Divide and Conquer Algorithms

Divide a problem into smaller sub-problems
Conquer them recursively
Merge the answers together so as to obtain the answer to the larger problem

Some examples are:

1. Thinking the solution of tower or Hanoi with 'n' disks as; one task to move n-1 disks to the temporary peg, move 'nth' disk to destination, and again move n-1 disks to the destination.
2. Merge sort: Dividing a file into subfiles and then sort the subfiles. After sorting sub-files, merge them.
3. Traversing only one half of the given range in the binary search tree.

Advantages:

Parallelism

If an algorithm can be made as divide and conquer, the sub-problems can be passed to other machines. Other machines work on the sub-problems and send result back to the main machine. The job of main machine is to divide the problems into sub-problems and assemble the result. Some of the problems that we applied this technique are binary search, binary tree search, merge sort etc.

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.

Disadvantages:

1. This approach uses recursion which makes the process little slow.
2. Another problem of a divide-and-conquer approach is that, for simple problems, it may be more complicated than an iterative approach, especially if large base cases are to be implemented for performance reasons. For example, to add *N* numbers, a simple loop to add them up in sequence is much easier to code than a divide-and-conquer approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

Dynamic Programming

Optimal substructure means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem. For example, the shortest path to a goal from a vertex in a graph can be found by first computing the shortest path to the goal from all adjacent vertices, and then using this to pick the best overall path. In general, we can solve a problem with optimal substructure using a three-step process:

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

The subproblems are, themselves, solved by dividing them into sub-subproblems, and so on, until we reach some simple case that is easy to solve.
To say that a problem has overlapping subproblems is to say that the same subproblems are used to solve many different larger problems. For example, in the Fibonacci sequence, F3 = F1 + F2 and F4 = F2 + F3 — computing each number involves computing F2. Because both F3 and F4 are needed to compute F5, a naïve approach to computing F5 may end up computing F2 twice or more. This applies whenever overlapping subproblems are present: a naïve approach may waste time recomputing optimal solutions to subproblems it has already solved.

In order to avoid this, we instead save the solutions to problems we have already solved. Then, if we need to solve the same problem later, we can retrieve and reuse our already-computed solution. This approach is called **memoization** (not memorization, although this term also fits). If we are sure we won't need a particular solution anymore, we can throw it away to save space. In some cases, we can even compute the solutions to subproblems we know that we'll need in advance.

Example:
A naive implementation of a function finding the nth member of the Fibonacci sequence, based directly on the mathematical definition:

```
function fib(n)
 {
      if n <= 0
            return 0;
      if n==1
            return 1;
      else
                  return fib(n − 1) + fib(n − 2);
 }
```

Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + **fib(2)**) + (**fib(2)** + fib(1))
4. ((fib(2) + fib(1)) + (**fib(1) + fib(0)**)) + ((**fib(1) + fib(0)**) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

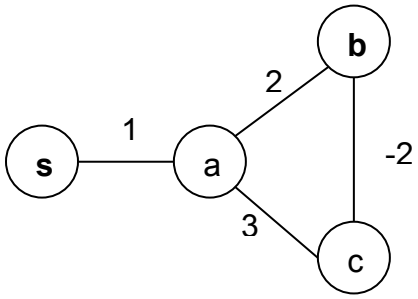
In particular, fib(2) was calculated twice from scratch. In larger examples, many more values of fib, or subproblems, are recalculated, leading to an exponential time algorithm.

Now, suppose we have a simple map object, m, which maps each value of fib that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(n) time instead of exponential time:

Greedy algorithms

Greedy algorithms tend to find the solution very fast. These algorithms choose the best option available for current state only (locally optimal). It lacks fore-sight seeing feature.

The search for minimum path between two nodes in shortest path algorithm is also a greedy algorithm. We just search for the edge for which the sum of weight and path up to that node is minimal. The shortest path also could not guarantee that the result found is the shortest path in case of negatively weighted graphs.



