Log Properties:

$\log_a 1 = 0, \quad \log_b a = 1, \quad \log_a x^y = y \log_a x, \quad \log_a xy = \log_a x + \log_a y, \quad \log_a \frac{x}{y} = \log_a x - \log_b y,$

$a^{\log_b x} = x^{\log_b a}, \quad \log_a x = \frac{\log_b x}{\log_b a} = \log_a b \log_b x, \quad x^{\log_{10} y} < x^{\log_2 y}$

Summation Rules:

$$\sum_{i=l}^{u} 1 = \underbrace{1 + \cdots + 1}_{(u-l+1 \text{ times})} = u-l+1 \quad \sum_{i=1}^{n} 1 = n \quad \sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \quad \sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1} n^{k+1} \quad \sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1} (a \neq 1) \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2 \quad \sum_{i=1}^{n} \lg(i) \approx n \lg n$$

Sum Manipulation Rules:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i \quad \sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i \quad \sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+l}^{u} a_i, \text{ where } l \leq m < u \quad \sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1} \quad \sum_{i=0}^{n} a_i = \sum_{i=1}^{n} a_i + a_0$$

Formal $O(n)$: $f(n) \in O(n)$ if $f(n) \leq cg(n) \forall n \geq n_0$
Formal $\Omega(n)$: $f(n) \in \Omega(n)$ if $f(n) \geq cg(n) \forall n \geq n_0$
Formal $\Theta(n)$: If $f(n) \in O(n) \& \Omega(n), f(n) \in \Theta(n)$
Pick C, n_0 such that rules apply.
Rule of Sums: $t_1 n \in O(g_1(n))$ AND $t_2 n \in O(g_2(n)), t_1(n) + t_2(n) \in O(max(1,2))$
Rule of Products: $f_1 \in O(g_1)$ AND $f_2 \in O(g_2) \implies f_1, f_2 \in O(g_1, g_2)$
Other Big-O Rule: $f_n \in O(g(n))$ AND $g_n \in O(h(n)) \implies f(n) \in O(h(n))$
Recurrence Relations: N-1, N-2, N-3... Find general case (n-i), pick i such that n-i results in base case, solve.
For Summations: $= T() + \_\_\_\_ \leftarrow$ Full recursive function
$T()$ = Recursive Call, replace this with full recursive function. $+\_\_\_\_ =$ Carry down by rote.
Power Replacement: $2^k = n \implies$ replace n with 2^k before solving.
Basic Math: $\frac{n}{2} = n^{-1}$

Exp	Const		Log		3rd Root	Sqrt		Lin	Linearith	Quad	Cubic	5th Power	Exp	Exp	!
$(\frac{a}{b})^n, b > a$	6	$\log\log(n)$	$\log(n)$	$(\log(n))^2$	$n^{\frac{1}{3}} + \log(n)$	$\sqrt{n}$	$\frac{n}{\log(n)}$	n	$n\log(n)$	n^2	n^3	$n - n^3 - 7n^5$	$(\frac{a}{b})^n, a > b$	2^n	$n!$
Base < 1			$\ln(n)$		$\sqrt[3]{n}$					$n^2 + \log(n)$			Base $> 1 \& < 2^n$		

$$\lim \text{ for orders of growth: } \lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 \implies t(n) < g(n) \\ c \implies t(n) = g(n) \\ \infty \implies t(n) > g(n) \end{cases} \quad \& \quad \lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t\prime(n)}{g\prime(n)}$$

Sorting Algorithms:
Bubble Sort: Steps through list, swapping adjacent items if they're in wrong order.
Function:
Best Case: $\Theta(n)$ Comparisons, $\Theta(1)$ Swaps
Worst/Avg Case: $\Theta(n^2)$ Comparisons & Swaps, Array in descending order.

Insertion Sort:
Output: Sorted array, from low→high

```
mark first element as sorted
for each unsorted element X
'extract' the element X
for j = lastSortedIndex down to 0
if current element j > X
move sorted element to the right by 1
break loop and insert X here
```

Best Case: $\Theta(n)$: Array is already in increasing order. ($O(n)$ comparisons, $O(1)$ swaps)
Worst & Avg Case: $\Theta(n^2)$: Array always has to swap (array in decreasing order)
Extraction n-1: Remove last element $\Theta(1)$
Extend n from n-1: Linear: $\Theta(n)$. Must still scan entire array to insert new element.

Selection Sort:
Function: Find smallest element during a run.
After run, swap with first unsorted position.
Best/Worst/Avg Case: $\Theta(n^2)$, $\Theta(n)$ swaps.
$\Theta(1)$ Space Complexity
Extraction n-1: $\Theta(n-1) \equiv \Theta(n)$ (find smallest element, remove)
Extend n from n-1: $\Theta(1)$ (everything else already sorted)

Closest-Pair:
Brute-Force Compare distances of all pairs of points, then take smallest. Worst Case: $\Theta(n^2)$
Exhaustive Search:

TSP/Knapsack/Assignment.
Generate all possible solutions, evaluate one-by-one, disqualifying unfeasible ones. Keep track of best ones. Return best ones.

