Lecture Notes on

Design & Analysis of Algorithms

G P Raja Sekhar
Department of Mathematics
I I T Kharagpur

Acknowledgements :

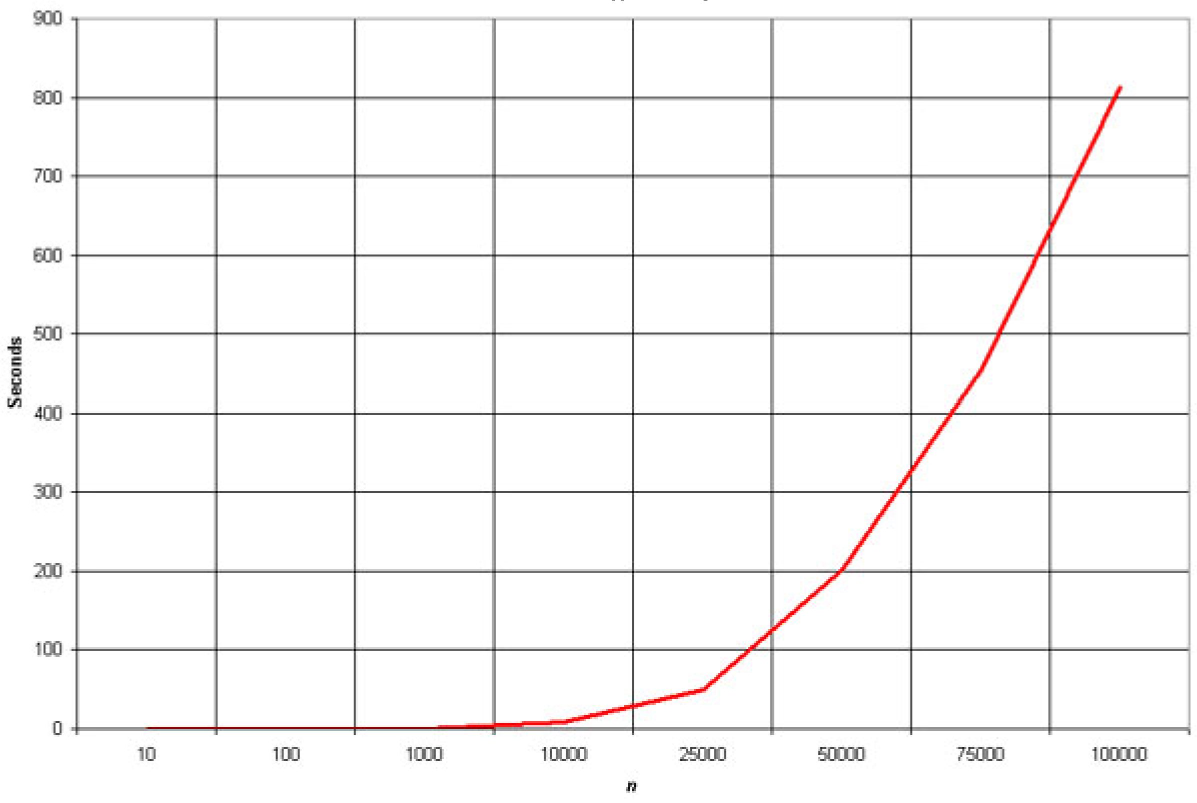
Ahmad Abdullah, Pronoy Sikdar, Anshul Kamra, Sugam Agrawal
& Students of DAA course 2006-07, Data Structures & Algorithms 2006-07.

BUBBLE SORT

The bubble sort is the oldest and simplest sort in use. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

Pros:
1) Simplicity and ease of implementation.
2) Auxiliary Space used is O (1).
Cons:
1) Very inefficient. General complexity is O (n^2). Best case complexity is O(n).

