Advanced Software Engineering

Assignment 1: Software Development Investigation

Barry O'Connor – N0813926

Table of Contents

Task 1: Complexity Analysis — Binary Search Trees 3
BST() – Constructor 4
~BST() – Destructor 4
BST(const BST &) – Copy Constructor 4
BST & operator=(const BST &) – Copy Assignment Operator 4
BST(BST &&) – Move Constructor 5
BST & operator=(BST &&) – Move Assignment Operator 5
ItemType* lookup(KeyType) – Search for a Node. 5
void insert(KeyType, ItemType) – Insert a Node 5
void remove(KeyType) – Remove a Node 6
void removeIf(std::function<bool(KeyType)>); 6
void displayEntries() – Display Tree Nodes 6
void displayTree() – Display Formatted Tree 7
Task 2: Justifying Implementation Choices 8
Task 2a: Understanding Performance Guarantees – Standard Library Components 8
Std::list 8
Std::map 9
Std::unordered_map 10
Task 2b: Analysing the Royal Software Engineer's Algorithm 12
Preferred implementation – List and unordered_map 12
Second Best implementation – List and map 12
Task 3: Performance Measurement - The Royal Software Engineer's Algorithm 14
Task 3b: Measuring the Royal Software Engineer's Algorithm 14

Figure 1- Graph showing results of the algorithm for both combinations of containers on a linear scale ..... **Error! Bookmark not defined.**
Figure 2 - Graph showing results of the algorithm per brick for both combinations of containers on a logarithmic scale ..... **Error! Bookmark not defined.**

Task 1: Complexity Analysis — Binary Search Trees

BST() – Constructor

Average Complexity: O(1) Best Case: O(1) Worst Case: O(1)

This function creates the BST Class and initialises the contents of the class and will run in constant time. This is simply because the initialisation of the class, including the call to the leaf() function, consists entirely of primitive operations which have a complexity of O(1), resulting in an average complexity of O(1). Both the best-case and worst-case for this function will be O(1) for the same reason.

~BST() – Destructor

Average Complexity: O(n) Best Case: O(n) Worst Case: O(n)

The destructor for the class calls deepDelete() which recursively deletes the left and right branches of the tree, visiting every node. The content of the function, aside from the recursive function calls, is simply a single delete statement and will result in a complexity of O(1). Since the recursive function must traverse the tree fully, the number of nodes 'n' will affect the complexity and result in an average complexity of O(n). The best and worst-case scenario would also be O(n) because no matter the size of the tree or how balanced it is, every node must always be visited.

BST(const BST &) – Copy Constructor

Average Complexity: O(n) Best Case: O(n) Worst Case: O(n)

The copy constructor calls deepCopy() which recursively traverses the tree and copies each node. In copying the node, the node constructor function is also called. Aside from the recursion, both functions consist of constant time statements, resulting in a complexity of O(1). Since the function must copy each node, the complexity overall will always be O(n) where n is the number of nodes. Again, the best and worst cases would also be O(n) because each node must be visited.

BST & operator=(const BST &) – Copy Assignment Operator

Average Complexity: O(n) Best Case: O(1) Worst Case: O(n)

This function assigns a copy of the tree and, depending on the state of the variable it is assigned to, calls deepCopy() and deepDelete() which both recursively traverse the tree and copies or deletes each node. In copying the node, the node constructor function is also called. The code within this function is constant time, as is the leaf() function, leaving the deepCopy() and DeepDelete() functions which both must visit each node giving them a complexity of O(n) and therefore an overall average complexity of O(n) for this function. However, this function does have very different worst and best-case scenarios. The best case is one where the variable the tree is being copied and assigned to, already contains the tree in which case the recursive functions are skipped resulting in a best-case complexity of O(1). In the worst case, all nodes must be visited by both functions resulting in a worst-case complexity of O(n) for each function and therefore an overall best case complexity of O(n) for the entire function.

BST(BST &&) – Move Constructor

Average Complexity: O(1) Best Case: O(1) Worst Case: O(1)

This function simply moves the tree by detaching the current root pointer and attaching the newly defined variable to the tree. This function operates fully in constant time and has a best, worst and average complexity of O(1) because of this.

