

Assessment Schedule – 2025**Scholarship Digital Technologies (93604)****Judgement Statement**

Performance Descriptors	Outstanding Scholarship		Scholarship		Just Below Scholarship		Below Scholarship
	8	7	6	5	4	3	2, 1, 0
Analysis, synthesis, and integration Candidates integrate their knowledge of programming and algorithmic thinking to design and develop solutions to complex unfamiliar problems.	Demonstrates exceptional problem-solving ability, creating informed and concise algorithmic solutions. Algorithms demonstrate optimal algorithmic design and scalability.		Demonstrates strong problem-solving skills and implements well-thought-out approaches to problems. Algorithms are well-structured and go beyond simple solutions.		Demonstrates understanding of the problems and identifies key steps towards creating solutions. Functional algorithms are developed, with some consideration for accuracy and efficiency.		Demonstrates minimal or no understanding of the problems. Basic algorithms are developed, but may not completely or accurately solve the problems.
Coding and efficiency Candidates demonstrate an understanding of efficient coding practices by developing solutions that optimise both time and space complexities.	Code is elegant, efficient, and highly readable. All three questions are completed using the most efficient solutions.		Code is well-written, containing no logic errors. Code functions correctly for all questions, but may contain some minor inefficiencies.		Code successfully handles sample data sets in at least two of the three questions. Code may be functional but contain inefficiencies.		Limited code or an incomplete attempt in one or more of the questions. Code is written, but may contain logic errors or fail to produce the correct output in one or more problems. Code may be simplistic or brute force only solutions.
Critical reflection Candidates show critical reflection on their chosen solutions by contrasting and comparing with other potential solutions and showing insightful observations about the quality and efficiency of their solution.	Provides strong, independent justification, with critical reflection on alternative approaches and problem-solving decisions.		Provides sound justification for algorithm choice and refinements.		Justifies key decisions, but lacks depth in algorithm selection or rationale.		Limited justification. Some reflection on decisions, but justification is weak or unclear. Does not suggest alternative solutions.

Evidence

Note: For EACH question, indications of sufficiency for 'Below Scholarship', 'Scholarship', and 'Outstanding Scholarship' collectively cover ALL parts of the question.

Question ONE	Evidence	Below Scholarship	Scholarship	Outstanding Scholarship
(a)	The final output will be: [1,2,3,4,5,6,7,8,9].	A range of the following:	A range of the following:	A range of the following:
(b)	The purpose of this algorithm is to sort an array of numbers in ascending order, using the bubble sort technique.	- Correct answer [1,2,3,4,5,6,7,8,9]. - Identifies sort in ascending order a list of integers.	- Correct answer [1,2,3,4,5,6,7,8,9]. - Identifies sort in ascending order a list of integers.	- Correct answer [1,2,3,4,5,6,7,8,9]. - Identifies sort in ascending order a list of integers.
(c)	Total comparisons = $100 * 99 = 9,900$ comparisons. This is because: - first pass: 99 comparisons - second pass: 99 comparisons ... and so on, for 100 passes. The formula is: $n * (n-1)$ where $n = 100$. Therefore: $100 * 99 = 9,900$.	- Correct answers for comparisons. - Shows working / applies formula. - Identifies no change in cost of algorithm if list is already sorted.	- Correct answers for comparisons. - Shows working / applies formula. - Identifies no change in cost of algorithm if list is already sorted.	- Identifies algorithm correctly as bubblesort. - Correct answers for comparisons. - Shows working / applies formula. - Identifies no change in cost of algorithm if list is already sorted.
(d)	Total comparisons = $1,000 * 999 = 999,000$ comparisons. This is because: - first pass: 999 comparisons - second pass: 999 comparisons ...and so on, for 1,000 passes. The formula is: $n * (n-1)$ where $n = 1,000$. Therefore: $1,000 * 999 = 999,000$. What's important to note here is that even though the list is already sorted. The algorithm will still perform all comparisons – it won't know the list is sorted until it completes all passes. No early termination occurs in this implementation. This is a key inefficiency of this particular bubble sort implementation. A more optimised version could include a flag to detect when no swaps occur in a pass, which would allow it to terminate early for already-sorted lists. However, the pseudocode shown doesn't include this optimisation, so it will always	- Correctly identifies cost of algorithm as $O(n^2)$. - Identifies nested loop: $n*n$ with or without -1. - Correctly identifies no change to cost with similar numbers. - Correctly explains why (no optimisations). ≥ 2 positives and ≥ 2 negatives of this algorithm in real-world use. - Writes an accurate quicksort, mergesort, heapsort or improved bubblesort. - May have minor error(s) in algorithm. - May have poor comments. - Correctly identifies cost of own algorithm.	- Explains that a lack of optimisation is why there is no change. - Correctly identifies cost of algorithm as $O(n^2)$. - Identifies nested loop: $n*n$ with or without -1. - Identifies the simplification to n^2 as the dominant term. - Correctly identifies no change to cost with similar numbers. - Correctly explains why (no optimisations). ≥ 2 positives and ≥ 2 negatives of this algorithm in real-world use - Explains suggested positives. - Explains suggested negatives.	- Explains that a lack of optimisation is why there is no change. - Correctly identifies cost of algorithm as $O(n^2)$. - Identifies nested loop: $n*n$ with or without -1. - Identifies the swap and comparisons as constant time operation. - Identifies the simplification to n^2 as the dominant term. - Correctly identifies no change to cost with similar numbers. - Correctly explains why (no optimisations) - Suggests an improvement/ optimisation.

