1. **Selection Sort:**
2. **Idea:**

Sample array:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 1 | 2 | 0 | 6 | 4 | 3 | 7 | 8 | 5 |

First, the array will be devided into two part:

* Sorted part: Initially, it is an empty part.
* Unsorted part: all elements in the array.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 1 | 2 | 0 | 6 | 4 | 3 | 7 | 8 | 5 |

Sorted part: red Unsorted part: yellow

Next, the algorithm will find the smallest element in the unsorted part and swap its value with the first element of the unsorted part.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 1 | 2 | 0 | 6 | 4 | 3 | 7 | 8 | 5 |

Min-element: blue

After the swap:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 9 | 6 | 4 | 3 | 7 | 8 | 5 |

…

…

Finally, the algorithm will end when the unsorted part becomes empty.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

1. **Pseudocode:**

SelectionSort(array, n)

for i from 0 to n-1 do

minIndex = i

for j from i+1 to n-1 do

if array[j] < array[minIndex] then

minIndex = j

end if

end for

if minIndex != i then

swap(array[i], array[minIndex])

end if

end for

end SelectionSort

1. **Complexity analysis:**

With basic operation: key comparisons array[j] < array[minIndex]. The number of key comparisons depends only on the array size.

So, sum of number of times the basic operation is:

Therefore, the time complexity of the selection sort will be

1. **Bubble Sort:**
   1. **Idea:**

Sample array:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 1 | 2 | 0 | 6 | 4 | 3 | 7 | 8 | 5 |

Initialize a loop to iterate through each element of the array. For each element (denoted as i), iterate from the first element to the element at position n - i - 2, where n is the number of elements in the array.

For each iterated element (denoted as j), swap j and j + 1 if arr[j] > arr[j+1].

Two elements just swapped: red

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 9 | 2 | 0 | 6 | 4 | 3 | 7 | 8 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 9 | 0 | 6 | 4 | 3 | 7 | 8 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 0 | 9 | 6 | 4 | 3 | 7 | 8 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 0 | 6 | 9 | 4 | 3 | 7 | 8 | 5 |

**…**

**…**

**…**

The algorithm will terminate if a complete iteration of j in any i loop completes without any swaps.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

* 1. **Pseudocode:**

BubbleSort(arr, n)

for i from 0 to n-1 do

swapped = false

for j from 0 to n-i-2 do

if arr[j] > arr[j+1] then

swap(arr[j], arr[j+1])

swapped = true

end if

end for

if swapped = false then

break

end if

end for

end BubbleSort

* 1. **Complexity analysis:**

With basic operation: key comparisons arr[j] > arr[j+1]. The number of key operation depends only on the array size. So, sum of number of times the basic operation is:

Therefore, the time complexity of the bubble sort will be

1. **Counting Sort:**

Sample array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 1 | 2 | 4 |

* 1. **Idea:**

Determine the range limits: Find the maximum and minimum values in the input array to establish the range of values for the elements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 1 | 2 | 4 |

Max-element: red Min-element: yellow

Count occurrences: Create a counting array to store the frequency of specific element values. Each element in the counting array corresponds to the number of times that value appears in the input array.

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 1 | 1 |

Accumulate: This count variable will accumulate to determine the final position of each element in the sorted array.

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 1 | 3 | 4 | 5 |

Sorting: Traverse in reverse the original array and place each element in its appropriate position in the result array based on the counting array.

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 1 | 3 | 4 | 4 |

Result array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | 4 |

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 4 | 4 |

Result array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 2 |  | 4 |

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | 2 | 4 | 4 |

Result array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  | 2 |  | 4 |

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | 2 | 3 | 4 |

Result array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  | 2 | 3 | 4 |

Counting array:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | 1 | 3 | 4 |

Result array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 2 | 3 | 4 |

* 1. **Pseudocode:**

CountingSort(arr, n)

max\_value = findMax(arr)

min\_value = findMin(arr)

count = new Array(max\_value - min\_value + 1)

for i from 0 to max\_value - min\_value do

count[i] = 0

for i from 0 to n-1 do

count[arr[i] - min\_value]++

for i from 1 to max\_value - min\_value do

count[i] += count[i-1]

output = new Array(n)

for i from n-1 down to 0 do

output[count[arr[i] - min\_value] - 1] = arr[i]

count[arr[i] - min\_value]--

for i from 0 to n-1 do

arr[i] = output[i]

end CountingSort

* 1. **Complexity analysis:**

Basic operation: assignment and addition inside 4 loops. The number of key comparisons depends on the array size and the max value of the array. So, sum of number of times the basic operations is:

With k is the maximum element of the input array.