Introduction to Algorithms

Time complexity, Asymptotic notations, Techniques- Divide and Conquer, Dynamic Programming, Greedy algorithms, Backtracking, Branch and Bound

Algorithm

- An algorithm is any well defined computational procedure that takes some values or set of values as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into output
- An algorithm is a finite set of instructions/steps that if followed accomplishes a particular task
- An algorithm must have the following:
  - Zero or more input
  - At least one output
  - Definiteness – each instruction is clear and unambiguous
  - Finiteness – algorithm must terminate after finite number of steps
  - Effectiveness – every instruction must be basic

Analysis of Algorithms

- Predicting the resources that the algorithm requires
- Two types of resources are generally considered – time and space
- In most cases, we are interested in finding the time complexity or running time of the algorithm
- Though space complexity becomes an important issue in solving problems where we have to handle large amounts of data- (the case with most of the data science related problems now-a-days)

Analysis of Algorithm

- The purpose of analysis of algorithms is two fold:
    - To obtain estimates of bounds on the storage or run-time which the algorithms needs to successfully process the input
    - To find an efficient algorithm for solving a particular problem
- There are two different ways of analyzing an algorithm:
    - Priori analysis
    - Posteriori analysis
- Space Complexity: amount of memory that the algorithm needs to run to completion
- Time Complexity: amount of computer time that the algorithm needs to run to completion

Time Complexity Analysis

- The time complexity T(p) of an algorithm p can be defined in terms of the number of steps the algorithm takes to run to completion
- The number of steps taken by an algorithm is proportional to the problem size
- For a given problem size, the time complexity can be defined in terms of:
    - number of operations performed
    - time of each operation
    - frequency of each operation
- It is assumed that each operation takes more or less same time to execute. Let this time be t then, an algorithm with n steps will take txn time to run
- The goal is to find the upper and lower bound on this run time

Asymptotic Notations

- asymptotic notations describe the bounds on the run time of an algorithm in terms of some function of the problems size
- Thus, asymptotic notations essentially find out a functional relationship between the problems size and time required to solve the problem using a particular algorithm
- These notations try to give an upper or lower bound (or both) on the run time of an algorithm
- We will mainly focus on three different types of notations $\mathrm{O}, o, \theta, \omega, \Omega$
- Among these Big-O notation is most commonly used

