Term Project Report

Mingkuan Pang, Harry Atulbhai Patel, Harshitha Vallabhaneni, Quan Hoang Nguyen Rashmitha Garige

*Matin Pirouz*

**Abstract**—In this report, we compared and analyzed the theoretical time complexity and experimental time complexity of eight sorting algorithms. Including insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort. Time complexities in this report include the best-case complexity, average-case complexity, and worst-case complexity for each of the sorting algorithms mentioned above. In the problem solving and analysis section, we solve the problem of finding whether, given a set *S of*  n integers, there exists two elements in *S* whose sum is exactly *x*. This problem is solved first in a brute force method where every possible pair of numbers is checked indiscriminately. Our goal is to solve the problem using a more efficient algorithm. To do this, we will be creating an array to store every integer we went through, then every time when we meet an integer we search if there exists another integer, we went through to make their sum to be exactly x.

1. **INTRODUCTION**

## Part 1: Comparison of Sorting Algorithms

The theoretical time complexity and experimental time complexity of sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heap sort, counting sort and radix sort) are compared. The theoretical time complexity (best, average, worst case) are recorded. Then, to get the time complexity of the experiment, different input array types and data sizes are passed to the sorting algorithm, which runs within the set time. The results then graphically show the temporal complexity of the experiment.

## Part 2: Problem Solving and Analysis

The problem that we solved was finding whether, given a set *S* of *n* of integers, if there exist two elements in *S* whose sum is integer *x*. To solve this problem using the brute force method, each value is compared to the other using a nested loop. One solution is to create an array A to store every integer that went through, and check whether x – i exists in the A, for the next integer i in the S, which will decrease the time complexity from

## Contributions

* + - compared the theoretical time complexities of multi- ple sorting algorithms to their experimental perfor- mance
    - determined what the best, average, and worst-case inputs are for all algorithms tested
    - implemented an efficient solution to solve whether there exist two elements whose sum is x in set S

## Comparison of Sorting Algotithms

## 2.1 Theoretical Time Complexities

**Insertion sort**, the best-case time complexity is *O*(*n*), it will be the input is already in order, therefore, for each element inserted, only the previous element needs to be examined.

For the general case, assuming that the input array is in the randomly order, the time complexity is the same as the worst case, which is

The worst case will be the input is totally reversed. In that case, the first element being considered when the second element is inserted, and the first two elements being considered when the third element is inserted…Then the time complexity on the worst case will be

**Selection sort**, because selection sort divides the array into two parts, one is ordered and the other is unordered, the algorithm selects elements from the unordered array and inserts them into the ordered array, so its time complexity is

For the three cases of the input, the time complexity is the same for the selection sort.

**Bubble sort**, the best-case time complexity is *O*(*n*), achievable with sorted input. One pass through the input size n, since is already sorted, no further traversal is needed.

The average-case and worst-case time complexity is *O*(*n*2), achievable with randomly sorted input and reverse sorted, In those two cases each element will compare with the rest of the input. Therefore, for those two cases, the time complexity will be

**Merge sort**, for merge sort, the time complexity will be

)

where is the algorithm divides the array into two parts, and is the time complexity for the algorithm to conquer the divided parts. It will not be affected by the order of the input array. Therefore, the time complexity for those three cases will be the same.

**Quicksort**, because quick sort is a recursion algorithm, therefore, the time performance of quicksort depends on the depth of the quicksort recursion. Best Case, on the best case, partition evenly divides the array into two parts every time. Therefore, if the size of input is n, then the time complexity will be

Average Case, on the average case, partition divides the array from somewhere in the middle, therefore, the time complexity will be the same as on the best case. Worst Case, on the worst case, the input is sorted or reversed, one of the divided parts is empty. on this case the time complexity will be

**Heap sort** is the continuous construction of the heap, after finish constructed the first heap, swapping head and tail, then adjust the heap to make it to be a max heap or min heap, then removing the head(tail) and reconstructing the heap with the rest part.The time complexity is

−

is the time complexity for the heap construction. is the time complexity for adjusting the heap. No matter how the order of the input is, the time complexity will stay the same.

**Counting sort**, the time complexity for counting sort is where the is the range of the input. For counting sort, no matter how the order of the input is, the time complexity will stay the same size.

