

République Algérienne Démocratique et Populaire

Ministère de l’Enseignement Supérieur et de la Recherche Scientifique

**Université des Sciences et de la Technologie Houari Boumediene**

**Faculté d’Informatique**

**Département des Systèmes Informatique**

**Last Name: ikram Last Name: rima**

**First Name: sirine First Name: djamaa**

**Matricule:202031026781 Matricule:202031075984**

**First Name: …………………….**

**Matricule:……………………..**

**project on Complexity Practical Work, Semester 1, Master's in Software Engineering.**

1. **Bubble Sort:**
   1. **Algorithm:**

the implementation of the BubbleSort algorithm in C:

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

void bubbleSort(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;

            }

        }

    }

}

* 1. **Theoretical Complexity :**

**Procedure BubbleSort (I/O: Array A[n] of integers ; I/n :integer)**

Begin **best case** **worst** **case**

Change = true ; 1 1 . while (Change=true) do 1 n

Change = false ; 1 n

for i=1 to n-1 do n-1 n(n-1)

if (A[i] > A[i+1]) then n-1 3n\*(n-1)

Swap(T[i], T[i+1]) ;

Change =true;

endif ;

done;

done ;

end.

The best case for BubbleSort is when the array is already sorted

The worst case for BubbleSort is when the array is sorted in descending order.

**BEST CASE:**

F(n)=1+1+1+n-1+n-1 = **2n+1**

So the asymptotic notation is linear : O(n).

**WORST CASE:**

F(n)= 1+n+n+n\*n-n+3n\*n-3n = **4n\*n-2n+1**

So the asymptotic notation is quadratic: O(n²).

* 1. **Graph**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | TbestE | TbestT | TworstE | TworstT |
| 100000 | 10 | 10 | 123 | 123 |
| 200000 | 22 | 19,99995 | 266 | 492,00123 |
| 300000 | 39 | 29,9999 | 543 | 1107,0037 |
| 400000 | 44 | 39,99985 | 889 | 1968,0074 |
| 500000 | 57 | 49,9998 | 1412 | 3075,0123 |

1. **Bubble Sort Optimal:**
   1. **Algorithm:**

the implementation of the BubbleSortOpt algorithm in C:

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

void bubbleSortOpt(int arr[], int n) {

    int m = n - 1;

    int change = 1;  // boolean variable

    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;

    }

}

* 1. **Theoretical Complexity :**

**Procedure BubbleSortOpt (I/O: Array A[n] of integers ; I/n :integer)**

Begin **best case** **worst case**

M=n-1; 2 2

Change = true ; 1 1 . while (Change=true) do 1 n

Change = false ; 1 n

for i=1 to n-1 do m 1+2+...+m . =n\*m(m+1)/2

if (A[i] > A[i+1]) then m n\*3[m(m+1)/2]

Swap(T[i], T[i+1]) ;

Change =true;

endif ;

done;

m=m-1; 2 2n

done;

end.

The best case for BubbleSortOpt is when the array is already sorted.

The worst case for BubbleSortOpt is when the array is sorted in descending order.

**BEST CASE:**

f(n)= 2+1+1+1+m+m+2 = 2m+7 = 2n-2+7 = **2n+5**

So the asymptotic notation is linear : O(n).

**WORST CASE:**

f(n) = 2+1+n+n+n(m(m+1)/2)+n(3[m(m+1)/2])+2n = 4n+3+2n(m\*m+m)

= **4n+3+2\*n\*n(n-1)**

So the asymptotic notation is quadratic: O(n²).

* 1. **Graph:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | TbestE | TbestT | TworstE | TworstT |
| 100000 | 29 | 20,50026 | 73 | 27,99986 |
| 200000 | 41 | 41 | 224 | 224 |
| 300000 | 62 | 61,49974 | 508 | 756,00126 |
| 400000 | 83 | 81,99949 | 829 | 1792,0045 |
| 500000 | 103 | 102,4992 | 1253 | 3500,0105 |

* **compare the two algorithms BubbleSort and BubbleSortOpt in the best and worst cases:**

|  |  |  |
| --- | --- | --- |
| **N** | **BubbleSort** | **BubbleSortOpt** |
| 100000 | 29 | 10 |
| 200000 | 41 | 22 |
| 300000 | 62 | 39 |
| 400000 | 83 | 44 |
| 500000 | 103 | 57 |

It can be clearly noticed that the execution time of the optimal algorithm decreases as soon as the values begin to be large because of the decrease in the number of iterations.