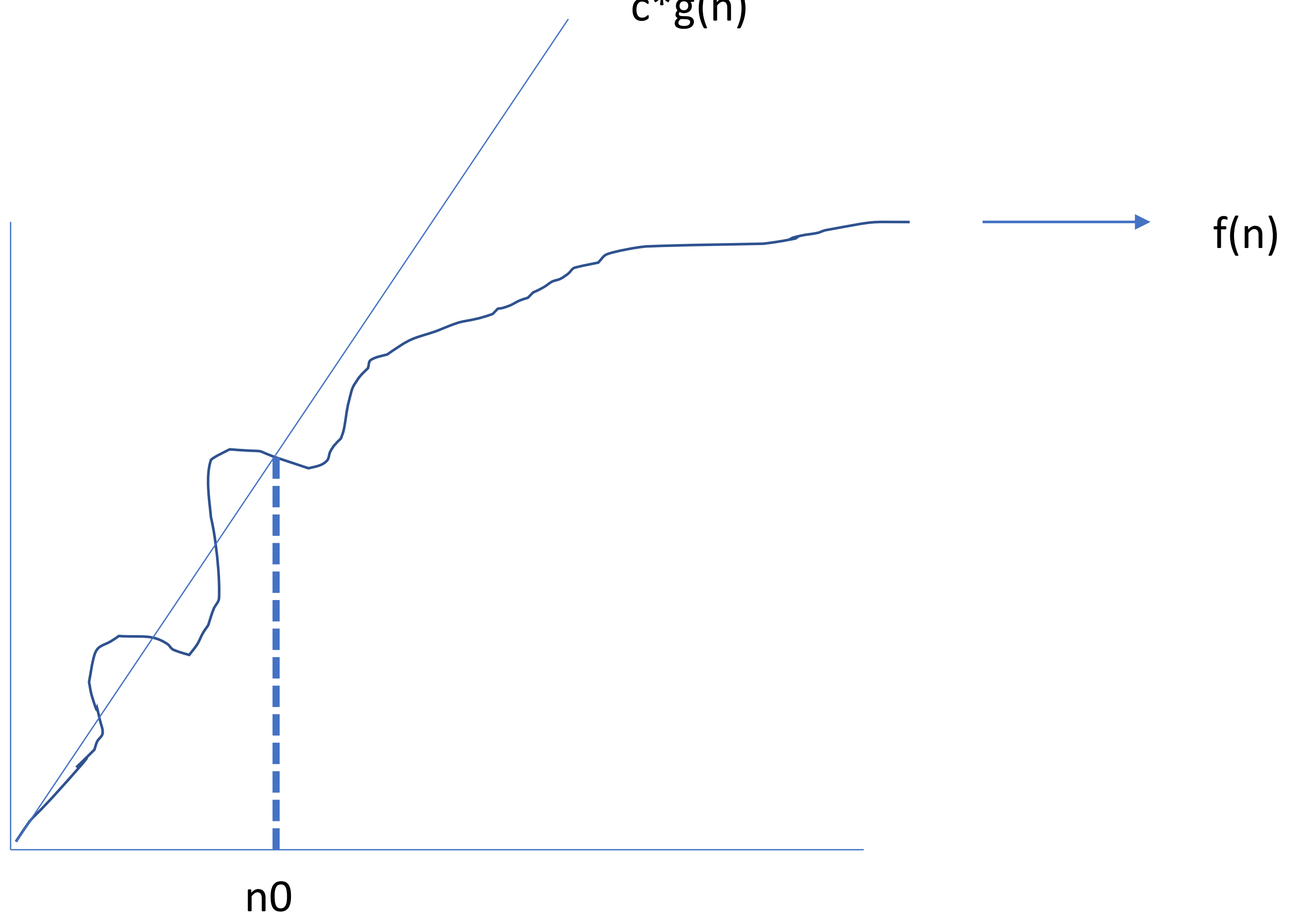
Big-O Notation

- For a given function f(n) we define order of g(n) to be the set of functions such that

