CSE 511: Advanced Algorithms

Project Report On

Designing an efficient algorithm that can search through unsorted collection of elements in less time than existing algorithms

Submitted By

Name	***ID***
Trina Saha	
Md. Shahadul Alam Patwary	

Table of Abbreviations

Abbreviation	***Explanation***
OLS	Optimized Linear Search
ULS	Un-optimized Linear Search
BFS	Bring Front Search
UMS	Unordered Map Search
OMS	Ordered Map Search
DAS	Dynamic Array Search

Objective

The goal of this project is to design an algorithm that can search through unsorted collection of elements with reduced running time/execution time than existing techniques.

Introduction

Searching efficiently in an unsorted collection of elements within the shortest possible time is one of the most challenging tasks in algorithms. If an array is sorted, then a binary search[1] could be performed in $O(\log_2(n))$ time. But in case of unsorted array, binary search[1] is not applicable. There are few existing approaches to search through unsorted array i.e. ***linear search (LS)***, ***sorting and applying binary search***[1], ***unordered map search (UMS),*** and ***ordered map search (OMS)*** etc.

Linear search (LS) is the simplest way of searching for an element in an array. It is basically nothing but iterating through each element of array sequentially and checking if it matches with the desired value. So, the time complexity of linear search is $O(n)$.

Sorting array and applying binary search[1] is not a brilliant idea. Because, if we assume that we sort an array using merge sort that takes $O(n\log_2(n))$ time in worst case and then apply Binary search[1] which takes $O(\log_2(n))$ time, the total time complexity becomes $O(n\log_2(n)) + O(\log_2(n)) = O(n\log_2(n))$ which is worse than simple linear search.

Unordered map search (UMS) is a searching technique that uses hashing. When an element is inserted into the unordered map, it generates hash value for that element. The hash value is used as the index and that element is inserted into the specified index (hash value). The function that generates hash value for a given element (key) is called hash function. So, searching an element in unordered map takes $O(1)$ time (time required by hash function). A limitation of this approach is that, an unordered map cannot have duplicate elements.

Ordered map search (OMS) is also a tree based searching technique that does not use hashing. Instead, it uses binary search tree[2] in which elements are stored in sorted order so that the principle of binary search[1] can be used to find an element. On average, each comparison allows the search operation to skip about half of the tree. So, each lookup takes time proportional to the logarithm of the number of items stored in the tree. Space complexity for this algorithm is $O(n)$ and time complexity is $O(\log_2(n))$. The limitation of this approach is same as the unordered map search, it cannot have duplicate elements.

Our goal is to design an algorithm that outperforms the existing searching algorithms and at the same time, overcomes the limitation(s).

Features

Our algorithm has following features:

- Search by index takes O(1) time (just like an array)
- Search by value takes $O(\log_2(n))$ time (just like binary search[1])
- No limitation of duplicate entries
- A hybrid combination of array and tree to achieve the maximum performance on searching without the requirement of elements being sorted

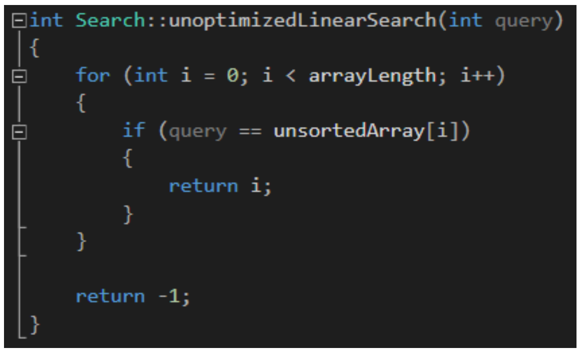
Our application has following features:

- Calculates time taken by five different algorithms
- Multiple scenarios to make sure that the benchmark resembles the real world performance instead of theoretical performance
- Calculates delta (time difference) between our newly designed algorithm and minimum of other algorithms for each scenario
- Calculates total delta and average delta to identify if our algorithm is faster or slower than existing algorithms
- As it is a 64 bit application, it runs natively on 64 bit windows operating system. So, there is no ***SysWoW64***[3] emulation going on under the hood and we are getting the raw performance results
- No compiler optimization was enabled to see how an algorithm actually performs