* 1. **conclusion :**

After studying both algorithms, bubble sort and optimized bubble sort, it is concluded that both methods have quadratic complexity in the worst case and linear complexity in the best case. Therefore, both methods are costly and slow. However, it remains that the optimized bubble sort method is more efficient than the regular bubble sort method.

1. **Gnome Sort:**
   1. **Algorithm:**

the implementation of the Gnome Sort algorithm in C:

#include <stdio.h>

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

void gnomeSort(int arr[], int n) {

    int i = 0;

    while (i < n-1) {

        if (i == 0 || arr[i] >= arr[i - 1]) {

            // Move towards the end of the array

            ++i;

        } else {

            // Swap and move towards the start of the array

            swap(&arr[i], &arr[i - 1]);

            --i;

        }

    }

}

* 1. **Theoretical Complexity :**

**Procedure GnomeSort (I/O: Array A[n] of integers ; I/n :integer)**

Begin **bestcase worstcase**

int i = 0; 1 1

while (i < n) { n n iteration

if (i == 0 || arr[i] >= arr[i - 1]) { 2(n-1) 1 ite: 1

// Move towards the end of the array 2 ite : 2

++i; 3 ite : 3

...

n-1 ite : n-1 fois

} else { 0 1 ite : 0

0 2 ite : 1

// Swap and move towards the start of the array 0 3 ite : 2

swap(&arr[i], &arr[i - 1]); 0 ...

--i; 0 n ite : n-1

}

}

The best case for GnomeSort is when the array is already sorted.

The worst case for GnomeSort is when the array is sorted in descending order.

**BEST CASE:**

F(n)= 1+n+2(n-1) = **3n-1**

So the asymptotic notation is linear : O(n).

**WORST CASE:**

F(n) = 1+2\*3\*= 1+6\*n(n-1)/2 = 1+3n(n-1) = **3n²+3n-2**

So the asymptotic notation is quadratic: O(n²).

* 1. **Graph:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **TbestE** | **TbestT** | **TworstE** | **TworstT** |
| 100000 | 7 | 7 | 54 | 54 |
| 200000 | 28 | 14,00002 | 270 | 215,99892 |
| 300000 | 31 | 21,00005 | 502 | 485,99676 |
| 400000 | 49 | 28,00007 | 698 | 863,99352 |
| 500000 | 54 | 35,00009 | 1155 | 1349,9892 |

* 1. **conclusion :**

After studying the algorithm corresponding to the Gnome Sort method, we conclude that this method has quadratic complexity in the worst case and linear complexity in the best case. Therefore, it is a slow and highly time- and space-consuming method. However, it remains more efficient than the Bubble Sort method and less efficient than the optimized Bubble Sort method.

1. **Radix Sort:**
   1. **Algorithm:**

the implementation of the Radix Sort algorithm in C:

// Function to get the key (digit at a specific position)

int key(int x, int i) {

int j, r;

    for (j=0; j<=i; j++) {

r= x%10;

x = x/10;

}

    return r;

}

// Function to reorder elements based on a specific digit position

void sortAux(int arr[], int n, int i) {

int T1[n], T2[n], j=0, k=0, l=0;

for(j=0;j<n;j++) {

T1[j] = key (T[j], i);

}

for(j=0; j<10; j++) {

for(k=0; k<n; k++){

if (T1[k] == j) {

T2[l] = T[k];

l++;

}

}

}

for(k=0; k<n; k++){

T[k] = T2[k];

}

    }

// Radix Sort function

void radixSort(int arr[], int n, int k) {

    // Perform sorting for each digit position

    for (int i = 0; i < k; ++i) {

        sortAux(T, n, i);

    }

}

* 1. **Theoretical Complexity :**

**function key(int x, int i):int {**

int j, r;

for (j=0; j<=i; j++) {  **= O(1)**

r= x%10;

x = x/10; }

return r;}

**Procedure sortAux (I/O: Array A[n] of integers ; I/n :integer; I/i :integer){**

int T1[n], T2[n], j=0, k=0, l=0;

for(j=0;j<n;j++) {

T1[j] = key (T[j], i);}

for(j=0; j<10; j++) {

for(k=0; k<n; k++){ ***+ + = 32n***

if (T1[k] == j) {

T2[l] = T[k];

l++;} }}

for(k=0; k<n; k++){

T[k] = T2[k];}}

**Procedure radixSort (I/O: Array A[n] of integers ; I/n :integer; I/k :integer)** {

int i;

for(i=0; i<k; i++) {

Sortaux(T, n, i);

}}

The complexity of this radix sort program is O(n \* k), where k is the number of digits in the numbers to be sorted, and n is the number of elements in the array. Each iteration involves an inner loop that traverses all elements of the array and sorts them based on the value of their i-th digit (where i is the iteration of the outer loop). Therefore, the total number of operations performed depends on the number of digits in the numbers to be sorted, the number of elements in the array, and the number of passes required for sorting.