Constant Factor: Binary Search:
Input: Array $[0 \cdots n-1]$ in ascending order, search key K. $m = \lfloor \frac{l+r}{2} \rfloor$

```
l = 0; r = n−1
while(l <= r):
m = (l+r)/2
if K = A[m] return M, Best: search key in middle entry.
else if: K < A[m], r = m−1, search key in left partition
else: l = m+1, search key was in right partition
return −1, Worst: Search key not found;
```

Example Worst Case: Array doesn't include search key. $\Theta(\log(n))$, avg case also $\Theta(\log(n))$
Best Case: Search Key is center element of array $\Theta(1)$

Variable Size Decrease:
Quicksort:
Pick pivot point. Reorder array such that all elements < pivot come before, and > after. (partition)
Recursively do such to all subarrays.
Best/Avg Case:$\Theta(n\log(n))$ Best Case: Array perfectly balanced
Worst Case: $\Theta(n^2)$
If Implemented right, can always avoid worst case: Pick 3 elements, take median. If sorted always take median.

Quickselect: Like Quicksort, but just ignores the half it doesn't need
Pick Pivot, partition data based off of pivot. Recurse into side that contains element looking for.
Best/Avg Case:$\Theta(n)$ pivot is median (perfect)
Worst Case:$\Theta(n^2)$ Pivot is first/last element (Array is already sorted)

Lomuto: Worst Case:$\Theta(n^2)$ if array already in order.

Hoare: Worst Case:$\Theta(n^2)$

DFS (LIFO)
Push onto stack, pop off stack to go back
go as deep as possible before backtracking.
If using AdjMatrix: $\Theta(vertices^2)$
If using AdjList: $\Theta(Vertices + Edges)$

BFS (FIFO)
Root → first layer → second layer → ...
If using AdjMatrix: $\Theta(vertices^2)$
If using AdjList: $\Theta(Vertices + Edges)$

Height of Tree: if(!node) ret 0
else if(node)
int A = height(node→left)
int B = height(node→right)
ret max(A,B)+1

Number of Nodes: if(!node) ret 0
else if(node)
int A = num_nodes(node→left)
int B = num_nodes(node→right)
ret A + B + 1

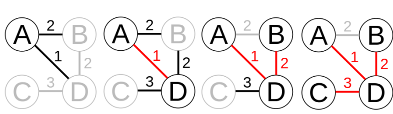
Divide & Conquer: Reminder: if $f(n)$ is a constant (no n), $d = 0$
Master Theorem: (works for $O(n), \Omega(n), \Theta(n)$), ($f(n)$ is the "split & combine" part)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)^d \text{ if:} f(n) \in \Theta(n^d) \text{ Where } d \geq 0 \text{ in recurrance, then: } T(n) \in \begin{cases} \Theta(n^d) \text{ if: } a < b^d, \\ \Theta(n^d \log n) \text{ if: } a = b^d, \\ \Theta(n^{\log_b a}) \text{ if: } a > b^d \end{cases}$$

Mergesort:
Input: Array A[0$\cdots$ n-1], of order-able elements.
Output: Array A[0$\cdots$ n-1], sorted in nondecreasing order.
Function: Divide in half until array size 1 (base case)
When base case hit: call Merge()
Best Case/Avg Case: $\Theta(n\log(n))$
Worst Case: $\Theta(n^2)$ if bad pivot, else $\Theta(n\log(n))$

Merge:
Input:Arrays B[0$\cdots$p-1], C[0$\cdots$q-1], Both sorted.
Output: Array A[0$\cdots$ p+q-1] of elements of B & C.
Function: Merge B,C into array A based on size of individual values
2 index at start of array
Find smaller, Insert. Move inserted index++ re-compare, etc.
Merge: $\Theta(n)$, Split: $\Theta(1)$

MST: Minimum: Tree with minimum weight. Spanning: Touches every vertex. Tree: No cycles.



Prim Example (one of two possible):
Above example has two solutions for MST. Prim's however only returns 1 (could be either).
Pick a node, add smallest edges & node connected, unless it forms a cycle.
Prim Efficiency: Adj Matrix: $O(V^2)$: Scanning takes that long. (Adj matrix size V*V). Sparse Graphs: $O(E\log_2 V)$: Each iteration is $log_2(v)$ time, Number iterations is 2*E, simplifies to E because Big-O.

Kruskal: Sorts all edges, walks through each edge, adding one at a time to the list. Don't add ones that create a cycle.
Kruskal Efficiency: $O(E\log V)$, where E = # Edges, V = # Verts
Dijkstra: Find Shortest Path, Given source node.

