Summations:

$$\sum_{i=1}^{n} 1 = 1_1 + 1_2 + \cdots + 1_n = n$$

$$\sum_{i=m}^{n} 1 = n - m + 1$$

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Finite Geometric Progression:

$$\sum_{k=0}^{n} ar^k (r \neq 0) = \frac{ar^{n+1} - a}{r - 1}, r \neq 1$$

Infinite Geometric Progression:

$$\sum_{k=0}^{\infty} ar^k \ (r \neq 0) = \frac{a}{1 - r}, |r| < 1$$

Asymptotic Notation:

$$\lim_{n \to \infty} \frac{T(n)}{g(n)} = \begin{cases} 0, & T(n) \in O(g(n)) \\ C, & T(n) \in \Theta(g(n)) \\ \infty, & T(n) \in \Omega(g(n)) \end{cases}$$

$\forall n \geq n_0$ for some positive constants c_1, c_2
$T(n) \in O(g(n))$: $T(n) \leq c * g(n)$
$T(n) \in \Omega(g(n))$: $T(n) \geq c * g(n)$
$T(n) \in \Theta(g(n))$: $c_1 g(n) \leq T(n) \leq c_2 g(n)$

Rankings
$O(1), O(\log n), O(n), O(n \log n)$,
$O(n^2), O(n^c), O(c^n), O(n!)$

Recurrence Relations:
A **recurrence relation** is an equation or inequality that describes a function in terms of its value on smaller inputs. They're useful for analyzing the running **time** of recursive algorithms.

Common Recurrences
$T(n) = 2T\left(\frac{n}{2}\right) + n = O(n \log n)$
$T(n) = T\left(\frac{n}{2}\right) + c = O(\log n)$
$T(n) = 2T(n-1) + 1 = O(2^n)$

BSearch: Time = $O(\log_3 n)$, Rec. Rel.: $T(n) = T\left(\frac{n}{3}\right) + C$
$\boldsymbol{BSearch}(Arr, S, E, key)$:
  $if\ S < E$:
    $size = (S + E)//3, p1 = S + size, p2 = S + 2 * size$
    $if\ Arr[p1] == key: return\ p1$
    $if\ Arr[p2] == key: return\ p2$
    $if\ key < Arr[p1]: return\ BSearch(Arr, S, p1, key)$
    $elif\ key > Arr[p2]: return\ BSearch(Arr, p2, E, key)$
    $else: return\ BSearch(Arr, p1, p2, key)$

Master Method:

$a \geq 1, b > 1, f(n) > 0$

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where:
$f(n)$ is a positive polynomial function

a and b are constants

Case 1

$n^d \ll n^{\log_b a}$ implies $T(n) = \Theta(n^{\log_b a})$

Case 2

$n^d == n^{\log_b a}$ implies $T(n) = \Theta(n^d \log n)$

Case 3

$n^d \gg n^{\log_b a}$ implies $T(n) = \Theta(n^d)$

Dynamic Programming:
A **recurrence formula** describes the **strategy** of an algorithm.

Requires **Optimal Substructure** and **Overlapping Solutions**.

Bottom-up vs **Top-down**. **Memoization**.

Fibonacci: naïve $= O(2^n)$, DP $= O(n)$

$$F(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F((n-2), & \text{otherwise} \end{cases}$$

LCS: naïve $= O(2^n)$, DP $= O(n * m)$
Optimal LCS: $O(m + n)$
$\boldsymbol{LongestCommonSubstring}[i, j]$

$$= \begin{cases} 0, & \text{if i=0 or j=0} \\ 1 + LCS[i-1, j-1], & \text{if str1[0] = str2[0]} \\ \max(LCS[i, j-1], LCS[i-1, j]), & \text{if str1[0]} \neq \text{str2[0]} \end{cases}$$

Steps: 1) Identify parameters, 2) Identify subproblem, 3) Define recursive formula, 4) Implement naïve recursive solution, 5) Turn recursive formulation into DP algorithm

Product sum: ($time$ & $space$) DP $= O(n)$
$\boldsymbol{Product\ Sum}$: $f(n)$

$$= \begin{cases} 0, & \text{if i=0} \\ v_i, & \text{if i=1} \\ \max(f(i-1) + v_i, f(i-2) + v_{i-1} * v_i) \end{cases}$$

