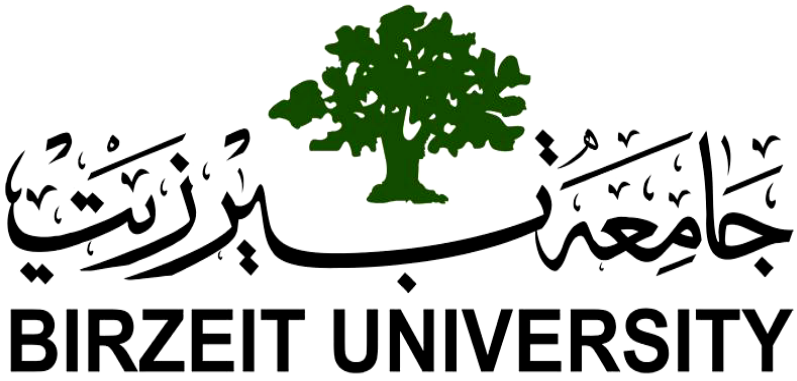

Faculty of Engineering & Technology
Electrical & Computer Engineering Department
COMP2421 (Fall2022/2023)
Project #4
Sorting

Prepared by:

Rivan Jaradat 1200081

Instructor: Dr. Ahmed Abusnaina

Section: 1

Date: 15/2/2022

1.Abstract:

The objective of this study is to distinguish and classify various sorting techniques and to ascertain the duration required for each to complete the sorting process.

Contents

2.Theory:

2.1. Gnome Sort:

It is an algorithm for sorting elements, called stupid sort. Its working principle is based on the iterative comparison between adjacent elements in the array and swapping them if they are in the wrong order, and it is a simple sorting algorithm that works by comparing the adjacent elements in the array and swapping them if they are in the wrong order. like bubble type. This type is distinguished by ensuring that the order of the elements is preserved during the sorting process. This technique can go back through the array when the switch is made, and this allows the current element to be compared with the previous one again.[1]

2.1.1.The algorithm :

1-Set the initial index i to 1

2-Compare the element at index i with the previous element (i-1)

3-If the element at index (i-1) is greater than or equal to the element at index i, swap the elements at index i-1 and i, and decrement i by 1

4-If the element at index (i-1) is less than the element at index i, increment i by 1

5-Repeat steps 2 to 4 until the entire array is sorted

2.1.2.the description:

This algorithm starts at index 1 in the array and compares the element at index i with the previous element (i-1). If the previous element is greater than the element at index i, the two elements are swapped and the value of i is decreased by 1. If the previous element is less than or equal to the element at index i, the value of i is incremented by 1. This process is repeated until all elements of the array are in order.

- Example:

  Let's use the Gnome Sort algorithm to sort the [4, 2, 1, 3] array:

  Set the initial index i to 1

  Compare the element at index i = 1 to the previous element (i-1), but since i = 1 there is no previous element to compare against yet.

  Increment i by one (i = 2) and compare the element in index 2 (2) and the previous element (4) and swap them (2, 4, 1, 3)

  Increment i by one (i = 3) and compare the element in index3 (1) and the previous element (4) and swap them (2,1,4,3)

  Increment i by one (i = 4) and compare the element in index4 (3) and the previous element (4) and swap them (2, 1, 3, 4)

  Set i to 1 and repeat the previous steps until you get an ordered array (1,2,3,4)

2.1.3. algorithm properties:

1-time complexity

Worst-case	Best-case	Average-case
O(n^2)	O(n)	O(n^2)

2-space complexity:

O(1)

In this sort, there is no need to reserve new memory, because it arranges the elements in the same places

3-The Gnome sort algorithm is stable

4- The Gnome sort sorting algorithm is in place sorting algorithm, because it sorts elements into the same places without allocating new memory

[2]

Note:

- If the array is arranged in ascending order the running time will be O(n) ,because it will pass once over the elements ,because they are ordered from smallest to largest.
- If the array is arranged in descending order, the runtime will be O(n^2), because it will rearrange it in ascending order and pass through the elements many times.
- If the array is not sorted, the runtime will be O(n^2), where the algorithm will need to pass multiple times over the elements to sort them.

