Introduction to Algorithms

An Algorithm is a sequence of steps that describe how a problem can be solved. Every computer program that ends with a result is basically based on an Algorithm. Algorithms, however, are not just confined for use in computer programs, these can also be used to solve mathematical problems and on many matters of day-to-day life. Based on how they function, we can divide Algorithms into multiple types. Let’s take a look at some of the important ones.

Typical steps in the development of algorithms:

1. Problem definition
2. Development of a model
3. Specification of the algorithm
4. Designing an algorithm
5. Checking the correctness of the algorithm
6. Analysis of algorithm
7. Implementation of algorithm
8. Program testing
9. Documentation preparation

Types of Algorithm

There are many types of Algorithms but the fundamental types of Algorithms are:

1. Recursive Algorithm

This is one of the most interesting Algorithms as it calls itself with a smaller value as inputs which it gets after solving for the current inputs. In more simpler words, It’s an Algorithm that calls itself repeatedly until the problem is solved.

Problems such as the Tower of Hanoi or DFS of a Graph can be easily solved by using these Algorithms.

For example, here is a code that finds a factorial using a recursion Algorithm:

```
Fact(y)

If y is 0

return 1

return (y*Fact(y-1))  /* this is where the recursion happens*/
```

2. Divide and Conquer Algorithm

This is another effective way of solving many problems. In Divide and Conquer algorithms, divide the algorithm into two parts, the first parts divides the problem on hand into smaller subproblems of the same type. Then on the second part, these smaller problems are solved and then added together (combined) to produce the final solution of the problem.

Merge sorting and quick sorting can be done with divide and conquer algorithms. Here is the pseudocode of the merge sort algorithm to give you an example:

```
MergeSorting(ar[], l,  r)

If r > l

    1.  Find the mid-point to divide the given array into two halves:

middle m = (l+r)/2

2.      Call mergeSorting for the first half:

Call mergeSorting(ar, l, m)

3.      Call mergeSorting for the second half:

Call mergeSorting(ar, m+1, r)

4.      Merge the halves sorted in step 2 and 3:

Call merge(ar, l, m, r)
```

3. Dynamic Programming Algorithm

These algorithms work by remembering the results of the past run and using them to find new results. In other words, dynamic programming algorithm solves complex problems by breaking it into multiple simple subproblems and then it solves each of them once and then stores them for future use.

Fibonacci sequence is a good example for Dynamic Programming algorithms, you can see it working in the pseudo code:

```
Fibonacci(N) = 0                                    (for n=0)

= 0                                                 (for n=1)

= Fibonacci(N-1)+Finacchi(N-2)                      (for n>1)
```

4. Greedy Algorithm

These algorithms are used for solving optimization problems. In this algorithm, we find a locally optimum solution (without any regard for any consequence in future) and hope to find the optimal solution at the global level.

The method does not guarantee that we will be able to find an optimal solution.

The algorithm has 5 components:

- The first one is a candidate set from which we try to find a solution.
- A selection function which helps choose the best possible candidate.
- A feasibility function which helps in deciding if the candidate can be used to find a solution.
- An objective function which assigns value to a possible solution or to a partial solution
- Solution function that tells when we have found a solution to the problem.

Huffman Coding and Dijkstra's algorithm are two prime examples where Greedy algorithm is used.

In Huffman coding, The algorithm goes through a message and depending on the frequency of the characters in that message, for each character, it assigns a variable length encoding. To do Huffman coding, we first need to build a Huffman tree from the input characters and then traverse through the tree to assign codes to the characters.

5. Brute Force Algorithm

This is one of the simplest algorithms in the concept. A brute force algorithm blindly iterates all possible solutions to search one or more than one solution that may solve a function. Think of brute force as using all possible combinations of numbers to open a safe.

Here is an example of Sequential Search done by using brute force:

```
Algorithm S_Search (A[0..n], X)

A[n] ← X

i ← 0

While A [i] ≠ X do

i ← i + 1

if  i < n return i

else  return -1
```

6. Backtracking Algorithm

Backtracking is a technique to find a solution to a problem in an incremental approach. It solves problems recursively and tries to get to a solution to a problem by solving one piece of the problem at a time. If one of the solutions fail, we remove it and backtrack to find another solution.

In other words, a backtracking algorithm solves a subproblem and if it fails to solve the problem, it undoes the last step and starts again to find the solution to the problem.

N Queens problem is one good example to see Backtracking algorithm in action. The N Queen Problem states that there are N pieces of Queens in a chess board and we have to arrange them so that no queen can attack any other queen in the board once organized.

Now let’s take a look at SolveNQ algorithm and Check Valid functions to solve the problem:

CheckValid(Chessboard, row, column)

```
Start

If there is a Queen at on the left of the current column then return false

If the queen is at upper-left diagonal, then return false

If the queen is at lower-left diagonal, then return false

Return true

End
```

SolveNQ(Board, Column)

```
Start
If all columns are full then return true
For each row present in the chess board
Do
If, CheckValid( board, x, column), then
Set the queen at cell (x, column) on the board
If SolveNQ(board, column+1) = True, then return true.
Else, remove the queen from the cell ( x, column) from board
Done
Return false
End
```

Algorithm Complexity

Big-O notation is the prevalent notation to represent algorithmic complexity. It gives an upper bound on complexity and hence it signifies the worst-case performance of the algorithm. With such a notation, it's easy to compare different algorithms because the notation tells clearly how the algorithm scales when input size increases. This is often called the *order of growth*.

