# Algorithms

**Recall:** If A reduces to B 🡺 A <= B 🡺 A is an easier problem to solve than B

## Reductions

**Types of Reductions:**

* Many-One Reduction (Mapping Reduction): This is where you transform instances of one problem (A) into instances of another (B) so a solution to B can be used to solve A.

Example: Vertex Cover to Independent Set: Given a graph G for Vertex Cover, we can create a complement graph G' (where edges become non-edges, and vice-versa) for Independent Set. A vertex cover in G is an independent set in G', and vice versa.

* Turing Reduction (Oracle Reduction): Here, you imagine you have a magic black box (an oracle) that can solve problem B. You leverage this oracle to solve problem A. It's less restrictive than many-one.

Example: Hamiltonian Cycle to TSP (Traveling Salesperson Problem): You can use an oracle for TSP (find the shortest tour visiting all nodes) to solve the Hamiltonian Cycle problem. Ask the oracle for the shortest tour, if its length equals the number of vertices, then there's a Hamiltonian Cycle.

**Why does this matter?**

* **Complexity Theory:**
  + Reductions help prove how difficult a problem is. If problem A reduces to B, which is known to be hard, then A is likely at least as hard as B. This is a cornerstone of classifying problems based on difficulty (e.g., NP-completeness).
* **Algorithm Design:**
  + By reducing a new problem A to a known problem B, you can leverage existing algorithms for B to solve A indirectly.
* **Classifying Problems:**
  + They help us categorize problems into complexity classes like P, NP, and NP-complete, indicating their relative difficulty.
* **Proving Hardness:**
  + If a problem A is known to be hard (e.g., NP-complete), and problem A reduces to problem B, then B is also at least as hard as A.

**Note:** Not all reductions are two-way, meaning that just because you solve B doesn’t mean you can necessarily solve A. It depends on the problem type. Things that affect solvability:

**Problem A:** Squaring a Number, **Problem B:** Multiplication, **A reduces to B 🡺 A <= B**

A <= B means we can use a solution to B to solve A. Reductions demonstrate that a problem is AT MOST as hard as another problem. (i.e. Squaring is AT MOST as hard as general multiplication)

Notice that this doesn't automatically mean we can efficiently go from B → A. Here's why:

1. **Loss of Information:**
   1. Some reductions can be 'lossy', they might reduce large instances of problem A into smaller, easier instances of Problem B. Reversing them would be impossible without having the lost information.
   2. Sometimes the structure of one problem inherently contains more information than another.
2. **Problem Class:** 
   1. Reductions often help us categorize problems as easy (in P) or hard (NP-complete). Sometimes you can reduce an easy problem to a hard one, but it doesn't give you an efficient way to solve the hard problem.
   2. Complexities of different problems can differ across reductions. (i.e. 2-SAT 🡺3-SAT)
      1. If I find a solution to 3-SAT then I have definitely found a solution to 2-SAT, but this is not true the other way around.

**Reduction Examples**

Vertex Cover 🡺 Independent Set

Independent Set 🡺 Vertex Cover

**Vertex Cover**

* Goal:
  + Find the smallest set of vertices in a graph that "covers" all edges, meaning at least one endpoint of every edge is in the set.
* Importance:
  + Applications in facility location (where to place cameras for full coverage), network security, and more.
* Complexity:
  + NP-complete (hard to find optimal solutions for large graphs)

**Independent Set**

* Goal:
  + Find the largest set of vertices in a graph where no two vertices within the set are connected by an edge.
* Importance:
  + Applications in scheduling (non-conflicting events), social networks, and resource allocation.
* Complexity:
  + Also NP-complete.

3-SAT 🡺 Clique

Clique 🡺 3-SAT

**3-SAT Problem**

* Goal:
  + A Boolean satisfiability problem where you're given a formula in conjunctive normal form (CNF) with exactly three literals per clause. You must determine if there's an assignment of TRUE or FALSE to the variables that makes the entire formula TRUE.

Example: (A ∨ ¬B ∨ C) ∧ (¬A ∨ B ∨ ¬C) is a 3-SAT formula.

