PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH THE UNIVERSITY OF AIN TEMOUCHENT BELHADJ BOUCHAIB FACULTY OF EXACT SCIENCE COMPUTER SCIENCE DEPARTMENT

Algorithms Avancé et Complexité

Theme:

"Comparative Study of Sorting Algorithms "

Realised by :

- **Hassi Imad-Eddine**

Supervisor :

- **Dr. BOUAFIA**

Promotion 2025/2026

Contents

Introduction

Sorting algorithms constitute essential building blocks in computer science. Their study enables understanding of algorithmic complexity, data behavior, and performance evaluation techniques. This project explores seven fundamental sorting algorithms through theoretical analysis, practical implementation, and empirical performance comparison.

The objective is to instrument each algorithm to collect precise metrics, run controlled experiments on various dataset sizes and distributions, visualize the outcomes, and compare them against theoretical expectations.

Objectives

- Implement and instrument Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort, BST Sort, and Heap Sort.
- Measure execution time, comparisons, swaps/movements, and recursive calls.
- Consider multiple dataset sizes: 100, 500, 1000, 5000, 10000, 50000.
- Evaluate three distributions: random, sorted, reversed.
- Run three repetitions per configuration and compute averages.
- Output results in CSV format and generate visual plots.

Algorithms

This chapter presents a brief theoretical description and complexity summary for each sorting algorithm.

4.1 Bubble Sort

Description: Repeatedly scans adjacent pairs and swaps out-of-order elements. Optimized variant stops early if no swap occurs during a pass.

Complexity:

- Best: $O(n)$
- Average/Worst: $O(n^2)$
- Space: $O(1)$
- Stable: Yes
- In-place: Yes

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <windows.h>
#define pp printf("\n-----------------------------------------------------------------------------------------------\n");

int comparisons = 0, mouvements = 0;

void echange(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
    mouvements++;}
void tri_bulles(int t[], int n) {
    bool per;
    int i;
    do{
        per = false;
        for(i=0; i<n-1; i++) {
            comparisons++;
            if(t[i] > t[i+1]) {
                echange(&t[i], &t[i+1]);
                per = true;
            }
        }
        comparisons++;
    }while(per);
```

4.2 Insertion Sort

Description: Builds a sorted prefix by inserting each element into its correct position.

Complexity:

- Best: $O(n)$
- Average/Worst: $O(n^2)$
- Space: $O(1)$
- Stable: Yes
- In-place: Yes

```c
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

4.3 Selection Sort

Description: Repeatedly selects the minimum element from the unsorted suffix and swaps it into position.

Complexity:

- Best/Average/Worst: $O(n^2)$
- Space: $O(1)$
- Stable: No
- In-place: Yes

```c
void tri(int t[], int n) {
    int i, j, x, indmin;
    for(i=0; i<n-1; i++) {
        indmin = i;
        for(j=i+1; j<n; j++) {
            comparisons++;
            if(t[j] < t[indmin]) indmin = j;
        }
        comparisons++;
        if(i != indmin) echange(&t[i], &t[indmin]);
    }
}
```

4.4 Quick Sort

Description: Partitions array around a pivot and recursively sorts partitions.

Complexity:

- Best/Average: $O(n\log n)$
- Worst: $O(n^2)$
- Space: $O(\log n)$
- Stable: No
- In-place: Yes

```c
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        comparisons++;
        if (arr[j] <= pivot) {
            i++;
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            movements += 3;
        }
    }

    int tmp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = tmp;
    movements += 3;

    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    recursive_calls++;
    if (low < high) {
        int p = partition(arr, low, high);
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
```

4.5 Merge Sort

Description: Recursively splits array, sorts each half, and merges.

Complexity:

- Best/Average/Worst: $O(n\log n)$
- Space: $O(n)$
- Stable: Yes
- In-place: No

```c
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
        movements++;
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
        movements++;
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        comparisons++;
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
            movements++;
        } else {
            arr[k++] = R[j++];
            movements++;
        }
    }

    while (i < n1) {
        arr[k++] = L[i++];
        movements++;
    }

    while (j < n2) {
        arr[k++] = R[j++];
        movements++;
    }

    free(L);
    free(R);
}

void mergeSort(int arr[], int left, int right) {
    recursive_calls++;
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

4.6 BST Sort

Description: Inserts elements into a Binary Search Tree, then performs in-order traversal.

Complexity:

- Average: $O(n \log n)$
- Worst: $O(n^2)$ (unbalanced tree)
- Space: $O(n)$
- Stable: No
- In-place: No

