Tugra Karakus N0912548

7th December 2022

SOFT30161: Advanced Software Engineering 202223 Full Year

Software Development Investigation

Contents

[Task 1: Analysis of your Binary Search Tree Implementation of a Dictionary 1](#_Toc122088244)

[lookup() 1](#_Toc122088245)

[insert() 1](#_Toc122088246)

[displayEntries() 1](#_Toc122088247)

[The destructor 1](#_Toc122088248)

[remove() 1](#_Toc122088249)

[displayTree() 1](#_Toc122088250)

[rotateLeft() and rotateRight() 1](#_Toc122088251)

[The copy constructor 1](#_Toc122088252)

[The move constructor 2](#_Toc122088253)

[The copy assignment operator 2](#_Toc122088254)

[The move assignment operator 2](#_Toc122088255)

[removeIf() 2](#_Toc122088256)

[Task 3: Justifying Implementation Choices 2](#_Toc122088257)

[std::list 2](#_Toc122088258)

[std::map 3](#_Toc122088259)

[std::unordered\_map 3](#_Toc122088260)

[Task 4: Understanding Data Structures 3](#_Toc122088261)

[Task 5b: Presenting the Royal Software Engineer's Results 5](#_Toc122088262)

[Task 5c: Evaluating the Royal Software Engineer's Algorithm 7](#_Toc122088263)

[Task 6d: Presenting the Royal Mathematician's Results 8](#_Toc122088264)

[Task 6e: Evaluating the Royal Mathematician's Algorithm 9](#_Toc122088265)

[Bibliography 10](#_Toc122088266)

[Figure 1 6](#_Toc122088267)

[Figure 2 6](#_Toc122088268)

[Figure 3 7](#_Toc122088269)

[Figure 4 8](#_Toc122088270)

[Figure 5 9](#_Toc122088271)

[Table 1 5](#_Toc122088272)

[Table 2 5](#_Toc122088273)

[Table 3 8](#_Toc122088274)

# Task 1: Analysis of your Binary Search Tree Implementation of a Dictionary

## lookup()

Performs a recursive lookup of a sought key. The best time complexity is O(1), when the root is the desired node. Average time complexity is O(log(n)), assuming the tree is balanced. The worst time complexity is O(n), when the tree acts as a linked list due to the tree being entirely unbalanced down one side, with the desired node at the end.

## insert()

Performs a recursive insert function. Has same time complexities as lookup(), due to it behaving the same. However, it inserts a new item instead of returning a lookup result. The best case time complexity is the same as lookup(), O(1); but this is the case when the tree is empty.

## displayEntries()

Performs a recursive display. The best time complexity is O(1), this is when the tree contains only one root node, this will not require any iteration or recursion through the tree. The average time complexity is O(n), this is due to recursively calling each node along with its children till the all the nodes in the tree are displayed. This is the case regardless pf Pre, In order or Post. The worst time complexity is O(n), when the tree acts as a linked list due to the tree being entirely unbalanced down one side. This subsequently requires the recursive calling of the function till it reaches the end of the nodes in the tree.

## The destructor

Performs a deepDelete() function. The average and worst time complexity is O(n), as it will need to iterate through every node and perform the operator delete function. The best time complexity is O(1), this is when only the root node will need to be deleted.

## remove()

Performs a recursive remove. The recursive function has the same average and worst time complexity as lookup() and insert(), assuming the tree is balanced. The best time complexity is O(1), this is when the root has no children and has no key.

## displayTree()

Performs a recursive displayTree. The best time complexity is O(1), where it would have to visit every node once. The average and worst time complexity is O(n), where the number of iterations of inner loop is proportional to the square of the iterations in the outer loop.

## rotateLeft() and rotateRight()

Performs the rotation function which changes the shape of the binary tree without changing the nodes that make up the tree. The best, average and worst time complexity is O(1), this is because regardless of the shape of the tree, in regards to number of nodes and balancing.

