Mini project report

Sorting Algorithms

MODULE: ADVANCED ALGORITHMS AND COMPLEXITY

MEMBER:

- MESGUI ALAE
- NAIT CHERIF SABRINEL

WORK REQUESTED BY: B.DELLA-HEDJAZI

2024-2025

Table of Contents

1) Bubble Sort

Bubble Sort is one of the simplest and most well-known algorithms in computer science for sorting an array of elements.

Algorithm principle:

- Traversal: Traverse the array from left to right, comparing each element with its immediate neighbor.
- Comparison and Swapping: If an element is greater than the next one, they are swapped, causing the largest element to progressively "bubble" to the end of the array.
- Repetition: After each pass, the last element is sorted. The process is repeated for the remaining unsorted elements.
- Stopping Condition: The algorithm stops when no swaps are made during a pass, indicating that the array is fully sorted.

Optimization:

- After each pass, the unsorted section decreases by one element.
- The algorithm can terminate early if no swaps are made, meaning the array is already sorted.

C program of BubbleSort procedure:

```c
void BubbleSort(int A[], int n) {
    bool Change = true;
    while (Change == true) {
        Change = false;
        for (int i = 0; i < n - 1; i++) {
            if (A[i] > A[i + 1]) {
                int temp = A[i];
                A[i] = A[i + 1];
                A[i + 1] = temp;
                Change = true;
            }
        }
    }
}
```

C program of BubbleSortOpt procedure:

```
void BubbleSortOpt(int A[], int n) {
    int m = n - 1;
    bool Change = true;
    while (Change == true) {
        Change = false;
        for (int i = 0; i < m; i++) {
            if (A[i] > A[i + 1]) {
                int temp = A[i];
                A[i] = A[i + 1];
                A[i + 1] = temp;
                Change = true;
            }
        }
        m--;
    }
}
```

Complexity of BubbleSort and BubbleSortOpt in the best and worst cases:

Best case:

If the array is already sorted, only one pass is needed to confirm no swaps are required.
The while loop executes only once because the value of change will be 0 and therefore we only do $n - 1$ iteration
$T(n) = n - 1$
Time Complexity**:** $T(n) = O(n)$

Worst case:

- **BubbleSort**

The while loop executes n times.
The inner loop runs $n - 1$ iterations in the first pass, $n - 2$ in the second, and so on.
Total comparisons: $n(n - 1)$
Time Complexity: $T(n) = O(n^2)$

- **BubbleSortOpt**

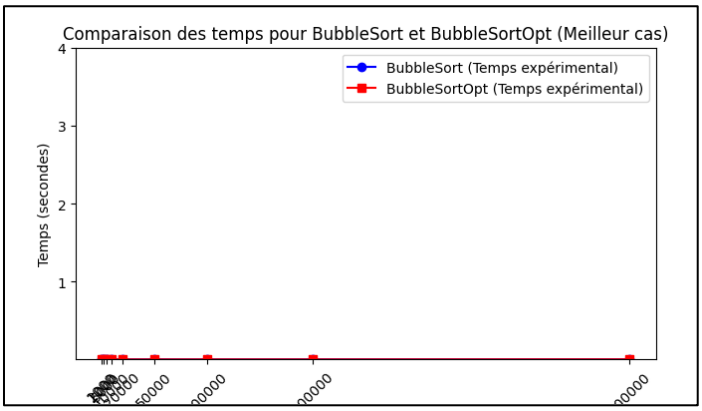
The while loop executes n times, but the number of iterations in the for loop decreases with each pass.
Total comparisons: $\frac{n(n-1)}{2}$
Time Complexity: $T(n) = O(n^2)$

Comparison of experimental complexity:

$T_1(n)$: BubbleSort $T_2(n)$: BubbleSortOpt

Best case :

Taille	1000	2000	5000	10000	20000	50000	100000	200000	500000
$T_1(n)$	0.000	0.000	0.000	0.001	0.000	0.001	0.000	0.001	0.001
$T_2(n)$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001