	perform all $n*(n-1)$ comparisons, regardless of the initial order of the items. This demonstrates why bubble sort can be particularly inefficient – it performs the same number of comparisons, regardless of whether the input is sorted or not.	• Explains difference in time complexity.	• Shows perception and insight. • Writes an accurate quicksort, mergesort, heapsort, or improved bubblesort. • May have minor error(s) in algorithm. • May have poor comments. • Correctly identifies cost of own algorithm. • Explains difference in time complexity. • Gives an example, comparing time complexity.	• Defines the improvement with the suggested optimisation. • ≥ 2 positives and ≥ 2 negatives of this algorithm in real-world use. • Explains suggested positives. • Explains suggested negatives. • Shows perception and insight. • Convincing communication. • Writes an accurate quicksort, mergesort, heapsort, or improved bubblesort. • Correctly identifies cost of own algorithm. • Explains difference in time complexity. • Gives an example, comparing time complexity. • Explains difference / similarity in space complexity. • Provides an extremely clear and strong evaluation of alternative algorithm.
--	---	---	---	--

(e)	The algorithm has: 1. An outer loop that runs from 0 to n-1. 2. For each iteration of the outer loop, an inner loop that runs from 0 to n-2. 3. Basic operations (comparisons and swaps) inside the nested loops. Break-down of the cost analysis: <i>Loop analysis</i> • Outer loop executes n times. • For each outer loop iteration, inner loop executes (n-1) times. • This creates n * (n-1) operations. • Which expands to $n^2 - n$ operations. <i>Inside the loops</i> Each iteration has: • One comparison operation (IF data[j] > data[j + 1]). • Potentially one swap operation (constant time). These are constant time operations. <i>Final analysis</i> • Total operations $\approx n^2 - n$. • As n grows larger, the -n term becomes insignificant. • The dominant term is n^2. Therefore, the Big-O notation for this bubble sort is $O(n^2)$.			
(f)	Even with a list of 1,000 identical items, this particular bubble sort implementation would still perform the same number of comparisons, making it an inefficient approach. This is explained below. <i>Given</i> • List size n = 1,000. • All items are identical (e.g. [42, 42, 42, ..., 42]). • Current implementation from the pseudocode.			

	<i>Analysis</i> - Number of comparisons - Same as before: $n * (n-1) = 1,000 * 999 = 999,000$ comparisons. - This is because the algorithm doesn't have any optimisation to detect identical elements. - Each comparison (IF data[j] > data[j + 1]) still executes. - Key differences from sorting distinct values While the number of comparisons remains the same (999,000) no swaps will ever occur, because no element is greater than another. This saves some execution time on swap operations. However, the comparison cost still dominates, keeping it $O(n^2)$. - Potential optimisations not present The current algorithm lacks: - early termination if no swaps occur in a pass - recognition of identical elements - any way of skipping unnecessary comparisons. - Inefficiencies - Still performs all comparisons, despite identical values. - Makes n passes through the array unnecessarily. - Cannot break early, even though the list is technically 'sorted' from the start. This case highlights a limitation of this basic bubble sort implementation – it will perform its full $O(n^2)$ operations, even in cases where it's clearly unnecessary. A more optimised sorting algorithm could, with identical elements, terminate early when no swaps are needed, and reduce the number of comparisons needed.			
--	---	--	--	--

	For a list of 1,000 identical items, an optimised algorithm could potentially determine if the list is sorted in $O(n)$ time with a single pass, but this implementation will still require $O(n^2)$ operations, making 999,000 comparisons regardless of the data content. This is a good example of why algorithm optimisation and considering edge cases is important in practical implementations.			
(g)	<i>Positive implications</i> 1. Simplicity and understanding • Very easy to understand and implement. • Good for educational purposes. • Simple to debug, due to straightforward logic. • Low cognitive overhead for developers. 2. Memory efficiency • In-place sorting algorithm. • Only requires $O(1)$ additional memory space. • No need for temporary arrays or significant extra storage. • Good for memory-constrained environments. 3. Stable sorting • Maintains relative order of equal elements • Important for certain applications where order preservation matters. • Reliable results for equal elements. 4. Good for small datasets • Performs reasonably well on very small lists (< 50 elements). • Simple implementation might outweigh performance concerns for tiny datasets. • Low overhead for minimal data.			