BST & operator=(BST &&) – Move Assignment Operator

Average Complexity: O(n) Best Case: O(1) Worst Case: O(n)

As with the move constructor, the move assignment operator function reassigns a pointer from the root of an existing BST to a new variable. However, this function must delete any existing nodes in the destination (if any are present) and so the deepDelete() function is present. As discussed previously the deepDelete() function has an average complexity of O(n), meaning that this function also has an average complexity of O(n). Since a check is made for the destination and source matching, which skips the need to delete, the best case complexity equates to O(1) and the worst-case remains as O(n).

ItemType* lookup(KeyType) – Search for a Node.

Average Complexity: O(log(n)) Best Case: O(1) Worst Case: O(n)

This function calls the recursive function lookupRec() which has an average complexity of O(log(n)). In searching for a node within a tree the complexity depends on the height h of the tree for a perfectly balanced tree. This is due to the search having to traverse from the root of the tree to the leaf furthest from the root. O(h) equates to $n = 2^h - 1$ which simplifies to O(log(n)) as an average for a perfectly balanced tree. Since this tree has no self-balancing mechanism, it could be possible to have a tree consisting only of left or right children. Such a tree would function like a List data structure and, due to having to access all nodes sequentially would have a complexity of O(n). The best-case scenario in searching a tree will be that the searched-for node is the root and this will result in a complexity of O(1).

void insert(KeyType, ItemType) – Insert a Node

Average Complexity: O(log(n)) Best Case: O(1) Worst Case: O(n)

This function calls the recursive function inserted() and follows a similar pattern as the lookup() function. This function also has an average complexity of O(log(n)) due to also being dependant on the height of the tree and insertions having to traverse from the root to the furthest leaf to find the correct insertion point. Again, because this is not a self-balancing tree, a worst-case scenario consisting of a sequence of left only or right only nodes would present a complexity of O(n) while a best-case complexity of O(1) would occur if the tree was empty and insertion was at the root node.

void remove(KeyType) – Remove a Node

Average Complexity: O(log(n)) Best Case: O(1) Worst Case: O(n)

This function follows the pattern for the previous functions and is again dependant on height, resulting in an average complexity of O(log(n)). Worst and best cases also follow suit with the best case remaining O(1) for deletion of the root node and a tree consisting of only left or right children will result in a worst-case of O(n).

While this function calls on the detatchMinimumNode() function, this second function also follows the functionality of the lookup() function in the main part, traversing until the lowest value node is found and detaching and returning it. The pattern of complexity for this recursive function also has an average of O(log(n)) and an identical pattern of best-case where the root of the passed sub-tree is the lowest resulting in best-case of O(1) and again, in a worst-case of a tree consisting of only left or right nodes causing a worst-case complexity of O(n).

Since both results have the same complexities, the overall complexity does not change.

void removeIf(std::function<bool(KeyType)>);

Average Complexity: O(n) Best Case: O(n) Worst Case: O(n)

This function uses a comparison function as a parameter and removes elements based upon that parameter. The function definition was not recursive so an iterative approach was taken, requiring the use of a stack to store processed nodes. The core of the function consists of two while loops, non-nested, one to iterate through every node in the tree to perform the comparison. Once this is complete, and to avoid altering the tree during the comparison, a second while loop calls the remove() function to remove the matched elements after the first while completes.

The first while will always have an average, best-case and worst-case complexity of O(n) because it must traverse the entire tree to perform the comparison at each node.

The second while loop calls remove() which has an average complexity of O(log(n)). This loop depends on the number of matches, m, which are then removed from the tree. This value can vary from 0 matches through to N matches. Overall this equates to O(m * log(n)).

For the entire function, the dominant value for each part if O(n) resulting in an average complexity of O(n). Since the comparison visits every node, the best and worst complexity will always be O(n) as well.

void displayEntries() – Display Tree Nodes

Average Complexity: O(n) Best Case: O(n) Worst Case: O(n)

In a similar vein to the copy constructor, this function must recursively traverse and display every node within the tree and the complexity is dependant on n, the number of nodes, resulting in an average complexity of log(n). Again, the best and worst-case complexities will also be O(n) since every node must be traversed in a tree of any size or shape.

void displayTree() – Display Formatted Tree

Average Complexity: O(n) Best Case: O(n) Worst Case: O(n)

This function also uses recursion and must also traverse and display every node within the tree. The complexity is, therefore, dependant on n, the number of nodes, resulting in an average complexity of O(n). Again, the best-case and worst-case complexities will also be O(n) since every node must be traversed in a tree of any size or shape.

