Nottingham Trent University
Department of Computer Science
Advanced Software Engineering
SOFT37002

November 2024

Development and Analysis of an Algorithm

Great Wall Problem

Hannah Jones
N1039942

Contents

1 Introduction

The Great Wall Problem presents a software engineering challenge involving the reconstruction of an original sequence of brick symbols from unordered data. This report documents the development and analysis of an algorithm designed to solve this problem efficiently, focusing on the use of test-driven development (TDD) to guide implementation. The algorithm employs custom data structures (a hash table for unsorted bricks and a singly linked list-based deque for the sorted sequence) optimised for the task's requirements. Detailed analysis of time complexity, implementation trade-offs and performance evaluation supports the decisions made, providing a comprehensive exploration of both theoretical and empirical considerations. Parts of the container implementations used in this project were adapted from code provided in the SOFT37002 module repository (Amálio, N.R., 2024). These are also referenced in the source files.

2 Test driven development application

I began the TDD process by creating basic integration tests to compare the actual output file with the expected output files. To facilitate this, I designed a header file for the GreatWall class (Figure 2) containing method declarations but no implementations. This approach allowed me to write tests for the methods before implementing them, adhering to TDD's test-first ethos, as illustrated in Figure 1. The initial test code, shown in Figure 3, includes one test for each test case provided in the test data, enabling straightforward debugging. These tests were categorised as integration tests because they assessed the end-to-end functionality, ensuring all components interacted correctly. This contributed to a breadth of test coverage of the codebase.

Additionally, I developed unit tests for each method within the GreatWall class to evaluate their functionality in isolation. These unit tests provided greater depth in the code coverage, complementing the breadth offered by the integration tests. The unit tests are also shown in Figure 1. To enhance readability and maintainability, I implemented a TestHelper class containing utility functions that streamlined the unit test code.

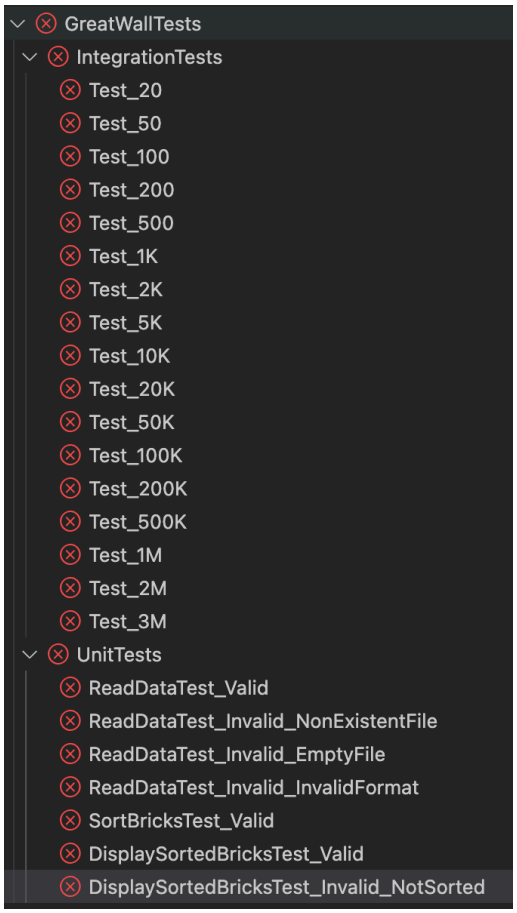

Figure 1 - A screenshot showing the test results of the first iteration of automated tests using the C++ Boost unit testing framework.

GreatWall.h > ...

```h
#include <string>
#include <vector>
using namespace std;

class GreatWall {
    private:
        string inputDataPath;
        // container for unsorted bricks
        // container for sorted bricks;

    public:
        GreatWall(const std::string& filePath);
        void readData();
        void sortBricks();
        string writeSortedBricksToFile() const;
        void displaySortedBricks() const;
};
```

Figure 2 - A screenshot showing the initial header file for the GreatWall class.

```cpp
void runIntegrationTest(const string testCaseName, const string inputFilePath, const string expectedOutputFilePath) {
    cout << "Running test case: " << testCaseName << endl;

    TestHelper::requireFileExists(testDataFilePath + inputFilePath);
    TestHelper::requireFileExists(testDataFilePath + expectedOutputFilePath);

    GreatWall wall(inputFilePath);
    wall.readData();
    wall.sortBricks();
    string actualOutputFilePath = wall.writeSortedBricksToFile();

    BOOST_REQUIRE_MESSAGE(
        TestHelper::areFileContentsEqual(expectedOutputFilePath, actualOutputFilePath),
        "The contents of the files do not match for test case: " << testCaseName
    );
}

BOOST_AUTO_TEST_SUITE(IntegrationTests)

BOOST_AUTO_TEST_CASE(Test_20) {
    runIntegrationTest("20", "20/input-pairs-20.txt", "20/result-sequence-20.txt");
}
```

Figure 3 - A screenshot showing the code for the tests written for the first iteration of TDD.

After setting up these initial tests, I did some analysis and decided to implement a hash table as the data structure for storing the unsorted bricks. I firstly implemented a test suite to test the individual methods of the hash table and then added the method implementations. Fortunately these tests passed first time and the results can be seen in Figure 4.