  f(n) = O(g(n))   if and only if ∃ positive constants c and n0 such that

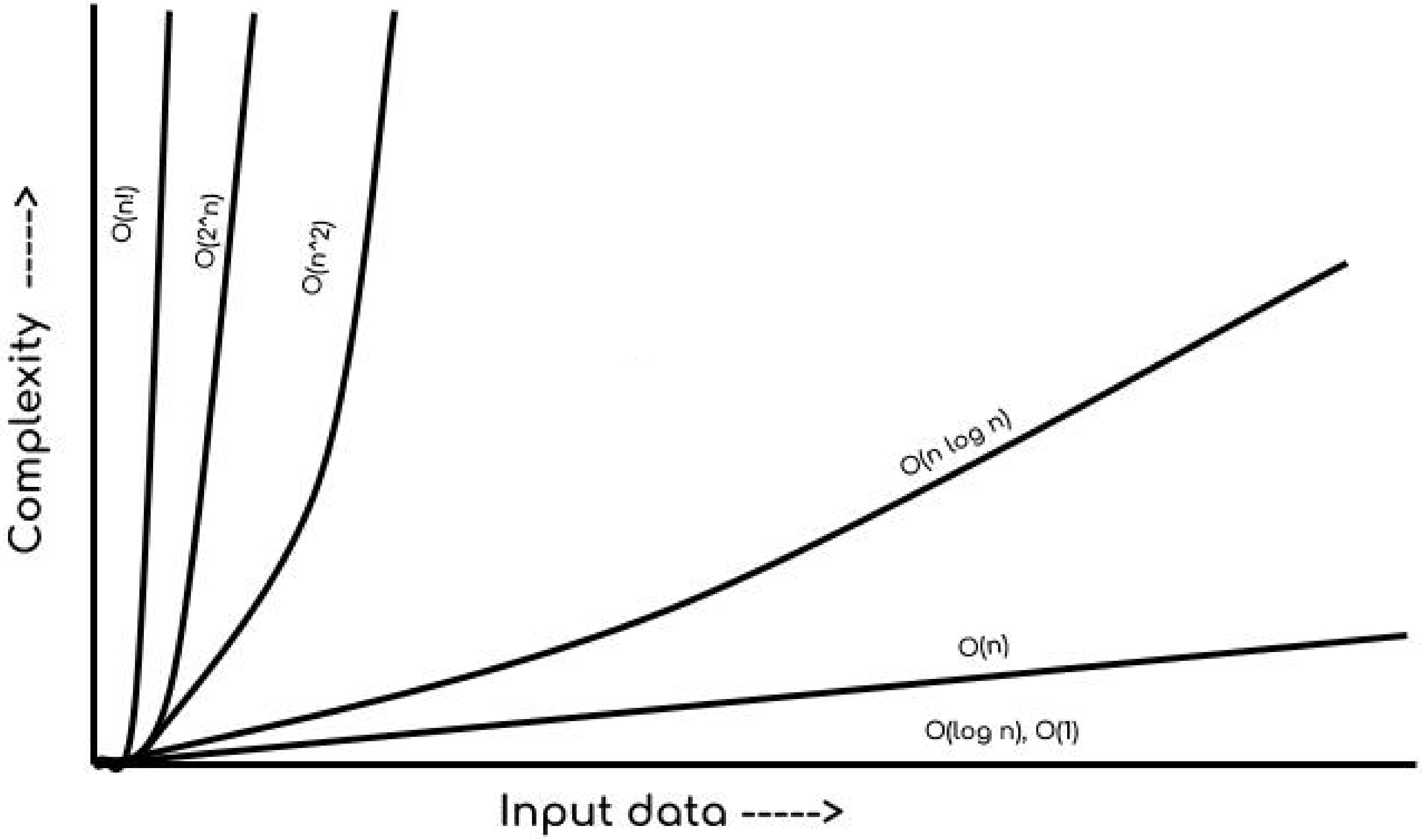
    f(n) <= c*g(n)   ∀   n>=n0

- f(n) essentially provides an upper bound on g(n) i.e. value of g(n) will always be less than f(n)
- For lower values of n, this might not be true but for sufficiently large n, the value is always lower than f(n)
- Establishes lowest upper bound


c*g(n)
f(n)
n0

Examples: Big-O

- 3n+3 = O(n) {c = 4: 4n n0 >= 3}
- $10n^2 + 4n + 2$ = O(n^2) $\{\leq 11n^2 \quad \forall n \geq 5\}$
- $6 * 2^n + n^2$ = O(2^n) $\{\leq 7 * 2^n \quad \forall\, n \geq 4\}$
- Here, n is the problem size
- It indicates the number of inputs to the algorithm
- O(1) or O(p) where p is a constant, indicates that the algorithm takes constant amount of time no matter what the problem size is
- That is, the algorithm is independent of the problem size


Complexity ----->
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Input data ----->

- The statement f(n) <= g(n) only says that g(n) is an upper bound on f(n) for all values of n>= n0, but it is silent about how good this upper bound is.

- But, for the statement f(n) <= g(n) to be informative, the function g(n) should be as small a function of n as one can come up with

(Big)- Ω Notation

- For a given function f(n) we define Ω(g(n)) to be the set of functions such that

  f(n) = Ω(g(n))   if and only if ∃ positive constants c and n0 such that

  f(n) >= c*g(n)   ∀   n>=n0

- g(n) essentially provides a lower bound on f(n) i.e. value of f(n) will always be greater than g(n)
- For lower values of n, this might not be true but for sufficiently large n, the value is always greater than f(n)
- This establishes highest lower bound

