Mini-Project: Sorting Algorithms

HADJ ARAB Adel   BECHAR Walid

January 23, 2025

1 Introduction

Sorting is one of the most classically studied families of algorithms, because they are among the modules essential for the good running of more advanced algorithms. The general principle of a sorting algorithm is to order (in ascending order for example) the objects of a collection of data (values), according to a comparison criterion (key – for us, values and keys are here confused: these are the elements of an array of integers). We generally carry out sorting by using an "in-place" approach: the sorted values are stored in the same array as the initial values (which therefore becomes an input-output parameter).

2 Sorting Algorithms

2.1 Bubble Sort

2.1.1 Algorithm

The **Bubble Sort** function in C sorts an array of integers in ascending order using the bubble sort algorithm. It iterates through the array multiple times, comparing and swapping adjacent elements if they are in the wrong order. The process is repeated until the array is sorted. An optimization is included with a **change flag** that allows the function to exit early if no swaps are made during a pass, indicating that the array is already sorted. This reduces unnecessary iterations and improves efficiency.

```
void bubble_sort(int arr[], int n) {
  int change = 1; // boolean variable
  while (change) {
    change = 0;
    for (int i = 0; i < n - 1; i++) {
      if (arr[i] > arr[i + 1]) {
        swap(&arr[i], &arr[i + 1]);
        change = 1;
      }
    }
  }
}
```

Listing 1: Bubble Sort implementation

The **Optimized Bubble Sort** function in C sorts an array of integers in ascending order using an optimized version of the bubble sort algorithm. It iterates through the array multiple times, comparing and swapping adjacent elements if they are in the wrong order. The process is repeated until the array is sorted. An optimization is included with a **change flag** that allows the function to exit early if no swaps are made during a pass, indicating that the array is already sorted. Additionally, the range of the inner loop is reduced by one after each pass, as the largest elements are moved to their correct positions at the end of the array. This reduces unnecessary comparisons and improves efficiency.

```
void optimized_bubble_sort(int arr[], int n) {
  int m = n - 1;
  int change = 1;

  while (change) {
    change = 0;
    for (int i = 0; i < m; i++) {
      if (arr[i] > arr[i + 1]) {
        swap(&arr[i], &arr[i + 1]);
        change = 1;
      }
    }
    m--;
  }
}
```

Listing 2: Optimized Bubble Sort implementation

2.1.2 Complexity

Both Bubble Sort and Optimized Bubble Sort have a worst-case and average-case time complexity of $O(n^2)$, where n is the number of elements to be sorted. This is because, in the worst case, each element needs to be compared with every other element, resulting in $n(n-1)/2$ comparisons.

However, the best-case time complexity differs between the two algorithms. For Bubble Sort, the best-case time complexity is $O(n)$ when the array is already sorted, as the algorithm will still iterate through the entire array once to confirm that no swaps are needed. For Optimized Bubble Sort, the best-case time complexity is also $O(n)$, but it can potentially reduce the number of comparisons further due to the decreasing range of the inner loop after each pass.

The space complexity for both algorithms is $O(1)$ as they are in-place sorting algorithms, meaning they do not require additional memory proportional to the input size.
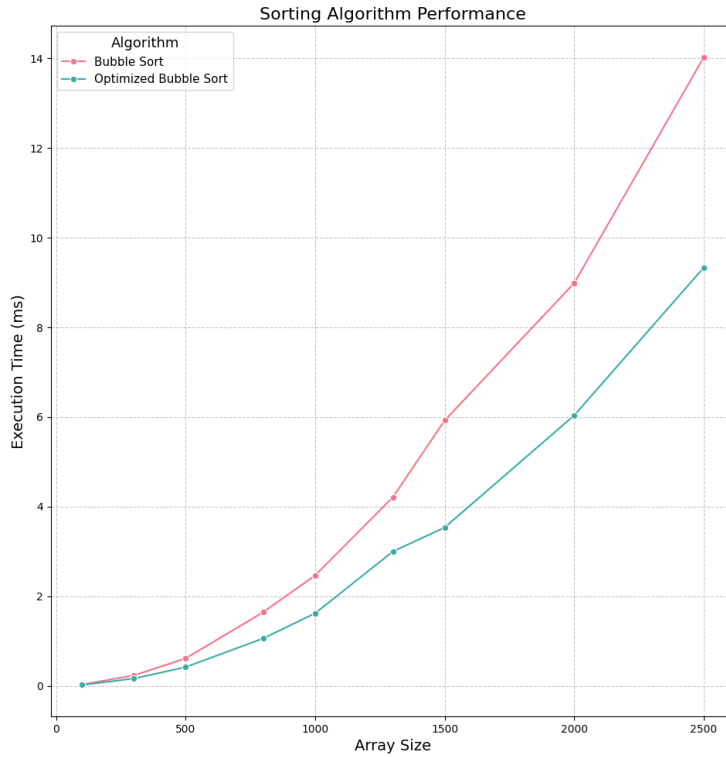
2.1.3 Results