	<i>Negative implications</i> - Performance issues $O(n^2)$ time complexity makes it extremely inefficient for large datasets. - As shown earlier, sorting 1,000 items requires 999,000 comparisons. - Doesn't scale well – doubling input size quadruples processing time. - Much slower than $O(n \log n)$ algorithms such as QuickSort or MergeSort. - Resource utilisation - High CPU usage, due to excessive comparisons. - Poor use of modern CPU cache mechanisms. - Not optimised for current hardware architectures. - Energy inefficient, due to unnecessary operations. - Business impact - Could cause significant delays in processing large datasets. - Higher operational costs, due to increased processing time. - Potential user dissatisfaction with slow performance. - Scalability issues as data grows. - Practical limitations - No early termination for already sorted lists. - Performs unnecessary work on partially sorted data. - Same performance regardless of input characteristics. - Not suitable for real-time applications with large datasets.			
--	--	--	--	--

	<i>Recommendations for real-world use</i> 1. Appropriate use cases: • Teaching and learning sorting concepts. • Very small datasets (< 50 elements). • When code simplicity is more important than performance. • Prototyping or proof of concept work. 2. Avoid using for: • production systems with large datasets • performance-critical applications • real-time processing requirements • systems that need to scale. 3. Better alternatives • Use built-in sorting functions (usually implement QuickSort or MergeSort). • Consider TimSort for Python applications. • Use HeapSort for memory-constrained environments. • Implement QuickSort for general-purpose sorting needs. 4. If bubble sort must be used: • add optimisation for early termination • consider hybrid approaches for larger datasets • implement bounds checking • add monitoring for performance impacts. The key takeaway is that while bubble sort has its place in education and simple applications, its use in production environments should be carefully considered, due to its performance characteristics. For most real-world applications, more efficient sorting algorithms or built-in sorting functions would be more appropriate choices.			
--	--	--	--	--

(h)	Example <pre>□ // Quicksort implementation- O(n log n) average case. FUNCTION quickSort(array, low, high) IF low < high THEN // Find the partition index pivotIndex = partition(array, low, high) // Recursively sort the left part quickSort(array, low, pivotIndex - 1) // Recursively sort the right part quickSort(array, pivotIndex + 1, high) ENDIF END FUNCTION // Partition function that places pivot in correct position FUNCTION partition(array, low, high) // Choose the rightmost element as pivot pivot = array[high] // Index of smaller element i = low - 1 // Compare each element with pivot FOR j FROM low TO high - 1 DO IF array[j] <= pivot THEN // Increment index of smaller element i = i + 1 swap array[i] with array[j] ENDIF ENDFOR // Place pivot in its correct position</pre>			
------------	---	--	--	--

	<pre> swap array[i + 1] with array[high] RETURN i + 1 END FUNCTION // Helper function to swap elements FUNCTION swap(array, i, j) temp = array[i] array[i] = array[j] array[j] = temp END FUNCTION // Initial data data = [8,5,7,4,1,2,6,3,9] // Call quickSort with initial parameters quickSort(data, 0, 8) // 8 is length(data) - 1</pre>			
--	---	--	--	--

(i)	Quicksort: Often the fastest in practice, due to its low constant factors. It has an average time complexity of $O(n \log n)$, but a worst-case of $O(n^2)$ if a poor pivot is consistently chosen, though this is rare with good pivot selection. Merge Sort: Has a guaranteed worst-case time complexity of $O(n \log n)$ and is a stable sort. It is not an in-place algorithm, requiring $O(n)$ additional memory. Heapsort: Has a guaranteed worst-case time complexity of $O(n \log n)$ and is an in-place algorithm with $O(1)$ space complexity. <i>Comparison with other algorithms</i> - vs MergeSort: - QuickSort: Better cache performance, in-place. - MergeSort: Stable, guaranteed $O(n \log n)$. - vs HeapSort: - QuickSort: Better average case performance. - HeapSort: Better worst case guarantee. - vs BubbleSort ($O(n^2)$): - QuickSort is significantly faster. - For $n=1,000$: - BubbleSort: ~1,000,000 comparisons - QuickSort: ~10,000 comparisons.			
-----	---	--	--	--