Task 2: Justifying Implementation Choices

Task 2a: Understanding Performance Guarantees – Standard Library Components

Std::list

In C++ a std::list is based upon a doubly-linked list which is a data structure where each node has two pointers, a pointer to the next and a pointer to the previous node, and so allows for traversal in both the forward and backward directions.

In terms of searching or accessing elements, this structure uses internal class properties which contain pointers to the front and back, meaning these nodes can always be found in constant time. Accessing these elements will always be via the pointer, so the best-case, worst-case and average complexity will be O(1). Accessing other elements by searching within the structure would require iterating through the nodes from either the front or the back node until the required node is found. This is not provided by the structure itself and would need to be implemented by the programmer. This would have a best case of O(1) where the desired node was the first node in the list or the desired node was referenced by a pointer externally, O(n) where the desired node was the last in the list and did not have an external pointer to identify the node, resulting in an average complexity of O(n).

To insert a node from the front or back, push_front() and push_back() can be used. Since all of these operations occur at the front and back of the structure, using the front and back pointers, this is a very quick operation. These pointers simply need to be adjusted to point to the new node for insertion and detach the pointer for deletion. This gives a best-case, worst-case and average complexity of O(1). If inserting more than one element, such as using the insert function, the complexity would depend on the number of elements inserted and would have an average complexity of O(n).

When dealing with insertion or deletion in any other part of the list, the operation to insert or delete remains O(n), however the complexity increases due to having to search for the node or its future position. As discussed previously, this type of search would have a best case of O(1) where the desired node was the first node in the list, O(n) where the desired node was the last in the list, resulting in an average complexity of O(n).

The following functions will be of use in the problem:

Constructor
- since we are creating an empty list, the best-case, worst-case and average complexity will simply be O(1), non-amortised.

Destructor
- the destructor for each element is called and storage deallocated giving this function a complexity of O(n), non-amortised.

front and back
- The complexity for this function is always O(1) , non-amortised, since it simply returns a reference to the element at the front or back of the list.

Push_front() and push_back()
- The complexity for this function is always O(1) , non-amortised, since it simply need to change where the relevant pointer is referencing.

Std::map

In C++ the underlying structure of a map is implemented using a red-black binary tree. This type of tree contains additional information internally to designate each node as black or red and uses this information to make the structure self-balancing. It should be pointed out that the balance is not perfect and a skewed tree can occur but the balancing mechanisms greatly reduce the degree to which the tree can skew and offer better worst-case time complexity than a standard BST. The structure follows four rules to maintain this level of balance:

1) Every node in the tree is designated either red or black.
2) The root node is always black.
3) A red node cannot have a parent or child that is also red.
4) Every path from any node to a leaf has the same number of black nodes (the black height of the tree).

When a node is inserted, it automatically becomes a red node and a process of recolouring and rotation occurs. The tree is checked to see if the above rules still apply and if not, recolouring and rotation occurs so that the new tree remains valid. Recoloring checks the colours of the parent and parent-sibling nodes and uses specific combinations of these to swap the colours of parent and parent-sibling nodes from red to black and vice versa. Rotation is where the tree is rotated to become more balanced and a process of recolouring can occur after this to keep the tree valid.

These actions guarantee a worst-case complexity of O(log(n)) for searching, insertion and deletion by ensuring that the tree can never get to the point where it is comprised only of left or right nodes.

The following functions will be of use in the problem:

Constructor
- since we are creating an empty map, the best-case, worst-case and average complexity will simply be O(1), non-amortised.

Destructor
- the destructor for each element is called and storage deallocated giving this function a complexity of O(n), non-amortised.

find()
- best-case O(1), non-amortised, where the root node is the item searched for.
- worst-case and average case O(log(n)), non-amortised, where the element is a leaf.

insert()
- best-case O(1), non-amortised, for an insertion into an empty tree where the element becomes the root node
- worst-case and average case O(log(n)), non-amortised, where the element is a leaf.