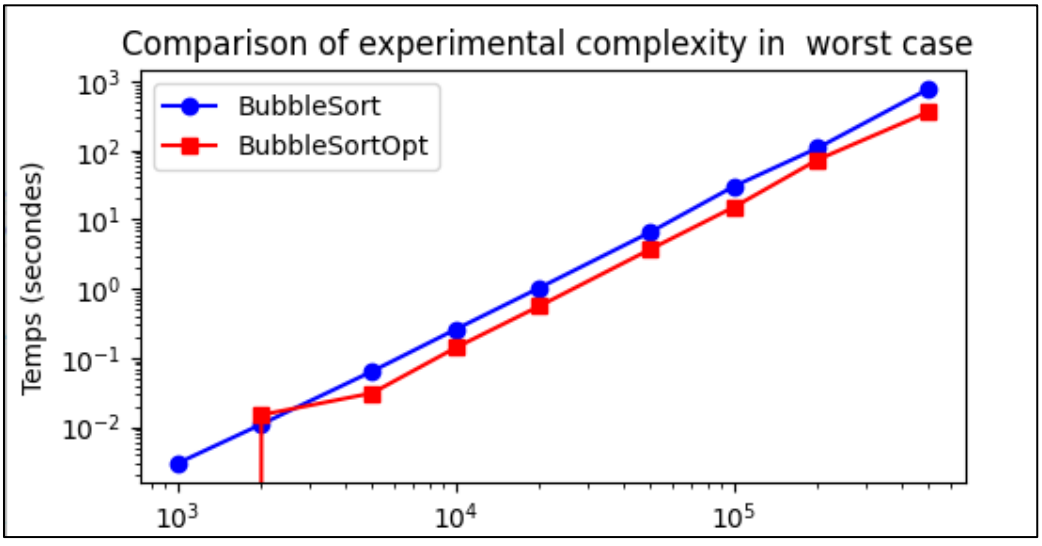


Both algorithms perform similarly with minimal time since the input is already sorted.

Worst case:

Taille	1000	2000	5000	10000	20000	50000	100000	200000	500000
$T_1(n)$	0.003	0.011	0.064	0.260	1.036	6.593	30.405	109.164	777.607
$T_2(n)$	0.000	0.015	0.031	0.141	0.564	3.731	15.235	73.826	362.124

The optimized version reduces the number of unnecessary comparisons by avoiding already sorted elements, BubbleSortOpt is consistently faster than BubbleSort



Conclusion
Bubble Sort and its optimized version (BubbleSortOpt) share the same best-case $O(n)$ and worst-case $O(n^2)$ time complexities. However, BubbleSortOpt demonstrates significant performance improvements by reducing unnecessary comparisons. This reduction in the number of comparisons leads to better performance, especially as the array size increases.

2) Gnome Sort

Algorithm Steps:

1. Start from the beginning of the array (index = 0).
2. Compare the current element with the next one (arr[index] and arr[index+1]).
3. If they are in order (arr[index] >= arr[index+1]), move forward (index++).
4. If they are out of order, swap them and move backward (index--).
5. If we reach the beginning of the array again, move forward (index++)
6. Repeat until the end of the array is reached.

Algorithm:

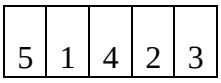
```
GNOME_SORT (A, n)
Set index = 0
While index < n :
    If index == 0 or A[index] >= A[index-1]:
         Move forward (index = index + 1)
    Else:
         Swap A[index] with A[index-1]
         Move backward (index = index - 1)
End
```

C algorithm:

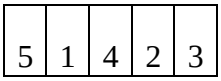
```c
void gnomeSort(int arr[], int n) {
    int index = 0;
   while (index < n) {
        if (index == 0 || arr[index] >= arr[index - 1]) {
            index++;
        } else {
            int temp = arr[index];
            arr[index] = arr[index - 1];
            arr[index - 1] = temp;
            index--;
        }
    }
}
```

Example of the algorithm's execution

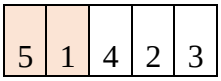
Initial Array



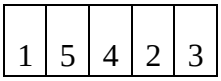
First Iteration: i=0, we move forward: i=1



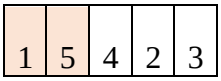
Compare 1 with 5: Swap, because 5 > 1, i=0



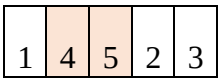
i=0 so we move forward: i=1



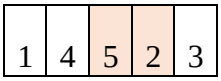
Compare 5 with 1, they are in order so i=2



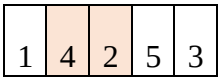
Compare 5 with 4: Swap, because 5 > 4, i=3



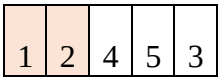
Compare 5 and 2: Swap, because 5 > 2, i=2