U. Knapsack: time $= O(nW)$, space $= O(W)$
$\boldsymbol{Unbound\ Knapsack}$: $f(x)$

$$= \begin{cases} 0, & \text{if no i s.t. } w_i \leq x \text{ or if x=0} \\ \max(f(x - w_i) + v_i \end{cases}$$

Rod Cutting: naïve=$O(2^{n-1})$, DP=$O(n^2)$
$\boldsymbol{Rod\ Cutting}$: $f(n)$

$$= \begin{cases} 0, & \text{if i = 0} \\ \max(f(n-i) + price[i-1]) & \text{for } 1 \leq i \leq n \end{cases}$$

Change-making: (time) naive=$O(n^A)$, DP = $O(An)$, (space) DP = $O(n)$

$$f(A) = \begin{cases} 0, & \text{if } A = 0 \\ \min(f[A - C_i] + 1), & \text{for i from 1 to n if } A > C_i \end{cases}$$

0-1 Knapsack: time=$O(nW)$, space=$O(nW)$
0-1 $\boldsymbol{Knapsack}$: $f(x, i)$

$$= \begin{cases} 0, & \text{if i=0 or x=0} \\ \max(v_i + f(x - w_i, i-1), f(x, i-1)) \end{cases}$$

$LCS(str1, str2, memo)$:
  $memo = [[0 * len2 + 1] * len1 + 1]$
  $if\ len1 \leq 0\ or\ len2 \leq 0$:
    $return\ 0$
  $if\ str1[len1 - 1] == str2[len2 - 1]$:
    $res = 1 + LCS(len1 - 1, len2 - 1)$
    $memo[len1][len2] = res, return\ res$
  $else: res = \max(LCS(len1, len2 - 1),\ LCS(len1 - 1, len2))$
    $memo[len1][len2] = res, return\ res$

Backtracking:
$PowerSet(inpList, res, choices)$:
  $if\ not\ inpList$:
    $res.append(choices)$
  $for\ ele\ in\ inpList$:
    $choices.append(inpList.pop(0))$
    $PowerSet(inpList, res, choices)$
    $choices.pop(0)$
    $PowerSet(inpList, res, choices)$

Used to solve problems where we want to **find all possible solutions**.

General Steps:
1) If in final state, do bookkeeping and return
2) Loop through all possible choices:
   a. Make a choice and check constraints
   b. Recurse to smaller problem
   c. Unmake choice

Permutations: $O(n!)$
PowerSet: $O(n * 2^n)$
N-Queens: $O(n!)$
Combination Sum w Repetition: $O(n^k)$
Combination Sum w/o Repetition: $O(n * 2^n)$

Keys: 1) Choice 2) Constraint 3) Goal

Greedy Algorithms:
Make the **best choice available** during each iteration and **don't look back**.

Benefits: Easy to implement and efficient.
Limitations: Does **not** always return optimal solutions.
Hard to design. Difficult to verify.

Requires **Greedy Choice Property** (locally optimal leads to globally optimal) and **Optimal Substructure**.
Huffman Encoding uses a MinHeap to store character frequency. Pop from MinHeap and build encoding tree. $O(n \log n)$.

Graphs:
Adjacency List: space = $\Theta(|V|^2)$, access time = $\Theta(1)$
Adjacency Matrix: space = $\Theta(|V| + |E|)$, access time = $\Theta(|E|)$

Breadth-First Search explores neighbors in the order they are visited. Uses a queue. Good for finding shortest paths.
Depth-First Search explores the most recently discovered vertices first. Uses a stack. Good for topological sorting, Traveling Salesman.

Prim's Algorithm:
Naïve time = $O(|V||E|)$, MinHeap time = $O(|E| \log|V|)$
$Prims(G)$:
  $result, visited = [\ ],\ [\ ]$
  $while\ len(visited) < |V|$:
    Find edge (a,b) where a is in visited and b is not, and (a,b) is minimal
    $result.append((a, b))$
    $visited.append(b)$
  $return\ result$

A **spanning tree** of a connected, undirected graph G is a tree that contains every vertex of G and every edge in the spanning tree is also an edge of G. A **minimum spanning tree** is the spanning tree of a graph with the least weight. Graphs can have multiple MSTs.
A complete graph with $|V|$ vertices has $|V|^{|V|-2}$ spanning trees.

