Chapter 1

1. Experiment studies: Implement algorithm in program. Run program with inputs of different size. Measure running time using language's time method. Plot results.
2. Theoretical analysis: Uses pseudocode. Characterize running time as function of input size, n.
3. Time complexity from fastest to slowest: O(1), O(log n), O(n), O(n log n), O(n^2), O(2^n).

Chapter 2: Hash Table

1. Collision: When 2 different keys hash to same value (same location in hash table).
2. Time complexity:
- Search: Best O(1), Worst O(n)
- Insert: Best O(1), Worst O(n)
3. Clustering in Chaining: Some linked lists are empty. Some have long chain. Worst case: O(n) - max length of chain.
- Clustering in Linear Probing: Some consecutive array slots are occupied. Some are empty. Worst case: O(n) - length of array.
- Clustering happens for both but linear probing search time is longer if many collisions.
4. Quadratic Probing: h(k) = (h(k) + j^2) % m, where j is # of attempts to resolve collision.
5. Memory in Chaining: extra memory for list.
- Memory in Linear Probing: fixed amount of memory.
6. Linear Probing: suited for caching - when you load an item the next item is always loaded.
7. Uniform hashing: Elements are spread evenly among indexes of a hash table. Allows search: O(1).
8. Good hash table size: Prime number not too close to power of 2.

Chapter 3: BST

1. Binary tree: Each node has only up to 2 children.
2. Level: # of edges in the path from root to this node. Root: level 0.
3. Full binary tree: Each node has two children except leaf nodes.
4. Complete binary tree: Full binary tree but missing only rightmost nodes on last level.
5. BST Insert: O(n), Search: O(n), Delete: O(n)
6. Balance Factor (node n) = Height(RightSubtree(N)) - Height(LeftSubtree(N)). 1, 0, -1 = balanced BST.
7. AVL (Balanced BST) Insert: O(lg n), Search: O(lg n), Delete: O(lg n)

8. AVL Insert: Rotation. See slide.
9. AVL Delete:
- No children [CASE 1]: Just delete.
- One child [CASE 2]: Connect parent node of deleted node to child of deleted node.
- Two children [CASE 3]: Find max node in to-delete node's left subtree. Swap max node with to-delete node. Delete to-delete node using case 1 or 2..
10. AVL Traversal: Preorder (parent, left, right), Inorder (left, parent, right), Postorder (left, right, parent).
11. AVL: 1. Easier to create. 2. Produce sorted result with inorder traversal. 3. Find min/max in O(lg n).

Chapter 4: Priority Queue and Heap

1. PQ (array) Dequeue: O(n), Enqueue: O(n).
2. PQ (heap) Enqueue: O(lg n), Dequeue: O(lg n). Height: O(lg n).
3. Maxheap: Complete binary tree. Value of each node >= values of its children.
4. Heap Dequeue: Swap rightmost node with root. Heapify down the 'root'.
5. Heapifying: Swapping upwards/downwards to form a new heap.
6. Heap Enqueue: Add new node to rightmost node. Heapify up to root.
7. Heap Left: 2 * Parent, Right = 2 * Parent + 1, Parent = (Child) / 2.
8. Enqueue one element at a time [METHOD 1]: O(n lg n) = n items, lg n for each item.
9. Create heap straightaway from array [METHOD 2]: O(n). Faster because not need to heapify leaf nodes.
   TIP: Heapify downwards starting with parent of last leaf node. Right to left.

Chapter 5: Graphs

1. Representation: Adjacency Matrix and Adjacency List.
2. AM: Direct connection between two specific vertices.
3. AL: Process linked list of an AL, node by node
4. AM: Spatial complexity O(n^2).
5. AL: Spatial complexity O(n^2).
6. Sparse graph (AL): Doesn't have many edges relative to # of vertices. Spatial complexity: O(n).
7. Dense graph (AL/AM): May have close to n ^ 2 edges. Spatial complexity: O(n ^ 2).
8.

	AL	**AM**
Space	N + m	n^2

IncidentEdges(v)	deg(v)	n
AreAdjacent(v, w)	min(deg(v), deg(w))	1
InsertVertex(o)	1	n^2
InsertEdge(v, w, o)	1	1
RemoveVertex(v)	deg(v)	n^2
RemoveEdge(e)	1	1

9. Connected graph: There is a path between every pair of vertices.
10. DFS (AL): O(n + m), deg(v) = 2m.
11. BFS (AL): O(n + m).
12. Directed acyclic graph (DAG): Digraph with no directed cycles.
13. Topological sorting: Only for DAG. O(n + m). See slide example.
    TIP: Label node with <u>no outgoing edges</u>.