  In general the run time will depend on the input data and the size of the array

2.2. Counting Sort:

Counting Sort is a sorting algorithm that does not use comparisons between elements to sort them. Instead, it sorts elements based on their frequency count. Counting Sort is often used as a subroutine in other sorting algorithms, such as Radix Sort. [3]

2.2.1.The Algorithm :

```
countingfunction(array):

max = maxValue(array)

arraySize = max + 1

countArray = new Array(arraySize)

sortedArray = new Array(array length) m:

for i = 0 to i < arraySize:

 countArray[i] = 0

for j = 0 to j < length of array:

    countArray[array[j]] += 1

for i = 1 to i < arraySize:

    countArray[i] += countArray[i-1]

for j = length of array - 1 to j >= 0:

    sortedArray[countArray[array[j]]-1] = array[j]

    countArray[array[j]] -= 1

return sortedArray
```

2.2.2. The description:
Here is an example thae explore the algorithm

Input array={1,4,5,2,2}

Max=5

Array size=5+1=6

count array

element	0	0	0	0	0	0
index	0	1	2	3	4	5

Store the count of each element at their respective index in count array

In our example the count of number 1 its 1 so we put 1 in index1

The count of number 2 it s so we put 2 in index 2 if we don't have number such as 3 we put 0 in their index

count array

element	0	1	2	0	1	1
index	0	1	2	3	4	5

Compute the prefix sum of the count array, which accumulates the counts of the previous elements to determine the index position of each element in the output array.

count array

element	0	1	3	3	4	5
index	0	1	2	3	4	5

use the last information to place each element in its correct position in the sorted array.

Input array={1,4,5,2,2}

count array

element	0	1	3	3	4	5

index	0	1	2	3	4	5

Go to index 1 in count array and get the element-1

1-1=0

0 the index of number 1 in the sorted array

Do that for all element then the output of soerted array will be

Sorted array={1,2,2,4,5}

2.3.3. algorithm properties:

1-time complexity

Worst-case	Best-case	Average-case
O(n + k)	O(n + k)	O(n + k)

2-Space complexity:

O(k)

3-stability:

Counting Sort is a stable sorting algorithm

4-In-place sorting:

The sort suffices only in the input array and does not need to reserve new memory, so it is an in-place sorting algorithm

- The running time of the algorithm if the input data array is:

1-Sorted (ascending): O(n+k). Since the input array is already sorted, there is no need to swap any elements during the cumulative count and sum operations.

2-Sorted (descending): O(n+k). this is because the algorithm will reverse the order of the input array, and then perform the sorting and merging operations, resulting in a slightly higher constant factor compared to the up state.

3-Not sorted: O(n+k). Because it will carry out all the steps until the matrix is arranged [4]

2.3. Tim Sort:

It is a sorting algorithm that combines merge sort and insertion sort. It works in many programming languages. It is used to sort arrays. It works by dividing the array into parts called Run, starting with sorting by insertion sort and then merge sort,The algorithm dynamically adjusts the size of runs depending on the size of the input matrix, making it effective for both small and large inputs.[5]

2.3.1.The algorithm

- Dividing an array into multiple blocks known as "runs"
- The size of each run is either 32 or 64
- The algorithm sorts each element within every run using insertion sort.
- The sorted runs are then merged together using the merge function of merge sort
- The size of the merged sub-arrays is doubled until the entire array is sorted.[6]

2.3.2.The description:

In this example, the concept of algorithm and how it works will become clear

Array input {3,1,4,2}

Divide the matrix into smaller parts [3, 1] and [4, 2]

Comparing the elements of each part separately using inclusion sort will become [1, 3] and [2, 4].