Figure 1: Bubble Sort and Optimized Bubble Sort running time as a function of n

2.2 Gnome Sort

2.2.1 Algorithm

The **Gnome Sort** function in C sorts an array of integers in ascending order by iteratively comparing each element with the next one and swapping them if they are out of order, moving the index backward after a swap to recheck previous elements, and moving it forward otherwise, continuing this process until the entire array is sorted, ensuring that each element is in its correct position by the end of the algorithm.

```c
void gnome_sort(int arr[], int n) {
  int i = 0;

  while (i < n - 1) {
    if (arr[i] <= arr[i + 1]) {
      i++;
    } else {
      swap(&arr[i], &arr[i + 1]);
      if (i > 0) {
        i--;
      } else {
        i++;
      }
    }
  }
}
```

Listing 3: Gnome Sort implementation

2.2.2 Complexity

Gnome Sort has a worst-case and average-case time complexity of $O(n^2)$, where n is the number of elements to be sorted. The best-case time complexity is $O(n)$ when the array is already sorted. The space complexity is $O(1)$ as it is an in-place sorting algorithm.
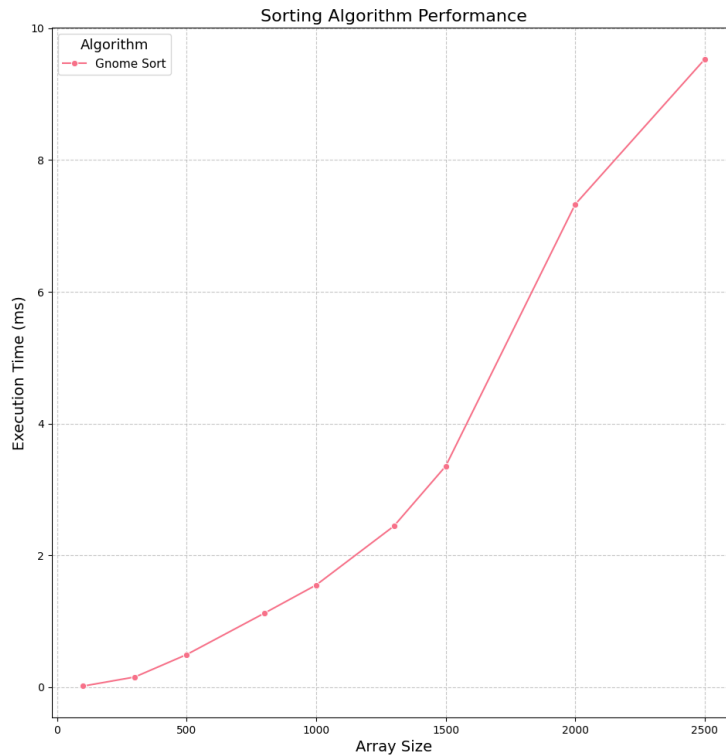
2.2.3 Results

Figure 2: Gnome Sort running time as a function of n

2.3 Radix Sort

2.3.1 Algorithm

The **key** function in C extracts the digit at a specific position (determined by the parameter **i**) from an integer **x**. It repeatedly divides **x** by 10 until the desired position is reached and then returns the digit at that position using the modulus operator.

```c
int key(int x, int i) {
  while (i > 0) {
    x = x / 10;
    i--;
  }
  return x % 10;
}
```

Listing 4: Key implementation

The **Sort Aux** function in C sorts an array of integers based on a specific digit (determined by the parameter 'i') using a counting sort approach, where it first counts the occurrences of each digit, then calculates the cumulative count, places the elements into an output array in sorted order according to the current digit, and finally copies the sorted elements back into the original array, ensuring that the array is partially sorted by the specified digit.

```c
void sort_aux(int arr[], int n, int i) {
  int *output = (int *)malloc(n * sizeof(int));
  int count[10] = {0};

  for (int j = 0; j < n; j++) {
    count[key(arr[j], i)]++;
  }
  for (int j = 1; j < 10; j++) {
    count[j] += count[j - 1];
  }
  for (int j = n - 1; j >= 0; j--) {
    output[count[key(arr[j], i)] - 1] = arr[j];
    count[key(arr[j], i)]--;
  }
  for (int j = 0; j < n; j++) {
    arr[j] = output[j];
  }

  free(output);
}
```