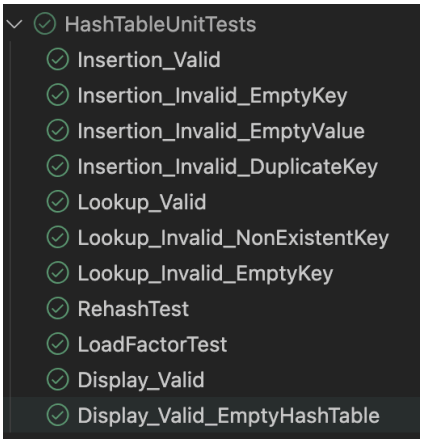

Figure 4 - A screenshot showing the test results of the unit tests testing the hash table class.

Once my hash table had been implemented and successfully tested, I was able to implement the GreatWall::readData() method which reads the data from the input file and adds it to the container of unsorted bricks (in this case, a hash table). I re-ran the full test suite and found that the 'ReadDataTest' test was now passing, as shown in Figure 5.

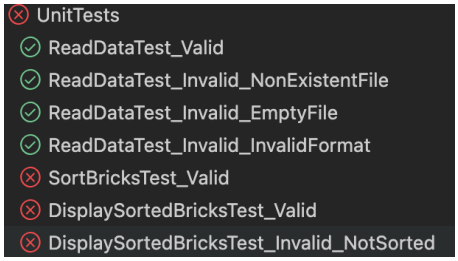

Figure 5 - A screenshot showing the unit test that verifies the correctness of the 'readData' method in the GreatWall class passing.

I then decided to implement a singly linked list as a deque (double-ended queue) as my container for storing the sorted bricks. I firstly implemented a test suite for this new container, based on the use cases I would need for the Great Wall problem (inserting elements at the front and back, and displaying all elements from front to back).

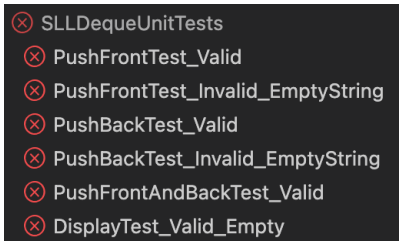

Figure 6 - A screenshot of the unit test results for the SLLDeque class before the method implementation.

I then implemented these methods, re-ran the tests and they passed, as shown in Figure 7.

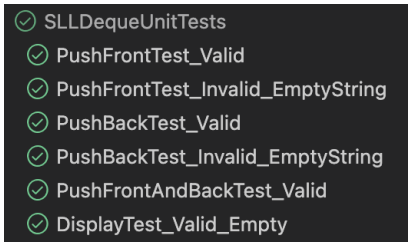

Figure 7 - A screenshot of the unit test results for the SLLDeque class after the method implementation.

After implementing the singly linked list-deque, I implemented the method of the GreatWall class to sort the bricks. I ran the unit test suite again and all the tests now passed - Figure 8.

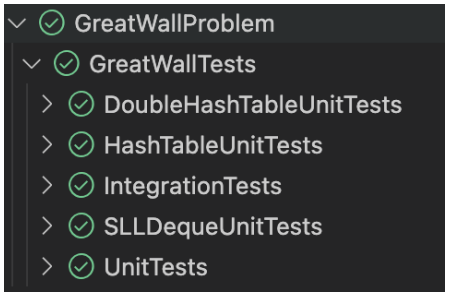

Figure 8 - A screenshot of the unit test results for the GreatWall problem after implementation.

Once the unit tests were all passing, I decided to conduct a few input/output tests by running the main function and manually inspecting the output. An example can be seen in Figure 9.

```
WMAC00FLYWJ:GreatWallProblem hannahjones$ build/main tests/TestData/20/input-pairs-20.txt
XLM
mgR
fYI
LYd
SXW
Tms
bJN
Iuq
XjC
RzX
LPx
FMi
MaZ
dVa
BSC
yxv
fMF
gsA
Hek
Ohe
```

Figure 9 - A screenshot of the output from a manual test of the GreatWall program. This output matches the expected output in result-sequence-20.txt in the provided test data files.

3 Test driven development reflection

Using TDD and automated testing tools for this case study was a valuable learning experience, with both rewarding and challenging aspects. One notable challenge was identifying issues within the test code itself. For instance, in one case, a test incorrectly expected an input file to contain 20 lines instead of 19 (Figure 10), initially leading me to believe there was a bug in my implementation. This meant I spent time debugging my implementation code searching for a bug that didn't exist. However, by resolving this error I deepened my understanding of the problem domain. This experience also highlighted the importance of carefully designing and validating test cases before moving onto the implementation.

```
BOOST_AUTO_TEST_CASE(ReadDataTest) {
    const string inputFilePath = testDataFilePath + "20/input-pairs-20.txt";
    GreatWall wall(inputFilePath);

    BOOST_CHECK_EQUAL(wall.getUnsortedBricks().size(), 20);    check wall.getUnsortedBricks().size() == 20 has failed [19 != 20]
    BOOST_CHECK_EQUAL(*wall.getUnsortedBricks().lookup("mgR"), "fYI"); // first value from input file
    BOOST_CHECK_EQUAL(*wall.getUnsortedBricks().lookup("LYd"), "SXW"); // middle value from input file
    BOOST_CHECK_EQUAL(*wall.getUnsortedBricks().lookup("BSC"), "yxv"); // last value from input file
    BOOST_CHECK_EQUAL(wall.getUnsortedBricks().lookup("nonexistent"), nullptr); // nonexistent value
}
```

Figure 10 - A screenshot of a failing unit test.

I also found writing unit tests before implementation to be rewarding. The sense of accomplishment when tests passed was satisfying and served as a tangible measure of progress. The structure enforced by TDD also encouraged me to think critically about the requirements and behaviour of my code before diving into development. This approach improved the robustness of my implementation by ensuring that functionality was verified incrementally. Additionally, the use of the C++ Boost unit testing framework and a Visual Studio Code extension to visualise test results streamlined the testing process and reduced the effort required to validate multiple test cases. I also liked the fact that once I had implemented my unit tests, I could easily and quickly run them again, which was useful when I refactored my implementation code and wanted to ensure no key functionality was broken.