Chapter 6 & 7: Sorting & Selection

Selection Sort: O(n ^ 2), Unstable, In-Place.
TIP: Swap max with current.
Array: 3 6 5 2 4, n = 5
I = 4, 3 4 5 2 <u>6</u>
I = 3, 3 4 2 <u>5 6</u>
I = 2, 3 2 <u>4 5 6</u>
I = 1, 2 <u>3 4 5 6</u> (sorted)

Heapsort: O(n lg n), Unstable, In-Place.
TIP: Represent array as PQ. Dequeue root. Put rightmost node as root. Heapify downwards.
Array: 3 6 5 2 4
PQ: 6 4 5 2 3

```
    6
   / \
  4   5
 /  \
2    3
```

I = 4, 5 4 2 3 <u>6</u> [Left subtree then right subtree hence 5 4 2 3 instead of 5 4 3 2]

Rightmost node as root.
   3
  / \
	Heapify downwards.
	   5
	  / \

4 5 / 2	4 3 / 2

I = 3, 4 2 3 5 6
I = 2, 3 2 4 5 6
I = 1, 2 3 4 5 6
I = 0, 2 3 4 5 6

Stable: Preserves order of duplicate keys.

Quick Sort: O(n^2) if pivot is not randomised. O(n log n) if pivot is randomised. Stable. Space inefficient. In-Place.
Space efficient = unstable.
Array: 99 11 77 55 33, first element as pivot
11 77 55 33 99*
11* 77 55 33 99*
11* 55 33 77* 99*
11* 33 55* 77* 99* (Sorted)

In-place: Produce results by overwriting input data structure.
Not-in-place: Store temporary result into an additional temporary data structure. At the end store final result into input data structure.

Merge Sort: O(n log n). Stable. Not-in-place. Height: O(log n).
Array: 107 55 100 98 33 75 3

p	**r**	**Array**
0	1	55 107 100 98 33 75 3
2	3	55 107 98 100 33 75 3
0	3	55 98 100 107 33 75 3
4	5	55 98 100 107 33 75 3
4	6	55 98 100 107 3 33 75
0	6	3 33 55 75 98 100 107

Bucket Sort: O(n+N)
29 25 3 49 9 37 21 43

0-9	10-19	20-29	30-39	40-49
3, 9		29, 25, 21	37	49

0-9	10-19	20-29	30-39	40-49
3, 9		21, 25, 29	37	49

3 9 21 25 29 37 49 (Sorted)

Radix Sort: O(d(n+N))
38 9 38 37 155 197 65
D = 1

0	1	2	3	4	5	6	7	8	9
					155, 65		37, 197	38, 38	9

155 65 37 197 38 38 09
D = 2

0	1	2	3	4	5	6	7	8	9
9			37, 38, 38		155	65			197

009, 037, 038, 038, 155, 065, 197
D = 3

0	1	2	3	4	5	6	7	8	9
009, 037, 038, 038, 065	155, 197								

9, 37, 38, 38, 65, 155, 197 (Sorted)

Radix Sort for Binary Numbers (b-bit): O(bn)

Quick Select: O(n^2)
Find 5th smallest number = k = 4. Last element as pivot.
77 99 22 66 55 44 11 88 33

~~22 11 33*~~ 77 99 66 55 44 88
Pi = 2 < k, look right

~~22 11 33*~~ 77 66 55 44 ~~88* 99~~
Pi = 7 > k, look left

~~22 11 33* 44*~~ 77 66 55 ~~88* 99~~
Pi = 3 < k, look right

~~22 11 33 44*~~ 55* 77 66 ~~88* 99~~
Pi = 4 = k
5th smallest element = 55

Chapter 8: Divide and Conquer

Divide and Conquer
1. Divide: Divide input data in 2 or more disjoint subsets.
2. Recur: Recursively solve subproblems.
3. Conquer: Combine solutions into a solution.

Merge Sort
1. Divide: Partition input sequence into 2 sequences S1 and S2 of about n/2 elements each.
2. Recur: Recursively sort S1 and S2.
3. Conquer: Merge S1 and S2 into a unique sorted sequence.

