

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

OUTSTANDING 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     

To sort a given list of integers in non-decreasing order.

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

B *I* U     

9900 comparisons.

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

B *I* U     

999000 comparisons.

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

B *I* U     

The cost of the algorithm is $O(n^2)$ in time complexity, and $O(1)$ in auxillary space complexity where n is the length of the array data.

The time complexity is $O(n^2)$ because the algorithm loops through the data twice, the outer loop repeats n times while the inner loop repeats $(n-1)$ times.

The auxillary space complexity is $O(1)$ because the algorithm operates in-place, the amount of memory used is not affected by the length of the array data.

- (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     

Because the algorithm has no condition for stopping early, the algorithm will spend a large amount of time performing unnecessary comparisons even though the list is already in non-descending order. This means the algorithm's best-case time complexity is still $O(n^2)$.

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

B I U     

The positive implications of using this algorithm in a real-world setting are that it takes no additional space to run, is simple to implement, and sorts the provided list efficiently if it is small and random. This sorting algorithm is also stable, which means it's guaranteed to preserve the order of equal elements, which is important in some real-world scenarios such as sorting data by first one criteria then another.

The negative implications of using this algorithm in a real-world setting include its high time complexity of $O(n^2)$, which makes its use impractical for extremely long lists due to its slow speed. It also cannot recognise when a list has been sorted and will continue to make unnecessary comparisons even for lists where only one or a few elements are out of place, as is often the case for real-life applications. These things make the algorithm very inefficient for use in many real-world settings.

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

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

B I U     

```
FUNCTION sort(data)
  //check if the list is already sorted, or only has one element
  sorted = TRUE
  FOR i FROM 1 TO length(data) - 1
    IF data[i] > data[i+1]
      sorted = FALSE
      BREAK
  ENDIF
ENDFOR
IF sorted
  RETURN data
ENDIF

half = floor(length(data)/2)
//I assume that for loops in pseudocode have inclusive bounds

//split list into two
l = ARRAY WITH LENGTH half
r = ARRAY WITH LENGTH (length(data) - half)
FOR i FROM 0 TO half-1
  l[i] = data[i]
ENDFOR

FOR i FROM half TO length(data) - 1
```

```

FOR i FROM half TO length(data) - 1
    r[i-half] = data[i]
ENDFOR

//recursively sort halves
l = sort(l)
r = sort(r)

//merge the sorted lists and return the answer.
//two pointers, one for each list, to track which element to add into the answer next
lptr = 0
rptr = 0
ansptr = 0
ans = ARRAY WITH LENGTH length(data)
WHILE lptr < length(l) AND rptr < length(r)
    //put the smaller element first in the answer. If both are equal, put the one from the left list first: this makes the algorithm stable
    IF lptr < length(l) AND l[lptr] < r[rptr]
        ans[ansptr] = l[lptr]
        lptr = lptr + 1
    ELSEIF lptr < length(l) AND l[lptr] > r[rptr]
        ans[ansptr] = r[rptr]
        rptr = rptr + 1
    ELSEIF lptr < length(l) AND l[lptr] = r[rptr]
        ans[ansptr] = l[lptr]
        lptr = lptr + 1
    ENDELSE
        ans[ansptr] = r[rptr]
        rptr = rptr + 1
    ENDIF
    ansptr = ansptr + 1
ENDWHILE
RETURN ans
ENDFUNCTION

PRINT sort(data)

```

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

B I U     

My algorithm has an average and worst case complexity of $O(n \log(n))$ because it splits the list $\log_2(n)$ times, and at each level of splitting it has to process all elements in the list. This is much better than the original algorithm's $O(n^2)$. However, also has an auxillary space complexity of $O(n \log(n))$, which is worse. This trade-off is usually acceptable for most real-world applications, as a high time complexity of $O(n^2)$ is usually much more problematic than some additional space complexity.

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

B I U ☰ ▼ ☷ ▼ ↶ ↷ ?