**Radix sort**, the time complexity for counting sort is ) where the d is the number of passes and k is the radix used (In this project, it is 10). For radix sort, no matter how the order of the input is, the time complexity will stay the same.

1. **EXPERIMENT**

## 3.1 Experimental Setup and Data Generation

The execution time of the sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort) are run on a desktop computer with AMD Ryzen 7 5800H CPU. The code is written in C++ and the Chrono library is used to time the sorting algorithms. The code is compiled and run using VScode with GCC compiler (g++). An array of the same value, random array, and reverse sorted array is chosen as input types. These three array types account for the worst-case, average-case, and best-case inputs for the mentioned sorting algorithms. For data size, *n* = 100*,* 1000*,* 10000*,* 50000*,* 100000 is chosen. For each input type and data size, the sorting algorithms are run 10 times and the average is recorded in milliseconds.

## Experimental Data

From the result of the experiment, radix sort, counting sort, merge sort, heap sort, and quicksort seem to perform the best out of all the tested sorting algorithms considering from Fig.1-Fig.3. Median quicksort, in particular, performed the best out of the three versions of quicksort (first pivot, median pivot, and random pivot). Insertion sort, selection

sort, and bubble sort performed the worst considering their worst-case and average-case time complexity is *O*(*n*2). De- spite insertion sort’s best-case time complexity being *O*(*n*) in Fig1; in Fig.2 and Fig.3, the overall performance of inser-

tion sort is worst than radix sort, counting sort, merge sort, heap sort, and quicksort using median pivot.

## Experimental Data Analysis

**Insertion sort**, the experimental graph in Fig.4 conform with the asymptotic analysis as the experimental and theoretical time complexity is *O*(*n*) for the best-case complexity. The average case and worst case in Fig.5 and Fig.6 for the experimental graph conform with the asymptotic analysis as both experimental and theoretical time complexity is

*O*(*n*2). The experimental graphs in Fig.5 and Fig.6 show a significant increase in time as data size increase from

*n* = 1000 to *n* = 10000.

**Selection sort**, in Fig.4 - Fig.6. as the input increases to *n* = 10000, a sudden jump in the time taken by the sorting algorithm. As increase the input more than *n* = 10000, the time taken increased significantly again, which conforms with the asymptotic analysis of *O*(*n*2) for best, worst, and average-case time complexities.

**Bubble sort**, the experimental graph in Fig.4 did not match with the asymptotic analysis of *O*(*n*) for the best-case time complexity. The experimental graph showed *O*(*n*2),

this is due to the array that was used to test the time taken

for bubble sort. The array of the same element was used, as a result, the way bubble sort was written in this experiment will continue to traverse until *n* 1 times, unless it is a strictly sorted array, where the elements are not equivalent. The average and worst-case experimental results conformed

−

with the asymptotic analysis of *O*(*n*2) in Fig.5 - Fig.6. There are significant spikes in time as the data size increase more

than *n* = 1000.

**Merge sort**, the experimental graphs in Fig4-Fig6 match

with the asymptotic analysis as the best, average, and worst-case time complexity is *O*(*nlogn*). The experimental graphs spike in time minimally as the data size increased more than *n* = 1000, across best, average, worst case experimental graphs. Since merge sort break input in half

