📝 Unit 1: Algorithm Basics

Experiment 1: Fibonacci Number (Recursion)

Line(s)	Code	Explanation
1	def fib_rec(n):	**Defines** the recursive function fib_rec that takes an integer n (the index) as input.
2	if n <= 1:	Checks the **base case** for recursion (Fibonacci of 0 or 1).
3	return n	If n is 0 or 1, returns n itself (Fib(0)=0, Fib(1)=1).
4	return fib_rec(n-1) + fib_rec(n-2)	**Recursive Step**: If $n > 1$, it returns the sum of the function called for $(n-1)$ and $(n-2)$, calculating the n-th Fibonacci number.

Experiment 1: Fibonacci Number (Iteration)

Line(s)	Code	Explanation

1	def fib_iter_series(n):	**Defines** the iterative function fib_iter_series that takes an integer n (series limit) as input.
2	a, b = 0, 1	**Initializes** two variables for the first two Fibonacci numbers: 0 and 1.
3	series = []	**Initializes** an empty list to store the resulting series.
4	for _ in range(n+1):	**Starts a loop** that runs $n+1$ times to generate $n+1$ elements (from index 0 up to the n-th element).
5	series.append(a)	**Adds** the current Fibonacci number (a) to the series list.
6	a, b = b, a + b	**Updates** the sequence: the new a becomes the old b, and the new b becomes the sum of the old a and b, generating the next Fibonacci number.
7	return series	**Returns** the complete list of Fibonacci numbers up to n.

Experiment 2: Binary Search (Recursion)

Line(s)	Code	Explanation
1	def bs_rec(arr, key, l, r):	**Defines** the recursive search function with the array, search key, left index (l), and right index (r).
2	if l > r:	Checks the **base case**: If the left index has crossed the right index, the key is not in the search space.
3	return -1	Returns -1 (Key Not Found).
4	m = (l + r) // 2	**Calculates** the middle index (m) of the current search space.
5	if arr[m] == key:	Checks if the middle element matches the search key.
6	return m	Returns the middle index (Position Found).
7	elif key < arr[m]:	Checks if the key is smaller than the middle element.
8	return bs_rec(arr, key, l, m - 1)	**Recursively searches** the left half.
9	else:	Otherwise, the key must be greater than the middle element.
10	return bs_rec(arr, key, m + 1, r)	**Recursively searches** the right half.

Experiment 2: Binary Search (Iteration)

Line(s)	Code	Explanation
1	def bs_iter(arr, key):	**Defines** the iterative search function.
2	l, r = 0, len(arr) - 1	**Initializes** the boundaries: l (low) to 0 and r (high) to the last index.
3	while l <= r:	**Search Loop**: Continues as long as the search space is valid (low is less than or equal to high).
4	m = (l + r) // 2	**Calculates** the middle index (m).
5	if arr[m] == key:	Checks if the middle element equals the key.
6	return m	Returns the index (Key Found).
7	elif key < arr[m]:	Checks if the key is smaller than the middle element.
8	r = m - 1	**Adjusts** the high boundary to m-1 to search the left half.
9	else:	Otherwise, the key must be greater.
10	l = m + 1	**Adjusts** the low boundary

		to m+1 to search the right half.
11	return -1	Returns -1 if the loop finishes without finding the key.

🚀 Unit 2: Divide and Conquer

Experiment 3: Quick Sort Algorithm

Line(s)	Code	Explanation
1	def quicksort(arr):	**Defines** the Quick Sort function.
2	if len(arr) <= 1:	Checks the **base case** for recursion.
3	return arr	Returns the list if it has 0 or 1 element (already sorted).
4	p = arr[len(arr) // 2] # pivot	**Selects the pivot** element (p), choosing the middle element.
5	left = [x for x in arr if x < p]	**Partitions** the list into elements strictly smaller than the pivot.
6	mid = [x for x in arr if x == p]	**Partitions** the list into

		elements equal to the pivot.
7	right = [x for x in arr if x > p]	**Partitions** the list into elements strictly greater than the pivot.
8	return quicksort(left) + mid + quicksort(right)	**Combines**: Recursively sorts the left and right partitions, then joins them with the mid (pivot) part in the center.