Question TWO	Evidence	Below Scholarship	Scholarship	Outstanding Scholarship
(a)	<i>Example</i> <pre> □ grid = [['*', '*', '#', '*', '*', '*'], ['#', '*', '#', '*', '#', '*'], ['*', '*', '*', '*', '#', '*'], ['*', '#', '#', '*', '*', '*'], ['*', '*', '*', '#', '*', '#'], ['*', '#', '*', '*', '*', '*']] #expected output 10 </pre>	A range of the following: - Creates a square 2d grid with * and # strings. - Creates an accurate expected output for the grid. - Describes the path accurately, with directions. - Explains BFS or a less optimal solution (Dijkstra's, DFS / Brute Force).	A range of the following: - Creates a square 2d grid with * and # strings. - Creates an accurate expected output for the grid. - Describes the path accurately, with directions. - Clearly explains an optimal solution (BFS). - Shows understanding of the algorithm.	A range of the following: - Creates a square 2d grid with * and # strings. - Creates an accurate expected output for the grid. - Describes the path accurately, with directions. - Identifies that there could be more than one path. - Accurately explains an optimal solution (BFS).
(b)	The goal is to go from top-left (0,0) to bottom-right (5,5) moving only on '*' cells, with only horizontal / vertical moves. This is one shortest path [cell indices shown as (row, col)]: - Start: (0,0) (0,1) (1,1) (2,1) (2,2) (2,3) (3,3) (3,4) (4,4) (5,4) - End: (5,5). Not including the starting point, there are 10 steps to get to (5,5). There are other routes that also reach the goal, but all other routes take more steps than this one.	- Shows understanding of the algorithm. - Writes a solution in any language. - May have minor error(s) in algorithm. - May have poor comments. - No queue used or commented. - Identifies some of queue, visited set, 2d array for grid, directions as list / struct / tuple. - Identifies BFS time complexity $O(n^2)$. - Explains how the code does or might be changed to return a value on no path.	- Shows insight into the problem. - Writes a BFS in any language. - May have minor error(s) in algorithm. - May have poor comments. - Identifies some of queue, visited set, 2d array for grid, directions as list / struct / tuple. - Justifies each data structure identified. - Identifies BFS time complexity $O(n^2)$. - Identifies BFS space complexity $O(n^2)$. - Explains how the code does or might be changed to return a value on no path.	- Shows comprehensive understanding of the algorithm. - Shows insight into the problem. - Writes a BFS in any language. - Identifies queue, visited set, 2d array for grid, directions as list / struct / tuple - Justifies each data structure identified. - Identifies time complexity $O(n^2)$. - Identifies space complexity $O(n^2)$. - Gives at least two examples of increasing size inputs. - Explains how the code does or might be

		• Identifies a weighted graph problem when # has traversal cost. • Identifies that BFS no longer works.	• Shows insight into the problem and its solution. • Identifies a weighted graph problem when # has traversal cost. • Identifies that BFS no longer works.	changed to return a value on no path. • Shows insight into the problem and its solution. • Explains clearly and convincingly. • Identifies a weighted graph problem. • Identifies that BFS no longer works. • Identifies that Dijkstras should be used instead.
(c)	The optimal solution to problems like these is to use a Breadth-First-Search (BFS) algorithm. BFS is ideal for finding the shortest path in an unweighted grid, because it explores all possible paths level by level. Here is a step-by-step breakdown of how to implement a BFS for this problem. This is a concise breakdown of a BFS implementation, likely for finding the shortest path in a grid. 1. Set-up • Use a Queue (FIFO) for cells to visit and a Visited Set to track coordinates, preventing cycles. • Initialise the search by enqueueing the start cell (0,0) and marking it visited. 2. Explore While the Queue isn't empty: • Dequeue the current cell (v). • If v is the destination, return its path length. 3. Advance For each of the four neighbours (u) of v: • If u is in-bounds, valid ('*'), and unvisited, mark u as visited and enqueue u with an incremented path length. 4. End If the loop finishes without reaching the destination, return - 1 (or handle initial invalid cases).			

(d)

□#This is an optimal solution in Python

```

from collections import deque

def shortest_path(grid):
    n = len(grid)
    # Directions: up, down, left, right
    directions = [(-1,0), (1,0), (0,-1), (0,1)]
    visited = [[False]*n for _ in range(n)]
    queue = deque()

    # Start from (0,0) if it's a valid position
    if grid[0][0] != '*':
        return -1

    queue.append((0, 0, 1)) # (row, col, path_length)
    visited[0][0] = True

    while queue:
        row, col, length = queue.popleft()

        # Check if we've reached the bottom-right cell
        if row == n-1 and col == n-1:
            return length

        for dr, dc in directions:
            r, c = row + dr, col + dc
            if 0 <= r < n and 0 <= c < n and not visited[r][c]
and grid[r][c] == '*':
                visited[r][c] = True
                queue.append((r, c, length+1))

    return -1 # If no path exists (should not happen as per
assumption)

```

	<pre> # Sample input grid = [['*', '*', '#', '*'], ['*', '#', '*', '*'], ['*', '*', '*', '#'], ['#', '*', '*', '*']] print(shortest_path(grid)) # Output: 6 □ </pre>			
(e)	For the shortest path grid problem, the main data structures used are: Queue (from collections.deque in Python) Purpose: Implements Breadth-First Search (BFS), the optimal algorithm for finding the shortest path in an unweighted grid. Justification: BFS explores all possible routes layer by layer, guaranteeing that the first time you reach the goal, it's via the shortest possible route. A queue is essential for BFS, because it ensures that nodes (grid cells) are explored in the correct order (first-in, first-out). Visited Matrix (visited 2D list) Purpose: Keeps track of which grid cells have already been explored. Justification: Prevents revisiting the same cell, which would lead to redundant processing and potential infinite loops. Ensures each cell is only processed once, optimising memory and time usage. Grid (input data) Purpose: Represents the problem's terrain, distinguishing traversable ('*') and blocked ('#') cells.			