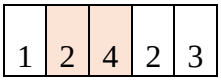
Compare 2 with 4: Swap, because 5 > 4, i=1



Compare 2 with 1: they are in order so i=2

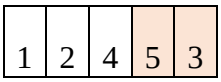


Compare 4 with 2: they are in order so i=3

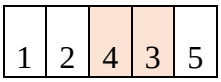


Compare 5 with 4: they are in order so, i=4

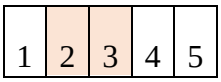
Compare 5 and 3: Swap, because 5 > 3, i=3



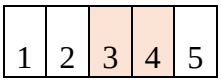
Compare 3 with 4: Swap, because 4 > 3, i=2



Compare 3 with 2: they are in order so i=3

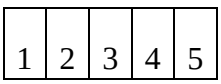


Compare 4 with 3: they are in order so, i=4



Compare 5 with 4: they are in order so, i=5, we reached the end

Final Sorted Array



Theoretical Complexity Analysis:

- **Best Case (O(n))**:

If the array is already sorted, the algorithm makes a single pass without any swaps.

- **Worst Case ($O(n^2)$)**:

If the array is in reverse order, multiple swaps are required, leading to a quadratic number of comparisons and swaps.

- **Average Case ($O(n^2)$)**:

Randomly shuffled arrays tend to behave similarly to the worst case for Gnome Sort.

Time Complexity Calculation:

The time complexity of Gnome Sort is determined by its single loop, which moves forward when elements are in order and backward when they are not. In the **best case** (already sorted array), the algorithm performs $\boldsymbol{n} - 1$ comparisons, resulting in $\boldsymbol{O}(\boldsymbol{n})$ time complexity.

In the **worst case**, the algorithm may repeatedly backtrack and re-check elements, requiring approximately $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$ comparisons and swaps, leading to $O(n^2)$ time complexity.

The **average case** is also **$O(n^2)$**, as the algorithm mixes forward and backward movements due to random order.

Comparison between the best and the worst case:

Table size (n)	10000	20000	50000	80000	100000
Best case	0.000	0.000	0.000	0.001	0.001
Worst case	0.573	2.282	16.351	38.645	62.905

3) Radix Sort

Radix Sort is a sorting algorithm that organizes elements based on individual digits or characters, processing them position by position.

Algorithm

The key function

```
 function key(I/x, i: integer): integer
var
    a, b, k: integer;
start
   b=1;
   for k from 0 to i-1 do :
        b ← b * 10

    a ← b * 10
    return (x % a) / b
end
```

The SortAux function

```
function SortAux(T: array of integers, n: integer, i: integer)
var
    max, tmp: integer
    Ttmp, Ttrie: array of integers
start
    max = 0
    for j = 0 to n - 1 do
        tmp = key(T[j], i)
        if tmp > max then
            max = tmp
        endif
    endfor

    for j = 0 to n - 1 do
        tmp = key(T[j], i)
        Ttmp[tmp] = Ttmp[tmp] + 1
    endfor

    for j = 1 to max do
        Ttmp[j] = Ttmp[j] + Ttmp[j - 1]
    endfor
```

```
    Ttrie = array[n] of 0
    for j = n - 1 downto 0 do
        tmp = key(T[j], i)
        Ttmp[tmp] = Ttmp[tmp] - 1
        Ttrie[Ttmp[tmp]] = T[j]
    endfor

    for j = 0 to n - 1 do
        T[j] = Ttrie[j]
    endfor
end
```

The RadixSort function

```
function RadixSort(T: array of integers, n: integer, k: integer)
start
    for i=0 to k-1 do
        SortAux(T, n, i)
    endfor
end
```