In words: The route in the second sample goes directly down four times until reaching the bottom, goes right twice, and then goes up, right twice, and down once to move around the blocker to reach the bottom-right corner.

If the steps are represented by the symbols L = left, R = right, U = up and D = down, the optimal path can be described as the string "DDDDRRURRD"

The path represented by numbers, where 0 is the top-left path and increasing numbers represent the path taken:

```
OrderOfTraversal = [
  ['0', '#', 'x', 'x', 'x'],
  ['1', '#', 'x', 'x', 'x'],
  ['2', '#', 'x', '#', 'x'],
  ['3', '#', '7', '8', '9'],
  ['4', '5', '6', '#', '10']
]
```

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

B I U ☰ ▼ ☷ ▼ ↶ ↷ ?

This problem is a shortest-path problem with no edge weights: that is, finding the shortest path from one point to another on a graph where moving from any node to any adjacent node has the same 'cost'. In this case, the nodes are the grid positions that are not blocked, and each node has an edge to up to four other nodes that represent its adjacent grid positions that are not blocked.

This means that it can be efficiently solved with a BFS (breadth first search) algorithm, which starts from the starting position, accesses all positions reachable from those positions, then all new positions reachable from those new positions, repeating until the end position is reached. This is guaranteed to reach the target in the minimal steps, as it first accesses all positions reachable in 1 step, then 2 steps, and so on.

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

B I U ☰ ▼ ☷ ▼ ↶ ↷ ?

```
#include
#include
using namespace std;

//this array of pairs represents the possible movements that can be made: left,up,down or right one step.
pair movement[4]{{0,1},{0,-1},{1,0},{-1,0}};

int main(){
  //The specifics of how input is provided does not matter, but I end up with an integer n, the size of the grid, and a two-
  dimensional array of characters representing the grid as shown in the sample.
  int n;
  cin >> n;
  char grid[n][n];
```

```

char grid[n][n];

for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        cin >> grid[i][j];
    }
}
//actual algorithm begins here

//the array dist records the distance: how many steps it takes to get to a grid coordinate.
int dist[n][n];
//initially set all values to negative 1, to represent that they have not been reached.
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        dist[i][j] = -1;
    }
}
//the distance to the starting cell is obviously 0, because we are there already.
dist[0][0] = 0;
//the pair of integers represents the coordinates of the cell.
queue< q;
q.push({0,0});
while(!q.empty()){
    //get the coordinates to process and remove it from the queue.
    int x = q.front().first, y = q.front().second;
    q.pop();
    //looping through possible positions that can be next
    for(int i = 0; i < 4; i++){
        int newx = x + movement[i].first, newy = y + movement[i].second;
        //check that the new x and y values are inside bounds, the new grid position is not blocked, and that it has not been
        previously visited
        if(newx >= 0 && newy >= 0 && newx < n && newy < n && grid[newx][newy] == "*" && dist[newx][newy] == -1){
            dist[newx][newy] = dist[x][y] + 1;
            q.push({newx,newy});
        }
    }
}
cout << dist[n-1][n-1];
}

```

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

B I U     

I used a queue, a pair(tuple) and a two-dimensional array to solve the problem.

The queue, being a first in first out data structure, works logically to implement BFS, as those pushed into the queue later are naturally processed after those pushed into the queue earlier already are.

The pair of integers represents the coordinates in a logical way, as the x and y components are logically grouped together.

The two-dimensional array is efficient for representing the distance to each grid position, as the size of the array is known in advance and does not change.

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

B I U

My approach has $O(n^2)$ time complexity in the worst and average cases, and a best case time complexity of $O(n)$. For example, if $n = 1000$ on a blank grid, my solution will access every position progressing through the diagonal before finally reaching the target, processing all 1000000 positions.

However it can find the route faster if more positions are blocked directly or indirectly.

For example, if on the same $n = 1000$ grid all positions are blocked except those forming a direct path to the target, my solution will only process at the minimum 1999 positions.

