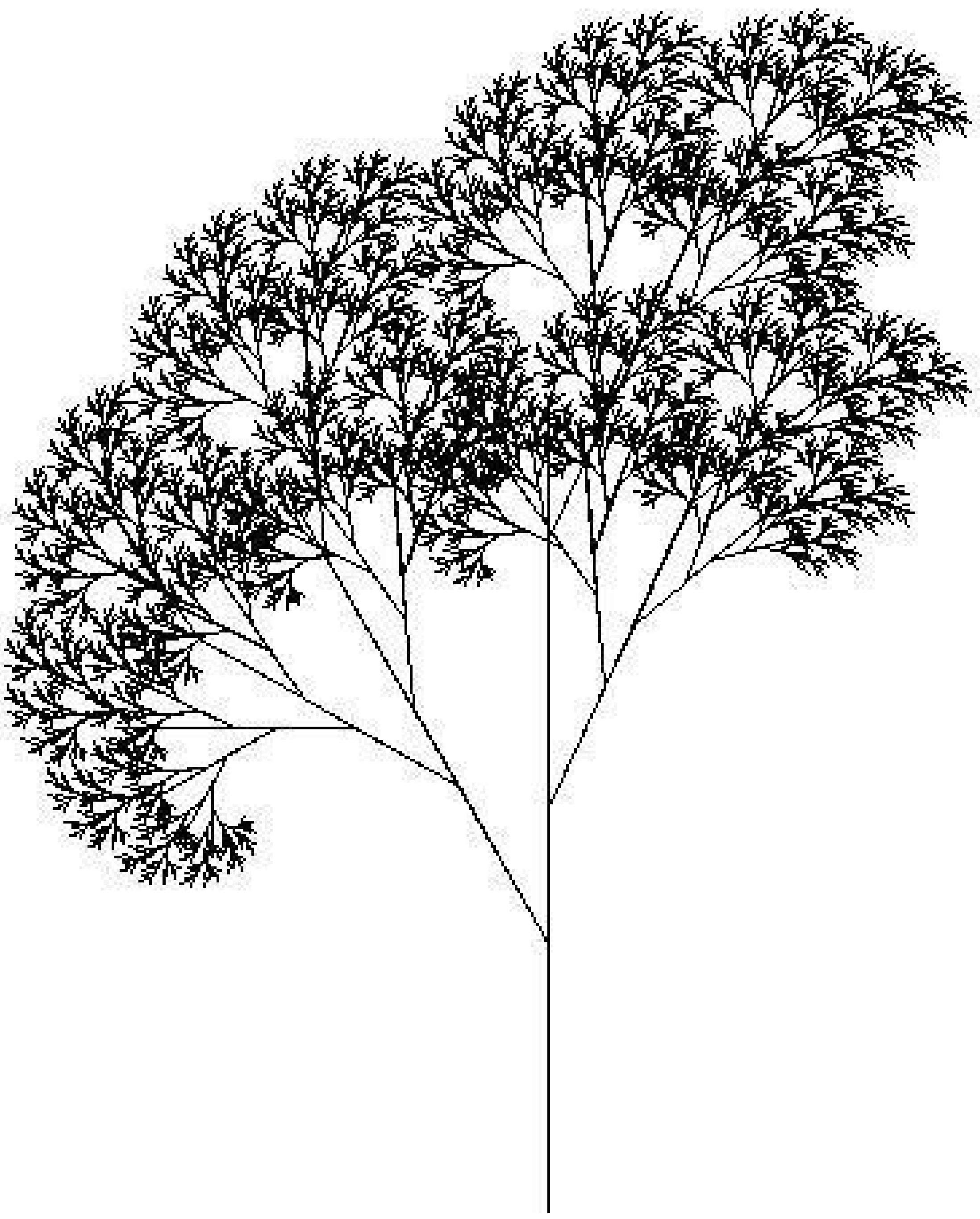
1.1 Recursive functions and algorithms

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)".



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

1.2 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.

Further information: Algebraic data type

1.2.1 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 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.

1.2.2 Coinductively defined data and corecursion

Main articles: Coinduction 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).

1.3 Types of recursion

1.3.1 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.

1.3.2 Indirect recursion

Main article: Mutual 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.

1.3.3 Anonymous recursion

Main article: 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.

1.3.4 Structural versus generative recursion

See also: Structural 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.

1.4 Recursive programs

1.4.1 Recursive procedures

Factorial

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

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

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:

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

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.

Greatest common divisor

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

Function definition:

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

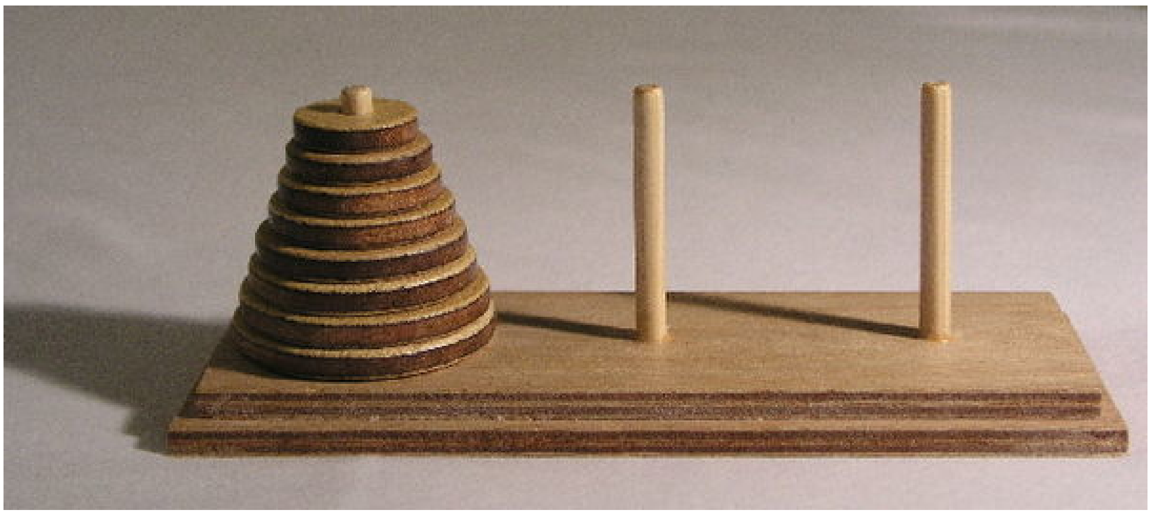
Recurrence relation for greatest common divisor, where $x\%y$ expresses the remainder of x/y :

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

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.

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



Towers of Hanoi

Main article: 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?

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:

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

$h_1 = 1$

Example implementations:

Although not all recursive functions have an explicit solution, the Tower of Hanoi sequence can be reduced to an explicit formula.[8]

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); }

1.4.2 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, 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

Main article: Linked list

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

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 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.

1.5 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.

1.5.1 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.

1.5.2 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.

1.5.3 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.

1.6 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}:

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:

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

1.6.1 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.

1.6.2 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.

1.6.3 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.

1.6.4 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.

1.7 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.

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.

1.8 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:

1.8.1 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)

1.8.2 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)				

1.9 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.

1.9.1 Shortcut rule (master theorem)

Main article: 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.

1.10 See also

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

1.10.1 Recursive functions

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

1.10.2 Books

- Structure and Interpretation of Computer Programs
- Walls and Mirrors

1.11 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.

1.12 Further reading

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

1.13 External links

- Jonathan Bartlett: "Mastering Recursive Programming"
- David S. Touretzky: "Common Lisp: A Gentle Introduction to Symbolic Computation"
- Matthias Felleisen: "How To Design Programs: An Introduction to Computing and Programming"
- Owen L. Astrachan: "Big-Oh for Recursive Functions: Recurrence Relations"

Chapter 2

Divide and conquer algorithm

In computer science, **divide and conquer** (**D&C**) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

Understanding and designing D&C algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved. As when proving a theorem by induction, it is often necessary to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding the proper generalization. These D&C complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion.

The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

2.1 Divide and conquer

The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list (or its analog in numerical computing, the bisection algorithm for root finding).[1] These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide and conquer algorithm". Therefore, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more subproblems.[2] The name **decrease and conquer** has been proposed instead for the single-subproblem class.[3]

An important application of divide and conquer is in optimization, where if the search space is reduced ("pruned") by a constant factor at each step, the overall algorithm has the same asymptotic complexity as the pruning step, with the constant depending on the pruning factor (by summing the geometric series); this is known as prune and search.

2.2 Early historical examples

Early examples of these algorithms are primarily decrease and conquer – the original problem is successively broken down into *single* subproblems, and indeed can be solved iteratively.

Binary search, a decrease and conquer algorithm where the subproblems are of roughly half the original size, has a long history. While a clear description of the algorithm on computers appeared in 1946 in an article by John Mauchly, the idea of using a sorted list of items to facilitate searching dates back at least as far as Babylonia in 200 BC.[4] Another ancient decrease and conquer algorithm is the Euclidean algorithm to compute the greatest common divisor of two numbers by reducing the numbers to smaller and smaller equivalent subproblems, which dates to several centuries BC.

There is no documentary evidence confirming that Carl Friedrich Gauss knew at least one fast method or algorithm, however in [5] is described how to get the DFT from Gauss's formulas, and then on the basis of the Karatsuba idea to obtain what is now called the Cooley-Tukey fast Fourier transform (FFT) algorithm. It should be noted without the addition of other formulas, the formulae of Gauss text does not give the FFT.

An early two-subproblem D&C algorithm that was specifically developed for computers and properly analyzed is the merge sort algorithm, invented by John von Neumann in 1945.[6]

Another notable example is the algorithm invented by Anatolii A. Karatsuba in 1960[7] that could multiply two n-digit numbers in $O(n^{\log_2 3})$ operations (in Big O notation). This algorithm disproved Andrey Kolmogorov's 1956 conjecture that $\Omega(n^2)$ operations would be required for that task.

As another example of a divide and conquer algorithm that did not originally involve computers, Donald Knuth gives the method a post office typically uses to route mail: letters are sorted into separate bags for different geographical areas, each of these bags is itself sorted into batches for smaller sub-regions, and so on until they are delivered.[4] This is related to a radix sort, described for punch-card sorting machines as early as 1929.[4]

2.3 Advantages

2.3.1 Solving difficult problems

Divide and conquer is a powerful tool for solving conceptually difficult problems: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. Similarly, divide and conquer only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height n to moving a tower of height $n - 1$.

2.3.2 Algorithm efficiency

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.

In all these examples, the D&C approach led to an improvement in the asymptotic cost of the solution. For example, if (a) the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size n, and (b) there is a bounded number p of subproblems of size ~ n/p at each stage, then the cost of the divide-and-conquer algorithm will be $O(n \log p n)$.

