Runtime Hierarchy: l<logn<sqrtn<n<nlogn<n^2<n^3<2^n<3^n<n^n
Master method: T(n)=aT(n/b) + f(n), where a >= 1, b>1, and f(n) is a given function.
Insertion Sort:
Quadratic - O(2n^2)
Computing Power: 10 billion instructions/sec.
Sorting Time (n=10M): 5.56 hours
Quadratic runtime for worst and average cases.
Best case: Linear runtime if the inner loop doesn't run.
Merge Sort:
Steps: Divide, Conquer, Combine.
Efficient - O(n log(n))
Computing Power: 10 million instructions/sec.
Sorting Time (n=10M): 19.38 minutes
Efficient O (n log(n)) runtime for all cases.
Loop Invariants:
Used to prove correctness. Involves initialization, maintenance, and termination steps.
Asymptotic Notations:
O-notation (Big-O): Upper bound: Describes the worst-case scenario.
Θ-notation (Theta): Tight bound: Represents a balanced and consistent growth.
Ω-notation (Big-Omega): Lower bound: Describes the best-case scenario.
Exponentiation by Squaring:
Optimizes the calculation of 2^n using recursion.
Solving **Recurrence Relations**:
Requires making a guess and proving it using mathematical induction.
Examples of recurrences and their solutions:
$T(n) = T(n - 1) + 1 \; yields \; O(n)$
$T(n) = 2T(n/2) + n \; yields \; O(n \log n)$
$T(n) = T(n - 1) + n^2 \; yields \; O(n^3)$
Heaps:
Max-Heap: Each parent node has a key greater than its children's keys.
Min-Heap: Each parent node has a key smaller than its children's keys.
Operations:
Heap-Maximum, Heap-Extract-Max, Heap-Increase-Key, Max-Heap-Insert.
Heap-Minimum, Heap-Extract-Min, Heap-Decrease-Key, Min-Heap-Insert.
Runtimes:
Operations take **O(logn) time for both Max Heap and Min Heap.**
Quicksort:
Follows Divide, Conquer, Combine
Worst Case: Theta(n^2)
Best Case: Theta(n log n)
Balanced: O(n log n)
Theorem: **Comparison sort algorithms** require at least Ω(n lg n) comparisons in the worst case.
Proof: In a decision tree for comparison sorting with n elements, we have n! ≤ the number of reachable leaves (l).
A tree with height h has at most 2^h leaves. Thus, n! ≤ l ≤ 2^h.Taking logarithms, we find h ≥ lg(n!) = Ω(n lg n).
Counting Sort:
Works for integers in the range 0 to k.
Time complexity is Θ(n) when k = O(n).
Counts elements less than or equal to x to place them in the output array.
Uses arrays B (sorted output) and C (temporary storage).
No comparisons are involved.
Dynamic Sets and Dictionaries:
Dynamic sets can change over time.
Dictionary operations include insert, delete, and membership tests.
Elements in a dynamic set have identifying keys.
Operations include Search, Insert, Delete, Minimum, Maximum, Successor, and Predecessor.
Stacks and Queues:
Stacks follow Last-In, First-Out (LIFO) policy.
Queues follow First-In, First-Out (FIFO) policy.
Operations include Push, Pop for stacks, and Enqueue and dequeue for queues.
Linked Lists:
Elements in a linked list have key attributes and pointers to the next and previous elements.
List-Search takes Θ(n) time for searching.
List-Insert and List-Delete take Θ(1) time for insertion and deletion.
Sentinels can simplify boundary conditions and make code more elegant.
Binary Trees:
A binary tree can be represented using linked data structures, where each node is an object with key attributes and pointers to its children and parent nodes. A binary tree has a root node, and if the root is nil, the tree is considered empty.
Binary Search Trees (BSTs):
A binary search tree is a binary tree where each node follows the binary search tree property and is typically used for efficient searching and sorting.
Specifically, for any given node:
All nodes in its left subtree have keys less than the node's key.
All nodes in its right subtree have keys greater than the node's key.
Inorder Tree Walk: This is a way to traverse a BST to print its keys in sorted order. The algorithm visits nodes in the order left child, current node, and right child, which ensures sorted order. **Time Complexity:** O(n) for an n-node tree.

Searching: To search for a key in a BST, you can use the Tree-Search or Iterative-Tree-Search procedures. Starting at the root, they recursively or iteratively traverse the tree based on comparisons of the key values. **Time Complexity:** O(h), where h is the height of the tree.

Minimum and Maximum: To find the minimum and maximum values in a BST, you can traverse left or right from the root until reaching a nil node. **Time Complexity:** O(h), where h is the height of the tree.