After that, merge sort is used by comparing the first element in each group, choosing the smallest of them, and placing it at the beginning of the sorted array

Tim Sort then compares the following items from the two phases (3 and 2). 2 is smaller, so it will be placed second in the sorted array.

The algorithm will continue this process, comparing the next elements for each run until all elements are combined into a single ordered array.

The final result will be {1,2,3,4 }

2.3.3. algorithm properties:

1-time complexity

Worst-case	Best-case	Average-case
O(n log n)	O(n)	O(n log n)

2-Space complexity:

O(n)
3-stability:

Tim Sort is a stable sorting algorithm

4-In-place sorting:
It requires extra memory for marge operation so its not in place algorithm

- the running time of the algorithms if the input data array is:
  1-Sorted (ascending): O(n) because the Tim sort can complete the sorting in a single pass by skipping the already sorted elements using the "galloping" technique.

  2-Sorted (descending): O(n log n) This is because the algorithm will reverse the order of the array and then sort and merge it

  3-Not sorted: O(n log n) This is because the algorithm first splits the array into smaller parts and sorts them using insertion sort, which takes time

2.4.cycle Sort:

Cyclic sorting is suitable for arrays that contain a large number of write operations in memory for sorting. Its principle of action is to divide the array into cycles, and to find the correct location for the element, but it is slow in arrays that contain a large number of elements.[7]

2.4.1.Algorithm :
cyclicSortfunction (arr):

```
    i = 0

    while i < length(arr):

        j = arr[i] - 1

        if j != i and arr[i] ! = arr[j]:

            swap(arr[i], arr[j])

        else

 i = i + 1

    return arr
```

[8]

2.4.2.the description:
Input array: [4, 3, 2, 1]

// Start by iterating over the array

for i in range(0, 3):

    // check if the value at that index is not equal to the index i + 1

    while array[i] != i+1:

// swap the value at the current index with the value at the index corresponding to its value.

temp = array[i]

array[i] = array[temp-1]

array[temp-1] = temp

// Continue iterating until all elements are in their correct positions.

Output array: [1, 2, 3, 4]

2.4.3. the algorithm properties:
1-time complexity

Worst-case	Best-case	Average-case
O(n log n)	O(n)	O(n log n)

2- Space complexity:

O(1)

3-stability:

cyclic Sort is a unstable sorting algorithm

4-In-place sorting:

The cyclic sort suffices only in the input array and does not need to reserve new memory, so it is an in-place sorting algorithm.

- the running time of the algorithms if the inputdata array is:
  1-Sorted (ascending): O(n) This is because cycle sort works by making the minimum number of writes to memory to sort an array, and an already sorted array requires only a few writes.
  2-Sorted (descending): O(n^2), This is because the cyclic sorting puts each element in its correct place, and in this case, at least, each element will be moved once in every cyclic

3-Not sorted: O(n^2), In this case, the sorting algorithm will fully implement its working principle, which is to divide the array into cycles and put each element in its correct position

2.5. Odd-even sort:

Odd-Even Sort Algorithm is a simple sorting algorithm that is used to sort a group of elements into small size arrays but it is less efficient than other sorting algorithms such as quicksort and merge for arrays containing a large number of elements, its working principle is to compare and swap pairs of adjacent elements in a loop until the array is sorted in its entirety. During each iteration of the loop, the algorithm processes odd and even even elements separately.