2.3.3 Parallelism

Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

2.3.4 Memory access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called *cache-oblivious*, because it does not contain the cache size as an explicit parameter.[8] Moreover, D&C algorithms can be designed for important algorithms (e.g., sorting, FFTs, and matrix multiplication) to be *optimal* cache-oblivious algorithms–they use the cache in a probably optimal way, in an asymptotic sense, regardless of the cache size. In contrast, the traditional approach to exploiting the cache is *blocking*, as in loop nest optimization, where the problem is explicitly divided into chunks of the appropriate size—this can also use the cache optimally, but only when the algorithm is tuned for the specific cache size(s) of a particular machine.

The same advantage exists with regards to other hierarchical storage systems, such as NUMA or virtual memory, as well as for multiple levels of cache: once a sub-problem is small enough, it can be solved within a given level of the hierarchy, without accessing the higher (slower) levels.

2.3.5 Roundoff control

In computations with rounded arithmetic, e.g. with floating point numbers, a divide-and-conquer algorithm may yield more accurate results than a superficially equivalent iterative method. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a D&C algorithm called pairwise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.[9]

2.4 Implementation issues

2.4.1 Recursion

Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack. A recursive function is a function that calls itself within its definition.

2.4.2 Explicit stack

Divide and conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue. This approach allows more freedom in the choice of the sub-problem that is to be solved next, a feature that is important in some applications — e.g. in breadth-first recursion and the branch and bound method for function optimization. This approach is also the standard solution in programming languages that do not provide support for recursive procedures.

2.4.3 Stack size

In recursive implementations of D&C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of stack overflow. Fortunately, D&C algorithms that are time-efficient often have relatively small recursion depth. For example, the quicksort algorithm can be implemented so that it never requires more than $\log_2 n$ nested recursive calls to sort n items.

Stack overflow may be difficult to avoid when using recursive procedures, since many compilers assume that the recursion stack is a contiguous area of memory, and some allocate a fixed amount of space for it. Compilers may also save more information in the recursion stack than is strictly necessary, such as return address, unchanging parameters, and the internal variables of the procedure. Thus, the risk of stack overflow can be reduced by minimizing the parameters and internal variables of the recursive procedure, or by using an explicit stack structure.

2.4.4 Choosing the base cases

In any recursive algorithm, there is considerable freedom in the choice of the *base cases*, the small subproblems that are solved directly in order to terminate the recursion.

Choosing the smallest or simplest possible base cases is more elegant and usually leads to simpler programs, because there are fewer cases to consider and they are easier to solve. For example, an FFT algorithm could stop the recursion when the input is a single sample, and the quicksort list-sorting algorithm could stop when the input is the empty list; in both examples there is only one base case to consider, and it requires no processing.

On the other hand, efficiency often improves if the recursion is stopped at relatively large base cases, and these are solved non-recursively, resulting in a hybrid algorithm. This strategy avoids the overhead of recursive calls that do little or no work, and may also allow the use of specialized non-recursive algorithms that, for those base cases, are more efficient than explicit recursion. A general procedure for a simple hybrid recursive algorithm is *short-circuiting the base case,* also known as *arm's-length recursion.* In this case whether the next step will result in the base case is checked before the function call, avoiding an unnecessary function call. For example, in a tree, rather than recursing to a child node and then checking if it is null, checking null before recursing; this avoids half the function calls in some algorithms on binary trees. Since a D&C algorithm eventually reduces each problem or sub-problem instance to a large number of base instances, these often dominate the overall cost of the algorithm, especially when the splitting/joining overhead is low. Note that these considerations do not depend on whether recursion is implemented by the compiler or by an explicit stack.

Thus, for example, many library implementations of quicksort will switch to a simple loop-based insertion sort (or similar) algorithm once the number of items to be sorted is sufficiently small. Note that, if the empty list were the only base case, sorting a list with *n* entries would entail maximally *n* quicksort calls that would do nothing but return immediately. Increasing the base cases to lists of size 2 or less will eliminate most of those do-nothing calls, and more generally a base case larger than 2 is typically used to reduce the fraction of time spent in function-call overhead or stack manipulation.

Alternatively, one can employ large base cases that still use a divide-and-conquer algorithm, but implement the algorithm for predetermined set of fixed sizes where the algorithm can be completely unrolled into code that has no recursion, loops, or conditionals (related to the technique of partial evaluation). For example, this approach is used in some efficient FFT implementations, where the base cases are unrolled implementations of divide-and-conquer FFT algorithms for a set of fixed sizes.[10] Source code generation methods may be used to produce the large number of separate base cases desirable to implement this strategy efficiently.[10]

The generalized version of this idea is known as recursion "unrolling" or "coarsening" and various techniques have been proposed for automating the procedure of enlarging the base case.[11]

2.4.5 Sharing repeated subproblems

For some problems, the branched recursion may end up evaluating the same sub-problem many times over. In such cases it may be worth identifying and saving the solutions to these overlapping subproblems, a technique commonly known as memoization. Followed to the limit, it leads to bottom-up divide-and-conquer algorithms such as dynamic programming and chart parsing.

2.5 See also

- Akra–Bazzi method
- Fork–join model
- Master theorem
- Mathematical induction
- MapReduce
- Heuristic (computer science)

2.6 References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (MIT Press, 2000).

[2] Brassard, G. and Bratley, P. Fundamental of Algorithmics, Prentice-Hall, 1996.

[3] Anany V. Levitin, *Introduction to the Design and Analysis of Algorithms* (Addison Wesley, 2002).

[4] Donald E. Knuth, *The Art of Computer Programming: Volume 3, Sorting and Searching*, second edition (Addison-Wesley, 1998).

[5] Heideman, M. T., D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform," IEEE ASSP Magazine, 1, (4), 14–21 (1984)

[6] Knuth, Donald (1998). *The Art of Computer Programming: Volume 3 Sorting and Searching*. p. 159. ISBN 0-201-89685-0.

[7] Karatsuba, Anatolii A.; Yuri P. Ofman (1962). "Умножение многозначных чисел на автоматах". *Doklady Akademii Nauk SSSR*. **146**: 293–294. Translated in “Multiplication of Multidigit Numbers on Automata”. *Physics-Doklady*. **7**: 595–596. 1963.

[8] M. Frigo; C. E. Leiserson; H. Prokop (1999). “Cache-oblivious algorithms”. *Proc. 40th Symp. on the Foundations of Computer Science*.

[9] Nicholas J. Higham, “The accuracy of floating point summation”, *SIAM J. Scientific Computing* **14** (4), 783–799 (1993).

[10] Frigo, M.; Johnson, S. G. (February 2005). “The design and implementation of FFTW3” (PDF). *Proceedings of the IEEE*. **93** (2): 216–231. doi:10.1109/JPROC.2004.840301.

[11] Radu Rugina and Martin Rinard, "Recursion unrolling for divide and conquer programs,” in *Languages and Compilers for Parallel Computing*, chapter 3, pp. 34–48. *Lecture Notes in Computer Science* vol. 2017 (Berlin: Springer, 2001).

2.7 External links

Chapter 3

Karatsuba algorithm

The **Karatsuba algorithm** is a fast multiplication algorithm. It was discovered by Anatoly Karatsuba in 1960 and published in 1962.[1][2][3] It reduces the multiplication of two n-digit numbers to at most $n^{\log_2 3} \approx n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3}$ when n is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. For example, the Karatsuba algorithm requires 3^{10} = 59,049 single-digit multiplications to multiply two 1024-digit numbers (n = 1024 = 2^{10}), whereas the classical algorithm requires $(2^{10})^2$ = 1,048,576.

The Karatsuba algorithm was the first multiplication algorithm asymptotically faster than the quadratic “grade school” algorithm. The Toom–Cook algorithm is a faster generalization of Karatsuba’s method, and the Schönhage–Strassen algorithm is even faster, for sufficiently large n.

3.1 History

The standard procedure for multiplication of two n-digit numbers requires a number of elementary operations proportional to n^2 , or $\Theta(n^2)$ in the big-O notation. Andrey Kolmogorov conjectured that the classical algorithm was *asymptotically optimal,* meaning that any algorithm for that task would require $\Omega(n^2)$ elementary operations.

In 1960, Kolmogorov organized a seminar on mathematical problems in cybernetics at the Moscow State University, where he stated the $\Omega(n^2)$ conjecture and other problems in the complexity of computation. Within a week, Karatsuba, then a 23-year-old student, found an algorithm (later it was called “divide and conquer”) that multiplies two n-digit numbers in $\Theta(n^{\log_2 3})$ elementary steps, thus disproving the conjecture. Kolmogorov was very agitated about the discovery; he communicated it at the next meeting of the seminar, which was then terminated. Kolmogorov did some lectures on the Karatsuba result at the conferences all over the world (see, for example, “Proceedings of the international congress of mathematicians 1962”, pp. 351–356, and also “6 Lectures delivered at the International Congress of Mathematicians in Stockholm, 1962”) and published the method in 1962, in the Proceedings of the USSR Academy of Sciences. The article had been written by Kolmogorov and contained two results on multiplication, Karatsuba’s algorithm and a separate result by Yuri Ofman; it listed “A. Karatsuba and Yu. Ofman” as the authors. Karatsuba only became aware of the paper when he received the reprints from the publisher.[2]

3.2 Algorithm

3.2.1 Basic step

The basic step of Karatsuba’s algorithm is a formula that allows one to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y , plus some additions and digit shifts.

Let x and y be represented as n -digit strings in some base B . For any positive integer m less than n , one can write the two given numbers as

$x = x_1 B^m + x_0$ $y = y_1 B^m + y_0$, where x_0 and y_0 are less than B^m . The product is then

$xy = (x_1B^m + x_0)(y_1B^m + y_0)$ $xy = z_2B^{2m} + z_1B^m + z_0$,

where

$z_2 = x_1y_1$ $z_1 = x_1y_0 + x_0y_1$ $z_0 = x_0y_0$.

These formulae require four multiplications, and were known to Charles Babbage.[4] Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z_0 and z_2 as before one can calculate

$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$

which holds since

$z_1 = x_1y_0 + x_0y_1$ $z_1 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0$.

A more efficient implementation of Karatsuba multiplication can be set as [5] $xy = (b^2 + b)x_1y_1 - b(x_1 - x_0)(y_1 - y_0) + (b + 1)x_0y_0$, where $b = B^m$.

3.2.2 Example

To compute the product of 12345 and 6789, choose $B = 10$ and $m = 3$. Then we decompose the input operands using the resulting base (B^m = *1000*), as:

12345 = **12** · *1000* + **345**

6789 = **6** · *1000* + **789**

Only three multiplications, which operate on smaller integers, are used to compute three partial results:

z_2 = **12 × 6** = 72

z_0 = **345 × 789** = 272205

z_1 = (**12** + **345**) × (**6** + **789**) − z_2 − z_0 = 357 × 795 − 72 − 272205 = 283815 − 72 − 272205 = 11538

We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base *1000* like for the input operands):

result = $z_2 \cdot B^{2m} + z_1 \cdot B^m + z_0$, i.e.

result = 72 · 1000^2 + 11538 · *1000* + 272205 = **83810205**.