Successor and Predecessor: These operations are used to find the next or previous key in sorted order, respectively. They involve finding the immediate neighbors of a given node. **Time Complexity:** O(h), where h is the height of the tree.

Insertion: To insert a new node with a given key into a BST, you use the Tree-Insert procedure. It places the new node in the correct position according to the BST property. Time Complexity: O(h), where h is the height of the tree.

Deletion: To delete a node with a given key from a BST, you use the Tree-Delete procedure. It handles cases where the node has no children, one child, or two children. **Time Complexity**: O(h), where h is the height of the tree.

Direct addressing is a simple and effective method for small key universes. It's ideal for dynamic sets where elements have unique keys drawn from a small universe $U = \{0, 1, ..., m - 1\}$. It employs an array, $T[0 .. m - 1]$, where each slot corresponds to a key in U. Slot k represents an element with key k, or it's set to nil if no such element exists.

The dictionary operations are:

Direct-Address-Search: Return T[k].

Direct-Address-Insert: Set T[x.key] = x.

Direct-Address-Delete: Set T[x.key] = nil.

Sometimes, the direct-address table can store elements directly, eliminating the need for external objects with pointers. In such cases, an indicator key can identify empty slots.

Hash tables offer a more space-efficient solution for cases where the number of keys stored (K) is much smaller than the universe (U).

Hash tables reduce storage requirements to $\Theta(|K|)$ while maintaining O(1) time for searching.

How Hashing Works:

Hash tables use a hash function h to map keys (k) from U to slots in an array $T[0 .. m - 1]$. Hash function h(k) maps key k to a specific slot (h(k)).

Ideally, each key is equally likely to hash to any slot. **Collisions occur when multiple keys hash to the same slot.**

Collision Resolution:

Chaining is a technique to handle collisions. Elements with the same hash value are placed in linked lists in the same slot. Each slot in the hash table contains a pointer to the head of the list.

Chained-Hash-Insert: Add an element at the head of the corresponding list.

Chained-Hash-Search: Search for an element with a key in the list of its hash slot.

Chained-Hash-Delete: Delete an element from the list of its hash slot.

Average-case time for unsuccessful AND successful searches is $\Theta(1 + \alpha)$, where α is the load factor (average number of elements in a chain).

Division Method: Use h(k) = k mod m for the hash function, but avoid m values that are powers of 2 to prevent uneven distributions.

Multiplication Method: Multiply the key k by a constant A ($0 < A < 1$), take its fractional part, then multiply by m and take the floor: $h(k) = \lfloor m(kA \bmod 1)\rfloor$. The choice of A is crucial and depends on your data. **Optimal Constants for Multiplication**: Knuth recommends $A = (\sqrt{5} - 1)/2$ for the multiplication method.

Value of m: It's often a power of 2 ($m = 2^p$) for practical implementation. Use the p most significant bits of the result as the hash value.

Depth-First Search (DFS):

RUNTIME: O(V + E), where V is the number of vertices (nodes) in the graph, and E is the number of edges. In the worst case, the algorithm visits each vertex and each edge once.

DFS is a strategy for exploring graphs, seeking to go as deep as possible along a branch before backtracking.

During DFS, vertices are colored white, gray, or black to indicate their state.

Depth-First Search can be represented as a depth-first forest, a collection of trees.

It also assigns discovery times (u.d) and finishing times (u.f) to vertices during the process.

DFS helps identify properties like descendant intervals and nesting, as per the Parenthesis theorem.

The White-Path theorem states that in a depth-first forest, a vertex v is a descendant of u if there is a path of white vertices between them.

Tree edge: Edge in depth-first search forest

Back edge: Edge connecting node to ancestor

Forward edge: Edge connecting node to descendant

Cross edge: Every other edge

Topological Sort:

RUNTIME: also O(V + E)

A topological sort is a linear ordering of vertices in a directed acyclic graph (DAG). It orders vertices so that if there's an edge from u to v, u comes before v in the order.

Topological sort is useful for tasks with dependencies, like scheduling.

The Topological-Sort algorithm computes the sort by running DFS to calculate finishing times, then reversing the order of these times to get the topological sort. The correctness of this approach is based on the properties of DFS, and it works for DAGs.

Strongly connected components in a directed graph are sets of vertices where every vertex in the set is reachable from every other vertex in the set.

Algorithm steps to find components:

Perform DFS on G to compute finishing times u.f for each vertex u.

Compute the transpose GT of the original graph G in O(V + E) time.

