Software Development Investigation

Advanced Software Engineering SOFT30161

Name - Luke Sutton

Student ID - N0913070

Date - 16/12/2022

A red sign with white text

Description automatically generated with medium confidence

Table of Contents

[Task 1 4](#_Toc122060832)

[Lookup(), Insert(), Remove() 4](#_Toc122060833)

[displayEntries(), The destructor, displayTree() 4](#_Toc122060834)

[rotateLeft() and rotateRight() 4](#_Toc122060835)

[The copy constructor 4](#_Toc122060836)

[The move constructor 5](#_Toc122060837)

[The copy assignment constructor 5](#_Toc122060838)

[removeIf() 5](#_Toc122060839)

[Task 3 5](#_Toc122060840)

[Justifying Task2a library container choices – Average Case 5](#_Toc122060841)

[Justifying Task2b library container choices – Worst Case 5](#_Toc122060842)

[Justifying Task2b Attempt 2 library container choices – Worst Case 5](#_Toc122060843)

[Justifying each step: 6](#_Toc122060844)

[Loading the data into main memory 6](#_Toc122060845)

[Arbitrarily choosing a starting point 6](#_Toc122060846)

[Constructing a result sequence 7](#_Toc122060847)

[Searching for bricks with northern symbols 7](#_Toc122060848)

[Adding to back of result sequence 8](#_Toc122060849)

[Searching for bricks using southern symbols 8](#_Toc122060850)

[Adding to front of result sequence 8](#_Toc122060851)

[Printing the results – Not Required 8](#_Toc122060852)

[Overall estimated average time complexity 2A 8](#_Toc122060853)

[Overall estimated worst case time complexity 2B 9](#_Toc122060854)

[Overall estimated worst case time complexity 2B Attempt 2 9](#_Toc122060855)

[Task 4 10](#_Toc122060856)

[std::list 10](#_Toc122060857)

[std::map 10](#_Toc122060858)

[std::unordered\_map 10](#_Toc122060859)

[Task 5 12](#_Toc122060860)

[Task 6 14](#_Toc122060861)

[Analysing the Royal Mathematician’s Algorithm – Worst case 14](#_Toc122060862)

[Analysing the data from task 6 15](#_Toc122060863)

[Appendix 18](#_Toc122060864)

[19](#_Toc122060865)

[20](#_Toc122060866)

[21](#_Toc122060867)

## Table of Figures

[Figure 1 A graph comparing task 2a's average run time to task 2b's average run time 13](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060868)

[Figure 2 Graph showing the logarithmic average run time for task 6 15](#_Toc122060869)

[Figure 3 Task2A Average Run Time 18](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060870)

[Figure 4 Task2B Average Run Time 18](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060871)

[Figure 5 Task2B Logarithmic Average Run Time 19](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060872)

[Figure 6 Task2A Logarithmic Average Run Time 19](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060873)

[Figure 7 Task6 Logarithmic Average Run Time 21](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060874)

[Figure 8 Task6 Average Run Time 21](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060875)

## Table of Tables

[Table 1 Timing results for task2A in seconds 12](#_Toc122060876)

[Table 2 Timing results for task2B in seconds 12](#_Toc122060877)

[Table 3 Table showing 6 result data in seconds 20](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060878)

[Table 4 Table showing 2B result data in seconds 20](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060879)

[Table 5 Table showing 2A result data in seconds 20](https://myntuac-my.sharepoint.com/personal/n0913070_my_ntu_ac_uk/Documents/Software%20Development%20Investigation.docx#_Toc122060880)

# Task 1

For the first task I have been asked to identify the time complexities of the algorithms implemented in my codebase submitted. These time complexities will include the best, average, and worst cases for each member function. When using these features with large datasets the constant costs can be disregarded as the dominant term grows much faster than the constant terms. For example an operation with cost O(n^2) will cause a much greater cost than an operation with a cost of O(n).

Due to the nature of binary trees, the best case of each function will only have 1 node in the tree and so many of the functions will have a best case notation of O(1). For average cases I have assumed the tree is balanced as when assessing the probabilities of a node being a left or right child it is a 50/50 chance, this leads to a balanced tree.

## Lookup(), Insert(), Remove()

The best case of lookup, insert and remove occurs when the sought after key is the root node of the tree. This means that the function has a big O notation of O(1).

The worst case occurs when the tree is perfectly imbalanced and the key is not found, the functions will have to go through every node in the tree to the very end. This means the search has a worst-case complexity of O(n).