**In precise terms:**

F(n)= k\* (+ + ) = **k\*32n**

in asymptotic notation : F1(n)=O(k\*n) = Ɵ(k\*n) linear complexity,

* 1. **Graph:**

|  |  |  |
| --- | --- | --- |
| **N** | **radixSortE** | **radixSortT** |
| 100000 | 14 | 4032 |
| 200000 | 27 | 8064 |
| 300000 | 41 | 12096 |
| 400000 | 58 | 16128 |
| 500000 | 86 | 20160 |

* 1. **Conclusion :**

After studying the algorithm corresponding to the radix Sort method, we conclude that this method has linear complexity in both the worst and best cases. Therefore, it is a fast method and is the fastest and most efficient among the three methods implemented so far.

1. **Quick Sort:**
   1. **Algorithm:**

the implementation of the quick Sort algorithm in C:

// Function to swap two elements in an array

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// Function to partition the array and return the index of the pivot

int partition(int arr[], int p, int r) {

    int pivot = arr[p];

    int i = p;

    int j = r;

    while (1) {

        while (arr[j] >= pivot) {

            j--;

        }

        while (arr[i] < pivot) {

            i++;

        }

        if (i < j) {

            swap(&arr[i], &arr[j]);

i++;j--;

        }

Else return j;

    }}

// Function to implement QuickSort

void quickSort(int arr[], int p, int r) {

    if (p < r) {

        int q = partition(arr, p, r);

        quickSort(arr, p, q);

        quickSort(arr, q + 1, r);

    }

}

* 1. **Theoretical Complexity :**

**BEST CASE:**

The partitioning method always produces two slices of equal size.

A = 2, B = 2 then

T(n) = 0 si n = 1

T(n) = 2\*T(n/2) + n si n>1

Resolution by substitution:

**T(n) = 2\*T(n/2) + n**

**= 2\*(2\*T(n/2)+n/2)+n = 22+T(n/22)+2n**

**= 2\*(2\*T(n/2)+n/22)+2n = 23 \* T(n/23)+3n**

**.**

**.**

**.**

**= 2k \* T(n/2k) + k n**

We assume n = 2k => k = log2 n

Then the complexity is : **O(n log n)**

**WORST CASE:**

The worst-case scenario occurs when the randomly chosen pivot is the smallest element in the list, and this situation repeats with each recursive call. If the pivot is the minimum of the list, one of the two recursive calls will be on an empty list, and the other on a list with n-1 elements.

Let T(n) be the number of comparisons made by the algorithm. Since the partitioning requires n-1 comparisons, we have:

T(n) = T(n-1)+n-1 = (1+2+...+n-1) = = n(n-1)/2 = **(n\*n-n)/2**.

Therefore, the complexity is in **O(n^2).**

* 1. **Graph:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **TbestE** | **TbestT** | **TworstE** | **TworstT** |
| 100000 | 18 | 18 | 15 | 15 |
| 200000 | 67 | 38,16742 | 86 | 60 |
| 300000 | 115 | 59,15291 | 164 | 135 |
| 400000 | 200 | 80,66966 | 260 | 240 |
| 500000 | 308 | 102,5815 | 417 | 375 |

* 1. **Conclusion :**

After studying the algorithm corresponding to the Quick Sort method, we conclude that this method has quadratic complexity in the worst case and subquadratic complexity in the best case. Therefore, it is a fast method, but it remains less efficient than the Distribution Sort method.

1. **Heap Sort:**
   1. **Algorithm:**

the implementation of the Heap Sort algorithm in C:

// Function to swap two elements in an array

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {

    int largest = i;         // Initialize largest as root

    int left = 2 \* i + 1;    // Left child

    int right = 2 \* i + 2;   // Right child

    // If left child is larger than root

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

        largest = left;

    }

    // If right child is larger than largest so far

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

        largest = right;

    }

    // If largest is not root, swap and recursively heapify the affected subtree

    if (largest != i) {

        swap(&arr[i], &arr[largest]);

        heapify(arr, n, largest);

    }

}

// Main function to perform Heap Sort

void heapSort(int arr[], int n) {

    // Build heap (rearrange array)

    for (int i = n / 2 - 1; i >= 0; i--) {

        heapify(arr, n, i);

