

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

TOP SCHOLAR



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

or equivalent printed output based on `print(array)` implementation

(b) What is the purpose of this algorithm?

B I U     

This algorithm uses a bubble sort procedure to sort a constant array `data` and output the result. The array is sorted into nondecreasing order in a stable fashion (*stable* in the sense that if there existed indices i and j such that $i < j$ and $\text{data}[i] == \text{data}[j]$ then the element initially at i would still appear before the element initially at j , even in the sorted array. Note that in the constant `data` provided, there are no duplicate keys (assuming a standard greater than comparator) and so it does not matter whether the sort is stable or not).

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

B I U     

The randomness of the 100 items does not matter, as this is an oblivious sort: there are no early exits and each for loop is completed to completion and performs the exact same comparisons. There are n iterations of the outer loop, with $n-1$ comparisons each, where $n=100$, giving 9900 comparisons in total.

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

B I U     

Following on from the argument in (c), let $T(n)=n*(n-1)$ be the number of comparisons, then $T(1000)=999000$ comparisons.

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

B I U     

Note, we use Knuth notation where $g(n)=\Theta(f(n))$ means $g(n)=O(f(n))$ and $g(n)=\Omega(f(n))$.

The algorithm as provided has an asymptotic time complexity of $\Theta(n^2)$, assuming comparisons can be done in $\Theta(1)$ constant time. It also has an asymptotic space complexity of $\Theta(n)$, only to store the input array, and $\Theta(1)$ beyond that, as the sorting is done in place.

The time complexity can be determined by observing that each comparison takes some constant time c , and all non-comparison operations take some time S (for startup). Note that the startup time does not depend on the size of the input, and so the total runtime is $c \cdot (n \cdot (n-1)) + S = cn^2 - cn + S = \Theta(n^2)$. Note that the constant factor does not matter in considering asymptotic time complexities, by definition of the Θ and O notations. In addition, the linear cn term becomes negligible compared to the cn^2 term as n approaches $+\infty$ ($cn/(cn^2) = 1/n$ goes to 0).

- (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     

As this is an oblivious sort, always performing the same comparisons based only on the value of n and not on the actual values of the array, the program would still proceed to perform 999000 comparisons, and still take on the order of 1000^2 time, regardless of the fact that the array was obviously already in nondecreasing order. This means that the algorithm does not utilise features of the specific data input into it.

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

B I U     

Using the algorithm as it is presented in a real world setting would be practically useless, as the 'data' array (or list depending on implementation) is constant before execution, and so the algorithm is deterministic in PRINTing 1,2,3,4,5,6,7,8,9. However, if we were to abstract it into a BUBBLE_SORT procedure, we would be able to run it on arbitrary arrays. This would be useful in an application which might have to sort data, either in something like a spreadsheet, or even in a maps application which would provide possible walking routes sorted by length, or restaurants sorted by rating. However, the $O(n^2)$ complexity is poor, especially in large real world settings involving much larger data (like the restaurants example, or like how there will be exponentially many possible walking routes) which needs to be sorted. In such a large scale example, I do not see why the application would not go with a more efficient algorithm such as mergesort or quicksort, or even use a datastructure such as a max heap in cases which are preprocessed on servers and not often updated (like the restaurant example). Already for even just a million items, the algorithm will likely take over a second and often over a minute on almost any modern hardware, performing 999999000000 comparisons, which is likely too slow to do in realtime in any application, ESPECIALLY when faster options are available.

There are a few positives to using such an algorithm, however. The first is that the array is sorted in place, which means that not a lot of memory is needed ($\Theta(n)$ compared to mergesort's $\Theta(\lg n)$). This makes it very useful for low memory environments such as in embedded systems programming. This alone however is not enough to vouch for the algorithm, as quicksort implementations often also run in-place, requiring only constant storage (or $O(\lg n)$ for recursion stack under a naive implementation, but this is pretty negligible, and can be achieved with less than a constant 50 registers on the device if so needed) beyond the input array, maintaining the same compact memory arrangement but running much faster. The second benefit, however, where bubble sort shines is that it is as aforementioned an oblivious sorting algorithm, requiring only a comparison and a swap to be implemented. This makes it great for systems where branching is not available, and where the same deterministic set of instructions is always executed. In fact, bubble sort matches the optimal complexity of $O(n^2)$ for an oblivious compare-swap sort without any branching.