The average case is O(log n). This is because the average cost of a lookup requires traversing through half the tree. When the cost of lookup is scaled to a larger data set we still must navigate half the tree’s nodes and so can disregard the constant costs of the other functions such as adding the lookup function to the stack frame or removing the node at the end of the search.

## displayEntries(), The destructor, displayTree()

The best case of these functions is O(1). If the tree only contains one root node the cost will simply be showing/deleting the value of that root node.

The worst case is O(n) as every node must be visited and displayed/deleted.

The average case is O(n), this is because we have to display/remove all the values in the tree, this means recursively calling each node and its children until we have visited all the nodes in the tree.

## rotateLeft() and rotateRight()

The best and worst case for a rotate is O(1), this is because the rotation does not scale with the number of nodes, it simply reassigns the pointers.

The average case is O(1), this is because no matter what tree it is, the rotation is 1 set of functions without any loops or reliance on iteration, recursion or loops.

## The copy constructor

The best case for a deep copy is O(1), when only 1 node is present to be copied to another dictionary.

The average and worst case for a deep copy is O(n), this is because every node in the dictionary must be visited to copy to the other dictionary using recursive calls.

## The move constructor

The best case for move is O(1), when there is only 1 node to move. The deep copy function does not need to recursively call itself and can just perform the move directly.

The average and worst case for move is O(n), similar to copy, this is because each node must be visited at least once by the deep delete worker function.

## The copy assignment constructor

The best case for a copy assignment is O(1), this occurs when only 1 node is assigned a new value from another existing object.

The average and worst case for the copy assignment is O(n), every node and its subsequent data must be visited and copied over from the existing object to the new one.

## removeIf()

The best case for removeIf is O(1), this occurs when the tree only has 1 node or when no nodes match the keys set to be removed.

The worst case is O(n x k), this occurs when all the nodes qualify to be removed and so the function must recursively pass through the whole tree removing all the nodes O(n). It must then iterate through the list of keys and remove them one by one which gives the other O(k), these combine to produce O(n^2) when k = n.

The average case is O(k log n). The O(log n) part of the complexity is due to remove acting similarly to lookup. The possible nodes that the function could check change logarithmically due to the height. The second part of the complexity O(k) is gained from iterating through all the keys that have passed the function check and have been entered into the vector. The size of K can have very little impact if the amount of keys tends towards zero, however if the number of keys is similar to the total number of nodes we could rewrite the complexity as O(n x log(n))

# Task 3

The library container options for the two tasks were: std::vector, std::list, std::map and std::unordered\_map.

## Justifying Task2a library container choices – Average Case

I chose to use the std::unordered\_map and its member functions for the vast majority of the task. This is because unordered\_map has an average case time complexity of O(1) for searching, inserting and deleting when using its member functions.

## Justifying Task2b library container choices – Worst Case

For Task2b I chose to use std::list for inserting the main list of bricks O(n) and std::map for searching the bricks for their matching pairs O(log n).

## Justifying Task2b Attempt 2 library container choices – Worst Case

For my second iteration of Task2b I inserted into std::map and then output the results to a std::list, this cut out (highlighted in red below) inserting the bricks into a main list initially as my first attempt for Task2b had the following workflow: Read file -> insert into list -> insert into map -> search map ->output to result.

## Justifying each step:

### Loading the data into main memory

#### 2A, 2B and 2B attempt 2

I iterated through the text file using ifstream’s getline() to get each line, this function is O(n) where n is the number of lines in the file. This is because getline has to iterate through every line in the file so the number of lines it has to read is directly proportional to the lines in the file.

Then the program splits the line into two, using the comma as a delimiter and std::strings member function substring() to acquire the two strings by searching through the characters on the line one by one. Substring() has a time complexity of O(m) where m is the number of characters being extracted. One thing to note is that there are 2 strings per line but as this is a constant factor we transform it from O(n/2) to O(n). These two combine to form O(n + m) but as the m is constant we can ignore it and use solely O(n.)

#### 2A

These strings were then inserted into two unordered maps, the average insertion into an unordered map is O(1).

Alternatively, I could have inserted these substrings into a vector with an average time complexity of O(n), however it is significantly slower and with large quantities of data the vector would have to keep resizing and copying over the old vector as it ran out of container space.

Map insertion is also slower than unordered map insertion, it has an average time complexity of O(log n) as std::map is based on a red black binary search tree where inserting new values requires iterating down the branches.