Bubble Sort Efficiency

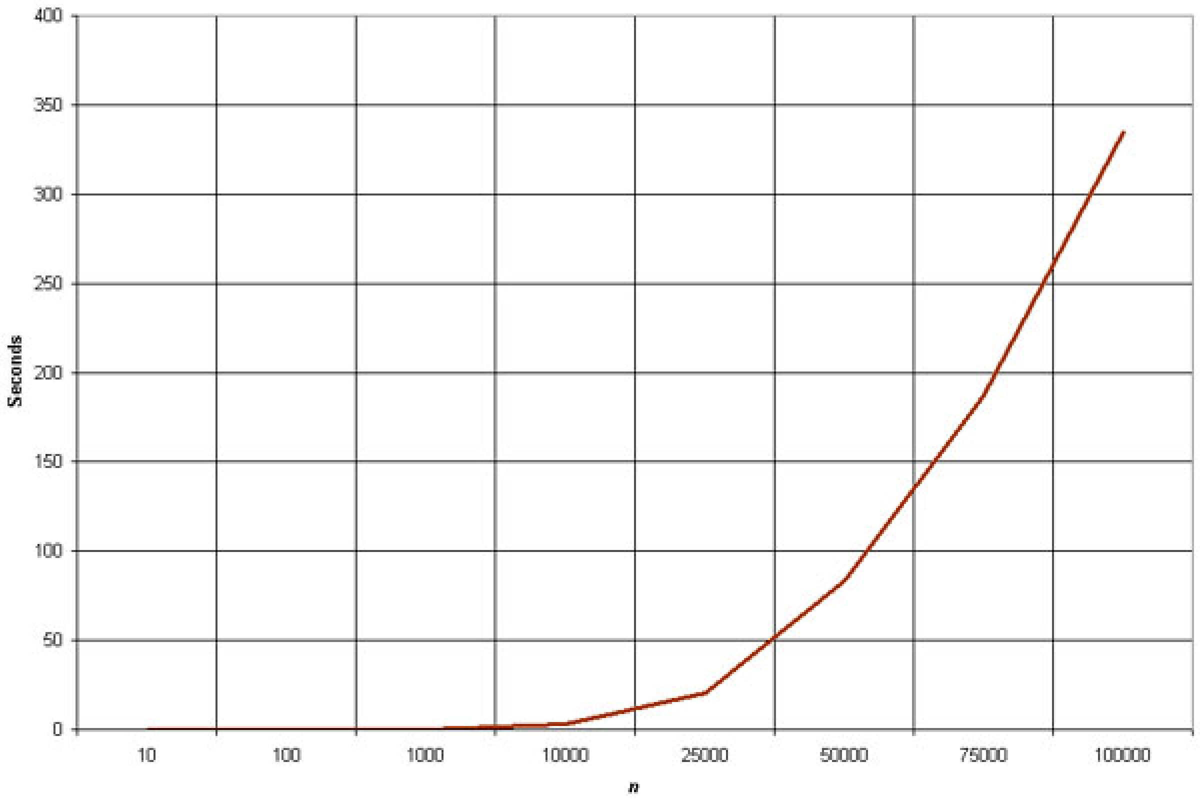


INSERTION SORT

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Pros: Auxiliary space used is O (1).
Cons: General Complexity is O (n^2). Best Case is O(n) when the list is already sorted.

Insertion Sort Efficiency



HEAP SORT

All elements to be sorted are inserted into a heap, and the heap organizes the elements added to it in such a way that either the largest value (in a max-heap) or the smallest value (in a min-heap) can be quickly extracted. Moreover, because this operation preserves the heap's structure, the largest/smallest value can be repeatedly extracted until none remain. Each time we delete (extract) the maximum, we place it in the last location of the array not yet occupied, and use the remaining prefix of the array as a heap holding the remaining unsorted elements. This gives us the elements in order.

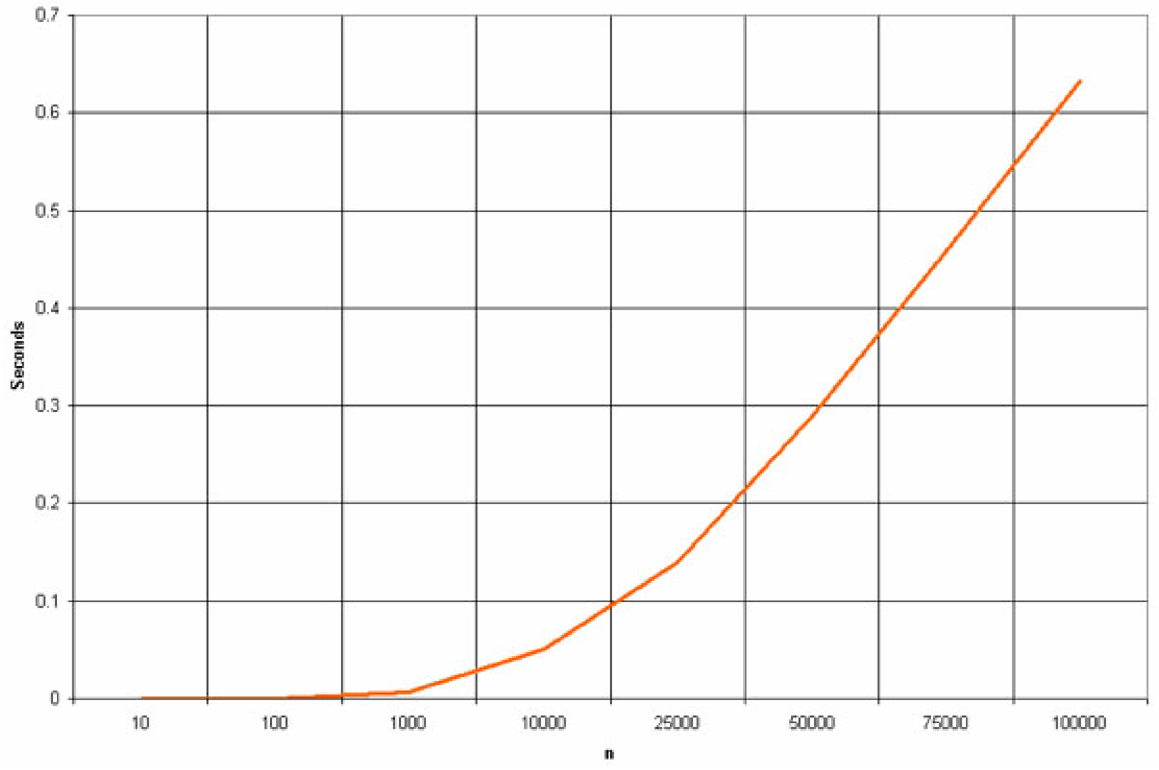
Pros:

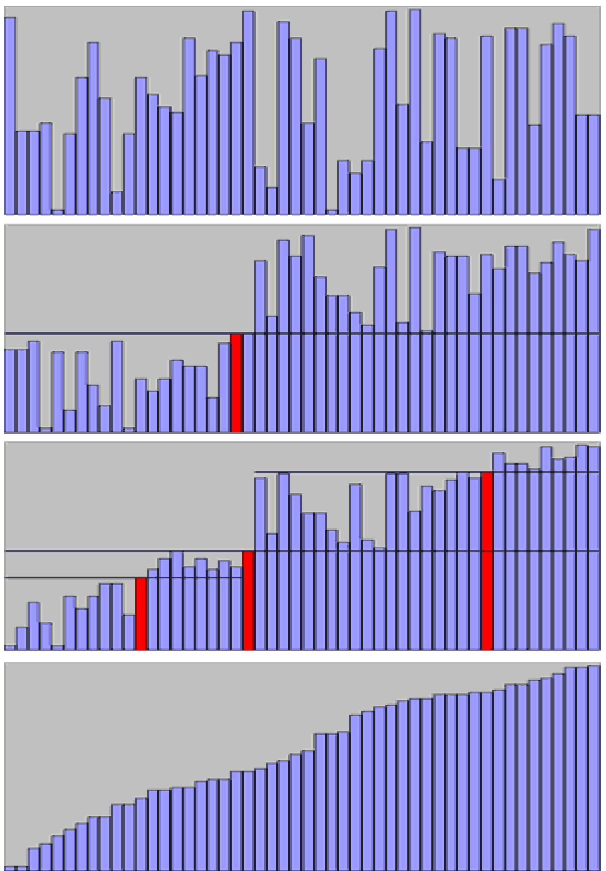
1) Time complexity of the algorithm is O (n log n).
2) Auxiliary Space required for the algorithm is O (1).
3) In-space and non-recursive makes it a good choice for large data sets.

Cons:

1) Works slow than other such DIVIDE-AND-CONQUER sorts that also have the same O (n log n) time complexity due to cache behavior and other factors.
2) Unable to work when dealing with linked lists due to non convertibility of linked lists to heap structure.

<u>Efficiency Of Heap Sort</u>

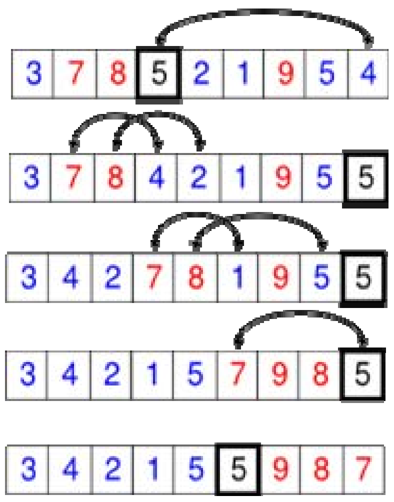




Quick Sort

Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is O (n log n).

(Quick sort in action on a list of random numbers. The horizontal lines are pivot values.)