## The copy constructor

Performs the deepCopy() function, this iterates through every node and its respective children- resulting in O(n). The best time complexity is O(1), only the root node is present. The average and worst are O(n),

## The move constructor

Performs the copying of data from original trees root to new trees root and proceeds to make original tree a null pointer. The time complexity for best, average and worst is O(1), this is due to the function having solely the action of changing two pointers.

## The copy assignment operator

Performs the deepDelete() function which clears the binary search tree structure, then if suitable performs deepCopy() function which returns a data structure that is equal value as the original tree. The average and worst time complexity is O(n), this is when both deepDelete() and deepCopy() are executed. The best time complexity is O(1).

## The move assignment operator

Performs the copying of the value of root to a new tree’s root then proceeds to delete the original root. This function always changes one pointer. The average and worst time complexity is O(1). The best time complexity is O(1), this is when the function does not take the long route through the if statement.

## removeIf()

Performs the removal of nodes from the tree that meet a specific condition. Nodes that need to be deleted are contained within a list that is created. The best time complexity is O(1), this is when the tree has only one root node, in this scenario the condition should not be met. The average time complexity is O(nlog(n)), this is when the tree is balanced. The nodes to be removed is within the condition. The searching algorithm will need to reach the nodes specified of the binary tree with the standard lookup algorithm, this in turn will create a vector of pointers to be deleted in a for loop, with complexity O(n). The worst time complexity is O(n2), this is when the specified condition meets all the potential nodes in the in the tree. The algorithm will need to search through all items in the tree and add them to the list it creates. After the list has all the elements it will need to be traversed and all nodes will need to be removed by the removal function.

# Task 3: Justifying Implementation Choices

## std::list

This is a container class in the C++ Standard Template Library. It is a type of sequence container that allows constant time insertion and removal of elements from anywhere within the container, as well as iteration through the elements in linear time. The sequence container represents a doubly linked list. This means that each element can be linked to its previous and next elements in the list.

The insert method allows the addition of element at a specific position in the list. It takes an iterator pointing to the position where the element should be inserted. The time complexity of this is O(n)

The push\_back and push\_front methods can be used to add elements to the end or beginning of the list. The time complexity of this is O(1)

The remove\_if method is used to remove elements from a list based on a specific condition. The elements are not actually deleted from the container- they are moved to the end of the container and returns an iterator to the new end of the range. The time complexity of this is O(n)

The erase method is used with remove\_if to ensure elements are removed. The time complexity of this is O(n) if the element being removed is not at the end of the list. Otherwise the time complexity is O(1) when the element being removed is at the end of the list.

## std::map

This is a container class in the C++ Standard Template Library. It is used to store a collection of key-value pairs/ it is implemented using a red black tree. This subsequently allows the efficient insertion, deletion and lookup of elements based on their keys, unlike unbalanced trees which will have worse time complexities.

The worst time complexity when searching an unbalanced binary tree is O(n). This is because the tree will behave as a singly linked list. When this is the case the algorithm must traverse through every single node in order. This behaviour is impossible to occur in a red black tree as it cannot become unbalanced. Subsequently the worst time complexity is O(log2(n))

The time complexity for insertion, deletion and search is O(log(n)). This is due to the time it takes to carry out these operations increases logarithmically as the number of elements in the map increases.

## std::unordered\_map

This is an associative container in the C++ Standard Template Library. It is implemented using a hash table; unlike std::map which is a self balancing binary search tree.

The time complexity for insertion, deletion and search is O(1). The time it takes to perform the operations does not depend on the number of elements in the hash table.

However, in the situation that two or more keys are mapped to the same array index a hash collision will occur. In the worst case of hash collisions, the time complexity of operations can degrade to O(n). Hash collisions cause elements to be stored in the same bucket.