Experiment 4: Merge Sort Algorithm

Line(s)	Code	Explanation
1	def mergesort(arr):	**Defines** the Merge Sort function.
2	if len(arr) <= 1:	Checks the **base case** for recursion.
3	return arr	Returns the list if it has 0 or 1 element (already sorted).
4	mid = len(arr) // 2	**Divide**: Finds the middle index to split the list.
5	left = mergesort(arr[:mid])	**Conquer**: Recursively applies Merge Sort on the left half.
6	right = mergesort(arr[mid:])	**Conquer**: Recursively applies Merge Sort on the

		right half.
7	res = []	**Initializes** the list for the merged result.
8	i = j = 0	**Initializes** pointers for the left list (i) and right list (j).
9	while i < len(left) and j < len(right):	**Merge Loop**: Continues as long as there are elements in both sorted halves to compare.
10	if left[i] <= right[j]:	Compares the smallest element in each half.
11, 12	res.append(left[i]); i += 1	Appends the smaller element from left and increments i.
13, 14	else: res.append(right[j]); j += 1	Appends the smaller element from right and increments j.
15	return res + left[i:] + right[j:]	**Combines**: Appends any remaining elements from the unexhausted list and returns the final sorted list.

Experiment 5: Matrix Multiplication (Divide and Conquer)

Line(s)	**Code**	**Explanation**
1	def multiply(A, B):	**Defines** the recursive matrix multiplication function.

2	n = len(A)	Gets the size of the square matrix ($n \times n$).
3	if n == 1: # base case	Checks the **base case** when the matrix is 1×1.
4	return [[A[0][0] * B[0][0]]]	Returns the direct multiplication result.
5	m = n // 2	Finds the mid-point (m) for splitting the matrix.
6-13	A11 = [r[:m] for r in A[:m]] ...	**Divide**: Splits matrices A and B into four $m \times m$ sub-matrices (blocks).
14	C11 = add(multiply(A11, B11), multiply(A12, B21))	**Conquer**: Recursively calculates the top-left result block C_{11}, which is $(A_{11}B_{11}) + (A_{12}B_{21})$.
15	C12 = add(multiply(A11, B12), multiply(A12, B22))	Calculates the top-right block C_{12}.
16	C21 = add(multiply(A21, B11), multiply(A22, B21))	Calculates the bottom-left block C_{21}.
17	C22 = add(multiply(A21, B12), multiply(A22, B22))	Calculates the bottom-right block C_{22}.
18, 19	result = [C11[i] + C12[i] for i in range(m)] + \ [C21[i] + C22[i] for i in range(m)]	**Combine**: Stitches the four resulting blocks back together to form the final $n \times n$ product matrix.

20	return result	Returns the final product matrix.

🗃️ Unit 2: Greedy Method

Experiment 6: Fractional Knapsack

Line(s)	Code	Explanation
1	def convert_weight(w):	**Defines** the weight conversion utility function.
2-7	if "kg" in w:...	**Standardizes** weight input strings into a single float value in kilograms.
8	items.sort(key=lambda x: x[2], reverse=True)	**Greedy Step 1**: Sorts the list of items based on their profit-to-weight ratio (x[2]) in decreasing order.
9	total_profit = 0	**Initializes** the total profit counter.
10	selected = []	**Initializes** a list to track items taken.
11	for p, w, r in items:	**Greedy Step 2**: Iterates through the items, starting with the highest ratio.

12	if capacity <= 0:	**Stop condition**: If the truck is full, break the loop.
13	if w <= capacity:	**Case 1**: Checks if the entire item fits.
14, 15	selected.append((p, w, 1.0)); total_profit += p	Takes the full item and adds its full profit.
16	capacity -= w	Reduces the remaining capacity by the item's full weight.
17	else:	**Case 2**: The item is too heavy to fit completely.
18	fraction = capacity / w	Calculates the **fraction** of the item that can fit.
19	selected.append((p, w, fraction))	Records the item and the fraction taken.
20	total_profit += p * fraction	Adds the proportional profit.
21	capacity = 0	Sets capacity to zero, as the knapsack is now full.