	Justification: Direct look-up to determine if a move to a cell is valid. Directions list Purpose: Stores the possible movement directions (up, down, left, right). Justification: Simplifies the code for neighbour exploration.																							
(f)	Time complexity Standard BFS is $O(V+E)$ where v is the number of vertices and e is the number of edges. In this case: • each cell in the grid is visited at most once • for an $N \times N$ grid, there are N^2 cells • each cell has up to 4 neighbours (up, down, left, right) • total time complexity: $O(N^2)$ (assuming most cells are traversable, i.e. '*'). Space complexity Standard BFS is $O(V)$ where v is the number of vertices. In this case: • the queue and the visited matrix both store at most N^2 entries • total space complexity: $O(N^2)$. Summary table <table><tr><th>Grid size</th><th>Number of cells</th><th>BFS steps (worst case)</th><th>Memory needed</th></tr><tr><td>4 x 4</td><td>16</td><td>≤ 16</td><td>Tiny</td></tr><tr><td>100 x 100</td><td>10,000</td><td>$\leq 10,000$</td><td>~100 KB</td></tr><tr><td>1,000 x 1,000</td><td>1,000,000</td><td>$\leq 1,000,000$</td><td>~10 MB</td></tr><tr><td>10,000 x 10,000</td><td>100,000,000</td><td>$\leq 100,000,000$</td><td>~1 GB</td></tr></table>	Grid size	Number of cells	BFS steps (worst case)	Memory needed	4 x 4	16	≤ 16	Tiny	100 x 100	10,000	$\leq 10,000$	~100 KB	1,000 x 1,000	1,000,000	$\leq 1,000,000$	~10 MB	10,000 x 10,000	100,000,000	$\leq 100,000,000$	~1 GB			
Grid size	Number of cells	BFS steps (worst case)	Memory needed																					
4 x 4	16	≤ 16	Tiny																					
100 x 100	10,000	$\leq 10,000$	~100 KB																					
1,000 x 1,000	1,000,000	$\leq 1,000,000$	~10 MB																					
10,000 x 10,000	100,000,000	$\leq 100,000,000$	~1 GB																					

(g)	How the solution handles no path • The BFS algorithm would attempt to explore all reachable cells starting from (0, 0). • If the destination (N-1, N-1) is never reached (because it is blocked off or surrounded by '#'), the queue would eventually become empty. • At this point, the algorithm would return -1 (as per the code: <code>return -1</code> at the end). Discussion can include mentions of visited set to avoid an infinite loop, the code queuing grid squares, and taking the 'First in First out' (FIFO). Therefore, when the queue is empty and they haven't reached the destination, there must NOT be a viable path.			
(h)	If the # symbols can be traversed but cost 4 (while * still costs 1), the problem becomes a weighted shortest path problem on a grid. This has the following impact on the approach: BFS is no longer optimal: • BFS works only for unweighted graphs (all moves have the same cost). • With varying costs, BFS may return a sub-optimal path, since it doesn't account for path cost – just the number of moves. Use Dijkstra's algorithm: • You must use Dijkstra's algorithm, which finds the minimum-cost path in a weighted graph. • Dijkstra's algorithm uses a priority queue to always explore the lowest-cost path first. Summary With cell costs, you must use Dijkstra's algorithm and a priority queue to ensure you always take the path with the lowest total cost, not just the smallest number of steps.			

Question THREE	Evidence	Below Scholarship	Scholarship	Outstanding Scholarship
(a)	The optimal approach is Dynamic Programming (Bottom-Up). Dynamic Programming (DP) approach <i>State definition:</i> Let <code>dp[i]</code> represent the minimum number of boxes needed to pack exactly <code>i</code> baked goods. <i>Base case:</i> Set <code>dp[0] = 0</code>, since zero goods require zero boxes. <i>Recurrence</i> For each total <code>i</code> (from 1 to N), and for each box size <code>b</code> in <code>boxes</code>: - If <code>i - b >= 0</code>, update: $dp[i] = \min(dp[i], dp[i - b] + 1)$ <i>Result</i> After filling the DP array, if <code>dp[N]</code> is still infinity (or a sentinel value), it is impossible to pack exactly N. Otherwise, <code>dp[N]</code> is the answer. Dynamic Programming: Top-Down with memoization <i>State definition:</i> Let <code>solve(i)</code> be a recursive function that returns the minimum number of boxes needed to pack exactly <code>i</code> baked goods. We use a <code>memo</code> array (e.g. <code>memo[i]</code>) to store the result of <code>solve(i)</code> once it is computed. <i>Base case:</i> Inside the <code>solve(i)</code> function: - If <code>i = 0</code>, return 0 (zero goods need zero boxes). - If <code>i < 0</code>, return infinity (an impossible path). <i>Recurrence:</i> Inside the <code>solve(i)</code> function: - If <code>memo[i]</code> is already computed (not a sentinel value), return <code>memo[i]</code>.	A range of the following: - Describes a solution [optimally Dynamic Programming (Bottom-Up), but may describe DFS / Brute Force]. - Writes a solution, which may be a sub-optimal algorithm. - May have minor error(s) in the algorithm. - May have poor comments. - Explains how the approach reaches the correct answer. - Breaks down the approach into steps. - Correct answer for time complexity. - Accurate analysis of their algorithm. - Shows understanding of the cost of their algorithm. - Explains why greedy fails. - Gives an example of how it fails. - Names an alternative solution (Top-Down / Bottom-Up / BFS / Brute Force / Greedy / Algebraic).	A range of the following: - Accurately describes a solution [optimally Dynamic Programming (Bottom-Up)]. - Shows sound understanding of the algorithm. - Writes an accurate solution (optimally DP Bottom-Up or Top-Down with memoization). - May have minor error(s) in their algorithm. - May have poor comments. - Explains how the approach reaches the correct answer. - Breaks down the approach into steps. - Explains ≥ 2 key strengths / weaknesses of the approach. - Correct answer for time complexity [optimally $O(n*b)$ for DP / BFS]. - Accurate analysis and understanding of the cost of their algorithm. - Explains why greedy fails. - Gives an example of how it fails. - Names an alternative solution (Top-Down /	A range of the following: - Accurately describes a solution [optimally Dynamic Programming (Bottom-Up)]. - Shows comprehensive understanding of the algorithm. - Shows insight into the problem. - Writes an accurate solution [optimally DP (Bottom-Up)]. - Explains how the approach reaches the correct answer. - Breaks down the approach into steps. - Explains ≥ 2 key strengths / weaknesses of the approach. - Shows comprehensive understanding of the approach. - Uses convincing communication in describing the approach. - Correct answer for time complexity [optimally $O(n*b)$ for DP / BFS]. - Correct answer for space complexity [optimally $O(n)$ for Bottom-Up DP / BFS]. - Accurate and comprehensive analysis