Examples – Big Ω

- 3n+3 = Ω(n) as 3n+3 >= 3n $\forall n$
- $10n^2 + 4n + 2$ = $\Omega(n^2)$ $as\ 10n^2 + 4n + 2 \geq 10n^2 \quad \forall n$
- $6 * 2^n + n^2$ = $\Omega(2^n)$ as $6 * 2^n + n^2 \geq 6 * 2^n \ \forall n$
- bound is always chosen as close as possible
- The function f(n) is only a lower bound on g(n) but for the statement f(n) = $\Omega(g(n))$ to be informative g(n) should be as large a function of n as possible

$\theta - Notation$

f(n) = $\theta(g(n)) \quad iff$

$\exists \quad positive\ constant \quad c1, c2\ and\ n0\ such\ that$

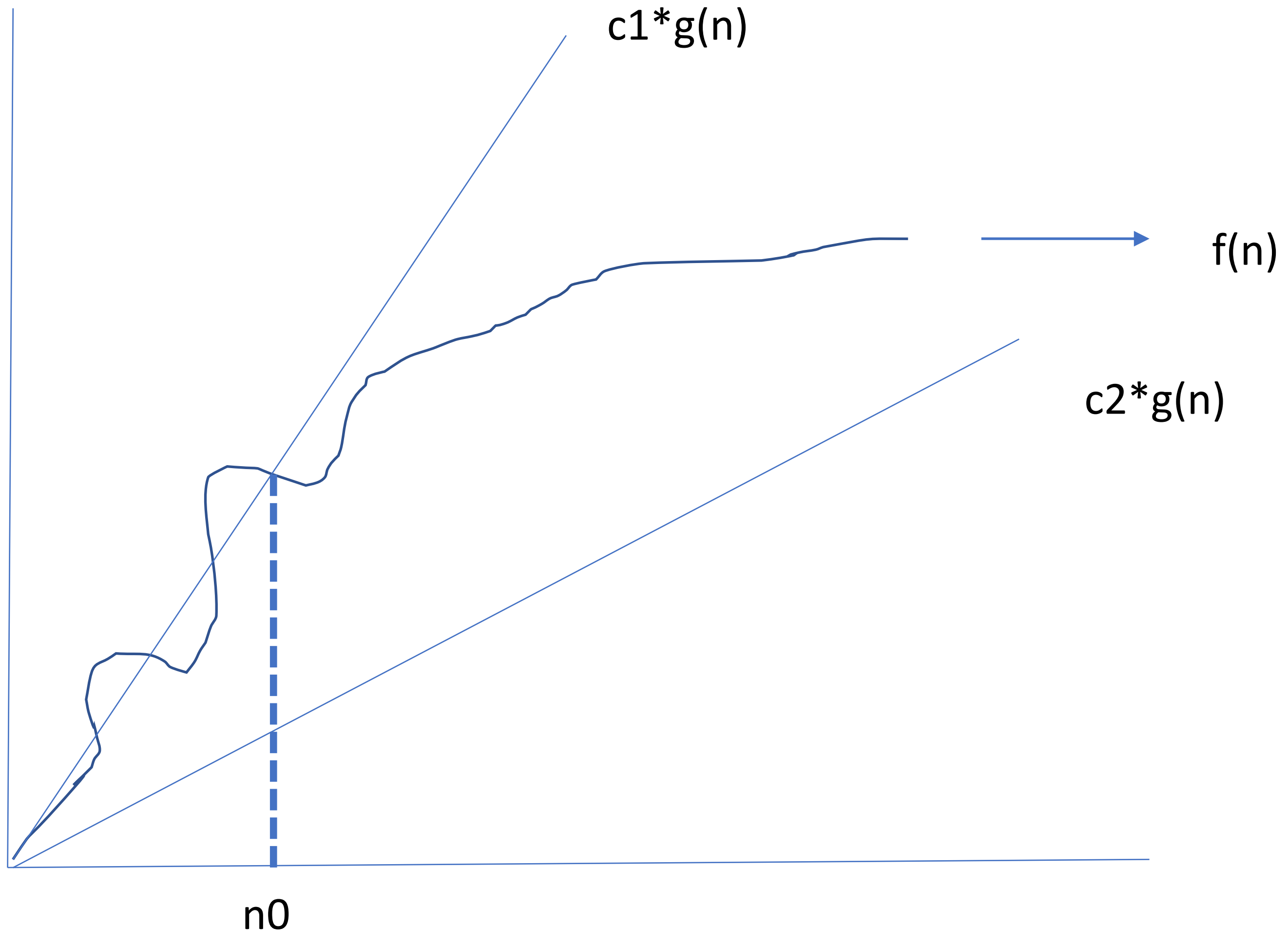
$$\mathrm{c1} * \mathrm{g(n)} \leq f(n) \leq c2 * g(n) \qquad \forall\, n \geq n0$$