List insertion has the same time complexity as unordered map for insertion O(1) but has a slower search O(n) due to having to iterate through the entire list to find the element.

#### 2B

These strings were then inserted into two lists, the worst case insertion into a list is O(1) as the values are added to the back using .pushBack().

List has the best worst case time complexity of O(1) out of the given containers, map’s is O(log n), vector’s is O(n) and unordered map’s is O(n).

#### 2B Attempt 2

The strings were inserted straight into a std::map, this had a higher complexity than the list O(log n) compared to O(1) but I was not using the list for anything except inserting straight into the map.

### Arbitrarily choosing a starting point

#### 2A

After the unordered maps were created I used the member function .begin() to find the first set of values in the first map. Begin() has a time complexity of O(1) as it only returns an iterator pointing to the first element in the map.

Once the bricks are already in an unordered map it is faster to use the begin() member function than to copy them to any other container such as a vector. This is because copying and inserting into a vector is O(n) even if getting the first element by indexing is O(1).

#### 2B

The starting point was chosen as the first word in list by using the list member function .front(), this has a worst case time complexity of O(1) due to only returning a pointer to the first element in the list.

#### 2B Attempt 2

The starting point was assigned using a local variable on map creation, so had a complexity of O(1).

### Constructing a result sequence

#### 2A

The result sequence was created by initialising an empty unordered map, this has an average time complexity of O(1) as there is no scaling involved.

The result sequence could have been created as a different container, but we want to be able to efficiently insert into it and search it later into the program and unordered map offers these operations with a time complexity of O(1).

#### 2B and 2B Attempt 2

The result sequence was created using a std::list, this was mainly due to the fact that list has the best worst case time complexity for inserting into the list O(1).

### Searching for bricks with northern symbols

#### 2A and 2B

Next the function to find the southern symbol names is called and passed the relevant arguments by reference (note this means no copying is needed and so the time complexity is O(1) rather than O(n)).

#### 2A

The function then uses a while loop in combination with unordered map’s member functions .find(), .end() and .empty() to find the correct keys in the map.

Find() as a member function has an average time complexity of O(1) this is because unordered maps use hashing to store and retrieve data. End() also has a time complexity of O(1), it is a function that simply returns an iterator pointing to one past the end of the elements in the map.

This loop calls find() multiple times to find all bricks after the startBrick/start point so overall this loop has a time complexity of O(n – k) where k is the number of bricks before the startBrick. The actual comparisons between the values found and local values on the stack have a constant time complexity of O(1) so can be disregarded.

Alternatively, list has a time complexity of O(n) as on average the algorithm must iterate through the list and examine whether the element is the one that satisfies the condition one by one. Searching a vector also has a time complexity of O(n) as it is required to perform the same operation as Listl; both containers do not offer any built-in functions to quickly locate an element that matches a condition. Map offers O(log n) search time complexity as it is implemented as a binary search tree so the height is log n of the number of nodes in the tree.

#### 2B

For the worst case we use map’s member function find(), this has a complexity of O(log n) as map is implemented as a binary tree.

### Adding to back of result sequence

#### 2A

The program inserts the found pair of symbols to an unordered map that can later be accessed for printing the results. This insertion into an unordered map has a complexity of O(1).

#### 2B and 2B attempt 2

The result is placed at the end of the result list via push\_Back(), this has a complexity of O(1) as it appends to the end of the list.

### Searching for bricks using southern symbols

#### 2A

The function then uses a while loop in combination with unordered map’s member functions .find(), .end() and .empty() to find the correct keys in the map.

The next brick is found by calling find() one extra time to find the startBrick key value’s second value, one other negligible difference is that instead of using the “bricks” unordered map it uses the reversed “bricksBack” unordered map.

#### 2B and 2B attempt 2

Similar to finding the southern symbols, this implementation makes use of an additional find() per loop, however we can dismiss this as it is a constant addition.

### Adding to front of result sequence

#### 2A

Found pairs are inserted into the result sequence map with a time complexity of O(1)

#### 2B

Uses .push\_front() instead of .push\_back(), however they share the same time complexity as std::list is implemented as a doubly lined list and as such can append to the front at O(1).

### Printing the results – Not Required

#### 2A, 2B and 2B attempt 2

No matter what container we choose, printing the list will always have a time complexity of O(n) as every result is printed on a new line.