Note that the intermediate third multiplication operates on an input domain which is less than two times larger than for the two first multiplications, its output domain is less than four times larger, and base-*1000* carries computed from the first two multiplications must be taken into account when computing these two subtractions; but note also that this partial result z_1 cannot be negative: to compute these subtractions, equivalent additions using complements to 1000^2 can also be used, keeping only the two least significant base-*1000* digits for each number:

z_1 = 283815 − 72 − 272205 = (283815 + 999928 + 727795) **mod** 1000^2 = 2011538 **mod** 1000^2 = 11538.

3.2.3 Recursive application

If n is four or more, the three multiplications in Karatsuba's basic step involve operands with fewer than n digits. Therefore, those products can be computed by recursive calls of the Karatsuba algorithm. The recursion can be applied until the numbers are so small that they can (or must) be computed directly.

In a computer with a full 32-bit by 32-bit multiplier, for example, one could choose $B = 2^{31}$ = 2,147,483,648, and store each digit as a separate 32-bit binary word. Then the sums $x_1 + x_0$ and $y_1 + y_0$ will not need an extra binary word for storing the carry-over digit (as in carry-save adder), and the Karatsuba recursion can be applied until the numbers to multiply are only one digit long.

3.2.4 Asymmetric Karatsuba-like formulae

Karatsuba's original formula and other generalizations are themselves symmetric. For example, the following formula computes

$$c(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 = a(x)b(x) = (a_2x^2 + a_1x + a_0)(b_2x^2 + b_1x + b_0)$$

with 6 multiplications in $\mathrm{GF}(2)[x]$, where $\mathrm{GF}(2)$ is the Galois field with two elements 0 and 1.

$$\begin{cases} c_0 = p_0, \\ c_1 = p_{012} + p_{02} + p_{12} + p_2, \\ c_2 = p_{012} + p_{01} + p_{12}, \\ c_3 = p_{012} + p_{01} + p_{02} + p_0, \\ c_4 = p_2, \end{cases}$$

where $p_i = a_ib_i,\ \ p_{ij} = (a_i + a_j)(b_i + b_j)$ and $p_{ijk} = (a_i + a_j + a_k)(b_i + b_j + b_k)$. We note that addition and subtraction are the same in fields of characteristic 2.

This formula is symmetrical, namely, it does not change if we exchange a and b in $p_i,\ \ p_{ij}$ and p_{ijk} .

Based on the second Generalized division algorithms ,[6] Fan et al. found the following asymmetric formula:

$$\begin{cases} c_0 = p_0 \\ c_1 = p_{012} + p_2 + m_4 + m_5 \\ c_2 = p_{012} + m_3 + m_5 \\ c_3 = p_{012} + p_0 + m_3 + m_4 \\ c_4 = p_2, \end{cases}$$

where $m_3 = (a_1 + a_2)(b_0 + b_2),\ \ m_4 = (a_0 + a_1)(b_1 + b_2)$ and $m_5 = (a_0 + a_2)(b_0 + b_1)$.

It is asymmetric because we can obtain the following new formula by exchanging a and b in $m_3,\ \ m_4$ and m_5 .

$$\begin{cases} c_0 = p_0 \\ c_1 = p_{012} + p_2 + m_4 + m_5 \\ c_2 = p_{012} + m_3 + m_5 \\ c_3 = p_{012} + p_0 + m_3 + m_4 \\ c_4 = p_2, \end{cases}$$

where $m_3 = (a_0 + a_2)(b_1 + b_2),\ \ m_4 = (a_1 + a_2)(b_0 + b_1)$ and $m_5 = (a_0 + a_1)(b_0 + b_2)$.

3.3 Efficiency analysis

Karatsuba's basic step works for any base *B* and any *m*, but the recursive algorithm is most efficient when *m* is equal to *n*/2, rounded up. In particular, if *n* is 2^k, for some integer *k*, and the recursion stops only when *n* is 1, then the number of single-digit multiplications is 3^k, which is n^c where $c = \log_2 3$.

Since one can extend any inputs with zero digits until their length is a power of two, it follows that the number of elementary multiplications, for any *n*, is at most $3^{\lceil \log_2 n \rceil} \leq 3n^{\log_2 3}$.

Since the additions, subtractions, and digit shifts (multiplications by powers of *B*) in Karatsuba's basic step take time proportional to *n*, their cost becomes negligible as *n* increases. More precisely, if *t*(*n*) denotes the total number of elementary operations that the algorithm performs when multiplying two *n*-digit numbers, then

$$T(n) = 3T(\lceil n/2 \rceil) + cn + d$$

for some constants *c* and *d*. For this recurrence relation, the master theorem gives the asymptotic bound $T(n) = \Theta(n^{\log_2 3})$.

It follows that, for sufficiently large n, Karatsuba's algorithm will perform fewer shifts and single-digit additions than longhand multiplication, even though its basic step uses more additions and shifts than the straightforward formula. For small values of n, however, the extra shift and add operations may make it run slower than the longhand method. The point of positive return depends on the computer platform and context. As a rule of thumb, Karatsuba is usually faster when the multiplicands are longer than 320–640 bits.[7]

3.4 Pseudocode

procedure karatsuba(num1, num2) if (num1 < 10) or (num2 < 10) return num1*num2 /* calculates the size of the numbers */ m = max(size_base10(num1), size_base10(num2)) m2 = m/2 /* split the digit sequences about the middle */ high1, low1 = split_at(num1, m2) high2, low2 = split_at(num2, m2) /* 3 calls made to numbers approximately half the size */ z0 = karatsuba(low1,low2) z1 = karatsuba((low1+high1),(low2+high2)) z2 = karatsuba(high1,high2) return (z2*10^(2*m2))+((z1-z2-z0)*10^(m2))+(z0)

3.5 References

[1] A. Karatsuba and Yu. Ofman (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". *Proceedings of the USSR Academy of Sciences*. **145**: 293–294. Translation in the academic journal *Physics-Doklady*, **7** (1963), pp. 595–596

[2] A. A. Karatsuba (1995). "The Complexity of Computations" (PDF). *Proceedings of the Steklov Institute of Mathematics*. **211**: 169–183. Translation from Trudy Mat. Inst. Steklova, 211, 186–202 (1995)

[3] Knuth D.E. (1969) *The Art of Computer Programming. v.2.* Addison-Wesley Publ.Co., 724 pp.

[4] Charles Babbage, Chapter VIII – Of the Analytical Engine, Larger Numbers Treated, Passages from the Life of a Philosopher, Longman Green, London, 1864; page 125.

[5] Torbjörn Granlund and the GMP development team, *The GNU Multiple Precision Arithmetic Library Manual, version 6.0.0*, Free Software Foundation, Inc., March 2014.

[6] Haining Fan, Ming Gu, Jiaguang Sun, Kwok-Yan Lam,"Obtaining More Karatsuba-Like Formulae over the Binary Field", IET Information security Vol. 6 No. 1 pp. 14-19, 2012.

[7]

3.6 External links

- Karatsuba's Algorithm for Polynomial Multiplication
- Weisstein, Eric W. "Karatsuba Multiplication". *MathWorld*.
- Bernstein, D. J., "Multidigit multiplication for mathematicians". Covers Karatsuba and many other multiplication algorithms.

Chapter 4

Merge sort

In computer science, **merge sort** (also commonly spelled **mergesort**) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.[1] A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.[2]

4.1 Algorithm

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).

2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

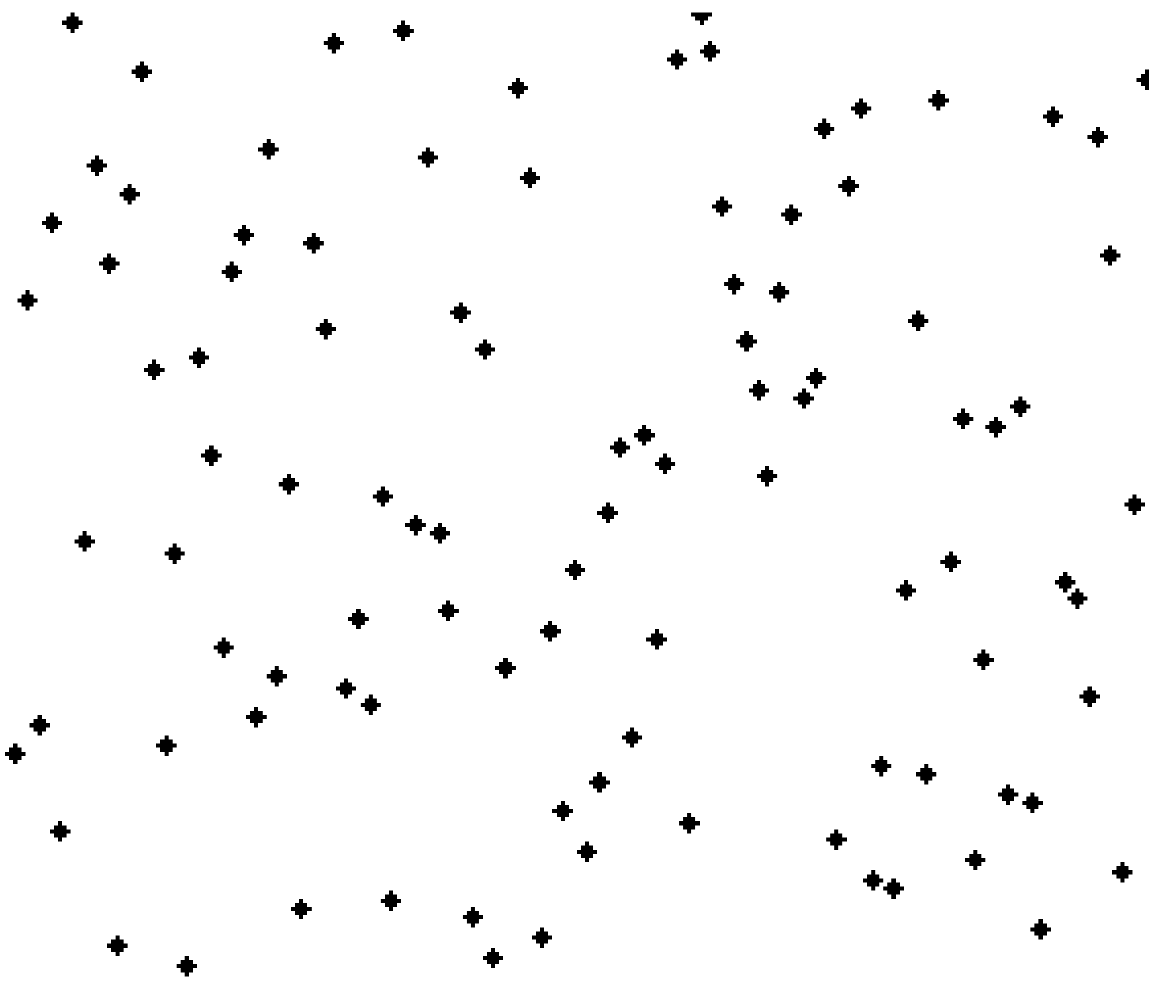
4.1.1 Top-down implementation

Example C-like code using indices for top down merge sort algorithm that recursively splits the list (called *runs* in this example) into sublists until sublist size is 1, then merges those sublists to produce a sorted list. The copy back step is avoided with alternating the direction of the merge with each level of recursion.