Listing 5: Sort Aux implementation

The **Radix Sort Helper** function in C sorts an array of integers by iterating through each digit from the least significant to the most significant, calling the **Sort Aux** function to sort the array based on the current digit, while the **Radix Sort** function determines the maximum number of digits in the largest number of the array and then calls **Radix Sort Helper** to perform the sorting process, ensuring the entire array is sorted by all digits.

```c
void radix_sort_helper(int arr[], int n, int k) {
  // k represents the maximum number of digits
  for (int i = 0; i < k; i++) {
    sort_aux(arr, n, i);
  }
}

void radix_sort(int arr[], int n) {
  // maximum number of digits is the number of digits of the
      largest number
  int imax = 0;
  for (int i = 1; i < n; i++) {
    if (arr[i] > arr[imax]) {
      imax = i;
    }
  }

  int k = (int)log10(arr[imax]) + 1;
  radix_sort_helper(arr, n, k);
}
```

Listing 6: Radix Sort implementation

2.3.2 Complexity

Radix Sort has a time complexity of $O(d \cdot (n + k))$, where d is the number of digits in the largest number, n is the number of elements, and k is the range of the digit values. The space complexity is $O(n + k)$.
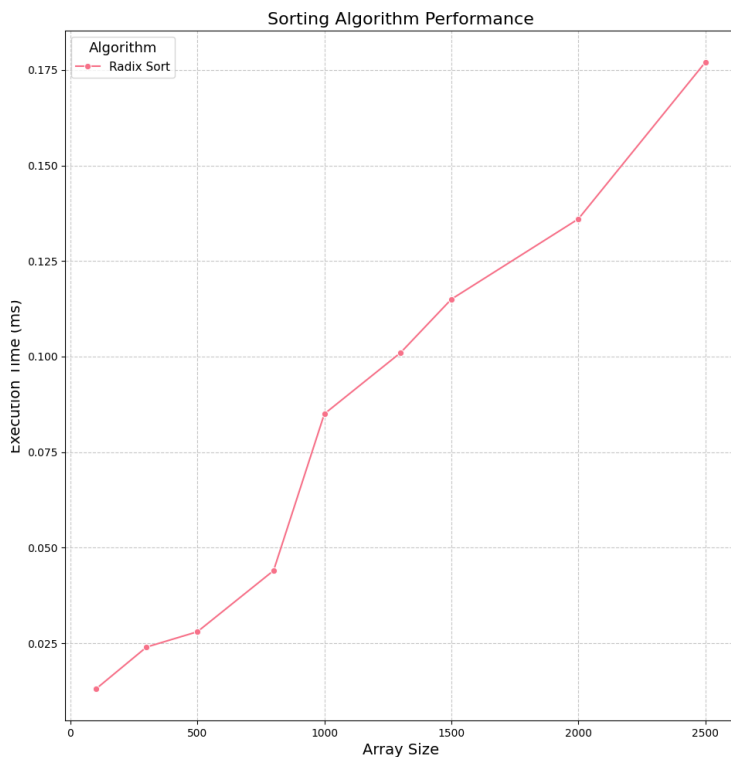
2.3.3 Results

Figure 3: Radix Sort running time as a function of n

2.4 Quick Sort

2.4.1 Algorithm

The **Partition** function in C rearranges the elements of an array around a pivot element (chosen as the first element of the array segment) such that all elements less than or equal to the pivot are on its left and all elements greater than the pivot are on its right, by iteratively moving indices from both ends towards the center, swapping elements as needed, and finally placing the pivot in its correct position, returning the index of the pivot.

```c
int partition(int arr[], int low, int high) {
  int p = arr[low];
  int i = low;
  int j = high;

  while (i < j) {
    while (arr[i] <= p && i <= high - 1) {
      i++;
    }
    while (arr[j] > p && j >= low + 1) {
      j--;
    }
    if (i < j) {
      swap(&arr[i], &arr[j]);
    }
  }
  swap(&arr[low], &arr[j]);
  return j;
}
```

Listing 7: Partition implementation

The **Quick Sort Helper** function in C recursively sorts an array of integers by partitioning the array around a pivot element and then sorting the subarrays on either side of the pivot, while the **Quick Sort** function serves as a wrapper that initializes the recursive process by calling **Quick Sort Helper** with the full range of the array.

```c
void quick_sort_helper(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);

    quick_sort_helper(arr, low, pi - 1);
    quick_sort_helper(arr, pi + 1, high);
  }
}

void quick_sort(int arr[], int n) { quick_sort_helper(arr, 0, n -
    1); }
```

Listing 8: Quick Sort implementation

2.4.2 Recurrence Equation

The recurrence equation for the Quick Sort algorithm can be derived based on the partitioning process. Let $T(n)$ be the time complexity of Quick Sort for an array of size n.

