Divide and conquer algorithms

From Wikipedia, the free encyclopedia

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 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. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

On the other hand, the ability to understand and design D&C algorithms is a skill that takes time to master. As when proving a theorem by induction, it is often necessary to replace the original problem by 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.

Contents

Decrease 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 decrease 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.

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.

An early example of a divide-and-conquer algorithm with multiple subproblems is Gauss's 1805 description of what is now called the Cooley-Tukey fast Fourier transform (FFT) algorithm,[5] although he did not analyze its operation count quantitatively and FFTs did not become widespread until they were rediscovered over a century later.

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, 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]

Advantages

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, decrease 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 $n - 1$.

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 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 there are a bounded number p of subproblems of size ~ n/p at each stage, then the cost of the divide-and-conquer algorithm will be $O(n \log n)$.

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.

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(s) 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.

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]

Implementation issues

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 is defined in terms of itself.

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.

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, and/or by using an explicit stack structure.

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 n+1 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]

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.

See also

- Akra–Bazzi method
- Fork–join model
- Master theorem
- Mathematical induction

- MapReduce
- Divide and rule
- Heuristic (computer science)

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 *Physics-Doklady* **7**: 595–596. 1963. Missing or empty `|title=` (help)
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" (http://www.fftw.org/fftw-paper-ieee.pdf) (PDF). *Proceedings of the IEEE* **93** (2): 216–231. doi:10.1109/JPROC.2004.840301 (https://dx.doi.org/10.1109%2FJPROC.2004.840301).
11. Radu Rugina and Martin Rinard, "Recursion unrolling for divide and conquer programs (http://people.csail.mit.edu/rinard/paper/lcpc00.pdf)," in *Languages and Compilers for Parallel Computing*, chapter 3, pp. 34–48. *Lecture Notes in Computer Science* vol. 2017 (Berlin: Springer, 2001).

External links

Retrieved from "https://en.wikipedia.org/w/index.php?title=Divide_and_conquer_algorithms&oldid=657459441"

Categories: Algorithms | Operations research | Optimization algorithms and methods

- This page was last modified on 21 April 2015, at 11:07.
-