// Array A[] has the items to sort; array B[] is a work array. TopDownMergeSort(A[], B[], n) { CopyArray(A, 0, n, B); // duplicate array A[] into B[] TopDownSplitMerge(B, 0, n, A); // sort data from B[] into A[] } // Sort the given run of array A[] using array B[] as a source. // iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set). TopDownSplitMerge(B[], iBegin, iEnd, A[]) { if(iEnd - iBegin < 2) // if run size == 1 return; // consider it sorted // split the run longer than 1 item into halves iMiddle = (iEnd + iBegin) / 2; // iMiddle = mid point // recursively sort both runs from array A[] into B[] TopDownSplitMerge(A, iBegin, iMiddle, B); // sort the left run TopDownSplitMerge(A, iMiddle, iEnd, B); // sort the right run // merge the resulting runs from array B[] into A[] TopDownMerge(B, iBegin, iMiddle, iEnd, A); } // Left source half is A[iBegin:iMiddle-1]. // Right source half is A[iMiddle:iEnd-1]. // Result is B[iBegin:iEnd-1]. TopDownMerge(A[], iBegin, iMiddle, iEnd, B[]) { i = iBegin, j = iMiddle; // While there are elements in the left or right runs... for (k = iBegin; k < iEnd; k++) { // If left run head exists and is <= existing right run head. if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) { B[k] = A[i]; i = i + 1; } else { B[k] = A[j]; j = j + 1; } } } CopyArray(A[], iBegin, iEnd, B[]) { for(k = iBegin; k < iEnd; k++) B[k] = A[k]; }

4.1.2 Bottom-up implementation

Example C-like code using indices for bottom up merge sort algorithm which treats the list as an array of *n* sublists (called *runs* in this example) of size 1, and iteratively merges sub-lists back and forth between two buffers:



Merge sort animation. The elements to sort are represented by dots.

// array A[] has the items to sort; array B[] is a work array void BottomUpMergeSort(A[], B[], n) { // Each 1-element run in A is already "sorted". // Make successively longer sorted runs of length 2, 4, 8, 16... until whole array is sorted. for (width = 1; width < n; width = 2 * width) { // Array A is full of runs of length width. for (i = 0; i < n; i = i + 2 * width) { // Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[] // or copy A[i:n-1] to B[] (if(i+width >= n)) BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B); } // Now work array B is full of runs of length 2*width. // Copy array B to array A for next iteration. // A more efficient implementation would swap the roles of A and B. CopyArray(B, A, n); // Now array A is full of runs of length 2*width. } } // Left run is A[iLeft :iRight-1]. // Right run is A[iRight:iEnd-1]. BottomUpMerge(A[], iLeft, iRight, iEnd, B[]) { i = iLeft, j = iRight; // While there are elements in the left or right runs... for (k = iLeft; k < iEnd; k++) { // If left run head exists and is <= existing right run head. if (i < iRight && (j >= iEnd || A[i] <= A[j])) { B[k] = A[i]; i = i + 1; } else { B[k] = A[j]; j = j + 1; } } } void CopyArray(B[], A[], n) { for(i = 0; i < n; i++) A[i] = B[i]; }

4.1.3 Top-down implementation using lists

Pseudocode for top down merge sort algorithm which recursively divides the input list into smaller sublists until the sublists are trivially sorted, and then merges the sublists while returning up the call chain.

function merge_sort(*list* m) // *Base case. A list of zero or one elements is sorted, by definition.* **if** length of m ≤ 1 **then return** m // *Recursive case. First, divide the list into equal-sized sublists // consisting of the first half and second half of the list. // This assumes lists start at index 0.* **var** left := empty list **var** right := empty list **for each** x **with index** i **in** m **do if** i < (length of m)/2 **then** add x to left **else** add x to right // *Recursively sort both sublists.* left := merge_sort(left) right := merge_sort(right) // Then merge the now-sorted sublists. **return** merge(left, right)

In this example, the merge function merges the left and right sublists.

function merge(left, right) **var** result := empty list **while** left is not empty **and** right is not empty **do if** first(left) ≤ first(right) **then** append first(left) to result left := rest(left) **else** append first(right) to result right := rest(right) // *Either left or right may have elements left; consume them.* // *(Only one of the following loops will actually be entered.)* **while** left is not empty **do** append first(left) to result left := rest(left) **while** right is not empty **do** append first(right) to result right := rest(right) **return** result

4.1.4 Bottom-up implementation using lists

Pseudocode for bottom up merge sort algorithm which uses a small fixed size array of references to nodes, where array[i] is either a reference to a list of size 2^{i} or 0. *node* is a reference or pointer to a node. The merge() function would be similar to the one shown in the top down merge lists example, it merges two already sorted lists, and handles empty lists. In this case, merge() would use *node* for its input parameters and return value.

function merge_sort(*node* head) // return if empty list **if** (head == nil) **return** nil **var** *node* array[32]; initially all nil **var** *node* result **var** *node* next **var** *int* i result = head // merge nodes into array **while** (result != nil) next = result.next; result.next = nil **for**(i = 0; (i < 32) && (array[i] != nil); i += 1) result = merge(array[i], result) array[i] = nil // do not go past end of array **if** (i == 32) i -= 1 array[i] = result result = next // merge array into single list result = nil **for** (i = 0; i < 32; i += 1) result = merge(array[i], result) **return** result

4.2 Natural merge sort

A natural merge sort is similar to a bottom up merge sort except that any naturally occurring runs (sorted sequences) in the input are exploited. Both monotonic and bitonic (alternating up/down) runs may be exploited, with lists (or equivalently tapes or files) being convenient data structures (used as FIFO queues or LIFO stacks).[3] In the bottom up merge sort, the starting point assumes each run is one item long. In practice, random input data will have many short runs that just happen to be sorted. In the typical case, the natural merge sort may not need as many passes because there are fewer runs to merge. In the best case, the input is already sorted (i.e., is one run), so the natural merge sort need only make one pass through the data. In many practical cases, long natural runs are present, and for that reason natural merge sort is exploited as the key component of Timsort. Example:

Start : 3-–4-–2-–1-–7-–5-–8-–9-–0-–6 Select runs : 3-–4 2 1-–7 5-–8-–9 0-–6 Merge : 2-–3-–4 1-–5-–7-–8-–9 0-–6 Merge : 1-–2-–3-–4-–5-–7-–8-–9 0-–6 Merge : 0-–1-–2-–3-–4-–5-–6-–7-–8-–9

Tournament replacement selection sorts are used to gather the initial runs for external sorting algorithms.

4.3 Analysis

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

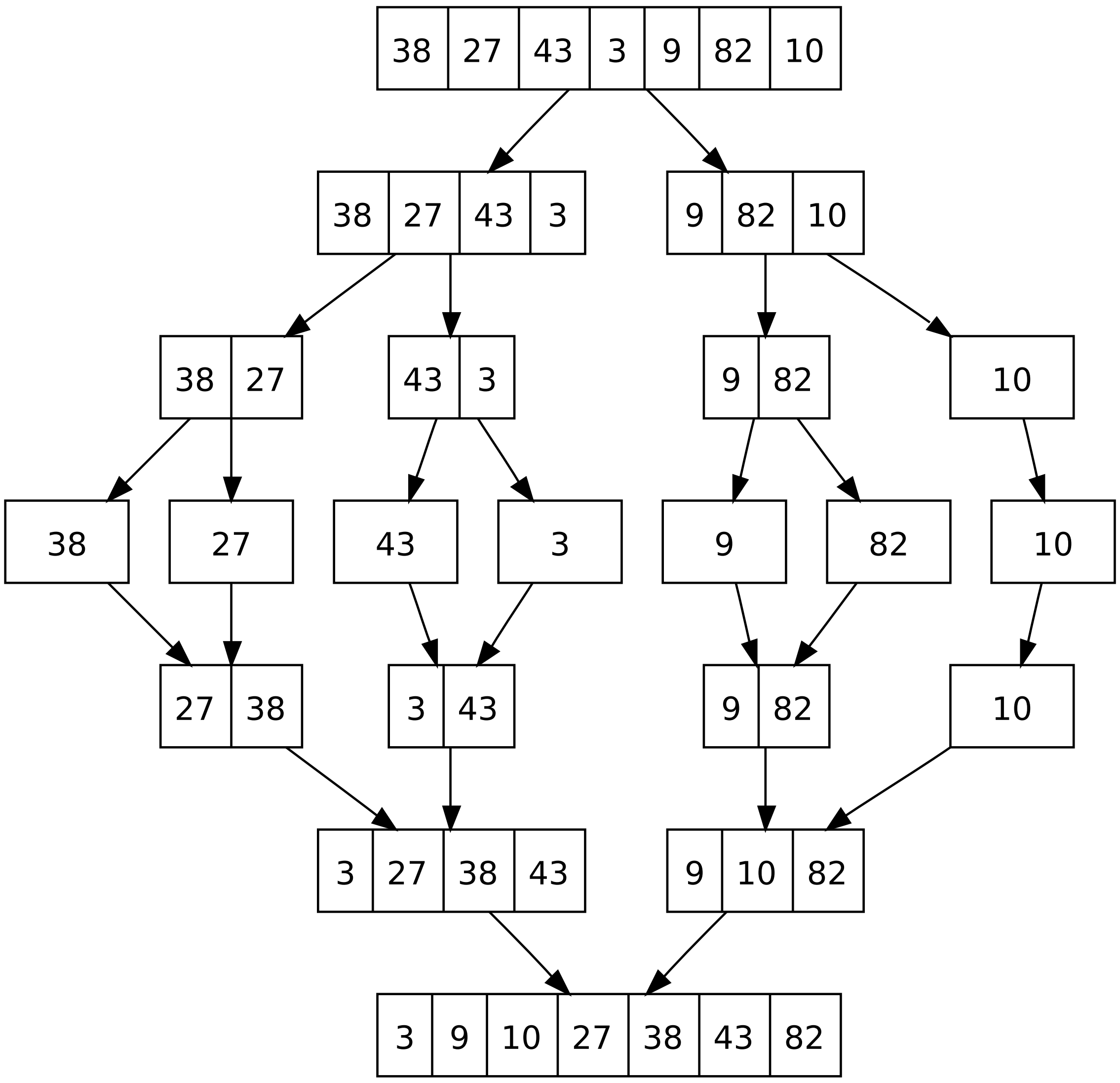
In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than ($n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$), which is between ($n \lg n - n + 1$) and ($n \lg n + n + O(\lg n)$).[4]

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha{\cdot}n$ fewer than the worst case where $\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k+1} \approx 0.2645$.