Std::unordered_map

In C++ the underlying structure of an unordered_map is implemented using a hash table which is essentially a set of buckets containing associative arrays. The hash table uses a hashing algorithm to create automatically generated hash values based upon the key of the key-value pair being stored. This hash will be unique and will therefore allow immediate access to the bucket associated with the data data in the same way that a standard array can access element[10] very quickly.

However, it should be pointed out that the efficiency of the hash table is dependant on the hashing algorithm. A less efficient algorithm might cause more collisions (duplicate hashes) to be created, slowing down the insertion process. With the worst hashing algorithm, where each insertion would cause a collision, the insertion time would be O(n). Bad hashing can also allow too many elements to be stored in the same bucket, forcing any search to iterate the bucket contents, resulting in a worst-case search compesity of O(n) where n is the number of elements in the bucket, not the total number of elements in the structure.

As with a standard array, a hash table's underlying structure is fixed and cannot be dynamically increased without additional overhead. While the structure has empty slots, insertion is a simple process of creating a pointer to the new data and storing it in the array element assigned to the hash value. This completes in O(1) time because the steps are so simple. When the structure is at capacity and no longer has any empty slots, it must be resized.

Resizing the structure involved creating a new structure, twice the size of the current one, in memory and copying the contents of the original across to the new structure. This will only happen when the structure is full, so does not happen on each insert and will happen less frequently as the structure grows. Since this involves copying all current elements, this has an associated cost of O(n) For example, a structure of size 2 will become a structure of size 4 (with 2 free slots) so only 2 insertions are possible before another resize is required while a structure of 1000 will resize to 2000, meaning 1000 inserts can be completed before the next resize is needed.

Table 1 - Table showing the cost of insertion into a hash table per element

Number of Elements	1	2	3	4	5	6	7	8	9
Size of Structure	1	2	4	4	8	8	8	8	16
Cost of insertion	1	1	1	1	1	1	1	1	1
Cost of resize	0	0	2	0	4	0	0	0	8
Total cost of insertion and resize	1	2	3	1	5	1	1	1	9

As demonstrated by the table above, this resizing only happens when N is an exact power of 2 and it is also possible to see that the total cost of an insertion can be averaged (amortised) as O(1). This is

because the O(n) operation can happen at most n/2 times and therefore (n(O(1) + O(n)) / n) which simplifies to O(n)/n or O(1) as an average cost.

The following functions will be of use in the problem:

Constructor
- since we are creating an empty map, the best-case, worst-case and average complexity will simply be O(1), non-amortised.

Destructor
- the destructor for each element is called and storage deallocated giving this function a complexity of O(n), non-amortised.

find()
- best-case O(1), non-amortised, where the root node is the item searched for.
- worst-case and average case O(n), non-amortised, where the structure must check each element in a bucket.

insert()
- best-case and average complexity of O(1), amortised
- worst-case O(n), non-amortised, where all elements in a bucket must be searched.

Task 2b: Analysing the Royal Software Engineer's Algorithm

Preferred implementation – List and unordered_map

As discussed previously, an unordered_map has an amortised insertion time of O(1) and a best and average lookup time of O(1) which is likely to be faster overall than a map, which has a complexity of O(log(n)) for these operations. In both cases, the list is simply the superior choice for the final wall due to the ability to traverse in both directions and since the problem only requires insertion and access at the back and front of the structure, a list will provide O(1) complexity for each operation.

Following the numbered steps of the algorithm, we can see the complexity of theses as being:

1) Create an unordered_map – complexity O(1).
2) Loop the input data and insert. N will be the number of elements and with an amortised complexity per insert of O(1), we get O(n * 1) or O(n)
3) Create a list – complexity O(1).
4) Insert 2 elements into the front of the list – both are O(1) resulting in O(1) overall
5) Searching the unordered_map will result in an average complexity of O(1) (worst case O(n) and inserting the result into the list will also be O(1)
6) This will involve searching in one direction for m values where m can be between 1 and n. Each search has an average complexity of O(1) (worst case O(n) resulting in an average complexity of O(m))(worst case O(nm))
7) Simply access the back() element in the list – O(1)
8) This will involve searching in the opposite direction for the remaining values. Since we have now searched m values, the value of elements this time will be equal to n-m. n-m added to the m we have from stage 6 is n-m +m which simply resolves to O(n) (worst case n O(n)) as an average complexity for stages 6 and 8.