While the theoretical analysis does suggest that the bubble sort provided becomes much slower for larger input sizes, it does still hold up decently for small arrays such as those with 10 elements in the example. Such arrays are actually very common in real applications (who will look through more than 10 restaurant options?), and bubble sort shines due to its linear iterative structure. Each loop iterates through the array from beginning to end, making this algorithm great for cache locality (on systems with a typical hierarchical cache). In addition, the lack of branching allows the branch predictor to essentially be 100% correct, and lets the CPU run full steam ahead without wasting time on conditional jump instructions. The simplicity of implementation means that one iteration could compile down to two or three instructions, which could be very efficient even if there are $10 \cdot 9 = 90$ iterations, and even resulting in it being faster than other sorting algorithms with a theoretically better complexity.

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

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

```
B I U ☰ ▼ ☷ ▼ ↶ ↷ ⓘ
// C++20 implementation of a more efficient algorithm

/* QUICKSORT FOR REFERENCE
#include

template
void quicksort(T* data, size_t size) {
    // size is required since C-style arrays are only a pointer to the first element.
    // if instead we were using a std::vector, this would not be needed, and implementation
    // would be fairly similar (using std::span instead of passing pointers)

    // base case, single element
    if (size <= 1) {
        return;
    }

    // static is only used so we do not assume the context of the caller has a mersenne twister already
    static std::mt19937 mt{std::random_device{}()}; // for randomising pivot choice
    std::uniform_int_distribution<> dist(0, size-1);
    size_t pivot = dist(mt);
    size_t less = 0; // count how many elements are partitioned to the left
    std::swap(data[pivot], data[0]);

    // partition around pivot
    for (size_t i = 1; i < size; ++i) {
        if (data[i] < data[0] || (data[i] == data[0] && i & 1)) {
            // move the element to the left if it is less than the pivot, or half of the time (odd index) if it is equal to the pivot
            // the equality condition allows us to maintain a good expected split even if the array is initially all equal values
            ++less;
            std::swap(data[i], data[less]);
        }
    }
    // swap the pivot back into place
    std::swap(data[0], data[less]);

    // recurse on both halves
    quicksort(data, less);
    quicksort(data + less + 1, size - 1 - less); // should be TRE optimised (Tail Recursion Elimination) by most modern compilers
}
*/ // QUICKSORT

#include
#include

template
std::vector mergsort(std::span v) {
    if (v.size() <= 1) return v; // base case only one element
    int n = v.size();

    // recursively sort left and right halves
```

```

// recursively sort left and right halves
std::vector left = mergesort(std::span(v, 0, n/2));
std::vector right = mergesort(std::span(v, n/2, (n+1)/2));

int lp = 0, rp = 0; // pointers to current elements in left and right
std::vector res;
while (res.size() < n) {
    if (lp == left.size()) {
        // left list is exhausted so must take from right
        res.push_back(right[rp]);
        ++rp;
        continue;
    }
    if (rp == right.size()) {
        // right list is exhausted so must take from left
        res.push_back(left[lp]);
        ++lp;
        continue;
    }
    if (left[lp] < right[rp]) {
        res.push_back(left[lp]);
        ++lp;
    } else {
        res.push_back(right[rp]);
        ++rp;
    }
}
return std::move(res);
}

```

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

B I U     

Observe that each nonempty call to mergesort can have no more than $O(\lg n)$ 'ancestor' calls. Why? Because going up to an ancestor call will at least double the size of the vector v , and such a doubling can only happen at most $O(\lg n)$ times before the size would exceed n . Further observe that in each 'layer' of the recursion, an index can only appear at most once. Thus, each layer has at most n elements to merge lists from. The list merging operation is $O(n)$, so regardless of how the indexes are distributed in the layer, the merging for a layer takes $O(a+b+c+\dots) = O(n)$ (if $a+b+c+\dots = n$). Then since there are $O(\lg n)$ layers, the time complexity of this algorithm is $O(n \lg n)$. This is much faster than the $O(n^2)$ of the original algorithm even for fairly small n , and I would probably prefer this algorithm even for sorting 100 elements. Note however that this comes at the cost of no longer being in place, with the input vector being duplicated for each layer. This copying still only requires $O(n \lg n)$ space, so the extra memory usage is not too noticeable, even for fairly large n (until n gets really really large, like over a billion).

