Algorithm: Types and Classification

by Manohar Prabhu - Sunday, September 04, 2011

https://gonitsora.com/algorithm-types-and-classification/

The speed of an algorithm is measured in terms of number of basic operations it performs.

Consider an algorithm that takes N as input and performs various operations.

The corelation between number of operations performed and time taken to complete is as follows-

(consider N as 1,000 and speed of processor as 1 Ghz)

Problem whose running time doesnot depend on input size-- constant time. (very small)

log N operations --- 10 ns

N operations --- 1 us

N*log N opeartions ---- 20 us

$latex N^2$ opeartions------- 1ms

$latex 2^N$ operations ----- $latex 10^{1224}$ seconds

N! operations ---- Unimaginable time.

Hence, it should be noted that every algorithm falls under certain class. From increasing order of growth they are classified as constant time algorithm, logarithmic algorithm, linear time algorithm, polynomial time algorithm and exponential time algorithm.

Formally, we denote the complexity of algorithm using asymptotic notation ?(n) [read Theta of n]

There are basically 3 asymptotic notation used. ? (theta), O (Big O), ? (omega).

Mathematically, these are defined as follows:-

For a given function g(n), we denote by ?(g(n)) the set of functions

?(g(n)) = {f(n) : there exist positive constants c1, c2, and n0 such that 0 ? c1g(n) ? f(n) ? c2g(n) for all n ? n0}.

For a given function g(n), we denote by O(g(n)) the set of functions

O(g(n)) = {f(n): there exist positive constants c and n0 such that 0 ? f(n) ? cg(n) for all n ? n0}.

For a given function g(n), we denote by ?(g(n)) the set of functions

?(g(n)) = {f(n): there exist positive constants c and n0 such that 0 ? cg(n) ? f(n) for all n ? n0}.

So what does this mean?

Informally, O(g(n)) establishes an upper bound on the function. This is used to denote the worst case runtime of an algorithm.

?(g(n)) defines two functions that bound the function g(n) from both top and bottom for appropriate values for constants c1,c2, n0. This is used to denote the average runtime of an algorithm.

?(g(n)) defines a lower bound of the function. We can use it to indicate the Best case runtime of an algorithm.

We will use these notations to indicate the time complexity of algorithms that will be discussed later.

Different types of algorithms:-

Every algorithm falls under a certain class.

Basically they are-

1) Brute force

2) Divide and conquer

3) Decrease and conquer

4) Dynamic programming

5) Greedy algorithm

6) Transform and conquer

7) Backtracking algorithm

Brute force algorithm:-

Brute force implies using the definition to solve the problem in a straightforward manner.

Brute force algorithms are usually the easiest to implement, but the disadvantage of solving a problem by brute force is that it is usually very slow and can be applied only to problems where input size is small.

Divide and conquer algorithm:-

In divide and conquer method, we divide the size of a problem by a constant factor in each iteration. This means we have to process lesser and lesser part of the original problem in each iteration. Some of the fastest algorithms belong to this class. Divide and conquer algorithms have logarithmic runtime.

Decrease and conquer algorithm:-

This kind of problem is same as divide and conquer, except, here we are decreasing the problem in each iteration by a constant size instead of constant factor.

Dynamic programming:-

The word 'dynamic' refers to the method in which the algorithm computes the result. Sometimes, a solution to the given instance of problem depends on the solution to smaller instance of sub-problems. It exhibits the property of overlapping sub-problems. Hence, to solve a problem we may have to recompute same values again and again for smaller sub-problems. Hence, computing cycles are wasted.

To remedy this, we can use dynamic programming technique. Basically, in dynamic programming, we “remember” the result of each sub-problem. Whenever we need it, we will use that value instead of recomputing it again and again.

Here, we are trading space for time. i.e. - we are using more space to hold the computed values to increase the execution speed drastically.

A good example for a problem that has overlapping sub-problem is the relation for Nth Fibonacci number.

It is defined as F(n)= F(n-1) + F (n-2) .

Note that the Nth Fibonacci number depends on previous two Fibonacci number.

If we compute F(n) in conventional way, we have to calculate in following manner

The similar colored values are those that will be calculated again and again. Note that F(n-2) is computed 2 times, F(n-3) 3 times and so on ... Hence, we are wasting a lot of time. In fact, this recursion will perform $latex 2^N$ operations for a given N, and it is not at all solvable for N>40 on a modern PC within at least a year.

The solution to this is to store each value as we compute it and retrieve it directly instead of re calculating it. This transforms the exponential time algorithm into a linear time algorithm.

Hence, dynamic programming is a very important technique to speed up the problems that have overlapping sub problems.

Greedy algorithm:-

For many problems, making greedy choices leads to an optimal solution. These algorithms are applicable to optimization problems.

In a greedy algorithm, in each step, we will make a locally optimum solution such that it will lead to a globally optimal solution. Once a choice is made, we cannot retract it in later stages.

Proving the correctness of a greedy algorithm is very important, since not all greedy algorithms lead to globally optimum solution.

For ex- consider the problem where you are given coins of certain denomination and asked to construct certain amount of money in inimum number of coins.

Let the coins be of 1, 5, 10, 20 cents

If we want change for 36 cents, we select the largest possible coin first (greedy choice).

According to this process, we select the coins as follows-

20

20 + 10

20 + 10 + 5

20 + 10 + 5 + 1 = 36.

For coins of given denomination, the greedy algorithm always works.

But in general this is not true.

Consider the denomination as 1, 3, 4 cents

To make 6 cents, according to greedy algorithm the selected coins are 4 + 1 + 1

But, the minimum coins needed are only 2 (3 + 3)

Hence, greedy algorithm is not the correct approach to solve the 'change making' problem.

Infact, we can use dynamic programming to arrive at optimal solution to this problem.

Transform and conquer:-

Sometimes it is very hard or not so apparent as to how to arrive at a solution for a particular problem.

In this case, it is easier to transform the problem into something that we recognize, and then try to solve that problem to arrive at the solution.

Consider the problem of finding LCM of a number. Brute force approach of trying every number and seeing if it is the LCM is not the best approach. Instead, we can find the GCD of the problem using a very fast algorithm known as Euclid's algorithm and then use that result to find the LCM as LCM (a , b) = (a * b) / GCD (a , b)

Backtracking algorithm:-

Backtracking approach is very similar to brute force approach. But the difference between backtracking and brute force is that, in brute force approach, we are generating every possible combination of solution and testing if it is a valid solution. Whereas, in backtracking, each time you generate a solution, you are testing if it satisfies all condition, and only then we continue generating subsequent solutions, else we will backtrack and go on a different path of finding solution.

A famous example to this problem is the N Queens problem. According to the problem, we are given a N X N sized chessboard. We have to place N queens on the chessboard such that no queens are under attack from any other queen.

We proceed by placing a queen in every column and appropriate row. Every time we place a queen, we check whether it is under attack. If so, then we will choose a different cell under that column. You can visualize the process like a tree. Each node in the tree is a chessboard of different configuration. At any stage if we are unable to proceed, then we backtrack from that node and proceed by expanding other nodes.

An advantage of this method over brute force is that the numbers of candidates generated are very less compared to brute force approach. Hence we can isolate valid solutions quickly.

Ex- for an 8 X 8 chess board, if we follow brute force approach, we have to generate 4,426,165,368 solutions and test each of them. Whereas, in backtracking approach, it gets reduced to 40,320 solutions.

In the next article, we will discuss application of different algorithms.

Author:- *Manohar Prabhu.*
