Study and Comparison of Various Sorting Algorithms

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science & Engineering

Thapar University, Patiala

By:
Ramesh Chand Pandey
(80632019)

Under the supervision of:
Mrs. Shivani Goel
Lecturer, CSED

JULY 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis report entitled, **"Study and Comparison of Various Sorting Algorithms"**, submitted by me in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Mrs.Shivani Goel and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(**Ramesh Chand Pandey**)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mrs. Shivani Goel)
Lecturer
Computer Science and Engineering Department
Thapar University, Patiala

Countersigned by:

(Dr. SEEMA BAWA)
Professor & Head
Computer Science & Engineering. Department
Thapar University
Patiala

(Dr. R.K.SHARMA)
Dean(Academic Affairs)
Thapar University,
Patiala.

Acknowledgment

I express my sincere and deep gratitude to my guide Mrs.Shivani Goel, Lecturer in Computer Science & Engineering Department, for the invaluable guidance, support and encouragement. She provided me all resource and guidance throughout thesis work.

I am thankful to Dr.(Mrs.) Seema Bawa, Head of Computer Science & Engineering department Thapar University Patiala, for providing us adequate environment, facility for carrying thesis work.

I would like to thank to all staff members who were always there at the need of hour and provided with all the help and facilities, which I required for the completion of my thesis.

I would also like to express my appreciation to my co-worker and my best friends Updesh, Mohit, Arun, Raju for motivation and providing interesting work environment. It was great pleasure in working with them during this thesis work.

At last but not the least I would like to thank God and mine parents for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Ramesh Chand Pandey

(80632019)

Abstract

Sorting is an important data structure operation, which makes easy searching, arranging and locating the information. We have discussed about various sorting algorithms with their comparison to each other. We have also try to show this why we have required another sorting algorithm, every sorting algorithm have some advantage and some disadvantage. Some sorting algorithms are problem oriented and their performance and efficiency depend on the problem. We have discussed various notations for calculating the complexity of sorting algorithm. We have also discussed about fundamental sorting algorithm and advance sorting algorithm with their advantage and disadvantage. There is various applications and classification of sorting algorithms, discussed in the starting. We have compared the sorting algorithm on the basis of various important factors, like complexity, memory, method etc.

After studying various sorting algorithms; I found that there is no such sorting algorithm which works on the basis of the priority, means if we want specific data display first after that general data. So in the last we have told about problem statement their solution and implementation. We have proposed sorting algorithm, which work on the basis of priority. Which specific data we want sort first, we will assign it priority so according the priority data will be display. We have implemented our proposed algorithm in C language, and various future works related to sorting algorithms and proposed algorithm will be cover soon in near future.

Contents

List of Table

List of Figures

CHAPTER 1

Sorting Algorithms

1.1 Introduction

Every thing in this world has some advantage and disadvantage, sorting is a data structure operation, which is used for making easy searching and arranging of element or record. There are many fundamental and advance sorting algorithms. Some sorting algorithms are problem specific means they work well on some specific problem not all the problem. Sorting algorithm help searching data quickly and in this way it saves time. Arranging the record or element in some mathematical or logical order is known as sorting. Mainly sorting may be either numerical or in alphabetical. In numerical sorting we will sort numeric value either in increasing order or decreasing order and in alphabetical sort we will sort the alphabetical value e.g. Name key [1].

I grew up with the bubble sort in common; I am sure with many colleagues having learnt one sorting algorithm, there seemed little point in learning than any others. It was hardly an exciting area of study. The efficiency with which sorting is carried out will often have a significant impact on the overall efficiency of a program. Consequently there has been much research and it is interesting to see the range of alternative algorithms that have been developed.

It is not always possible to say that one algorithm is better than another, as relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted; In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order. Different algorithms will perform differently according to the data being sorted. Four common algorithms are the exchange or bubble sort, the selection sort, the insertion sort and the quick sort [2].

The selection sort is a good one to use with students. It is intuitive and very simple to program. It offers quite good performance, its particular strength being the small number

of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons.

1.2 Classification of Sorting

We can classify the sorting algorithm in different category

i) Swap –Based: Sorts begin conceptually with the entire list, and exchange particular pairs of elements (adjacent elements or elements with certain step like in Shell sort) moving toward a more sorted list.

ii) Merge –Based Swap: Creates initial "naturally" or "unnaturally" sorted sequences, and then add either one element (insertion sort) or merge two already sorted sequences.

iii) Tree-Based Sorts: For the data, at least conceptually, in a binary tree; there are two different approaches, one based on heaps, and the other based on search trees.

iv) Other Category: This is the sorts which use additional key-value information, such as radix or bucket sort [6].

We can also classify sorting algorithms by several other criteria:

i) Computational Complexity: In terms of the size of the list (n). Typically, good average number of comparisons is O(n log n) and bad is O(n^2). Asymptotic analysis does not tell about algorithms behavior on small list or worst-case behavior. Worst-case behavior is probably more important than average. For example quick sort requires $O(n^2)$ comparisons in case of already sorted arrays: a very important in practice case. Sorting algorithms which only use an abstract key comparison operation always need at least O (n log n) comparisons on average; while sort algorithms which exploit the structure of the key space cannot sort faster than O (n log k) where k is the size of the key space**.** Please note that the number of comparison is just convenient theoretical metric. In reality both moves and comparisons matter.

ii) Stability: stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then distinction does not make any sense. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records *R* and *S* with the same key and with *R* appearing before *S* in the original list, *R* will also appear before S in the sorted list.

iii) Memory Usage: One of the large classes of algorithm is in-place sorting. They are generally slower than algorithms that use additional memory; additional memory can be used for mapping of key space. Most fast stable algorithms use additional memory. With the current 4G of memory of more and virtual memory used in all major OS, the old emphasis on algorithms that does not require additional memory should be abandoned. Speedup that can be achieved by using of a small amount of additional memory is considerable and it is stupid to ignore it. Moreover you can use pointers and then additional space of the size N actually becomes size of N pointers. In real life when the records usually have size several times bigger then the size of a pointers (4 bytes in 32 bit CPUs) that makes huge difference and make those methods much more acceptable then they look form purely theoretical considerations.

iv) The difference between worst case and average behavior: Quick sort is efficient only on the average, and its worst case is n^2, while heap sort has an interesting property that the worst case, is not much different form the average case.

v) Behaviors on practically important data sets : The data set which we want to sort can come in three form completely sorted, inversely sorted and almost sorted. Some algorithms does not take any advantage of sorted order of data sequence like quick sort.

vi) Recursion: Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).

vii) Whether or not they are a comparison sort: A comparison sort examines the data only by comparing two elements with a comparison operator [7].

1.3 Method of Sorting

The various method of sorting is as follows:

i) Exchange sorts: Bubble sort, Cocktail sort, Comb Sort, Gnome sort, Quick sort .

ii) Selection sorts: Selection sort, Heap sort Smooth sort, Strand sort Insertion sort .

iii) Insertion sorts: Shell sort, Tree sort, Library sort, Patience sort.

iv) Merge sort: Merge sort.

v) Non-comparison sorts: Radix sort, Bucket sort, Counting sort, pigeonhole sort, Tally sort .

vi) Others: Topological sorting, Sorting Network [7].

1.4 Application of Sorting

Every one likes proper arrangement, if there will be proper arrangement then finding and selecting of the thing will become quite easy. There are various applications where sorting has well demand [28]:

i) Searching: Searching is most important application of sorting .The Binary search take advantage of sorting in searching the element.

ii) Closest pair: If we have many data set similar to each other then sorting is a only technique that will help you to making closest pair.

iii) Element uniqueness: By using sorting we can decide easily any delicacy is present in the data set or not.

iv) Frequency distribution: Given a set of *n* items, which element occurs the largest number of times in the set? If the items are sorted, we can sweep from left to right and count them, since all identical items will be lumped together during sorting.

v) Selection: If data will be sorted then finding the largest element or smallest element from data set will be quite easy

Chapter 2

Fundamental Sorting Algorithms

There are several popular and well-researched fundamental sorting algorithms. The some important fundamental sorting algorithms are as follow

- Bubble Sort
- Insertion Sort
- Selection sort
- Quick Sort
- Merge Sort
- Heap Sort
- Shell Sort
- Radix Sort
- Bucket Sort
- Counting Sort

2.1 Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest. 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 (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant $O(n)$ level of complexity. In general case its complexity is $O(n^2)$. While the insertion, selection and shell sorts also have $O(n^2)$ complexities, they are significantly more efficient than the bubble sort [19].