However, the constant factor behind both complexities is likely higher here than in the bubble sort algorithm. This is due to the overhead of recursion and setting up multiple stack frames. The parent frame is dependent on all subframes and so we cannot employ tail recursion elimination like in quicksort, nor ideally an iterative structure like in bubble sort.

Note that if more was known about the data (such as, is it always a permutation like in the example input?) then smarter strategies such as a countsort could be employed to abuse this information. However, an $O(n \lg n)$ time algorithm is optimal for a sort which only uses comparisons (this is further highlighted by the generic type in the C++ implementation, to avoid trying fancy things about unknown data). This is since any comparison sort defines a decision tree, and one which should work on all $n!$ permutations provided to it. Each comparison in the tree creates two new children, so the 'height' or 'max depth' of the tree is logarithmic in the number of nodes, but $\lg(n!) = \Theta(n \lg n)$ and so there will always exist an input for any comparison based sort which provokes such behaviour, and we can conclude that mergesort as provided is indeed (at least asymptotically) optimal.

It is however important to note for implementation that the mergesort provided is not a stable sort like the bubble sort in the question.

INCOMPLETE QUICKSORT EXPLANATION

Firstly, observe that each element can only be the pivot at most once, after which it is unaffected by any recursive calls. Every comparison includes the pivot, and thus, any two indices i and j in the array can only be compared at most once. Also, observe that the runtime of the algorithm is dominated by the number of comparisons which take place in the main 'for' loop of the function. So we can say that:

$T(n) = O(E(n))$, where $E(n)$ is the expected number of comparisons for an array of length n .
 Note that $E(n) = \sum_{i=0}^{n-1} \left(\sum_{j=i+1}^{n-1} P(i \text{ and } j \text{ are compared}) \right)$
 $= n(n-1)/2 \times \text{mean probability of two indices being compared}$

For any indices i and j , let datastar be the sorted array and their sorted order be istar and jstar , so that $\text{datastar}[\text{istar}] = \text{data}[i]$ and $\text{datastar}[\text{jstar}] = \text{data}[j]$. Then indices i and j are compared if and only if i or j is chosen as a pivot before all k with $\text{istar} < k\text{star} < \text{jstar}$. Intuitively, this makes sense since if k is sorted between i and j , then they will be split into different partition halves during the partition around k , and will never be compared against each other. Then consider the indices $\text{istar} \dots \text{jstar}$. There are $\text{jstar} - \text{istar} + 1$ such indices, and they can be permuted in any order, so the probability that the permutation starts with istar or jstar is:

$2 \times (\text{jstar} - \text{istar} - 1)! / (\text{jstar} - \text{istar} + 1)!$

where $!$ denotes the factorial function.

Simplifying gives $P(i \text{ compared to } j) = 2 / ((\text{jstar} - \text{istar})(\text{jstar} - \text{istar} + 1))$

Page 2

QUESTION TWO

Consider the following problem.

You are given a grid of characters of dimensions N by N. Calculate the fastest way to get from the top left corner to the bottom right corner.

You may only travel through grid positions that contain a '*' symbol and must avoid positions that have a '#' symbol. In each step you can travel right, left, up or down, but not diagonally.

You can assume that there will always be at least one available path to the destination.

Sample input:

```
grid = [  
    ['*', '*', '#', '*'],  
    ['*', '#', '*', '*'],  
    ['*', '*', '*', '#'],  
    ['#', '*', '*', '*']  
]
```

Output:

6

(a) Provide a second, larger, input and expected output data set for this problem.