Overall the algorithm should have the average complexity:

O(1 + n + 1 + 2 + 2 + 1 + n) this simplifies to O(7 + 2n) and since we can remove constants, this should result in an average value of O(n) for the entire algorithm using a list and unordered_map.

Second Best implementation – List and map

As discussed previously, a map has an insertion time of O(lon(n)) and a best and average lookup time of O(log(n)) which is likely to be slower overall than an unordered_map, making the map a second best choice for this problem. As before, the list offers the best fit for the wall due to the ability to traverse in both directions and since the problem only requires insertion and access at the back and front of the structure, a list will provide O(1) complexity for each operation.

Following the numbered steps of the algorithm, we can see the complexity of theses as being:

1) Create a map – complexity O(1).
2) Loop the input data and insert. N will be the number of elements and with a complexity per insert of O(log(n)) for the map, we get O(n * log(n)) or O(n(log(n)))
3) Create a list – complexity O(1).
4) Insert 2 elements into the front of the list – both are O(1) resulting in O(1) overall

5) Searching the map will result in an average complexity of O(log(n)) and inserting the result into the list will also be O(1)
6) This will involve searching in one direction for m values where m can be between 1 and n. Each search has an average complexity of O(log(n)) resulting in an average complexity of O(m(log(n))
7) Simply access the back() element in the list – O(1)
8) This will involve searching in the opposite direction for the remaining values. Since we have now searched m values, the value of elements this time will be equal to n-m. n-m added to the m we have from stage 6 is n-m +m which simply resolves to n, giving O(n(log(n)) as an average complexity for stages 6 and 8.

Overall the algorithm should have the average complexity:

O(1 + n(log(n)) + 1 + 2 + 2 + 1 + n(log(n))) this simplifies to O(7 + 2*n(log(n))) and since we can remove constants, this should result in an average value of O(n(log(n))) for the entire algorithm using a list and map.

Task 3: Performance Measurement - The Royal Software Engineer's Algorithm

Task 3b: Measuring the Royal Software Engineer's Algorithm

Code to time the algorithm was added and several results were recorded. The values in the following table represent the mean time per brick taken for each time the program was ran. To try and eliminate some of the variation, these results were again averaged to come up with a final average value that could be plotted on the graph.

It was interesting to see that there was an immediate difference in the range of the values being returned. The average value for the map varied relatively little, in the region of plus or minus a few hundred nanoseconds. This gave an overall more predictable, steady set of values. The unordered_map on the other hand could vary by several microseconds between readings. This could be due to the possibility of the O(n) worst case search occasionally happening or perhaps something in the resize took longer than expected.

Table 2- results gained testing the algorithm using a list and unordered_map

Execution time for the algorithm using a List and Unordered Map						
Number of Bricks	Result 1 (ms)	Result 2 (ms)	Result 3 (ms)	Result 4 (ms)	Result 5 (ms)	Average of Results (ms)
1000	17.5	17.508	17.711	17.361	17.435	17.503
2000	34.984	35.024	35.124	34.886	35.064	35.0164
5000	90.97	91.015	91.805	91.145	91.545	91.296
10,000	190.42	191.09	191.44	192.01	190.59	191.11
20,000	384.48	383.08	385.36	383.1	383.28	383.86
50,000	950.7	968.15	960.9	967.05	963.7	962.1
100,000	1956.8	1967.5	1969.5	1965.9	1967.7	1965.48
200,000	3956.8	3943	3953.6	3919.8	3924.8	3939.6
500,000	9723.5	9836	9850	9719.5	9748.5	9775.5
1,000,000	20367	20258	20198	20346	20240	20281.8