However, TDD also presented some difficulties. The need to amend tests before making functional changes to the codebase occasionally disrupted my flow of thought. This was particularly noticeable when implementing significant changes or debugging edge cases, as switching between the tests and code felt time-consuming. Furthermore, creating a comprehensive suite of tests to cover all possible scenarios required careful planning and added upfront effort.

Reflecting on this experience, a key takeaway is the critical role of well-designed test cases in ensuring the effectiveness of TDD. Ambiguous or incorrect test logic can lead to unnecessary debugging and delays, emphasising the need for clarity and thoroughness in test design. Another important lesson is that while TDD requires more effort initially, it pays off by catching issues early and reducing debugging time later in the development cycle. Finally, I have come to appreciate TDD's role in enforcing a disciplined and systematic approach to software development, which I plan to apply in future projects.

4 Implementation choices

4.1 Performance trade-offs

4.1.1 Data structure to store unsorted bricks

When designing my system, I considered implementing several data structures. When doing so, there were a number of functions and use cases to consider. For the first data structure to contain the unsorted bricks, I had to consider the performance of two use cases: inserting the bricks into the container and then searching through the container to find each brick. Based on this, I decided to use either a hash table or a self-balancing binary search tree (i.e. an AVL tree, named after Adelson-Velsky and Landis). I ruled out data structures such linked lists and dynamic arrays early on, as linear lookups would be inefficient, with an amortised O(N) time complexity for each brick.

Table 1 - Performance trade offs for using a hash table compared to an AVL tree for storing the unsorted bricks.

	Hash table			**AVL tree**		
	Best	**Average**	**Worst**	**Best**	**Average**	**Worst**
Insertion	O(1)	O(1)	O(N)	O(logN)	O(logN)	O(logN)
Lookup	O(1)	O(1)	O(N)	O(logN)	O(logN)	O(logN)
Overall (N operations)	O(N)	O(N)	$O(N^2)$	O(NlogN)	O(NlogN)	O(NlogN)

The reasons behind the time complexities given in Table 1 are as follows.

A hash table uses an array as its underlying structure, where each entry is accessed via a hash function. Hash tables often use separate chaining (linked lists) or open addressing to handle collisions.

In a hash table, the insertion complexity for the best case is O(1). When there are no collisions, a hash table insertion is O(1). The hash function computes an index in constant time and if the target bucket is empty, the element can be placed immediately. The average case is O(1) because with a good hash function and a load factor maintained below a certain threshold (often around 0.7), collisions are infrequent and most elements can be placed quickly. The worst case is O(N). If many keys hash to the same index (hash collision), all entries may end up in a single bucket (linked list or sequence of probing steps). In this case, insertion may take O(N) in a degenerate case where the bucket contains all elements.

The best case for the lookup complexity in hash table is O(1). Like insertion, if there's no collision, the hash function quickly locates the item in constant time. The average case for lookup is O(1), because with a well-distributed hash function, most lookups are O(1) since few items need to be examined within a single bucket. The worst case is O(N) because where all items collide into one bucket, a lookup involves traversing all items in that bucket, leading to O(N) complexity.

Therefore, the overall complexity for N insertion and lookup operations in the hash table in the best and average case is O(N). For N insertions or lookups, the total time is O(N) since each operation typically costs O(1). The overall worst case for the hash table is $O(N^2)$. This is because if every insertion and lookup takes O(N) (due to severe collision issues), then N operations take $O(N^2)$.

An AVL tree is a self-balancing binary search tree. Each node keeps track of the height difference (balance factor) between its left and right subtrees to maintain balance, ensuring that the tree's height remains O(logN). The tree's height is logarithmic because the balancing operations guarantee that the tree does not become unbalanced, and its height remains proportional to the logarithm of the number of nodes in the tree.

The best, average and worst case for insertion time complexity in an AVL tree is O(logN). Insertion is an O(logN) operation because, to insert a new node, you must traverse from the root down to the appropriate leaf node. Each step in the traversal eliminates half of the remaining nodes, similar to a binary search. Since the tree is balanced, this traversal takes O(logN) time. After inserting the new node, the tree may require

rebalancing, but this rebalancing only affects the height of the tree logarithmically. As a result, even in the worst case, insertion remains an O(logN) operation.

The best, average and worst case for the lookup complexity in an AVL tree is also O(logN). Lookups in an AVL tree always take O(logN) time because the tree is balanced and maintains O(logN) height. To find a specific node, the lookup starts at the root and traverses down the tree, making a decision at each node about whether to go left or right. With the tree being balanced, each step cuts the search space in half, resulting in O(logN) time for the lookup. Even in the worst case, the complexity remains O(logN) because the AVL tree never degenerates into a linear structure, even if extensive rebalancing occurs.

Therefore the overall complexity for N operations in an AVL tree in the best, average and worst case is O(NlogN). Since each insertion or lookup takes O(logN) time due to the tree's balanced structure, performing N operations (insertions and lookups) will take O(NlogN) time in total.

I chose a hash table over an AVL tree because of its superior amortised performance for lookup and insertion operations, which are central to the algorithm. Hash tables excel in these types of operation, offering O(1) average-case complexity which is significantly faster than the O(logN) complexity of AVL trees, making hash tables the more efficient choice for this scenario.