* Importance:
  + Foundational problem in theoretical computer science, has implications for many real-world optimization and verification problems.
* Complexity:
  + NP-complete - considered very hard to solve efficiently for large instances.

Create the graph by giving each literal (A and ~A …) a vertex and putting an edge between the vertices in the same clause (A connects to ~B and C, ~B connects to A and C, C connects to A and ~B)

Connect literals from different clauses IFF they are not negations (i.e. A to ~A)

Clique Problem:

* Goal:
  + Given an undirected graph and a number k, the task is to find if there's a clique of size k (a subset of k vertices where every two vertices are connected).
  + Complete sub-graph with k-vertices.

Example: In a social network, finding cliques can represent groups of people who all know each other.

* Importance:
  + Used in various fields like social network analysis, bioinformatics and pattern recognition.
* Complexity:
  + Also NP-complete.

Independent Set 🡺 3-Coloring

Note: This reduction is one-way. While finding an independent set helps with 3-coloring, solving a 3-coloring problem doesn't necessarily give you the optimal independent set size. If you don’t care about being optimal, then the reduction is two-way.

3-Coloring Problem

* Goal:
  + Given a graph, the task is to assign one of three colors to each vertex so that no two adjacent vertices share the same color. Essentially you want to color the graph with three colors such that no two nodes linked by an edge have the same color.
* Importance:
  + Used in resource allocation (assigning tasks to different machines without conflicts), register allocation in compilers, and scheduling with limited resources.
* Complexity:
  + Also NP-complete.

Reduction: Independent Set to 3-Coloring (HOW)

* Transformation:
  + Given a graph G for the Independent Set problem, we construct a new graph G' for 3-coloring:
    - Each vertex in G becomes a vertex in G'.
    - We add edges in G' to connect all vertices that were not connected in G. Then connect all nodes to a new node 'X'.
  + Why ‘X’ ?
* Independent Set in G:
  + If we have an independent set in G, those vertices can all be assigned the same color in G' (let's say red), because they don't share edges. All other nodes in G' can be assigned one of the two remaining colors since they are all connected and share an edge with 'X' which is assigned the third color.
* 3-coloring in G':
  + If G' can be 3-colored, then the vertices assigned the same color as 'X' form an independent set in G, because they are not adjacent in G due to their connection with 'X' in G'.
* Key Insight:
  + There exists an independent set in G if and only if there exists a valid 3-coloring in G'. This is because a set of independent nodes in G can be colored differently in G' (one color for each node), and vice versa, a valid 3-coloring in G' implies an independent set in G.
* Why doesn't it work the other way?
  + Finding an efficient solution to the Independent Set problem doesn't automatically provide an efficient solution to the 3-coloring problem because the reduction only guarantees that a 3-colorable graph G' has an independent set of a certain size, it doesn't help you find a valid coloring for any graph.

## Edit Distance

The Edit Distance Problem calculates the minimum number of changes required to transform one string into another through insertion, deletion, and substitution edits.

**Operations**

* Insertion: Adding a single character into the string.
* Deletion: Removing a single character from the string.
* Substitution: Replacing one character with a different one.

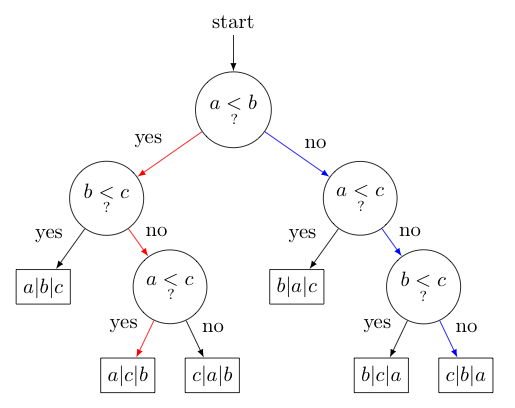
## Sorting

**Why O(N \* Log(N))?**

The best and worse case runtimes for sorting are based on the possible configurations of unsorted data. For a list of N elements, there are N! configurations for the elements, only one of which is a completely sorted list.

