|  |  |
| --- | --- |
| ***Insertion Sort*** |  |
| Description | *Insertion sort is one of the simplest type of sorting as sorting is done by iterating through the array using a key value every time a single key is selected and after iterate through the whole array the exact location of that particular key is found and hence one element is sorted in the same manner the whole the array is sorted in ‘n2’ iterations where n is the number of elements in array this sorting technique is simple but not suitable for large input as loop will be large and algorithm takes more time hence this algorithm is suitable for a limited input its implementation is simple we just take a key value usually 2nd element of array and iterate it through the array and compare it with all the index either it is larger or small then the neighbour index after finding its location the left side array is sorted and after all the loop iterations all the array is sorted although its a simple algorithm but it becomes so time consuming in its worst case as n2 time for n inputs if n becomes so larger the time of compilation becomes so large so this algorithm is only good for its best case but as new algorithms are found it is assumed that it can’t performs better than the advanced algorithm so it becomes useless as the same task is performed in less time.* |
| Pseudo code | *for j=2 to A.length( )*  *key=A[j]*  *i=j-1*  *while i>0 AND A[i]>key*  *A[i+1]=A[i]*  *i=i-1*  *A[i+1]=key* |
| Python code | *def insertion\_sort(arr):*  *for j in range(1, len(arr)):*  *key = arr[j]*  *i = j-1*  *while i >= 0 and key < arr[i] :*  *arr[i + 1] = arr[i]*  *i -= 1*  *arr[i + 1] = key* |
| Time Complexity Analysis |  |
| Proof of correctness |  |
| Strengths | * *Perform well for small no of inputs (best case)* * *Implementation is simple* * *Less space is required* |
| Weakness | * *As input is large it become time effective as n2 time is required* * *Not as efficient as other advanced algorithms* * *Need to iterate to all the array again and again* |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Merge Sort*** |  |
| Description | *Merge sort is one of the most respected algorithm used in data structures it is an algorithm which uses divide and conquer rule to sort an array this sorting in worst case uses O(n log n) time the working technique is that it simply divide an array into 2 parts and keeps on doing so until array contains only one element as one element is already sorted so the merge sort algorithm keeps on dividing the array into 2 parts recursively until array size becomes one then the merge sort use a merge function which combines (conquer) all the one sized array into single array that the resulted array is fully sorted hence we say that merge sort only divide the arras and the merge function actually combines them in an sorted array the biggest factor about merge sort is that in best case, in worst case and in average case it takes the same time which is O(n log n) as this time is much more good than O(n2) so that is the one reason to use merge sort rather then insertion sort as it doesn’t depends on number of inputs it have same effect either we use it for large input or for small as recursion use O(n log n) so in case we have small input it becomes little costly as due to divide and conquer (recursion) some space is required on each call so it take more space than insertion sort as a result it is better to use merge sort algorithm for sorting large data.* |
| Pseudo code | *Merge\_Sort(A,a,b)*  *if(a==b)*  *return*  *else*  *m=floor(a+b/2)*  *Merge\_Sort(A,a,m)*  *Merge\_Sort(A,m+1,b)*  *Merge\_Sort(A,b,m)*  *return*  *merg(A,p,q,r)*  *n1=q-p+1*  *n2=r-q*  *L = [0] \* (n1)*  *R = [0] \* (n2)*  *for i=0 to n1*  *L[i] = A[p + i]*  *for j=0 to n2*  *R[j] = A[q + 1 + j]*  *i=0*  *j=0*  *k=p*  *while i < n1 and j < n2*    *if(L[i]<=R[j])*  *A[k] = L[i]*  *i=i+1*  *else*  *A[k] = R[j]*  *j=j+1*  *k=k+1*  *while i < n1*  *A[k] = L[i]*  *i += 1*  *k += 1*  *while j < n2*  *A[k] = R[j]*  *j += 1*  *k += 1* |
| Python code | *def merg(A,p,q,r):*  *n1=q-p+1*  *n2=r-q*  *L = [0] \* (n1)*  *R = [0] \* (n2)*    *for i in range (0,n1):*      *L[i] = A[p + i]*  *for j in range(0,n2):*    *R[j] = A[q + 1 + j]*  *i=0*  *j=0*  *k=p*  *while i < n1 and j < n2:*  *if(L[i]<=R[j]):*  *A[k] = L[i]*  *i=i+1*  *else:*  *A[k] = R[j]*  *j=j+1*  *k=k+1*  *while i < n1:*  *A[k] = L[i]*  *i += 1*  *k += 1*  *while j < n2:*  *A[k] = R[j]*  *j += 1*  *k += 1*  *def mergeSort(A, p, r):*  *if((p<r)):*  *q=(p+r)//2*  *mergeSort(A,p,q)*  *mergeSort(A,q+1,r)*  *merg(A,p,q,r)* |
| Time Complexity Analysis |  |
| Proof of correctness |  |
| Strengths | * *Take less time for large data as compared to other algorithms* * *Use divide and conquer technique rather than iteration* * *It is consistent in time not depends on input* |
| Weakness | * *Not efficient for small input as compared for other algorithms* * *Requires more space for temporary arrays* * *In best case ( array is sorted ) it goes to same process takes O(n log n) time* |
| Dry Run |  |