While the worst-case complexity of hash tables is O(N) due to potential collisions, this can be mitigated by using an effective hash function and managing the load factor. In contrast, AVL trees guarantee O(logN) performance even in the worst case, but their balancing operations during insertions add overhead that was unnecessary for this algorithm. Therefore, the hash table provided a better solution for this use case.

4.1.2 Data structure to store sorted bricks

When deciding on the data structure for the unsorted bricks, I had to consider three use cases:

- Insertion at the front
- Insertion at the back
- Traversing all elements from front to back (for displaying).

The structures I considered using were a doubly linked list (DLL), singly linked list (SLL), a singly linked list as a deque (double-ended queue) and a dynamic array.

A binary search tree (BST) is unsuitable for storing sorted bricks in this algorithm because it does not efficiently support the specific operations needed to traverse the sequence of bricks in both eastward and westward directions. While a BST is optimised for fast searching based on key values, it organises elements hierarchically, not in a linear sequence, which makes it inefficient for the algorithm’s requirement of sequentially adding bricks to the result sequence. The algorithm requires inserting bricks based on their adjacency in both directions (east and west), but a BST does not inherently track relationships between consecutive bricks, making it difficult to efficiently find the next brick in the required direction. This would necessitate additional logic to maintain the proper east-west order, negating the advantages of the BST’s search efficiency. Instead, a linear data structure, such as a linked list or dynamic array, is better suited for this task, as it allows for efficient insertion at both ends and straightforward traversal in a sequential manner.

A hash table is also unsuitable for storing the sorted bricks in this algorithm because it does not maintain any order of the elements, which is essential for the east-west sequential traversal required. While a hash table offers efficient lookups and insertions, it stores elements based on hash values, not in a linear or sorted sequence. As a result, it cannot easily support the traversal of bricks in a specific directional order (east or west) since the adjacency relationships between bricks are not preserved. This would make it difficult to efficiently find the next brick in the sequence without additional complex operations. A linear data structure, such as a linked list or dynamic array, is better suited, as it maintains the required sequence for both eastward and westward traversal.

The amortised time complexities for the suitable data structures for the use cases identified are displayed in Table 2.

Table 2 - Time complexities for different containers to store the sorted bricks.

	DLL			SLL			Deque (SLL-based)			Dynamic array		
	Best	Ave.	Worst	Best	Ave.	Worst	Best	Ave.	Worst	Best	Ave.	Worst
Insertion (front)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(N)	O(N)	O(N)
Insertion (back)	O(1)	O(1)	O(1)	O(N)	O(N)	O(N)	O(1)	O(1)	O(1)	O(1)	O(1)	O(N)
Traversal	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)

For all containers, traversal would require O(N) in all scenarios. This is because traversal always requires visiting each node once, regardless of how many nodes are in the container or how the container is structured. Therefore, the insertion efficiency was the only factor to consider when choosing the container.

The DLL provides O(1) insertion complexity at both the front and back because it maintains pointers to both the head and tail nodes, allowing new nodes to be added directly at either end without the need to traverse the list. By adjusting just a few pointers - updating the head or tail and linking the new node - insertions at the ends can be performed in constant time.

A singly linked list (SLL) has O(1) insertion complexity at the front, as new nodes can be added directly by adjusting the head pointer to point to the new node, which then points to the original first node. However, insertion at the back is O(N) because, without a tail pointer, the list must be traversed from the head to find the last node before adding a new node at the end.

The SLL-based deque is an adaptation of the DLL as a queue container implemented in-class. It would maintain a head pointer and a tail pointer like the DLL as a queue, however I realised there was no need for each node to contain pointers to the previous and next nodes. This is because the elements only need to be traversed from front to back, and not back to front. Therefore, I adapted the original structure to be SLL-based rather than DLL-based, making the solution more memory efficient. The maintenance of the head and tail pointers meant that insertion at the front and back was very efficient, because regardless of the number of elements in the container, insertion had a fixed time operation. This is because insertion at the front involved pointing the new node at the current head, then setting the new node as the head, and insertion at the back involved pointing the current tail to the new node, then setting the new node as the tail. Both of these operations are fixed-time, pointer reassignments independent of the number of elements, giving a time complexity of O(1).

In a dynamic array, insertion at the back is typically O(1) in amortised time. This is because, under normal circumstances, adding a new element only requires placing it in the next available slot in the array, which is a constant-time operation. However, if the array's capacity has been reached, resizing is necessary, which involves allocating a new, larger array and copying over all existing elements. This resizing process takes O(N) time, but because it happens infrequently (usually only when capacity is doubled or increased by a fixed increment), the amortised cost of insertion remains O(1). Insertion at the front of a dynamic array, however, is O(N), as shifting all elements one position to the right is necessary to make space for the new element at the front, requiring a full traversal and move of all existing elements.

Based on the time complexities identified in Table 2, the best options were either the DLL or SLL-based deque. I decided to use an SLL-based deque. The choice of an SLL-based deque over a DLL was made primarily for its memory efficiency while still maintaining the required O(1) insertion time at both ends. A DLL requires each node to store two pointers: one to the previous node and one to the next, which increases memory usage, especially as the number of elements grows. In contrast, the SLL-based deque only stores a single pointer per node, pointing to the next node, reducing memory overhead. Since the algorithm only requires sequential traversal from front to back (eastward and westward directions), the extra pointer in the DLL to track the previous node is unnecessary. This makes the SLL-based deque a more suitable and optimised choice for this use case, offering both time and space efficiency.