The graph clearly shows the n^2 nature of the bubble sort. A fair number of algorithm purists (which means they've probably never written software for a living) claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items.

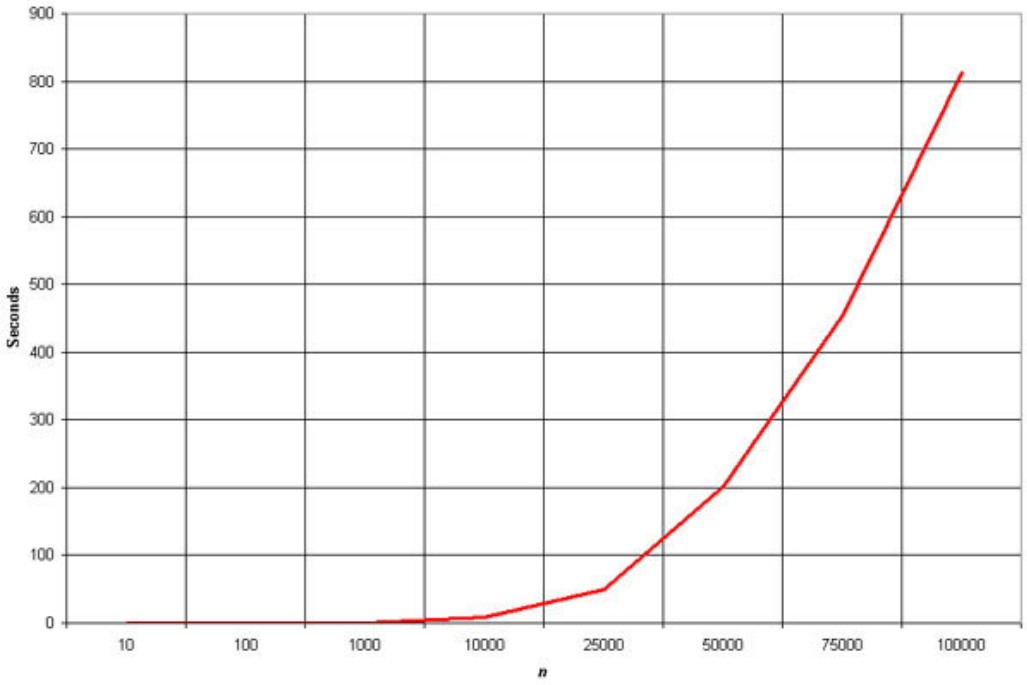

Figure 2.1: Bubble Sort Efficiency [19]

Bubble sort is an iterative algorithm that makes n - 1 passes on a list of n items. On the first pass, bubble sort starts at the head of the list and compares the first two items, if the items are out of order, they are swapped, and then the second and third items are compared and swapped if necessary. The comparison and swapping continue to the end of the list. After the first iteration, the largest item is in the n-th position.

The algorithm repeats itself, stopping one item earlier each pass. The algorithm completes when only the first two items are compared. The bubble-sorting algorithm is not code efficient. The advantage of bubble sort is its simplicity.

2.1.1Advantage: Simplicity-and-ease-of-implementation.
2.1.2 Disadvantage: Code inefficient.

2.2 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. Like the bubble sort the insertion sort has a complexity of $O(n^2)$. Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort [20].

The graph demonstrates the n^2 complexity of the insertion sort. The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertions sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Insertion sort is probably the first sorting algorithm that is taught in programming classes. It is far from efficient in terms of the number of comparisons, but is very compact in terms of code space required. As such, the insertion sort algorithm may be useful for sorting small lists.

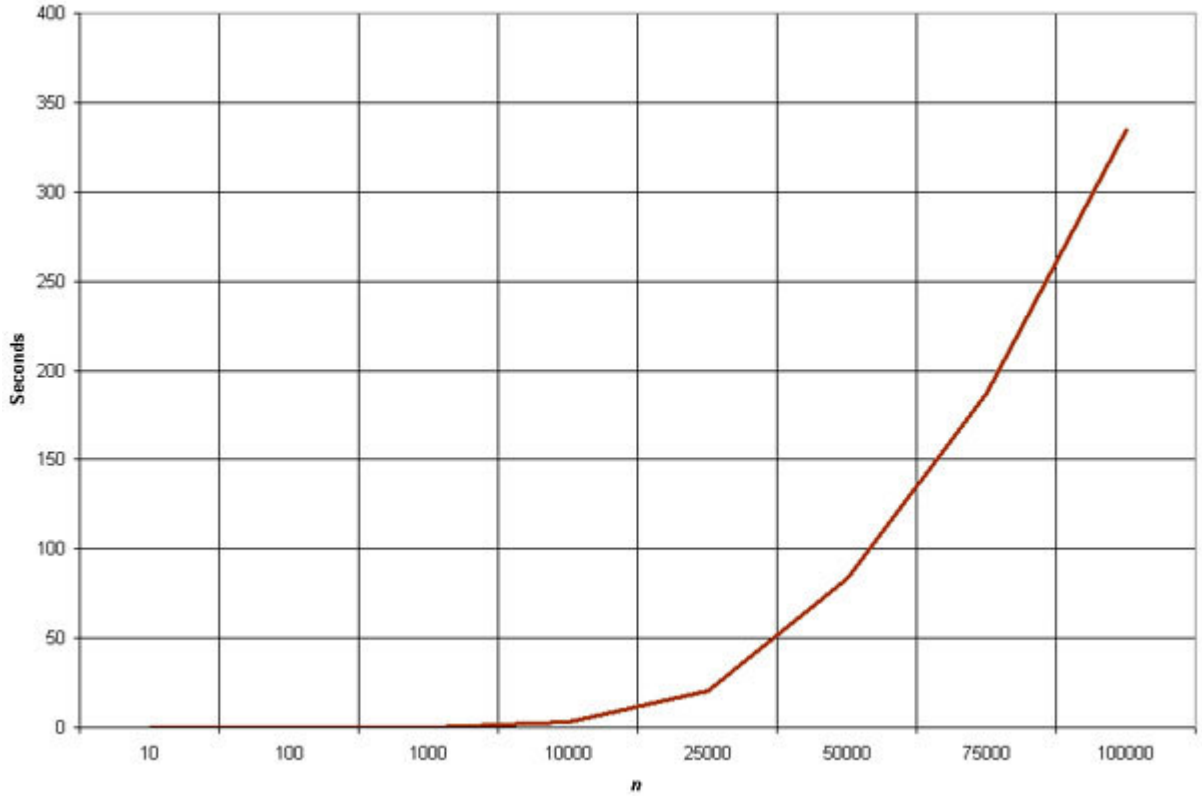

Figure 2.2: Insertion Sort Efficiency [20]

Insertion sort is an iterative algorithm that makes n - 1 passes on a list of n items. On the first pass the second item in the unsorted list is inserted in the proper position for a sorted list containing the first two items. The next pass, the third item is inserted in the proper position for a list-sorted to list containing the first three items. After the k-th iteration the first k + 1 items in the list are in sorted order. The algorithm continues until the n-th item is inserted into its position.

2.2.1Advantage: Relatively simple and easy to implement. Twice faster than bubble sort.

2.2.2 Disadvantage: Inefficient for large lists.

2.3 Selection Sort: The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$ [21].

.