Experiment 7: Prim’s Algorithm

Line(s)	**Code**	**Explanation**
1	import heapq	**Imports** the heapq module to use a min-heap for the priority queue.

2	visited = [False] * n	**Initializes** an array to track which offices are already in the MST.
3	pq = [(0, 0)] # (cost, starting office)	**Initializes** the min-heap with the starting office 0, with a cost of 0.
4	total = 0	**Initializes** the minimum total cost of the network.
5	while pq:	**MST Loop**: Continues as long as there are edges to explore.
6	cost, office = heapq.heappop(pq)	**Greedy Choice**: Extracts the smallest cost connection (edge) to reach an office.
7	if visited[office]:	Checks if the extracted office is already connected (to avoid cycles/redundancy).
8	continue	If visited, skip to the next iteration.
9	visited[office] = True	**Adds** the office to the connected network.
10	total += cost	Adds the minimum cost edge weight to the total cost.
11	for edge_cost, neighbor in graph[office]:	Iterates over all connections from the newly

		added office.
12	if not visited[neighbor]:	Checks if the neighbor is an unvisited office.
13	heapq.heappush(pq, (edge_cost, neighbor))	**Adds** the connection to the priority queue so it can be considered in the next greedy choice.

Experiment 8: Kruskal's Algorithm

Line(s)	**Code**	**Explanation**
1-4	def find(p, x): ... return p[x]	**Find Operation**: Returns the root of the set containing x with path compression for efficiency.
5	def union(p, a, b):	**Union Operation**: Merges the sets containing a and b.
6	ra, rb = find(p, a), find(p, b)	Finds the roots of a and b.
7	if ra != rb:	**Cycle Check**: If roots are different, no cycle is formed.
8, 9	p[rb] = ra; return True	**Merges**: Unites the two sets and returns True (edge accepted).

10	return False	Returns False (sets are already connected, edge rejected as it forms a cycle).
11	edges.sort()	**Greedy Step 1**: Sorts all connection routes (edges) by their cost (w) from smallest to largest.
12	parent = list(range(n))	**Initializes** the DSU structure: every zone is its own set/parent.
13	total = 0	**Initializes** the minimum total cost.
14	for w, u, v in edges:	**Greedy Step 2**: Iterates through the edges from cheapest to most expensive.
15	if union(parent, u, v):	Tries to merge the sets of nodes u and v.
16	total += w	If union is True (no cycle formed), the edge cost is added to the total.

🧩 Unit 3: Dynamic Programming

Experiment 9: 0/1 Knapsack Problem

Line(s)	Code	Explanation
1-3	weights = [int(w*1000)...]	**Pre-processing**: Scales all weights and the capacity W to integers representing grams.
4	dp = [[0]*(W+1) for _ in range(n+1)]	**Initializes** the DP table (rows: n items, columns: W capacity).
5	for i in range(1, n+1):	Outer loop: Iterates through each item, from 1 to n.
6	for w in range(W+1):	Inner loop: Iterates through every possible capacity w.
7	if weights[i-1] <= w:	**Decision Check**: Checks if the current item's weight allows it to be included at this capacity w.
8, 9	dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])	**Choice**: Sets $dp[i][w]$ to the maximum of: (1) **Including** the item (value + max value of remaining space), or (2) **Excluding** the item (max value without it).
10	else:	If the item is too heavy for the current capacity w.
11	dp[i][w] = dp[i-1][w]	**Choice**: Must exclude the item, so value is the same

		as the previous item's value at this capacity.
12	chosen = []	**Initializes** list for selected items (for backtracking).
13	w = W	Starts backtracking from the final maximum capacity.
14	for i in range(n, 0, -1):	Backtracking loop: Iterates backward through the items.
15	if dp[i][w] != dp[i-1][w]:	**Selection Check**: If the maximum value is different, item i must have been included.
16	chosen.append(i-1)	Records the index of the selected item.
17	w -= weights[i-1]	Reduces the capacity w and continues checking the state before item i was added.