Vertex	A	B	C	D (root)
WT	INF	INF	INF	0
ST	-1	-1	-1	D
IN?	0	0	0	1

Hashing: Best/Avg: $\Theta(1)$, $\Theta(n)$ worst. Unless using a BST: then $\Theta(\log n)$ for worst/avg
Tabulation vs Memoization
Memoization can recall values instead of always recomputing.
Memoization is always recursive.
Tabulation builds table iteratively from the bottom-up.
Memoization:+RT Eff: for many overlapping values **OR** sparse table
Tabulation:+RT Eff: for dense table **OR** recursion overhead too expensive
Floyd's: Dijkstra but for every single node. Better than Dijkstra for a Dense Graph (less overhead). Outputs a 2D Array. (Adj Matrix)
Warshall's: Computes transitive closure of directed graphs: (Is there a path?) Returns a 2D adj matrix where each 1 shows that there's a path from vertex x to vertex y.
Each row: From vertex i to vertex j.
Each column: To vertex i from vertex j.

```
for every step, compute all the nodes I can get to.
Call it again, passing in computed step.
Eventually it will output the solution.
```

Heap: Children of $A[n]$ are $A[2N]$ and $A[2N+1]$
Nodes Left, empties on right.
Heapsort: In place algorithm. $\Theta(n \log n)$
Heap Properties: BST: Filled L→R. All levels are full, except possibly last level.
Increasing Key: Increase Priority of an Item. → FixUp()
Decreasing Key: Decrease Priority of an Item. → FixDown()
FixUp:

```
fixUp(item a[], int k) {
  while((k>1) && (a[k/2] < a[k])) {
    exchange(a[k], a[k/2]); k = k/2;
  }
}
```

FixDown:

```
fixDown(item a[], int k, int N) {
  int j;
  while(2*k <= N) {
    j = 2*k;
    if ((j < N) && less ((a[j], a[j+1]))) j++;
    if (!less(a[k], a[j])) break;
    exchange a[k], a[j]); k = j;
  }
}
```

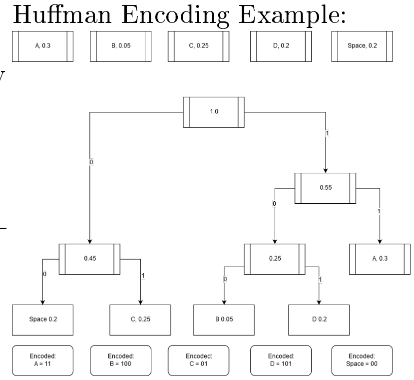
Insertion: $O(\lg N)$ Insert item @ end of heap, then call FixUp().
Deletion: $O(\lg N)$ Delete MAX element [first]). Exchange first/last, then delete last, then call FixDown().
Batch Init: Top-Down $O(N \lg N)$ time, $O(N)$ extra space (Run through counter, inserting). Bottom Up: $O(N)$ time, $O(1)$ extra space. Starts at bottom, insert, call fixDown()
Heapsort:

```
Build heap from unordered array.
Find Max element A[1].
Swap A[n] with A[1], moves max element to end of array.
Discard node n from heap.
Run FixDown(a,1, heap.size()).
```

Dynamic Knapsack:
Input: i = index number corresponding to item
Input: j = index number corresponding to capacity
Output: Optimal subset of values.
Start at largest (i, j), create recursion tree based on branches. Stop at leaves: $(0, x) \vee (x, 0)$. Any repeated calculations (excluding (0,x), (x,0)) in the tree are marked with an * (if the * appears at the root of a subtree, **AND** the subtree is a duplicate of another subtree, one of the two subtrees must be ignored [do NOT count duplicates inside ignored]), any cells not in the recursion tree are marked with a -.

```
MFKnapsack(i, j) {
  if(Table[i,j] < 0) {
    if(j < Weight[i]) {
        Branch LeftcurrentValue = MFKnapsack(i−1,j);
        Note, this path gives an empty right subnode
    }
    else {
      currentValue = max:
        Branch LeftMFKnapsack(i−1,j)
        Branch RightValue[i]+MFKnapsack(i−1, j−Weight[i]));
    }
    else {
      Print Asterisk Here
    }
    Table[i,j] = currentValue;
  }
  return Table[i,j];
}
```

Huffman Encoding Example:



P/NP: A lower bound Ω is tight if we know that there exists an algorithm with efficiency of lower bound.
Decision tree: Pick arbitrary values within scope of problem and compare, then continue until every case covered. Think a massive conditional statement.
Tree Preorder:

```
if(!node)ret;
dosomething(node);
traverse(node→left);
traverse(node→right);
```

Tree Inorder:

```
if(!node) ret;
traverse(node→left);
dosomething(node);
traverse(node→right);
```

Tree Postorder:

```
if(!node) ret;
traverse(node→left);
traverse(node→right);
dosomething(node);
```