The optimal solution algorithm will have a complexity analysis of O(n). This is due to this solution having list with unordered map. Meaning O(n) x O(n)= O(2n)= O(n)

The worst solution algorithm will have a complexity analysis of O(nlog(n). This is due to this solution having list with unordered map. Meaning O(n) x O(log(n))= O(nlog(n))

# Task 4: Understanding Data Structures

A data structure that can store randomly inserted bricks and search them efficiently can be an approach for the algorithm. This can be achieved by std::map or std::unordered\_map.

The std::list container is the least fitting for implementation of a data structure. This is due to the search operation being inefficient when the required brick is situated at the end of the list. However, initial creation is efficient. Also, after a brick is found it can be removed quickly from the list. Therefore, subsequent searches are improved in efficiency.

Implementation of std::map would enable efficient operations such as finding bricks and deleting them once they have been found. However this can be slower than std::unordered\_map which uses a hash table. This is due to operations potentially needing to re balance the tree to ensure ordering. In a map the order is determined by the key values in the red black tree. The subsequent order of elements in a map is therefore determined by the system, appose to a user. The time complexity is O(log(n))

On the other hand std::unordered\_map utilises a hash table. This means ordering is not required, this subsequently makes creation and searching faster. However deletion of bricks once found is slower than std::map. In an unordered map the order is determined by the hash function in the hash table. The order of the elements in a hash table can change as the table grows, or as elements are inserted/removed. The time complexity is O(1)

The other data structure needed is one that can insert data to the front and back of the data structure. A list would be most appropriate for this implementation due to the requirement of maintaining the sorted data. The list can have bricks pushed front or back. Furthermore, the list uses a user defined order, unlike map and unordered map.

For the first data structure an unordered map to store the unsorted data with a list to store the sorted data is best. This is due to the unordered map having fast random access. The total number of bricks is known, this allows more efficient insertion and searching due to the hash table’s ability of its size being set. Time complexity of insertion and search is O(1)

The first data structure could also be achieved by a map instead of an unordered map. This is due to map having an average time complexity of O(log2(n)). Map has quick searching, but slow insertion. Due to map being implemented by a red black tree during insertion it may need to recolour the tree a number of times. This has time complexity of O(log2(n)). Altogether this approach is worse than unordered map and list.

The second data structure would be best as a list. This is due to the time insertion being constant at both front and back. This is exactly what the data structure requires.

