

Data structures

**Research Papers 2**

**Insertion Algorithms**

Name : Abdulghaffar Abed

ID : 1180071

Instructor : Dr. Radi jarar

S# : 4

**Counting sort**

* Is a sort technique depend on the value ( key ) between a specified range .
* It works using counter to count the distinct key for each value in the array then do some arithmetic operation to get array with sorted elements.

**Counting Sort Algorithm :-**

1. Initialize an array of range from 0 to 9 or find the maximum element in the array and initialize array of length max+1 in both choices put all elements in the auxiliary array 0s.
2. Count each element in the given array and place the count in the appropriate index , if there is two or more identical number each time found identical number increment value in the same appropriate index .
3. Modify the count array (auxiliary array) by adding the previous counts .
4. Create new array with places equal to the number of elements in the given array .
5. we put the objects in the new array depending on the Corresponding values represent the places in the count array and decrement the count in the count array .
6. Now we have new sorted array , we are done.

* **Time complexity :** O(n + x) where n is the number of elements in the given array and x is the range of the input.
* **Space complexity :** O(max) where max is the maximum element in the given array , while the range of elements increment the space complexity also increment .

|  |  |
| --- | --- |
| Input data array | Complexity |
| Sorted ascending | O(n + x) |
| Sorted descending | O(n + x) |
| Not sorted | O(n + x) |

In all the above cases the complexity still the same because the

counting sort depend Basically on the count value .

* The counting sort is **stable** .
* The counting sort is **not in-place** .
* It’s **not** a comparison sorting
* We can extended to cover negative numbers in the counting sort .
* **Counting sort Applications :**
* When we use linear complexity
* If the given array contains smaller integers with multiple counts .

**Shell sort**

* Is a generalized version of insertion sort , the idea of shell sort is to allow exchange for far items and that reduce number of movement according to the insertion result number of movement position is the mainly difference between insertion and shell sort .
* In this sorting we compare elements that are a distance apart rather than adjacent.

**Shell Sort Algorithm :-**

1. Find gap for each pass using this mathematical equation

* for the first gap => gap = n/2
* second or more gap => gap ”k”= (gap”k-1”) \* ½

where n is the number of elements in the given array , k is the

gap number .

1. now we compare from the beginning of the given array to the next element with regard to the gap
2. if the right of the gap larger than the left we keep it in its position, Else if the right smaller than the left we shift the right to the left position and the left to the right position .
3. if the given array has even number of elements then when gap =1 we are done .
4. else if the given array has odd number of elements then when gap =1 we should compare the last element in the array with all other elements and we are done .
5. Now we have sorted array .

* Time complexity :

1. Worst case : less than or equal to O( n^2 )
2. Average case : O(n^3/2 ) around O(n\*log n)
3. Best case : O(n ^5/4 ) around O(n\*log n)

* Space complexity : The space complexity for shell sort is O(1).

|  |  |
| --- | --- |
| Input data array | Complexity |
| Sorted ascending | O(n ^5/4 ) => O(n\*log n) |
| Sorted descending | O( n^2 ) |
| Not sorted | O(n\*log n) |

* The shell sort is **unstable** because we’re sorting gaps before do the last sort .
* The shell sort **in-place** sorting algorithm.
* **Shell Sort Applications :**
* Instead of insertion sort when the close elements are far apart .
* Calling a stack is overhead .
* Recursion exceeds a limit .

**Cycle sort**

* Is a comparison sort algorithm that divide the array into cycles , we have an array with n nodes and an edge directed from node A to node B if the element in node A must be in the position of B node we put A in B position and save B node in outer variable .

**Cycle sort Algorithm :-**

1. We look to all cycles one by one .
2. Take X element index (frequently the first element in the array) by counting the number of elements that are smaller than X .
3. If there is no element smaller than X ,Then save X in its position .
4. Else if we find one or more elements smaller than X :

* Calculate the number of elements smaller than X using counter .
* Put X at the counter index ( array [counter] ) .
* Save the element in the counter index in outer variable for example ( item ) .
* One cycle is completed .
* We don’t come back to cycle starting point .

1. Repeat the above steps with item element to find the correct position .
2. Keep doing this till the array be sorted .

* **Time complexity :**

1. Worst case : O( n^2 )
2. Average case : O( n^2 )
3. Best case : O( n^2 )

* **Space complexity :** The space complexity for cycle sort is O(1) .

|  |  |
| --- | --- |
| Input data array | Complexity |
| Sorted ascending | O( n^2 ) |
| Sorted descending | O( n^2 ) |
| Not sorted | O( n^2 ) |

* The cycle sort is **unstable** .
* The cycle sort is **in-place** sorting algorithm .
* **Cycle Sort Applications :**

for situations where memory write or swap operations are costly .

## References :

**Counting sort :-**

* <https://www.geeksforgeeks.org/counting-sort/>
* <https://www.youtube.com/watch?v=7zuGmKfUt7s&feature=youtu.be>

**Shell sort :-**

* <https://www.geeksforgeeks.org/shellsort/>
* <https://www.youtube.com/watch?v=ddeLSDsYVp8&t=317s>

**Cycle sort :-**

* <https://en.wikipedia.org/wiki/Sorting_algorithm#Stability>
* <https://www.geeksforgeeks.org/cycle-sort/>
* <https://www.youtube.com/watch?v=gZNOM_yMdSQ>