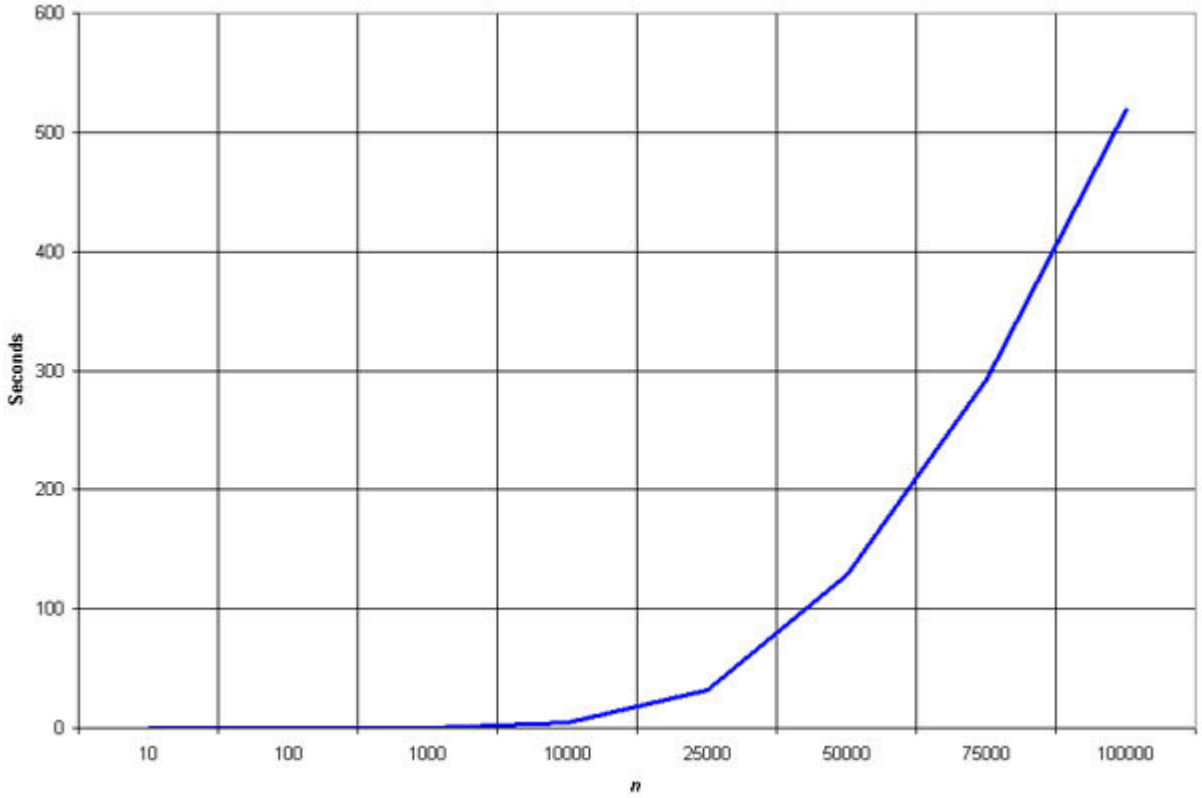

Figure 2.3: *Selection Sort Efficiency [21]*

The Selection sort is the unwanted stepchild of the n^2 sorts. It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there isn't really any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

2.3.1 Advantage: Simple and easy to implement.

2.3.2 Disadvantage: Inefficient for large lists, so similar to the more efficient insertion sort , the insertion sort should be used in its place.

2.4 Quick Sort

The Quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students)[5].

The recursive algorithm consists of four steps (which closely resemble the merge sort):

i) If there is one or less element in the array to be sorted, return immediately.

ii) Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)

iii) Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.

iv) Recursively repeats the algorithm for both halves of the original array.

The efficiency of the algorithm is mainly impacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort, $O(n^2)$, occurs when the list is sorted and the left-most element is chosen. Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of $O(n \log n)$. The quick sort is by far the fastest of the common sorting algorithms. It's possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general-case sorting there isn't anything faster.

As soon as students figure this out, their immediate impulse is to use the quick sort for everything - after all, faster is better, right? It's important to resist this urge - the quick sort isn't always the best choice. As mentioned earlier, it's massively recursive (means that for very large sorts, you can run the system out of stack space pretty easily). It's also

a complex algorithm - a little too complex to make it practical for a one-time sort of 25 items.

.

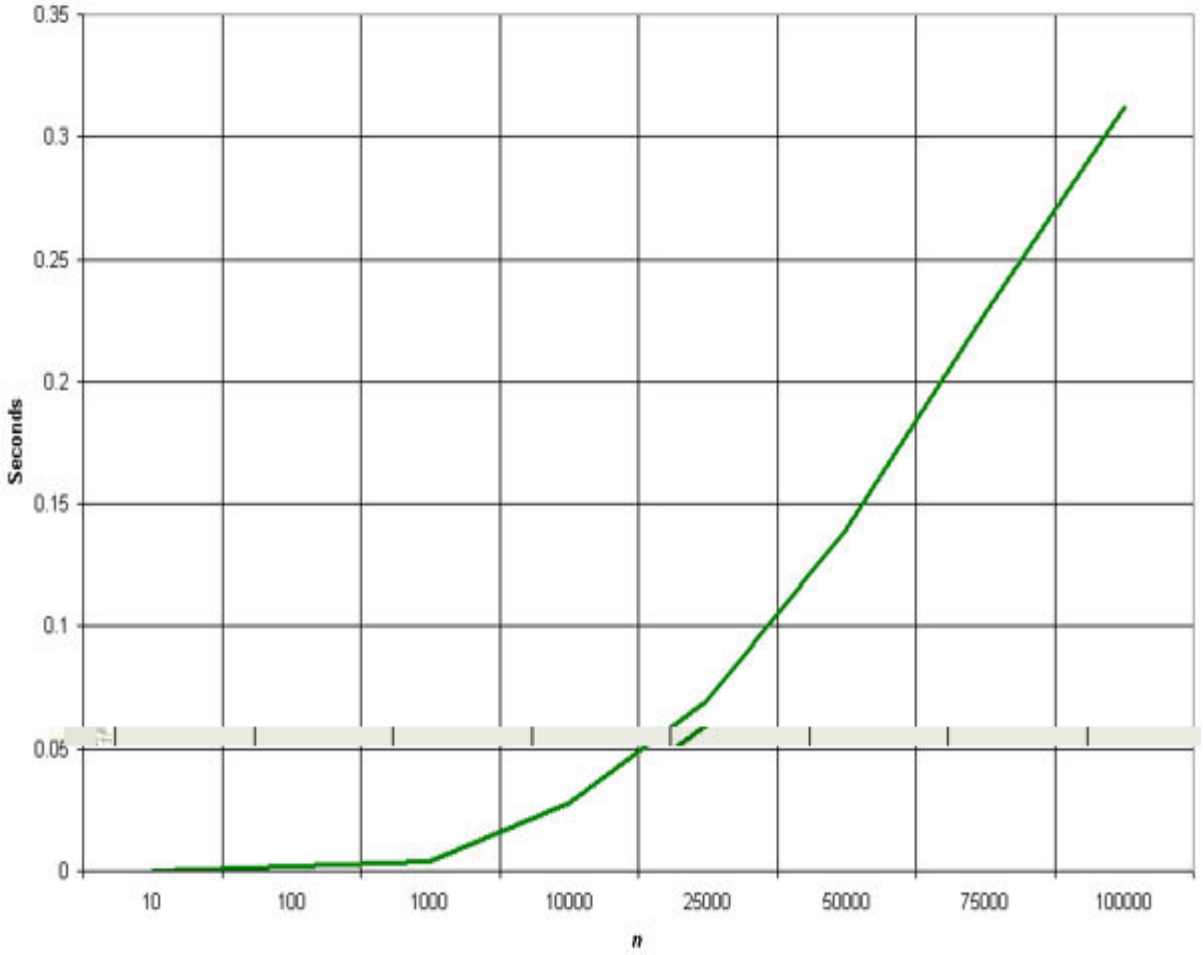

Figure 2.4: Quick Sort Efficiency [5]

In most cases the quick sort is the best choice if speed is important (and it almost always occurs). Use it for repetitive sorting, sorting of medium to large lists, and as a default choice when you're not really sure which sorting algorithm to use. Ironically, the quick sort has horrible efficiency when operating on lists that are mostly sorted in either forward or reverse order - avoid it in those situations.