Perform DFS on GT, considering vertices in order of decreasing u.f.

Output the vertices of each tree in the depth-first forest formed in step 3 as separate strongly connected components.

A breadth-first tree is a subgraph of the original graph formed during a BFS. It is a tree structure rooted at the source vertex and includes all reachable vertices. Tree edges connect vertices in the order they are discovered in the BFS process.

Print-Path Procedure: Given a graph G with a BFS tree, this procedure prints out the vertices on the shortest path from the source s to a target vertex v.

Greedy Algorithms:

Activity Selection Problem: Choose activities with earliest finish times. Ensure maximum number of non-overlapping activities.

Optimal Substructure: For a set of activities, find the best subset of compatible ones. Greedy choice involves activities with earliest finish times.

Greedy Strategy: Always pick the activity that leaves room for more. Choose activities sorted by increasing finish times.

Recursive Greedy Algorithm: Pick the first activity, then solve the remaining subproblem. Repeat until no more activities left.

Iterative Greedy Algorithm: Pick the first activity with the earliest finish time. Keep adding compatible activities with earliest finish times.

Greedy-Choice Property: Make choices that seem best at the moment. Choices depend on the current problem, not future ones.

```
HEAD-DEQUEUE(Q, x):
 if QUEUE-EMPTY(Q)
    error "underflow"
 else
 x = Q[Q.head]
 if Q.head == Q.length:
    Q.head = 1
 else Q.head = Q.head + 1
 return x
```

```
TAIL-DEQUEUE(Q, x):
  if QUEUE-EMPTY(Q)
   error "underflow"
  else
 if Q.tail == 1:
  Q.tail = Q.length
 else Q.tail = Q.tail - 1
 return Q[Q.tail]
```

```
ENQUEUE(Q, x)
   if QUEUE-FULL(Q)
        error "overflow"
   else
      Q[Q.tail] = x
      if Q.tail == Q.length
         Q.tail = 1
      else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
   if QUEUE-EMPTY(Q)
      error "underflow"
   else
      x = Q[Q.head]
      if Q.head == Q.length
         Q.head = 1
      else Q.head = Q.head + 1
      return x
```

```
HEAD-ENQUEUE(Q, x):
if QUEUE-FULL(Q)
   error "overflow"
if Q.head == 1:
   Q.head = Q.length
else Q.head = Q.head - 1
Q[Q.head] = x
```

```
TAIL-ENQUEUE(Q, x):
if QUEUE-FULL(Q)
   error "underflow"
if Q.tail == 1:
   Q.tail = Q.length
else Q.tail = Q.tail - 1
return Q[Q.tail]
```

```
TREE-MIN-RECURSIVE(x):
if x.left ≠ NIL:
  return TREE-MIN-RECURSIVE(x.left)
else:
  return x
```

```
TREE-MAX-RECURSIVE(x):
if x.right ≠ NIL:
  return TREE-MAX-RECURSIVE(x.right)
else:
  return x
```

```
LIST-REVERSE(L)
     p[1] = NIL
     p[2] = L.head
     while p[2] != NIL
          p[3] = p[2].next
          p[2].next = p[1]
          p[1] = p[2]
          p[2] = p[3]
     L.head = p[1]
```

```
PRINT-BINARY-TREE(T)
     x = T.root
     if x != NIL
          PRINT-BINARY-TREE(x.left)
          print x.key
          PRINT-BINARY-TREE(x.right)
```

```
PREORDER-TREE-WALK(x)
   if x != NIL
      print x.key
      PREORDER-TREE-WALK(x.left)
      PREORDER-TREE-WALK(x.right)

POSTORDER-TREE-WALK(x)
      if x != NIL
           POSTORDER-TREE-WALK(x.left)
           POSTORDER-TREE-WALK(x.right)
           print x.key
```

```
TREE-PREDECESSOR(x):
if(x.left!=NIL)
   return TREE-MAXIMUM(x.left)
y=x.p
while(y!=NIL and x==y.left)
     x=y
     y=x.p
 return y
```

```
Exponentiator(n):
   if n = 0:
      return 1
   else if n mod 2 = 0:
      x =
Exponentiator(n/2)
      return x * x
   else:
      x =
Exponentiator((n-1)/2)
      return 2 * x * x
```

```
RECURSIVE-TREE-I
NSERT(T, z)
   if T.root == NIL
      T.root = z
   else INSERT(NIL,
T.root, z)
```

```
LINEAR-SEARCH(A, key)
  for i = 1 to length(A)
     if A[i] equals key
        return i // Key found at
index i
  return -1 // Key not found in
the array
```

```
SelectionSort(A):
   n = length(A)  // Get the
number of elements in the
array
   for i from 1 to n - 1:
      min_index = i  // Assume
the current element is the
smallest
      for j from i + 1 to n:
         if A[j] < A[min_index]:
            min_index = j  //
Found a smaller element
      if min_index != i:
         // Swap the current
element with the smallest
element found
         swap A[i] with
A[min_index]
```

```
MIN-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] < A[i]
smallest = l
else smallest = i
if r ≤ A.heap-size and A[r] < A[smallest]
smallest = r
if smallest != i
exchange A[i] with A[smallest]
MIN-HEAPIFY(A, smallest)


Heap-Minimum(A):
    return A[1]
```

```
Min-Heap-Insert(A, key):
    A.length = A.length + 1
    A[A.length] = ∞

Heap-Decrease-Key(A, A.length, key)
```

```
Heap-Extract-Min(A):
    if A.length < 1:
        error "Heap underflow"
    min_val = A[1]
    A[1] = A[A.length]
    A.length = A.length - 1
    Min-Heapify(A, 1)
    return min_val
```

```
Heap-Decrease-Key(A, i, key):
    if key > A[i]:
        error "New key is larger than current key"
    A[i] = key
    while i > 1 and A[Parent(i)] > A[i]:
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    largest = i

    if l ≤ A.heap-size and A[l] > A[i]
        largest = l

    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r

    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)


Heap-Maximum(A):
    return A[1]
```

```
Max-Heap-Insert(A, key):
    A.length = A.length + 1
    A[A.length] = -∞

Heap-Increase-Key(A, A.length, key)
```

```
Heap-Extract-Max(A):
    if A.length < 1:
        error "Heap underflow"
    max_val = A[1]
    A[1] = A[A.length]
    A.length = A.length - 1
    Max-Heapify(A, 1)
    return max_val
```

```
Heap-Increase-Key(A, i, key):
    if key < A[i]:
        error "New key is smaller than current key"
    A[i] = key
    while i > 1 and A[Parent(i)] < A[i]:
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

