**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY**

**UNIVERSITY OF SCIENCE**

**FACULTY OF INFORMATION TECHNOLOGY**



**DATA STRUCTURES & ALGORITHMS**

**SORTING ALGORITHMS PROJECT**

**Group: 10**

**Instructors:**

Mr. Bùi Huy Thông

Mrs. Trần Thị Thảo Nhi

**Group members:**

20127169 – Phạm Huy Hoàng

20127458 – Đặng Tiến Đạt

20127566 – Hoàng Quốc Nam

20127637 – Võ Lê Anh Thông

# **INTRODUCTION**

First of all, Group 10 would like to send our sincere thanks to the University of Science have brought Data Structures & Algorithms to the education program. Specially, we would like to thank deeply our practice lecturers & instructors *– Mr. Bùi Huy Thông* and *Mrs. Trần Thị Thảo Nhi*, who have taught and imparted valuable knowledge to us during school time. During studying times, we have improved ourselves with many useful skills, serious and effective – learning spirit. This will definitely be valuable knowledge, plant the seed for us to go out into the wider world.

After this mid – term project, we, Group 10, have improved a lot about communication skills, time management skills and problem – solving. So precious that we are learnt how to work like a team, listen to our partner, ask for helps and share difficulties with each other. Besides, acquiring knowledge by search on the web is one of the necessary skills needed in the Faculty of Information Technology.

Data Structures & Algorithms is an interesting subject, very useful with high practicality. Guaranteed to provide enough knowledge, associated with the actual demand of students. However, because of our limited knowledge and receptive ability, even we have tried our best but certainly, our project is hard to avoid deficiencies and mistakes, we are hope that our report is considered and feedback to make our project more complete.

Sincerely yours.

**II. INFORMATION**

After many days of studying and absorbing knowledge from the lecturer, group 10 understood the application and convenience of the sort technique in programming.

In this project, we were asked to implement all algorithm (for ascending order only), besides we chose Set 2 (11 algorithms):

* Selection Sort.
* Insertion Sort.
* Bubble Sort.
* Shaker Sort.
* Shell Sort.
* Heap Sort.
* Merge Sort.
* Quick Sort.
* Counting Sort.
* Radix Sort.
* Flash Sort.
* **Data size**
* Data size of input array: n.
* Array: a (begins at a[0] and ends at a[n-1]).
* The element at i position of array a: a[i] (∀i ∈ [0; n-1]).
* Data size of input array: 10000 ≤ n ≤ 500000.
* Data size of each element: 0 ≤ a[i] ≤ n.
* **Charts and tables**
* Line graph showing the running time of each function.
* The time unit in the graphs is *millisecond* (ms).
* Bar chart showing the comparisons of each function.
* **Compiling**
* To compile the program, we use the compile command:

**g++ main.cpp DataGenerator.cpp -o <file\_name>.exe**

**III. ALGORITHM PRESENTATION:**

1. **Selection Sort**
2. **Ideas**

* The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the sorted list. The algorithm maintains two subarrays in a given array, the subarray which is already sorted and the remaining subarray which is unsorted. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:
  + 1: Set a marker for the unsorted section at the front of the list
  + 2: Repeat steps 3 - 5 until one number remains in the unsorted section
  + 3: Compare all unsorted numbers in order to select the smallest one
  + 4: Swap this number with the first number in the unsorted section
  + 5: Advance the marker to the right one position
* Example: **𝑎 = {20,12,10,15, 2}**

|  |  |  |
| --- | --- | --- |
| Step | Array a | Explain |
| 1 | {| 20,12,10,15, **2**} | Initially the sorted part has nothing and the smallest element of the unsorted part is **2**, we will bring the element **2** first and increase the length of the sorted part. |
| 2 | {2 | 20,12,**10**,15} | Now that the smallest element of the unsorted part is **10**, we bring **10** over the sorted part. |
| 3 | {2 ,10 ,| 20 ,**12** ,15} | Now that the smallest element of the unsorted part is **12**, we bring **12** over the sorted part. |
| 4 | {2 ,10 ,12 ,| 20 ,**15**} | Now that the smallest element of the unsorted part is **15**, we bring **15** over the sorted part. |
| 5 | {2 ,10 ,12 ,15 ,| 20 } | At this point, the algorithm stops. |