B I U       

```
grid = [  
    ['*', '*', '*', '#', '#', '*'],  
    ['*', '*', '*', '*', '*', '*'],  
    ['*', '*', '*', '#', '#', '*'],  
    ['*', '#', '*', '#', '#', '*'],  
    ['*', '*', '*', '#', '*', '*'],  
    ['*', '*', '*', '#', '*', '*'],  
    ['*', '*', '*', '#', '*', '*'],  
    ['*', '*', '*', '#', '*', '*']  
]
```

expected_output = 12

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

B I U       

Let (i,j) denote the cell in row i from the top (starting at 1) and column j from the left (with the leftmost column being column 1), so that the goal is to go from cell (1,1) to cell (6,6).

Clearly there is a very large wall in the fourth column, and it can only be travelled to in the second row, so the path must go through cell (2,4). It is possible to go to cell (2,4) from (1,1) in an unobstructed path in 3 steps (down, right, right, right) and it can be shown that no shorter path exists (because the Manhattan distance is 3). Then there exists only one path to the end, going to the right and then snaking down around obstacles. Specifically, the answer steps right right down left down down right. Note that we avoid going onto (1,6) as it is a 'dead end' and would require us to step onto it from (2,6) and then backtrack back to (2,6) gaining us nothing except two extra unwanted steps.

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

B I U     

The grid can be modelled (perhaps implicitly) as a graph, where nodes denote empty cells and edges denote orthogonal adjacencies between empty nodes (as we cannot travel diagonally). Then the problem becomes an SSSP, Single Source Shortest Path problem, where we are interested in the shortest path between the node corresponding to the top left and bottom right corners. The graph is actually very sparse, with no node having more than four edges, and so a typical SSSP algorithm should work very well.

We further notice that each step takes the same much time, whether left right up or down, and so the graph which models this grid is actually unweighted, with all vertices of weight 1. This allows us to take the simpler approach of a BFS compared to a Dijkstra search.

We then notice something as inspired by our example in part (b). In part (b) we realised that stepping on (1,6) was futile, because it meant we got to cell (2,6) in a suboptimal way (which required two more steps). We can utilise something like this in our algorithm, where once a node of the graph has been visited with some distance, we will immediately discard considering visiting it from a node with a higher distance, as this would be suboptimal and that prefix of the path could always be replaced to make a better one.

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

B I U     

```
// C++20 implementation
#include
#include
#include
#include // for std::pair

struct Cell {
    size_t x, y;
};

// grid is passed as a std::vector, where grid[i][j] denotes the 'type' of the cell in row i and column j
// '*' denotes an empty square, and '#' denotes an impassable obstacle
// returns the LENGTH of the shortest path between the corners (NOTE: does not return the path itself)
size_t shortest_path(const std::vector& grid) {
    if (grid.empty()) return 0;
    size_t width = grid[0].size();
    size_t height = grid.size();

    std::queue> q; // stores cells and their distance from UL corner
```

```

std::vector> seen(height + 1, std::vector(width+1, false)); // to avoid backtracking unnecessarily
q.emplace(Cell{0,0}, 0); // it takes 0 time to go to UL corner
while (!q.empty()) {
    auto [cell, distance] = q.front();
    q.pop_front();
    size_t x = cell.x, y = cell.y;
    if (seen[y][x]) continue;
    seen[y][x] = true;
    if (x == width-1 && y == height-1) {
        // reached end tile
        return distance;
    }
    if (x > 0 && grid[y][x-1] == "") q.emplace(Cell{x-1, y}, distance+1);
    if (x + 1 < width && grid[y][x+1] == "") q.emplace(Cell{x+1, y}, distance+1);
    if (y > 0 && grid[y-1][x] == "") q.emplace(Cell{x, y-1}, distance+1);
    if (y + 1 < height && grid[y+1][x] == "") q.emplace(Cell{x, y+1}, distance+1);
}
return -1; // not found; this line should be unreachable as the problem statement specifies that a path always exists
}