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case. In terms of moves, merge sort's worst case complexity is O(n log n)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

Merge sort's most common implementation does not sort in place;[5] therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only $n/2$ extra spaces).



A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

4.4 Variants

Variants of merge sort are primarily concerned with reducing the space complexity and the cost of copying.

A simple alternative for reducing the space overhead to $n/2$ is to maintain *left* and *right* as a combined structure, copy only the *left* part of *m* into temporary space, and to direct the *merge* routine to place the merged output into *m*. With this version it is better to allocate the temporary space outside the *merge* routine, so that only one allocation is needed. The excessive copying mentioned previously is also mitigated, since the last pair of lines before the *return result* statement (function *merge* in the pseudo code above) become superfluous.

One drawback of merge sort, when implemented on arrays, is its $O(n)$ working memory requirement. Several in-place variants have been suggested:

- Katajainen *et al.* present an algorithm that requires a constant amount of working memory: enough storage space to hold one element of the input array, and additional space to hold $O(1)$ pointers into the input array. They achieve an $O(n \log n)$ time bound with small constants, but their algorithm is not stable.[6]

- Several attempts have been made at producing an *in-place merge* algorithm that can be combined with a standard (top-down or bottom-up) merge sort to produce an in-place merge sort. In this case, the notion of "in-place" can be relaxed to mean "taking logarithmic stack space", because standard merge sort requires that amount of space for its own stack usage. It was shown by Geffert *et al.* that in-place, stable merging is possible in $O(n \log n)$ time using a constant amount of scratch space, but their algorithm is complicated and has high constant factors: merging arrays of length n and m can take $5n + 12m + o(m)$ moves.[7] This high constant factor and complicated in-place algorithm was made simpler and easier to understand. Bing-Chao Huang and Michael A. Langston[8] presented a straightforward linear time algorithm *practical in-place merge* to merge a sorted list using fixed amount of additional space. They both have used the work of Kronrod and others. It merges in linear time and constant extra space. The algorithm takes little more average time than standard merge sort algorithms, free to exploit O(n) temporary extra memory cells, by less than a factor of two. Though the algorithm is much faster in practical way but it is unstable also for some list. But using similar concept they have been able to solve this problem. Other in-place algorithms include SymMerge, which takes $O((n + m) \log (n + m))$ time in total.[9] Plugging such an algorithm into merge sort increases its complexity to the non-linearithmic, but still quasilinear, $O(n (\log n)^2)$.

An alternative to reduce the copying into multiple lists is to associate a new field of information with each key (the elements in *m* are called keys). This field will be used to link the keys and any associated information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used. This is a standard sorting technique, not restricted to merge sort.

4.5 Use with tape drives

Merge sort type algorithms allowed large data sets to be sorted on early computers that had small random access memories by modern standards. Records were stored on magnetic tape and processed on banks of magnetic tape drives, such as these IBM 729s.

An external merge sort is practical to run using disk or tape drives when the data to be sorted is too large to fit into memory. External sorting explains how merge sort is implemented with disk drives. A typical tape drive sort uses four tape drives. All I/O is sequential (except for rewinds at the end of each pass). A minimal implementation can get by with just 2 record buffers and a few program variables.

Naming the four tape drives as A, B, C, D, with the original data on A, and using only 2 record buffers, the algorithm is similar to Bottom-up implementation, using pairs of tape drives instead of arrays in memory. The basic algorithm can be described as follows:

1. Merge pairs of records from A; writing two-record sublists alternately to C and D.
2. Merge two-record sublists from C and D into four-record sublists; writing these alternately to A and B.
3. Merge four-record sublists from A and B into eight-record sublists; writing these alternately to C and D
4. Repeat until you have one list containing all the data, sorted—in $\log_2(n)$ passes.

Instead of starting with very short runs, usually a hybrid algorithm is used, where the initial pass will read many records into memory, do an internal sort to create a long run, and then distribute those long runs onto the output set. The step avoids many early passes. For example, an internal sort of 1024 records will save 9 passes. The internal sort is often large because it has such a benefit. In fact, there are techniques that can make the initial runs longer than the available internal memory.[10]

A more sophisticated merge sort that optimizes tape (and disk) drive usage is the polyphase merge sort.

4.6 Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multilevel memory hierarchies are used. Cache-aware versions of the merge sort algorithm, whose operations have been specifically chosen to minimize the movement of pages in and out of a machine's memory cache, have been proposed. For example, the **tiled merge sort** algorithm stops partitioning subarrays when subarrays of size S are reached, where S is the number of data items fitting into a CPU's cache. Each of these subarrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion. This algorithm has demonstrated better performance on machines that benefit from cache optimization. (LaMarca & Ladner 1997)

Kronrod (1969) suggested an alternative version of merge sort that uses constant additional space. This algorithm was later refined. (Katajainen, Pasanen & Teuhola 1996)

Also, many applications of external sorting use a form of merge sorting where the input get split up to a higher number of sublists, ideally to a number for which merging them still makes the currently processed set of pages fit into main memory.

4.7 Parallel merge sort

Merge sort parallelizes well due to use of the divide-and-conquer method. Several parallel variants are discussed in the third edition of Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*.[11] The first of these can be very easily expressed in a pseudocode with fork and join keywords:

// Sort elements lo through hi (exclusive) of array A. **algorithm** mergesort(A, lo, hi) **is if** lo+1 < hi **then** *// Two or more elements.* mid = ⌊(lo + hi) / 2⌋ **fork** mergesort(A, lo, mid) mergesort(A, mid, hi) **join** merge(A, lo, mid, hi)

This algorithm is a trivial modification from the serial version, and its speedup is not impressive: when executed on an infinite number of processors, it runs in $\Theta(n)$ time, which is only a $\Theta(\log n)$ improvement on the serial version. A better result can be obtained by using a parallelized merge algorithm, which gives parallelism $\Theta(n / (\log n)^2)$, meaning that this type of parallel merge sort runs in

$$\Theta\left((n\log n)\cdot\frac{(\log n)^2}{n}\right)=\Theta((\log n)^3)$$

time if enough processors are available.[11] Such a sort can perform well in practice when combined with a fast stable sequential sort, such as insertion sort, and a fast sequential merge as a base case for merging small arrays.[12]

Merge sort was one of the first sorting algorithms where optimal speed up was achieved, with Richard Cole using a clever subsampling algorithm to ensure $O(1)$ merge.[13] Other sophisticated parallel sorting algorithms can achieve the same or better time bounds with a lower constant. For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW parallel random-access machine (PRAM) with n processors by performing partitioning implicitly.[14] Powers[15] further shows that a pipelined version of Batcher's Bitonic Mergesort at $O((\log n)^2)$ time on a butterfly sorting network is in practice actually faster than his $O(\log n)$ sorts on a PRAM, and he provides detailed discussion of the hidden overheads in comparison, radix and parallel sorting.

4.8 Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$. On typical modern architectures, efficient quicksort implementations generally outperform mergesort for sorting RAM-based arrays. On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java, the Arrays.sort() methods use merge sort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.[16] The linux kernel uses merge sort for its linked lists.[17] Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7,[18] on the Android platform,[19] and in GNU Octave.[20]

4.9 Notes

[1] Knuth (1998, p. 158)

[2] Jyrki Katajainen and Jesper Larsson Träff (1997). “A meticulous analysis of mergesort programs”.

[3] Powers, David M. W. and McMahon Graham B. (1983), “A compendium of interesting prolog programs”, DCS Technical Report 8313, Department of Computer Science, University of New South Wales.

[4] The worst case number given here does not agree with that given in Knuth's *Art of Computer Programming, Vol 3*. The discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal

[5] Cormen; Leiserson; Rivest; Stein. *Introduction to Algorithms*. p. 151. ISBN 978-0-262-03384-8.

[6] Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). “Practical in-place mergesort”. *Nordic J. Computing*. **3** (1): 27–40. CiteSeerX 10.1.1.22.8523.

[7] Geffert, Viliam; Katajainen, Jyrki; Pasanen, Tomi (2000). “Asymptotically efficient in-place merging”. *Theoretical Computer Science*. **237**: 159–181. doi:10.1016/S0304-3975(98)00162-5.

[8] Huang, Bing-Chao; Langston, Michael A. (March 1988). “Practical In-Place Merging”. *Communications of the ACM*. **31** (3): 348–352. doi:10.1145/42392.42403.

[9] Kim, Pok-Son; Kutzner, Arne (2004). *Stable Minimum Storage Merging by Symmetric Comparisons*. European Symp. Algorithms. Lecture Notes in Computer Science. **3221**. pp. 714–723. CiteSeerX 10.1.1.102.4612. doi:10.1007/978-3-540-30140-0_63. ISBN 978-3-540-23025-0.

[10] Selection sort. Knuth's snowplow. Natural merge.

[11] Cormen et al. 2009, pp. 797–805

[12] V. J. Duvanenko, “Parallel Merge Sort”, Dr. Dobb's Journal, March 2011

[13] Cole, Richard (August 1988). “Parallel merge sort”. *SIAM J. Comput.* **17** (4): 770–785. doi:10.1137/0217049

[14] Powers, David M. W. Parallelized Quicksort and Radixsort with Optimal Speedup, *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.

[15] David M. W. Powers, Parallel Unification: Practical Complexity, Australasian Computer Architecture Workshop, Flinders University, January 1995

[16] OpenJDK Subversion

[17] linux kernel /lib/list_sort.c

[18] jjb. "Commit 6804124: Replace "modified mergesort" in java.util.Arrays.sort with timsort". *Java Development Kit 7 Hg repo*. Retrieved 24 Feb 2011.

[19] "Class: java.util.TimSort<T>". *Android JDK Documentation*. Archived from the original on January 20, 2015. Retrieved 19 Jan 2015.

[20] "liboctave/util/oct-sort.cc". *Mercurial repository of Octave source code*. Lines 23-25 of the initial comment block. Retrieved 18 Feb 2013. Code stolen in large part from Python's, listobject.c, which itself had no license header. However, thanks to Tim Peters for the parts of the code I ripped-off.