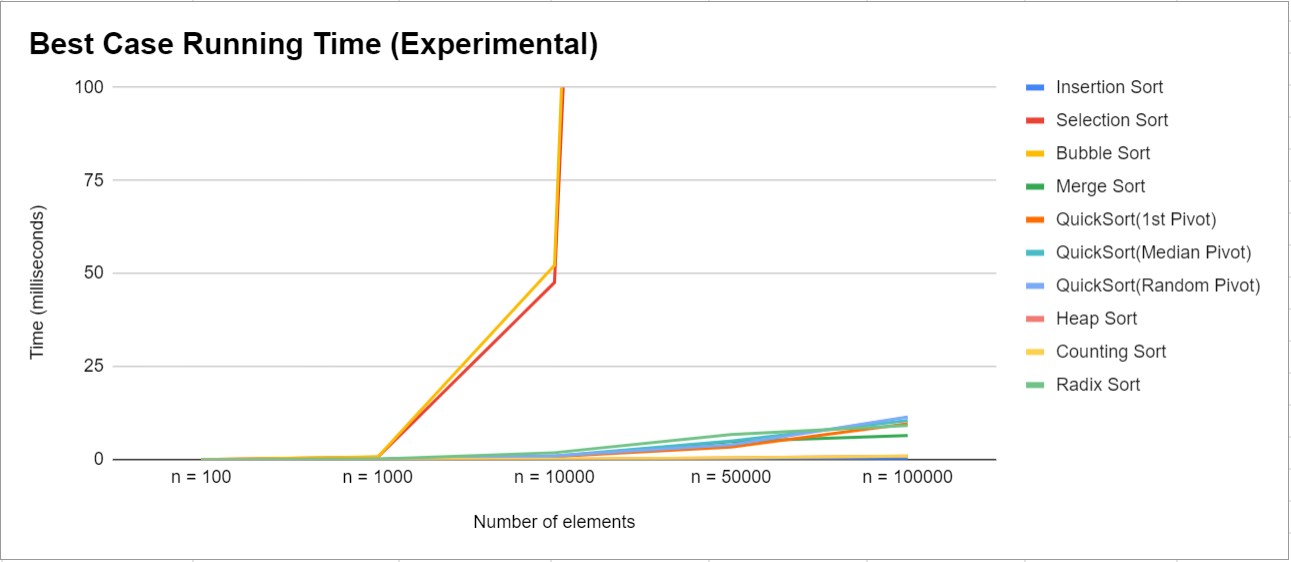


Fig. 4

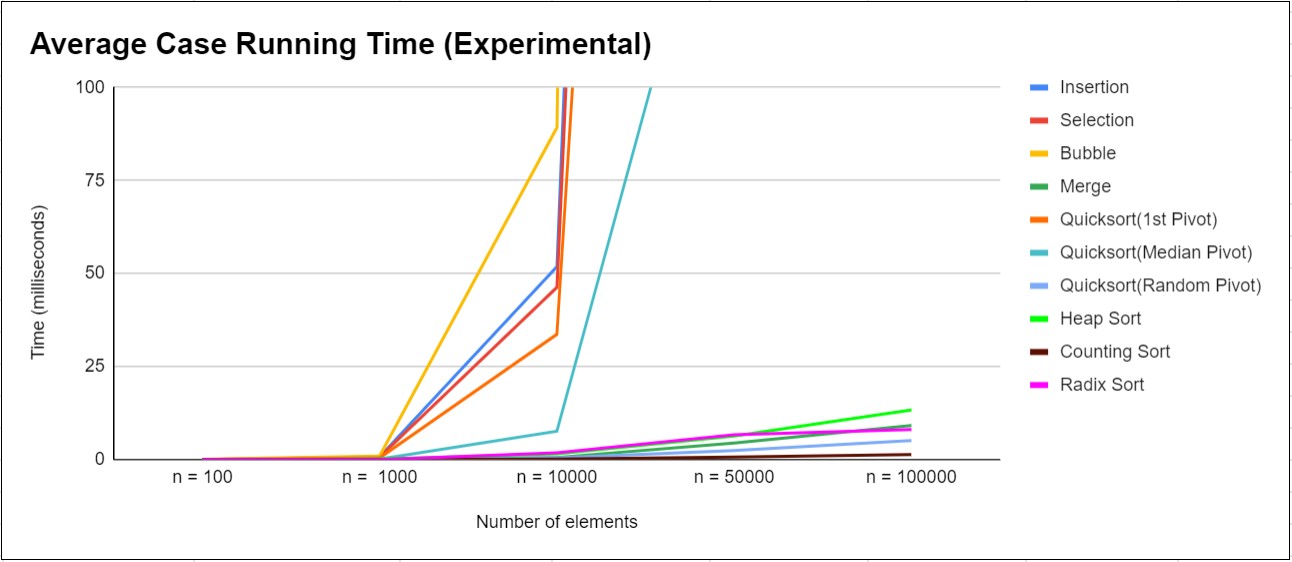


Fig. 5

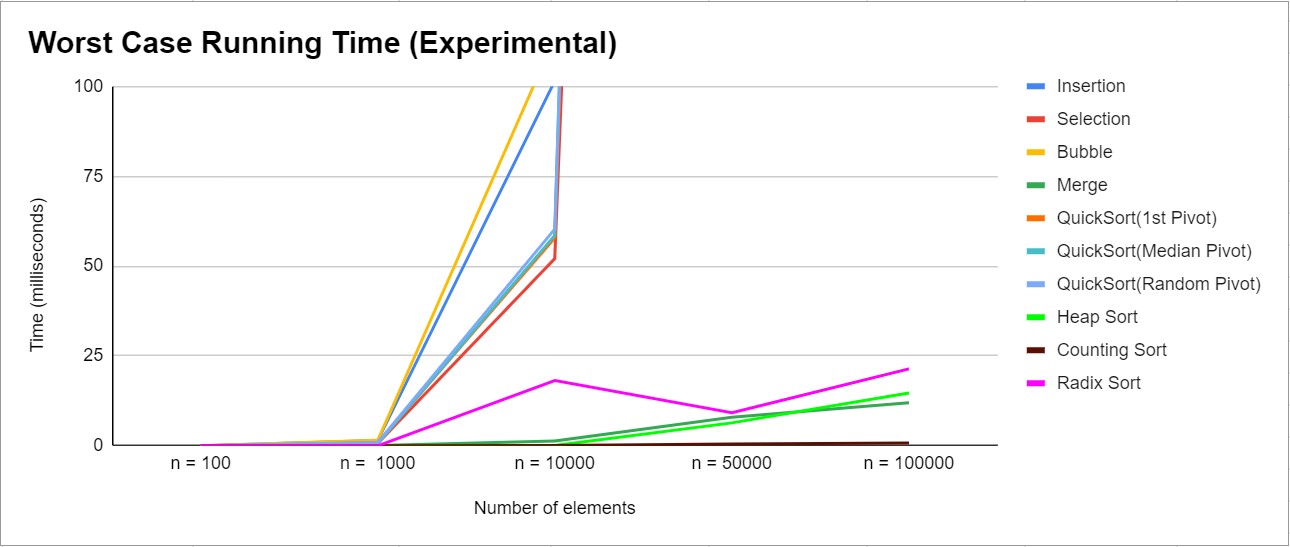


Fig. 6

recursively regardless of input, the experimental result conforms with the asymptotic analysis.

**Quicksort**, first element pivot, median pivot, and

random pivot have best-time complexity conforms with the asymptotic analysis of *O*(*nlogn*). In Fig.4, the time taken spikes minimally as the input increase more than *n* = 10000 and *n* = 50000. The experimental average time complexity in Fig.5 did not conform with the asymptotic

analysis of *O*(*nlogn*). The experiment result revealed *O*(*n*2)

average-case time complexity. Due to the pivot choice,

the input is partitioned unevenly by a significant margin, resulting in the average-case running time of *O*(*n*2). The pivot choice affects the chance of uneven partitions. The

experimental worst-case time complexity in Fig.6 conforms with asymptotic analysis of *O*(*n*2) as the reverse sorted input causes significant uneven partitions of the input. As

the input size increase more than *n* = 1000 and *n* = 10000, the time taken spiked significantly.

**Heap sort**, the experimental results in Fig.4 - Fig.6 best, average, worst-case conformed with the asymptotic analysis of *O*(*nlogn*) for all three cases. Heap sort disregard input type, build a heap and maintain heap after each swap *n* 1 times. As a result, the experimental result across three experiment graphs are conforming with the asymptotic analysis.

−

**Counting sort**, the experimental results in Fig.4 - Fig.6 for best, average, and worst-case time complexity conform with the asymptotic analysis of *O*(*n* + *r*); where *n* is the input size and *r* is the range of the input. The experimental results showed linear run time, which corresponds with the asymptotic analysis *O*(*n*) or *O*(*r*) depending on whether the size of the input is greater or the range of the input is greater.

**Radix sort**, the best-case experimental result in Fig.4 conforms with the asymptotic analysis of *O*(*d* (*n* + *k*)). The number of digits are equivalent for all the elements of an input, resulting in no extra counting sorts to be performed to accommodate for elements with extra digits. The average-case experimental result in Fig.5 conforms with the asymptotic analysis of *O*(*d* (*n* + *k*)). The difference in the number of digits across all elements is not significantly apart. However, since extra counting sort will be performed to accommodate the elements with more digits, the time taken is longer than the best case shown in Fig.4. The worst- case experimental does not conform with the asymptotic analysis of *O*(*d* (*n* + *k*)). The error might be that since the input for the worst case is a randomly sorted input, the digits for some run might have varied by a significant amount due to a randomly generated input, causing the graph shown in Fig.6.