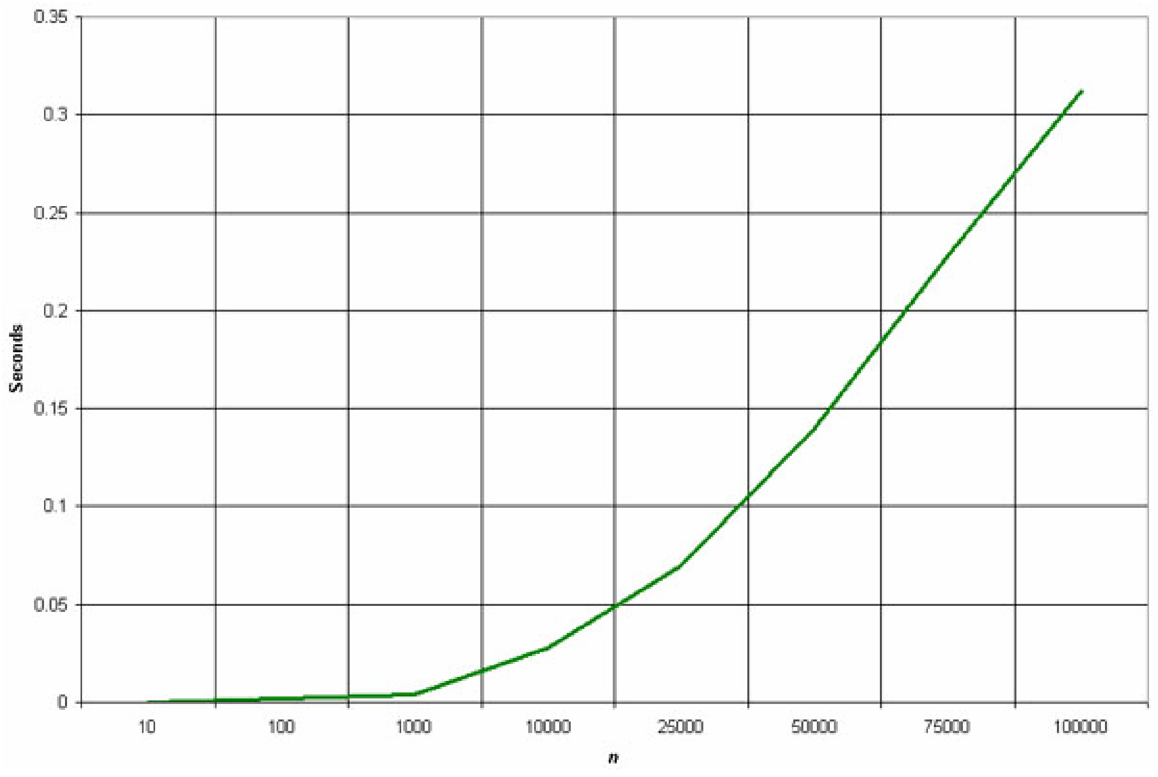

(In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are large.)

.

Pros:
1) One advantage of parallel quick sort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sub list is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.
2) All comparisons are being done with a single pivot value, which can be stored in a register.
3) The list is being traversed sequentially, which produces very good locality of reference and cache behavior for arrays.

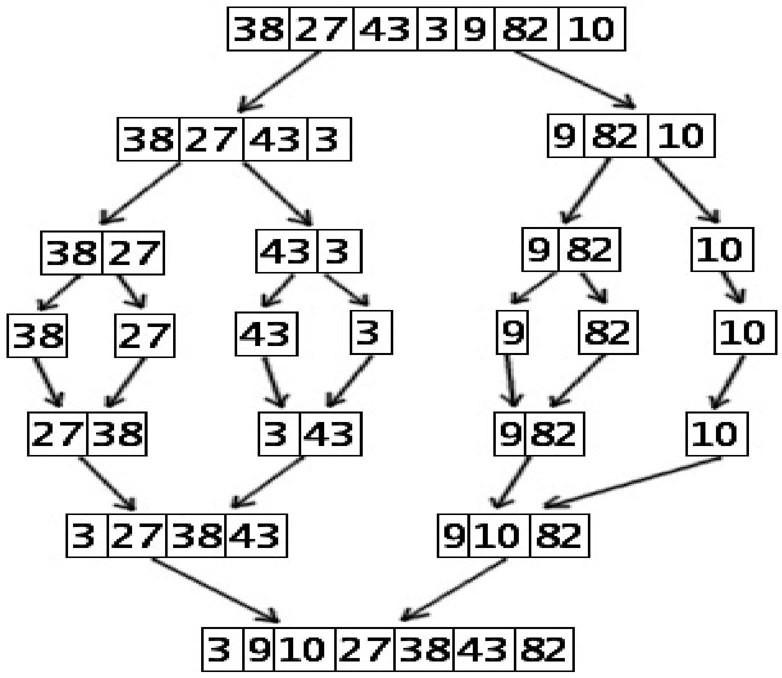
Cons:
1) Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.
2) Its worst case has a time complexity of O (n^2) which can prove very fatal for large data sets. Competitive sorting algorithms

A graphical plot of the time complexity of the Quick Sort gives a pictorial overview about the efficiency of the algorithm. The plot is as follows


0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Seconds
10
100
1000
10000
25000
50000
75000
100000
n