Experiment 10: Travelling Salesman Problem (TSP) DP

Line(s)	**Code**	**Explanation**
1	dp = [[INF]*(1<<n) for _ in range(n)]	**Initializes** the DP table for bitmasking: n rows (current city) and 2^n columns (visited mask).

2	dp[0][1] = 0	**Base Case**: The time to start at city 0 (mask 00...1) is 0.
3	for mask in range(1<<n):	Outer loop: Iterates through all possible visited subsets (mask).
4	for u in range(n):	Second loop: Iterates through the **current city** (u).
5	if mask & (1<<u):	Checks if city u is present in the current mask.
6	for v in range(n):	Innermost loop: Iterates through every possible **next city** (v).
7	if not (mask & (1<<v)):	Checks if city v is **unvisited**.
8	`new_mask = mask	(1<<v)`
9, 10	dp[v][new_mask] = min(dp[v][new_mask], dp[u][mask] + dist[u][v])	**DP Transition**: Updates the minimum time to reach city v with the new visited set, checking if the path via u is shorter.
11	ans = INF	**Initializes** the final answer.
12	full = (1<<n) - 1	Creates the mask where all n cities are set (fully visited).
13	for i in range(n):	Checks every city i that could be the last visited city.

14	ans = min(ans, dp[i][full] + dist[i][0])	**Final Step**: Finds the minimum cost from the last city i back to the starting city 0.

Experiment 11: Floyd–Warshall Algorithm

Line(s)	Code	Explanation
1	for k in range(n):	**Outermost Loop**: Iterates over every possible intermediate node k (the "via" point).
2	for i in range(n):	Second loop: Iterates over every possible source node i.
3	for j in range(n):	Innermost loop: Iterates over every possible destination node j.
4	dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])	**Relaxation**: Updates the distance $dist[i][j]$ if going $i \to k \to j$ is shorter than the direct or currently known shortest path.

Experiment 11: Bellman–Ford Algorithm

Line(s)	Code	Explanation
1	def bellman_ford(n, edges, src):	**Defines** the Bellman-Ford function.
2	dist = [float('inf')] * n	**Initializes** all distances to infinity.
3	dist[src] = 0	**Sets** the distance of the source node to 0.
4	for _ in range(n - 1):	**Main Relaxation Loop**: Repeats the process $n-1$ times.
5	for u, v, w in edges:	Inner loop: Iterates through all available edges $(u \to v)$ with weight w.
6	if dist[u] + w < dist[v]:	**Relaxation Check**: Checks if going $u \to v$ results in a shorter path to v.
7	dist[v] = dist[u] + w	Updates the distance to v if a shorter path is found.
8	# Check for negative cycle	**Begins Cycle Check**.
9	for u, v, w in edges:	Performs one extra iteration over all edges.
10	if dist[u] + w < dist[v]:	**Negative Cycle Detection**: If any distance still improves, a negative cycle exists.

11, 12	print("Negative cycle detected!"); return None	Reports the cycle and stops.
13	return dist	Returns the final shortest distances.

↩️ Unit 4: Backtracking

Experiment 12: 4-Queens Problem

Line(s)	Code	Explanation
1	def safe(r, c):	**Defines** the safety check function for row r, column c.
2	for i in range(r):	Loops through all rows i **above** the current row r.
3	if board[i][c] or (c-(r-i) >= 0 and board[i][c-(r-i)]) or (c+(r-i) < N and board[i][c+(r-i)]):	Checks for conflicts: **Same Column** (board[i][c]) OR **Left Diagonal** OR **Right Diagonal**.
4	return False	Returns False if any conflict is found.
5	return True	Returns True if no conflicts are found.
6	def solve(r=0):	**Defines** the recursive backtracking function,

		starting at row 0.
7	if r == N:	**Base Case**: Checks if all N queens have been placed successfully.
8-11	# print solution	Prints the valid arrangement found.
12	return	Ends the recursion for this solution.
13	for c in range(N):	Loops through all columns c in the current row r.
14	if safe(r, c):	Checks if placing a queen here is safe.
15	board[r][c] = 1	**Place Queen**: Tentatively places the queen.
16	solve(r+1)	**Recurse**: Tries to place the next queen in the next row.
17	board[r][c] = 0	**Backtrack**: Resets the spot (removes the queen) if the path fails or a solution is found, allowing the loop to try the next column.