1. **Complexity evaluations**

* The number of comparisons to be performed is 𝑂( ). For each stage we need a permutation, because so we need 𝑛 − 1 permutation, or 𝑂(𝑛) permutation.
* **Time Complexities:**
* Worst Case: If we want to sort in ascending order and the array is in descending order then, the worst case occurs 𝑂( ).
* Best Case: It occurs when the array is already sorted 𝑂( ).
* Average Case: It occurs when the elements of the array are in jumbled order (neither ascending nor descending) 𝑂( ).
* **Space Complexities:** O(1).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 189 | 1732 | 4823 | 19315 | 175109 | 490158 |
| Nearly Sorted | 103 | 897 | 2467 | 10024 | 70386 | 207837 |
| Sorted | 57 | 510 | 1415 | 5627 | 78660 | 228204 |
| Reversed | 194 | 1322 | 3612 | 9717 | 90676 | 253643 |

* Run time in millisecond table:

1. **Insertion Sort**
2. **Ideas**

* Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part (in this case, it will be placed at the position where the element a[i] larger than a[i-1] and smaller than a[i+1]).

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:
* 1: Iterate from a[1] to a[n] over the array.
* 2: Compare the current element (key) to its predecessor.
* 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element. And all elements which from its their index to position 0 are larger than the key, swap them with the top.
* Ex: There is an unsorted array:

**a = {12, 11, 13, 5, 6}**

|  |  |  |
| --- | --- | --- |
| Stage | Array | Explanation |
| 0 | 12, 11, 13, 5, 6 | In the beginning, the array doesn’t have no change, start at a[1] = 11. |
| 1 | 11, 12, 13, 5, 6 | Now, we compare and see that a[0] = 12 > a[1], so we will swap them to place at suitable positions (11 < 12) |
| 2 | 11, 12, 13, 5, 6 | Then, a[1] < a[2] (12 < 13), so we continue the loop. |
| 3 | 5, 11, 12, 13, 6 | We can see a[3] < a[2], so we keep compare with a[1], a[0]. All of them is larger than a[3], so we place 5 at a[0]. |
| 4 | 5, 6, 11, 12, 13 | At a[5] = 6, we still compare with a[4],a[3],…  But we can see that, 5 < 6 < 11, so we will place 6 at position between them. |
| 5 | 5, 6, 11, 12, 13 | The algorithm finishes here. We will have a sorted array. |

1. **Complexity evaluations**

* **Time complexity:**
* Average complexity: O(n2) → Randomized data.
* Best Case: O(n) → Sorted Data (+ Nearly Sorted Data).
* Worst Case: O(n2) → Reversed Data.
* **Space Complexity**: O(1).Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 48 | 427 | 1221 | 4869 | 46944 | 134285 |
| Nearly Sorted | 0 | 1 | 1 | 1 | 1 | 1 |
| Sorted | 0 | ~ 0 | ~ 1 | 1 | 1 | 1 |
| Reversed | 95 | 866 | 2379 | 9728 | 95357 | 271759 |

1. **Variants / Improvements**

* Shell Sort.
* Binary Insertion Sort.

1. **Bubble Sort**
2. **Ideas**

**-**Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.

-         If at least one swap has been done, repeat step 1.

