

No part of the candidate's evidence in this exemplar material may be presented in an external assessment for the purpose of gaining an NZQA qualification or award.

S

93604

SCHOLARSHIP EXEMPLAR



Mana Tohu Mātauranga o Aotearoa
New Zealand Qualifications Authority

Scholarship 2025 Digital Technologies

Time allowed: Three hours

This assessment has THREE questions. Attempt ALL questions.

Page 1

Make sure you have the paper Resource Booklet 93604R.

INSTRUCTIONS

This assessment has THREE questions, each relating to a different programming problem. You should attempt all three questions.

Each question has several parts. These must all be answered in as much detail as possible.

Code can be written in:

- Pseudocode
- Python 3
- C++
- C
- C#
- Java
- JavaScript.

Code cannot be run, so care must be taken to manually debug and check accuracy. The markers will take appropriate steps to ensure this is considered.

QUESTION ONE

Consider the following pseudocode:

```
data = [8,5,7,4,1,2,6,3,9]
```

```
n = length(data)
```

```
FOR i FROM 0 TO n - 1:
```

```
    FOR j FROM 0 TO n - 2:
```

```
        IF data[j] > data[j + 1] THEN:
```

```
            swap data[j] with data[j + 1]
```

```
        ENDIF
```

```
    ENDFOR
```

```
ENDFOR
```

```
PRINT data
```

(a) What is the final output of this algorithm?

B *I* U

[1, 2, 3, 4, 5, 6, 7, 8, 9]

(b) What is the purpose of this algorithm?

B *I* U

sort numbers in ascending order (bubble sort)

(c) How many comparisons would it take on a list of 100 items in random order?

B *I* U

$100(100 - 1) = 9900$ comparisons

(d) How many comparisons would it take on a list of 1,000 items in sorted order?

B *I* U

$1000(1000 - 1) = 999000$ comparisons

(e) What is the cost of this algorithm, and how did you come to this conclusion?

B *I* U

Algorithmic cost is expressed in terms of time and space complexity.

- time: $O(n^2)$
- space: $O(1)$ auxillary

The runtime of the algorithm scales with the square of n (length of input) as there are 2 nested for loops with a range dependent on the size of n . Each for loop repeats operations roughly n times. So the time complexity will be $n * n$ or n^2 . This tells us that doubling the size of the input will approximately quadruple the runtime of the algorithm.

The memory consumption of the algorithm is described by the size of the input and the extra memory the it allocates. This extra memory usage is what is called auxillary memory and is often reflected in the space complexity of the algorithm. Since bubble sort swaps numbers in place without creating a new array or list that is dependent on the size of n , it has a constant or $O(1)$ space complexity. Numbers can be swapped using XOR or addition-subtraction trick and strings can be swapped using a temporary variable dedicated for that sole purpose. Although the "data" consumes $O(n)$ memory the actual cost of the algorithm is calculated using auxillary memory which is $O(1)$.

- (f) What would the implications be for the cost of this algorithm when run on a list of 1,000 items that all had the same value?

B I U     

The algorithm does not terminate early using checks so it would execute both loops and perform all $O(n^2)$ comparisons to completion despite the uniformity of the data. There would be 999000 comparisons instead of only 999 required to check if the data is already sorted which has time complexity of $O(n)$. A list of numbers that are all the same is considered to be a sorted list because there exists no permutation of items in the list that is unique. The algorithm could reduce cost by implementing an $O(n)$ check to see if the list is already sorted to bring down best case time complexity to $O(n)$ instead of $O(n^2)$.

- (g) What are the implications, both positive and negative, of using this algorithm in a real-world setting?

B I U     

Sorting algorithms are important for real world settings as many operations are more efficient on sorted, organised data. For example, binary search allows systems to find a number in a sorted collection of numbers (not necessarily consecutive or free from duplicates). This concept of binary search also applies to humans as we unknowingly use it to find a word in a dictionary by comparing the position of the first letter in the alphabet. This would only work if the dictionary was sorted lexicographically. The time complexity of binary search is $O(\log n)$ as it halves the search space each step. Unsorted data forces us to use linear search which is $O(n)$ as in the worst case you have to check every item for a match.

However, this sorting algorithm is mathematically inefficient. The fastest time complexity can be achieved in $O(n \log n)$ by the best comparison sort algorithms which is much more efficient than bubble sort. Using bubble sort in the real world would result in lots of wasted compute time whcih would waste energy in the form of electricity. It may lead to unnecessary trade-offs that discourage us from sorting data and opting for a linear search in place of binary search.