Topological Sort sorts DAGs based on dependency flow. Solutions are not unique.
Time = $O(|V| + |E|)$
$\boldsymbol{TopoSort}(G)$:
  $result = [\ ]$
  $while\ (unvisited\ nodes)$:
    $Helper(curr\_node)$
  $return\ result.reverse()$

  $Helper(curr\_node)$:
    $mark\ curr\_node\ as\ visited$
    $for\ node\ in\ curr\_node.neighbors$:
      $if\ node\ not\ in\ visited$:
        $Helper(node)$
    $result.append(curr\_node)$

Dijkstra's Algorithm is a greedy algorithm used for finding the shortest path from a single source in a nonnegatively weighted graph. **Recurrence relation**:
$dist[v] = \min(dist[v], weight[u, v] + dist[u])$
MinHeap Time = $O((|V| + |E|) \log|V|)$, Array Time = $O(|V|^2)$
$\boldsymbol{Dijkstra}(G, s)$:
  Add all nodes to minheap w distance of infinity
  Set source node's distance to 0 in minheap
  $visited = [\ ]$
  $while\ queue$:
    $curr\_node = queue.popleft()$
    $visited.append(curr\_node)$
    $for\ v\ in\ curr\_node.neighbors$:
      $dist\_v = \min(dist\_v, weight(curr\_node, v) + dist\_curr)$
      Set v's distance to dist_v in minheap

Kruskal's Algorithm:
Naïve time = $O(|V||E|)$, disjoint set/union find time = $O(|E| \log|V|)$
$Kruskals(V, E)$:
  Sort E by increasing weight
  $for\ v\ in\ V$:
    $makeSet(v)$
  $MST = [\ ]$
  $for\ (u, v)\ in\ sorted\_E$:
    $if\ findSet(u) \neq findSet(v)$:
      $MST.append((u, v))$
      $Union(u, v)$
  $return\ MST$

P vs NP:
P is the class of all problems that can be both solved and verified in polynomial time.
NP is the class of all problems that can be solved in non-deterministic polynomial time and verified in polynomial time.
NP-Hard is the class of all problems that are at least as hard as the hardest problems in NP. Every problem in NP can be reduced in polynomial time to a problem in NP-Hard.
NP-Complete is the class of all problems that are in NP-Hard and in NP.
Reduction is the process of converting inputs for one problem into equivalent inputs for another problem. Must occur in polynomial time to be useful. Reducing problem X to problem A is denoted by $X \leq_p A$. Always reduce from the known hard problem to the unknown problem. Reduction shows that problem X is **no harder than** problem A.
Decision Problems have a Yes/No answer.

Steps for proving that problem X is NP Complete:
1. Show that X is in NP (is poly time verifiable).
2. Show that known NP-Hard problem A_{Hard} can be reduced to X in polynomial time.
3. Show that if an algorithm exists that solves X, then we can obtain a solution to A_{Hard} from the solution to X using a polynomial time transformation.
An **approximation ratio** shows how close an approximation algorithm is to the optimal solution:
$\rho(n) = \max\left(\frac{C}{C^*}, \frac{C^*}{C}\right)$ where C is the approx. solution and C^* is the optimal solution.
Cook-Levin Theorem showed that Boolean (Circuit) Satisfiability is NP-Hard and thus NP-Complete.
NP-Complete Proofs:
Circuit SAT reduces to **3SAT**.
3SAT reduces to **Independent Set**.
Hamiltonian Cycle reduces to **Traveling Salesperson**.

Approximation Algorithms:
$\boldsymbol{VertexCover}(G)$:
  $solution = \{\ \}, edges =$ all edges in G
  $while\ E\ is\ not\ empty$:
    pick arbitrary edge in E w/ vertices (u,v)
    $solution = union(solution, \{u, v\})$
    remove from E all edges connected to u or v
  $return\ solution$
$\rho(VertexCover) = 2$

$\boldsymbol{NearestNeighborTSP}(G)$:
  pick arbitrary starting vertex
  while there are unvisited vertices:
    go to closest unvisited neighboring vertex
  return to starting vertex
$\boldsymbol{MST\_TSP}(G)$:
  Find MST of G
  Use MST to create a walk using vertices given by MST
  Create Hamiltonian Cycle based on the path
$\rho(MST\_TSP) = 2$