**Stirling’s Approximation**

* Runtime: O(N log (N))
* <https://en.wikipedia.org/wiki/Stirling%27s_approximation>



***Sorting Decision Tree for all N! possible orderings of the list.***

Sterling's Approximation plays a crucial role in establishing the lower bound for comparison-based sorting algorithms, demonstrating why algorithms like Merge Sort, Heap Sort, and Quick Sort have an inherent time complexity of Ω(N log N).

1. **The Decision Tree Model:**

When we visualize a comparison-based sorting algorithm, we can represent it as a binary decision tree. Each internal node is a comparison between two elements (e.g., is A > B?), and each leaf node represents one of the possible permutations of the input data.

For a set with N distinct elements, there are N! (N factorial) possible orderings, which means our decision tree must have at least N! leaf nodes to account for every possible outcome.

1. **Height of the Tree & Number of Comparisons:**

The height of a binary tree corresponds to the worst-case number of comparisons needed to sort a given input. This is because, in the worst case, we'd have to traverse from the root of the tree down to the furthest leaf, making a comparison at each level (node).

For a balanced binary tree, the height is log base 2 of the number of leaf nodes. Given we have N! leaf nodes, the height should be log2(N!).

1. **Stirling's Approximation:**

We know that calculating N! for large N is computationally intense. Here’s where Stirling’s Approximation simplifies things. It tells us that N! can be approximated to: √(2πn) \* (n/e)^n

By taking the log base 2 of both sides: log2(N!) ≈ (N log2 N) - (N log2 e)

1. **Lower Bound Derivation:**

From steps 2 and 3, we've determined that the height of the decision tree, which represents the number of comparisons in the worst case, is approximately (N log2 N) - (N log2 e).

Removing the constants and lower order terms yields N log N

The Omega notation (Ω) signifies a lower bound. Thus, we can say that the minimum number of comparisons required for a comparison-based sorting algorithm is Ω(N log N).

**Notes:**

* Height (h): The height of a binary tree is the length of the longest path from the root to a leaf node. Think of it as the number of "levels" in the tree.
* Maximum Number of Leaves: In a perfectly balanced, full binary tree, the maximum number of leaves is directly related to the height of the tree. This maximum number is: 2^h (2 raised to the power of the height)

## RSA Encryption & Decryption

RSA, named after Rivest, Shamir, and Adleman, is a cornerstone of public-key cryptography. This means it uses two keys: a **public key for encryption** and a **private key for decryption**.

**Key Generation Process:**

* **Choosing Primes:**
  + Two distinct large prime numbers, p and q, are chosen. The larger they are, the more secure the encryption.
* **Calculating the Modulus:**
  + The modulus is calculated by multiplying the two primes: n = p \* q.
* **Determining the Totient:**
  + The totient of n, φ(n), is calculated as φ(n) = (p-1)\*(q-1).
* **Selecting the Public Exponent:**
  + An integer e is chosen such that 1 < e < φ(n), and e is co-prime with φ(n) (i.e., they share no common factors other than 1).
* **Calculating the Private Exponent:**
  + The private exponent d is calculated such that it satisfies the equation: d \* e ≡ 1 (mod φ(n)). This means that d is the modular multiplicative inverse of e, modulo φ(n).

**The Public Key:** This comprises of n and e.

**The Private Key:** This is d, which must be kept secret.

**Encryption Formula**

To encrypt a message m, the following formula is used:

**Ciphertext c = m^e (mod n)**

The **message m** is raised to the power of the **public exponent e**, and then the result is taken **modulo n**.

**Decryption Formula**

To decrypt the ciphertext c, the following formula is utilized:

**Message m = c^d (mod n)**

The **ciphertext** is raised to the power of the **private exponent d** and then the result is taken **modulo n**.

**Security of RSA:**

The strength of RSA hinges on two mathematical problems:

* The Factoring Problem: The difficulty of factoring large composite numbers. Given a large n, it is computationally difficult to determine its prime factors p and q.
* The RSA Problem: The difficulty of determining the private exponent d given the public exponent e and the modulus n.