In the **worst case**, the pivot element divides the array into two subarrays of sizes 0 and $n-1$. Therefore, the recurrence relation for the worst case is:

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n)$$

We can expand this recurrence relation as follows:

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

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

$$\vdots$$

$$T(2) = T(1) + O(2)$$

$$T(1) = T(0) + O(1)$$

Summing all these equations, we get:

$$T(n) = O(n) + O(n-1) + \cdots + O(1)$$

This forms an arithmetic series, which sums to:

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

In the **best case**, the pivot element divides the array into two equal subarrays of size $n/2$. Therefore, the recurrence relation for the best case is:

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

To solve the best-case recurrence relation, we use the Master Theorem for divide-and-conquer recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a = 2$, $b = 2$, and $f(n) = O(n)$. According to the Master Theorem, since $a = b^c$ where $c = 1$, we have:

$$T(n) = O(n \log n)$$

Thus, the **best-case** and **average-case** time complexity of Quick Sort is $O(n \log n)$, while the **worst-case** time complexity is $O(n^2)$.
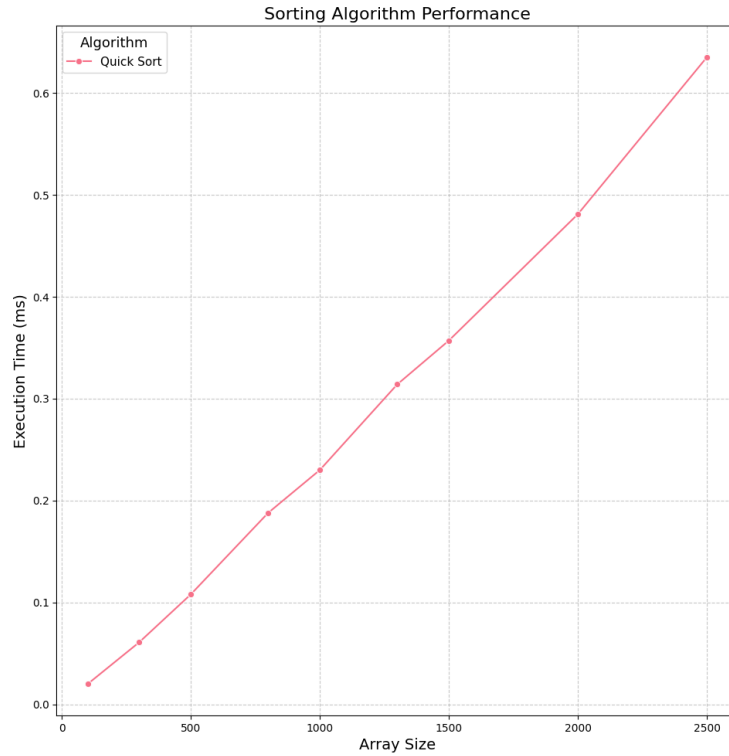
2.4.3 Results

Figure 4: Quick Sort running time as a function of n

2.5 Heap Sort

2.5.1 Algorithm

The **Heapify** function in C maintains the heap property of a binary heap by comparing a node with its children and swapping it with the largest child if necessary, then recursively applying the same process to the affected subtree, ensuring that the subtree rooted at the given node becomes a valid max-heap.

```
void heapify(int arr[], int n, int i) {
  int largest = i;
  int left = 2 * i + 1;  // Left child
  int right = 2 * i + 2; // Right child

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

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

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

Listing 9: Heapify implementation

The **Heap Sort** function in C sorts an array of integers in ascending order by first building a max heap from the array, then repeatedly extracting the maximum element from the heap and placing it at the end of the array, and finally re-heapifying the remaining elements until the entire array is sorted.

