**Simple recursive algorithm**

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

If a set or a function is defined recursively, then a recursive algorithm to compute its members or values mirrors the definition. Initial steps of the recursive algorithm correspond to the basis clause of the recursive definition and they identify the basis elements. They are then followed by steps corresponding to the inductive clause, which reduce the computation for an element of one generation to that of elements of the immediately preceding generation.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

**Backtracking algorithm**

Backtracking is an algorithm for capturing some or all solutions to given computational issues, especially for constraint satisfaction issues. The algorithm can only be used for problems which can accept the concept of a “partial candidate solution” and allows a quick test to see if the candidate solution can be a complete solution. Backtracking is considered an important technique to solve constraint satisfaction issues and puzzles. It is also considered a great technique for parsing and also forms the basis of many logic programming languages.

Backtracking helps in solving an overall issue by finding a solution to the first sub-problem and then recursively attempting to resolve other sub-problems based on the solution of the first issue. If the current issue cannot be resolved, the step is backtracked and the next possible solution is applied to previous steps, and then proceeds further. In fact, one of the key things in backtracking is recursion. It is also considered as a method of exhaustive search using divide and conquer. A backtracking algorithm ends when there are no more solutions to the first sub-problem. Backtracking is an algorithm which can help achieve implementation of non determinism. It takes a depth-first search of a given issue space. It is used mostly in logic programming languages like Prolog. Wherever backtracking can be applied, it is faster than the brute force technique, as it eliminates a large number of candidates with a single test.

**Divide and conquer algorithm**

In computer science, divide and conquer 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, finding the closest pair of points, syntactic analysis, and computing the discrete Fourier transform.

A typical Divide and Conquer algorithm solves a problem using following three steps.

1. Divide: Break the given problem into subproblems of same type.

2. Conquer: Recursively solve these subproblems

3. Combine: Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

1). Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5) Strassen’s Algorithm is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen’s algorithm multiplies two matrices in O(n^2.8974) time.

6) Cooley–Tukey Fast Fourier Transform (FFT) algorithm is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(nlogn) time.

7) Karatsuba algorithm for fast multiplication it does multiplication of two n-digit numbers in at most 3 n^{\log\_23}\approx 3 n^{1.585}single-digit multiplications in general (and exactly n^{\log\_23} when n is a power of 2). It is therefore faster than the classical algorithm, which requires n2 single-digit products. If n = 210 = 1024, in particular, the exact counts are 310 = 59,049 and (210)2 = 1,048,576, respectively.

**Dynamic programming algorithm**

Dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not always guarantee an optimal solution, when a dynamic programming algorithm does, because picking locally optimal choices may result in a bad global solution. One advantage of a greedy algorithm over a dynamic programming algorithm is that the greedy algorithm is often faster and simpler to calculate. Some greedy algorithms (such as Kruskal's or Prim's for minimum spanning trees) are known to lead to the optimal solution.

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems. If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called "divide and conquer" instead. This is why merge sort and quick sort are not classified as dynamic programming problems.

*Optimal substructure* means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.

*Overlapping sub-problems* means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

This can be achieved in either of two ways:

* *Top-down approach:* This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.
* *Bottom-up approach:* Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems.

Some programming languages can automatically memoize the result of a function call with a particular set of arguments, in order to speed up call-by-name evaluation (this mechanism is referred to as call-by-need). Some languages make it possible portably. Some languages have automatic memoization built in, such as tabled Prolog and J, which supports memoization with the M. adverb.[4] In any case, this is only possible for a referentially transparent function. Memoization is also encountered as an easily accessible design pattern within term-rewrite based languages such as Wolfram Language

The term *dynamic programming* was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he refined this to the modern meaning, referring specifically to nesting smaller decision problems inside larger decisions, and the field was thereafter recognized by the IEEE as a systems analysis and engineering topic. Bellman's contribution is remembered in the name of the Bellman equation, a central result of dynamic programming which restates an optimization problem in recursive form.

**Greedy algorithm**

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage[1] with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

A greedy algorithm is a mathematical process that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit.

Such algorithms are called greedy because while the optimal solution to each smaller instance will provide an immediate output, the algorithm doesn’t consider the larger problem as a whole. Once a decision has been made, it is never reconsidered. Greedy algorithms work by recursively constructing a set of objects from the smallest possible constituent parts. Recursion is an approach to problem solving in which the solution to a particular problem depends on solutions to smaller instances of the same problem. The advantage to using a greedy algorithm is that solutions to smaller instances of the problem can be straightforward and easy to understand. The disadvantage is that it is entirely possible that the most optimal short-term solutions may lead to the worst possible long-term outcome.