If the shortest path algorithm is applied in given graph (to go to node '**b**' from node'**s**'), then the algorithm will choose edge with length '2' at node 'a' (s, a, b). And hence the minimum path will be path will be of length '3'. But in this case, the path (s, a, c, b) has length '2' only. Therefore the previous prediction was wrong.

Greedy algorithms produce good solutions on some mathematical problems, but not on others. Most problems for which they work well have **two properties**:

Greedy Choice Property

We can make whatever choice seems best at the moment and then solve the sub problems that arise later. The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the sub problem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never *reconsiders its choices*. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Optimal Substructure
A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems.

When greedy-type algorithms fail
For many other problems, greedy algorithms may produce the unique worst possible solutions. One example is the shortest path algorithm mentioned above.

For most problems, 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. 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 optimisation methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, Dijkstra's algorithm for finding Single-Source Shortest paths, and the algorithm for finding optimum Huffman trees.

Backtracking:

The backtracking method is based on the systematically investigation of the possible solutions where through the procedure, set of possible solutions are rejected before even examined that will reduce the search area. This method of reducing the path to optimal solution is known as backtracking.

An important requirement which must be fulfilled is that there must be the proper hierarchy in the systematically produce of solutions so that sets of solutions that do not fulfill a certain requirement are rejected before the solutions are produced. For this reason the examination and produce of the solutions follows a model of non-cycle graph for which in this case we will consider as a tree. The root of the tree represents the set of all the solutions. Nodes in lower levels represent even smaller sets of solutions, based on their properties. Obviously, leaves will be isolated solutions. It is easily understood that the tree (or any other graph) is produced during the examination of the solutions so that no rejected solutions are produced. When a node is rejected, the whole sub-tree is rejected, and we backtrack to the ancestor of the node so that more children are produced and examined. Because this method is expected to produce subsets of solutions which are difficult to process, the method itself is not very popular.

Example: Four Queens

The four Queens problem is to place 4 Queens on the chess board of size 4 so that no Queen can take another. The basic rule is that a queen can take another piece that lies on the same row, the same column, of the same diagonal (either direction) as the queen. The board has 4 rows and 4 columns.

*	?	?	?
x	x	*	?
x	x	x	x

Dead end
(a)

*	?	?	?
x	x	x	*
x	*	x	x
x	x	x	x

Dead end
(b)

x	*	?	?
x	x	x	*
*	x	x	x
x	x	*	x

Solution
(c)

		*	
*			
			*
	*		

Solution
(d)

Here:

* represents queen.
? represents other legitimate choice that we have not yet tried.
x represents invalid position.

We shall need to put one queen in each row of the board. Let us first try to place the queen as far to the left in the column as we can. Such a choice is shown in the first column of part (a). Next we move on the second row, here first two columns are guarded by the queen in row 1. Column 3 and 4 are free, so we place queen in column 3 and mark 4 with a question mark(?). Now, move on to row 3, but we find all four squares are guarded by one of the queens in first two rows. We have now reached to dead end.

When we reach dead end, we must backtrack by going back to the most recent choice we have made and trying another possibility. This situation is shown in part (b). Here, queen in second row is moved to second possible position. Now, column 2 is only available in the third row but all four columns in the forth row is guarded. Hence, we again have reached a point where no other queens can be added, and must backtrack.

At, this point we have no another choice for row 2, so, we must move all the way back to row 1 and move the queen to the next possible position, the column 2. Now, we find that, in row 2 only column 4 is unguarded, so queen must go there. Then in row 3, column 1 is the only possibility, and in row 4, only column 3 is possible. This placement of queens, however, leads to a solution to the problem of four non-attacking queens on the same 4x4 board.

Local Search Algorithm:

One of the trends to find optimal solution is:

1. Find some set of solutions
2. Start with a random solution
3. Make some modifications on the solutions so that it becomes better and better.
4. Repeat step 3 until we don't find a situation in which further transformation has no improvement.

