Faculty of Engineering & Technology – Computer Science Department

Second Semester 2020 – 2021

Data Structures and Algorithms

COMP2421

Project 4: Sorting Algorithms

Name: Ahmaide Al-Awawdah

ID: 1190823

Section: 4

Instructor: Dr. Radi Jarrar

Date: 24th May 2021

Introduction

A Sorting Algorithms is an algorithm that is used to rearrange a list of data (array, linked list, array list,…etc) according to a comparison operator that desides the order of the elements.

The three types of sorting Algorithms that will be discusses here are:

1- Bogo Sort.

2- Odd-Even sort

3- Pancake sort.

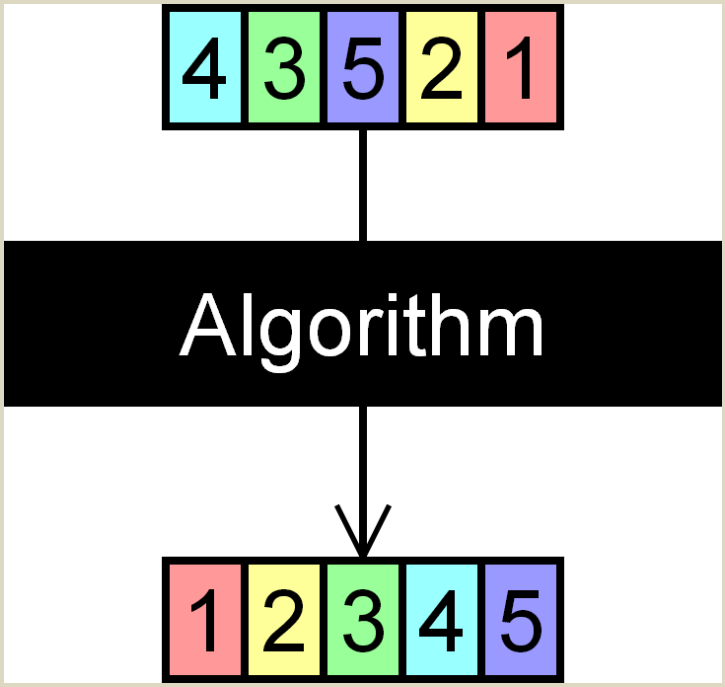

Fig 1

Table of Content

1. Bogo Sort

BogoSort is one of the dumpest methods to sort an array its also known as (slow sort, premutation sort, stupid sort) its ineffctive and its probably the worst way to sort elements, where it keeps on shuffling the order of the elements randomly until the are in the correct order, and each time it should check if the array was sorted or not. If the array had too mony elements to sort it will probably take forever since there will be a massive number of cases for the elements to be sorted as, the time to excude the code doesn't depend on anything since the elements are shuffled randomly each time, for example if the array had 9 elements the opertaion might go from one to thousands of trys with a need to check each time weather the elements were sorted or not, so this method is and never will be recommended for sorting array data.

How the code works

First the sorting function will need a boolean function to check weather the elements were sorted or not while the elements are still not sorted, each one will be shuffled with another random element of the array in each loop until they finally become in the right order.

So this method will need 2 functions, the first one will sort the data and they other one will check if the data is sorted or not by comparing each element with the one after it to check if any of the elements are not sorted and it will return a boolean value, after calling the sorting function for thr array it will go on a loop the will end until the checking function gives out that the array is sorted, in that time each element of the array will be randomly shifted with another element.

Run time and complexity

Each time of suffling the array the checking method is implemented which has a loop that compares all elements to the one after them so it will take O(n), each time the array is shuffled each element will be randomly swapped by onther element so the loop for that one will aslo take O(n), However the outer loop that will go on and on until the array is sorted doesn't depent on any value(doesn't have any upper bound), there is a chance that the array will be sorted from the first time which will take O(n) because the array will have to be

checked, the average case will be $O(n^n)$ however in the worst case scenario the time complexity will be $O(\infty)$ if the sorted array never shows up.

Space complexity: O(1), Is it stable: NO, Is it in place: No
Sorted ascending: O(n), Sorted decreasing: $O(\infty)$, Not sorted: $O(\infty)$

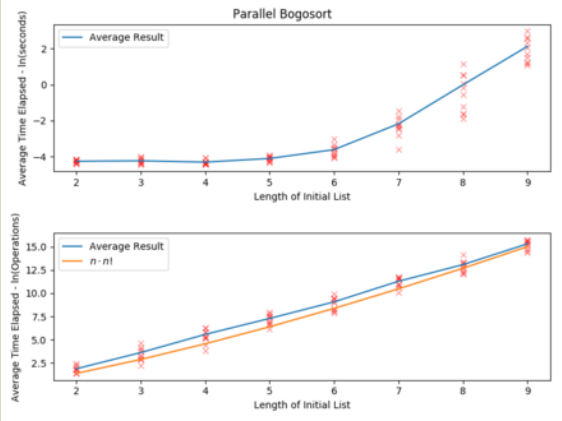

Fig 2

The Algorithem

```
method BogoSort (array A)
   while  not sorted (list)
      n  ← length of (A)
      for  i ← 0 to n do
         c ← random number % n
         swap A[i] ↔ A[c]
      end for
   end while
end method

method sorted (array A)
   while i ← 0 to length of(A) – 1
      if  A[i] > A[i+1] do
         return false
         end method
      end if
   end while
   return true
end method
```

