**Bubble Sort:**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

**First Pass:**

( 5 1 4 2 8 ) –> ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 5 4 2 8 ) –> ( 1 4 5 2 8 ), Swap since 5 > 4

( 1 4 5 2 8 ) –> ( 1 4 2 5 8 ), Swap since 5 > 2

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )

( 1 4 2 5 8 ) –> ( 1 2 4 5 8 ), Swap since 4 > 2

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

**Third Pass:**

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

**The Speciality Of Bubble Sort:**

void bubbleSort(int arr[], int n)

{

int i, j;

for (i = 0; i < n-1; i++)

**// Last i elements are already in place**

for (j = 0; j < n-i-1; j++)

{

if (arr[j] > arr[j+1])

{

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

}

}

}

**Worst and Average Case Time Complexity:** O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

**Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

**Sorting In Place:** Yes

**Stable:** Yes

**Average Case Time Complexity:**Check the number of iterations inner loop is making:

**(n-1) + (n-2) + (n-3) + ... + 1 times. So it is O(n + (n-1) + (n-2) + (n-3) + ... + 1) = O(n(n+1)/2) = O(n^2)**

**Insertion Sort:**

**Algorithm**

// Sort an arr[] of size n

insertionSort(arr, n)

Loop from i = 1 to n-1.

……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

void insertionSort(int arr[], int n)

{

int i, key, j;

for (i = 1; i < n; i++)

{

key = arr[i];

j = i-1;

**/\* Move elements of arr[0..i-1], that are**

**greater than key, to one position ahead**

**of their current position \*/**

while (j >= 0 && arr[j] > key)

{

arr[j+1] = arr[j];

j = j-1;

}

**/\* Shifting is being done. Since, arr[j]> key that means Now, key will be placed somewhere between 0 to j index. So, arr[j] for current j needs to be right shifted\*/**

arr[j+1] = key;

}

**Time Complexity:**

**Insertion Sort Worst Case and Average Case Time Complexity: **

**An Insertion Sort Time Complexity Based Question:**

Question : How much time Insertion sort takes to sort an array of size n in below form?

arr[] = 2, 1, 4, 3, 6, 5,….i, i-1, …..n, n-1

Answer : At first look, it seems like Insertion Sort would take O(n2) time, but it actually takes O(n) time

How? Let us take a closer look at below code.

Check the algorithm:

**/\* Function to sort an array using insertion sort\*/**

void insertionSort(int arr[], int n)

{

for (int i = 1; i < n; i++)

{

int key = arr[i];

int j = i-1;

/\* Move elements of arr[0..i-1], that are

greater than key, to one position ahead

of their current position \*/

while (j >= 0 && arr[j] > key)

{

arr[j+1] = arr[j];

j = j-1;

}

arr[j+1] = key;

}

}

If we analyze the input carefully we see that every element is only one position away from its position in sorted array. The outer for loop will run till ‘n’ and the inner while loop would take “constant” steps of 1 swap and 2 comparisons. Since, while loop takes constant time and for loop runs for ‘n’ element, so overall complexity is O(n)

**Another Hint:** Time taken by insertion sort is proportional to the number of inversions in the array.

In above example type, number of inversions is n/2, so overall time complexity is O(n)

**Time complexity of insertion sort when there are O(n) inversions?**

Given an array arr[], a pair arr[i] and arr[j] forms an inversion if arr[i] < arr[j] and i > j. For example, the array {1, 3, 2, 5} has one inversion (3, 2) and array {5, 4, 3} has inversions (5, 4), (5, 3) and (4, 3)

Now, consider the insertion sort function again:

void insertionSort(int \*arr,int n)

{

int key,j;

for(int i=0;i<n;i++)

{

key=arr[i];

j=i-1;

while(j>0&&arr[j]>key)

//arr[j]>key

//will make the sort stable

{

arr[j+1]=arr[j];

//right shifting by one position

j--;

}

arr[j+1]=key;

}

}

If we take a closer look at the insertion sort code, we can notice that every iteration of while loop reduces one inversion. The while loop executes only if i > j and arr[i] < arr[j]. Therefore total number of while loop iterations (For all values of i) is same as number of inversions. Therefore overall time complexity of the insertion sort is O(n + f(n)) where f(n) is inversion count. If the inversion count is O(n), then the time complexity of insertion sort is O(n). In worst case, there can be n\*(n-1)/2 inversions. The worst case occurs when the array is sorted in reverse order. So the worst case time complexity of insertion sort is O(n2).