Experiment 14: Graph Coloring

Line(s)	**Code**	**Explanation**
1	def safe(node, c, graph,	**Defines** the safety check

	color):	for assigning color c to node.
2	for i in range(len(graph)):	Loops through all other vertices i.
3	if graph[node][i] == 1 and color[i] == c:	**Conflict Check**: If i is connected to node AND i has the same color c, it's a conflict.
4	return False	Returns False if a conflict exists.
5	return True	Returns True if the color assignment is safe.
6	def solve(node, graph, m, colors, color):	**Defines** the recursive function to color the graph.
7	if node == len(graph):	**Base Case**: Checks if all vertices have been successfully colored.
8	return True	Returns True (solution found).
9	for c in colors:	Loops through all available colors c.
10	if safe(node, c, graph, color):	Checks if the current color is safe for the node.
11	color[node] = c	**Assign Color**: Tentatively assigns the color.
12	if solve(node+1, graph, m, colors, color):	**Recurse**: Tries to color the next node.

13	return True	If the rest of the coloring succeeds, propagate True up.
14	color[node] = None	**Backtrack**: Resets the color if the recursive call fails, so the loop can try the next color.
15	return False	If all colors fail for the current node, return False.

Experiment: Rat in a Maze

Line(s)	Code	Explanation
1	def safe(r, c):	**Defines** the safety check for moving to cell (r, c).
2	return 0 <= r < n and 0 <= c < n and maze[r][c] == 1 and not vis[r][c]	Checks four conditions for a move: **Inside boundaries**, **Open cell** (1), and **Not visited**.
3	def solve(r, c, path):	**Defines** the recursive function to find paths from (r, c).
4	if r == n-1 and c == n-1:	**Base Case**: Checks if the rat reached the destination.
5	paths.append(path.copy())	Saves a copy of the valid path found.

6	return	Ends recursion for this path.
7	vis[r][c] = 1	**Mark Visited**: Marks the current cell as visited for the current path exploration.
8	for dr, dc, move in [(1,0,'D'), (0,1,'R'), (-1,0,'U'), (0,-1,'L')]:	Loops through all four possible directions (Down, Right, Up, Left).
9	nr, nc = r+dr, c+dc	Calculates the coordinates of the next cell.
10	if safe(nr, nc):	Checks if the move to (nr, nc) is safe.
11	path.append((nr, nc))	Adds the new cell to the current path list.
12	solve(nr, nc, path)	**Recurse**: Tries to find the rest of the path from the new cell.
13	path.pop()	**Backtrack**: Removes the last cell from the path list to try the next direction.
14	vis[r][c] = 0	**Backtrack**: Unmarks the current cell as visited, allowing it to be used in other paths.

✂️ Unit 4: Branch and Bound (BB)

Experiment: Travelling Salesman Problem (TSP) Branch & Bound

Line(s)	Code	Explanation
1	def tsp(curr, visited, cost, path, dist, n):	**Defines** the recursive Branch & Bound function.
2	global best_cost, best_path	Declares variables that track the best solution globally.
3	if len(path) == n:	**Base Case**: Checks if the partial route has visited all n attractions.
4	total = cost + dist[curr][0]	Calculates the total tour cost by adding the final leg back to the start (city 0).
5	if total < best_cost:	Checks if this complete tour is better than the current best.
6, 7	best_cost = total; best_path = path + [0]	Updates the global best solution.
8	return	Ends recursion for this complete tour.
9	for nxt in range(n):	Loops through all possible next cities (nxt) to visit.
10	if not visited[nxt]:	Checks if the city has not been visited yet.
11	new_cost = cost +	Calculates the accumulated