4.2 Algorithm analysis

In this section, I will analyse and compare the performance consequences of using different combinations of container types for implementing the Royal Software Engineer's algorithm, which sorts bricks by traversing a sequence from east to west and vice versa. Specifically, I will evaluate the use of a hash table and a singly linked list (SLL)-based deque for the unsorted bricks and sorted bricks containers, respectively. I will also provide justifications for the chosen combination of containers based on their time complexity and suitability for the algorithm's requirements.

Table 3 - Overall best, average and worst case time complexity for the implemented Great Wall solution, using a hash table and SLL-based deque.

Step	Description	Best	Average	Worst
1	Inserting N bricks into the hash table	O(N)	O(N)	$O(N^2)$
2	Looking up N bricks from the hash table	O(N)	O(N)	$O(N^2)$
3	Inserting N bricks into SLL as deque	O(N)	O(N)	O(N)
4	Displaying all N bricks in SLL as deque	O(N)	O(N)	O(N)
	Overall	**O(N)**	**O(N)**	**$O(N^2)$**

Step 1 involves reading the data from the input file and inserting N bricks into the unsorted bricks hash table. The best and average case time complexity of reading and inserting each brick is O(1) (see Table 1). With N bricks, the overall best and average cases are both O(N). The worst case complexity for inserting one element into a hash table however is O(N), when all element hash to the same bucket due to hash collisions. This means that the overall worst case scenario of inserting N bricks would have a time complexity of $O(N^2)$.

Looking up an element in the hash table has a best and average case time complexity of O(1) (see Table 1), and with hash collisions a worst case complexity of O(N). When expanded to account for looking up all N elements, the overall best and average case complexity for a hash table lookup is O(N), and the overall worst case is $O(N^2)$.

Inserting a brick to the front or back of the SLL as a deque is O(1) in all cases (best, average and worst). This is because regardless of the number of elements in the list, the time taken to insert the new node and adjust the pointers is constant. Therefore, the overall time complexity for inserting N bricks into the SSL as a deque is O(N) across all cases.

Displaying all bricks in the SLL as a deque involves traversing the entire structure and visiting each element to display it regardless of structure, requiring O(N) time in all cases (best, average and worst).

Therefore, the overall time complexity for the algorithm I have implemented for the Great Wall problem is as follows:

Best case: **O(N)** = O(N) + O(N) + O(N) + O(N) = O(4N) = O(N) as constant terms are ignored in asymptotic notation
Average case: **O(N)** = O(N) + O(N) + O(N) + O(N) = O(4N) = O(N)
Worst case: **$O(N^2)$** = $O(N^2)$ + $O(N^2)$ + O(N) + O(N) = $O(2N^2 + 2N)$ = $O(N^2)$ because N^2 is the dominant term

Table 4 - Comparison of the overall best, average and worst case time complexities for different containers that could have been used in the Great Wall solution.

Unsorted →	**Hash table**			**AVL**		
Sorted ↓	**Best**	**Average**	**Worst**	**Best**	**Average**	**Worst**
DLL	O(N)	O(N)	$O(N^2)$	O(NlogN)	O(NlogN)	O(NlogN)
SLL	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
SLL-based deque	O(N)	O(N)	$O(N^2)$	O(NlogN)	O(NlogN)	O(NlogN)
Dynamic array	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$

Table 4 shows the comparison of the different containers that could have been used for the Great Wall problem. It shows the overall best, average and worst case time complexities for the suitable containers. This reinforces my choice of the hash table and SLL-based queue, as in the best and amortised cases it performs better. Although an AVL tree would lead to a better worst-case scenario, with an efficient hashing algorithm and threshold value, we can be confident the worst case scenario for the hash table is very unlikely.

Table 4 also highlights another suitable combination of choices - a DLL and hash table, however as discussed previously, the SLL-based deque offers a more memory efficient solution.

Overall, this combination of containers ensures that the algorithm is both time-efficient (with O(N) amortised time complexity) and memory-efficient, making it well-suited for handling large datasets of bricks efficiently.

5 Evaluation of performance

Table 5 - The performance data recorded from my implementation of the Great Wall solution for each input size, N.

Input size, N	Time taken (nanoseconds)	Time taken (seconds)	Time taken per brick (nanoseconds)	Memory usage (kilobytes)
20	211,527	0.000211527	10,576	40,960
50	708,479	0.000708479	14,170	16,384
100	1,329,579	0.001329579	13,296	20,480
200	1,123,153	0.001123153	5,616	40,960
500	6,174,189	0.006174189	12,348	69,632
1,000	7,836,866	0.007836866	7,837	184,320
2,000	45,029,686	0.045029686	22,515	335,872
5,000	176,114,984	0.176114984	35,223	880,640
10,000	90,413,416	0.090413416	9,041	1,654,784
20,000	421,218,587	0.421218587	21,061	3,063,808
50,000	1,268,486,641	1.268486641	25,370	8,126,464
100,000	1,395,163,718	1.395163718	13,952	15,630,336
200,000	4,902,335,331	4.902335331	24,512	28,975,104
500,000	6,619,718,623	6.619718623	13,239	88,944,640
1,000,000	10,437,071,671	10.437071671	10,437	160,735,232
2,000,000	19,176,915,715	19.176915715	9,588	279,740,416
3,000,000	37,001,442,907	37.001442907	12,334	316,641,280