- establishes both upper and lower bound

➢ $3n \leq \ 3n + 2 \ \leq \ 4n \qquad\qquad \forall n0 \geq 2$

➢ $3n + 2 \ = \ \theta(n)$

➢ This method also establishes tight bound


c1*g(n)
f(n)
c2*g(n)
n0

o-Notation

f(n) = o(g(n)) iff 0<= f(n) < c*g(n) $\forall\, n \geq n0$

or

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Establishes lose upper bound

Examples: o-Notation

- $3n + 2 = o(\mathrm{nlog}\, n)$

  $= o(n^2)$

  $= o(2^n)$

  $\neq o(n)$

We always try to find lose bound not tight

$\omega - Notation$

f(n) = ω(g(n))   iff   0 < c*g(n) < f(n) $\forall$ $n \geq n0$

or

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Establishes lose lower bound

Examples - ω Notation

- $$\begin{aligned} 10n^3 + 6 &= \omega(n) \\ &= \omega(n^2) \\ &\neq \omega(n^3) \end{aligned}$$

We always try to find lose lower bound

Comparing Functions

Transitivity:

$$
\begin{array}{lll}
f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) & \text{imply} & f(n) = \Theta(h(n))\,, \\
f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) & \text{imply} & f(n) = O(h(n))\,, \\
f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) & \text{imply} & f(n) = \Omega(h(n))\,, \\
f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) & \text{imply} & f(n) = o(h(n))\,, \\
f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) & \text{imply} & f(n) = \omega(h(n))\,.
\end{array}
$$

Reflexivity:

$$\begin{aligned} f(n) &= \Theta(f(n)) , \\ f(n) &= O(f(n)) , \\ f(n) &= \Omega(f(n)) . \end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$\begin{aligned} f(n) &= O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n)) , \\ f(n) &= o(g(n)) \quad \text{if and only if} \quad g(n) = \omega(f(n)) . \end{aligned}$$

Trichotomy

- For any two real numbers a and b, exactly one of the following holds:
  - a=b
  - a<b
  - a>b
- However, this is not true for asymptotic notation
- It is possible that for a function neither f(n) = O(g(n)) nor f(n) = Ω(g(n)) holds
- E.g. the two functions $n\ and\ n^{1+\sin(n)}$ cannot be compared using asymptotic notation

Time Complexity Analysis

- While carrying out time complexity analysis for an algorithm, we normally work out time for several values of n. This is done because:

  - Asymptotic analysis tells us the behavior only for sufficiently large values of n. For smaller values of n, the run-time may not follow the asymptotic curve. To determine the break-even point beyond which the asymptotic curve is followed, we need to examine the times for several values of n

  - Even in the region where asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve because of the effect of low order terms that are ignored in the asymptotic analysis

Best, Worst and Average case analysis

- When an algorithm is run in minimum time (i.e. minimum number of steps), we call it the Best-Case Execution
- When an algorithm is run in maximum time (i.e. maximum number of steps), we call it Worst-Case Execution
- The average case lies somewhere in between best and worst case

➢ Normally, we report only the worst-case running time for an algorithm due to several reasons:

- The worst case running time of an algorithm is an upper bound on the running time or execution time for any input instance. Knowing this gives us a guarantee that the algorithm will never take time longer than this
- For some algorithms, the worst case fairly often occurs. E.g. Searching for a particular data item in the list when the list doesn’t contain it, the worst case fairly often occurs
- The average case is roughly as bad as the worst case. Moreover, it may not be apparent what constitutes an average case input for a particular algorithm
- The scope of average-case analysis is limited, because it may not be apparent what constitutes an “average” input for a particular problem. Often we assume that all inputs of a given size are equally likely