Quick sort is a divide and conquer algorithm with two phases, a partition phase and a sort phase. In the partition phase all list items less than or equal to a *pivot* item are placed in one partition, while all items greater than a pivot are placed in another partition. For my implementation, I always designate the first item in the list as the pivot. To partition, We maintain a left and a right search pointer. The left pointer starts at the second item in the list and the right pointer starts at the last item in the list, then move the left pointer to the right until it finds an item greater than the pivot or points to the same item as the right pointer. If the left pointer crosses the right pointer, the list is partitioned at the crossing point. Otherwise, the right pointer is moved left until it crosses the left pointer or finds an item less than or equal to the pivot. If the right pointer crosses the left pointer, the list is partitioned at the crossing point. Otherwise the items pointed to by the two pointers are swapped, and we begin the process of moving the left and right pointers again.

The sort phase is really simple. Just call the quick sort function twice, once passing the left partition as the list to be sorted, and then passing the right partition as the list to be sorted.

Quick sort has an average order of operation of $O(n \times \log(n))$, but when the partitions are not anywhere near even, the order of operation approaches $O(n^2)$. For sorted lists and reverse sorted lists quick sort is an $O(n^2)$ algorithm.

2.3.1Advantage: Fast and efficient.

2.3.2 Disadvantage: Show horrible result if list is already sorted.

2.5 Merge Sort

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of $O(n \log n)$.Elementary implementations of the merge sort make use of three arrays - one for each half of the data set and one to store the sorted list in. There are

non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines [22].

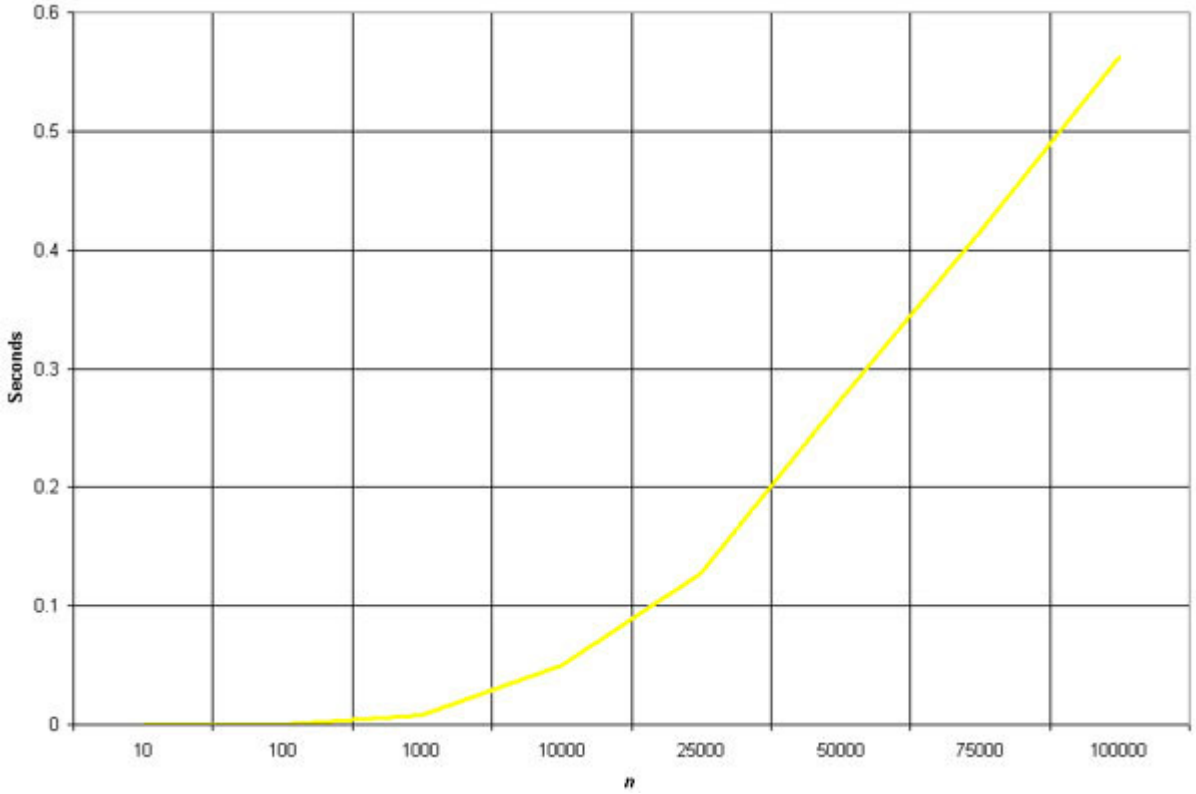

Figure 2.5: Merge Sort Efficiency [22]

The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array. This additional memory requirement makes it unattractive for most purposes- the quick sort is a better choice most of the time and the heap sort is a better choice for very large sets. Like the quick sort the merge sort is recursive which can make it a bad choice for applications that run on machines with limited memory; like quick sort, merge sort is a divide and conquer algorithm with two phases. In the case of merge sort, the phases are a partition phase, and a merge phase. In the partition phase, if the list of items to be sorted contains more than one item, the list is split in half and the merge sort algorithm is applied on each half of the list. At the end of the partition phase, the algorithm has two sorted list halves. The merge phase merges the two-sorted halves into one list. To merge two lists, compare the lowest

value in one list with the lowest value in the other list. Remove the lower of the two values from its list and place it at the end of the merged list. Repeat this process until there are no more items in either list. Unlike quick sort merge sort always partitions the list in half, so it is always an order O(n × log(n)) algorithm.

2.5.1 Advantage: Well suited for large data set.

2.5.2 Disadvantage: At least twice the memory requirements than other sorts.

2.6 Heap Sort

The heap sort is the slowest of the $O(n \log n)$ sorting algorithms, but unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. This makes it the most attractive option for *very* large data sets of millions of items. The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

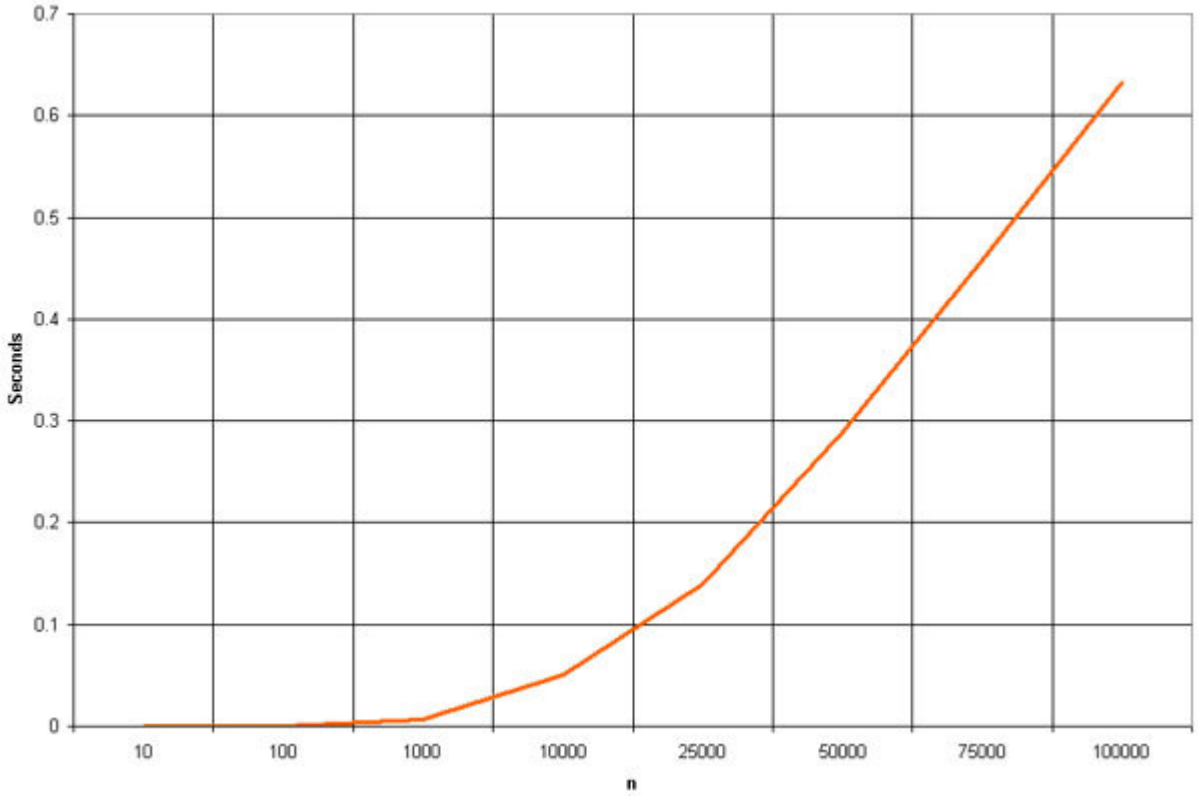

Figure 2.6: Heap Sort Efficiency [24]