<u>MERGE SORT</u>

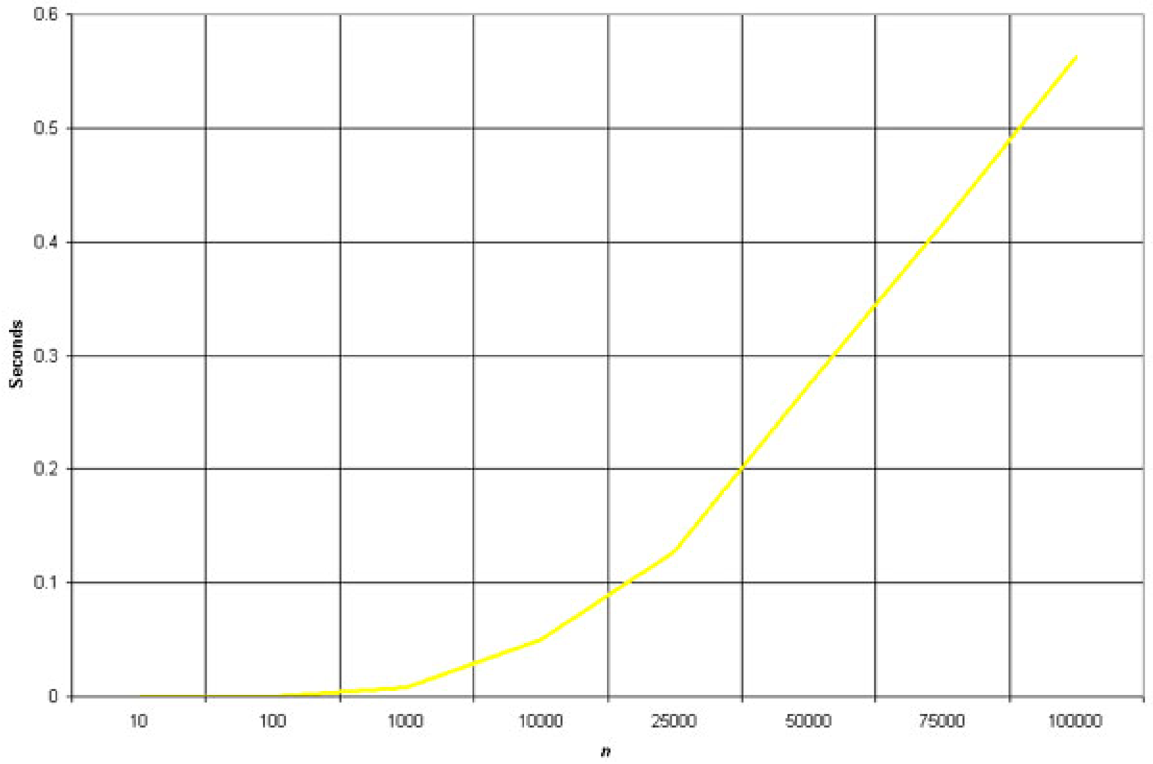
The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of *O* (*n* log *n*). Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort.



Pros:
1) Marginally faster than the heap sort for larger sets.
2) Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.
3) Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Cons:
1) At least twice the memory requirements of the other sorts because it is recursive. This is the BIGGEST cause for concern as its space complexity is very high. It requires about a Θ (n) auxiliary space for its working.
2) Function overhead calls (2n-1) are much more than those for quick sort (n). This causes it to take more time marginally to sort the input data.

A graphical plot of the time complexity of the Merge Sort gives a pictorial overview about the efficiency of the algorithm. The plot is as follows



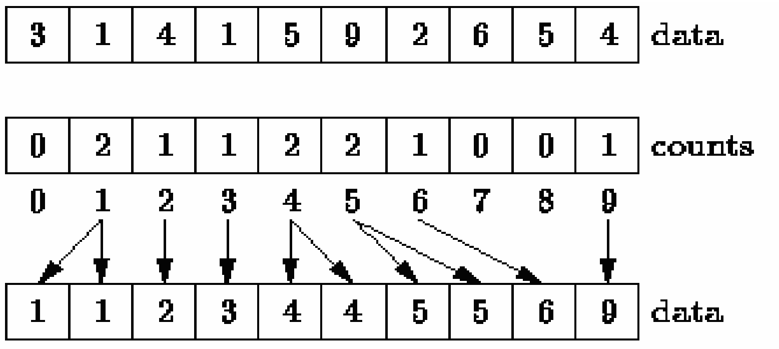
Thus we can see that the time taken to sort data of increasing instance sizes increases very slowly hence until around 20000 - 25000, and then increases at an increasing rate. So it is better to use it under the size limit of 20000 - 25000 numbers.

But usage of about Θ (n) auxiliary space for its working will charge us huge space for such high count of numbers. So depending on the restriction imposed by the availability of free space we will have to consider the input range for which we could apply this sort. For e.g. it will take around 100 - 200 kB of space to sort data of the size of 20000 numbers (considering that each number uses about 4 bytes of memory).

COUNTING SORT

Counting sort is an algorithm used to sort data whose range is pre-specified and multiple occurrences of the data are encountered. It is possibly the simplest sorting algorithm. The essential requirement is that the range of the data set from which the elements to be sorted are drawn, is small compared to the size of the data set.

For example, suppose that we are sorting elements drawn from **{0, 1... m-1}**, i.e., the set of integers in the interval **[0, *m*-1]**. Then the sort uses ***m*** counters. The ith counter keeps track of the number of occurrences of the ith element of the universe. The figure below illustrates how this is done.