**Additional Notes:**

* **Euler’s Totient Theorem**
  + a^(p - 1) = 1 (mod p)
  + a^((p - 1)\*(q - 1)) = 1 (mod p \* q) : GCD(p, q) = 1
  + **Understanding the Mechanics:**
    - Think of a clock with 'n' hours. Start at 1 and keep multiplying by 'a', but after every multiplication, take the remainder when dividing by 'n' (this is like wrapping around the clock). Euler’s Totient Theorem guarantees that once you've done this φ(n) times, you'll land back at 1, no matter what 'a' you started with (IFF 'a' and 'n' are co-prime).
* **Modular Multiplicative Inverse**
  + An inverse of “a (mod m)” is a number “b” that satisfies the property: a \* b = 1 (mod m)
  + The relationship between the public exponent “e” and the private exponent “d” is as follows: e \* d = 1 (mod (p - 1)\*(q - 1))
    - c = m^e (mod n)
    - m' = c^d (mod n)
    - m’ = m^(c\*d) (mod n) 🡺 (c\*d) = 1 (mod (p - 1)\*(q - 1)) 🡺 e\*d = k \*((p - 1)\*(q - 1)) + 1
    - Euler’s Totient Theorem:
      * M^((p - 1)\*(q - 1)) = 1 (mod p\*q) 🡺 m’ = m^(k\*((p - 1)\*(q - 1))) \* m (mod n)
      * (m^((p - 1)\*(q - 1)))^k \* m (mod n)
      * Recall: x^((p - 1)\*(q - 1)) = 1 (mod n)
      * (1)^k \* m (mod n)

## Convex Hull

Imagine you have a bunch of nails scattered on a flat surface. You want to wrap a rubber band around all the nails, but without any slack or weird dips inwards. The convex hull problem is about finding the shape that the rubber band would make.

So, the goal of the convex hull problem is to find the smallest possible shape, following the convex rule, that encloses all the given points. This shape is called the convex hull, and its corners are the most extreme points of the original set.

**Definitions**

* Convex:
  + In simpler terms, convex means no pointy bits sticking inwards. If you draw a straight line between any two points on the rubber band shape, the entire line segment should also lie inside the shape.
* Hull:
  + This refers to the outer boundary created by the rubber band.

**Key Operations for Solving the Convex Hull Problem Intuitively**

1. Identifying the Extremities:
   1. Intuition:
      1. The outermost points of our nail collection will definitely be part of the rubber band stretch, right? They form the 'corners' of our hull.
   2. How it helps: By identifying these extreme points (like the leftmost, rightmost, topmost, and bottommost), we start to build the framework of our convex hull.
2. Determining Orientation:
   1. Intuition:
      1. Imagine three nails A, B, and C. If you were to walk from A to B to C, would you make a left turn or a right turn? This is about understanding the direction or 'turn' between points.
   2. How it helps:
      1. Knowing the orientation (clockwise or counterclockwise) helps us decide if a point would be inside or outside the rubber band stretch. For a convex hull, we are typically interested in one of the orientations.
3. Line Segment Intersection:
   1. Intuition
      1. Given two line segments formed by points that we believe are on the hull, do they cross each other? This can happen if we've made a mistake and included a point that’s actually inside our rubber band boundary.
   2. How it helps:
      1. Checks can be made to ensure that as we build the hull, no lines are crossing. This ensures we maintain a convex shape.
4. Comparing Angles or Slopes:
   1. Intuition:
      1. If you stand at one nail, at what angle do you see the other nails? Are some directly in your line of sight, or are they off to the side?
   2. How it helps:
      1. Comparing angles or slopes between points allows us to determine which points are relevant to the hull. By sorting points based on their polar angle relative to a starting point, you can imagine "wrapping" the rubber band around the points in a logical order.

**Scenario / Algorithm Notes**

You've correctly identified the leftmost point, A, as part of the hull and are now trying to find the next point to include as you build the hull counterclockwise. We'll label the points we're considering B, C, and D, all of which are to the right of A.