Constant runtime is represented by O(1)O(1); linear growth is O(n)O(n); logarithmic growth is O(logn)O(logn); log-linear growth is O(nlogn)O(nlogn); quadratic growth is O(n2)O(n2); exponential growth is O(2n)O(2n); factorial growth is O(n!)O(n!). Their orders of growth can also be compared from best to worst: O(1)<O(logn)<O($\sqrt{\boxed{n}}$)<O(n)<O(nlogn)<O(n2)<O(n3)<O(2n)<O(10n)<O(n!)O(1)<O(logn)<O(n)<O(n)<O(nlogn)<O(n2)<O(n3)<O(2n)<O(10n)<O(n!)

In complexity analysis, only the dominant term is retained. For example, if an algorithm requires 2n3+logn+42n3+logn+4 operations, its order is said to be O(n3)O(n3) since 2n32n3 is the dominant term. Constants and scaling factors are ignored since we are concerned only about asymptotic.

Audrey Nasar gives formal definitions of Big-O. Wikipedia lists orders of common functions.

- What does it mean to state best-case, worst-case and average time complexity of algorithms?

  Let's take the example of searching for an item sequentially within a list of unsorted items. If we're lucky, the item may occur at the start of the list. If we're unlucky, it may be the last item in the list. The former is called *best-case complexity* and the latter is called *worst-case complexity*. If the searched item is always the first one, then complexity is O(1)O(1); if it's always the last one, then complexity is O(n)O(n). We can also calculate the *average complexity*, which will turn out to be O(n)O(n). The term "complexity" normally refers to worst-case complexity.

  Mathematically, different notations are defined (example is for linear complexity):

- Worst-case or upper bound: Big-O: O(n)O(n)

- Best-case or lower bound: Big-Omega: $\Omega(n)\Omega(n)$
- Average-case: Big-Theta: $\Theta(n)\Theta(n)$
  As an example, Quicksort's complexity is $\Omega(nlogn)\Omega(nlogn)$, $\Theta(nlogn)\Theta(nlogn)$ and $O(n2)O(n2)$.

  There's also *amortized complexity* in which complexity is calculated by averaging over a sequence of operations.

- Why should we care about an algorithm's performance when processors are getting faster and memories are getting cheaper?

  Complexity analysis doesn't concern itself with actual execution time, which depends on processor speed, instruction set, disk speed, compiler, etc. Likewise, the same algorithm written in assembly will run faster than in Python. Programming languages, hardware and memories are external factors. Complexity is about the algorithm itself, the way it processes the data to solve a given problem. It's a software design concern at the "idea level".

  It's possible to have an inefficient algorithm that's executed on high-end hardware to give a result quickly. However, with large input datasets, the limitations of the hardware will become apparent. Thus, it's desirable to optimize the algorithm first before thinking about hardware upgrades.

  Suppose your computer can process 10,000 operations/sec. An algorithm of order $O(n4)O(n4)$ would take 1 sec to process 10 items but more than 3 years to process 1,000 items. Comparatively, a more efficient algorithm of order $O(n2)O(n2)$ would take only 100 secs for 1,000 items. With even larger inputs, better hardware cannot compensate for algorithmic inefficiency. It's for this reason algorithmic complexity is defined in terms of asymptotic behaviour.

- Are there techniques to figure out the complexity of algorithms?

  Instead of looking for exact execution times, we should evaluate the number of high-level instructions in relation to the input size.

A single loop that iterates through the input is linear. If there's a loop within a loop, with each loop iterating through the input, then the algorithm is quadratic. It doesn't matter if the loops process only alternative items or skip a fixed number of items. Let's recall that complexity ignores constants and scaling factors. Likewise, a loop within a loop, followed by another loop, is quadratic, since we need to consider only the dominant term.

A recursive function that calls itself n times is linear, provided other operations within the function don't depend on input size. However, a recursive implementation of Fibonacci series is exponential.

A search algorithm that partitions the input into two parts and discards one of them at each iteration, is logarithmic. An algorithm such as Mergesort that partitions the input into halves at each iteration, plus does a merge operation in linear time at each iteration, has a log-linear complexity.

- If an algorithm is inefficient, does that mean that we can't use it?

  Polynomial complexity algorithms of order O(nc)O(nc), for c > 1, may be acceptable. They can be used for inputs up to thousands of items. Anything exponential can probably work for only inputs less than 20. Algorithms such as Quicksort that have complexity of O(n2)O(n2) rarely experience worst-case inputs and often obey Θ(nlogn)Θ(nlogn) in practice. In some case, we can preprocess the input so that worst-case scenarios don't occur. Likewise, we can go with sub-optimal solutions so that complexity is reduced to polynomial time.

  In practice, a linear algorithm can perform worse than a quadratic one if large constants are involved and n is comparable to these constants. It's also important to analyze every operation of an algorithm to ensure that non-trivial operations are not hidden or abstracted away within libraries.