Table 3 - results gained testing the algorithm using a list and map

Execution time for the algorithm using a List and map						
Number of Bricks	Result 1 (ms)	Result 2 (ms)	Result 3 (ms)	Result 4 (ms)	Result 5 (ms)	Average of Results (ms)
1000	25.007	24.671	24.779	25.143	24.932	24.9064
2000	51.11	51.328	51.32	50.922	51.538	51.2436
5000	133.13	133.82	133.325	134.675	134.975	133.985
10,000	281.75	286.22	287.29	284.23	282.4	284.378
20,000	599.06	601.22	602.54	606.44	602.08	602.268
50,000	1577.25	1604.1	1574.9	1585.05	1578.3	1583.92
100,000	3333.1	3305.1	3264.8	3342.1	3294.3	3307.88
200,000	6858.8	6870.6	6764.6	6751.2	6804.6	6809.96
500,000	17838	17872.5	17845	17672.5	17853	17816.2
1,000,000	37457	38244	38070	37882	37880	37906.6

Plotting the above data onto a graph resulted in the following graphs. These have been combined into one graph for comparison and an additional graph using logarithmic scale to show the rate of change and the varying scale more easily is included.

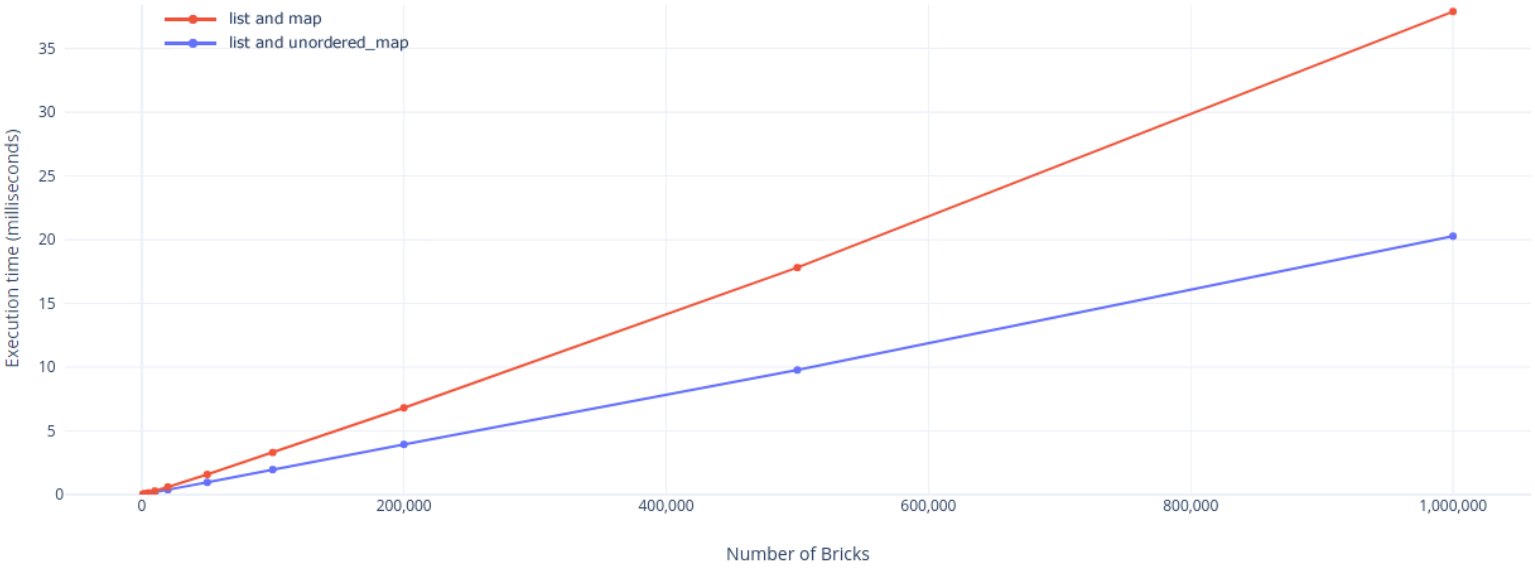