```

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

B I U     

In this problem, I used a queue (in particular, the C++ `std::queue` implementation) to store distance data to nodes. This ensures that the graph is traversed in bfs order, that is, that any candidate X with a distance greater than candidate Y will be processed in an order where X is processed later than Y, and in particular, X and Y distances differ by no more than 1. We prove this by strong induction on the iteration:

Proof: Clearly, this is true initially, when the queue only contains the single starting element, and so the base case holds. Then, in any iteration of the while loop, let d be the distance of the top element. After d is removed, only elements with distance $d+1$ are pushed onto the queue. However, we know that the difference between the distances of the first and last elements of the queue is at most 1 by the inductive hypothesis, and so appending elements with distance $d+1$ will preserve the ordering property. Also, the next element of the queue has distance at least d by the ordering property, and so the distance-differs-by-at-most-one property is also preserved after each push operation. This concludes the proof.

What this means for our search is that since we always take elements from the front of the queue and push to the back, the search is conducted in a breadth first manner. This ensures that every time a cell is reached for its first time, the distance to that cell is minimum, and so we can confidently move on without having to revisit cells multiple times. It also means that the first time we visit the end node, we can know for sure that we have found the shortest path and return immediately, without waiting for the rest of the search to conclude.

The reason for using a queue data structure, as opposed to a stack or dynamic vector is that a queue is implemented so that back insertion (enqueue) and front extraction (dequeue) are $O(1)$ operations. This is often done by maintaining start and end pointers and storing the queue in a ring buffer (which can grow like a `std::vector` does as needed, resulting in amortised $O(1)$ push), and with only pointer arithmetic required for the `pop_front()` operation. In contrast, using a vector would take $O(|\text{vector}|)$ for each `pop_front` operation, as a vector must guarantee that the first element resides at its base pointer, thus requiring every element to shift over when a single element is deleted from the front. As enqueueing and dequeuing are the most common operations in the algorithm, and executed in every while loop iteration, we would like these to be as fast as possible, and a queue is the perfect data structure for the job.

In addition, I used a 2-dimensional bit vector (`std::vector` specialised template) to store data about whether each node was visited or not. Using a 2-dimensional vector abuses the natural indexing of nodes in a grid, allowing us to look up and modify entries in the `seen` table in guaranteed $O(1)$ constant time (requiring simple pointer arithmetic). Using the specialised `std::vector` template further compresses the memory required to store this information to a single bit per cell, effectively reducing memory costs by up to a factor of 64 depending on the architecture and implementation. This means that while the space complexity is still $O(n^2)$, it is much lighter in practice due to the constant factor (compared to using for example a `std::vector`), and potentially even more of it could fit in CPU cache, resulting in better cache locality and more cache hits (possibly even beyond just left-right movement).

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

B I U     

In the worst case, the search must visit each node once. Let the grid have height n and width m . Considering any node is $O(1)$ (amortized due to queue operations), and so our algorithm runs in $O(nm)$ time. Why? Because each node can be pushed onto the queue at most 4 times, once by each of its neighbours (and not more, since a node is not fully considered more than once by courtesy of the 'seen' mask), and each push/pop takes $O(1)$ time. In fact, we can prove that it is $\Theta(nm)$, as seen on the example of an empty grid, where all nm cells are set to '*'. As proof, consider the theorem in (e). By that theorem, the end node must be pushed onto the queue later than any other node in the grid, as it has the furthest distance from the origin. In fact, in the empty grid, all distances are simply Manhattan distances, and so the furthest node (x,y) from $(0,0)$ is the one which maximises $x+y$. Thus, all $nm-1$ nodes will be pushed onto the queue before it and the algorithm will take $\Omega(nm)$ time. For this case, as the runtime is both $O(nm)$ and $\Omega(nm)$, it is $\Theta(nm)$.

So we have shown that our time complexity scales as $O(nm)$. However, in practice the algorithm will run a constant factor faster for more dense grids. Such as for example, when there is only one direct path using only Right and Down steps to the exit, only $O(n+m)$ nodes will be pushed into the queue, resulting in an $O(n+m)$ runtime on this grid. Of course, no grid will require less than $n+m$ steps due to the Manhattan distance between the start and end, so the runtime is $\Omega(n+m)$.