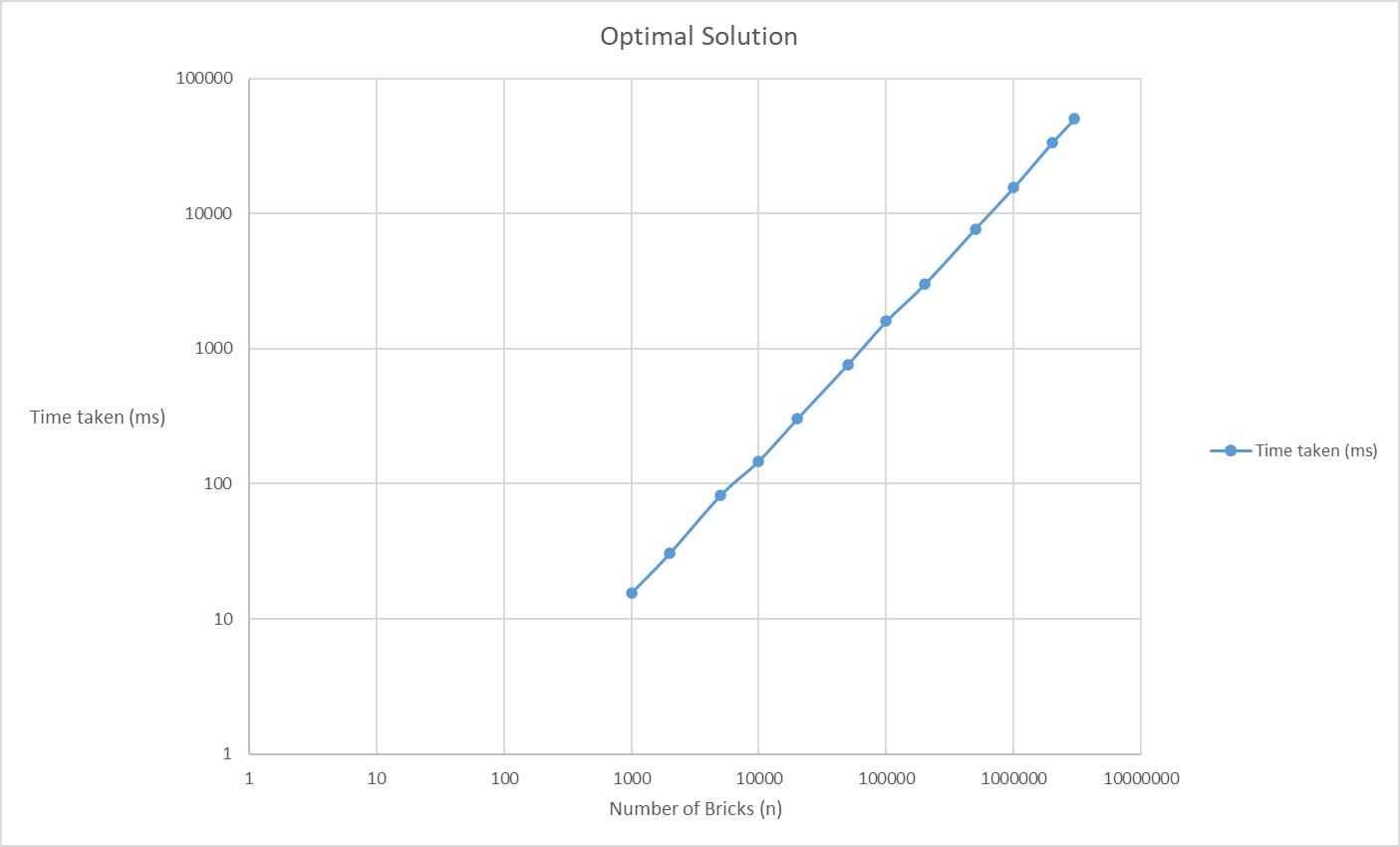
# Task 5b: Presenting the Royal Software Engineer's Results