**QuickSort:**

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

**Always pick first element as pivot.**

**Always pick last element as pivot (implemented below)**

**Pick a random element as pivot.**

**Pick median as pivot.**

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

**Quick Sort Time Complexity:**

**Analysis of QuickSort**

Time taken by QuickSort in general can be written as following.

**T(n) = T(k) + T(n-k-1) + theta(n)**

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

**Worst Case:** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + theta(n)

which is equivalent to

T(n) = T(n-1) + theta(n)

The solution of above recurrence is theta(n2).

**Best Case:** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + theta(n)

The solution of above recurrence is theta(nLogn). It can be solved using case 2 of Master Theorem.

**Quick sort:**

Why Quick Sort is preferred over Merge Sort for sorting Arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

**Tail Quick Sort:**

In QuickSort, partition function is in-place, but we need extra space for recursive function calls. A simple implementation of QuickSort makes two calls to itself and in worst case requires O(n) space on function call stack.

The worst case happens when the selected pivot always divides the array such that one part has 0 elements and other part has n-1 elements.

**Can we reduce the auxiliary space for function call stack?**

We can limit the auxiliary space to O(Log n). The idea is based on tail call elimination. As seen in the previous post, we can convert the code so that it makes one recursive call. For example, in the below code, we have converted the above code to use a while loop and have reduced the number of recursive calls.

**void quicksort(int \*arr,int l,int r)**

{

if(arr==NULL)

{

printf("The arr passed as pointer to this function is NULL\n");

exit(EXIT\_FAILURE);

}

if(l>r)

{

return;

}

while(l<=r)

{

int pivot\_index=return\_partition\_index(arr,l,r);

quicksort(arr,l,pivot\_index-1);

l=pivot\_index+1;

}

}

however, Although we have reduced number of recursive calls, the above code can still use O(n) auxiliary space in worst case. In worst case, it is possible that array is divided in a way that the first part always has n-1 elements. For example, this may happen when last element is chooses as pivot and array is sorted in decreasing order.

**We can optimize the above code to make a recursive call only for the smaller part after partition. Below is implementation of this idea.**

Although we have reduced number of recursive calls, the above code can still use O(n) auxiliary space in worst case. In worst case, it is possible that array is divided in a way that the first part always has n-1 elements. For example, this may happen when last element is chooses as pivot and array is sorted in decreasing order.

/\* This QuickSort requires O(Log n) auxiliary space in

worst case. \*/

void quickSort(int arr[], int low, int high)

{

while (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

int pi = partition(arr, low, high);

// If left part is smaller, then recur for left

// part and handle right part iteratively

if (pi - low < high - pi)

{

quickSort(arr, low, pi - 1);

low = pi + 1;

}

// Else recur for right part

else

{

quickSort(arr, pi + 1, high);

high = pi - 1;

}

}

}

**Hoare’s vs Lomuto partition scheme in QuickSort:  
  
Lomuto’s Partition Scheme**

partition(arr[], lo, hi)

pivot = arr[hi]

i = lo // place for swapping

for j := lo to hi – 1 do

if arr[j] <= pivot then

swap arr[i] with arr[j]

i = i + 1

swap arr[i] with arr[hi]

return I

**Implementation Of This Is Following:**

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

**Hoare’s Partition Scheme:**

partition(arr[], lo, hi)

pivot = arr[lo]

i = lo - 1 // Initialize left index

j = hi + 1 // Initialize right index

// Find a value in left side greater

// than pivot

do

i = i + 1

while arr[i] < pivot

// Find a value in right side smaller

// than pivot

do

j = j - 1

while arr[i] > pivot

if i >= j then

return j

swap arr[i] with arr[j]

int partition(int arr[], int low, int high)

{

int pivot = arr[low];

int i = low - 1, j = high + 1;

while (true)

{

// Find leftmost element greater than

// or equal to pivot

do

{

i++;

} while (arr[i] < pivot);

// Find rightmost element smaller than

// or equal to pivot

do

{

j--;

} while (arr[j] > pivot);

// If two pointers met.

if (i >= j)

return j;

swap(arr[i], arr[j]);

}

}

Hoare’s scheme is more efficient than Lomuto’s partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.