This makes my approach fairly efficient for larger grids as it's not possible to reach a worst case time complexity in general lower than $O(n^2)$ on this problem. However, on certain types of grids with less blockers, it may be solved faster by a guided algorithm such as A* which uses a heuristic to guide the BFS, prioritising movements that move towards the end.

My approach uses $O(n^2)$ auxiliary memory to store the dist array.

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

B I U

If the sample data did not have a viable path to the target, my solution would return -1. The two-dimensional array dist is initialised to -1, representing that a position has not been reached, and is only updated when it is reached through another cell. If there is no viable path to the target, the target position is never reached and so its distance is never updated from -1. Therefore, my solution would output -1.

(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

If the # symbols could be traversed at a cost of 4, this would significantly change my approach. This means that the current BFS algorithm is no longer suitable, as it relies on all movement having the same cost to find the shortest path correctly.

In this case I may modify my approach in two ways:

The first way is using a shortest path algorithm that works on weighted graphs such as Dijkstra's algorithm. However, this will lower my performance as Dijkstra's algorithm must always access the node that has the shortest distance to the starting node. For this a priority queue is used, which is an abstract data structure that can be implemented in various ways (usually a heap) but always takes $O(\log(n))$ time to insert an element where n is the number of elements already in the priority queue. The average time complexity of Dijkstra's algorithm is $O((V+E) \log(V))$ in general, where V is the number of vertices/nodes and E is the number of edges connecting those vertices/nodes. In this case, the average time complexity of using Dijkstra's algorithm is $O(n^2 \log(n^2))$, a noticeable slowdown.

However this is not necessary for this case, as this is not a general shortest-path problem with various weights, instead there are only two weights, 1 and 4. Instead, I can modify my BFS algorithm to add the coordinates with # to the queue, but when they are processed, simply push them back to the back of the queue 3 times, incrementing its own distance each time, and then only actually process the next time, keeping track of how many times I've done this with a separate array. This maintains the same time complexity, although it decreases my performance in the worst case by a factor of 4. However this is much better than the slowdown from using Dijkstra, and is the way I would solve this version of the problem.

I would modify part of my code in the following way: (bold is new code, I've also removed the check for if a coordinate is blocked when choosing whether to add it to the queue)

```
int timeVisitedBlocked[n][n]{0};
while(!q.empty()){
    //get the coordinates to process and remove it from the queue.
    int x = q.front().first, y = q.front().second;
    q.pop();
    if(grid[x][y] == '#' && timeVisitedBlocked[x][y] < 3){
        timeVisitedBlocked[x][y]++;
        dist[x][y]++;
        queue.push({x,y})
        //don't process the cells reachable from this cell yet, so continue
        continue;
    }
    for(int i = 0; i < 4; i++){
        int newx = x + movement[i].first, newy = y + movement[i].second;
        if(newx >= 0 && newx < n && newy < n && dist[newx][newy] == -1){
            dist[newx][newy] = dist[x][y] + 1;
            q.push({newx,newy});
        }
    }
}
```

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     

I would use an iterative dynamic programming approach, where I use the results of previous calculations for the new calculation. The specifics of the algorithm are explained in comments.