	• Otherwise, compute the result: <code>result = 1 + min(solve(i - b))</code> for all box sizes <code>b</code> in <code>boxes</code>. • Store this <code>result</code> in <code>memo[i]</code> before returning it. <i>Result:</i> The final answer is the return value of the initial call, <code>solve(N)</code>. If <code>solve(N)</code> returns infinity, it is impossible to pack exactly <code>N</code>, so return -1. <i>Summary:</i> By using dynamic programming, we ensure that every possible subtotal is considered with the fewest boxes, guaranteeing the optimal (smallest number) solution. These two methods are $O(n*b)$ where <code>n</code> is the target number of baked goods and <code>b</code> is the number of sizes of boxes. Any other method is considered inferior.	• Describes how the alternative will solve the problem. • Compares the alternative to their original in cost.	Bottom-Up/BFS/Brute Force/Greedy/Algebraic). • Describes how the alternative will solve the problem. • Compares the alternative to their original in cost. • Provides at least one advantage OR one disadvantage of the alternative. • Shows some understanding of both solutions.	and understanding of the cost of their algorithm. • Shows perception and insight in analysis. • Demonstrates convincing communication in their answer. • Explains why greedy fails. • Gives an example of how it fails. • Names an alternative solution (Top-Down/Bottom-Up/BFS/Brute Force/Greedy/Algebraic). • Describes how the alternative will solve the problem. • Compares the alternative to their original in cost. • Provides at least one advantage OR one disadvantage of the alternative. • Shows comprehensive understanding of both solutions. • Clear and persuasive communication.
(b)	<i>Example</i> □#Optimal solution - Bottom up dynamic programming <pre>def min_boxes(boxes, N): # Dynamic programming array; dp[i] = min boxes to pack i # baked goods dp = [float('inf')] * (N + 1) dp[0] = 0 # Zero baked goods needs zero boxes</pre>			

	<pre> for i in range(1, N + 1): for box in boxes: if i - box >= 0 and dp[i - box] != float('inf'): dp[i] = min(dp[i], dp[i - box] + 1) return dp[N] if dp[N] != float('inf') else -1 # Example usage: print(min_boxes([3, 5, 7], 11)) # Output: 3 print(min_boxes([2, 4], 7)) # Output: -1 </pre>			
(c)	Bottom-Up Dynamic Programming approach <i>Idea</i> The bottom-up method builds the solution from smaller subproblems up to the target value (N). We compute the minimum number of boxes needed for every possible total of baked goods from 0 up to N using the available box sizes. <i>Step-by-Step process</i> Initialisation: • Create a DP array: <code>dp = [0, inf, inf, ..., inf]</code> of length <code>N+1</code>. • <code>dp[i]</code> represents the minimum number of boxes needed to pack exactly <code>i</code> baked goods. • Set <code>dp[0] = 0</code> (zero goods require zero boxes). Building the table For each total <code>i</code> from 1 to N: • For each box size <code>b</code> in <code>boxes</code>: ◦ If <code>i - b >= 0</code> (i.e. we can use this box size): ◦ Update: <code>dp[i] = min(dp[i], dp[i-b] + 1)</code>. This means: To make <code>i</code>, if we use box <code>b</code>, we need one more box than the minimum to make <code>i-b</code>.			