Methodology

Initially we implemented linear search in two different ways-

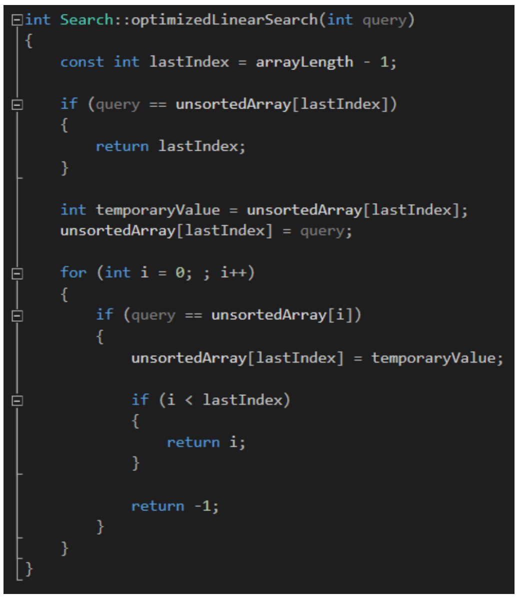
i. **Un-optimized Linear Search (ULS)**: Traditional approach



```
int Search::unoptimizedLinearSearch(int query)
{
    for (int i = 0; i < arrayLength; i++)
    {
        if (query == unsortedArray[i])
        {
            return i;
        }
    }

    return -1;
}
```

Figure 1: Code snippet ULS

ii. **Optimized Linear Search (OLS)**: This algorithm was taken from ***GeeksforGeeks***[4]. It requires less number of comparisons than traditional linear search.



```
int Search::optimizedLinearSearch(int query)
{
    const int lastIndex = arrayLength - 1;

    if (query == unsortedArray[lastIndex])
    {
        return lastIndex;
    }

    int temporaryValue = unsortedArray[lastIndex];
    unsortedArray[lastIndex] = query;

    for (int i = 0; ; i++)
    {
        if (query == unsortedArray[i])
        {
            unsortedArray[lastIndex] = temporaryValue;

            if (i < lastIndex)
            {
                return i;
            }

            return -1;
        }
    }
}
```

Figure 2: Code snippet OLS

Then we tried to figure out a way to improve the performance of the linear search with no additional space requirement(s). So, we introduced the concept of window, which is nothing but ¼th of array length (from the beginning of the array). The idea was that, whenever user searches for an element in the array, if that element is found and is outside the window, then we just swapped that element with the element that is contained in the window index and brought that element to the front of the array. So, we named this algorithm ***Bring Front***

Search (BFS). Theoretically, ***BFS*** should perform the same as linear search in worst case but on average, it should perform better. Then we conducted a benchmark among these three algorithms and checked which one performed faster. We had an interesting finding while conducting this benchmark. Though we did not expect any performance improvement on ***OLS*** over ***ULS***, but ***OLS*** significantly outperformed ***ULS***. And as expected, ***BFS*** significantly outperformed both ***OLS*** and ***ULS***.

Then we wanted to see how our ***BFS*** performs against hashing based searching ***UMS*** and tree based searching ***OMS***. For that we have used standard C++ ***unordered map***[5] and ***map***[6] and ran our benchmark. (Note: As ***UMS*** and ***OMS*** stores data in <key, value> pair, we inserted values of unsorted array as keys of the map and corresponding indices of unsorted array as value of the map). And unfortunately both ***UMS*** and ***OMS*** performed significantly better than ***BFS***.

Then we added another test in our benchmark to see how ***UMS*** and ***OMS*** performed while we want to get an element by its index number. Both the algorithms took a huge amount of time to retrieve the index. Whereas, array takes O(1) time to retrieve an element by index number.

Now we needed to design an algorithm that has the goodness of both ***array*** and ***binary search tree***[2]. To do so, we designed a hybrid container and we are calling that ***Dynamic Array (DA)***. Within ***DA***, we have implemented a slightly modified (modification needed to allow duplicate entries of elements) linked list based binary search tree[2] (self implementation) which is used to store the actual data of the array. In this way we can search for a particular element by value within O($\log_2$(n)) time. And for achieving the behavior of an array (searching by index in O(1) time), we used a dynamically allocable array of pointers which points to the actual element of the tree. This requires additional O(n) space to store the location/memory address of the actual element of the tree. Below is a graphical representation of our hybrid ***Dynamic Array (DA)***-