C algorithm:

```
int key(int x, int i) {
    int b = 1;
    for (int k = 0; k < i; k++) {
        b *= 10;
    }
    int a = b * 10;
    return (x % a) / b;
}
void SortAux(int* T, int n, int i) {
    int* Ttmp = (int*)calloc(10, sizeof(int));
    for (int j = 0; j < n; j++) {
        int digit = key(T[j], i);
        Ttmp[digit]++;
    }
    for (int j = 1; j < 10; j++) {
        Ttmp[j] += Ttmp[j - 1];
    }

    int* Ttrie = (int*)malloc(n * sizeof(int));
    for (int j = n - 1; j >= 0; j--) {
        int digit = key(T[j], i);
        Ttrie[--Ttmp[digit]] = T[j];
```

```
    }

    for (int j = 0; j < n; j++) {
        T[j] = Ttrie[j];
    }

    printf("Array after sorting by digit %d: ", i + 1);
    for (int j = 0; j < n; j++) {
        printf("%d ", T[j]);
    }
    printf("\n");

    free(Ttmp);
    free(Ttrie);
}

int findMaxDigits(int* T, int n) {
    int maxVal = 0;
    for (int i = 0; i < n; i++) {
        if (T[i] > maxVal) {
            maxVal = T[i];
        }
    }
    int digits = 0;
    while (maxVal > 0) {
        maxVal /= 10;
        digits++;
    }
    return digits;
}

void RadixSort(int* T, int n, int k) {
    for (int i = 0; i < k; i++) {
        SortAux(T, n, i);
    }
}
```

Complexity :

Key Function

The for loop in key executes i+1 times
Since i is small, the complexity can be considered as O(1)

SortAux function

- First Loop**:**Finds the maximum value of the digits at position I, executes n times, so O(n).
- Second Loop**:**Creates and fills the occurrence table Ttmp,executes n times, so O(n).
- Third Loop: Calculates the cumulative positions in Ttmp,Loop size is max+1, where maxis the maximum possible value of a digit (typically 9 for decimal integers).Thus, O(1)
- Fourth Loop**:**Fills the sorted table Ttrie,executes n times, so O(n).
- Fifth Loop**:**Copies Ttrie into T,executes n times, so O(n).

Total complexity of SortAux: $\boldsymbol{O(n) + O(n) + O(1) + O(n) + O(n) = O(n)}$

Fonction RadixSort

SortAux call:Each SortAux call has a complexity of O(n).
Number of calls to SortAux k times.
Total complexity of RadixSort:$O(k * n)$

Example:

```
                                        RadixSort
Array before sorting: 3024 170 45 75 90 1802 2 66
Number of digits (k): 4
Array after sorting by digit 1: 170 90 1802 2 3024 45 75 66
Array after sorting by digit 2: 1802 2 3024 45 66 170 75 90
Array after sorting by digit 3: 2 3024 45 66 75 90 170 1802
Array after sorting by digit 4: 2 45 66 75 90 170 1802 3024
Time taken to sort the array: 0.003000 seconds
Sorted array: 2 45 66 75 90 170 1802 3024
```

4) Quick Sort

L'algorithme Quicksort repose sur le principe de diviser pour régner : à chaque étape, il choisit un pivot, partitionne le tableau en deux parties (éléments inférieurs et supérieurs au pivot), puis applique récursivement le tri sur ces deux parties.

C program of partition procedure:

```c
int partition(int tab[], int d, int f) {
    int eltPivot = tab[d];
    int i = d + 1;
    int j = f;
    int x;
    while (1) {
        while (i <= f && tab[i] <= eltPivot) {
            i++;
        }
        while (tab[j] > eltPivot && j >= d) {
            j--;
```

```
        }
        if (i >= j) {
            break; }
        x = tab[i];
        tab[i] = tab[j];
        tab[j] = x;  }
    tab[d] = tab[j];
    tab[j] = eltPivot;
    return j;
```

C program of quicksort procedure:

```
void quickSort(int tab[], int p, int r, int (*partitionFunc)(int[], int, int)) {
    if (p < r) {
        int q = partitionFunc(tab, p, r);
        quickSort(tab, p, q - 1, partitionFunc);
        quickSort(tab, q + 1, r, partitionFunc);
    }
}
```

Complexity:

The complexity depends on the choice of the pivot, the complexity is between $O(n\ log\ n)$ and $O(n^2)$.

Best case:

The pivot chosen divides the array into two equal halves in every recursive step. For example, selecting the middle element as the pivot ensures the array is always split evenly.

C program of partition procedure in the best case:

```
int partitionBestCase(int tab[], int d, int f) {
    int mid = (d + f) / 2;
    int eltPivot = tab[mid];
    int i = d;
    int j = f;
    int x;
    tab[mid] = tab[d];
    tab[d] = eltPivot;
    while (1) {
        while (i <= f && tab[i] <= eltPivot) {
            i++;
```

```
        }
        while (tab[j] > eltPivot && j >= d) {
            j--;
        }
        if (i >= j) {
            break;
        }
        x = tab[i];
        tab[i] = tab[j];
        tab[j] = x;
    }
    tab[d] = tab[j];
    tab[j] = eltPivot;
    return j;
}
```

The algorithm makes two recursive calls on subarrays of size $\frac{n}{2}$. These calls will continue recursively until the array is divided down to base cases of size 1.