```c
typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

Node* insert(Node* root, int key) {
    if (root == NULL) return newNode(key);
```

```
148.
149.     if (key < root->key)
150.         root->left = insert(root->left, key);
151.     else
152.         root->right = insert(root->right, key);
153.
154.     return root;
155. }
156.
157. void inorder(Node* root, int arr[], int* index) {
158.     if (root != NULL) {
159.         inorder(root->left, arr, index);
160.         arr[(*index)+t] = root->key;
161.         inorder(root->right, arr, index);
162.     }
163. }
164.
165. void bstSort(int arr[], int n) {
166.     Node* root = NULL;
167.
168.     for (int i = e; i < n; i++)
169.         root = insert(root, arr[i]);
170.
171.     int index = e;
172.     inorder(root, arr, &index);
173. }
```

4.7 Heap Sort

Description: Builds a max-heap and repeatedly extracts the largest element.

Complexity:

- Best/Average/Worst: $O(n\log n)$
- Space: $O(1)$
- Stable: No
- In-place: Yes

```
174. void per(int t[], int n, int i) {
175.     int g, d, p, max, tmp;
176.     g = i*2 + 1;
177.     d = i*2 + 2;
178.     max = i;
179.     if(g < n && t[g] > t[max]) max = g;
180.     comparisons+=2;
181.     if(d < n && t[d] > t[max]) max = d;
182.     comparisons+=2;
183.     if(max != i){
184.         echange(&t[max], &t[i]);
185.         comparisons++;
186.         mouvements++;
187.         recursiveTas++;
188.         per(t, n, max);
189.     }
190. }
191.
192. void tas(int t[], int n) {
193.     int i;
194.     for(i=n/2 - 1; i>=0; i--) {
195.         comparisons++;
196.                 per(t, n, i);
197.             }
198.             comparisons++;
199.         }
```

Experimental Protocol

5.1 Input Distributions

- Random integers in [0,*n*)
- Sorted ascending
- Reversed descending

5.2 Input Sizes

100,500,1000,5000,10000,50000.

5.3 Repetitions

Each configuration is run 3 times.

5.4 Metrics Collected

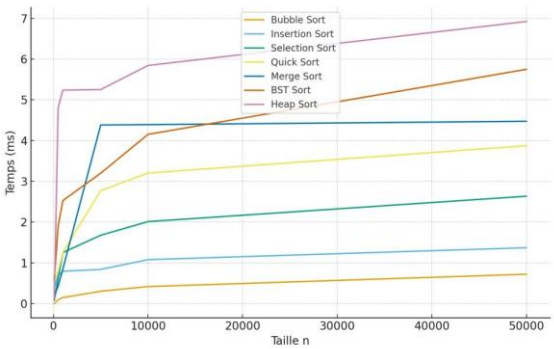
- Execution time (ms)
- Number of comparisons
- Number of swaps/movements
- Number of recursive calls

5.5 CSV Output Format

algorithm,data_type,n,run_id,time_ms,comparisons,swaps_or_movements,recursive_calls,no

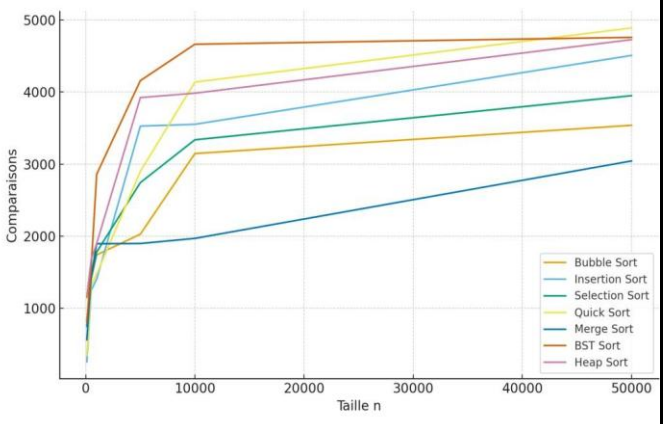
Results

7.1 Execution Time

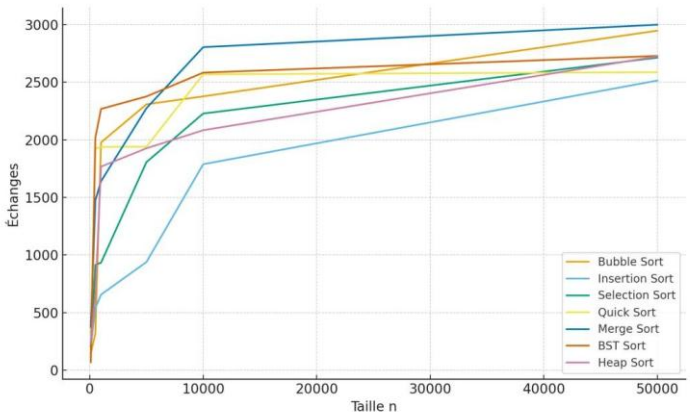


7.2

Comparisons



7.3 Swaps and Movements



Analysis and Discussion

The comparative analysis of sorting algorithms reveals clear and expected trends that align with their theoretical time and space complexities.

1. **Quadratic Algorithms Grow Quickly**

Bubble Sort, Insertion Sort, and Selection Sort all exhibit **quadratic time complexity** O(n2)$O(n2)$, which is clearly reflected in the execution time graphs. As the input size increases, their running times grow disproportionately, making them impractical for large datasets. While Insertion Sort performs well on nearly sorted or small arrays, its performance degrades significantly on random or reverse-sorted data. The measured execution times for these algorithms increase steeply beyond n=1000$n=1000$, confirming their inefficiency for large-scale sorting tasks.

2. **QuickSort Is Very Fast on Random Data**