If we ignore all the negligible constant time and just look at the highest complexities, we get the following:

## Overall estimated average time complexity 2A

Reading of file using getline = O(n)

Using substring for each line of file to find each character in the file = O(m), where m is the length of the returned string

Choosing startBrick = O(1)

Creating result sequence = O(1)

Add southern symbols = O(n - startBrick) of twoSouth

Add northern symbols = O(startBrick – n) of twoNorth

twoSouth + twoNorth = O(n)

Overall estimated average time complexity = O(n + m) + O(1) + O(n) = O(n) + O(n) = O(n)

## Overall estimated worst case time complexity 2B

Reading of file using getline = O(n)

Substring = O(m)

Choosing startBrick = O(1)

Creating result sequence = O(1)

Iterate through list to insert into map = O(n)

Add southern symbols =

While loop iterating through the bricks and for each loop: O(n)

Use map’s member function find()to find bricks (north and south = O(log n)

🡪 These combine to O(n x log n) as the find function is inside the loop.

Overall estimated worst case time complexity = O(n + m) + O(1) + O(1) + O(n) + O(n log n) = O(n) + O(n) + O(n log n) = O(n (log n))

## Overall estimated worst case time complexity 2B Attempt 2

Reading of file using getline = O(n)

Substring = O(m)

Choosing startBrick = O(1)

Use map’s member function find() to find bricks (north and south = O(n log n)

Overall estimated worst case time complexity = O(n + m) + O(1) + O(1) + O(n) + O(n log n) = O(n) + O(n) + O(n log n) = O(n (log n))

This resulted in the same time complexity as 2B but should produce a faster real world run time due to one less loop.

# Task 4

## std::list

Standard list is based off a doubly linked list data structure. (Drowell, 2022) (Harvard University, 2022) (Standardization, 2017). Std::list is a sequence structure where each element is connected to the previous and next element by a pointer.

Insertion and deletion into the std::list is O(1), this is true for the average and worst case. We can assume this as the only operations that must occur for insertion or deletion is to change pointers of the element before and after the node getting inserted or deleted. The list itself does not need to be copied or moved in memory as the list is traversed through pointers rather than physically being next to each other in memory addresses. In task 2b I used .push\_back() and .push\_front() to insert elements into the list. These functions have constant time complexity O(1) (cplusplus.com, 2022) (cplusplus.com, 2022) As I never inserted or deleted in the middle of the list amortised time does not need to be calculated as I did not need to iterate and then perform the insertion/deletion.

Searching a list is O(n) for average and worst case. This is because for each search we begin at the start of the list and iterate through by following pointers until we find the element we are looking for. This iteration of n cannot be any greater than the number of the elements in the list as the container has an end pointer that isn’t changed. In task 2b I used .begin() and .end() to return a pointer to the respective beginning and one past the end of the list of elements, these functions have a time complexity of constant time (cplusplus.com, 2022) (cplusplus.com, 2022). These member functions ensure the list only iterated through the elements and did not iterate into unallocated memory.

## std::map

Standard library map is based off of a red black binary search tree data structure. (Standardization, 2017). The map itself is split into key and their subsequent values, these key value pairs are kept in order by following a strict weak ordering comparison of keys.

Insertion into std::map has a time complexity of O(log n) (cplusplus.com, 2022) for average and worst case when not using a hint. This time complexity can be guaranteed as searching a binary tree for the correct position to insert a node takes O(log n) time. This is because each time the algorithm evaluates the search for a node it can eliminate half the nodes that do not qualify the search. This means the time complexity is logarithmic to the number of elements in the tree as the search function of O(log n) has a much greater impact than the simple insertion at the end of O(1).

In task 2b I used .find() to search for the correct brick, this will have iteratively searched through the ordered map to find the matching key element as I described in the insertion paragraph above, giving O(log n) time (cplusplus.com, 2022). I also used end() to return one past the end of the map, this has a constant time complexity of O(1) (cplusplus.com, 2022)

## std::unordered\_map

Standard unordered\_map is based off a HashMap data structure (Standardization, 2017). (cplusplus.com, 2022). Each pair in an unordered map has a key and a value, the library uses the key and passes it into a function to return a unique value of type size\_t. The unordered map uses this hash value/bucket index to organise its elements as each key is assigned a bucket. A bucket is a container that holds a single element in the map that can grow and shrink dynamically.