Table

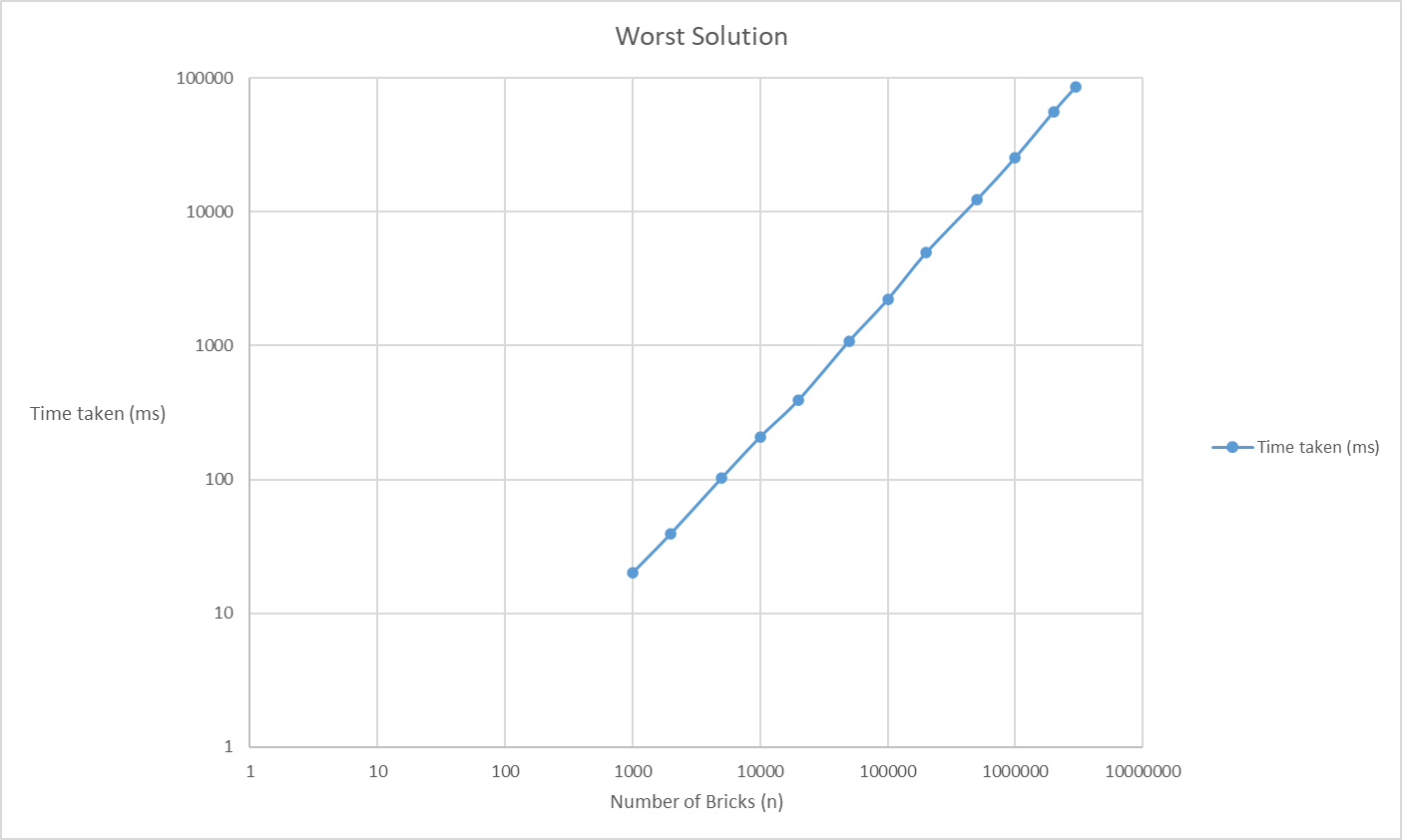


Table

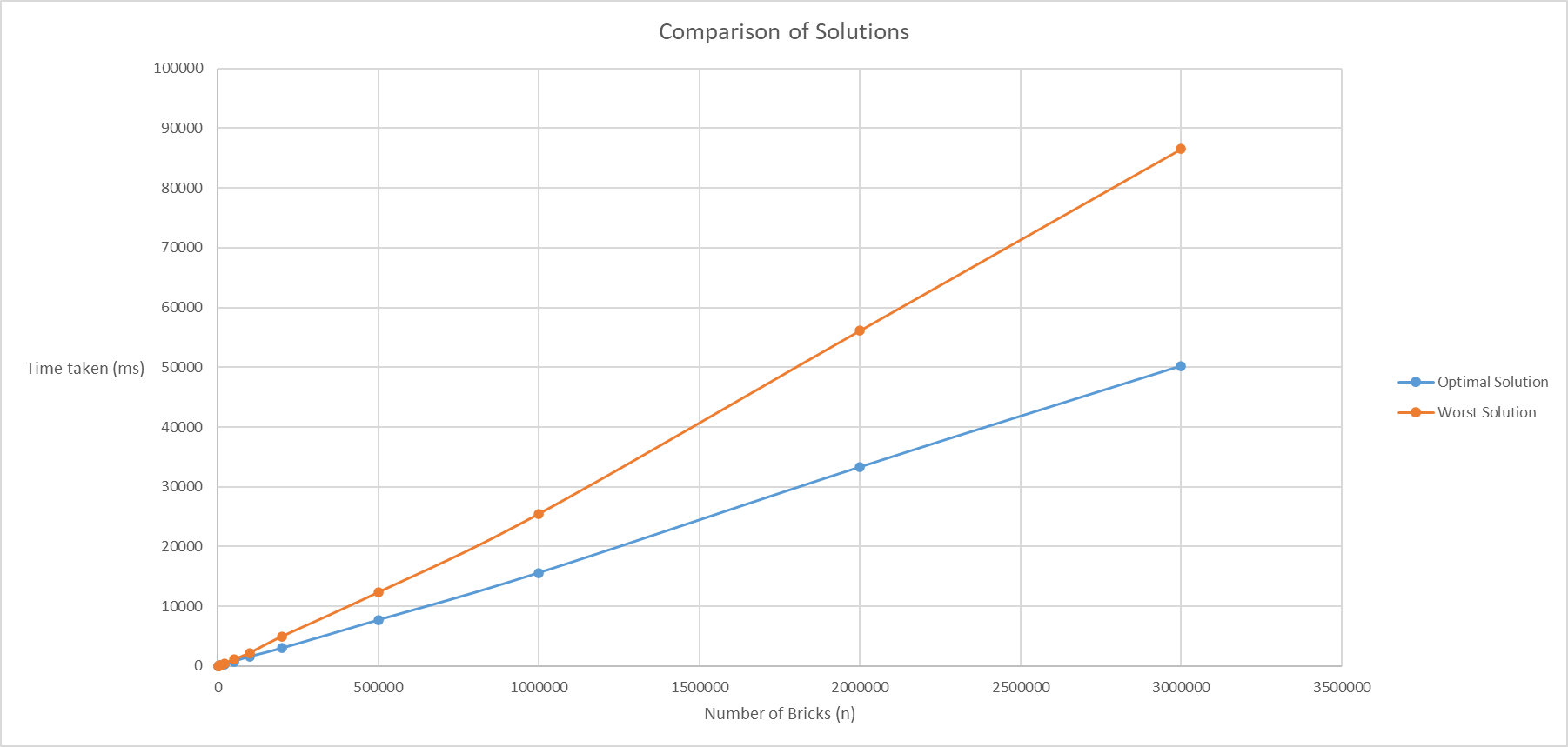




Figure



Figure



Figure

# Task 5c: Evaluating the Royal Software Engineer's Algorithm

Table 1 displays the results table for six runs of the optimal solution. The average was calculated by runs 2-6 due to the first run being treated as a *warmup* run. The first run had faster times than all subsequent runs. This aided the decision to disregard this run when calculating the average.

Figure 1 displays the optimal solution. The graph is on a logarithmic scale, this helps display the fact that the growth is constant. The graph has a complexity of O(n).

Table 2 displays the results table for six runs of the worst solution. Just like the optimal solution the average was calculated by runs 2-6. This was to ensure the average was calculated by the same number of runs (5) as the optimal average. Similarly, the first run was increasingly slower than the subsequent runs.

Figure 2 displays the optimal solution. The graph is on a logarithmic scale, this helps display the growth of time taken as the number of bricks increase. The graph has a complexity O(nlog(n)).

The results indicate that the optimal solution was faster than the worst solution. The graph in Figure 3 shows that the optimal solution time complexity is below that of the worst solution. The difference in time taken for higher bricks clearly displays the efficiency of the optimal solution.