∗

∗

∗

## Is the Number of Comparisons A Good Predictor of Execution Time?

Based on the experimental results, the number of compar- isons is a good predictor of execution time for the sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort). As

shown in Fig.2, the average case running time graph, *O*(*n*2)

sorting algorithms like insertion sort, selection sort, and

bubble sort takes longer to sort a input array compared to

*O*(*nlogn*) algorithms like merge sort, heap sort, and quick- sort. In Fig.3, the similar trend occurs, the *O*(*n*2) worst- case algorithms like quicksort, insertion sort, selection sort,

and bubble sort are slower than the *O*(*nlogn*) worst-case algorthms like merge sort and heap sort. The counting sort and radix sort algorithm does not apply in this case since they are not comparison based sorting algorithms.

1. **PROBLEM SOLVING AND ANALYSIS**

There are other factors to consider to implement the more efficient solution to the problem mentioned in [1.2:](#_bookmark0) what the size of the hash table should be, should certain values be omitted from insertion into the hash table, and how to eliminate duplicates. The size of the hash table should be enough to accommodate the *n* amount of integers in the set, be a prime number for the hash function, and be large enough to maintain around a 75% load factor [[2].](#_bookmark2) A simple way to determine what the size would be the next prime number after *n* 1*.*3. An early idea that we considered in reducing the amount of numbers that need to be searched was to remove numbers that are *< x* since two positive integers could not be the sum of *x* if one of them were larger than *x*. This however would not take into consideration negative numbers; in that case one of the values could be larger than *x* if the other were negative e.g. *x* = 8*, i* = 10*, j* = 2*, i* + *j* = *x*. Due to this, all numbers of the set should be included in the hash table. However, if a number is repeated multiple times, this would cause unnecessary searches. To combat this, the integers are stored in an object that contains the number and the frequency of the number in the hash table. This insures that each integer is only inserted once into the hash table. This approach is illustrated in Fig.7.

∗

−

**Algorithm 1:** Brute Force Algorithm

**1** Input: *A, n, x*

**2** Output: *true or false*

**3 for** *i* 0 *< n* **do**

←

**4** *num*1 = *A*[*i*]

**5 for** *j i* + 1 *< n* **do**

←

**6** *num*2 = *A*[*j*]

**7 if** *num*1 + *num*2 = *x* **then**

**8 return** *true*

**9 end**

**10 end**

**11 end**

**12 return** *false*

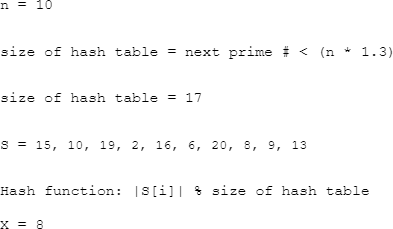
## Brute Force Algorithm and Analysis

For **Algorithm 1** the input is array *A* storing the values of set *S*, size of array *n*, and value *x*. *num*1 is initialized to *A*[*i*] and *num*2, in a nested loop, is initialized to *A*[*j*] where *j* = *i* + 1. The solution is found when *num*1 + *num*2 = *x* and the output is *true*. The output is *false* when the outer loop reaches it’s end condition.

The worst case for **Algorithm 1** would be if there are no values in *A* whose sum is *x*. In that case, for every value of *A*, the inner loop would execute n times which would check

**Algorithm 2:** Efficient Algorithm

**1** Input: *H, m, x*

**2** Output: *true or false*

**3 for** *i* 0 *< m* **do**

←

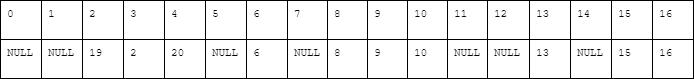
**4 if** *H*[*i*]*.frequency* ! = 0 **then**

**5** *y* = *x H*[*i*]*.n*

−

**6** *ind* = *hashkey*(*y, m*)

**7 while** *H*[*ind*]*.frequency* ! = 0 **do**

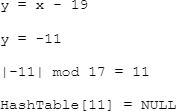
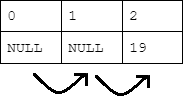