4.10 References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort". *Nordic Journal of Computing*. **3**. pp. 27–40. ISSN 1236-6064. Retrieved 2009-04-04.. Also Practical In-Place Mergesort. Also
- Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching*. The Art of Computer Programming. **3** (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
- Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field". *Soviet Mathematics - Doklady*. **10**. p. 744.
- LaMarca, A.; Ladner, R. E. (1997). "The influence of caches on the performance of sorting". *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97)*: 370–379.
- Sun Microsystems. "Arrays API". Retrieved 2007-11-19.
- Sun Microsystems. "java.util.Arrays.java". Retrieved 2007-11-19.

4.11 External links

- Animated Sorting Algorithms: Merge Sort at the Wayback Machine (archived 6 March 2015) – graphical demonstration
- Dictionary of Algorithms and Data Structures: Merge sort
- Mergesort applet with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.1 - Merge Sort

Chapter 5

Strassen algorithm

Not to be confused with the Schönhage–Strassen algorithm for multiplication of polynomials.

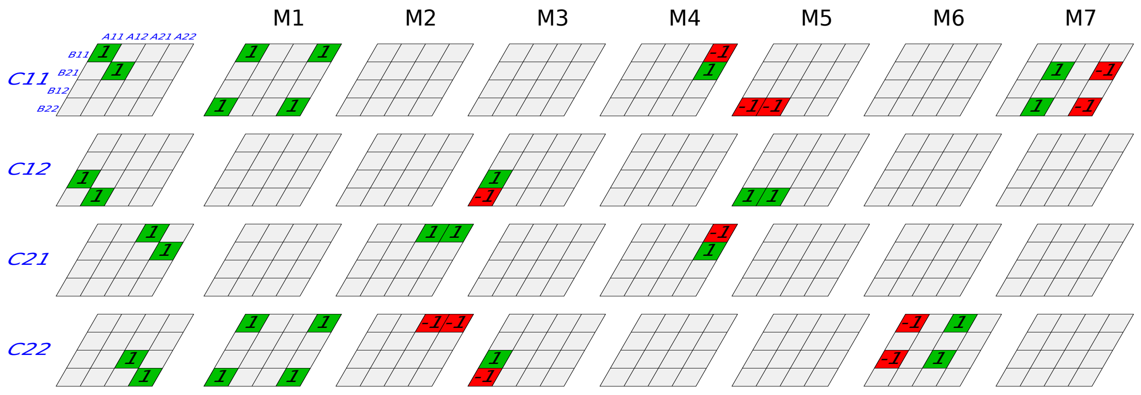
In linear algebra, the **Strassen algorithm**, named after Volker Strassen, is an algorithm for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithms for extremely large matrices.

Strassen's algorithm works for any ring, such as plus/multiply, but not all semirings, such as min/plus or boolean algebra, where the naive algorithm still works, and so called combinatorial matrix multiplication.

5.1 History

Volker Strassen first published this algorithm in 1969 and proved that the n^3 general matrix multiplication algorithm wasn't optimal. The **Strassen algorithm** is only slightly better, but its publication resulted in much more research about matrix multiplication that led to faster approaches, such as the Coppersmith-Winograd algorithm.

5.2 Algorithm



The left column represents 2x2 matrix multiplication. Naïve matrix multiplication requires one multiplication for each "1" of the left column. Each of the other columns represents a single one of the 7 multiplications in the algorithm, and the sum of the columns gives the full matrix multiplication on the left.

Let A, B be two square matrices over a ring R. We want to calculate the matrix product C as

$$\mathbf{C} = \mathbf{A}\mathbf{B} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices A, B are not of type $2^n \times 2^n$ we fill the missing rows and columns with zeros.

We partition *A*, *B* and *C* into equally sized block matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

with

$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

then

$$\mathbf{C}_{1,1} = \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1}$$
$$\mathbf{C}_{1,2} = \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2}$$
$$\mathbf{C}_{2,1} = \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1}$$
$$\mathbf{C}_{2,2} = \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}$$

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the *Ci,j* matrices, the same number of multiplications we need when using standard matrix multiplication.

Now comes the important part. We define new matrices

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$$
$$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$
$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$$
$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$
$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$
$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$
$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

only using 7 multiplications (one for each M_k) instead of 8. We may now express the $C_{i,j}$ in terms of M_k, like this:

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

We iterate this division process *n* times (recursively) until the submatrices degenerate into numbers (elements of the ring *R*). The resulting product will be padded with zeroes just like *A* and *B*, and should be stripped of the corresponding rows and columns.

Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough submatrices, for which those algorithms are more efficient. The particular crossover point for which Strassen's algorithm is more efficient depends on the specific implementation and hardware. Earlier authors had estimated that Strassen's algorithm is faster for matrices with widths from 32 to 128 for optimized implementations.[1] However, it has been observed that this crossover point has been increasing in recent years, and a 2010 study found that even a single step of Strassen's algorithm is often not beneficial on current architectures, compared to a highly optimized traditional multiplication, until matrix sizes exceed 1000 or more, and even for matrix sizes of several thousand the benefit is typically marginal at best (around 10% or less).[2]

5.3 Asymptotic complexity

The standard matrix multiplication takes approximately $2N^3$ (where $N = 2^n$) arithmetic operations (additions and multiplications); the asymptotic complexity is $\Theta(N^3)$.

The number of additions and multiplications required in the Strassen algorithm can be calculated as follows: let $f(n)$ be the number of operations for a $2^n \times 2^n$ matrix. Then by recursive application of the Strassen algorithm, we see that $f(n) = 7f(n-1) + \ell 4^n$, for some constant ℓ that depends on the number of additions performed at each application of the algorithm. Hence $f(n) = (7 + o(1))^n$, i.e., the asymptotic complexity for multiplying matrices of size $N = 2^n$ using the Strassen algorithm is

$$O([7+o(1)]^n) = O(N^{\log_2 7+o(1)}) \approx O(N^{2.8074})$$

The reduction in the number of arithmetic operations however comes at the price of a somewhat reduced numerical stability,[3] and the algorithm also requires significantly more memory compared to the naive algorithm. Both initial matrices must have their dimensions expanded to the next power of 2, which results in storing up to four times as many elements, and the seven auxiliary matrices each contain a quarter of the elements in the expanded ones.

5.3.1 Rank or bilinear complexity

The bilinear complexity or **rank** of a bilinear map is an important concept in the asymptotic complexity of matrix multiplication. The rank of a bilinear map $\phi : \mathbf{A} \times \mathbf{B} \to \mathbf{C}$ over a field $\mathbf{F}$ is defined as (somewhat of an abuse of notation)

$$R(\phi/\mathbf{F}) = \min\left\{ r \,\middle|\, \exists f_i \in \mathbf{A}^*, g_i \in \mathbf{B}^*, w_i \in \mathbf{C}, \forall \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}, \phi(\mathbf{a},\mathbf{b}) = \sum_{i=1}^{r} f_i(\mathbf{a}) g_i(\mathbf{b}) w_i \right\}$$

In other words, the rank of a bilinear map is the length of its shortest bilinear computation.[4] The existence of Strassen's algorithm shows that the rank of 2×2 matrix multiplication is no more than seven. To see this, let us express this algorithm (alongside the standard algorithm) as such a bilinear computation. In the case of matrices, the dual spaces **A*** and **B*** consist of maps into the field **F** induced by a scalar **double-dot product**, (i.e. in this case the sum of all the entries of a Hadamard product.)

It can be shown that the total number of elementary multiplications L required for matrix multiplication is tightly asymptotically bound to the rank R, i.e. $L = \Theta(R)$, or more specifically, since the constants are known, $\frac{1}{2}R \leq L \leq R$. One useful property of the rank is that it is submultiplicative for tensor products, and this enables one to show that $2^n \times 2^n \times 2^n$ matrix multiplication can be accomplished with no more than 7^n elementary multiplications for any n. (This n-fold tensor product of the 2×2×2 matrix multiplication map with itself—an nth tensor power—is realized by the recursive step in the algorithm shown.)

5.3.2 Cache behavior

Strassen's algorithm is cache oblivious. Analysis of its cache behavior algorithm has shown it to incur

$$\Theta\left(1 + \frac{n^2}{b} + \frac{n^{\log_2 7}}{b\sqrt{M}}\right)$$

cache misses during its execution, assuming an idealized cache of M lines, each of b bytes.[5]:13

5.4 Implementation considerations

The description above states that the matrices are square, and the size is a power of two, and that padding should be used if needed. This restriction allows the matrices to be split in half, recursively, until limit of scalar multiplication is reached. The restriction simplifies the explanation, and analysis of complexity, but is not actually necessary;[6] and in fact, padding the matrix as described will increase the computation time and can easily eliminate the fairly narrow time savings obtained by using the method in the first place.

A good implementation will observe the following:

- It is not necessary or desirable to use the Strassen algorithm down to the limit of scalars. Compared to conventional matrix multiplication, the algorithm adds a considerable $O(n^2)$ workload in addition/subtractions; so below a certain size, it will be better to use conventional multiplication. Thus, for instance, if you start with matrices that are 1600x1600, there is no need to pad to 2048x2048, since you could subdivide down to 25x25 and then use conventional multiplication at that level.

- The method can indeed be applied to square matrices of any dimension.[2] If the dimension is even, they are split in half as described. If the dimension is odd, zero padding by one row and one column is applied first. Such padding can be applied on-the-fly and lazily, and the extra rows and columns discarded as the result is formed. For instance, suppose the matrices are 199x199. They can be split so that the upper-left portion is 100x100 and the lower-right is 99x99. Wherever the operations require it, dimensions of 99 are zero padded to 100 first. Note, for instance, that the product M_2 is only used in the lower row of the output, so is only required to be 99 rows high; and thus the left factor $(A_{2,1} + A_{2,2})$ used to generate it need only be 99 rows high; accordingly, there is no need to pad that sum to 100 rows; it is only necessary to pad $A_{2,2}$ to 100 columns to match $A_{2,1}$.

Furthermore, there is no need for the matrices to be square. Non-square matrices can be split in half using the same methods, yielding smaller non-square matrices. If the matrices are sufficiently non-square it will be worthwhile reducing the initial operation to more square products, using simple methods which are essentially $O(n^2)$, for instance:

- A product of size [2*N* x *N*] * [*N* x 10*N*] can be done as 20 separate [*N* x *N*] * [*N* x *N*] operations, arranged to form the result;

- A product of size [*N* x 10*N*] * [10*N* x *N*] can be done as 10 separate [*N* x *N*] * [*N* x *N*] operations, summed to form the result.

These techniques will make the implementation more complicated, compared to simply padding to a power-of-two square; however, it is a reasonable assumption that anyone undertaking an implementation of Strassen, rather than conventional, multiplication, will place a higher priority on computational efficiency than on simplicity of the implementation.