1. **Step-by-step descriptions**

-         Example:         𝑎 = {2, 3, 4, 5, 1}

|  |  |  |
| --- | --- | --- |
| **Step** | **Array a** | **Explain** |
| 1 | {2, 3, 4, 5, 1} | Unsorted |
| 2 | {2, 3, 4, 5, 1} | We begin with the loop from left to right, we compare two adjacent number : because 2 < 3 , ok |
| 3 | {2, 3, 4, 5, 1} | 3 < 4 , ok |
| 4 | {2, 3, 4, 5, 1} | 4 < 5 also ok |
| 5 | {2, 3, 4, 5, 1} | In here we have 5 > 1 so we swap 5 and 1 ; To moving the large number to the right side of array |
| 6 | {2, 3, 4, 1, 5} | 4 > 1 we swap 4 and 1 |
| 7 | {2, 3, 1, 4, 5} | 3 > 1 we swap 3 and 1 |
| 8 | {2, 1, 3, 4, 5} | 2 > 1 we swap 2 and 1, We continue until we don’t have any swap operation. |
| 9 | {1, 2, 3, 4, 5} | We have a sorted array |

1. **Complexity evaluations**

* In Bubble Sort,  n-1 comparisons will be done in the 1st pass,  n-2 in 2nd pass,  n-3 in 3rd pass and so on.
* **Time Complexity**
* Worst and Average Case Time Complexity:  O(n2) Worst case occurs when the array is reverse sorted.
* Best Case Time Complexity: O(n). Best case occurs when the array is already sorted.
* **Space Complexity:** O(1).
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 143 | 1,134 | 3,156 | 14,259 | 124,189 | 967,101 |
| Nearly Sorted | 424 | 2,753 | 7,945 | 32,280 | 289,596 | 841,542 |
| Sorted | 146 | 1,139 | 3 | 13,260 | 125,771 | 358,911 |
| Reversed | 387 | 3,348 | 9,797 | 41,499 | 376,010 | 1,094,703 |

1. **Variants/improvements**

* One way to improve this algorithm is to phase it out, instead of just introducing  small elements in front, we can put the largest element in back.
* Another way is shaker sort , this is an improvement of bubble sort.
* Improve this process itself is to stop browsing as soon as the array is sorted, it has a name is Optimized bubble sort.

1. **Heap Sort**
2. **Ideas**

* The Heap sort algorithm has the basic idea of separating the array into two parts, one is sorted in ascending order and the other part is to maintain it as a max-heap.
* Max-heap is defined as a heap with the maximum value at the top of the heap. Max-heap is a binary tree with a parent node that is larger than the two children.

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:
* Stage 1: Adjust the original number sequence to heap
* Stage 2: Sort the array based on the heap
* Step 1: Bring the largest element to the position standing at the end of the sequence

+ r = n – 1

+ swap(a[0], a[r])

* Step 2:
* Remove the largest element from the heap: r = r – 1.
* Edit the rest of the range from a[0], a[1], ..., a[r] to a heap.
* Step 3: If r > 0 (heap has elements): Repeat step 2. Opposite: stop.
* Example:

**a= {20, 12, 2, 15, 10} , n = 5**

* When setting heap using array, I consider element standing at position i will have left child at position 2\*i + 1 and right child at position 2 \* i + 2

**Stage 1: Adjust the original number sequence to heap**

(Note: the red cabinet is the element being considered, the blue element is the child of the element under consideration)

|  |  |  |
| --- | --- | --- |
| Position | Array | Explanation |
| 1 | {**20, 12, 2, 15, 10**} | Due to the elements in the segment [n/2; n-1] will have no children so we skip it without considering, we will consider from the element at position n/2 - 1. Element 1 has two children, 15 and 10, due to stopping max-heap, we need put 15 up instead of 12 |
| 0 | {**20, 15, 2, 10, 12**} | We consider the element at position 0, this element has 2 subvalues, the value 15 and 2 are both less than 20, so we do nothing.  Complete phase 1. |

