|  |  |
| --- | --- |
| **Sorting Algorithm** | **Presoude Code** |
| Insertions Sort | For a =2 to n  Key=A[2]  b=a-1  while b > A[b] > Key  A[b+1]=A[b]  b--  A[b+1] =key |
| Selection Sort | for a =1 to A.length  min=a  for j =a+1 to A.length  if A [j] < A [min]:  min=j  if min!= a :  Swap A[a],A[min] |
| Merge Sort | MergeSort (A ,low ,high )  if low < high:  m= ((low+high-1)/2)  MergeSort(A , low ,mid)  MergeSort(A ,mid+1 ,high)  Merge(A, low ,mid ,high)  Merge ( A , low , mid , high)  Create two array left and right and store the data  compare left and right element which one is small add in a new array after this done store data should be given to A. |
| Counting Sort | for I in range (size):          j=arr[I]          count[j]+=1        for I in range (1, k+1):          count[I]+=count[i-1]        for I in range (size-1, -1, -1):          j = arr[I]          count[j]-=1          output[count[j]] =arr[I]  First make second array that store how much time the number occur in the original array then take sum of that array and store it. After that run reverse loop and check the place of element in second array and according to that sort it in original array. After that element is sort then decrease the value of second array by one. |
| Quick Select | if left == right:          return arr[left]      pivot Index = partition (arr, left, right)      if k == pivot Index:          return arr[k-1]      else if k < pivot Index:          right = pivot Index -1          return (Quick Select (arr, left, pivot Index-1, k))        else:          left = pivot Index+1          return (Quick Select (arr, pivotIndex+1, right, k))  In this first of all find the pivot index then after that if the desire value is equal to that index, then return then number on that index. If the value is less then the pivot index then again run quick select on left part of array else run quick select on right part of array. |
| Quick Sort | Quick Sort (arr, low, high):      if low < high:          pi= partition (arr, low, high)          Quick Sort (arr, low, pi-1)          Quick Sort (arr, pi+1, high)  Again, in this first find pivot point and gather all the numbers less then pivot point into the left side of array and gather all the greater number into the right side of array and again call quick sort on both of these ends until the array is fully sort. |
| Bucket Sort | Bucket (arr, n):      bucket= [[] for f in range (n)]      output= []      for I in range (0, n-1):          bucket[(floor(n\*arr[I]))]. append(arr[I])        for I in range (0, Len (bucket)):          Insertion Sort (bucket[I])      for each in bucket:          output += each      return output  This Kind of sorting is specially use for floating points. If the number is not floating point, then make it. After that make 10 buckets from 0 to 9. Multiply each number with 10 and place it in the bucket if the bucket contains more than one number then sort that bucket with insertion sort and merge all the buckets. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Best Case**  **Time Complexity** | **Worst Case**  **Time Complexity** | **Average Case**  **Time Complexity** | **Strength of the sorting algorithm** | **Weeknesses** |
| Insertions Sort | O(n) | O(n^2) | O(n^2) | **1.** Stable algorithm means first element of duplicate come first the second duplicate element.  **2.** Perform well in small data set.  **3.** Space require is minimal. | **1.** Does not deal well for the huge list  **2.** Does not perform well as other algorithm  **3.** Useful for only small list |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | **1.** Arrange of data does not matter  **2.** Perform well in small data set  **3.** | **1.** Does not deal well for the huge list  **2.** Scanning of the whole array  **3.** unstable algorithm |
| Merge Sort | O(nlogn) | O(nlogn) | O(nlogn) | **1.**Stable algorithm means first element of duplicate come first the second duplicate element  **2.** Perform well for larger data set  **3.** | **1.** Goes through whole process if the list is sorted  **2.** Usage of memory  **3.** Slow for small data set |
| Bubble Sort | O(n) | O(n^2) | O(n^2) |  |  |
| Bucket | O(n+k) | O(n^2) | O(n+k) |  |  |
| Quick Sort | O(nlogn) | O(n^2) | O(nlogn) |  |  |
| Counting Sort | O(n+k) | O(n+k) | O(n+k) |  |  |