# Task 6d: Presenting the Royal Mathematician's Results

Table 3



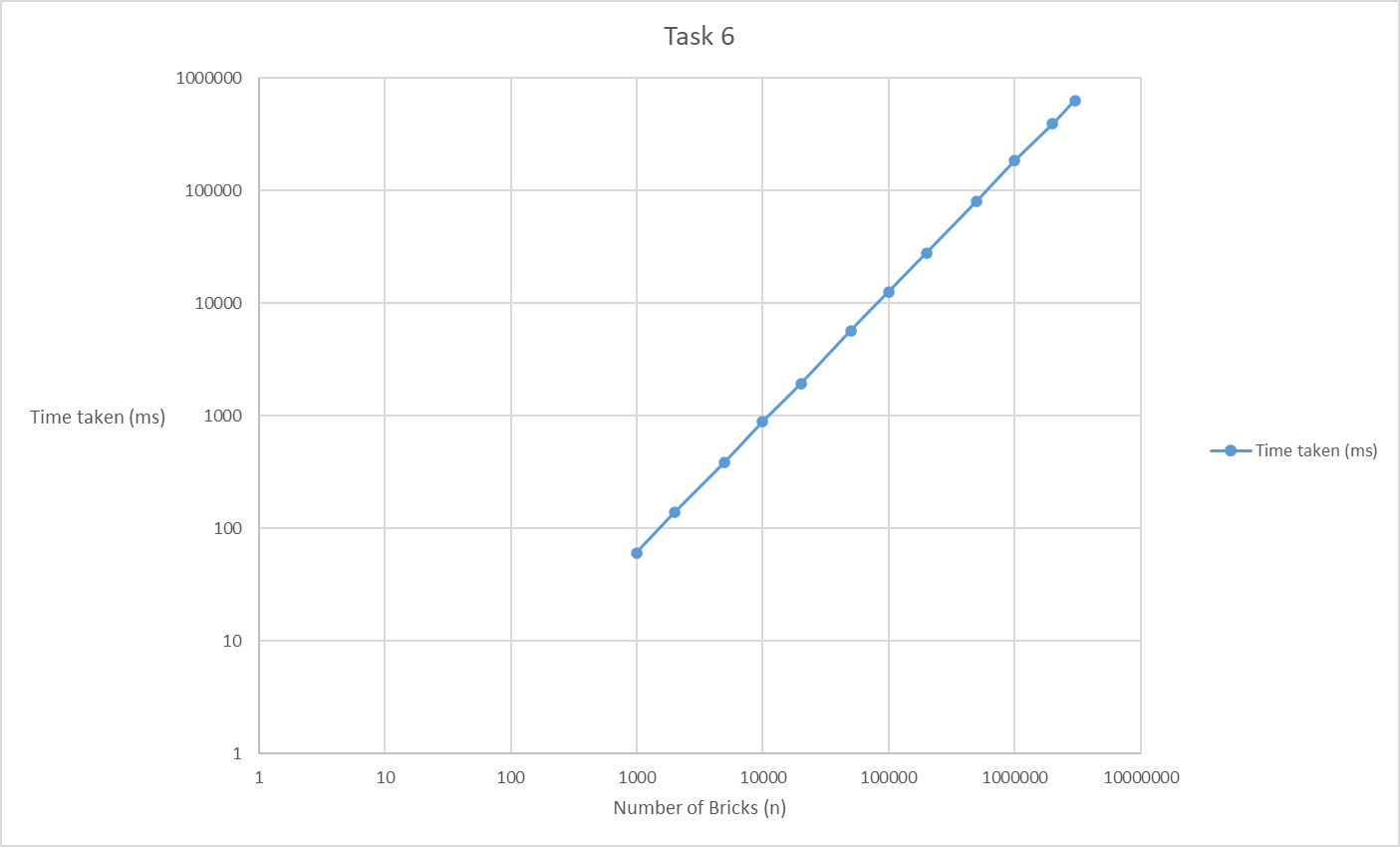


Figure 4

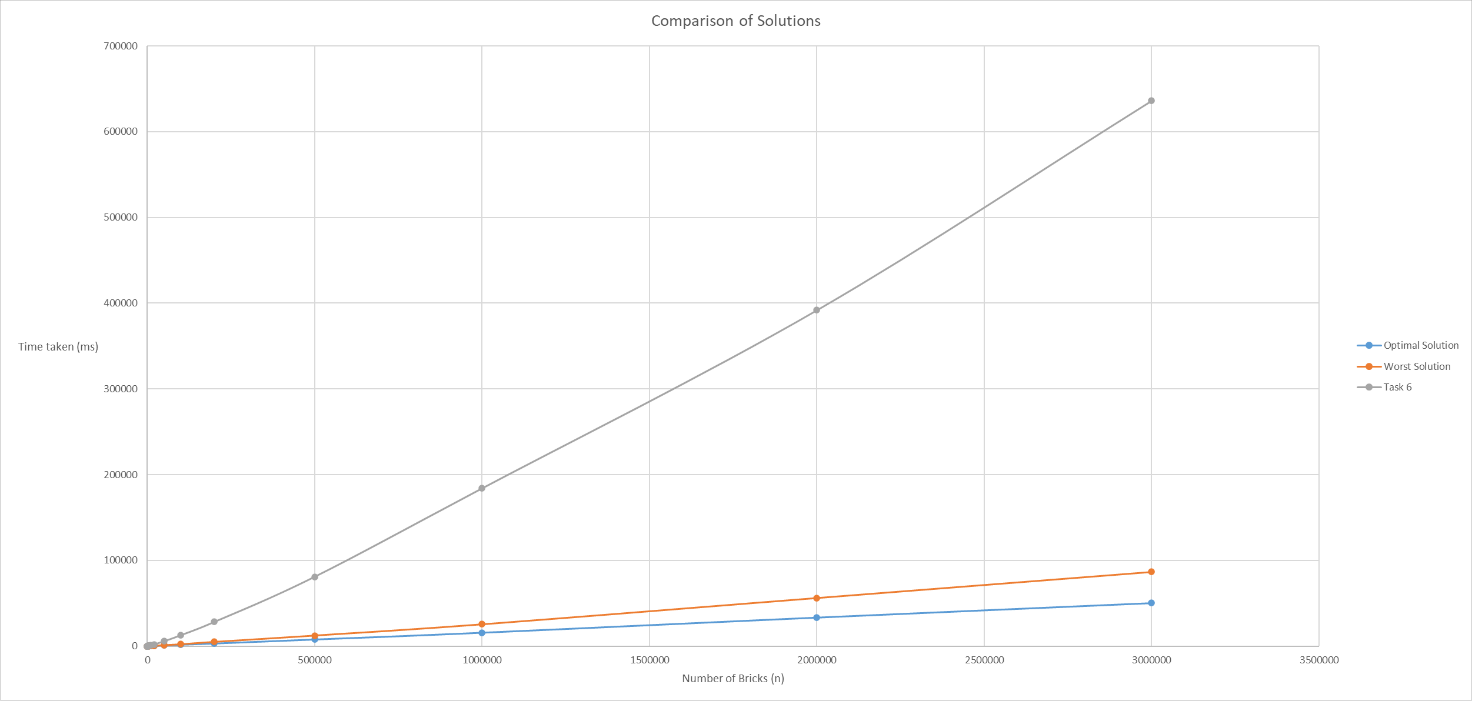


Figure 5

# Task 6e: Evaluating the Royal Mathematician's Algorithm

Figure 4 displays the time taken for the Royal Mathematician's Algorithm. The graph is on a logarithmic scale, this helps deal with skewness in the data as there are discrepancies in the time taken for the

Figure 5 compares the optimal and worst solutions from the Royal Software Engineer's Algorithm with the solution for the Royal Mathematician's Algorithm. The graph clearly indicates just how much longer the Royal Mathematician's Algorithm takes compared to that of either solution in the Royal Software Engineer's Algorithm. The Royal Mathematician's Algorithm shows a strong linear correlation of results.

# Bibliography

(n.d.). Retrieved from cppreference.com: https://en.cppreference.com/w/cpp/container/list

(n.d.). Retrieved from cppreference.com: https://en.cppreference.com/w/cpp/container/map

(n.d.). Retrieved from cppreference.com: https://en.cppreference.com/w/cpp/container/unordered\_map

(2022, Jun 29). Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/which-data-structure-is-used-by-map/

(2022, Nov 09). Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/map-vs-unordered\_map-c/