The space usage is simply one bit for each cell in the 'seen' vector, as well as at most 4 occurrences of each node in the queue. Thus the space complexity is $\Theta(nm + 4nm) = \Theta(nm)$, and this is regardless of the specific input.

Since both time and space scale as nm , this means that on grids of double the width, the algorithm will run roughly twice as long and use roughly twice as much memory. For grids of maybe half the width but 8 times the height, they will run (roughly) four times as long with four times as much memory and so on.

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

B I U     

My solution would return $(\text{size_t})(-1)$. The specific value depends on the system, as does the size of size_t . However, size_t is an unsigned type and on typical 64-bit architectures will often be a 64-bit unsigned integers. Typecasting -1 to a size_t will then return $2^{64} - 1$. What this should indicate to the user is that no path was found. This is clearly an error value, and must ALWAYS be tested for by the caller of the function, unless they can be absolutely certain that a path exists. A path distance of $2^{64}-1$ does not make much sense. Perhaps a better way to enforce this would be to throw an exception (which MUST be handled) or to use something akin to a Rust Result type. However, the context of this program is unknown, and so -1 is used as a placeholder, with the actual implementation being more of an implementation detail. In fact, this $2^{64}-1$ value is actually already decent in some applications such as sorting mazes by difficulty, where difficulty corresponds to a shortest path. If not explicitly tested for, $2^{64}-1$ is indicating that if a path did exist, it would be effectively infinitely long, and that this maze should be sorted higher than any other.

(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     

Our approach still works, we simply need to push more cells to the queue. In the below snippet, an extra condition can be added to check for walls, and the cells pushed with $\text{distance}+4$ instead of $\text{distance}+1$. However, note that the theorem in part (e) is no longer true, and so we can no longer simply use a queue and guarantee shortest path. Instead, we must use a data structure which ENFORCES this order. The perfect choice is a min heap, which allows insertion in $O(\lg \text{size})$ and removal of the min element (like with q.front()) in $O(\lg \text{size})$ as well. In C++, this can be implemented using a `std::priority_queue` (though care should be taken, as the default ordering for a `std::priority_queue` uses a max heap and not a min heap). Then the complexity of this algorithm becomes

$O(nm \lg(n+m))$ instead of $O(nm)$. However, in practice most grids will likely not contain more than a billion cells, and so the extra log factor will increase the runtime by no more than a factor of 30. The space usage will remain the same. The implementation with a min heap instead of a raw queue is now no longer a BFS, but instead actually exactly an implementation of Dijkstra's algorithm*.

```
// SNIPPET
if (x > 0 && grid[y][x-1] == '*') q.emplace(Cell{x-1, y}, distance+1);
if (x + 1 < width && grid[y][x+1] == '*') q.emplace(Cell{x+1, y}, distance+1);
if (y > 0 && grid[y-1][x] == '*') q.emplace(Cell{x, y-1}, distance+1);
if (y + 1 < height && grid[y+1][x] == '*') q.emplace(Cell{x, y+1}, distance+1);
```

*note that in Dijkstra's original algorithm, the heap uses a DECREASE_KEY operation, so there are at most nm entries in the heap, and not $4nm$. However, this is simply a constant factor and it includes extra overhead which could actually make our algorithm efficiency worse.

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 $\equiv$ $\vee$ $\equiv$ $\vee$ $\leftarrow$ $\rightarrow$ $\textcircled{?}$

It is immediately clear that a greedy approach does not work. For example, the above testcase shows that taking 7 as our first choice would be unwise, as it then becomes impossible to make a total of 11 baked goods. The problem is also clearly very closely related to the unbounded knapsack problem, as well as the subset sum problem. And indeed, it is clear that this problem can also be solved with dynamic programming. Indeed, the problem exhibits optimal substructure. That is, if we would like to make a partial sum of `k` baked goods using say `M` of some prefix of the boxes, and we have found some optimal way to do so, we should always do it that optimal way. Any solution that uses more than `M` boxes to create a similar partial `k` will still be a solution if we replace those particular `>M` boxes with the `M` optimal boxes instead, and so it will definitely not be optimal and does not need to be considered. There are also overlapping subproblems, since for example in the above example, making '8' baked goods cares about how to make '5' baked goods (because we can take a box of size 3 and have 5 left over) but so does making '10' baked goods (because we can take a box of size 5 and have 5 left over). Thus, in an efficient algorithm we should like to avoid recalculating the value for 5 baked goods more than once.