2. Odd-Even Sort

Odd-Even / Even-Odd sort is a simple sorting algorithm, it is also known is as block sort, this method goes on two transposition sorts (Odd-Even then Even-Odd), this is a good, easy, effective to sort an array, where first it compares each odd orderd elements with its next even ordered elements and switch their order if they weren't in the correct order then it compares each even ordered element with its next odd ordered element until the array is 100% sorted in the needed order which will take the array size number of times in the worst case.

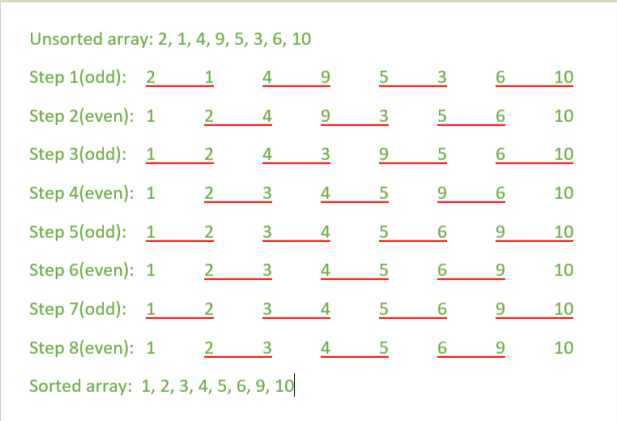

Fig 3

How the code works

First a boolean variable will be initialized in order to check weather the array was sorted or not, a while loop will keep on going until the array is sorted and the boolean gives a true value, in each loop at the beginning the boolean variable will be set to true and then there will be two for loops the first one will go on each odd element and compare it with its next element, if they werent sorted in the correct order then they will be swapped and the boolean variable will take a false value, the second for loop will go on each even element and compare it with its next elements and repeat the rest of the steps from the first for loop, so in each loop of the first while loop the boolean variable is set to true if it stays true after the two for loops then it will exit the while loop as the array is sorted.

Run time and complexity

The while loop that will keep on going until the array is sorted, its worst case will be when the last sorted element should be the first ordered or when the first ordered will be the last, in this case the loop will go on as the number of elements so its time complexity will be O(n), the inner two loops one will only go on the odd elemens so it will take half elements and the other one will go on the even elements which will also take half of the elements and together they will take all the elements which which will take O(n) and since they are inside another loop that will take O(n) if the array was already sorted which is the best case it will take a time of O(n), otherwise the worst case of number of times will be n^2 for them so the worst case time complexity will be $O(n^2)$ which is most likely to happen.

Space complexity: O(1), Is it Stable: Yes, Is it in place: No
Sorted ascending: O(n), Sorted decreasing: $O(n^2)$ Not sorted: $O(n^2)$

The Algorithm

```
method Odd-EvenSort(array A)
   boolean sorted ← false
   int n ← length of A
   while A not sorted
      sorted ← true
      while  i ← 0 to n-1
         if A[i] > A[i+1] do
            swap A[i] ↔ A[i+1]
            sorted ← false
         end if
         i = i x 2
      end while
      while i ← 1 to n-1
         if A[i] > A[i+1] do
            swap A[i] ↔ A[i+1]
            sorted ← false
         end if
         i = i x 2
      end while
   end while
end method
```

3. Pancake Sort

The pancake sorting method is one of the algorithms that place the maximum element at the end one by one, as it finds the maxium value and if it wasn't sorted in its spot then it reverses with all its previous elements as it becomes at the top then reverses it again with all the other unsorted elements after it so it becomes at its imposed location as the first element after all the unsorted smaller element with the larger elements that were sorted before after it until all elements are finally sorted.

How the code works

First there will be a variable that indicates the last location in which the largest unsorted element can be located at, where the required element will remain each time between it and the first variable, this variable will increase each time a new element is added to the sorted elements at the end of the array. A loop will start where the value will equal to the number of the elements of the array and it will end when it equals one, in each loop the largestvalue will be seached for, if it wasn't the last element of the unsorted elements then it will be reversed with all its previous elements so it becomes the first element then it will be reversed again with all the other unsorted element until it goes after all the unsorted elements, then the variable of the last location that the largest unsorted elemet will be decreased by one in order to find the next largest value and reverses it with the same two ways so it can be sorted from the end of the array and it keeps going on until the last location for the largest unsorted element is the first element, then the array will be sorted.