Greedy algorithms are often used in ad hoc mobile networking to efficiently route packets with the fewest number of hops and the shortest delay possible. They are also used in machine learning, business intelligence (BI), artificial intelligence (AI) and programming.

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, and the algorithm for finding optimum Huffman trees.

**Branch and bound algorithm**

Branch and bound (BB, B&B, or BnB) is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The algorithm depends on the efficient estimation of the lower and upper bounds of a region/branch of the search space and approaches exhaustive enumeration as the size (n-dimensional volume) of the region tends to zero.

The method was first proposed by A. H. Land and A. G. Doig in 1960 for discrete programming, and has become the most commonly used tool for solving NP-hard optimization problems. The name "branch and bound" first occurred in the work of Little et al. on the traveling salesman problem.

Branch and bound algorithms are generally used for optimization problems

- As the algorithm progresses, a tree of subproblems is formed

- The original problem is considered the “root problem”

- A method is used to construct an upper and lower bound for a given problem

- At each node, apply the bounding methods

* If the bounds match, it is deemed a feasible solution to that particular subproblem
* If bounds do not match, partition the problem represented by that node, and make the two subproblems into children nodes

- Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed

**Brute force algorithm**

Brute force is a type of algorithm that tries a large number of patterns to solve a problem. In some cases, they are extremely simple and rely on raw computing power to achieve results.

A common example of a brute force algorithm is a security threat that attempts to guess a password using known common passwords. Such an algorithm might also try dictionary words or even every combination of ASCII strings of a certain length.

Brute force algorithms also have non-security applications. For example, a technique known as data dredging is a brute force method of looking for statistically significant patterns in data.

In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

The brute-force approach consists in generating the combinatorial objects and testing each object to see if it satisfies some specified constraints. Problems like the traveling salesman, knapsack, and bin-packing can be solved by using an exhaustive search.

The ***traveling salesman problem*** tries to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city whether it started. Since each tour is a permutation of the n cities, and since we can fix the start city, the time efficiency of this algorithm is Θ((n-1)!).

The ***knapsack problem***, for given n items with their benefits and volume, tries to find the most valuable subset (i.e. the one with the greatest benefit) without exceeding the capacity of the knapsack. The solution is one of all subsets of the set of objects and therefore the time efficiency of this algorithm is Θ(2n).

In the ***assignment problem*** we have n persons that have to be assigned n jobs, one person per job. The problem is represented as a cost matrix, where c(i,j) is the cost of assigning job j to person i. The solution is a vector (a1, a2, ..an) such that person i is assigned job ai and the sum ∑ c(i,ai) is minimum.

**Randomized algorithm**

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input so that they always terminate with the correct answer, but where the expected running time is finite (Las Vegas algorithms, example of which is Quicksort), and algorithms which have a chance of producing an incorrect result (Monte Carlo algorithms, example of which is Monte Carlo algorithm for MFAS) or fail to produce a result either by signaling a failure or failing to terminate.

In the second case, random performance and random output, the term "algorithm" for a procedure is somewhat questionable. In the case of random output, it is no longer formally effective. However, in some cases, probabilistic algorithms are the only practical means of solving a problem.

In common practice, randomized algorithms are approximated using a pseudorandom number generator in place of a true source of random bits; such an implementation may deviate from the expected theoretical behavior.

*Randomized algorithms* are used when presented with a time or memory constraint, and an average case solution is an acceptable output. Due to the potential erroneous output of the algorithm, an algorithm known as amplification is used in order to boost the probability of correctness by sacrificing runtime. Amplification works by repeating the randomized algorithm several times with different random subsamples of the input, and comparing their results. It is common for randomized algorithms to amplify just parts of the process, as too much amplification may increase the running time beyond the given constraints. It usually designed in one of two common forms: as a Las Vegas or as a Monte Carlo algorithm. A Las Vegas algorithm runs within a specified amount of time. If it finds a solution within that timeframe, the solution will be exactly correct; however, it is possible that it runs out of time and does not find any solutions. On the other hand, a Monte Carlo algorithm is a probabilistic algorithm which, depending on the input, has a slight probability of producing an incorrect result or failing to produce a result altogether.