Insertion into an unordered\_map has a time complexity of O(1) for average case and O(n) for worst case (cplusplus.com, 2022). Sometimes the insertion complexity may be higher, std::unordered map will try to distribute the elements evenly across the buckets. However, if the map becomes full, it will rehash all the elements and allocate more buckets to improve performance. This can increase the time complexity of operations like insert for a one-off operation. However, over a large amount of operations this cost will be relatively small as the increase of performance by rehashing offsets the initial cost, so the amortised time will still tend towards O(1).

Begin() has a time complexity of O(1) (cplusplus.com, 2022) as it returns a single iterator pointing to the first element in the unordered map and does not use iteration or recursion.

End() has a time complexity of O(1) (cplusplus.com, 2022) and returns a single iterator pointing to the last element in the unordered map and does not use iteration or recursion.

Find() has a time complexity of O(1) (cplusplus.com, 2022) for average case. Find is implemented by computing the hash value using the hash function that was created when the map was initialised. By using the hash value, we can locate the bucket with the associated value and the iterate through the values in the bucket, on average there is 1 element in the bucket. If all the values are stored in one bucket, we get a worst case time complexity of O(n)(cplusplus.com, 2022) as we have to iterate through all the elements.

Empty() has a time complexity of O(1) (cplusplus.com, 2022), it checks if the size is equal to 0 and returns a bool.

# Task 5

To test the programs from task 2 I completed five runs of single file testing, this means for each testing file provided it had a warmup run (so that elements being loaded into memory would have less effect on the accuracy of results) and then four runs of data were collected. Data quantities were omitted below 10,000 elements, this was to try and reduce the noise from background memory processes on the machine from affecting the runtime speed.

Full data tables with the warmup run are available in the appendix as well as graphs showing the tasks separately. Seconds has been chosen as the unit of measurement so that readers can accurately read the data and relate it to real world experience.

The data tables can be found below:

Table 1 Timing results for task2A in seconds

Table

Description automatically generated

Table 2 Timing results for task2B in seconds

A picture containing graphical user interface

Description automatically generated

The data clearly shows that task 2a completes faster than task2b at higher quantities of data. This supports the earlier hypothesis, task2a was predicted to have an average complexity of O(n) and task2b was predicted to have a worst case complexity of O(n log(n))

The real world run time for the algorithms is affected by constants and lower impact complexities. For example for three million bricks Task2a shows an average of 45 seconds and Task2b shows 61 seconds. For the notation to perfectly match the real world example if we use any average and add its log value it should equal 2b’s average.

For example if we use the 3 million test, 45 + log(45) = 74 seconds which is not quite equal to 61.37 seconds as shown in 2b’s table.

Both algorithms can be clearly seen in the graph labelled figure 2, the log graph confirm the linear growth as the lines plotted are relatively straight when scaled correctly onto a logarithmic time axis.

Figure 1 A graph comparing task 2a's average run time to task 2b's average run time

Figure 2 A graph comparing task 2a's logarithmic run time to task 2b's logarithmic run time

# Task 6

## Analysing the Royal Mathematician’s Algorithm – Worst case

The Royal Mathematician’s algorithm contains many different types of operations, many of them are considered O(1) for worst case analysis. For this analysis most of these efficient operations will be omitted to not affect the clarity of the breakdown.

Opening the input file and iterating line by line to the end = O(n)

For each line in the file iterate through each character = O(m)

Pushback onto the tape lists A and B = O(n)

Sort the tapes by iterating using sort() and a lambda function= O(n log n). For each pair in the tape, merge sort it using character comparison, the worst case occurs when the tape requires the maximum number of iterations to sort.

Comparing tape A to B = O(n^2) where n is the number of elements in tape A. We get O(n^2) from the nested loop, for each element in tape A we have to iterate through tape B until the second element of tape A is bigger than tapeB’s first. If this occurs on the last element every time we have to iterate through the whole of B for every pair in A.

Working out the distance between the start of the tape and the current iterator = O(1)

Get iterators to the start of the tapes = O(1)

Iterating through tape A = O(n)

Iterating while d < Number of elements = O(n)

Deep copying tape C = O(n) and sorting the tapes again = O(n)

Iterating through tape A to find matching values in B = O(n^2). As stated earlier if for each pair in A we don’t find the matching pair until the last value of B we have to iterate through the B tape for every single pair in the A tape.

Comparing pair values = O(1)

Inserting data onto back of tape lists using push\_back() = O(1)

Adjusting pointers = O(1)

Merging tapes by comparing values and inserting = O(1) (cplusplus.com, 2022)

Overall, the worst-case time complexity is = O(n^2) as we can discard the complexities that have a lesser effect on the runtime and there is no function that has a greater time complexity than O(n^2).