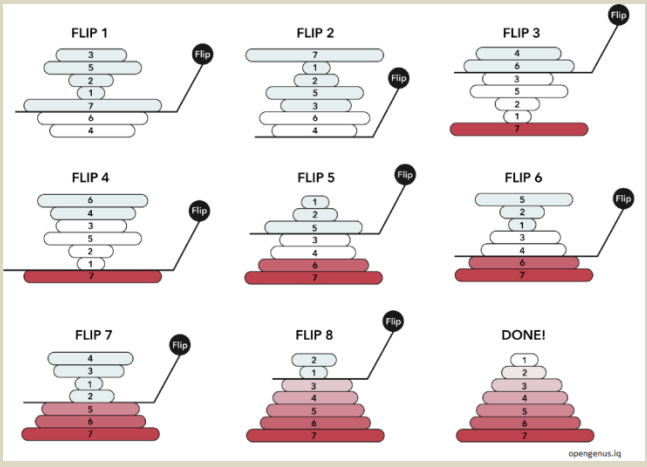

Fig – 4

Run time and complexity

Since this method required to look for the largest elemet of each period, as the number of periods equal to the number of elements in the array this will take a time complexity O(n), Finding the largest value before reversing it will mean searching the whole period between the first element and the last unsorted element in order to see which element has the largest value, this at worst case will seach the whole array in the first loop in order to find the the element that has the largest value in the whole array which will take O(n) this searching loop will be happening each time the loop for the periods happens so it will take $O(n^2)$, The two reversing loops are also inside the period loop, the first reversing loop will reverse the first half of elements with the other half and the biggest number of elements will be n-1 since it wont reverst if the largest element was last the worst case will be if at was before the last which will take a time that is equal to 0.5(n-1) = O(n), and for that the second reverse it will be on all the elements at the same case where it will reverse the first half with the second which will take 0.5(n) = O(n) the best case of it will not matter since the finding the max value will also be equal to the wosre case of the reverse loop and they are both in the same loop so it will take $O(n^2)$ for all cases.

Space complexity: O(log n), Is it Stable: No, Is it in place: No
Sorted ascending: $O(n^2)$, Sorted decreasing: $O(n^2)$, Not sorted: $O(n^2)$

The Algorithm

```
Method PancakeSort(array A)
  while  S ← length of (A) to 1
    MAX  ←  the order of max value in A until the element S
    If MAX+1 = S
      Reverse all elements in array A until MAX
      Reverse all elements in array A until (S-1)
    End if
    S = S-1
  End while
End method

Method to find the order of the MAX value(array A, until element E)
  Give p ← 0
  While  i ← 0  to  E
```

```
      If A[i] > A[p]
         p = i
      end if
      i = i+1
   end while
   return the value of p
end method

method to reverse array elements until a certain element (array A, until element E)
   L ← E
   i ← 0
   while  L<i
      swap the value of A[i] and A[L]
      i = i +1
      L = L – 1
   End while
End method
```

Summary

The table below (Table 1) compares between all three sorting algorithms.

The sorting algorithm	Bogo sort	Odd-Even Sort	Pancake Sort
Best case time complexity	O(n)	O(n)	$O(n^2)$
average case time complexity	$O(n^n)$	$O(n^2)$	$O(n^2)$
Worst case time complexity	O(∞)	$O(n^2)$	$O(n^2)$
Sorted ascending	O(1)	O(n)	$O(n^2)$
Sorted decreasing	O(∞)	$O(n^2)$	$O(n^2)$
Not sorted	O(∞)	$O(n^2)$	$O(n^2)$
Space complexity	O(1)	O(1)	O(Log n)
Is it stable	No	Yes	No
Is it in place	No	No	NO
Recommending using it	No	Yes	Yes

Table 1

References

Bogo sort

- Wikipedia
  https://en.wikipedia.org/wiki/Bogosort
- GeeksForGeeks
  https://www.geeksforgeeks.org/bogosort-permutation-sort/

Odd-Even sort

- Wikipedia
  https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort
- GeeksForGeeks
  https://www.geeksforgeeks.org/odd-even-sort-brick-sort/

Pancake sort

- Wikipedia
  https://en.wikipedia.org/wiki/Pancake_sorting
- GeeksForGeeks
  https://www.geeksforgeeks.org/pancake-sorting/
- LeetCode
  https://leetcode.com/problems/pancake-sorting/

Figures

- Figure 1
  https://www.growingwiththeweb.com/2014/06/sorting-algorithms.html
- Figure 2
  https://www.wikiwand.com/en/Bogosort
- Figure 3
  https://www.geeksforgeeks.org/odd-even-transposition-sort-brick-sort-using-pthreads/
- Figure 4
  https://iq.opengenus.org/pancake-sort/