Quick Sort
1. Divide: Pick a random element x as pivot. Partition S into
- L: Elements < x.
- E: Elements = x.
- G: Elements > x.
2. Recur: Sort L and G.
3. Conquer: Join L, E and G.

Master Theorem: T(n) = aT(n/b) + f(n)
Case 1 [n ^ (log b a) > f(n)]: T(n) = O(n ^ (log b a))
Case 2 [f(n) > n ^ (log b a) by logarithmic factor]: T(n) = O(n ^ (log b a) * log ^ (k+1) n)
Case 3 [f(n) > n ^ (log b a)]: T(n) = O(f(n))

Chapter 9: Greedy Algorithms

Greedy method: Algorithm design paradigm with the elements:
- Configurations: Different choices, collections, or values to find.

- Objective function: A score assigned to configurations, which we want to maximize/minimize.

Greedy-choice property: Global-optimal solution can be found by a series of local improvements from a starting configuration.

Fractional Knapsack: O(n log n)
Knapsack weight = 20 kg

Item no	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Weight (kg)	7	4	3	9	8	4	5
Benefit (RM)	70	16	45	45	40	80	10
Value	**10**	**4**	**15**	**5**	**5**	**20**	**2**

Select Item No.	**Weight**	**Benefit**
5	4	80
2	3	45
0	7	70
3 or 4	6	30

Total Benefit = 225

Shortest Path: Find a path of minimum total weight between two vertices.
Dijkstra: Works only for positive-weight edges. O((n + m) log n) for AL, O(m log n) for connected graph.
TIP: SELECT LOWEST DISTANCE(S) (Continue from available).
Edge relaxation: Remove cycle.

Minimum Spanning Trees (MST): Spanning tree of a weighted graph with minimum total edge weight.
Prim-Jarnik: O((n + m) log n) for AL, O(m log n) for connected graph.
TIP: SELECT MIN EDGE (Continue from available). REMOVE EDGE THAT FORMS CYCLE.
Kruskal: O((n + m) log n).
TIP: SELECT NEXT MIN EDGE. REMOVE EDGE THAT FORMS CYCLE.

Chapter 10: Dynamic Programming

Dynamic Programming:

- Simple subproblems: Subproblems are defined in terms of a few variables.
- Subproblem optimality: Global optimum value are defined in terms of optimal subproblems.
- Subproblem overlap: The subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Matrix Chain Multiplication: O(n^3)

I = row. J = column.

M[i,j] = 0 if i = j, else min i<=k<j { M[i,j] = M[i,k] + M[k+1,j] + $p_{i-1}p_kp_j$}

A1	A2	A3	A4
25 x 10	10 x 15	15 x 5	5 x 35

	A1	**A2**	**A3**	**A4**
A1	0	(1,1)(2,2): 25 x 10 x 15 = 3750	(1,1)(2,3): 750 + 25 x 10 x 5 = 2000 [MIN] (1,2)(3,3): 3750 + 25 x 15 x 5 = 5625	(1,1)(2,4): 7875 + 25 x 10 x 35 = 16625 (1,2)(3,4): 3750 + 2625 + 25 x 15 x 35 = 19500 (1,3)(4,4): 2000 + 25 x 5 x 35 = 6375 [MIN]
A2		0	(2,2)(3,3): 10 x 15 x 5 = 750	(2,2)(3,4): 2625 + 10 x 15 x 35 = 7875 (2,3)(4,4): 750 + 10 x 5 x 35 = 3375 [MIN]
A3			0	(3,3)(4,4): 15 x 5 x 35 = 2625
A4				0

0/1 Knapsack: O(nW)
W = 10

Item	**1**	**2**	**3**	**4**
Weight	5	4	6	3
Value	10	40	30	50

I \ W	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	**0**	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	**40**	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	**90**

Items: 4, 2
Weights: 3 + 4 = 7
Values: 50 + 40 = 90

Floyd–Warshall: I skip.

Chapter 11: Text Processing

Brute Force: O(nm)

Boyer-Moore: O(mn + s), average: O(m+n)
Text = ACTCTCCATGATTAGTCACTC
Pattern = TAGTCACTC

Character	T	A	G	C
Last Occurrence	7	5	2	8

Match letters from right to left.
If mismatch letter is inside pattern, align to match.

If mismatch letter is not inside pattern, shift whole word past mismatch location.