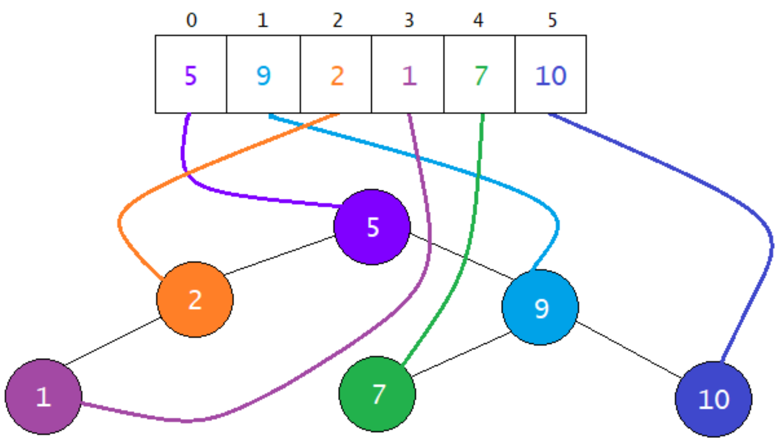


Figure 3: Graphical representation of DA

In Figure 3, we can see that, element '1' is physically located in the tree. And element '1' can be found in the 3rd index of the array. The array does not contain copy of the actual element '1' nor the actual element itself. Instead, the array contains a pointer to the actual element. After implementing this ***DA*** algorithm, we ran benchmark and compared with the existing algorithms and as expected, ***DA*** significantly outperformed all the above mentioned algorithms.

Benchmark and Comparison

Our benchmarking utility is designed in such a way that user can enter the array length. The utility then generates an array of specified length with random integers. Then it generates another array of queries which is half the length of the specified array length. There are six different scenarios in which the benchmarking is performed to get the approximate real world performance of the algorithms. The scenarios are-

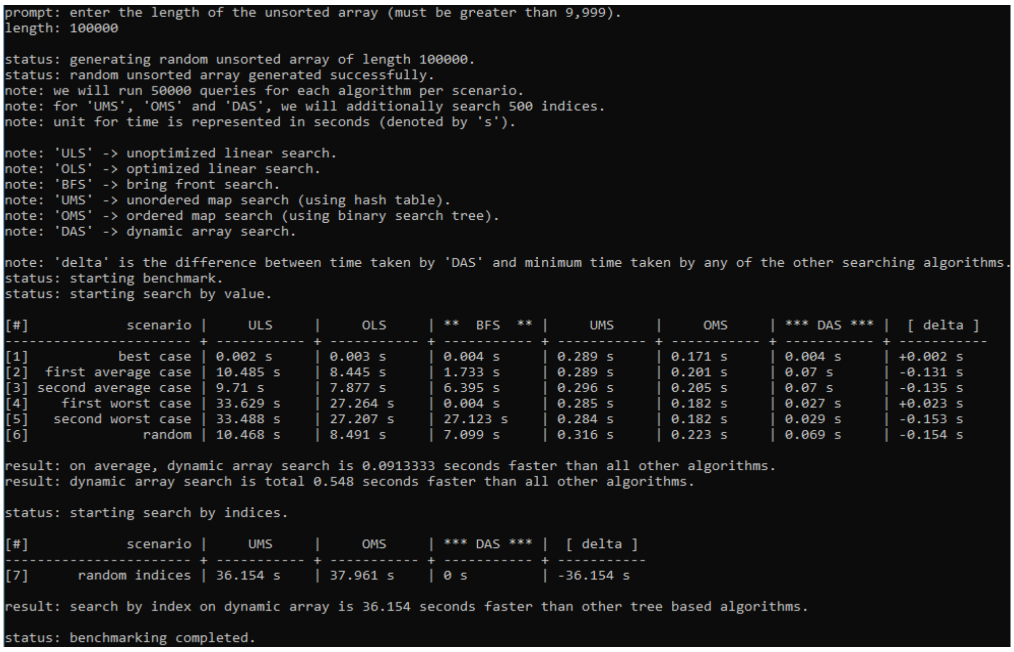
1) ***Best case scenario:*** In this scenario, all the queries are set to the value that is contained in the first index of the unsorted-array.
2) ***First average case scenario:*** In this scenario, (length of array / 100) number of queries is *repeatedly* set to the queries array starting from the last index of the array.
3) ***Second average case scenario:*** All the values from the last index of the array up-to (last index of <u>array</u> – last index of <u>queries</u>) are set to the queries array.
4) ***First worst case scenario:*** All the queries are set to the value that is contained in the second last index of the unsorted-array.
5) ***Second worst case scenario:*** All the queries are set to a value that is not present in the array. This way, we can find out how slow/fast one algorithm can finish searching the entire search space (when no element is found).
6) ***Random case scenario:*** In this case, queries are generated randomly between 1 and length of the array. This is the closest representation of how one algorithm should perform in real.