We can define a dp state `dp[i][j]` = minimum boxes required to make EXACTLY `j` using the first `i` types of boxes. Then:
$$dp[i][j] = \min(dp[i-1][j], dp[i][j - \text{boxes}[i]] + 1)$$

There are $O(nA)$ states, where `n` is the number of boxes and `A` is the target sum, and each transition takes $O(1)$, resulting in an $O(nA)$ time algorithm.

My next thought is that since each dp row only depends on the previous one, there is no point in storing all $O(nA)$ dp state values, and we can instead only store the last row, performing the update inline. This reduces memory usage to $O(A)$.

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

```
B I U     
// C++20 implementation
#include

// sets a=b if a is -1 or if b < a
// we do not set if b == 0, because it implies that it came from a (-1) + 1 value in our algorithm
void checkmin(int& a, int b) {
    if (b == 0) return;
    if (a == -1 || b < a) a = b;
}

int minimum_packing(const std::vector& boxes, int target) {
    int n = boxes.size();
    std::vector dp(target+1, -1); // dp[j] contains the minimum boxes needed to hold exactly j, or -1 if impossible
    dp[0] = 0; // it is really easy to take 0 baked goods, this is the base case

    for (size_t box_index = 0; box_index < n; ++box_index) {
        for (size_t carry = boxes[box_index]; carry <= target; ++carry) {
            checkmin(dp[carry], dp[carry - boxes[box_index]] + 1);
        }
    }

    return dp.back();
}
```

(c) Explain how your solution solves the problem.

```
B I U     
The solution effectively builds the dp table described in (a). The transition is as aforementioned, with  $dp[i][j] = \min(dp[i-1][j], dp[i][j - \text{boxes}[i]] + 1)$ . The table may look confusing because of the way it is built in place. This is due to the memory optimisation. Indeed, when  $dp[\text{carry}]$  is compared to  $dp[\text{carry} - \text{boxes}[\text{box\_index}]] + 1$ , it automatically assumes the value of  $dp[\text{carry}]$  from the previous row of the dp (the dp row is not cleared between 'for box_index' iterations). Thus, this line is effectively identical to the transition as written mathematically. The  $\text{carry} - \text{boxes}[\text{box\_index}]$ th value will already be set in this iteration, and thus account for the fact that there are unlimited count of every type of box.
```

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

```
B I U     
The outer loop obviously iterates exactly  $n$  times, and the inner loops iterations can be upper bounded by  $\text{target}+1$  (assuming no boxes have negative size, which would indeed be absurd). The assignment inside each loop iteration is a single line, with a simple call to checkmin(). checkmin() clearly takes  $\Theta(1)$  time, and uses  $\Theta(1)$  space, as it uses no external space beyond the variables passed to the function. Thus, the algorithm as a whole takes  $O(n * \text{target})$  time. Indeed, this is the tightest upper bound we can make without knowing anything about the input data, since if all boxes had size 1 then the inner loop would iterate  $\text{target}$  times in each outer loop iterations, resulting in  $\Omega(n * \text{target}) = \Theta(n * \text{target})$  time.

In terms of space, this algorithm is effectively as efficient as possible. The space complexity is  $O(\text{target})$ , as only a one dimensional dp vector is stored in any iteration. If  $\text{target} = O(n)$ , that is, the target to be made is roughly on the order of boxes, then  $O(\text{target}) = O(n)$ , and we cannot do better since we must at least store the 'boxes' array in the first place.

What is very annoying about such a solution is that it runs in pseudopolynomial time. That is, it is very easy to specify a large target, such as a billion, and our algorithm will nonetheless try to make a billion iterations and try to allocate multiple billion bytes worth of memory. This is regardless of the input size, which is only logarithmically many more bits. Thus, in effect our algorithm is actually exponential in the input size, making it possible that there exist real world applications where it simply is not effective. This dependence on the target can be reduced, but at the cost of an exponential dependence on the number of types of boxes instead. However, for cases where the target is 'reasonable,' this algorithm is the currently best known in the literature.
```