- (h) Write a more efficient algorithm to achieve the same purpose.

data = [8,5,7,4,1,2,6,3,9]

B I U     

Optimised Integer Bubble Sort

```
n = len(data)
swapped = True
upper = n - 1
```

```

swapped = True
upper = n - 1

while swapped and upper > 0: # while list is unsorted and all sublists have not been checked
    swapped = False
    for i in range(upper):
        if data[i] > data[i + 1]: # numeric swapping with no temp variables
            data[i + 1] += data[i]
            data[i] = data[i + 1] - data[i]
            data[i + 1] -= data[i]
            swapped = True # swap has occurred so list may be unsorted
    upper -= 1 # the last element is guaranteed to be max so sort sublist

print(data)

```

(i) Evaluate the efficiency of your algorithm compared to the original one.

B I U     

My algorithm still has $O(n^2)$ time and $O(1)$ space complexity. However, it performs better on average due to improvements in the constant factor of the time complexity. It also has a significantly better best case time complexity of $O(n)$ due to the addition of the "swapped" variable which checks whether a swap has occurred to determine whether further comparisons are required. If no swaps occurred during an iteration, it means the data is sorted so the algorithm can terminate early. For a sorted list of n elements only $n - 1$ comparisons need to take place. The decreasing "upper" bound for each iteration roughly halves the number of comparisons required, making it 2 times faster on average. I could have implemented merge sort for $O(n \log n)$ complexity but it would've taken more time to write it.

(b) Describe the route your second sample takes that gives the provided output.

B I U     

My sample has no walls (#) so is a blank grid. There are 8 choose 5 possible routes to take but only the distance matters. Since there are no obstacles placed in such a way that prevents only down and right moves, the distance will be the minimum of $2(N - 1) = 8$. Any route consisting of only down and right moves will achieve the minimum of 8 moves.

(c) Explain how you might go about solving this problem.

B I U     

The grid can be modelled using a graph data structure. The shortest path through the graph can then be found using BFS as the graph is essentially unweighted. The walls (#) will be excluded from the graph as it is not possible to traverse them. This leaves only the free (*) squares in the graph. Adjacent cells will have an edge connecting them in the graph, allowing a BFS to find the single source shortest path.

(d) Write a solution to the above problem in your chosen language.

B I U     

BFS (Python)

```
from collections import deque
```

```
N # given
grid # given
```

```
q = deque()
q.append([0, 0, 0]) # push the top left * cell
```

```
visited = [(False) * N) for _ in range(N)]
dist = [(0) * N) for _ in range(N)]
```

```
delta_x = [0, 0, 1, -1] # x axis adjacency
delta_y = [1, -1, 0, 0] # y axis adjacency
```

```
while q: # while queue not empty
    x, y, d = q.popleft()
    visited[x][y] = True # mark as visited to avoid infinite loops
    dist[x][y] = d # update distance from source
    if x == N - 1 and y == N - 1: break # terminate if reached bottom right
    for dx in delta_x:
        nx = x + dx
        if 0 <= nx < N: # check x axis bounds
            for dy in delta_y:
                ny = y + dy
                if 0 <= ny < N: # check y axis bounds
                    if not visited[nx][ny] and grid[nx][ny] != "#": q.append([nx, ny, d + 1]) # only add unvisited non # nodes
```

```

if 0 <= nx < N: # check x axis bounds
    for dy in delta_y:
        ny = x + dy
        if 0 <= ny < N: # check y axis bounds
            if not visited[nx][ny] and grid[nx][ny] == " ": q.append([x, y, d + 1]) # only add unvisited non # node

print(dist[N - 1][N - 1])

```

(e) What data structure(s) did you use to solve the problem? Justify your decision(s).

B I U $\equiv$ $\vee$ $\equiv$ $\vee$ $\leftarrow$ $\rightarrow$?

I used a double ended queue data structure to perform breadth first search on the graph. The most efficient data structure for BFS is a queue and a double ended queue satisfies the requirements for a queue. Queues are first in first out data structures that perform **dequeue** (remove from front) and **push** (add to back) operations in $O(1)$ time complexity. Lists on the other hand have $O(n)$ time complexity for **dequeue** operation which can significantly decrease speed for large input sizes. Lists have to remove the element at index 0 and then shift all elements over by 1 index which is what causes the $O(n)$ time. Although graphs typically rely on a list or dictionary data structure to represent the graph as an adjacency list or adjacency matrix, I decided to traverse the graph implicitly to save memory and time constructing an explicit graph. Cells that are orthogonal implicitly share an edge as shown in the **delta_x** and **delta_y** code part.

(f) Explain, with examples, how your approach would scale to larger grid sizes.

B I U $\equiv$ $\vee$ $\equiv$ $\vee$ $\leftarrow$ $\rightarrow$?