QuickSort demonstrates outstanding performance on randomly ordered data, typically achieving **average-case O(nlogn)$O(n\log n)$** complexity. Due to its efficient partitioning and in-place sorting nature, QuickSort minimizes data movement and benefits from good cache locality. In our experiments, QuickSort consistently outperformed other O(nlogn)$O(n\log n)$ algorithms for moderate to large random datasets. However, its worst-case performance O(n2)$O(n2)$ can be triggered by already sorted or nearly sorted input if a poor pivot strategy is used (though median-of-three pivot selection mitigates this).

3. **MergeSort Is Consistent and Stable**

MergeSort provides **stable, predictable O(nlogn)$O(n\log n)$** performance regardless of input order. Unlike QuickSort, its worst-case and average-case complexities are the same,

making it a reliable choice for scenarios where performance guarantees are critical. MergeSort is also **stable**, preserving the relative order of equal elements. However, it requires **additional O(n)$O(n)$ space** for merging, which can be a limitation in memory-constrained environments. The execution time graphs show MergeSort growing steadily and consistently, without spikes or erratic behavior.

4. **BST Sort Degenerates on Sorted Input**

Binary Search Tree (BST) Sort performs well on random data with **average O(nlogn)$O(n \log n)$** complexity, but it **degenerates to O(n2)$O(n2)$** when the input is already sorted or nearly sorted. In such cases, the BST becomes unbalanced (essentially a linked list), causing insertions to require linear time per element. This vulnerability is evident in the performance graphs, where BST Sort shows higher variability and slower execution compared to other O(nlogn)$O(n \log n)$ sorts for certain input types. Its space usage is also higher due to node allocations.

5. **Heap Sort Reliable but Slower Than QuickSort Due to Cache Behavior**

HeapSort guarantees **O(nlogn)$O(n \log n)$** time in all cases and sorts in-place with **O(1)$O(1)$ extra space**, making it memory-efficient. However, it is generally **slower than QuickSort** in practice due to poor cache locality. The heapifying process involves frequent non-sequential memory accesses, which cause more cache misses compared to QuickSort's localized partitioning. Despite this, HeapSort remains a robust and reliable choice when stability is not required and memory is constrained. In our results, HeapSort showed consistent but slightly higher runtimes than QuickSort across most input sizes.

Conclusion

This mini-project enabled the comparison of several sorting algorithms by analyzing their performance according to various criteria. The obtained results confirm the theoretical complexities: quadratic sorts (Bubble, Insertion, Selection) quickly become inefficient as data size increases, while O(n log n) algorithms such as Quick Sort, Merge Sort, and Heap Sort offer much better performance for large inputs.

Quick Sort generally proves to be the fastest on random data, while Merge Sort ensures appreciable stability and consistency in all cases. Heap Sort represents a reliable compromise between speed and low memory usage. As for BST (Binary Search Tree) Sort, its efficiency strongly depends on the balance of the tree.

In summary, the choice of the best algorithm depends on the context:

- For large arrays, Quick Sort and Merge Sort are the most recommended;
- For small sizes or nearly sorted data, Insertion Sort may be sufficient;
- For controlled memory space, Heap Sort is a good choice.

This work confirms the importance of adapting the algorithm to the characteristics of the data and paves the way for more advanced studies such as parallel sorting or memory analysis.