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

```
B I U ≡ ∨ ≡ ∨ ↶ ↷ ⓘ
#include
using namespace std;

const int inf = 1000000000
int main(){
    //The specifics of how input is provided does not matter, but I end up with an integer m, the number of boxes, an array
    //representing the sizes of each box, and the integer n, the number of baked goods I must pack.
    int m;
    cin >> m;
    int size[m];
    for(int i = 0; i < m; i++){
        cin >> size[i];
    }
    int n;
    cin >> n;

    //actual algorithm begins here:

    //this is what's called the 'state' in dynamic programming.
    //in this case dp[i] represents how many boxes it takes to perfectly pack i baked goods.
    int dp[n+1];
    dp[0] = 0;
    for(int i = 1; i <= n; i++){
        //set dp at first to an infinity value, as the question asks for the minimum.
        dp[i] = inf;
    }
    //important dp part
    for(int i = 0; i < m; i++){
        //importantly, j iterates forwards. This means that I can use the same box an infinite number of times.
        for(int j = size[i]; j <= n; j++){
            //this line is what's called 'state transfer', it means I'm using previous states to calculate the current one.
            //what this means is: either you leave the best solution as is (possibly no solution yet), or you can form this number of baked
            //goods with the current size of box plus some other combination of boxes that's been previously calculated.
            //the better option of those two is chosen and becomes the new value of dp[j].
            dp[j] = min(dp[j], dp[j-size[i]] + 1);
        }
    }
    //if dp[n] is infinity, that means there was no way to pack exactly n baked goods.
    if(dp[n] == inf){
        cout << -1;
    }else{
        cout << dp[n];
    }
}
```

(c) Explain how your solution solves the problem.

B *I* U

My solution always solves the problem correctly because any state calculation depends on previous states which are guaranteed to be the best possible up to this point.

For the example input, my algorithm runs in the following way:

At the start, the dp array is:

[0,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf]

In the first iteration, the size of the box is 3. After iterating, the dp array becomes:

[0,inf,inf,1,inf,inf,2,inf,inf,3,inf,inf]

In the second iteration, the size of the box is 5. After iterating, the dp array becomes:

[0,inf,inf,1,inf,1,2,inf,2,3,2,3]

In the third iteration, the size of the box is 7. After iterating, the dp array becomes:

[0,inf,inf,1,inf,1,2,1,2,3,2,3]

dp[n] is 3. This is the correct answer to the problem.

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

B *I* U

The time complexity of this algorithm is $O(N*m)$, where N is the number of goods that must be baked and m is the number of boxes. This is because I have a nested loop, the outer loop iterates m times and the inner loop $n+1$ times.

The auxiliary space complexity of this algorithm is $O(N)$, where N is the number of goods. This is because the dp array has size n .

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

B *I* U

A 'greedy' solution will not provide an accurate result because picking the best solution at each step, which is what a greedy algorithm does, does not guarantee the best answer overall in this scenario.

In the given example, a greedy solution may attempt to start by using the largest boxes, then the next largest ones and so on. Such an algorithm will use the boxes of size 7, then a box of size 3, and then conclude that there is no way to perfectly pack 11 baked goods.

(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

This problem may also be solved with a recursive algorithm with memoization, storing the results of previous calls.

$f(x)$ calculates the minimum number of boxes to perfectly pack n baked goods. $f(x)$ would make recursive calls to $f(x-\text{size}[0])$, $f(x-\text{size}[1])$ et cetera, and return the minimum of those function calls + 1. The function would then store the result of $f(x)$ in an array, so that if a function call has already happened before the result can simply be looked up from the array. Then, $f(N)$ would give the answer.

The solution would have similar time and space complexity as my dp solution, however, it would be overall less efficient because of the additional memory and time it takes to manage the function call stack.

Outstanding Scholarship

Subject: Digital Technologies

Standard: 93604

Total score: 22

Q	Grade score	Marker commentary
One	07	Demonstrates clear understanding of the algorithm, its number of operations for a given input, and its deficiencies. The candidate has written an improved algorithm to perform the same task that has a more efficient average time complexity and includes an initial check to avoid running the sort if the list is already sorted. They were able to talk about the trade-offs present in selecting an approach.
Two	08	The candidate showed a good understanding of the problem, identifying that it is a graph traversal problem. To solve the problem, they implemented an accurate Breath-First Search and included comments that justify each of the structures they used. Responses to questions about their approach showed a high level of understanding about the algorithm, and its strengths and drawbacks.
Three	07	The candidate's algorithm to solve the given problem was correct and optimal. They demonstrated a strong understanding of state transitions and how to handle unbounded constraints in dynamic programming. The candidate's analysis and reflection was of an equally high standard.