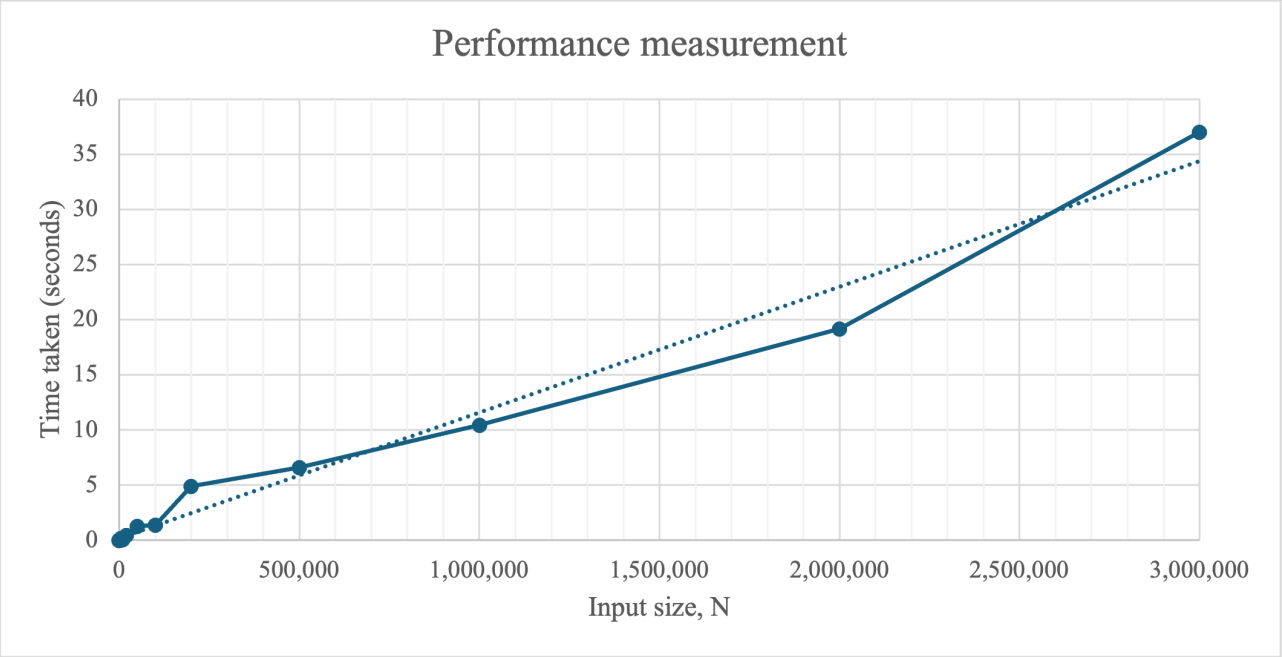

Figure 11 - A graph showing the performance of my Great Wall solution. The line of best fit is shown by the dotted line.

Figure 11 shows a linear relationship between the input size N and the time taken in seconds based on the data recorded in Table 5. This reinforces the performance analysis completed previously, and proves that the prediction of a time complexity of O(N) in Table 4 for the algorithm is correct. Figure 12 shows this data with logarithmic scales, allowing the recordings of the smaller values of N to be compared in more detail. This further supports the algorithm's time complexity of O(N), because the linear relationship between N and the time taken is shown clearly in the graph, further highlighted by the dotted line of best fit.

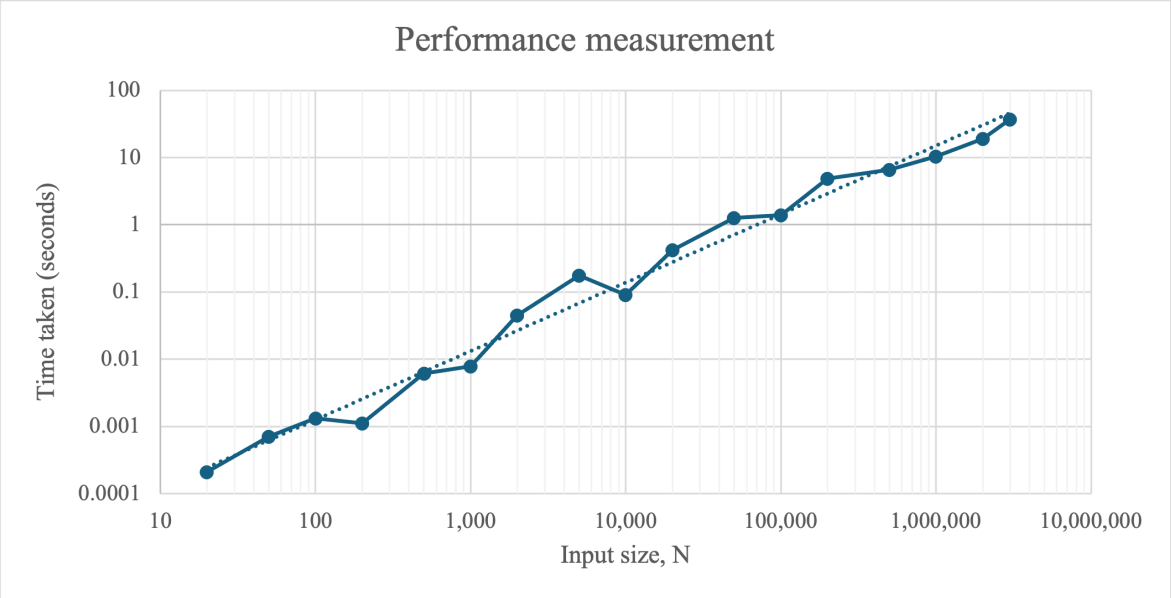

Figure 12 - A graph showing the performance of my Great Wall solution with logarithmic scales. The line of best fit is shown by the dotted line.