**Stage 2: sort array based on heap**

(Note : character "|" to split array into 2 parts heap and sorted part)

|  |  |  |
| --- | --- | --- |
| Repeats | Array | Explanation |
| 1 | {**20, 15, 2, 10, 12, |**} | Firstly, we see element 20 at the end of the array, then bring element 12 up instead of element 20 |
| 1 | {**12, 15, 2, 10, | 20**} | We swap positions of values 12 and 15 to ensure max-heap. |
| 1 | {15**, 12, 2, 10, | 20**} | Element 12 is at place, we stop iteration 1. Now element 12 has only 1 child, because element 20 is already outside the current management area. |
| 2 | {15**, 12, 2, 10, | 20**} | Move 15 backwards and move the element 10 up instead of |
| 2 | {**10, 12, 2, | 15, 20**} | Swap the position of element 10 and 12 to ensure max-heap |
| 2 | {**12, 10, 2, | 15, 20**} | Element 10 is already in place and has 0 children. Stop the 2nd iteration |
| 3 | {**12, 10, 2, | 15, 20**} | Bring 12 back and bring the element 2 up instead of. |
| 3 | **{2, 10 | 12, 15, 20}** | Swap position 2 elements 2 and 10 to ensure max-heap. |
| 3 | **{10, 2 | 12, 15, 20}** | Element 2 is already in place and has 0 children. Stop the 3nd iteration. |
| 4 | **{10, 2 | 12, 15, 20}** | Bring 10 back and bring the element 2 up instead of |
| 4 | **{2 | 10, 12, 15, 20}** | Element 0 is already in place and has 0 children. Stop the 4nd iteration |

1. **Complexity evaluations**

* **Time complexity:**
  + Average complexity: O(nlogn)
  + Best Case: O(n).
  + Worst Case: O(nlogn)
* **Space Complexity**: O(1)
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 2 | 13 | 18 | 38 | 158 | 234 |
| Nearly Sorted | 1 | 5 | 8 | 17 | 59 | 97 |
| Sorted | 1 | 4 | 7 | 19 | 54 | 97 |
| Reversed | 3 | 10 | 10 | 19 | 55 | 97 |

1. **Merge Sort**
2. **Ideas**

* In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.
* The concept of Divide and Conquer involves three steps:
* Divide the problem into multiple small problems.
* Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
* Combine the solutions of the subproblems to find the solution of the actual problem.

1. **Step-by-step descriptions**

Example:**𝑎 = {14, 7, 3, 12, 9, 11, 6, 2}**

|  |  |  |
| --- | --- | --- |
| **Step** | **Array a** | **Explain** |
| 1 | {14,17, 3, 12} and {9,11,6,2} | We will break these subarrays into even smaller subarrays ,until we have multiple subarrays with a single element in them. |
| 2 | {14,17} and  {3, 12} and {9,11} and {6,2} | Continue divide array. |
| 3 | {14} and {17} and {3} and {12} and {9} and {11} and {6} and {2} | Now we have multiple subarrays with single element in them |
| 4 | {14,17} and  {3, 12} and {9,11} and {2,6} | Then we have to merge all these sorted subarrays, step by step to form one single sorted array. |
| 5 | {3,12, 14, 17} and {2,6,9,11} | We continue merge small arr until we got sorted original array |
| 6 | {2, 3, 6, 9, 11, 12, 14, 17} | Now we got a sorted array |

1. **Complexity evaluations**

* Merge Sort is quite fast and has a time complexity of O(n\*log2(n)). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.
* We divide a number into half in every step, it can be represented using a logarithmic function, which is log n and the number  of steps can be represented by log2(n) + 1(at most)
* Also, we perform a single step operation to find out the middle of any subarray, i.e. O(1).
* And to merge the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required.
* Hence the total time for mergeSort function will become n(log n + 1), which gives us a time complexity of O(n\*log n).
* **Time Complexity:**
* Worst Case: O(n \* log2(n))
* Best Case: O(n \* log2(n))
* Average Case: O(n \* log2(n))
* **Space Complexity:** O(n)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 7 | 14 | 19 | 40 | 139 | 240 |
| Nearly Sorted | 2 | 7 | 16 | 20 | 54 | 94 |
| Sorted | 3 | 6 | 10 | 19 | 58 | 97 |
| Reversed | 6 | 16 | 69 | 40 | 144 | 184 |