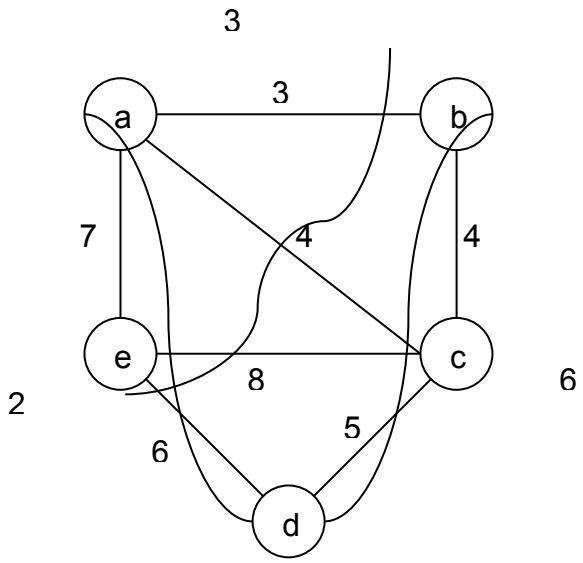
The solution might not be optimal but it definitely discards the worst ones. As the time complexity of such method is concerned if the solution is transformed for relatively high number of times, then it might be better to search all of the solution rather than transforming.

Therefore, one requirement of this approach is that, the set of transformations should be restricted to a small set. The transformations are called as 'Local Transformation', the search for solution within small set of transformation is known as 'local search' and the algorithms that use this method of solution finding are known as 'local search algorithm'.

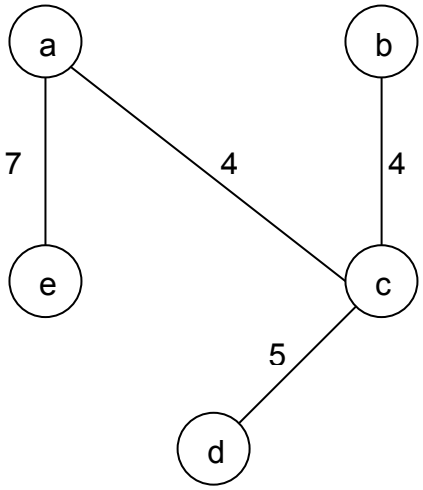
Example: Minimum spanning tree

The general idea behind local search approach to construct minimum spanning tree is that;

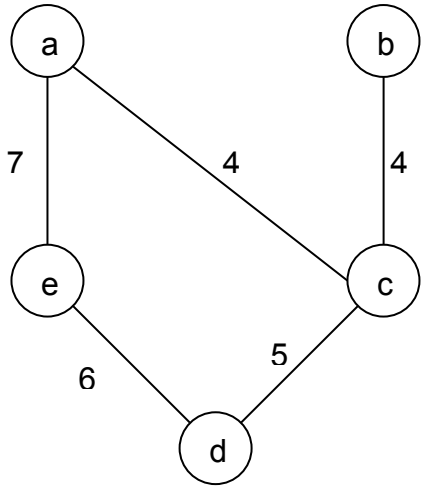
1. Find minimum path between the nodes.
2. Add accordingly.
3. Try to add using other edge that is not in the current spanning tree and that makes unique cycle.
4. Then delete an edge with highest cost.



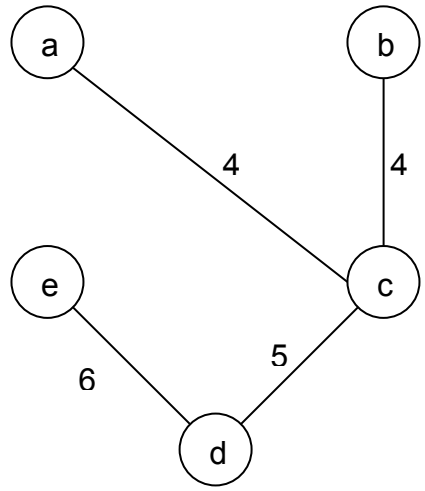
At first, we want to have following tree;



Now, let's add edge (d,e) and let's search for the cycle.



If we remove the edge (a,e) then, it will form;



Here, the cost of tree is reduced from 20 to 19.