As mentioned above, the heap sort is slower than the merge and quick sorts but doesn't use multiple arrays or massive recursion like they do. This makes it a good choice for really large sets, but most modern computers have enough memory and processing power to handle the faster sorts unless over a million items are being sorted. The "million item rule" is just a rule of thumb for common applications - high-end servers and workstations can probably safely handle sorting tens of millions of items with the quick or merge sorts. But if you're working on a common user-level application, there's always going to be some yahoo who tries to run it on junk machine older than the programmer who wrote it, so better safe than sorry. Heap sort a popular in place $O(n \times \log(n))$ sort algorithm. Unlike quick sort and merge sort heap sort does not require additional memory allocations to store pieces of the list being sorted. Heap sort is a two phase algorithm; it has a heap building phase and a promotion phase. A heap is a binary tree with the propriety that every node is greater than its children. For a list of 1 .. N items this means that item i is greater than its children, items (i / 2) and ((i / 2) + 1).

Once the heap is built, the largest item will be at the root of the heap. To sort simply swap the first item in the list with N th item in the list. Next rebuild the heap with items 1.. (N - 1). Repeat the process of swapping and rebuilding the heap with one less item, until only two items remain.

2.6.1 Advantage: Well suited for large data set, do not use massive recursion like merge and quick sort.

2.6.2 Disadvantage: Less efficient than quick sort and merge sort.

2.7 Shell Sort

Invented by Donald Shell in 1959, the shell sort is the most efficient of the $O(n^2)$ class of sorting algorithms. Of course, the shell sort is also the most complex of the $O(n^2)$ algorithms. The shell sort is a "diminishing increment sort ", better known as a "comb sort" to the unwashed programming masses. The algorithm makes multiple passes through the list, and each time sorts a number of equally sized sets using the insertion sort. The size of the set to be sorted gets larger with each pass through the list, until the set consists of the entire list. (Note that as the size of the set increases, the number of sets to be sorted decreases.) This sets the insertion sort up for an almost-best case run each iteration with a complexity that approaches $O(n)$.The items contained in each set are not contiguous - rather, if there are i sets then a set is composed of every ith element.

For example, if there are 3 sets then the first set would contain the elements located at positions 1, 4, 7 and so on. The second set would contain the elements located at positions 2, 5, 8, and so on; while the third set would contain the items located at positions 3, 6, 9, and so on.

The size of the sets used for each of the iteration has a major impact on the efficiency of the sort. The shell sort is by far the fastest of the N^2 class of sorting algorithms. It's more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort its closest competitor.

The shell sort is still significantly slower than the merge, heap and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed is hypercritical. It's also an excellent choice for repetitive sorting of smaller lists. Shell sort is an optimization of the insertion sort proposed by D. L. Shell.

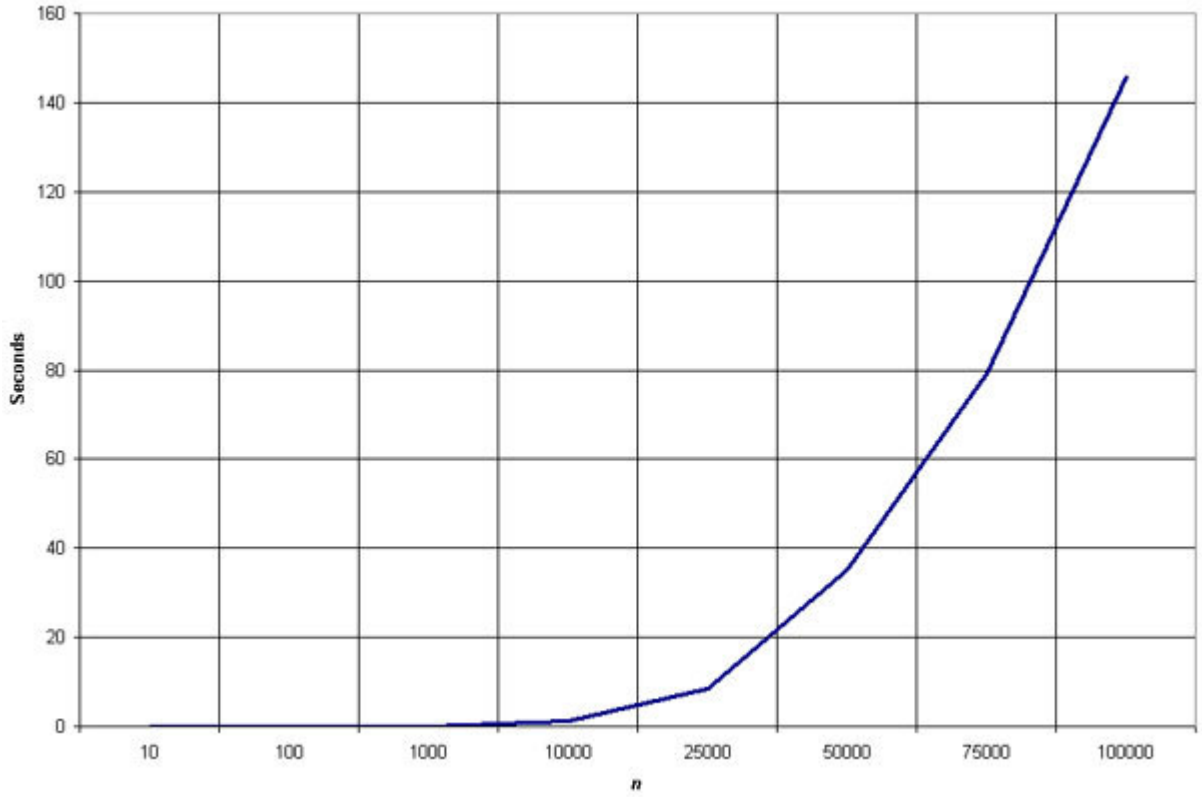

Figure 2.7: Shell Sort Efficiency [23]

2.7.1 Advantage: Efficient for medium-size lists.

2.7.2 Disadvantage: Complex algorithm, not nearly as efficient as the merge, heap and quick sorts.

2.8 Radix Sort

Radix sort is an unusual sort algorithm. All the sort algorithms discussed so far compare the elements being sorted to each other. The radix sort doesn't. A radix sort uses one or more keys to sort values. The key of a value indicates the sort grouping for a value on a particular pass. Successive passes with different keys may be used to sort a list of elements.

On a radix sort pass, a list of items to be sorted is processed from beginning to end. One at a time, the items in the list are placed at the end of a list of items with the same key value. After all items have been processed, the lists of key values are reassembled smallest key to largest key [26].

A somewhat intuitive example of how to use a radix sort might be alphabetizing strings of length N. Each pass of the radix sort should use a character in the string as its key. The less intuitive part about alphabetizing is that radix sort passes should be from performed from the last character to the first character. The first pass will use the N^{th} character, the second pass will use the $(N - 1)^{th}$ character, ..., the N^{th} pass will use the first character.

Example: Sorting 3 Characters (start with third character)

Unsorted	vbn	asd	qwe	vgy	qaz
First Pass	asd	qwe	vbn	vgy	qaz
Second Pass	qaz	vbn	vgy	asd	qwe
Third Pass	asd	qaz	qwe	vbn	vgy

Table 2.1: Radix sort Example [26]

Sorting unsigned integers is similar to sorting strings. If we use a radix of 10, we can sort integer's one digit at a time. Like strings, integers should be sorted least significant digit to most significant digit.