<i>Final step:</i> • After filling in the table, <code>dp[N]</code> tells us the minimum number of boxes to pack exactly <code>N</code> baked goods. • If <code>dp[N]</code> is still infinity, it means that it's impossible to pack exactly <code>N</code> using the given box sizes. <i>Key strengths</i> • Every possible combination of boxes to make every subtotal is considered. • The minimum is updated at each step, so the best (smallest number of boxes) solution is always recorded. • No subproblem is missed, and no combination is skipped. • No call stack is used, like in the recursive solution. • The algorithm simply writes a value in array (very fast and memory-efficient). • However, harder to understand than top-down. Top-Down with memoization <i>Approach</i> • This solution uses recursion to explore all combinations of box sizes to pack exactly <code>N</code> baked goods. • Memoization is used to store results for subproblems so we don't recompute them, making the algorithm efficient. <i>How it works</i> Recursive function The recursive function (<code>dp(remaining)</code>) computes the minimum number of boxes needed to pack <code>remaining</code> baked goods. Base cases: • If <code>remaining == 0</code>: We need 0 boxes (no goods left to pack). • If <code>remaining < 0</code>: This is impossible (overshot the target), so return infinity. <i>Try all options</i> • For each box size, try to pack one box and recursively solve for the smaller problem: <code>remaining - box</code>. • For every choice, calculate the total boxes used (<code>res + 1</code>). • Track the minimum over all choices.			
---	--	--	--

	<i>Memoization</i> • Store the result for each remaining value in a dictionary (memo) after computing it. • If we encounter the same remaining value again, simply return the cached result instead of recomputing. <i>Final step</i> After recursion, check if the minimum found is infinity (impossible). If so, return -1; otherwise, return the minimum. <i>Key strengths</i> • Easier to understand than bottom-up. • Easier to implement than bottom-up. • Every possible combination of boxes to make every subtotal is considered. • Without memoization, the same subproblems are solved repeatedly, which is inefficient. • Memoization ensures each subproblem is solved only once, reducing the time complexity drastically. • However, it runs slow in the real world, due to call stack overhead.			
(d)	Optimal Dynamic Programming: Bottom-Up <i>Time complexity</i> • Let N be the target number of baked goods. • Let B be the number of box sizes [len(boxes)]. <i>Main loop:</i> <pre> for i in range(1, N + 1): for box in boxes: # update dp[i] </pre> • The outer loop runs N times. • The inner loop runs B times for each <i>i</i>. • Each step inside is O(1). <i>Total operations:</i> O(N * B).			

Space complexity

- The DP array `dp` has size $N + 1$.
- No other data structures scale with N or B .

Total space:

$O(N)$.

Summary table

Complexity	Value
Time	$O(N * B)$
Space	$O(N)$

Efficiency discussion

- Time efficiency: This is very efficient for reasonable values of N (e.g. up to tens or hundreds of thousands) and small to moderate B . It is much faster than brute-force recursion, which is exponential in N .
- Space efficiency: The space usage is minimal, just a single array of size $N+1$.

Conclusion

The bottom-up solution is efficient and scalable for practical input sizes. Its $O(N * B)$ time complexity makes it suitable for problems where both N and B are not excessively large.

Second best: Top-Down with memoization*Time complexity*

Let: N = target number of baked goods

B = number of box types `[len(boxes)]`.

How many unique subproblems?

- Each recursive call is for a distinct value of remaining (from N down to 0).
- There are $N + 1$ possible values for remaining (0, 1, ..., N).

For each subproblem, the function iterates through all box sizes (B).

	<i>Total work</i> Each subproblem (unique remaining) is solved only once (thanks to memoization), and for each, we do $O(B)$ work. So, time complexity = $O(N * B)$. Space complexity (with Memoization storage): – the memo dictionary stores up to $N + 1$ entries. <i>Call stack</i> In the worst case, the recursion stack can be up to N deep (if we always subtract the smallest box size). <i>DP storage + call stack:</i> – $O(N)$ memo + $O(N)$ call stack = $O(N)$ space. <i>Summary table</i> <table><tr><th>Complexity</th><th>Value</th></tr><tr><td>Time</td><td>$O(N * B)$</td></tr><tr><td>Space</td><td>$O(N)$</td></tr></table> <i>Efficiency discussion</i> • With memoization, the top-down approach is just as efficient as bottom-up DP, since each subproblem is solved at most once. • The recursive structure is sometimes easier to write or reason about, but both approaches scale similarly. <i>Contrast:</i> • Without memoization: Exponential time, lots of repeated work. • With memoization: Polynomial time, efficient and scalable. <i>Conclusion</i> The top-down memoized solution is efficient, with time and space complexities of $O(N * B)$ and $O(N)$ respectively, making it suitable for large inputs.	Complexity	Value	Time	$O(N * B)$	Space	$O(N)$			
Complexity	Value									
Time	$O(N * B)$									
Space	$O(N)$									
(e)	A greedy solution always chooses the largest possible box size that fits into the remaining baked goods, repeating this process until all goods are packed or no box fits. While this strategy is fast and sometimes produces the optimal answer, it does not guarantee the smallest number of boxes in all cases.									