Like Lomuto’s partition scheme, Hoare partitioning also causes Quicksort to degrade to O(n^2) when the input array is already sorted, it also doesn’t produce a stable sort.

Note that in this scheme, the pivot’s final location is not necessarily at the index that was returned, and the next two segments that the main algorithm recurs on are (lo..p) and (p+1..hi) as opposed to (lo..p-1) and (p+1..hi) as in Lomuto’s scheme.

I.e. So, Hoare’s partition scheme calls quicksort recursively for lo..p and p+1…hi

And,Lomuto’s for (lo..p-1) and (p+1..hi)

**Why Merge Sort Is Preferred Over Quick Sort In Case Of Linked List:**

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

**Merge Sort:**

I remember the solution. I remember the recurrence relation. From recurrence relation, I can prove the time complexity using Master’s algorithm.

**Heap Sort:**

void MaxHeap::max\_heapify(int i)

{

int largest=arr[i];

int largest\_index=i;

if(2\*i+1<size && arr[2\*i+1]>largest)

{

largest=arr[2\*i+1];

largest\_index=2\*i+1;

}

if(2\*i+2<size && arr[2\*i+2]>largest)

{

largest=arr[2\*i+2];

largest\_index=2\*i+2;

}

if(largest!=arr[i])

{

int temp=arr[i];

arr[i]=arr[largest\_index];

arr[largest\_index]=temp;

max\_heapify(largest\_index);

}

}

void MaxHeap::build\_max\_heap()

{

int index=(size-1)>>1;

//last level nodes already follow the max heap property

for(;index>=0;index--)

{

max\_heapify(index);

}

//this I know

}

void MaxHeap::heap\_sort()

{

build\_max\_heap();

int sizeval=size;

for(int i=size-1;i>=1;i--)

{

//now, after build\_heapify the maximum element is at the root

swap(arr[0],arr[i]);

size--;

max\_heapify(0);

}

size=sizeval;

}

First, build\_max\_heap should be done. After that, heap sort will be called. Otherwise, heap\_sort can internally first calls build\_max\_heap()

The HEAPSORT procedure takes time O(n lg n), since the call to BUILD-MAXHEAP

takes time O(n) and each of the n 1 calls to MAX-HEAPIFY takes

time O(lg n)

**Where Is Heap Sort Used Practically?**

Although QuickSort works better in practice, the advantage of HeapSort worst case upper bound of O(nLogn).

MergeSort also has upper bound as O(nLogn) and works better in practice when compared to HeapSort. But MergeSort requires O(n) extra space

HeapSort is not used much in practice, but can be useful in real time (or time bound where QuickSort doesn’t fit) embedded systems where less space is available (MergeSort doesn’t fit)

**Selection Sort:**

/\* a[0] to a[n-1] is the array to sort \*/

int i,j;

int n;

/\* advance the position through the entire array \*/

/\* (could do j < n-1 because single element is also min element) \*/

for (j = 0; j < n-1; j++)

{

/\* find the min element in the unsorted a[j .. n-1] \*/

/\* assume the min is the first element \*/

int iMin = a[j];

/\* test against elements after j to find the smallest \*/

for (i = j+1; i < n; i++)

{

/\* if this element is less, then it is the new minimum \*/

if (a[i] < a[iMin])

{

/\* found new minimum; remember its index \*/

iMin = i;

}

}

if (iMin != j)

{

swap(a[j], a[iMin]);

}

}

**Which Sort Makes Less Number Of Writes In Memory? (Among the known ones, selection sort**

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with EEPROMs or Flash memory, where each write reduces the lifespan of the memory.