```
void heap_sort(int arr[], int n) {
  // Build max heap
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);

  // Extract elements from heap one by one
  for (int i = n - 1; i > 0; i--) {
    swap(&arr[0], &arr[i]);
    heapify(arr, i, 0);
  }
}
```

Listing 10: Heap Sort implementation

2.5.2 Complexity

Heap Sort has a time complexity of $O(n \log n)$ for all cases (worst, average, and best). The space complexity is $O(1)$ as it is an in-place sorting algorithm.
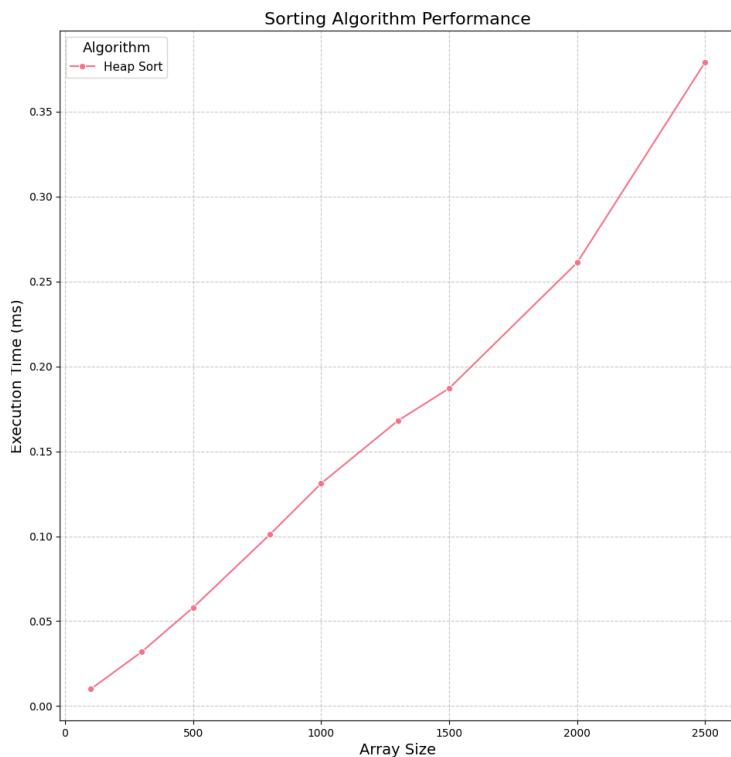
2.5.3 Results

Figure 5: Heap Sort running time as a function of n

3 Evaluation

In this section, we conduct a comprehensive evaluation of the performance of the implemented sorting algorithms. We meticulously compare their execution times across a diverse range of input sizes to provide a thorough analysis. This comparison will highlight the efficiency and scalability of each algorithm, offering valuable insights into their practical applications and limitations.
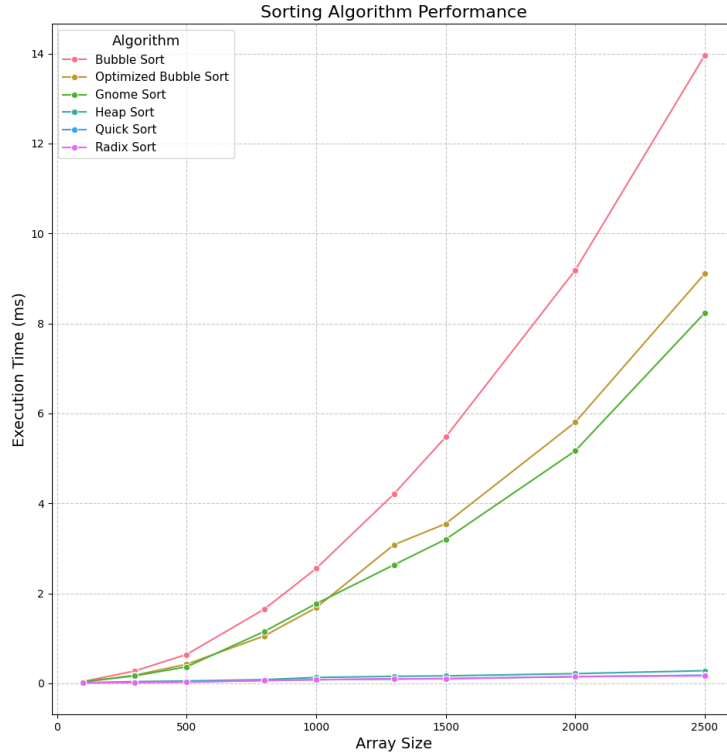
Figure 6: All Sorting Algorithms running time as a function of n

4   Conclusion

In this project, we have successfully implemented and rigorously tested a variety of sorting algorithms. Through detailed analysis, we have studied their computational complexities and compared these theoretical complexities with empirical evaluations of their running costs. The experimental results have consistently aligned with our theoretical expectations, thereby validating the efficiency and effectiveness of the sorting algorithms under study. This comprehensive evaluation not only reinforces the theoretical foundations of these algorithms but also provides practical insights into their performance across different scenarios. Our findings underscore the importance of understanding algorithmic complexity and its real-world implications, ultimately contributing to the broader field of computer science and algorithm design.