We have conducted our benchmark in the following way. We considered an array of length 100,000 for which our benchmarking tool ran 50,000 queries on each algorithms on six different scenarios and measured the time (in seconds) taken by each algorithm.

Our program was written in Visual Studio 2012 Ultimate and was compiled using Visual C++ compiler 2012 (supports C++11[7]) which generated 64 bit native executable. Benchmarks were performed in two different hardware and software configurations.

Configuration 1

- Processor: Intel Pentium N3520 Quad Core 2.16 GHz
- RAM: 4 GB DDR3
- Operating System: Windows 10

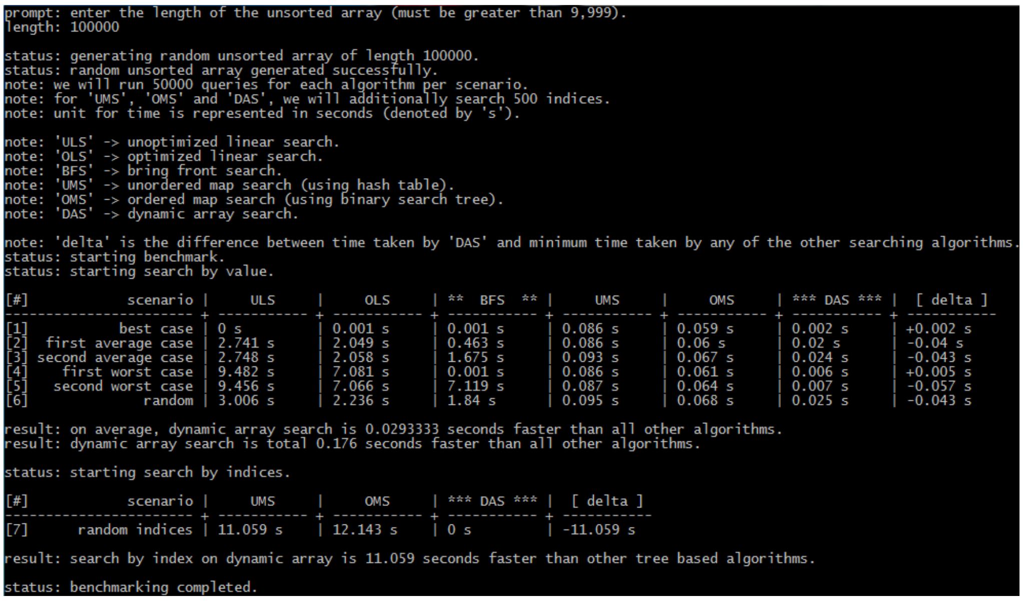


```
prompt: enter the length of the unsorted array (must be greater than 9,999).
length: 100000

status: generating random unsorted array of length 100000.
status: random unsorted array generated successfully.
note: we will run 50000 queries for each algorithm per scenario.
note: for 'UMS', 'OMS' and 'DAS', we will additionally search 500 indices.
note: unit for time is represented in seconds (denoted by 's').

note: 'ULS' -> unoptimized linear search.
note: 'OLS' -> optimized linear search.
note: 'BFS' -> bring front search.
note: 'UMS' -> unordered map search (using hash table).
note: 'OMS' -> ordered map search (using binary search tree).
note: 'DAS' -> dynamic array search.

note: 'delta' is the difference between time taken by 'DAS' and minimum time taken by any of the other searching algorithms.
status: starting benchmark.
status: starting search by value.

[#]           scenario |    ULS     |    OLS     | **  BFS  ** |    UMS     |    OMS     | *** DAS *** |  [ delta ]
---------------------- + ---------- + ---------- + ---------- + ---------- + ---------- + ---------- + ----------
[1]          best case | 0.002 s    | 0.003 s    | 0.004 s    | 0.289 s    | 0.171 s    | 0.004 s    | +0.002 s
[2]  first average case | 10.485 s   | 8.445 s    | 1.733 s    | 0.289 s    | 0.201 s    | 0.07 s     | -0.131 s
[3] second average case | 9.71 s     | 7.877 s    | 6.395 s    | 0.296 s    | 0.205 s    | 0.07 s     | -0.135 s
[4]    first worst case | 33.629 s   | 27.264 s   | 0.004 s    | 0.285 s    | 0.182 s    | 0.027 s    | +0.023 s
[5]   second worst case | 33.488 s   | 27.207 s   | 27.123 s   | 0.284 s    | 0.182 s    | 0.029 s    | -0.153 s
[6]             random | 10.468 s   | 8.491 s    | 7.099 s    | 0.316 s    | 0.223 s    | 0.069 s    | -0.154 s

result: on average, dynamic array search is 0.0913333 seconds faster than all other algorithms.
result: dynamic array search is total 0.548 seconds faster than all other algorithms.

status: starting search by indices.

[#]           scenario |    UMS     |    OMS     | *** DAS *** |  [ delta ]
---------------------- + ---------- + ---------- + ---------- + ----------
[7]     random indices | 36.154 s   | 37.961 s   | 0 s        | -36.154 s

result: search by index on dynamic array is 36.154 seconds faster than other tree based algorithms.

status: benchmarking completed.
```