This gives the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Where: $T(n)$ represents the time complexity for sorting an array of size n.

$2T\left(\frac{n}{2}\right)$ represents the two recursive calls on subarrays of size n/2.

$O(n)$ represents the time taken to partition the array.

Prove the recurrence relation:

$T(0) = T(1) = 0$

$T(n) = 2T\left(\frac{n}{2}\right) + n$

Development of the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^1 T\left(\frac{n}{2^1}\right) + 1.n$$

$$= 2[2T\left(\frac{\frac{n}{2}}{2}\right) + \frac{n}{2}] + n = 4T(n/4) + 2n = 2^2 T\left(\frac{n}{2^2}\right) + 2.n$$

$$= 4\left[2T\left(\frac{\frac{n}{4}}{2}\right) + \frac{n}{4}\right] + 2n = 8T(n/8) + 3n = 2^3T\left(\frac{n}{2^3}\right) + 3.n$$

…

$$T(n) = 2^kT\left(\frac{n}{2^k}\right) + k.n$$

$$\frac{2^k}{n} = 1 \qquad 2^k = n$$

$$\log 2^k = \log n$$

$$k\ log_2 = \log n$$

$$k = log_2\, n$$

$$T(n) = n\,T\left(\frac{n}{n}\right) + n.log_2\, n$$

$$= n * 1 + nlog_2 n$$

$$= O(nlog_2\, n)$$

We have shown that the time complexity of QuickSort in the best case is $O(nlogn)$

Worst case:

The pivot chosen is always the smallest or largest element in the array, leading to highly unbalanced partitions. For example, if the last element is chosen as the pivot in a sorted or reverse-sorted array, the result will be one partition with size $n - 1$ and another with size 0.

```
int partitionWorstCase(int tab[], int d, int f) {
    int eltPivot = tab[f];
    int i = d - 1;
    int j;
    for (j = d; j < f; j++) {
        if (tab[j] <= eltPivot) {
            i++;
            int temp = tab[i];
            tab[i] = tab[j];
            tab[j] = temp;
        }
    }
    int temp = tab[i + 1];
    tab[i + 1] = tab[f];
```

```
    tab[f] = temp;
    return i + 1;
}
```

The recurrence relation:

T(0)=T(1)=0(base case)
T(n)=n+T(n−1)

Development of the recurrence relation:

$T(n) = T(n-1) + n$

$T(n-1) = T(n-2) + n - 1$

$T(n-2) = T(n-3) + n - 2$

…. $T(1) = 0$

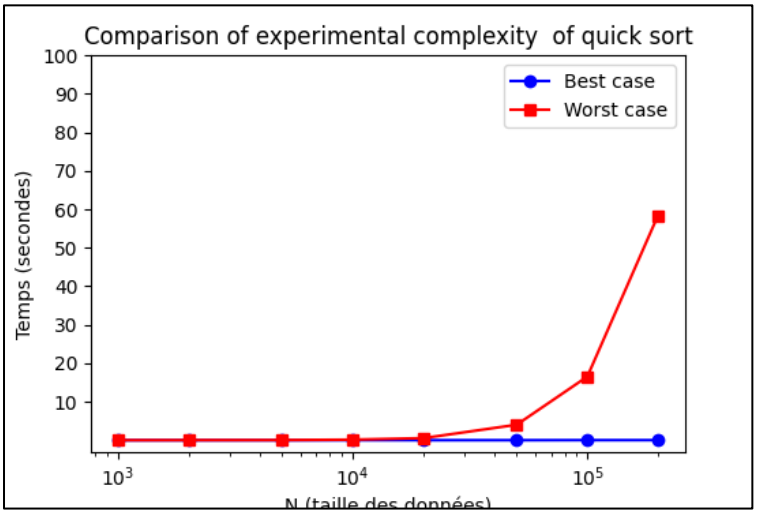
$$T(n) = n + (n-1) + (n-2) + \cdots + 3 + 2 + T(1) = \frac{n(n+1)}{2} = \frac{n^2}{2} = n^2$$

$T(n) = O(n^2)$

We have shown that the time complexity of QuickSort in the worst case is $O(n^2)$

Experimental comparison:

n	**1000**	**2000**	**5000**	**10000**	**20000**	**50000**	**100000**	**200000**
Best case	0.000	0.000	0.000	0.000	0.000	0.000	0.016	0.015
Worst case	0.000	0.015	0.032	0.141	0.531	4.02	16.534	58.375



Conclusion:
The experimental values for the best case are very small, often close to zero. This behavior aligns with the expected complexity $O(nlogn)$, as the algorithm is highly optimized for this scenario. On the other hand, the experimental data for the worst case increase rapidly with n, which is consistent with a quadratic complexity $O(n^2)$.