1. **Quick Sort**
2. **Ideas**

* Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:
  + 1: If the range has less than two elements, return immediately as there is nothing to do.
  + 2: Otherwise pick a value, called a pivot, that occurs in the range.
  + 3: Partition the range: reorder its elements, while determining a point of division, so that all elements with values less than the pivot come before the division, while all elements with values greater than the pivot come after it.
  + 4: Recursively apply the quicksort to the sub-range up to the point of division and to the sub-range after it, possibly excluding from both ranges the element equal to the pivot at the point of division.
* Example: There is an unsorted array: 𝑎 = {4,1,0,3, 2}

|  |  |  |
| --- | --- | --- |
| Stage | Array | Explanation |
| 1 | {4,1,0,3, **2**} | Select **2** as a pivot. Divide into two arrays [1, 0] and[4, 3] |
| 2 | { [**1**, 0] ,2 ,[ 4, 3]} | In [1, 0] we select **1** as a pivot. Divide into two arrays [1] and [0]. |
| 3 | { 0, 1, 2, [ **4**, 3]} | In [4, 3] we select **4** as a pivot. Divide into two arrays [4] and [3]. |
| 4 | { 0, 1, 2, 3, 4} | We have a sorted array |

1. **Complexity evaluations**

* The complexity of the quick sort algorithm depends on how we choose the pivot element:
* **Best case & Average case:** **O(n\*log2n)** (In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size).
* **Worst case: O(n2)**. (This may occur if the pivot happens to be the smallest or largest element in the list, it is divided into 2 arrays of length 1 and (n – 1).
* **Space Complexity: O(log2n).**
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 3 | 6 | 10 | 19 | 61 | 93 |
| Nearly Sorted | 0 | 1 | 2 | 5 | 13 | 23 |
| Sorted | 1 | 2 | 2 | 3 | 10 | 16 |
| Reversed | 0.1 | 2 | 4 | 7 | 25 | 38 |

1. **Radix Sort**
2. **Ideas**

* If in other algorithms, the basis for sorting is always comparing the value of two elements, then radix-sort is based on the postal trial taxonomy. It doesn't care at all about comparing element values, and the sorting and sorting order itself creates the ordering of the elements.

1. **Step-by-step descriptions**

* Example:

**a= {29, 88, 52, 19, 43} , n = 5**

* Sort digits by units

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  | 19 |
|  |  | 52 | 43 |  |  |  |  | 88 | 29 |

→ After reviewing and comparing the unit row, the array becomes:

52, 43, 88, 29, 19

* Sort element by tens

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  | 19 | 29 |  | 43 | 52 |  |  | 88 |  |

→ After reviewing and comparing the ten rows, the array becomes

⇨**19, 29, 43, 52, 88**

1. **Complexity evaluations**

* **Time complexity:**
* With a sequence of n numbers, each number has a maximum of m digits, the algorithm performs m times the batching and concatenating operations. In the batching operation, each element is considered exactly once, also when concatenating. . Thus, the cost of implementing the algorithm is obviously O(2mn) = O(n)
* **Space Complexity**: O(1)
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 1 | 6 | 8 | 16 | 57 | 48 |
| Nearly Sorted | 1 | 3 | 8 | 11 | 35 | 58 |
| Sorted | 2 | 3 | 5 | 9 | 35 | 58 |
| Reversed | 3 | 8 | 8 | 12 | 35 | 70 |

1. **Shaker Sort**
2. **Ideas**

* The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. At the end of the first iteration, the largest number will reside at the end of the array.
* The second stage loops through the array in the opposite direction- starting from the item just before the most recently sorted item and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