Red = Mismatch, Blue = Match

A	C	T	C	T	C	C	A	T	G	A	T	T	A	G	T	C	A	C	T	C
T	A	G	T	C	A	C	T	**C**												
	T	A	G	T	C	A	C	**T**	**C**											
							T	A`	**G**	T	C	A	C	T	**C**					
								T	A	G	T	C	A	**C**	**T**	**C**				
												T	**A**	**G**	**T**	**C**	**A**	**C**	**T**	**C**

KMP: O(m+n)
Text = abaababaabac
Pattern = abaabac

j	0	1	2	3	4	5	6
Prefix	a	b	a	a	b	a	c
Failure	0	0	1	1	2	3	0

Prefixes = Prefix[0] to right except Prefix[j]
Suffices = Prefix[j] to left except Prefix[0]
Some examples:

j	1	2	4	5
Prefixes	a	a, ab	a, ab, aba, abaa	a, ab, aba, abaa, abaab
Suffices	b	a, ba	b, ab, aab, baab	a, ba, aba, aaba, baaba
Length	0	1	2	3

Match letters from left to right.

a	b	a	a	b	a	b	a	a	b	a	c
a	b	a	a	b	a	c					
Mismatch happens at j = 6, we look at Failure[j - 1] = Failure[5] = **3**, so pattern starts at **3** positions before mismatch location.											

			a	b	a	a	b	a	c		
Mismatch happens at j = 3, we look at Failure[j - 1] = Failure[2] = **1**, so pattern starts at **1** position before mismatch location.											
					a	b	a	a	b	a	c

Huffman Coding: I skip.

Chapter 12: P and NP

Polynomial time: A time bound of the form O(n^k) for some fixed k.
Complexity class: A collection of languages.

P: The complexity class consisting of all languages accepted by polynomial-time algorithms.
NP: The complexity class consisting of all languages accepted by polynomial-time non-deterministic algorithms or verified by polynomial-time algorithms.

A problem (language) L is **NP-hard** if every problem in NP can be reduced to L in polynomial time.
L is **NP-complete** if it's in NP and is NP-hard.
NP-complete problems: 0/1 knapsack and travelling salesman tour.

Tables

Worst Time Complexity for Data Structures

Data Structure	**Search**	**Insert / Enqueue**	**Remove / Dequeue**
Hash Table	O(n)	O(n)	O(n)
BST	O(n)	O(n)	O(n)
AVL	O(lg n)	O(lg n)	O(lg n)
Priority Queue (Array)		O(n)	O(n)
Priority Queue (Heap)		O(lg n)	O(lg n)

Spatial Complexity for Graphs

Graph	Sparse	Dense
Adjacency List	O(n)	O(n^2)
Adjacency Matrix	O(n^2)	

Worst Time Complexity for Graph Algorithms

Algorithm	Time Complexity
DFS (Adjacency List)	O(n + m)
BFS (Adjacency List)	O(n + m)
Topological Sort (DAG)	O(n + m)

Worst Time Complexity for Sorting and Selection Algorithms

Algorithm	In-Place	Stable	Time Complexity
Selection Sort	Yes	No	O(n^2)
Heap Sort	Yes	No	O(n lg n)
Quick Sort	Yes	Space-inefficient: Yes Space-efficient: No	Pivot not randomise: O(n^2) Pivot randomise: O(n lg n)
Merge Sort	No	Yes	O(n lg n)
Bucket Sort	No	Yes	O(n+N)
Radix Sort	No	Yes	O(d(n+N))
Radix Sort for Binary Numbers (b-bit)	No	Yes	O(bn)
Quick Select	-	-	O(n^2)

Worst Time Complexity for Greedy Algorithms

Algorithm	**Time Complexity**
Fractional Knapsack	O(n lg n)
Dijkstra (Shortest Path)	Adjacency List: O((n + m) log n) Connected Graph: O(m log n)
Prim (MST)	Adjacency List: O((n + m) log n) Connected Graph: O(m log n)
Kruskal (MST)	O((n + m) log n)

Worst Time Complexity for DP

Algorithm	**Time Complexity**
Matrix Chain Multiplication	O(n^3)
0/1 Knapsack	O(nW)
Floyd–Warshall	O(n^3)

Worst Time Complexity for Pattern Matching

Algorithm	**Time Complexity**
Brute Force	O(nm)
Boyer-Moore	O(nm + s)
KMP	O(n + m)