Among the sorting algorithms that we generally study in our data structure and algorithm courses, Selection Sort makes least number of writes (it makes O(n) swaps). But, Cycle Sort almost always makes less number of writes compared to Selection Sort. In Cycle Sort, each value is either written zero times, if it’s already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

**Counting Sort:**

// C Program for counting sort

#include <stdio.h>

#include <string.h>

#define RANGE 255

// The main function that sort the given string arr[] in

// alphabatical order

void countSort(char arr[])

{

// The output character array that will have sorted arr

char output[strlen(arr)];

// Create a count array to store count of inidividul

// characters and initialize count array as 0

int count[RANGE + 1], i;

memset(count, 0, sizeof(count));

//this will take θ(k) where K is range+1

// Store count of each character

for(i = 0; arr[i]; ++i)

++count[arr[i]];

//this will take θ(n)

// Change count[i] so that count[i] now contains actual

// position of this character in output array

for (i = 1; i <= RANGE; ++i)

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

///this will take θ(k) where K is range+1 (actually, here, count[0] is not so important

// Build the output character array

for (i = 0; arr[i]; ++i)

{

output[count[arr[i]]-1] = arr[i];

**//cumulative counting**

**//0 indexing**

--count[arr[i]];

}

//final loop again takes θ(n)

// Copy the output array to arr, so that arr now

// contains sorted0 characters

for (i = 0; arr[i]; ++i)

arr[i] = output[i];

}

So, θ(n+k) combined. Now, Since, we usually do counting sort when k=n. An important property of counting sort is that it is stable: numbers with the same

value appear in the output array in the same order as they do in the input array. That

is, it breaks ties between two numbers by the rule that whichever number appears

first in the input array appears first in the output array. Normally, the property of

stability is important only when satellite data are carried around with the element

being sorted. Counting sort’s stability is important for another reason: counting

sort is often used as a subroutine in radix sort. As we shall see in the next section,

in order for radix sort to work correctly, counting sort must be stable.

**Radix Sort:**

RADIX-SORT: (A, d)

1 for i =1 to d

2 use a stable sort to sort array A on digit i

**// Driver program to test above function**

int main()

{

char arr[] = "geeksforgeeks";//"applepp";

countSort(arr);

printf("Sorted character array is %sn", arr);

return 0;

}

**This is easy.**

//but, cumulative count maintaining is important.

**Bucket Sort:**

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.

Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.

Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.

The idea is to use bucket sort. Following is bucket algorithm.

**bucketSort(arr[], n)**

**1) Create n empty buckets (Or lists).**

**2) Do following for every array element arr[i].**

**.......a) Insert arr[i] into bucket[n\*array[i]]**

**3) Sort individual buckets using insertion sort.**

**4) Concatenate all sorted buckets.**

void bucketSort(float arr[], int n)

{

**// 1) Create n empty buckets**

vector<float> b[n];

**// 2) Put array elements in different buckets**

for (int i=0; i<n; i++)

{

int bi = n\*arr[i]; // Index in bucket

b[bi].push\_back(arr[i]);

}

**// 3) Sort individual buckets**

for (int i=0; i<n; i++)

sort(b[i].begin(), b[i].end());

**// 4) Concatenate all buckets into arr[]**

int index = 0;

for (int i = 0; i < n; i++)

for (int j = 0; j < b[i].size(); j++)

arr[index++] = b[i][j];

}

**Time Complexity:**Since, the insertion sort runs in a quadratic time, the running time of bucket sort is:



We now analyze the average case running time of bucket sort, by computing the expected value of running time, (and how are we going to take the expectation over the input distribution. Taking expectations of both time and using linearity of expectation), we have:

 (expectation function has a different bracket)

Now, Using linearity of Expectation, this becomes:



Or,



We claim that, 

For I=0,1,……n-1. It is no surprise that each bucket I has the same value of .

Since, each value in the input array A is equally likely to fall in any bucket. To prove equation , we define indicator random variables:

falls in bucket i]

For i=0,1,2,…..n-1 and j=1,2,…n

Thus, 

To compute the , we need to expand the square and regroup terms:



=

=

=

Where, the last line follows by linearity of expectation, We evaluate the two summations separately. Indicator random variable Xij is 1 with probability and 0 otherwise. And therefore:



=

Where, , the variables  and Xik are independent

Hence,

 (because, )

=

Now, replace those values in the equation.



=

=

=2-

So,  (proved)