1. **Step-by-step descriptions**

-         Example: **𝑎 = {4, 1, 0, 3, 2}**

We use ‘‘ | ’’ to control the sorted and unsorted element

|  |  |  |
| --- | --- | --- |
| **Step** | **Array a** | **Explain** |
| 0 | {|4, 1, 0, 3, 2|} | Unsorted |
| 1 | {|4, 1, 0, 3, 2|} | We begin with the loop from left to right, we compare two adjacent number : because 4 < 1 , so we swap 4 and 1 |
| 3 | {|1, 4, 0, 3, 2|} | 4 > 0 swap 4 and 0 |
| 4 | {|1, 0, 4, 3, 2|} | 4 < 3 swap 4 and 3 |
| 5 | {|1, 0, 3 , 4, 2|} | 4 < 2 swap 4 and 2 |
| 6 | {|1, 0,3,2 |,4} | Now we finish the first task. Next is the second task, we check the array from last to first element. |
| 7 | {|1, 0, 3, 2 |,4} | 2 < 3 we swap 2 and 3 |
| 8 | {|1, 0, 2, 3 |,4} | 0 < 2 we do nothing |
| 9 | {|1, 0, 2, 3 |,4} | 0 < 1 we swap 0 and 1 |
| 10 | {0, 1,| |2,3,4} | We got a sorted array |

1. **Complexity evaluations**

* **Time Complexity:**
* Worst and Average Case Time Complexity: O(n2).
* Best Case Time Complexity: O(n). Best case occurs when the array is already sorted.
* **Space Complexity**: O(1)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 338 | 3506 | 11323 | 38166 | 372197 | 983565 |
| Nearly Sorted | 1 | 1 | 2 | 1 | 1 | 2 |
| Sorted | 1 | 1 | 1 | 1 | 1 | 1 |
| Reversed | 408 | 3002 | 8353 | 32113 | 340792 | 740424 |

* Running time in milisecond table:

1. **Shell Sort**
2. **Ideas**

* Shell sort algorithm is an improved method of Insert Sort. The term Insert sort resentment compares and replaces elements that are close together. Here the idea of the sorting method is that the beginning of the original sequence into sequences of elements at a distance “h” from each other.

1. **Step-by-step descriptions**

* Step 1: Initialize h values
* Step 2: Split the list into smaller sublists corresponding to h
* Step 3: Sort these sublists using insertion sort (Insert sort)
* Step 4: Iterate to when the list is sorted
* Example:

**𝑎 = {20, 12, 10, 15, 2} , n = 5**

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Distance | Array a | Explain |
| 1 | h = n/2 = 2 | {**20, 12, 10, 15, 2**} | Starting with the element h = 2, we compare 20 and 10, but 20 > 10, so 10 should be permuted first |
| 2 | h = 2 | {**10, 12, 20, 15, 2**} | With h = 2 we continue to compare 12 and 15 but 12 < 15 should stay in place. "Distance" has run out of array start creating a new step |
| 3 | h = h/2 =1 | {**10, 12, 20, 15, 2**} | With h = 1 we compare 10 and 12 but 10 < 12 should stay in place. |
| 4 | h = 1 | {**10, 12, 20, 15, 2**} | With h = 1 we compare 12 and 20 but 12 < 20 should stay in place. |
| 5 | h =1 | {**10, 12, 20, 15, 2**} | With h = 1 we compare 20 and 15 but 20 > 15 (not in the right place) we swap positions of 20 and 15. |
| 6 | h = 1 | {**10, 12, 15, 20, 2**} | With h = 1 we compare 20 and 2 but 20 > 2 (not in the right place) we swap positions of 20 and 2. Here we end the array traversal with h = 1 and end the algorithm... |

1. **Complexity evaluations**

* Currently, the evaluation of Shellsort solutions leads to very complex mathematical problems, some of which have not been proven yet. However, the efficiency of the algorithm depends on the "Distance" selected. In the case of choosing the length sequence according to the formula h[i] = (h[i-1] – 1)/2 and h[k] = 1, k = log2 - 1 (where k is the length of the sequence Distance), the algorithm has an equivalent complexity of ≈ n1,2 << n2 .
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 2 | 6 | 12 | 20 | 74 | 124 |
| Nearly Sorted | 1 | 2 | 2 | 8 | 14 | 26 |
| Sorted | 1 | 2 | 3 | 8 | 14 | 27 |
| Reversed | ~0 | 3 | 5 | 9 | 22 | 36 |

1. **Counting Sort**
2. **Ideas**

* Counting sort is a sorting technique based on numbers between a specific range. It works by counting the number of elements having distinct key values. Then doing some arithmetic to calculate the position of each object in the output sequence.