Figure 1- comparison of both implementations on a linear scale

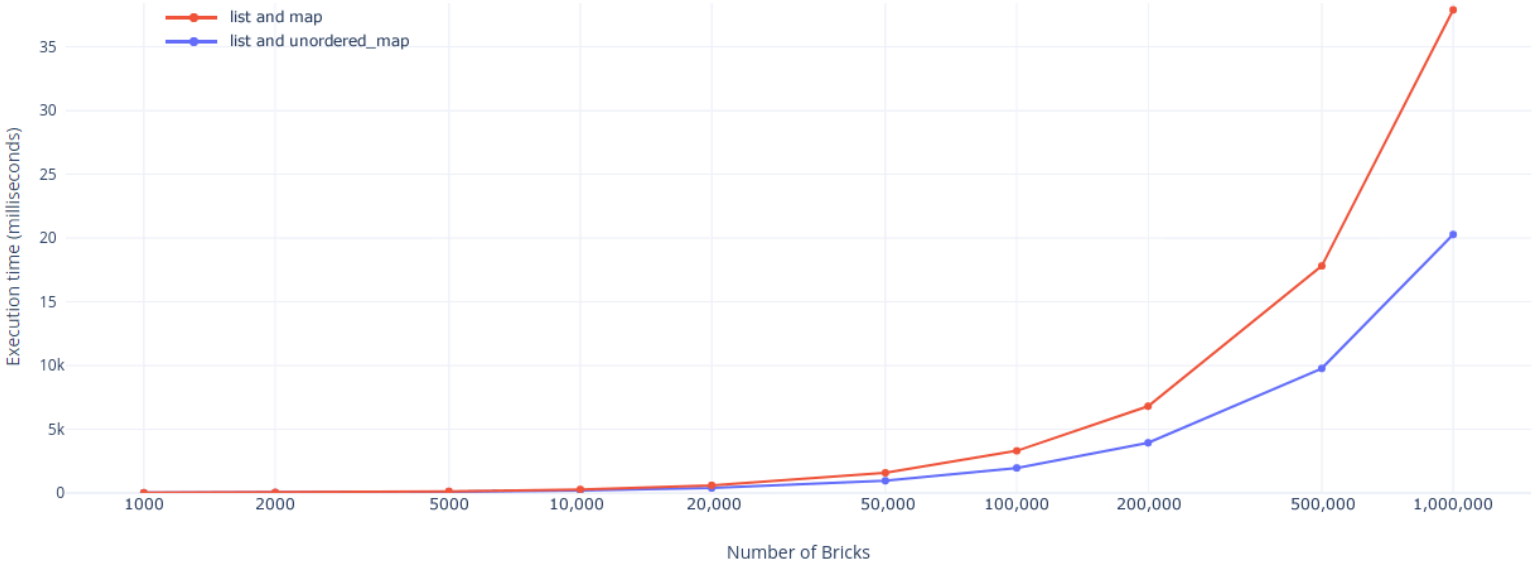

Figure 2- Graph showing the relationship between the implementations on a logarithmic scale

The graph in Figure 2 relationship between the two implementations with the implementation using the map taking increasingly more time. This graph shows well that there is a relationship for both with n, the number of bricks, which is as expected.

By changing the x axis to a logarithmic scale, we can compensate for the range of the given data which has lots of readings in the 0-10000 scale and then several readings for relatively large

numbers of bricks. By using this scale we can see the rate of change well and can see that for the blue line (unordered_map), if we take the value at 10,000 and the value at 20,000 the execution time roughly doubles. The same is true for 100,000 and 200,000. This is a strong indicator that the line is increasing linearly with n which backs up the prediction that an implementation with an unordered_map would have an average complexity of O(n).