Analysing the runtime data reveals an unexpected deviation where the runtime for N=200 is faster than for N=100. This counterintuitive result may stem from system-level optimisations such as caching effects or hardware-specific behaviours. For instance, modern CPUs often use cache lines to store data temporarily for faster access, and the dataset for N=200 may fit more efficiently into the CPU cache compared to N=100, reducing memory access times. Additionally, measurement inconsistencies, such as background processes or slight variations in CPU load during testing, can contribute to such anomalies, especially when processing smaller datasets where runtime differences are more pronounced. While these deviations do not align

perfectly with theoretical predictions, they are expected in real-world performance measurements and do not undermine the overall trend of linear growth in runtime as N increases.

Furthermore, the time taken for the algorithm to process each brick is shown in Figure 13. This graph highlights significant variation for smaller values of N, further visible in the graph in Figure 14 using a logarithmic scale for N. This variability is attributed to fixed overheads, such as input/output operations and memory allocation, which contribute disproportionately to the total runtime at smaller scales. Additionally, measurement noise from hardware processes can amplify these variations.

However, as N increases, the time per brick stabilises to a consistent value. This behaviour reflects the dominance of the core algorithmic operations over fixed overheads at larger scales, confirming the linear O(N) complexity of the implementation. The stabilisation demonstrates that the total runtime grows proportionally with the input size, validating the efficiency of the algorithm.

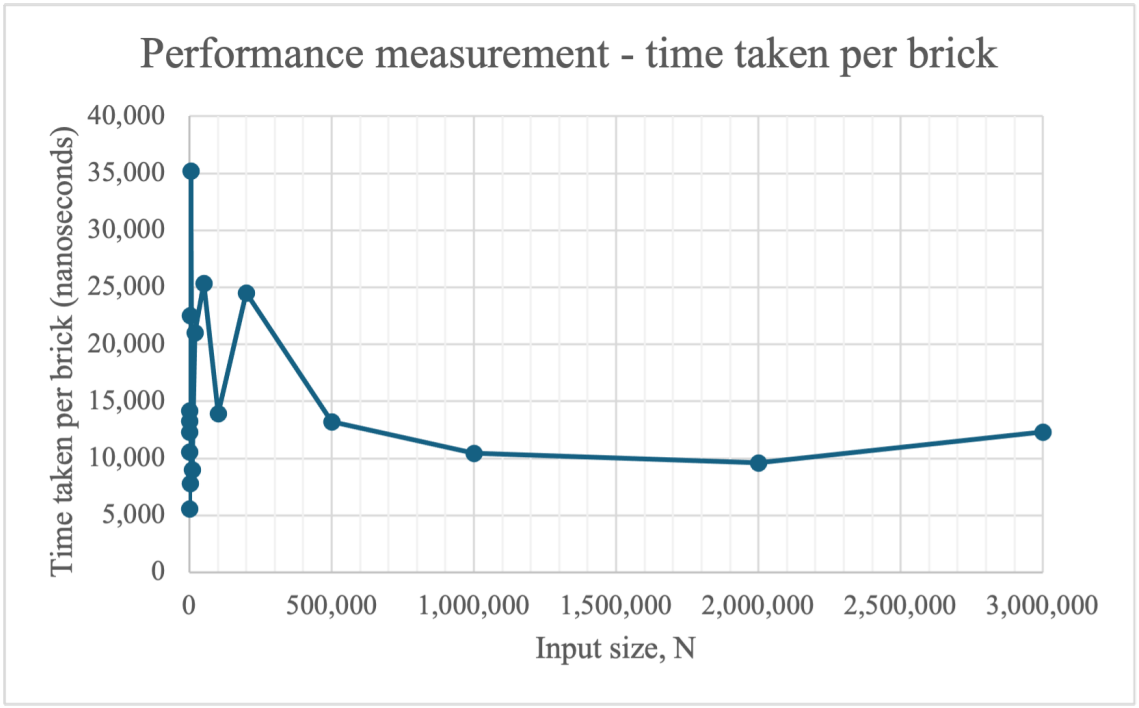

Figure 13 - A graph showing the time taken to process each brick depending on the input size, N.

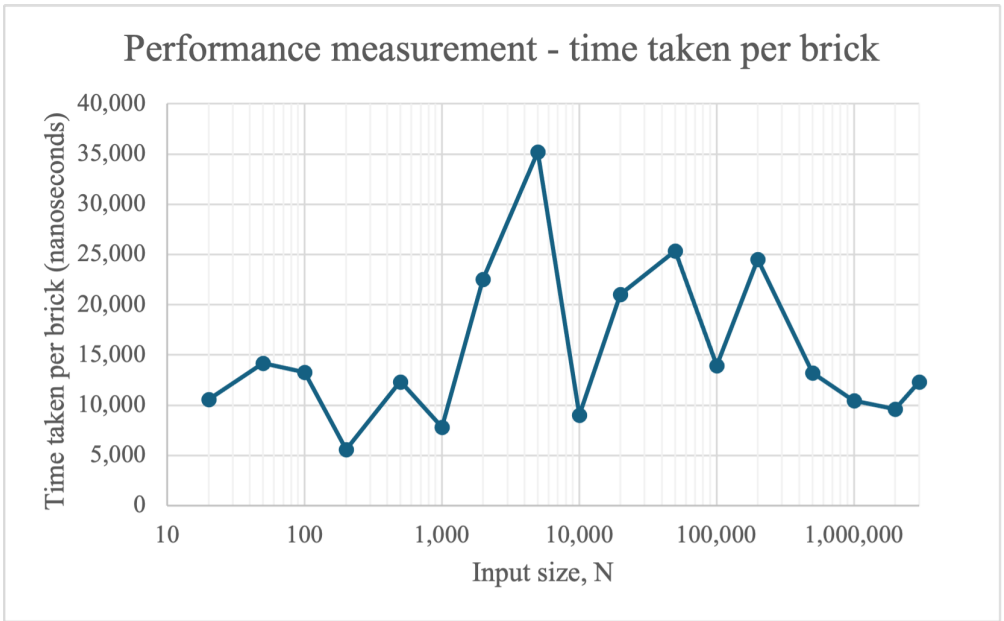

Figure 14 - A graph showing the time taken to process each brick depending on the input size, N, shown using a logarithmic scale.