Counting Sort

In the figure above, the universal set is assumed to be **{0, 1… 9}**. Therefore, ten counters are required - one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. E.g., the sorted sequence in the above example contains no zeroes, two ones, one two, and so on.

Algorithm :

```
CountingSort (Elements array[], int array_size)
Elements count[array_size];
for(i =0; i < array_size; i++)
        count[i] = 0;
for(j =0; j < array_size; j++)
        ++count[array[j]];
for(i =0, j=0; i < array_size; i++)
        for(; count[i]>0; --count[i])
                array[j++] = i;
```

Pros:

1) The algorithm has a time complexity of **<u>O (n+m)</u>**, where n is the number of data while m is the range of the data, which implies that the most efficient use will be when **<u>m<<n</u>**. In that case the time complexity will turn out to be linear.
2) This sort works optimally in the case when the data is uniformly distributed.

Cons:

1) If the range m>>n, the complexity will not be linear in n and thus this sort does not remain useful anymore. This is because chances of introduction of gaps, that is counters for those elements which do not exist in the list, will cause a higher space complexity.

<u>Some Examples Of Efficient Use of Counting Sort :</u>

1) Ranking 300 hundred students on the basis of their score out of 50.
2) Sorting 1000 people with respect to the occurrence of their birthday in a year

Radix Sort

A **radix sort** is an algorithm that can rearrange integer representations based on the processing of individual digits in such a way that the integer representations are eventually in either ascending or descending order. Integer representations can be used to represent things such as strings of characters (names of people, places, things, the words and characters, dates, etc.) and floating point numbers as well as integers. So, anything which can be represented as an ordered sequence of integer representations can be rearranged to be in order by a radix sort.

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are:

1. least significant digit (LSD) radix sorts and
2. most significant digit (MSD) radix sorts.

LSD radix sorts process the integer representations starting from the least significant digit and move the processing towards the most significant digit

MSD radix sorts process the integer representations starting from the most significant digit and move the processing towards the least significant digit.

The integer representations that are processed by sorting algorithms are often called "keys," which can exist all by themselves or be associated with other data. LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as b, c, d, e, f, g, h, i, j, ba would be lexicographically sorted as b, ba, c, d, e, f, g, h, i, j.

If lexicographic ordering is used to sort variable-length integer representations, then the representations of the numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key for the purpose of determining sorted order.

EXAMPLE1:

Here we can sort binary numbers also. Consider a group of 4 bit binary numbers. The list is given by :

1001, 0010, 1101, 0001, 1110

STEP 1:

1st Arrange the list of numbers according to the least significant bit. The sorted list is given by:

0010, 1110, 1001, 1101, 0001

STEP2:

Then arrange the list of numbers according to the next significant bit. The sorted list is given by:

1001, 1101, 0001, 0010, 1110

STEP3:

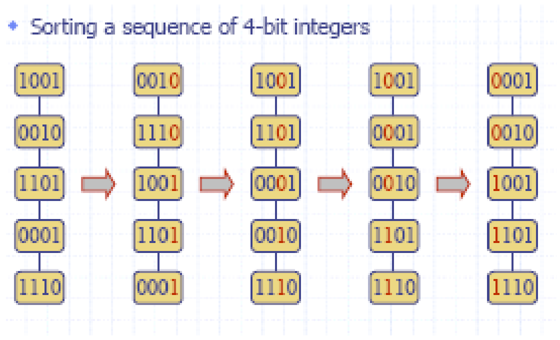
Then arrange the list of numbers according to the 2nd significant bit. The sorted list is given by:

1001, 0001,0010, 1101, 1110

STEP4:

Then arrange the list of numbers according to the most significant bit. The sorted list is given by:

0001, 0010, 1001, 1101, 1110



EXAMPLE2:

We can apply radix sort for decimal numbers also. Consider a group of numbers. It is given by the list:

523, 153, 088, 554, 235

STEP1:

Sort the list of numbers according to the ascending order of least significant bit. The sorted list is given by:

523, 153, 554, 235, 088

STEP2:

Then sort the list of numbers according to the ascending order of 1st significant bit. The sorted list is given by:

523, 235, 153, 554, 088

STEP3:

Then sort the list of numbers according to the ascending order of most significant bit. The sorted list is given by:

088, 153, 235, 523, 554

EXAMPLE 3:

Similarly consider another example. We are having a group of 7 elements . In the 1st pass the numbers are arranged according to the least significant bit. In the 2nd pass the numbers are arranged according to the next significant bit. In the 3rd pass the numbers are arranged according to the most significant bit. After the 3rd pass we get the list of numbers arranged in ascending order.

INPUT	**1st pass**	**2nd pass**	**3rd pass**
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Example for MSD radix sort:

Here we consider the most significant bit 1st. Arrange the list according to the most significant bit. Consider a list of numbers.