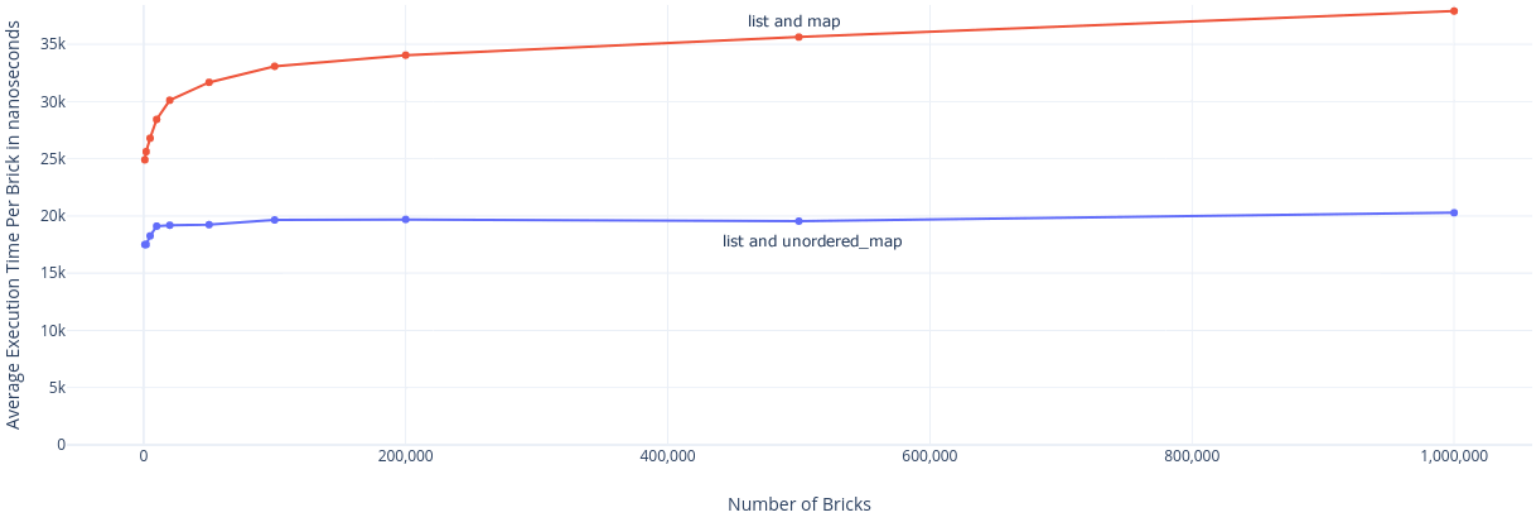

Figure 3 - graph of average execution time per brick

The red line in Figure 2 also has an obvious relationship with n, however the increase at each step is greater than the increase seen in the blue line. By combining this with the graph in Figure 3, we can also tell that the average time per brick forms a graph shoting a complexity of O(log n). In figure 3 we can see the value is increasing in a somewhat linear fashion similar (although greater) than the O(n) blue line. This suggests both are being affected by N multiplied by an underlying value. Since we know that the individual bricks are exevuted in O(log n), the overall complexity must be O(n log n). This lends weight to the prediction that this algorithm using a map would produce a complexity of O(n log n).

Task 4: The Royal Mathematician's Algorithm

Task 4a: Analysing the Royal Mathematician's Algorithm

put the pairs on a tape - O(n)
Copy the tape - O(n)
sort tape A - O(n log n)
sort tape B - O(n log n)
pass tapes in sequence - O(n)
copy pairs to new tape - O(n)
Add 2 pairs to final tape - O(1)

Recursive: n * (1 + ½ + ¼ + 1/16 …). Sum of brackets is always less than 2. N*2 = O(n)

copy tape a to c - O(n)

sort C by first - O(n log n)

sort A by second - O(n log n)

advance tapes:

each step advances one or more tapes - O(1)

some steps involve appending to a tape - O(1)

if A is exhausted

if true - worst case is O(n log n), if false no statements

sort B1 - worst-case O(n log n)

merge B1 and B O(N) worst case

replace t - O(1)

replace A by A1 - O(N)

replace B by B1 - O(N)

Sort Tape B - O(n log n)

For the code outside recursive function, the highest complexity os O(n log n). Inside the function the highest complexity is O(n log n) which is multiplied by the factor of the recursion, giving an overall average complexity of $O(n^2 \log n)$.

Task 4a: Analysing the Royal Mathematician's Algorithm

Execution time for the algorithm using lists as tapes						
Number of Bricks	Result 1 (ms)	Result 2 (ms)	Result 3 (ms)	Result 4 (ms)	Result 5 (ms)	Average of Results (ms)
1000						
2000						
5000						
10,000						
20,000						
50,000						
100,000						
200,000						
500,000						
1,000,000