My BFS algorithm has a time complexity of $O(N^2)$ as it has to traverse every cell in the grid in the worst case to find a valid path. For the sample grid in the question all $4^2 = 16$ cells have to be traversed as there are no completely enclosed spaces in the grid. There is no heuristic guiding the BFS so in a blank maze it will search through all squares before it gets to the bottom right. For larger grids, it may be possible for my algorithm to perform better than $O(N^2)$ if there are long walls and completely enclosed regions, reducing the number of possible reachable nodes within a distance of $2(N - 1)$ which is the minimum required to reach the bottom right corner in a blank grid.

For example, in the grid (* has been replaced with + due to italics):

```

+ # ^ ^ ^
+ # ^ ^ ^
+ # ^ ^ ^
+ # # # #
+ + + + +

```

There is only a single path to the bottom right corner and none of the ^ get traversed, meaning for grids in this pattern, the time complexity is $O(N)$.

(g) What would **your** solution return if the sample data did not have a viable path to the target? Discuss.

B I U $\equiv$ $\vee$ $\equiv$ $\vee$ $\leftarrow$ $\rightarrow$?

My solution would return 0 as all the distances in the "dist" 2d array are initialised to 0 before the algorithm runs. I chose 0 as the default value as it is obviously impossible for the distance to be 0 if $N > 1$. It may have been more appropriate to initialise it to -1 to account for the $N = 1$ edge case but it would possibly mean having to use signed integers which halve the maximum value that can be stored.

(h) What if the # symbols could be traversed, but they would 'cost' a value of 4. How would this impact your approach?

B I U $\equiv$ $\vee$ $\equiv$ $\vee$ $\leftarrow$ $\rightarrow$?

If the # squares could be traversed, BFS would no longer return the optimal path as it would ignore regions enclosed in #. Dijkstra's algorithm would need to be used to account for the varying cost of # compared to *.

The queue would need to be replaced with a priority queue and edge weights would need to be assigned depending on whether a * or # was encountered. The $d + 1$ would be $d + 4$ for #. Priority queues store items with the lowest value at the front but this added complexity comes with an $O(\log n)$ cost instead of $O(1)$ seen in queues.

My current algorithm would likely overestimate the cost of travelling to the bottom right.

Page 3

QUESTION THREE

Consider the following problem.

Perfect pastry packing

You run a bakery that sells pastries that are all the same size. These pastries are supplied to customers in different sized boxes. Calculate the smallest number of boxes that you can use to perfectly pack a number of pastries.

You are given an array of boxes where `boxes[i]` represents how many pastries can be stored in that size of box. You have an unlimited supply of boxes. You will be given a number `N` that represents how many pastries you need to pack.

Write a function that returns the minimum number of boxes needed to perfectly pack `N` pastries. If it is not possible to perfectly pack `N` pastries, the function should return `-1`.

Example 1:

`boxes = [3, 5, 7]`

`N = 11`

Output:

3

Explanation: The best way to pack 11 baked goods is using $5 + 3 + 3$ (3 boxes in total).

Example 2:

`boxes = [2, 4]`

`N = 7`

Output:

-1

Explanation: It is not possible to pack exactly 7 baked goods with boxes of size 2 and 4.

(a) Describe your approach to solving this problem to ensure you have the smallest number of boxes.

B I U     

This problem is a variation of the coin change problem in which the target sum should be achieved using the least number of "coins" or "boxes" in this case. I will use recursion to solve smaller subproblems and use it to piece together a solution for the original `N`.

(b) Write a solution to this problem in your chosen language.

```
B I U     
INFINITY = 10 ** 9 # impossible to achieve N
memo = [-1] * (N + 1) # list of -1 to indicate value has not been calculated yet

def solve(n):
    if n < 0: return INFINITY
    if n == 0: return 0
    if memo[n] != -1: return memo[n]
    counts = []
    for b in boxes:
        counts.append(solve(n - b) + 1)
    memo[n] = min(counts)
    return memo[n]

count = solve(N)
if count >= INFINITY: print(-1)
else: print(count)
```

(c) Explain how your solution solves the problem.

```
B I U     
My solution breaks down the problem into several smaller subproblems. The algorithm initially takes N and then performs a depth first search by subtracting the value of box 0 from N at each step until the remaining number of pastries is either 0 (solved) or less than 0 (not possible through this path). Once it hits those base cases, the recursive function returns 0 to indicate 0 pastries can be solved with 0 boxes or INFINITY to indicate it is impossible to achieve. This value bubbles up back to the parent function which then takes the minimum of all recursive paths. The reason I used infinity was so that evaluating the minimum would be simpler as INFINITY would never be the minimum number of boxes unless all paths were impossible.

Taking a look at N = 11 and boxes = [3,5, 7], the algorithm finds the solution as 3 by calling solve(11) -> solve(8) -> solve(5) -> solve(0).
```