170, 045, 075, 090, 002, 024, 802, 066

1. Sort the list by most significant digit .It gives:

   045, 075, 090, 002, 024, 066, 170, 802

2. Then sort the list by the next significant digit (10s place):

   002, 802, 024, 045, 066, 075, 170, 090

3. Then sort the list by least significant digit (1s place) :

002, 024, 045, 066, 075, 090, 170, 802

ALGORITHM:

RADIX-SORT (*A* ,*d*)

```
1) for i ← 1 to d;
2) do use a stable sort to sort Array A on digit i // counting sort will do the job//
```

Complexity Analysis:

The running time depends on the stable sort used as an intermediate stage of the sorting algorithm. When each digits is in the range 1 to k, and k is not too large, counting sort can be taken. In case of counting sort, each pass over n d-digit numbers takes O ($n + k$) time. There are d passes, so the total time for radix sort is $\Theta(n + k)$ time. There are d passes, so the total time for Radix sort is $\Theta(d (n+ k))$. When d is constant and $k = \Theta(n)$, the Radix sort runs in linear time.

Disadvantages

Speed of radix sort largely depends on the inner basic operations and if operations are not efficient enough radix sort can be slower than some other algorithms such as quick sort or merge sort. These operations include the insert delete function of the sublists and the process of isolating the digit we want.

Radix Sort can also take up more space than other sorting algorithms, since in addition to the array that will be sorted; we need to have a sublist for each of the possible digits or letters. If pure English words are sorted then atleast 26 sublists are needed and if alpha numeric words are sorted then probably more than 40 sublists are required.

Radix Sort is also less flexible as compared to other sorts.

BUCKET SORT

Bucket Sort is a sorting method that subdivides the given data into various buckets depending on certain characteristic order, thus partially sorting them in the first go. Then depending on the number of entities in each bucket, it employs either bucket sort again or some other ad hoc sort. Bucket sort runs in linear time on an average. It assumes that the input is generated by a random process that distributes elements uniformly over the interval 1 to m.

Algorithm:

```
BUCKET_SORT (A)

1. n ← length [A]
2. For i = 1 to n do
3.     Insert A[i] into list B[A[i]/b]    where b is the bucket size
4. For i = 0 to n-1 do
5.     Sort list B with Insertion sort
6. Concatenate the lists B[0], B[1], . . B[n-1] together in order.
```

Assumptions :

- **The data lies in a range R with d digits (d ~ log R ~ n)** which is of the order of n or lesser.
- **The data is uniformly distributed** throughout the range, i.e. empty buckets are rare and more or less the same numbers of entities fall in each bucket.
- **The data is discreet**. For e.g. integers, a mixture of integers and numbers with decimals would again not be feasible as it would increase the number of digits.

Complexity : O(nd)

This is because the comparisons are being done n times at each of the‘d’ levels that the data has to pass before getting sorted. Thus the overall complexity is nd. Thus we can mathematically derive that if n.d < n log n, then
=> d< log n => log (range) < log n => range<=n

SUGGESTED IMPROVISATIONS WHEN THE DATA SET IS VERY LARGE

A normal bucket sort uses the decimal system as its base … ie it sorts the data first by the dth digit from the left into 10 bucket . Then it sorts numbers in each bucket by the d-1 th digit and so on . Thus the total complexity is O(nXd)
But suppose we are sorting very large amounts of data … eg n ~ 10^32 and range ~ 10^30 (31 digits)

Then a normal bucket sort would go this way

Suppose a number is represented as ->

A= x1*10^30 + x2*10^29 + ……. +x30*10^1+x31*10^0
B= ….
C=…..

What bucket sort does is it sorts the data by x1 in the first sweep , by x2 in the second sweep and so on total in 31 sweeps making the complexity O(31N) . total space usage is of the order 10^31

IMPROVISATION

Instead of using the decimal system we could use a higher order system like 10^5
Ie range 's of the order 100000^6 ie 7 digits only !!

So each number would then be represented as

A=a*100000^6+ b*100000^5 + ….. + g*100000^0
B= ….
C= ….

Here thus we would be using 100000 buckets instead of 10 in the normal bucket sort !!

Time complexity would thus be reduced to O(n*d1) = O (7n) from)(31n) in this example !!

Thus for such a large N we've reduced the time complexity by approximately 5 times !!

So this would enable us to work with large data sets whose range R can be even slightly greater than n

Space requirements however would increase in this case and would be of the order 10^35