Figure 4: Benchmark result on Configuration 1

Configuration 2

- Processor: Intel Core i7 7500U Dual Core 2.70 GHz (Quad Core with Hyperthreading)
- RAM: 8 GB DDR4
- Operating System: Windows 7



```
prompt: enter the length of the unsorted array (must be greater than 9,999).
length: 100000

status: generating random unsorted array of length 100000.
status: random unsorted array generated successfully.
note: we will run 50000 queries for each algorithm per scenario.
note: for 'UMS', 'OMS' and 'DAS', we will additionally search 500 indices.
note: unit for time is represented in seconds (denoted by 's').

note: 'ULS' -> unoptimized linear search.
note: 'OLS' -> optimized linear search.
note: 'BFS' -> bring front search.
note: 'UMS' -> unordered map search (using hash table).
note: 'OMS' -> ordered map search (using binary search tree).
note: 'DAS' -> dynamic array search.

note: 'delta' is the difference between time taken by 'DAS' and minimum time taken by any of the other searching algorithms.
status: starting benchmark.
status: starting search by value.

[#]           scenario |    ULS     |    OLS     | **  BFS  ** |    UMS     |    OMS     | *** DAS *** |  [ delta ]
---------------------- + ---------- + ---------- + ---------- + ---------- + ---------- + ---------- + ----------
[1]          best case | 0 s        | 0.001 s    | 0.001 s    | 0.086 s    | 0.059 s    | 0.002 s    | +0.002 s
[2]  first average case | 2.741 s    | 2.049 s    | 0.463 s    | 0.086 s    | 0.06 s     | 0.02 s     | -0.04 s
[3] second average case | 2.748 s    | 2.058 s    | 1.675 s    | 0.093 s    | 0.067 s    | 0.024 s    | -0.043 s
[4]    first worst case | 9.482 s    | 7.081 s    | 0.001 s    | 0.086 s    | 0.061 s    | 0.006 s    | +0.005 s
[5]   second worst case | 9.456 s    | 7.066 s    | 7.119 s    | 0.087 s    | 0.064 s    | 0.007 s    | -0.057 s
[6]             random | 3.006 s    | 2.236 s    | 1.84 s     | 0.095 s    | 0.068 s    | 0.025 s    | -0.043 s

result: on average, dynamic array search is 0.0293333 seconds faster than all other algorithms.
result: dynamic array search is total 0.176 seconds faster than all other algorithms.

status: starting search by indices.

[#]           scenario |    UMS     |    OMS     | *** DAS *** |  [ delta ]
---------------------- + ---------- + ---------- + ---------- + ----------
[7]     random indices | 11.059 s   | 12.143 s   | 0 s        | -11.059 s

result: search by index on dynamic array is 11.059 seconds faster than other tree based algorithms.

status: benchmarking completed.
```

Figure 5: Benchmark result on Configuration 2

Though the benchmark result window is self explanatory, **** BFS **** and ***** DAS ***** are self implementation (marked with 2 and 3 stars). **[delta]** is the difference between time taken by ***DAS*** and minimum of time taken by any other algorithms.