Example: Sorting Integers (start with least significant digit)

Unsorted	1356	2453	7579	5624	1234
First Pass	2453	5624	1234	1356	7579
Second Pass	5624	1234	2453	1356	7579
Third Pass	1234	1356	2453	7579	5624
Fourth Pass	1234	1356	2453	5624	7579

Table: 2.2 Radix sort Example [26]

Each time `Radix Sort` is called, it makes two passes on the data being sorted. The first pass counts the number of values with each possible key; the second pass places the data in its sorted order. Counting the number of values for each key allows the data to be sorted into a fixed size array instead of a linked list or similar data structure. Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the one's digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Table: 2.3 *Radix sort Example [26]*

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sub lists above. Now, we gather the sub lists (in order from the 0 sub list to the 9 sub list) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The order in which we divide and reassemble the list is extremely important, as this is one of the foundations of this algorithm.

Now, the sub lists are created again, this time based on the ten's digit:

Digits	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Table: 2.4 Radix sort Example[26]

Now the sub lists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sub lists are created according to the hundred's digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

Table: 2.5 *Radix sort Example [26]*

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array. Radix sort is very simple, and a computer can do it fast. When it is programmed properly, radix sort is in fact one of the fastest sorting algorithms for numbers or strings of letters.

2.8.1 Advantage: Fast and complexity does not depend on the number of data.

2.8.2 Disadvantages: Less flexible, take more memory, very difficult to write radix sort that can sort all data type, radix sort take more time to write the data. Radix sort depend on digit or letter. The speed of Radix sort largely depends on the inner basic operations, and if the operations are not efficient enough, Radix sort can be slower than some other algorithms such as Quick sort and Merge sort. These operations include the insert and delete functions of the sub lists and the process of isolating the digit you want.

In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for

additional digits that need sorting. This can be one of the slowest parts of Radix sort and it is one of the hardest to make efficient.

2.9 Bucket Sort

Bucket sort runs in linear time on the average. It assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1).The idea of Bucket sort is to divide the interval [0, 1) into n equal-sized subintervals, or buckets, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over (0, 1), we don't expect many numbers to fall into each bucket. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each iteration [25].

The code assumes that input is in n-element array A and each element in A satisfies $0 \leq A[i] \leq 1$. We also need an auxiliary array B[0 . . n -1] for linked-lists (buckets).

The bucket sort work as follow:

Bucket Sort (A)

1. $n \leftarrow$ length [A]
2. For i = 1 to n do
3. Insert A [i] into list B[nA[i]]
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.

Example

Given input array A[1..10]. The array B[0..9] of sorted lists or buckets after line 5. Bucket i holds values in the interval [i/10, (i +1)/10]. The sorted output consists of a concatenation in order of the lists first B[0] then B[1] then B[2] ... and the last one is B[9].

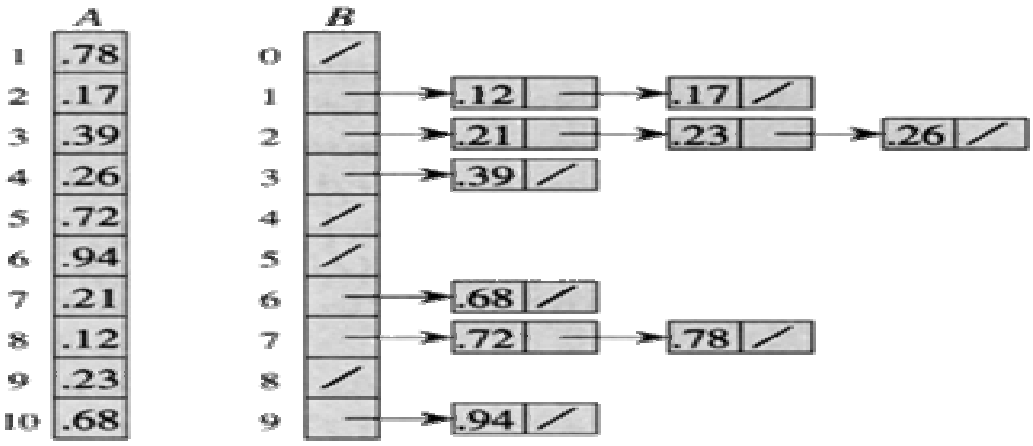

Figure 2.8: Bucket sort Example

2.9.1 Advantage: Bucket sort runs in linear time in the average case.

2.9.2 Disadvantage: Its complexity depends on another sorting algorithm**.**

2.10 Counting Sort

Counting sort is a sorting algorithm which (like bucket sort) takes advantage of knowing the range of the numbers in the array A to be sorted. It uses this range to create an array C of this length. Each index i in array is then used to count how many elements in *A* have the value *i*. The counts stored in *C* can then be used to put the elements in *A* into their right position in the resulting sorted array [27].

Counting sort is a stable sort and has a running time of $\Theta(n+k)$, where n and k are the lengths of the arrays *A* (the input array) and *C* (the counting array), respectively. In order for this algorithm to be efficient, k must not be much larger than n. The indices of *C* must run from the minimum to the maximum value in *A* to be able to index *C* directly with the values of *A*. Otherwise, the values of *A* will need to be translated (shifted), so that the minimum value of *A* matches the smallest index of *C*.

2.10.1 Advantage: Counting sort may be the best algorithm for sorting numbers whose range is between 0 and 100. However counting sort can be used in radix sort to sort a list

of numbers whose range is too large for counting sort to be suitable alone. Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the $\Omega(n \log n)$ lower-bound for sorting is inapplicable.

2.10.2 Disadvantage: The length of the counting array *C* must be at least equal to the range of the numbers to be sorted (that is, the maximum value minus the minimum value plus 1). This makes counting sort impractical for large ranges in terms of time and memory needed .It is probably unsuitable for sorting a list of names alphabetically.

Chapter 3

Advance Sorting Algorithm

Advance sorting algorithm is some modification of fundamental sorting algorithm. Generally advance-sorting algorithm is problem specific and shows their best behavior for appropriate problem.

The list of various advance sorting is as follow-

- Library Sort
- Cocktail Sort
- Comb Sort
- Gnome Sort
- Patience Sort
- Pigeonhole Sort
- Smooth Sort
- Strand Sort
- Topological Sort
- Tally Sort
- Postman Sort

3.1 Library Sort

Library sort, or gapped insertion sort is a sorting algorithm that uses an insertion sort, but with gaps in the array to accelerate subsequent insertions. The name comes from an analogy: Suppose a librarian were to store his books alphabetically on a long shelf, starting with the as at the left end, and continuing to the right along the shelf with no spaces between the books until the end of the Zs. If the librarian acquired a new book that belongs to the B section, once he finds the correct space in the B section, he will have to move every book over, from the middle of the Bs all the way down to the Zs in order to make room for the new book. This is an insertion sort. However, if he were to leave a space after every letter, as long as there was still space after B, he would only have to

move a few books to make room for the new one. This is the basic principle of the Library sort [14].

Library sort is a stable comparison sort and can be run as an online algorithm; however, it was shown to have a high probability of running in O(nlog(n)) time (comparable to quick sort), rather than an insertion sort's $O(n^2)$. Its implementation is very similar to a skip list. The drawback to using the library sort is that it requires extra space for its gaps (the extra space is traded off against speed).

3.2 Cocktail Sort

Also known as bidirectional bubble sort, cocktail shaker sort, shaker sort (which can also refer to a variant of selection sort), ripple sort, shuttle sort or happy hour sort, is a variation of bubble sort that is both a stable sorting algorithm and a comparison sort. The algorithm differs from bubble sorting that sorts in both directions each pass through the list. This sorting algorithm is only marginally more difficult than bubble sort to implement, and solves the problem in bubble sort [8].

3.2.1 Differences from Bubble Sort

Cocktail sort is a slight variation of bubble sort. It differs in that instead of repeatedly passing through the list from bottom to top, it passes alternately from bottom to top and then from top to bottom. It can achieve slightly better performance than a standard bubble sort. The reason for this is that bubble sort only passes through the list in one direction and therefore can only move items backward one step in each of the iteration.

An example of a list that proves this point is the list (2,3,4,5,1), which would only need to go through one pass of cocktail sort to become sorted, but if using an ascending bubble sort would take four passes. However one cocktail sort pass should be counted as two-bubble sort passes. Typically cocktail sort is less than two times faster than bubble sort.