(d) Analyse your solution in terms of its efficiency.

```
B I U     
There are 2 other possible paths other than 11, 8, 5, 0 as discussed before for the sample case due to the 3 different permutations of 2 size 3 boxes and 1 size 5 box: 3, 3, 5; 3, 5, 3; 5, 3, 3. This is called redundancy, leading to naive DFS to have an exponential time complexity. Each step, all box sizes have to be evaluated recursively. For a large number of boxes, the runtime increases rapidly as multiple combinations have to be considered. It can be assumed that the time complexity is  $O((N/\min(\text{boxes}))^{\text{len}(\text{boxes})})$  where  $\text{len}(\text{boxes})$  is the number of boxes and  $\min(\text{boxes})$  is the size of the smallest box. With dynamic programming, it is possible to eliminate duplicate combinations by using a memo. When solve(3) gets called in the order solve(11) -> solve(8) -> solve(3) it will memoize the solution for solve(3) so that when the branch solve(11) -> solve(6) -> solve(3) gets called the solution is already stored and no further calculations have to be carried out. This marginally improves the time complexity but it's hard to get a closed form expression for this as it highly depends on both N and the boxes but it can be proved that the time complexity approaches  $O(N/\min(\text{boxes}))$  as duplicate combinations are eliminated, decreasing the effect of the exponent  $\text{len}(\text{boxes})$ .
```

(e) Given the examples supplied, explain why a 'greedy' solution would not provide an accurate result.

B I U   ↶ ↷ 🔄

Greedy solutions only work in edge cases. For example after sorting boxes by size if a box was double the size of the previous, a greedy solution would work as there would be no reason to use a smaller more than once, eliminating the need to search for combinations. However, this is often not the case and smaller sized boxes need to be used multiple times to total a specific N. In the first sample case where $N = 11$, a greedy solution would use the 7 box, be left with 4, then use the 3 box and be left with 1. The greedy solution will incorrectly output - 1 and not make use of the 5 box. This is a combinatorial problem dealing with discrete sums, so greedy solutions are unlikely to work.

(f) This problem could be solved in several ways. Describe how else it could be solved, and evaluate your solution against those other methods.

B I U   ↶ ↷ 🔄

An iterative approach could be used instead of recursion. However, iterative dynamic programming for this problem is much more complicated to arrive at or implement. The code would be longer and be harder to implement but it would achieve better runtime despite having the same time complexity. This is because recursive algorithms often have the overhead of pushing and popping from the function call stack.

The other way people might solve this is by generating all possible combinations of which there are roughly $(N/\min(\text{boxes}))^{\text{len}(\text{boxes})}$ in the worst case. For example, for $N = 11$, you can calculate combinations that use (3) 0, 1, 2, 3 times, (5) 0, 1, 2 times, and (7) 0 or 1 time. That is a total of 24 cases. For larger N, if the sizes of the boxes are relatively much smaller, then the number of combinations of boxes to consider grows exponentially.

Scholarship

Subject: Digital Technologies

Standard: 93604

Total score: 17

Q	Grade score	Marker commentary
One	07	The candidate showed a strong understanding of the given algorithm, its complexity, and its use in the real world. The code was valid and efficient (relative to the original Bubble Sort). It demonstrated a clear understanding of Bubble Sort's mechanics and two key $O(n^2) \rightarrow O(n)$ optimisations, as well as an interesting, albeit unconventional, method for integer swapping.
Two	05	The candidate has correctly identified that a Breadth-First Search is an optimal solution to find the shortest path across the grid. The algorithm they have written for this contains the correct structure and components, although it includes some logical errors. However, the candidate's code is potentially algorithmically invalid, due to three logical failures that would lead to infinite loops or incorrect results: • Infinite loop: Instead of adding neighbours to the queue, the code re-adds the current node repeatedly, causing the algorithm to stall and eventually crash. • Illegal movement: By nesting loops for x and y changes, the code attempts diagonal moves and 'jumps' that violate the rule of only moving to adjacent neighbours (up, down, left, right). • Coordinate swap: A typo calculates the new Y-coordinate using the X-axis value, causing the path to move to incorrect, unrelated cells on the grid. Summary verdict: The implementation attempts a Breadth-First Search (BFS), but contains errors because it never actually traverses the grid, ignores movement constraints, and miscalculates basic coordinates.
Three	05	The candidate has provided a high-quality solution. They have correctly identified this as an optimisation problem suitable for Dynamic Programming and successfully implemented a memoized recursive approach. Feedback for Q3(d): They correctly identified that overlapping subproblems (redundancy) are the issue with the naive solution. However, their complexity analysis has errors. The naive complexity is actually exponential ($O(M^{\text{depth}})$), not polynomial. Furthermore, memoization does not 'marginally' improve this – it drastically reduces it to $O(N * t_{\text{count}(\text{boxes})})$, which is a known closed-form expression. You calculate N distinct values, and for each value, you iterate through the box types once.