<i>Why Greedy fails: Example breakdown</i> Example from above: • Boxes: [3, 5, 7] • Target: $N = 11$. Greedy approach: 1. Pick the largest box ≤ 11: 7 Remaining: $11 - 7 = 4$. 2. Pick the largest box ≤ 4: 3 Remaining: $4 - 3 = 1$. 3. No box fits into 1 $\rightarrow$ failed to pack exactly 11. <i>Optimal solution (DP or brute force):</i> • Use boxes: $3 + 3 + 5 = 11$. • Number of boxes: 3 (could also be $7+3+1$ if 1 were an option). <i>Another example:</i> • Boxes: [2, 3] • Target: $N = 6$. Greedy approach: $3 + 3 = 6$ (2 boxes: this works, and is optimal here). But for targets like $N = 7$: • $3 + 3 = 6$, left with 1 (no box for 1), but • $2 + 2 + 3 = 7$ (3 boxes, if possible). <i>Key reason greedy fails</i> • Greedy doesn't look ahead: By always picking the largest box, it can leave a remainder that cannot be filled with available box sizes, even if using smaller boxes earlier would have made the total possible. • Combination possibilities: The optimal solution may require a combination of smaller boxes that greedy overlooks. <i>Summary</i> Greedy fails because it may lead to a dead end (unpackable remainder) and does not consider all possible combinations. Only dynamic programming or exhaustive search guarantees the smallest number of boxes for every target.			
--	--	--	--

(f)	The Perfect Pastry Packing problem is a variant of the coin change or minimum partitioning problem. There are several other ways to solve this type of problem. Brute Force recursive (Top-down, no Memoization) <i>How it works</i> Recursively try every combination of boxes to pack the goods. Complexity: • Time: $O(B^N)$ (exponential; B = box types, N = goods). • Space: $O(N)$ (recursion stack). Advantages: • Easy to implement for small N and B. • Finds the correct answer. Disadvantages: • Infeasible for large N or B. • Extremely slow, due to repeated subproblems. Top-Down recursive with Memoization (DP) <i>How it works</i> Like brute force, but caches partial results. Complexity: • Time: $O(N * B)$. • Space: $O(N)$. Advantages: • Guarantees the smallest number of boxes. • Efficient, even for large N. Disadvantages: • Requires extra memory for caching. • Slightly more complex to implement than brute force. Bottom-Up Dynamic Programming <i>How it works</i> Builds up the answer in a DP array from 0 to N. Complexity:			
-----	--	--	--	--

• Time: $O(N * B)$. • Space: $O(N)$. Advantages: • Efficient and easy to reason about. • No recursion stack issues. • Guarantees optimal result. Disadvantages: • Requires array of size $N+1$. • May be less intuitive for some than recursion. Greedy approach <i>How it works:</i> Always pick the largest box possible for the remaining goods. Complexity: • Time: $O(N * B)$ (worst-case, but usually better in practice). • Space: $O(1)$. Advantages: • Simple and fast. • Good for specific cases (like coin denominations with certain properties). Disadvantages: • Does NOT always give the optimal answer. • Can lead to dead ends (see previous examples). Mathematical / algebraic approaches <i>How it works:</i> Try to use number theory (Diophantine equations) or integer programming. Complexity: Variable – can be NP-hard for arbitrary box sizes. Advantages: • May work for special cases. • Useful for formal proofs of possibility/impossibility.			
---	--	--	--

Disadvantages:

- Not practical for general box sizes.
- Complex for implementation.

BFS (Breadth-First search)

How it works:

Treat each number of goods as a node; try to reach N from 0 by adding box sizes.

Complexity:

- Time: $O(N * B)$.
- Space: $O(N)$.

Advantages:

- Guarantees optimal answer.
- Useful for finding all possible packing combinations.

Disadvantages:

- Similar to DP in practice.
- Less cache-friendly than bottom-up DP.

Evaluation

Method	Correctness	Optimality	Efficiency	Ease of implementation	Scalability
Brute Force	Yes	Yes	Poor	Easy	Poor
DP Top-Down	Yes	Yes	Good	Moderate	Good
DP Bottom-Up	Yes	Yes	Good	Moderate	Good
Greedy	Sometimes	No	Good	Easy	Good
BFS	Yes	Yes	Good	Moderate	Good
Algebraic	Sometimes	Sometimes	Varies	Hard	Varies

	Conclusions • Dynamic Programming (top-down with memoization or bottom-up) is the best general-purpose solution – it is efficient, guarantees optimality, and is scalable. • Brute force is suitable only for very small inputs. • Greedy is not reliable unless box sizes have specific mathematical properties (like coins with canonical denominations). • BFS is theoretically sound and can be useful for variants of the problem. • Algebraic approaches are more suitable for theoretical analysis than for implementation.			
--	--	--	--	--

Cut Scores

Scholarship	Outstanding Scholarship
13 – 19	20 – 24