```
def stack_dfs(graph):
    time = 0
    stack = []
    for u in graph.vertices:
        if u.color == 'WHITE':
            stack.append((u, 1, 0))
            while stack:
                u, step, index = stack.pop()
                if step == 1:
                    u.color, time, u.d, step = 'GRAY', time + 1, time + 1, 2
                if step == 2:
                    adjacent_white = [(v, i) for i, v in enumerate(graph.adjacency_list[u][index:], index) if v.color == 'WHITE']
                    if not adjacent_white:
                        u.color, time, u.right = 'BLACK', time + 1, time + 1
                    else:
                        v, new_index = adjacent_white[0]
                        v.parent = u
                        stack.append((u, 2, new_index + 1))
                        u, step, index = v, 1, 0
```

```
MEMOIZED-LCS-LENGTH(X, Y, i, j)
    if c[i, j] > -1
        return c[i, j]
    if i == 0 or j == 0
        return c[i, j] = 0
    if x[i] == y[j]
        return c[i, j] = LCS-LENGTH(X, Y, i - 1, j - 1) + 1
    return c[i, j] = max(LCS-LENGTH(X, Y, i - 1, j), LCS-LENGTH(X, Y, i, j - 1))
```

```
PRINT-LCS(c, X, Y, i, j)
    if c[i, j] == 0
        return
    if X[i] == Y[j]
        PRINT-LCS(c, X, Y, i - 1, j - 1)
        print X[i]
    else if c[i - 1, j] > c[i, j - 1]
        PRINT-LCS(c, X, Y, i - 1, j)
    else
        PRINT-LCS(c, X, Y, i, j - 1)
```

```
Insertion-Sort(A):
for j = 2 to A.length
 key = A[j]
 i = j − 1
 while i > 0 and A[i] > key
 A[i + 1] = A[i]
  i = i − 1
 A[i + 1] = key
```

```
Merge(A, p, q, r):
n1 = q − p + 1
n2 = r − q
for i = 1 to n1
L[i] = A[p + i − 1]
for j = 1 to n2
R[i] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
if L[i] ≤ R[j]:
A[k] = L[i]
i = i + 1
else:
A[k] = R[j]
 j = j + 1
```

```
Partition(A, p, r):
x = A[r]
i = p − 1
for j = p to r − 1
if A[j] ≤ x
i = i + 1
exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
Quicksort(A, p, r): if p < r:
q = Partition(A, p, r)
Quicksort(A, p, q − 1)
Quicksort(A, q + 1, r)
```