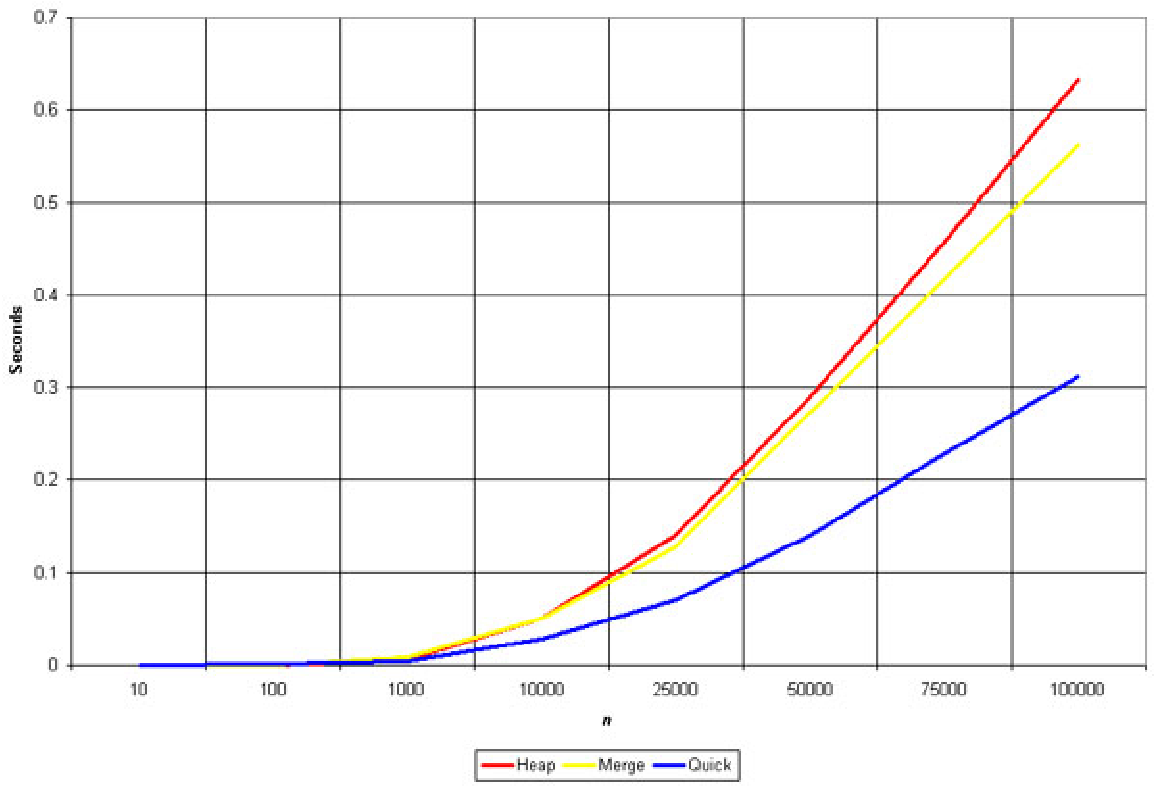
So the problem here lies in optimizing the space complexity Vs time complexity for a system with given requirements !!

DRAWBACKS

For a non uniform distribution this fails to be so efficient because the empty buckets would unnecessarily cause slowdown.

Solution ➔ For large data sets with non uniform distribution, find the ranges with maximum density (again above a particular threshold) and apply bucket sort there. For rest of the parts use some other algorithm.

COMPARATIVE ANALYSIS OF VARIOUS SORTING TECHNIQUES :



Analysis of different comparison

Name	**Time**	**Complexity**		
	best	**average**	**worst**	**Space Complexity**
bubble	O(n)	-	O(n^2)	O(n)
insertion	O(n)	O(n^2)	O(n^2)	O(n)
selection	O(n^2)	O(n^2)	O(n^2)	O(n)
quick	O(logn)	O(nlogn)	O(n^2)	O(n + log n)
merge	O(nlogn)	O(nlogn)	O(nlogn)	O(2n)
heap	O(nlogn)	O(nlogn)	O(nlogn)	O(n)

So after analyzing the performance of various algorithms we have studied so far we suggest a Master's algorithm …. implementing it by breaking the sorting into certain divisions/buckets and then analyzing each bucket and sorting it with an algorithm that works optimally for the data in the given bucket .

- This we do by first finding the range of the data set of 'n' elements if it is not specified by the user , call it R then divide the data initially into B buckets .
- We also ask the user whether he wants to optimize his data more in terms of space complexity or time .
- Now we define a constant b as the bucket size (ie b = R/B) , ie the range of each of the buckets .
- Let k = n_i/b
- say k > 1 , then it ➔ no of elements in that bucket is greater than or almost equal to the range ; which is the optimum condition for bucket sort . so we apply bucket sort for that range . note : because of the concentration of elements in this range …. We assume the numbers to be probably evenly distributed for the range .
- If k>>1 say 10 , then it is the optimum condition for counting sort .
- If not , then we check the no of elements in the bucket , n_i and use it to optimize the subsort according to best available algorithm .
- We use quicksort for n_i <5000-10000 for even though it is the fastest in terms of time complexity …. For higher amounts of data the space complexity will slow it down due to extra storage required for the recursive calls .
- Then for higher numbers if we want to optimize the time we use merge sort and if space optimization is given more preference then for 5000-10000<n_i <75000 we merge sort …. Beyond that space complexity creates issues and we use heap sort .
- Finally we sort all the individual buckets and print the final result .