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | 1 | 4 | 1 | 2 | 7 | 5 | 2 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |

* 1: Find out the maximum and minimun elements (let then be *max, min*) from the given array. In the instance, max = 7, min = 1).
* 2: Initialize an array (count) of length max + 1 with all elements 0. This array is used for storing the count of the elements in the array.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

* 3: Store the count of each element at their respective index in count array.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | 2 | 2 | 0 | 1 | 1 | 0 | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

* 4: Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | 2 | 4 | 4 | 5 | 6 | 6 | 7 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

* 5: Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.
* 6: After placing each element at its correct position, decrease its count by one.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **1** | 4 | 1 | 2 | **7** | 5 | 2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | **2** | 4 | 4 | 5 | 6 | 6 | **7** |

2 - 1

7 - 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **output** | 1 | **1** | 2 | 2 | 4 | 5 | **7** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | 1 | 4 | 1 | 2 | 7 | 5 | 2 |

1. **Complexity evaluations**

* There are mainly four main loops.

|  |  |
| --- | --- |
| **Loop** | **Complexity** |
| 1: Generate Count array | n |
| 2: Store cumulative sum of the elements of the count array | Count.size = max – min + 1 = k |
| 3: Find the index of each element of the original array in the count array | n |
| 4: Copy the output array to the origin array | n |

* **Time complexity:** Because there isn’t any compare, so:
* Average complexity: O(n + k)
* Best Case: O(n + k)
* Worst Case: O(n + k)

( k is the range of input )

* **Space Complexity**: O(n + k).
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 1 | 1 | 4 | 4 | 11 | 18 |
| Nearly Sorted | 1 | 1 | 3 | 5 | 14 | 21 |
| Sorted | 0.1 | 1 | 2 | 7 | 14 | 23 |
| Reversed | 1 | 3 | 2 | 5 | 14 | 20 |

1. **Flash Sort**
2. **Ideas**

* The idea of the algorithm is to divide the elements into 𝑚 different segments, the element 𝑎[𝑖] will be in the segment: 1 + **(m = 0.43 \* n)**
* After fission is complete, the algorithm will arrange the elements in the same segment by the insertion sort algorithm.

1. **Step-by-step descriptions**

a = {4, 1, 0, 3, 2} ; n = 5 → m = 3

→ a = {1, 0 | 3, 2 | 4}

|  |  |
| --- | --- |
| Element | Segment |
| 4 | 3 |
| 1 | 1 |
| 0 | 1 |
| 2 | 2 |
| 3 | 2 |

* In the next step, we will use insertion sort to sort each segment:

→ a = {1, 0 | 3, 2 | 4}

* At this point, the algorithm stops.

1. **Complexity evaluations**

* **Time complexity:**
* Average complexity: O(n + m)
* Best Case: O(n + m)
* Worst Case: O(n2)
* **Space Complexity**: O(m).
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 1 | 5 | 8 | 23 | 63 | 114 |
| Nearly Sorted | 2 | 5 | 9 | 15 | 43 | 53 |
| Sorted | 2 | 5 | 7 | 16 | 47 | 52 |
| Reversed | 1 | 4 | 4 | 15 | 27 | 40 |

# **IV. EXPERIMENTAL RESULT & COMMENTS**

# **Randomized data**

# **Nearly sorted data**

# **Sorted data**

# **Reverse sorted data**

**V. References**

**-** [**https://www.geeksforgeeks.org/heap-sort/**](https://www.geeksforgeeks.org/heap-sort/)

Sách Cấu trúc dữ liệu và giải thuật 2003 trường đai học khoa học tự nhiên

<https://www.geeksforgeeks.org/shellsort/>

Sách Cấu trúc dữ liệu và giải thuật 2003 trường đai học khoa học tự nhiên

<https://www.geeksforgeeks.org/radix-sort/>

Sách Cấu trúc dữ liệu và giải thuật 2003 trường đai học khoa học tự nhiên