    }

    // Extract elements from the heap one by one

    for (int i = n - 1; i > 0; i--) {

        // Move the current root to the end

        swap(&arr[0], &arr[i]);

        // Call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

* 1. **Theoretical Complexity :**

**Procedure heapify (I/O: Array A[n] of integers ; I/n :integer; I/i :integer){**

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) {

swap(&arr[i], &arr[largest]);

heapify(arr, n, largest);

}

}

T(n)=T(2n/3)+O(1)( Comparison between the left and right elements, as well as the root and nodes, is conducted before any permutation is carried out)

T(n)=O(log(n))

**Procedure heapSort (I/O: Array A[n] of integers ; I/n :integer){**

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i); }

for (int i = n - 1; i > 0; i--) {

swap(&arr[0], &arr[i]);

heapify(arr, i, 0);

}}

heap complexity = complexity of heapify + The execution of the loop from 2 to n for heapify

T(n)=O(n)+(n-1)O(log(n))=**O(nlog(n))**

The complexity of this sorting algorithm is O(n log n), where n is the size of the array to be sorted.

During the heap construction in a bottom-up manner, the cost to build the heap is O(n) as we need to visit each node of the heap exactly once to determine its final position. However, for each visited node, we may need to make one or more comparisons to maintain the heap properties, incurring a cost of O(log n) operations each time. Additionally, each extraction of the minimum element from the heap may also cost O(log n) for maintaining the heap properties.

As a result, the overall complexity of the heap sort algorithm is O(n log n). This is because we need to build the heap in O(n) time, perform n extractions of the minimum element with O(log n) cost each time, resulting in a total of O(n log n) operations.

* 1. **Graph :**

|  |  |  |
| --- | --- | --- |
| **N** | **TheapE** | **TheapT** |
| 50000 | 12 | 2923,931 |
| 100000 | 27 | 6222,492 |
| 200000 | 43 | 13194,25 |
| 300000 | 70 | 20448,81 |
| 400000 | 89 | 27887,02 |

* 1. **Conclusion:**

After studying the algorithm corresponding to the Heap Sort method, we conclude that this method has subquadratic complexity in both the worst and best cases. Therefore, it is a very fast method and is the fastest and least costly among all the methods studied. Thus, it is the most efficient method.

* **Comparison of the 6 sorting algorithms in terms of execution time in the best case:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **N** | **bubbleopt** | **bubble** | **gnome** | **heap** | **quick** | **radix** |
| 100000 | 20,49923 | 10 | 7 | 6222,4924 | 18 | 129024 |
| 200000 | 40,99898 | 19,99995 | 14,000023 | 13194,248 | 38,167416 | 258048 |
| 300000 | 61,49872 | 29,9999 | 21,000047 | 20448,807 | 59,1529096 | 387072 |
| 400000 | 81,99846 | 39,99985 | 28,00007 | 27887,021 | 80,6696639 | 516096 |
| 500000 | 102,4982 | 49,9998 | 35,000093 | 35461,797 | 102,58146 | 645120 |

* **Comparison of the 6 sorting algorithms in terms of execution time in the worst case:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **N** | **bubbleopt** | **bubble** | **gnome** | **heap** | **quick** | **radix** |
| 100000 | 27,99986 | 123 | 54 | 6222,4924 | 15 | 129024 |
| 200000 | 224 | 492,0012 | 215,99892 | 13194,248 | 60 | 258048 |
| 300000 | 756,0013 | 1107,004 | 485,99676 | 20448,807 | 135 | 387072 |
| 400000 | 1792,004 | 1968,007 | 863,99352 | 27887,021 | 240 | 516096 |
| 500000 | 3500,01 | 3075,012 | 1349,9892 | 35461,797 | 375 | 645120 |

**Final Conclusion :**

Sorting algorithms are among the classics in computer science. They can be grouped into two categories: slow versions, whose complexity is generally O(N^2), and fast versions, which are more efficient but more complex to implement, and often have a complexity close to O(N log N). Some have a complexity close to that of slow versions, but they minimize the cost of data exchanges.

We will summarize all the points studied in this project in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithms | Best case | Worst case | notice |
| Heap sort | O(nlog(n)) | O(nlog(n)) | The faster one |
| Radix sort | O(n) | O(n) | faster |
| Quick sort | O(nlog(n)) | O(n²) | The slower one |
| Optimized Bubble sort | O(n) | O(n²) | costly |
| Gnome sort | O(n) | O(n²) | Costly and slow |
| Bubble sort | O(n) | O(n²) | The cost |