5) Heap sort

Algorithm Steps:

1. **Building a max heap**: A complete binary tree where each parent node is greater than its children.
2. **Heapifying**: The largest element is placed at the root.
3. **Sorting**: Swap the root with the last element and reduce the heap size, then repeat the process.

Algorithm:

```
HEAPIFY(array, n , root):
    largest = root
    leftChild = 2 * root + 1
    rightChild = 2 * root + 2
    if leftChild < n & array[leftChild] > array[largest]:
        largest = leftChild
    if rightChild < n & array[rightChild] > array[largest]:
        largest = rightChild
    if largest is not root:
        swap array[root] with array[largest]
        HEAPIFY(array, n , largest)
HEAPSORT(array, n):
    # Step 1: Build a max heap
    for i from (n  / 2 - 1) to 0:
        HEAPIFY(array, n , i)
    # Step 2: Extract elements from the heap
    for i from n - 1 to 1:
        swap array[0] with array[i]  # Move current root to end
        HEAPIFY(array, i, 0)        # Call heapify on reduced heap
```

C algorithm:

```c
void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
   for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
     heapify(arr, i, 0);
    }
}
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}
```

Complexity Analysis:

Time Complexity:

- **Best Case:** $O(n\ log(n))$
- **Average Case:** $O(n\ log(n))$
- **Worst Case:** $O(n\ log(n))$

How the Complexity is Derived:

- **Heap Construction:** $\boldsymbol{O(n)}$ – Building the max heap involves $\frac{n}{2}$ iterations

- **Heap Sort: $\boldsymbol{O(n\,log(n))}$**

After building the heap, we need to repeatedly remove the maximum element (root).

For each removal:

- Swap the root with the last element
- Perform **heapify** to restore the max heap property, it takes $O(log(n))$
- This is done $n-1$ times (for all elements except the last).
- The total complexity for this phase is: $(n-1)\cdot O(log(n)) = \boldsymbol{O(nlog(n))}$

Comparison between the best and the worst case:

Table size (n)	10000	20000	50000	80000	100000
Best case	0.003	0.005	0.012000	0.027	0.036
Worst case	0.003	0.005	0.012000	0.029	0.037

6) Conclusion

Theoretical complexity:

Algorithm	Best case	Worst case	Space complexity	Best use for
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$	**Small, nearly sorted datasets**
Gnome Sort	$O(n)$	$O(n^2)$	$O(1)$	**Small, nearly sorted datasets**
Radix Sort	$O(k * n)$	$O(k * n)$	$O(n + k)$	**Sorting integers with small digit ranges.**
Quick Sort	$O(n\ log(n))$	$O(n^2)$	$O(log(n))$	**Large datasets**
Heap Sort	$O(n\ log(n))$	$O(n\ log(n))$	$O(1)$	**Large datasets**

Analysis of Experimental Results:

- The Bubble Sort, Gnome Sort, and Optimized Bubble Sort are the most resource-intensive algorithms, with a quadratic complexity. Their execution time increases rapidly as the data size grows.
- The Optimized Bubble Sort represents a small improvement by reducing the number of iterations, but it does not change the overall complexity order.
- The Radix Sort algorithm is a special case; it is highly efficient for sorting integers with a fixed number of digits, but its disadvantage lies in the creation of an additional table. Therefore, it is not recommended in environments with limited central memory.
- The Heap Sort and Quick Sort algorithms both have a complexity of $O(nlog(n))$resulting in much faster execution times compared to quadratic sorts.
- Quick Sort is slightly more efficient than Heap Sort, but it relies heavily on choosing a good pivot. The experimental results clearly validate the theoretical complexity calculations.
- In summary, algorithms with $O(n^2)$complexity are significantly slower than those with $O(nlog(n))$complexity, showing that transitioning from a quadratic to a logarithmic complexity sorting algorithm can greatly optimize execution time.