## Analysing the data from task 6

Graphical user interface

Description automatically generated with low confidenceTo test the program from task 6 I completed five runs of single file testing, this means for each testing file provided it had a warmup run (so that elements being loaded into memory would have less effect on the accuracy of results) and then four runs of data were collected. Data quantities were omitted below 10,000 elements, this was to try and reduce the noise from background memory processes on the machine from affecting the runtime speed.

A full data table with a higher degree of accuracy and the warmup run is available in the appendix. Seconds has been chosen as the unit of measurement so that readers can accurately read the data and relate it to real world experience.

Overall, the run time seems to grow linearly with the brick element increase, this can be visualised in the graph below.

Figure 2 Graph showing the logarithmic average run time for task 6

Earlier in the report I predicted a complexity of O(n^2) and the data seems to support this. We can see this increase of O(n^2) in the data table, for example at 1million bricks the average run took 174 seconds, when we doubled the bricks to 2 million, the time multiplied by 2.29. The extra 0.29 likely comes from constant values and less impactful O(log n) functions that are ignored when evaluating big O notation.

The mathematician’s algorithm ended up performing much worse than the software engineer’s algorithm, this can be visualised in the graph below.

The large difference in runtime probably can be accounted for due to all the additional iterations in the mathematician’s implementation, this cannot be easily visualised but shows itself clearly here.

From the logarithmic graph we can see the growth of run time for task 6 is constant as the line is straight. We can also see that task 2a and 2b are growing very similarly to task 6, showing their own O(n) and O(n log n) complexities respectively.

# 

# References

cplusplus.com. (2022). Retrieved from https://cplusplus.com/reference/map/map/insert/

cplusplus.com. (2022). *Map::Find*. Retrieved from https://cplusplus.com/reference/map/map/find/

cplusplus.com. (2022). *std::list*. Retrieved from https://cplusplus.com/reference/list/list/begin/

cplusplus.com. (2022). *std::list::end*. Retrieved from https://cplusplus.com/reference/list/list/end/

cplusplus.com. (2022). *std::list::merge*. Retrieved from https://cplusplus.com/reference/list/list/merge/

cplusplus.com. (2022). *std::list::push\_back*. Retrieved from https://cplusplus.com/reference/list/list/push\_back/

cplusplus.com. (2022). *std::list::push\_front*. Retrieved from https://cplusplus.com/reference/list/list/push\_front/

cplusplus.com. (2022). *std::map::end*. Retrieved from https://cplusplus.com/reference/map/map/end/

cplusplus.com. (2022). *std::map::find*. Retrieved from https://cplusplus.com/reference/map/map/find/

cplusplus.com. (2022). *std::unordered\_map::begin*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/begin/

cplusplus.com. (2022). *std::unordered\_map::empty*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/empty/

cplusplus.com. (2022). *std::unordered\_map::end*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/end/

cplusplus.com. (2022). *std::unordered\_map::find*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/find/

cplusplus.com. (2022). *std::unordered\_map::find*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/find/

cplusplus.com. (2022). *std::unordered\_map::insert*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/insert/

cplusplus.com. (2022). *unordered\_map*. Retrieved from https://cplusplus.com/reference/unordered\_map/unordered\_map/

Drowell, E. (2022). *Big-O Cheat Sheet*. Retrieved 2022, from https://www.bigocheatsheet.com/

Harvard University. (2022). *C++ Data structures*. Retrieved from https://cs61.seas.harvard.edu/site/2021/Section1/#:~:text=std%3A%3Alist%20(doubly%2Dlinked,implements%20a%20doubly%2Dlinked%20list.

Standardization, I. O. (2017). *Working Draft, Standard for Programming.* ©ISO/IEC. Retrieved from https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf

# Appendix

Figure 3 Task2A Average Run Time

Figure 4 Task2B Average Run Time

# 

Figure 5 Task2B Logarithmic Average Run Time

Figure 6 Task2A Logarithmic Average Run Time

# Table Description automatically generatedTable Description automatically generatedTable Description automatically generated

Table 3 Table showing 6 result data in seconds

Table 4 Table showing 2B result data in seconds

Table 5 Table showing 2A result data in seconds

# 

Figure 7 Task6 Logarithmic Average Run Time

Figure 8 Task6 Average Run Time