Polynomial and Non-Polynomial Time Algorithm

- For most of the problems, we often come across the best algorithm for their solutions have computing times that cluster into two groups:
- The first group consists of problems whose solution times are bounded by polynomials of small degree e.g. ordered searching (o(logn)), polynomial evaluation (o(n)), sorting(o(nlogn)) and storing editing (o(mn))
- The second group comprises of problems whose best known algorithms are bounded by non-polynomial time. E.g. TSP(O($n^2 * 2^n$)) and Knap-sack problem (O(2^n))
- We are interested in finding a polynomial time algorithm for any given problem if possible

Decision Problem and Optimization Problem

- Any problem that has only a two-class solution either 0(no) or 1(yes) is called a decision problem
- An algorithm for a decision problem is termed as decision algorithm
- Any problem that involves identification of an optimal value (either maximum or minimum) of a given cost function is known as an optimization problem
- Many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem is solvable in polynomial time

Greedy Algorithms

- It is hard to define exactly what is meant by greedy algorithms
- An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion
- Often more than one greedy algorithms may be designed for a particular problem
- If a greedy algorithm is able to solve a non-trivial problem, it indicates that there exists a local decision rule that can be used to construct optimal solutions for the problem
- In some cases, greedy algorithms can find close to optimal solutions
- In general, it is easy to implement greedy algorithm for almost any problem however, finding cases in which they work well and proving that they work well is a challenge
- Algorithms for finding shortest path and minimum spanning tree are based on greedy technique

Divide and Conquer

- Divide and conquer refers to a class of algorithmic techniques where we break the input into several parts, solve the problem in each part recursively, and then combine the solutions to these sub-parts into an overall solution
- For many problems, this technique can prove to be a simple and yet powerful method
- Analyzing the running time here generally involves solving a recurrence relation
- In many cases, divide and conquer is combined with other algorithm design techniques to give efficient algorithms
- Example: Merge sort algorithm that you will see later is based on divide and conquer technique

Dynamic Programming

- Greedy approach for problem solving is the most natural and intuitive way
- Divide and conquer technique proves helpful in cases where no natural greedy algorithm can be identified
- However, in more complex cases even the divide and conquer technique may not give reasonable improvement over the brute force search
- In such cases, a more powerful and subtle design technique known as dynamic programming is applied
- This technique draws its intuition from divide and conquer and is essentially opposite of greedy technique

- Here, we implicitly explore the space of all possible solutions, by carefully decomposing things into a series of subproblems and then build up correct solutions to larger and larger subproblems
- Thus, we follow a bottom-up approach in solving the problem
- As we try to design correct algorithm by implicitly exploring the solution space, there is always danger of reaching close to a brute-force search, application of this technique therefore requires carefully designing the algorithm
- It should be noted that although dynamic programming works by examining exponentially large set of solutions, it never examines them all explicitly, this careful balancing of limited examination and building solution by combining the subproblems that makes this technique tricky to get used to
- Floyd’s algorithm for finding all pair shortest path is based on dynamic programming

Backtracking

- It is a general technique to find all possible solutions to a given computational problem
- It is mainly used for constraint satisfaction problems
- It builds the solution in an incremental manner and discards a candidate as soon as it realizes that the candidate might possibly not lead to a solution
- In practice, this technique also solves the problem in a recursive manner by satisfying some constraint, a candidate is rejected as soon as it is unable to satisfy the constraint

Branch and Bound

- This technique is used in optimization problem, i.e. for finding the optimal solution to a given problem
- This is the main difference between backtracking and branch and bound
- Here, the candidate solutions are thought to be arranged in the form of a rooted tree and the solution is explored by traversing through the branches of the tree by solving the subsets of problems
- Each branch is first tested for the upper and lower bounds expected on the solution and the branch is discarded if it doesn’t satisfy the bounds
- Branch and bound is mainly used for discrete and combinatoric optimization problems