2.5.1.The algorithtm:

```
function oddEvenSort(array A)

sorted = false
 while not sorted
sorted = true
for i = 1 to n-1 step 2
 if A[i] > A[i+1]
swap A[i] and A[i+1]
sorted = false
 for i = 2 to n-1 step 2
 if A[i] > A[i+1]
 swap A[i] and A[i+1]
sorted = false
 return A
```

2.5.2.The describtion:

The odd-Even Sort function takes array A as input and starts with the sorting process. During each iteration of the loop, the function first processes the odd indexed elements of the array in the loop from 1 to n-1, and swaps adjacent elements if they are in the wrong order, i.e. the previous element is greater than the current element. The function then processes the even indexed elements of the array in a for loop from 2 to n-1, swapping back adjacent elements if they are in the wrong order. It continues in this way until it stops swapping

Example:

here's an example of using the odd-even sort algorithm to sort an array of 4 elements:

Starting array: [4, 2, 1, 3]

1st pass (odd indices):

Compare and swap 4 and 2 -> [2, 4, 1, 3]

Compare and swap 1 and 3 -> [2, 4, 1, 3]

2nd pass (even indices):

Compare and swap 2 and 4 -> [2, 4, 1, 3]

Compare and swap 1 and 3 -> [2, 4, 1, 3]

3rd pass (odd indices):

Compare and swap 2 and 1 -> [1, 4, 2, 3]

Compare and swap 4 and 3 -> [1, 3, 2, 4]

4th pass (even indices):

Compare and swap 1 and 3 -> [1, 3, 2, 4]

Compare and swap 2 and 4 -> [1, 3, 2, 4]

2.5.3. the algorithm properties:

1-time complexity

Worst-case	Best-case	Average-case
O(n^2)	O(n)	O(n^2)

2- Space complexity:

O(1)

3-stability:

Odd-Even Sort is a stable sorting algorithm

4-In-place sorting:

The sort suffices only in the input array and does not need to reserve new memory, so it is an in-place sorting algorithm.

- The running time of the algorithm if the input data array is:

1-Sorted (ascending): O(n). Because the algorithm basically works on arranging it in ascending order, so when the array is in ascending order, the algorithm will not need to do any swapping.

2-Sorted (descending): O(n^2). This is because the algorithm will need to compare and swap each adjacent element to sort the array and perform all steps of the algorithm, resulting in n^2 comparisons and swaps.

3-Not sorted: O(n^2). This is because the algorithm will need to compare and swap each adjacent element to sort the array, resulting in n^2 comparisons and swaps.

3.Summary :

Here is a summary the five sorting algorithms:

Tim Sort is best for large, unordered arrays. It has advantages such as a stable sort and the ability to handle nearly sorted arrays efficiently, but can have high memory usage.

Counting sort is best for small integer arrays with a known range. It runs in linear time, requires little additional memory, and is simple to implement, but can only be used for integers and may not be practical for large ranges.

Cyclic Sort is best for small, unordered arrays. It runs in O(n) time and is an in-place sort, but may not be practical for large arrays and can only be used for specific data types.

gnome Sort is best for large, unordered arrays. It has a fast average case time and is an in-place sort, but can have high worst-case time complexity.

Odd-Even Sort is best for small to medium-sized, unordered arrays. It can be used for parallel processing and has a stable sort, but has a high worst-case time complexity and may not be practical for large arrays. Ultimately, the best sorting algorithm depends on the specific data set being sorted and the performance requirements of the application.

4.conclousion

The purpose of this research project was to investigate and report on five new sorting algorithms. Throughout the project, we gained a comprehensive understanding of sorting algorithms, including their properties and how to analyze their running time based on different input data scenarios. We also learned how to effectively research and report on scientific topics. Overall, this project allowed us to develop valuable knowledge and skills in the field of sorting algorithms and research, which will be useful in our academic and professional pursuits.

5. References

[1]- https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/

[2]- https://en.wikipedia.org/wiki/Gnome_sort

[3]- https://en.wikipedia.org/wiki/Counting_sort

[4] https://www.geeksforgeeks.org/counting-sort/

[5]- https://en.wikipedia.org/wiki/Timsort

[6]- https://www.javatpoint.com/tim-sort

[7]- https://www.geeksforgeeks.org/cycle-sort/

[8]- https://www.javatpoint.com/cycle-sort

[9]- https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort

[10]- https://www.geeksforgeeks.org/odd-even-sort-brick-sort/