3.2.2Complexity

The complexity of cocktail sort in big O notation is $O(n^2)$ for both the worst case and the average case, but it becomes closer to $O(n)$ if the list is mostly ordered before applying the sorting algorithm, for example, if every element is at a position that differs at most k ($k \geq 1$) from the position it is going to end up in, the complexity of cocktail sort becomes $O(k * n)$.

3.3 Comb Sort

In computer science, comb sort is a relatively simplistic sorting algorithm originally designed by Wlodek Dobosiewicz in 1980 and later rediscovered and popularized by Stephen Laceyand Richard Box, who described it in Byte Magazine in April 1991. Comb sort improves on bubble sort and rivals in speed more complex algorithms like Quick sort. The basic idea is to eliminate turtle*s*, or small values near the end of the list. In a bubble sort these slow the sorting tremendously. (In bubble sort, when any two elements are compared, they always have a gap (distance from each other) of 1. The basic idea of comb sort is that the gap can be much more than one. (Shell sort is also based on this idea, but it is a modification of insertion sort rather than bubble sort.)[9].

3.4 Gnome Sort

Gnome sort is a sorting algorithm, which is similar to insertion sort, except that moving an element to its proper place is accomplished by a series of swaps, as in bubble sort. Gnome Sort is based on the technique used by the standard Dutch Garden Gnome. Here is how a garden gnome sorts a line of flower pots. Basically, he looks at the flower pot next to him and the previous one; if they are in the right order he steps one pot forward, otherwise he swaps them and steps one pot backwards. Boundary conditions: if there is no previous pot, he steps forwards; if there is no pot next to him, he is done. It is conceptually simple, requiring no nested loops. The running time is $O(n^2)$, but in practice the algorithm can run as fast as Insertion sort. The algorithm always finds the first place

where two adjacent elements are in the wrong order, and swaps them. It takes advantage of the fact that performing a swap can introduce a new out-of-order adjacent pair only right before or after the two swapped elements. It does not assume that elements forward of the current position are sorted, so it only needs to check the position directly before the swapped elements [10].

Example:

If we wanted to sort an array with elements [4] [2] [7] [3], here is what would happen with each of the iteration of the while loop:

[4] [2] [7] [3] (initial state. i is 1 and j is 2.)
[4] [2] [7] [3] (did nothing, but now i is 2 and j is 3.)
[4] [7] [2] [3] (swapped a[2] and a[1]. now i is 1 and j is still 3.)
[7] [4] [2] [3] (swapped a[1] and a[0]. now i is 1 and j is still 3.)
[7] [4] [2] [3] (did nothing, but now i is 3 and j is 4.)
[7] [4] [3] [2] (swapped a[3] and a[2]. now i is 2 and j is 4.)
[7] [4] [3] [2] (did nothing, but now i is 4 and j is 5.)
at this point the loop ends because i isn't equal to 4.

3.5 Patience Sort

Patience sorting is a sorting algorithm, based on a solitaire card game that has the property of being able to efficiently compute the length of the longest increasing subsequence in a given array [15].

3.5.1 The card game

The game begins with a shuffled deck of cards, labeled 1,2,…,n. The cards are dealt one by one into a sequence of piles on the table, according to the following rules.

i) Initially, there are no piles. The first card dealt forms a new pile consisting of the single card.

ii) Each new card may be placed either on an existing pile that has a higher value card on the top (if any such pile exists), thus increasing the number of cards in that pile, or to the right of all of the existing piles, thus forming a new pile.

iii) When there are no more cards remaining to deal, the game ends.

The object of the game is to finish with as few piles as possible.

3.5.2 Algorithm for sorting

Given an n-element array with an ordering relation as an input for the sorting, consider it as a collection of cards, with the (unknown in the beginning) statistical ordering of each element serving as its index. Note that the game never uses the actual value of the card, except for comparison between two cards, and the relative ordering of any two array elements is known. Now simulate the patience sorting game, played with the greedy strategy, i.e., placing each new card on the leftmost pile that is legally possible to use. At each stage of the game, under this strategy, the labels on the top cards of the piles are increasing from left to right. To recover the sorted sequence, collect the minimum card from the top row each time.

3.5.3 Complexity

When no assumption is made about values, the greedy strategy can be implemented in $O(n \log(n))$ comparisons in worst case. In fact, one can implement it with an array of stacks ordered by values of top cards and, for inserting a new card, use a binary search, which is $O(\log p)$ comparisons in worst case, where p is the number of piles. But there is no known efficient way ($O(n \log(n))$ worst case) of using the structure obtained by the greedy strategy to complete the sorting. The card with the least value is at top of leftmost pile but when this card is removed this is no longer the case in the new configuration, and some work has to be done. Finding the next card by searching it among all tops of piles, as in the implementation, gives a $O(n^2)$ worst case. Inserting the first pile in the remaining list of piles, with respect to the top cards ordering, can cost $O(n)$ each time and so gives also a $O(n^2)$ worst case (take k+1,…,2k,1,…,k as starting list of values).

3.6 Pigeonhole Sort

Pigeonhole sorting, also known as count sort, is a sorting algorithm that is suitable for sorting lists of elements where the number of elements (n) and the number of possible key values (N) are approximately the same. It requires $\Theta(n + N)$ time[16].

The pigeonhole algorithm works as follows:

i) Set up an array of initially empty "pigeonholes", one pigeonhole for each value in the range of keys. Each pigeonhole will contain a list of values having that key.

ii) Go over the original array, adding each object to the list in its pigeonhole.

iii) Iterate over the pigeonhole array in order, and put elements from non-empty holes back into the original array.

For example, suppose we were sorting these value pairs by their first element:

- (5, "hello")
- (3, "pie")
- (8, "apple")
- (5, "king")

For each value between 3 and 8 we set up a pigeonhole, and then move each element to its pigeonhole:

- 3: (3, "pie")
- 4:
- 5: (5, "hello"), (5, "king")
- 6:
- 7:
- 8: (8, "apple")

We then iterate over the pigeonhole array in order and move them back to the original list.

The difference between pigeonhole sort and counting sort is that in counting sort, the auxiliary array does not contain lists of input elements, only counts:

- 3: 1
- 4: 0
- 5: 2
- 6: 0
- 7: 0
- 8: 1

Using this information we can perform a series of exchanges on the input array that puts it in order; but the elements never leave the input array.

For arrays where N is much larger than n, bucket sort is a generalization of this sort, which is more efficient in space and time.

3.7 Smooth Sort

Smooth sort is variation of the heap sort. Like heap sort, smooth sort's upper bound is $O(n \log n)$. The advantage of smooth sort is that it comes closer to $O(n)$ time if the input is already sorted to some degree, whereas heap sort averages $O(n \log n)$ regardless of the initial sorted state. Due to its complexity, smooth sort is rarely used [11].

3.8 Strand Sort

It works by repeatedly pulling sorted sub lists out of the list to be sorted and merging them with a result array. Each of the iteration through the unsorted list pulls out a series of elements, which were already sorted, and merges those series together.

The name of the algorithm comes from the "strands" of sorted data within the unsorted list, which are removed one at a time. It is a comparison sort due to its use of comparisons when removing strands and when merging them into the sorted array [12].

The strand sort algorithm is O (n log n) in the average case. In the best case (a list which is already sorted) the algorithm is linear or O (n). In the worst case (a list which is sorted in reverse order) the algorithm is O (n^2).

Strand sort is most useful for data, which is stored in a linked list, due to the frequent insertions and removals of data. Using another data structure, such as an array, would greatly increase the running time and complexity of the algorithm due to lengthy insertions and deletions. Strand sort is also useful for data, which already has large amounts of sorted data, because such data can be removed in a single strand.

Example

	Sub list	Sorted List
3 1 5 4 2		
	3 5	
1 4 2		3 5
2	1 4	3 5
2		1 3 4 5
	2	1 3 4 5
		1 2 3 4 5

Table 3.1: Strand sort [12]

i) Parse the Unsorted List once, taking out any ascending (sorted) numbers.

ii) The (sorted) Sub list is, for the first iteration, pushed onto the empty sorted list.

iii) Parse the Unsorted List again, again taking out relatively sorted numbers.

iv) Since the Sorted List is now populated, merge the Sub list into the Sorted List.

v) Repeat steps 3-4 until both the Unsorted List and Sub list are empty.