	dist[curr][nxt]	cost if we travel to nxt.
12	if new_cost < best_cost: # branch pruning	**Pruning Step**: If the current partial cost is less than the current best complete tour, continue exploring.
13	visited[nxt] = True	Marks the next city as visited.
14	tsp(nxt, visited, new_cost, path + [nxt], dist, n)	**Recurse**: Explores the route with the next city added.
15	visited[nxt] = False	**Backtrack**: Unmarks the city as visited to allow other branches to explore it later.

Experiment 16: 0/1 Knapsack Branch & Bound

Line(s)	**Code**	**Explanation**
1	def bound(node, n, W, items):	**Defines** the upper bound calculation function (the heuristic).
2	if node.weight >= W: return 0	If the current weight exceeds capacity, the maximum future value is 0.
3-8	val = node.value... while i < n and wt + items[i][0] <= W:	**Greedy Full**: Adds the next unexamined items completely until capacity is

		reached (as they have the best ratio).
9-11	if i < n: val += (W - wt) * (items[i][1] / items[i][0])	**Greedy Fractional**: Adds a *fraction* of the very next item to fill the remaining capacity, establishing the **upper bound**.
12	return val	Returns the calculated upper bound.
13	while queue:	**BB Loop**: Processes nodes from the queue (BFS exploration).
14	cur = queue.pop(0)	Extracts the current node to be explored.
15	if cur.bound < best_value:	**Pruning Check**: If the upper bound is less than the current actual best value, skip this branch.
16	continue	Skips (prunes) the sub-tree rooted at cur.
17-20	next_level = cur.level + 1... w, v = items[next_level]	Identifies the next item to consider.
21-27	# Include item (left child)... if left.bound > best_value: queue.append(left)	**Inclusion Branch**: Creates a node where the item is included. If its **bound** is better than best_value, it's added to the queue.
28-33	# Exclude item (right child)... if right.bound > best_value:	**Exclusion Branch**: Creates a node where the item is excluded. If its **bound** is

	queue.append(right)	better than best_value, it is added to the queue.

🧭 Unit 5: Heuristic Search

Experiment: A* Heuristic Search

Line(s)	Code	Explanation
1	def heuristic(u, v, coords):	**Defines** the heuristic function $h(u, v)$ (estimated remaining cost).
2-4	if coords and u in coords... return math.hypot(x1-x2, y1-y2)	If coordinates are available, calculates the straight-line **Euclidean distance** as the heuristic.
5	return 0	If no coordinates, returns 0 (A* defaults to Dijkstra's algorithm).
6	def astar(graph, src, dst, coords=None):	**Defines** the A* search function.
7	pq = [(0+heuristic(src,dst,coords), 0, src, None)]	**Initializes** the Priority Queue with the source node. The key is $f = g + h$.
8	dist = {src:0}	**Initializes** the actual cost (g) map.
9	while pq:	**Search Loop**: Continues

		until the queue is empty.
10	f,g,u,par = heapq.heappop(pq)	**Extracts** the node u with the minimum total estimated cost f.
11-13	if u in visited: continue ... visited.add(u); parent[u]=par	Checks if u has been finalized; if not, marks it as visited and records the parent.
14	if u==dst:	**Goal Check**: If the destination is reached, stop the search.
15-18	# reconstruct path... return list(reversed(path)), dist[dst]	Reconstructs the shortest path.
19	for v,w in graph.get(u,[]):	Iterates over neighbors v of the current node u.
20	ng = g + w	Calculates ng, the new actual cost (g) to reach v via u.
21	if v not in dist or ng < dist[v]:	**Relaxation Check**: If a shorter path to v is found.
22	dist[v]=ng	Updates the distance (g).
23	heapq.heappush(pq, (ng + heuristic(v,dst,coords), ng, v, u))	**Adds** v to the PQ, keyed by the new $f = g + h(v, \text{dst})$.