In configuration 1 (figure 4), we can see that, in *best case scenario* ***ULS*** is the fastest and our ***DAS*** is 0.002 seconds slower than ***ULS***. In *first and second average case scenario*, ***DAS*** is ~0.133 seconds faster than ***OMS*** (which is the second fastest). In *first worst case scenario* our ***BFS*** performs the best (takes only 0.004 seconds) and ***DAS*** is the second fastest (takes only 0.027 seconds) which is 0.023 seconds slower than ***BFS***. In *second worst case and random case scenarios*, ***DAS*** outperforms all the algorithms (approximately ~0.15 seconds faster). So, in case of search by value; on average, ***DAS*** algorithm is 0.09 seconds faster and, in total it is 0.548 seconds faster. In case of search by index; ***DAS*** is approximately ~36 seconds faster than both ***UMS*** and ***OMS***.

In configuration 2 (figure 5), we can see that, in *best case scenario* ***ULS*** is the fastest and our ***DAS*** is 0.002 seconds slower than ***ULS***. In *first and second average case scenario*, ***DAS*** is ~0.04 seconds faster than ***OMS*** (which is the second fastest). In *first worst case scenario* our ***BFS*** performs the best (takes only 0.001 seconds) and ***DAS*** is the second fastest (takes only 0.006 seconds) which is 0.005 seconds slower than ***BFS***. In *second worst case and random case scenarios*, ***DAS*** outperforms all the algorithms (approximately ~0.05 seconds faster). So, in case of search by value; on average, ***DAS*** algorithm is 0.029 seconds faster and, in total it is 0.176 seconds faster. In case of search by index; ***DAS*** is approximately ~11 seconds faster than both ***UMS*** and ***OMS***.

As we can see, in both low end (Pentium) and high end (Core i7) processors, ***BFS*** performed better than all other algorithms only in **First Worst Case Scenario** and outperformed both ***ULS*** and ***OLS*** in almost all the scenarios. ***DAS*** was significantly faster than all other algorithms in almost all scenarios.

Advantages

Our proposed algorithm, ***DAS*** has following advantages over any other existing algorithms-

- DAS has the goodness of both array and tree
- Searching by index can be performed in constant time just like an array
- Searching by value can be performed by using the same principle as *binary search*[1] even though the elements in DAS are not in sorted order (ascending or descending)

Disadvantages

Though our algorithm performs faster than any other existing algorithms, it has some disadvantages-

- Space complexity is O(n + n) because, the tree has space complexity of O(n) and the array of pointers (pointer to pointer) which points to the actual nodes of the tree requires O(n) space.
- Though it is a very rare case, we shall consider that fact that, if depth or height of the tree is very large, holding the entire tree in memory might be challenging.

Summary

We started this project with an aim of designing a faster searching algorithm for unsorted collection of elements. At first we designed ***BFS*** which performed relatively better than linear search but failed to prove its potential against ***UMS*** and ***OMS***. Theoretically, UMS can search for a value in constant time, O(1) which should be fastest algorithm for searching. But after performing benchmark on two different configurations (both hardware and software) we found that in reality, execution time of hash functions in relatively larger than tree traversal time in our proposed algorithm, ***DAS***; although our algorithm has time complexity O($\log_2$(n)). Searching by index in both ***UMS*** and ***OMS*** performs horrible (time complexity O(n)) whereas, ***DAS*** can search by index in constant time, O(1). It also overcomes the limitation of map searching (*no-duplicate-entry*).

In conclusion we can say that, our proposed algorithm ***Dynamic Array Search (DAS)*** has faster execution time than any other existing algorithms.

Future Works

In future, we would like to optimize the code even further in order to increase the performance significantly. We might also introduce multithreading in ***DAS*** to utilize multi-core processors. Compiler level optimization might also be very beneficial for performance increase.

References

[1] https://en.wikipedia.org/wiki/Binary_search_algorithm

[2] https://en.wikipedia.org/wiki/Binary_search_tree

[3] https://en.wikipedia.org/wiki/WoW64

[4] https://www.geeksforgeeks.org/search-element-unsorted-array-using-minimum-number-comparisons/

[5] https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/

[6] https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

[7] https://en.wikipedia.org/wiki/C%2B%2B11