* Step 1: Establish a reference line
  + Action: Draw a line segment from Point A to Point B. This serves as our baseline.
  + Target Orientation We're building the hull counterclockwise, so our target is to make a left turn.
* Step 2: Evaluate the next point.
  + Action We introduce point C into the mix. To determine the orientation of the turn formed by points A, B, and C, imagine yourself walking from A to B and then to C. Did you turn left or right?
  + Target Orientation: We are still aiming for a left turn as we're building the hull counterclockwise. If walking from A to B to C is a right turn then we know point B cannot be part of the hull.
* Step 3: Making decisions.
  + Action: If the turn was to the right (clockwise) when moving from A to B to C, we know B isn’t the next point on our hull because it would create an inward dip. We would then need to check the orientation of A, C, and another point such as D. If ACD makes a left turn then we know we’re moving in the correct direction and can continue building the hull counterclockwise.

**Additional Considerations:**

* Dynamic Target Orientation Once you've gone around and identified all the points on the upper portion of the hull, you will reach the rightmost extremity and will now be working downward. At this point, your target orientation would shift to the right or clockwise to maintain the outward shape of the hull.
* Mathematical Determination: While visualizing turns is helpful for understanding the concept, orientation can also be determined mathematically using the slope between points or the sign of the cross product of vectors created by the points.

**Relevant Equations**

* Slope:
  + m = (y2 - y1) / (x2 - x1)
* Angle(x, y)
  + Arctan(m)
* Polar Angle
  + Arctan2(y, x)
* **Slope vs. Angle:**
  + Slope measures steepness, while the angle provides directionality in relation to the x-axis.
* **Polar Angle:**
  + Great for determining relative positioning around a central point, which ties into some convex hull algorithms.

## Theoretical Computer Science

Theoretical Computer Science is a subset of general computer science and mathematics that focuses on mathematics aspects of computer science such as the theory of computation, formal language theory, lambda calculus, and type theory.

**Fundamentals**

* Automata Theory: The study of abstract machines and the problems they can solve.
  + Finite state machines: Simple models of computation with limited memory.
  + Pushdown automata: Can model computations with a basic stack structure.
  + Turing machines: The theoretical foundation of general-purpose computers.
  + Computability Theory: Focuses on what can and can't be computed.
* Decidability: Whether a theoretical algorithm exists to solve a problem definitively.
  + The Halting Problem: The fundamental example of an undecidable problem.
  + Church-Turing Thesis: Explores the limits of what can be effectively calculated.
* Computational Complexity Theory: Studies the resources (time, space) needed for computation.
  + Time complexity: How the runtime of algorithms scales with input size (Big-O notation).
  + Space complexity: How much memory algorithms require.
  + Complexity classes: P, NP, NP-Complete, and their relationships.

**Algorithms & Data Structures**

* Algorithm Design: Systematic methods for solving computational problems efficiently.
  + Sorting, searching, graph algorithms: Foundational techniques.
  + Greedy algorithms, dynamic programming: Classic algorithm paradigms.
* Data Structures: Ways to organize data for efficient operations.
  + Arrays, lists, stacks, queues: Basic building blocks.
  + Trees, graphs, heaps: Specialized structures for different problem domains.

**Theory of Programming Languages**

* Formal Languages and Grammars: Mathematical models describing language syntax.
  + Regular expressions, context-free grammars: Language types with different expressive power.
* Compilers and Interpreters: How programs are translated and executed.
* Type Systems: Methods to categorize data and enforce correctness in programs.

**Additional Areas**

* Cryptography: Techniques for secure communication.
* Quantum Computation: Explores computation using quantum-mechanical phenomena.
* Information Theory: The quantification, storage, and communication of information.
* Computational Biology: Applying computer science to solve biological problems.
* Computational Geometry: Algorithms for geometric shapes and objects.
* Distributed Computing: Systems of multiple computers working together.

## Additional Observations

Red-Black Tree Visualization: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

The Halting Problem

One question that I had pertained to whether or not a function could be written to determine the asymptotic runtime of a given function, but solving this problem would solve the Halting Problem.

A = Halting Problem

B = Asymptotic Runtime Function

A <= B

Key Point: The ability to determine if a program terminates does not automatically convey information about the speed at which it might terminate.

Reference: https://cs.stackexchange.com/questions/32219/when-problem-a-reduces-to-problem-b-which-problem-is-more-complex