**8 if** *H*[*ind*]*.n* = *y* **then**

**9 if** *y* = (*x/*2) ***and***

*H*[*ind*]*.frequency <* 2 **then**

**10** + + *ind*



**11 break**

**12 end**

**13 return** *true*

**14 end**

**15 if** *ind >*= *m* 1 **then**

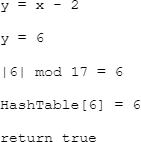
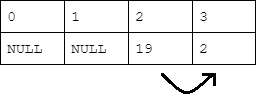
−

**16** *ind* = 0

**17 end**

**18 else**

**19** + + *ind*



**20 end**

**21 end**

**22 end**

**23 end**

**24 return** *false*

Fig. 7: The algorithm loops until an nonempty slot is found, calculates *y* and it’s hash key, and determines whether *y* is in the hash table or not. If yes, return true if not iterate until end of hash table.

the sub-array of *A*[*i..n*] for the solution. The total running time of this can be described as Θ(*n*2). The best case would be if the first two values of *A* have the sum *x*. In that case,

total running time would be Θ(1) since no loops would occur; *num*1 and *num*2 would both be initialized and their sum would be x and the return statement is triggered. In the average case, the solution would be found in the middle of A but an accurate representation of the time complexity is

still *O*(*n*2).

## Efficient Algorithm and Analysis

For **Algorithm 2** the input is array *H*, size of the array *m*, and value *x*. Array *H* is the hash table storing the elements of set *S*.The elements themselves are actually objects storing both the number and the frequency that they occur in *S*. If *n* is the amount of elements in set *S*, *m* is the first prime number greater than *n* 1*.*3 which is the amount of slots in the hash table. The algorithm iterates through every slot of *H*. If there is an nonempty slot (line 4) then *y* is initialized to *x* minus the number at *H*[*i*]. The variable *ind* contains the hash key for *y*. The hash function used in this case is *y* modulo *m*. The while loop at line 7 is used for linear probing to determine whether *y* is present in the hash table. If *y* is present in *H* the output is true. Otherwise, after probing is complete *i* is incremented and the next slot in *H* containing a number is checked. Once all the slots in *H* are checked that means that there were not two numbers present whose sum is *x* and the output is false. The if statement at line 9 is used to check in case the two integers whose sum is *x* are equivalent e.g. 3 + 3 = 6. In that case the frequency of *y* is checked to make sure that number is present in set *S* more than one time.

∗

The worst case for **Algorithm 2** would be if every integer in set S had the same hash key. This would mean that this algorithm would act similarly to **Algorithm 1**; the loop used for probing would instead execute *n* times and check the sub array *H*[*i..n*] to find if *y* is present in *H*. This instance would lead to a running time of Θ(*n*2). However,

unless specifically induced this is not likely to occur. If uniform hashing is assumed, the amount of probes in an unsuccessful search is constant and is at most 1*/*(1 *α*) where *α* is the load factor *n/m* [[3].](#_bookmark3) The load factor of *H* is at most .75 so the maximum amount of probes for an unsuccessful search would be 1*/*(1 *.*75) or 4 assuming uniform hashing. Since probing time complexity is constant *O*(4), the time complexity of the whole algorithm would be *O*(*m*), m being the size of the hash table.

−

−

# CONCLUSION

**ACKNOWLEDGMENTS**

We would like to express our special thanks to Professor Matin Pirouz as well our University who gave us this opportunity to do this project. We offer sincere appreciation for the learning opportunities provided by the university. Our completion of this project could not have been accom- plished without the support of each one our group mem- bers: Mingzan Liu, Leon Kantikov, Harsh Mohan Sason, Gustavo Lita-Chaves and Joel Ramirez-Mariscal. All of our dedication and determination helped us in the completion and submission of this project.

**REFERENCES**

1. Young, J. & Potler, A. First occurrence prime gaps. *Mathematics of Computation* **52**, 221–224 (1989). URL [http://www.jstor.org/stable/](http://www.jstor.org/stable/2008665) [2008665.](http://www.jstor.org/stable/2008665)
2. (2015). URL [https://docs.oracle.com/javase/6/docs/api/java/](https://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html) [util/HashMap.html.](https://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html)
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to algorithms* (MIT press, 2009).