Figure 15 demonstrates a clear linear trend between memory usage and input size. This indicates that the memory required by the algorithm grows proportionally with the input size N, consistent with a space complexity of O(N). This behaviour is expected, as the algorithm primarily relies on two containers: a hash table for unsorted bricks and a singly linked list-based deque for sorted bricks, both of which require memory allocation directly proportional to the number of elements stored. Each brick contributes a fixed amount of memory to the containers, leading to a linear increase in memory usage as N grows.

Notably, the first result when N=20 exhibits a significant increase in memory usage compared to expectations. This anomaly appears consistently across runs and may be attributed to caching mechanisms or initialisation overheads within the runtime environment. Such behaviour is often observed when memory allocation for data structures triggers additional system-level operations, such as reserving buffer space, aligning memory regions, or caching for optimised future performance. After this initial irregularity, the absence of significant deviations or spikes in the graph further supports the efficiency of the chosen data structures in managing memory allocation without overheads or fragmentation. This linear space complexity ensures that the algorithm remains scalable and suitable for handling large datasets.

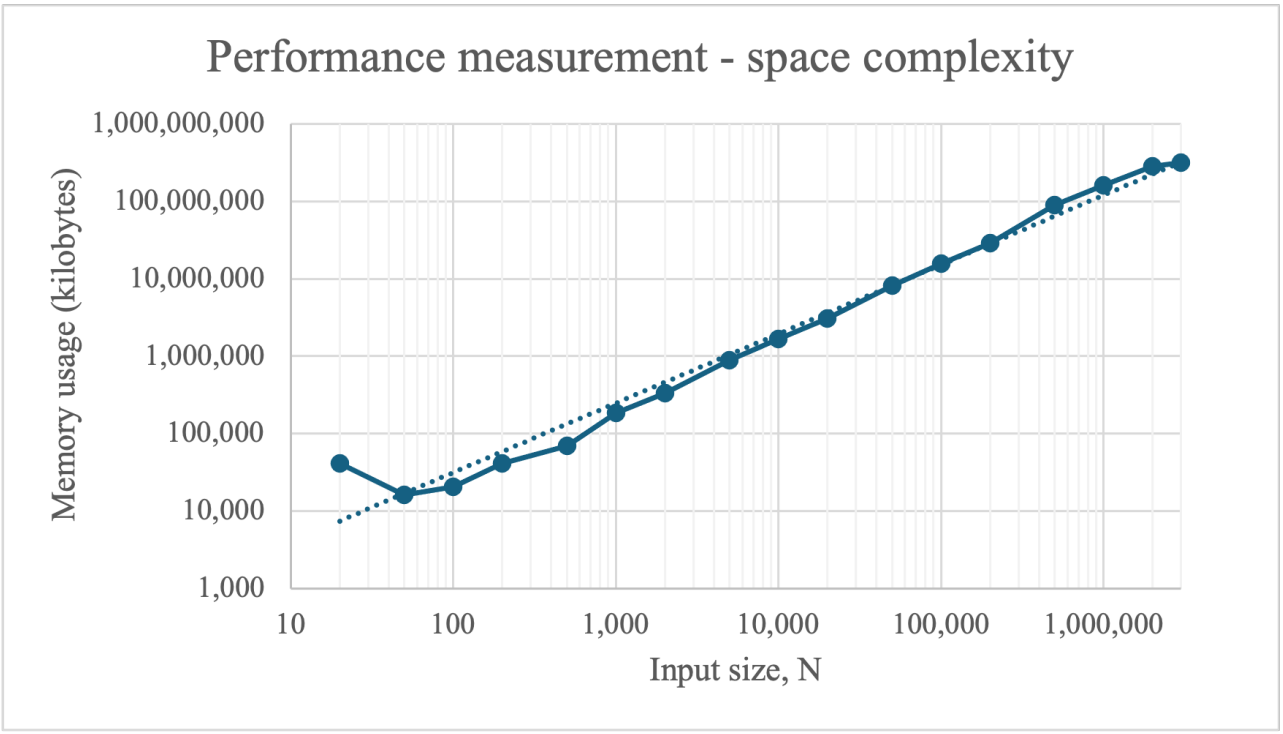

Figure 15 - A graph showing the memory usage of the algorithm for different values of N on logarithmic scales.

6 Conclusion

This project highlights the effectiveness of TDD and custom data structures in addressing the Great Wall Problem. The chosen combination of a hash table and an SLL-based deque proved to be both time- and memory-efficient, aligning with the problem's constraints. Empirical performance results confirmed the algorithm's linear time complexity, validating the theoretical analysis. Reflecting on this experience, the disciplined approach enforced by TDD significantly enhanced code robustness, though it introduced challenges during major functional changes. Overall, the algorithm demonstrates scalability and efficiency, offering a strong solution to the Great Wall Problem.

7 References

Amálio, N.R., 2024. *SOFT37002 Module Git Repository*. Nottingham Trent University. Available at: https://olympus.ntu.ac.uk/CMP3RODRIN/ASE-DA [Accessed 17 November 2024].