In practice, Strassen's algorithm can be implemented to attain better performance than conventional multiplication even for small matrices, for matrices that are not at all square, and without requiring workspace beyond buffers that are already needed for a high-performance conventional multiplication.[7]

5.5 See also

- Computational complexity of mathematical operations
- Gauss–Jordan elimination
- Coppersmith–Winograd algorithm
- Z-order matrix representation
- Karatsuba algorithm, for multiplying *n*-digit integers in $O(n^{\log_2 3})$ instead of in $O(n^2)$ time
- Gauss's complex multiplication algorithm multiplies two complex numbers using 3 real multiplications instead of 4

5.6 References

[1] Skiena, Steven S. (1998), "§8.2.3 Matrix multiplication", *The Algorithm Design Manual*, Berlin, New York: Springer-Verlag, ISBN 978-0-387-94860-7.

[2] D'Alberto, Paolo; Nicolau, Alexandru (2005). *Using Recursion to Boost ATLAS's Performance* (PDF). Sixth Int'l Symp. on High Performance Computing.

[3] Webb, Miller (1975). "Computational complexity and numerical stability". *SIAM J. Comput*: 97–107.

[4] Burgisser, Clausen, and Shokrollahi. *Algebraic Complexity Theory.* Springer-Verlag 1997.

[5] Frigo, M.; Leiserson, C. E.; Prokop, H.; Ramachandran, S. (1999). *Cache-oblivious algorithms* (PDF). Proc. IEEE Symp. on Foundations of Computer Science (FOCS). pp. 285–297.

[6] Higham, Nicholas J. (1990). "Exploiting fast matrix multiplication within the level 3 BLAS" (PDF). *ACM Transactions on Mathematical Software*. **16** (4): 352–368. doi:10.1145/98267.98290.

[7] Huang, Jianyu; Smith, Tyler; Henry, Greg; van de Geijn, Robert (2016). *Strassen's Algorithm Reloaded*. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16).

- Strassen, Volker, *Gaussian Elimination is not Optimal*, Numer. Math. 13, p. 354-356, 1969
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 28: Section 28.2: Strassen's algorithm for matrix multiplication, pp. 735–741.

5.7 External links

- Weisstein, Eric W. "Strassen's Formulas". *MathWorld*.

(also includes formulas for fast matrix inversion)

- Tyler J. Earnest, *Strassen's Algorithm on the Cell Broadband Engine*
- *Simple Strassen's algorithms implementation in C (easy for education purposes)*

Chapter 6

Master theorem

For the result in enumerative combinatorics, see MacMahon Master theorem.
For the result about Mellin transforms, see Ramanujan's master theorem.

In the analysis of algorithms, the **master theorem** provides a solution in asymptotic terms (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms. It was popularized by the canonical algorithms textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein. Not all recurrence relations can be solved with the use of the master theorem; its generalizations include the Akra–Bazzi method.

6.1 Introduction

Consider a problem that can be solved using a recursive algorithm such as the following:

procedure T(n : size of problem) **defined as: if** n < some constant k **then exit** Create $'a'$ subproblems of size n/b in d(n) time **repeat** for a total of $'a'$ times T(n/b) **end repeat** Combine results from subproblems in c(n) time **end procedure**

In the above algorithm we are dividing the problem into a number of subproblems recursively, each subproblem being of size *n*/*b*. This can be visualized as building a call tree with each node of the tree as an instance of one recursive call and its child nodes being instances of subsequent calls. In the above example, each node would have a number of child nodes. Each node does an amount of work that corresponds to the size of the sub problem n passed to that instance of the recursive call and given by $f(n)$. The total amount of work done by the entire tree is the sum of the work performed by all the nodes in the tree.

Algorithms such as above can be represented as a recurrence relation $T(n) = a\ T\left(\frac{n}{b}\right) + f(n)$, where $f(n) = d(n) + c(n)$ in the above procedure. This recursive relation can be successively substituted into itself and expanded to obtain expression for total amount of work done.[1]

The Master theorem allows us to easily calculate the running time of such a recursive algorithm in Θ-notation without doing an expansion of the recursive relation above.

6.2 Generic form

The master theorem concerns recurrence relations of the form:

$$T(n) = a\,T\!\left(\frac{n}{b}\right) + f(n) \text{ where } a \in \mathbb{N},\ 1 < b \in \mathbb{R}$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- *n* is the size of the problem.

- *a* is the number of subproblems in the recursion.
- *n*/*b* is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- *f* (*n*) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

It is possible to determine an asymptotic tight bound in these three cases:

6.2.1 Case 1

Generic form

If $f(n) = O(n^c)$ where $c < \log_b a$ (using big O notation)
then:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Example

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

As one can see from the formula above:

$$a = 8,\ b = 2,\ f(n) = 1000n^2 \text{ , so}$$
$$f(n) = O(n^c) \text{ , where } c = 2$$

Next, we see if we satisfy the case 1 condition:

$$\log_b a = \log_2 8 = 3 > c$$

It follows from the first case of the master theorem that

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^3\right)$$

(indeed, the exact solution of the recurrence relation is $T(n) = 1001n^3 - 1000n^2$, assuming $T(1) = 1$).

6.2.2 Case 2

Generic form

If it is true, for some constant $k \geq 0$, that:

$$f(n) = \Theta\left(n^c \log^k n\right) \text{ where } c = \log_b a$$

then:

$$T(n) = \Theta\left(n^c \log^{k+1} n\right)$$

Example

$T(n) = 2T\left(\frac{n}{2}\right) + 10n$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ c = 1,\ f(n) = 10n$$
$$f(n) = \Theta\left(n^c \log^k n\right) \text{ where } c = 1, k = 0$$

Next, we see if we satisfy the case 2 condition:

$$\log_b a = \log_2 2 = 1 \text{ , and therefore, } c = \log_b a$$

So it follows from the second case of the master theorem:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right) = \Theta\left(n^1 \log^1 n\right) = \Theta\left(n \log n\right)$$

Thus the given recurrence relation *T*(*n*) was in Θ(*n* log *n*).

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = n + 10n\log_2 n$, assuming $T(1) = 1$.)

6.2.3 Case 3

Generic form

If it is true that:

$$f(n) = \Omega\left(n^c\right) \text{ where } c > \log_b a$$

and if it is also true that:

$af\left(\frac{n}{b}\right) \le kf(n)$ for some constant $k < 1$ and sufficiently large n (often called the *regularity condition*)

then:

$$T\left(n\right) = \Theta\left(f(n)\right)$$

Example

$T(n) = 2T\left(\frac{n}{2}\right) + n^2$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ f(n) = n^2$$
$$f(n) = \Omega\left(n^c\right) \text{ , where } c = 2$$

Next, we see if we satisfy the case 3 condition:

$$\log_b a = \log_2 2 = 1 \text{ , and therefore, yes, } c > \log_b a$$

The regularity condition also holds:

$2\left(\frac{n^2}{4}\right) \le kn^2$, choosing $k = 1/2$

So it follows from the third case of the master theorem:

$$T(n) = \Theta(f(n)) = \Theta\left(n^2\right).$$

Thus the given recurrence relation $T(n)$ was in $\Theta(n^2)$, that complies with the $f(n)$ of the original formula.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 2n^2 - n$, assuming $T(1) = 1$.)

6.3 Inadmissible equations

The following equations cannot be solved using the master theorem:[2]

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$

  a is not a constant; the number of subproblems should be fixed

- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

  $n^{\log_b a}$

- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

  $a < 1$

- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

  f(n), which is the combination time, is not positive

- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

  case 3 but regularity violation.

In the second inadmissible example above, the difference between $f(n)$ and $n^{\log_b a}$ can be expressed with the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n/\log n}{n^{\log_2 2}} = \frac{n}{n \log n} = \frac{1}{\log n}$. It is clear that $\frac{1}{\log n} < n^{\epsilon}$ for any constant $\epsilon > 0$. Therefore, the difference is not polynomial and the Master Theorem does not apply.

6.4 See also

- Akra–Bazzi method

6.5 Application to common algorithms

6.6 Notes

[1] Duke University, “Big-Oh for Recursive Functions: Recurrence Relations”, http://www.cs.duke.edu/~{ }ola/ap/recurrence.html

[2] Massachusetts Institute of Technology (MIT), “Master Theorem: Practice Problems and Solutions”, http://www.csail.mit.edu/~{ }thies/6.046-web/master.pdf

[3] Dartmouth College, http://www.math.dartmouth.edu/archive/m19w03/public_html/Section5-2.pdf

6.7 References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw–Hill, 2001. ISBN 0-262-03293-7. Sections 4.3 (The master method) and 4.4 (Proof of the master theorem), pp. 73–90.
- Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundation, Analysis, and Internet Examples*. Wiley, 2002. ISBN 0-471-38365-1. The master theorem (including the version of Case 2 included here, which is stronger than the one from CLRS) is on pp. 268–270.