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

B I U     

There are many possible greedy strategies which could be employed. The most obvious, however, is to always try to take the box with the largest capacity, as long as we still stay below or equal to our target.

It is immediately clear that the proposed greedy approach does not work. For example, the provided testcase shows that taking 7 as our first choice would be unwise, as it then becomes impossible to make a total of 11 baked goods. This is not simply an 'edge case' to be hard coded and worked around, but a fundamental flaw in the assumptions that this greedy algorithm makes.

An alternative greedy algorithm, though somewhat contrived, could be to always take the box which ensures the remainder is as highly divisible as possible. The intuition is that highly divisible numbers will be more likely possible to make. For example, in the first example, the algorithm would dictate to start by taking 5, since if we want to make 11 the remainder will be 6, which is divisible by both 2 and 3. Then repeating, we should take 3 and 3, and the greedy algorithm gives the correct answer of 5,3,3. It also correctly identifies that the second example has an impossible packing. However, it fails with the following test case:

1 2 7

target = 8

the greedy algorithm would dictate to take 2, so that the remaining 6 is highly divisible. However, after taking 2 and leaving a remainder of 6, 3 more boxes are required and the greedy algorithm wrongly returns 4, when there is in fact a 2 box solution (1+7) which goes through an 'unfavorable' 7 state. Again, this is a fundamental flaw of the assumptions of this greedy algorithm.

(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     

One solution is to solve it much like our dp solution, but recursively. That is, we call $\text{dp}(n_boxes, \text{target})$, which recursively calls the dp states it needs to calculate. This works due to optimal substructure. If the intermediate results are appropriately memoised (yes that is spelt correctly), and subsequent calls have early exit if the value is known, then this results in an algorithm effectively equivalent to the one provided. It will have identical time complexity, but the space complexity will be $O(n * \text{target})$ instead of $O(\text{target})$, due to possibly a nonzero density of the dp grid being required. While it will have the same time complexity, it will likely run slower in practice, due to the overhead of recursion, as well as poor cache locality and possibly complexity due to the memoisation (depending on implementation, for example overhead of hashing or using a red-black tree).

If a similar approach is used without memoisation, then the time complexity blows up to be exponential in the number of box options, since each 'layer' of the recursion will call two more copies of itself in the previous layer, resulting in 2^n stack frames and 2^n calls. This sudden exponential growth is due to the fact that the algorithm no longer leverages the overlapping subproblems of the problem, and this is a strictly worse algorithm than the one with memoisation.

Of course, a similar dp to ours could be used without the memory efficiency improvement. This would result in another $O(n * \text{target})$ time algorithm, for much the same reasons as our own, but with $O(n * \text{target})$ space. I see no reason to use such an algorithm.

The only possible algorithm which is wholly unrelated to our approach is to brute force on combinations of box types to choose. In order to do this, each box type could be duplicated with 'copies' of itself, which are Once as big, Twice as big, Four times as big, Eight times as big, and so on, until $2^k * \text{boxes}[i]$ would exceed target. Of course, the bigger boxes will 'cost' more to take. This ensures that any combination of number of boxes taken is possible using a 0/1 selection of the new duplicated boxes. It is not hard to see that there will be $O(n * \lg(\text{target}))$ boxes after duplication, and for M items there are 2^M subsets to test, resulting in a time complexity of $O(2^n * (n * \lg(\text{target}))) = O(\text{target}^n)$. This algorithm is strictly worse than our algorithm of $O(n * \text{target})$, but it could be feasible for very small targets. I see no reason to not use our algorithm over this brute force one, however.