Mini-Project: Sorting Algorithms

HADJ ARAB Adel BECHAR Walid

January 13, 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 **sorted 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.

```c
void bubble_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    int sorted = 1;
    for (int j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        swap(&arr[j], &arr[j + 1]);
        sorted = 0;
      }
    }
    if (sorted) {
      return;
    }
  }
}
```

Listing 1: Bubble Sort implementation

2.1.2 Complexity

Bubble 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.

2.1.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.1.4 Results

2.2 Gnome Sort

2.2.1 Algorithm

The following C code

```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 2: Matrix Multiplication

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 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.2.4 Results

2.3 Radix Sort

2.3.1 Algorithm

The following C code

```
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 3: Matrix Multiplication

```
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);
  }
}
```

Listing 4: Matrix Multiplication

```c
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 5: Matrix Multiplication

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 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.3.4 Results

2.4 Quick Sort

2.4.1 Algorithm

The following C code

```
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 6: Matrix Multiplication

```
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 7: Matrix Multiplication

2.4.2 Complexity

Quick Sort has a worst-case time complexity of $O(n^2)$, which occurs when the pivot selection is poor. However, the average-case and best-case time complexity is $O(n \log n)$. The space complexity is $O(\log n)$ due to the recursive stack space.

2.4.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.4.4 Results

2.5 Heap Sort

2.5.1 Algorithm

The following C code

```c
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]) // Changed < to >
    largest = left;

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

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

Listing 8: Matrix Multiplication

```c
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 9: Matrix Multiplication

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 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.5.4 Results

3 Conclusion

In this project, we put into practice and tested some sorting algorithms, studied their complexity, and compared theoretical complexity with the evaluation of running cost. The experimental results matched the theoretical expectations, confirming the efficiency of the studied sorting algorithms.