6.8 Text and image sources, contributors, and licenses

6.8.1 Text

- **Recursion (computer science)** *Source:* https://en.wikipedia.org/wiki/Recursion_(computer_science)?oldid=771865382 *Contributors:* Edward, Michael Hardy, Kku, Dwo, Aenar, Thilo, Mattflaschen, Tea2min, Giftlite, Macrakis, Derek Parnell, Andreas Kaufmann, Abdull, MMSequeira, Paul August, Nigelj, R. S. Shaw, Liao, Diego Moya, Jeltz, Kenyon, Mahanga, Linas, Mindmatrix, MattGiuca, Ruud Koot, Graham87, Raymond Hill, Jclemens, Rjwilmsi, Leeyc0, Salix alba, Essayemyoung4009, CalPaterson, Nihiltres, Quuxplusone, Ver, Chobot, Wavelength, Grafen, Dijxtra, Trovatore, Jpbowen, Dbfirs, Rwalker, Tcsetattr, Cedar101, JLaTondre, RandallZ, Brahle, Linkminer, SmackBot, InverseHypercube, McGeddon, Bigbluefish, Philiprogers, Gilliam, LinguistAtLarge, Thumperward, Timneu22, Nbarth, David Morón, JonHarder, Tobyink, Clements, T-borg, Almkglor, Loadmaster, Valepert, Tmcw, AlainD, Shabbirbhimani, CRGreathouse, CBM, Pierre de Lyon, Cydebot, NotQuiteEXPComplete, BetacommandBot, Thijs!bot, Escarbot, PhiLiP, Gioto, Salgueiro~enwiki, Lfstevens, JAnDbot, Magioladitis, Walkeraj, Abednigo, David Eppstein, E-pen, R'n'B, Mange01, Maurice Carbonaro, Coppertwig, Shoessss, Mythrill, Oshwah, DaoKaioshin, Nxavar, Awl, Viggio, Pi is 3.14159, Volkan YAZICI, Sara.noorafkan, Hariva, Martarius, ClueBot, Cchantep~enwiki, Jeshan, Mild Bill Hiccup, Erudecorp, Tim32, Dsamarin, M4gnum0n, XLinkBot, Brentsmith101, Drolz09, Dsimic, Addbot, Ghettoblaster, Poco a poco, Merarischroeder, Btx40, Mohamed Magdy, Tide rolls, Fryed-peach, Luckas-bot, Quadrescence, Yobot, Pcap, KamikazeBot, Tempodivalse, AnomieBOT, Materialscientist, Citation bot, GB fan, ArthurBot, GrouchoBot, RibotBOT, Constructive editor, Medanat, Citation bot 1, DrilBot, XxTimberlakexx, RedBot, Babayagagypsies, MoreNet, WillNess, John lindgren, Forgefight, EmausBot, Tranhungnghiep, Chharvey, D.Lazard, Bxj, RenaissanceBug, Simurgia, Arnaud741, Carmichael, Brian Tansley, Walfredo424-NJITWILL, ClueBot NG, Widr, BG19bot, Wikimpan, Uri-Levy, Chmarkine, Jon.weldon, TheJJJunk, Splendor78, Jochen Burghardt, Mark viking, Gratimax, Carrot Lord, Comp.arch, Monkbot, Peturb, Jacob p12, Lomotos10, Phạm Nguyễn Trường An, Paul STice, Rakshit93, Lemondoge, Sri Nikitha Sarikonda, Cortis1, Ushkin N, Bender the Bot, SweetCisvestite, Akanji Paul and Anonymous: 196
- **Divide and conquer algorithm** *Source:* https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm?oldid=763063247 *Contributors:* XJaM, Pnm, Kku, TakuyaMurata, Looxix~enwiki, Stevenj, Poor Yorick, Dcoetzee, Daniel Quinlan, Furrykef, AndrewKepert, Head, Fredrik, Adam78, Giftlite, Jorge Stolfi, LiDaobing, Phe, Zamfi, Mycompsimm, El C, Bobo192, R. S. Shaw, Neg, Nsaa, Mdd, Varuna, ThePedanticPrick, MIT Trekkie, Finem, Linas, Madmardigan53, Ruud Koot, WadeSimMiser, Qwertyus, Bhadani, VKokielov, DevastatorIIC, Tardis, Taichi, Chobot, Roboto de Ajvol, YurikBot, Amshali, Irrevenant, Dtrebbien, Goffrie, BOT-Superzerocool, Josh3580, Garion96, Thijswijs, SmackBot, McGeddon, Unyoyega, Atomota, Bluebot, Nbarth, Mlpkr, Vina-iwbot~enwiki, SashatoBot, G-Bot~enwiki, BrownHairedGirl, Jake-helliwell, Tawkerbot2, Destructor~enwiki, Nczempin, Cydebot, Jfcorbett, Krauss, IrishPete, Dhrm77, JAnDbot, .anacondabot, Magioladitis, David Eppstein, JaGa, Drewmutt, Pomte, J.delanoy, LokiClock, Philip Trueman, Rei-bot, LanceBarber, Wolfrock, SieBot, Antzervos, OKBot, Excirial, Quercus basaseachicensis, Bob man801, Thingg, Pichpich, DragonFury, Addbot, Mortense, CL, Jarble, Luckas-bot, Yobot, AnomieBOT, Ailun, Citation bot, LilHelpa, Xqbot, Ekaratsuba, Almabot, Algoritmist, Prunesqualer, UTQ Shadow, Citation bot 1, Boxplot, Trappist the monk, ArbitUsername, Bluefist, PowerPaul86, EmausBot, Dcirovic, Tolly4bolly, Staszek Lem, Bomazi, ChuispastonBot, Petrb, ClueBot NG, Widr, Riteshrit20, Nullzero, Zebulon84, Solomon7968, Bloghog23, Monkbot, Kurousagi, Jewkub and Anonymous: 101
- **Karatsuba algorithm** *Source:* https://en.wikipedia.org/wiki/Karatsuba_algorithm?oldid=752936243 *Contributors:* Michael Hardy, Stevenj, Cherkash, Nikola Smolenski, Dcoetzee, Doradus, Ruakh, Mattflaschen, Giftlite, Jorge Stolfi, Spoon!, LutzL, Linas, Shreevatsa, BD2412, Vincent Lefèvre, Rjwilmsi, Parerga, Chobot, Bgwhite, Arado, SmackBot, Berland, Cronholm144, Jafet, CRGreathouse, Ksoileau, Cydebot, Verdy p, Jamesjiao, Liquid-aim-bot, Darklilac, MER-C, Javazen, David Eppstein, VolkovBot, Enkya, Dmcq, Babloyi, JackSchmidt, Adrianwn, Mild Bill Hiccup, Alexbot, Bob man801, Aigdonia, XLinkBot, Marc van Leeuwen, Addbot, LaaknorBot, PV=nRT, Yobot, Rubinbot, Unready, Poolisfun, Charvest, Smallman12q, Douglas W. Jones, X7q, Steve Quinn, Xnn, EmausBot, WikitanvirBot, Nkyaelly, KuduIO, Staszek Lem, GermanJoe, ClueBot NG, Solomon7968, JingguoYao, Jeff Erickson, Kizzlemonster, Rhuaidhri, Andylamp, Nstrch, Wqwt, Theo Kortekaas, Faslof, Bender the Bot, Soubhik and Anonymous: 91
- **Merge sort** *Source:* https://en.wikipedia.org/wiki/Merge_sort?oldid=772302381 *Contributors:* Damian Yerrick, Lee Daniel Crocker, Mav, The Anome, JohnOwens, Chris-martin, Pnm, Shellreef, Dcljr, TakuyaMurata, Eric119, Minesweeper, Rl, Timwi, Dcoetzee, Ww, Kbk, Daniel Quinlan, Jogloran, Joshk, Furrykef, Thue, Garo, Robbot, Fredrik, Tomchiukc, Romanm, JerryFriedman, Jleedev, Tea2min, Decrypt3, Giftlite, DocWatson42, Frencheigh, T0m, JF Bastien, Neilc, Chowbok, Pgan002, Knutux, Onco p53, Wilagobler, Nickls, Sam nead, Naku~enwiki, Oskar Sigvardsson, Rfl, Discospinster, Rich Farmbrough, Guanabot, Rspeer, ArnoldReinhold, Sperling, Kenb215, Danakil, MisterSheik, Adambro, Jpl~enwiki, Nyenyec, BrokenSegue, Mikewebkist, Haham hanuka, Ossi~enwiki, Jumbuck, Alansohn, Sligocki, Swift, Caesura, ReyBrujo, Mattjohnson, Dr. Gonzo, WojciechSwiderski~enwiki, Forderud, Kenyon, Bobrayner, Daniel Geisler, Soultaco, Mntlchaos, Nuno Tavares, Merlinme, Jacobolus, Ruud Koot, Vorn, Fred J, GregorB, Tommunist, Palica, Pfalstad, Wbeek, Qwertyus, GrundyCamellia, Grammarbot, Rjwilmsi, Vexis, Bayard~enwiki, Bernard van der Wees, Bubba73, Nguyen Thanh Quang, SLi, StuartBrady, FlaBot, Ewlyahoocom, DevastatorIIC, Comocomocomocomo, Intgr, CiaPan, CJLL Wright, Chobot, DVdm, Garas, Cactus.man, Peterl, Dbagnall, Hyad, Jengelh, Stephenb, Zhaladshar, Mipadi, Fashnek, Mikeblas, Black Falcon, Lt-wiki-bot, Abu adam~enwiki, Cedar101, GraemeL, Donhalcon, Thijswijs, SmackBot, Damonkohler, Kalebdf, Renku, Delldot, Scott Paeth, Gilliam, Ohnoitsjamie, Andy M. Wang, Sirex98, Oli Filth, Silly rabbit, Nbarth, DHN-bot~enwiki, Nixeagle, Avb, EdSchouten, Allan McInnes, Easwarno1, HoserHead, Zophar1, Cybercobra, Duckbill, OranL, Oxaric, ThomasMueller, SeanAhern, Itmozart, Andrei Stroe, Nmnogueira, Cuzelac, Cyclopaedic, Sir Nicholas de Mimsy-Porpington, Apoorbo, Noah Salzman, Dammit, MTSbot~enwiki, Negrulio, PavelY, UncleDouggie, GDallimore, Amiodusz, Jafet, Xueshengyao, CRGreathouse, Ahy1, Bakanov, Mblumber, Thijs!bot, Towopedia, Radiozilla, Ablonus, AntiVandalBot, WinBot, Widefox, VineetKumar, Jirka6, Wootery, Martinkunev, SiobhanHansa, Antientropic, CobaltBlue, Dima1, Eleschinski2000, Cic, Destynova, Dbingham, Pkrecker, Maju wiki, Glrx, Ceros, J.delanoy, Acalamari, Faridani, Sanjay742, ScottBurson, Vanished user 39948282, Alain Amiouni, TXiKiBoT, Oshwah, Daztekk, Amahdy, Liko81, JhsBot, Sychen, SashaMarievskaya, Dmcq, Rhanekom, SieBot, Coffee, Artagnon, Laoris, Fizo86, Keilana, Ctxppc, Garrettw87, Hariva, Xhackeranywhere, Novas0x2a, ClueBot, DFRussia, PipepBot, Fyyer, Garyzx, N26ankur, DragonBot, Excirial, Bender2k14, TobiasPersson, Johnuniq, XLinkBot, EAspenwood, Oğuz Ergin, C. A. Russell, Bill wang1234, Hoof1341, Addbot, Fyrael, Miskaton, Rcgldr, MrOllie, Download, Worch, Lightbot, Ninjatummen~enwiki, Geeker87, Yobot, Tohd8BohaithuGh1, Fraggle81, Timeroot, NotARusski, Naderra, AnomieBOT, Erel Segal, Kingpin13, Citation bot, ArthurBot, JimVC3, Octotron, Locobot, A3ng3l, Shadowjams, Captain Fortran, Kracekumar, FrescoBot, X7q, Mctpyt, Ahmadsh, Mecej4, Citation bot 1, Base698, I dream of horses, Jonesey95, Mutinus, RedBot, Tuttu4u, Michele bon, Schorzman78, Deanonwiki, Patmorin, TBloemink, RjwilmsiBot, Wintonian, EmausBot, Immunize, Dewritech, Shashwat2691, Bor75, Yodamgod, Sven nestle2, Mlhetland, Duvavic1, Eserra, Swfung8, Klrste, Josh Kehn, ClueBot NG, Jheld88, Strcat, Hofmic, Widr, Jk2q3jrklse,

6.8.2 Images

6.8.3 Content license