3.9 Topological Sort

In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG), is a linear ordering of its nodes in which each node comes before all nodes to

which it has outbound edges. Every DAG has one or more topological sorts. Topological sorting is sometimes also referred to as ancestral ordering [15].

More formally, define the partial order relation *R* over the nodes of the DAG such that *x R y* if and only if there is a directed path from *x* to *y*. Then, a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

Examples

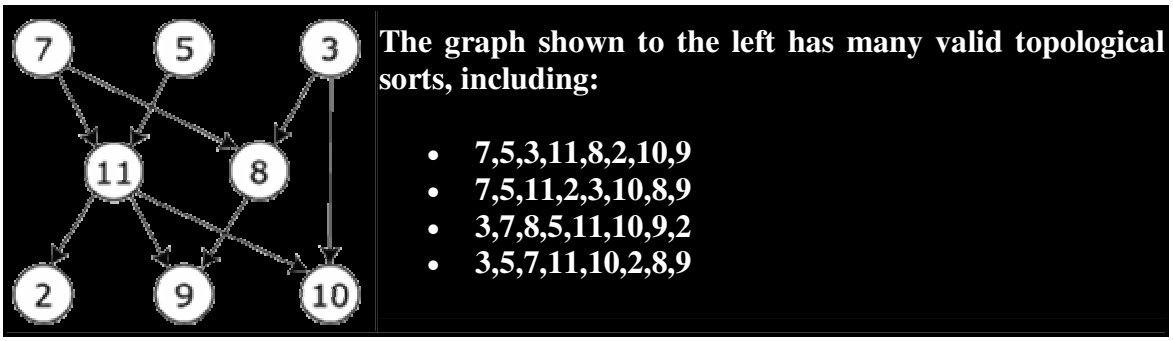

Figure 3.1: Topological sort [15]

If the graph was a DAG, a solution is contained in the list L (the solution is not unique). Otherwise, the graph has at least one cycle and therefore a topological sorting is impossible.

3.9.1 Uniqueness

If a topological sort has the property that all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed Hamiltonian path in the DAG. If a Hamiltonian path exists, the topological sort order is unique; no other order respects the edges of the path. Conversely, if a topological sort does not form a Hamiltonian path, the DAG will have two or more valid topological orderings.

3.10 Tally Sort

A well-known variant of counting sort is tally sort, where the input is known to contain no duplicate elements, or where we wish to eliminate duplicates during sorting. In this case the count array can be represented as a bit array; a bit is set if that key value was

observed in the input array. Tally sort is widely familiar because of its use in the book Programming Pearls as an example of an unconventional solution to a particular set of limitations [17]. The tally sort work as follow

- Find the highest and lowest elements of the set
- Count the different elements in the array. (E.g. Set [4,4,4,1,1] would give three 4's and two 1's)
- Accumulate the counts. (E.g. starting from the first element in the new set of counts, add the current element to the previous.)
- Fill the destination array from backwards: put each element to its count position.

3.11 Postman Sort

This is a sorting that sorts an arbitrarily large number of records in less time than any other algorithm based on comparison sorting can do. For many commonly encountered files, time will be strictly proportional to the number of records. It can sort an arbitrary group of fields with arbitrary collating sequence on each field faster than any other program available [18].

When a postal clerk receives a huge bag of letters he distributes them into other bags by state. Each bag gets sent to the indicated state. Upon arrival, another clerk distributes the letters in his bag into other bags by city. So the process continues until the bags are the size one man can carry and deliver. This is the basis for sorting method which I call the postman's sort. Suppose we are given a large list of records to be ordered alphabetically on a particular field. Make one pass through the file. Each record read is added to one of 26 lists depending on the first letter in the field. The first list contains all the records with fields starting with the letter "A" while the last contains all the records with fields starting with the letter "Z". Now we have divided the problem down to 26 smaller sub problems. Now we address sub problem of sorting all the records in the sub list corresponding to key fields starting with the letter "A". If there are no records in the "A" sub list we can proceed to deal with the "B" sub list. If the "A" sub list contains only one record it can be

written to output and we are done with that sub list. If the "A" sub list contains more that one record, it must be sorted then output. Only when the "A" list has been disposed of we can move on to each of the other sub lists in sequence. The records will be written to the output in alphabetical order. When the "A" list contains more than one record it has to be sorted before it is output. What sorting algorithm should be used? Just like a real postman, we use the postman sort . Of course we just apply the method to the second letter of the field. This is done to greater and greater depths until eventually all the words starting with "A" are written to the output. We can then proceed to deal with sub lists "B" through "Z" in the same manner.

Chapter 4

Complexity & Comparison of Various Algorithms

This is very important to know about what is complexity of my sorting algorithm in term of time and space. If we blindly use sorting without considering complexity of sorting algorithm then it will be very harmful. Now important question is: How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- Memory usage
- Disk usage
- Network usage

Be careful to differentiate between, Performances: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code. Complexity, how do the resource requirements of a program or algorithm scale, i.e., what happens, as the size of the problem being solved gets larger? Complexity affects performance but not the other way around. The time required by a method is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- One arithmetic operation (e.g., +, *)
- One assignment
- One test (e.g., x == 0)
- One read
- One write (of a primitive type)

Some methods perform the same number of operations every time they are called. For example, consider the size method, of the Sequence class always performs just one operation: `return numItems`; the number of operations is independent of the size of the sequence. We say that methods like this (that always perform a fixed number of basic operations) require constant time. Other methods may perform different numbers of

operations, depending on the value of a parameter or a field. For example, for many of the Sequence methods, the number of operations depends on the size of the sequence. We call the important factor (the parameters and/or fields whose values affect the number of operations performed) the problem size or the input size.

When we consider the complexity of a method, we don't really care about the exact number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? Double? Increase in some other way? For constant-time methods like the *size* method, doubling the problem size does not affect the number of operations (which stays the same).

4.1 Different Notation for Calculating Complexity

To show the complexity of the sorting algorithm in time and space, we use some asymptotic notation. These notations help us predict the best, average and poor behavior of the sorting algorithm. The various notations are as follow:

- *O (x) = Worst Case Running Time*
- *Ω (x) = Best Case Running Time*
- *Θ (x) = Best and Worst case are the same*

4.1.1 Big-O Notation

- **Definition:** A theoretical measure of the execution of an algorithm usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f (n) = O(g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".
- **Formal Definition:** f (n) = O (g (n)) means there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$. The values of c and k must be fixed for the function-

f-and-must-not-depend-on-n.

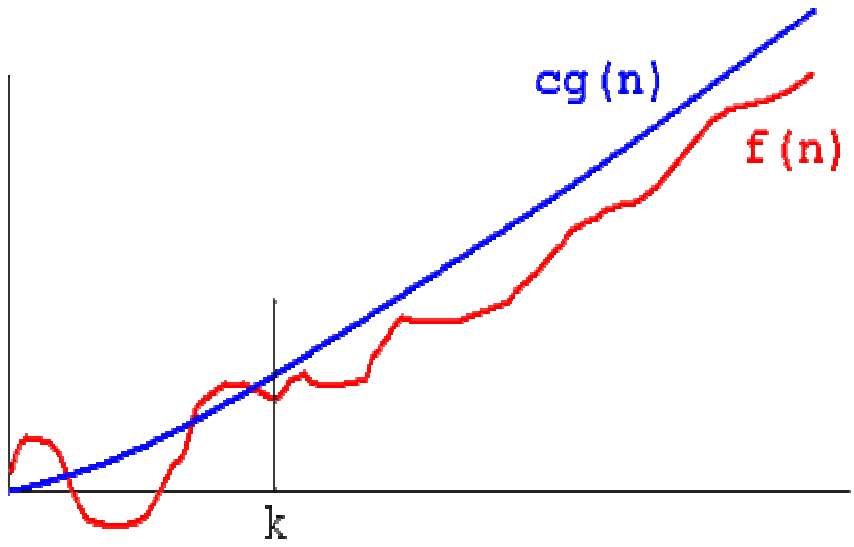

Figure 4.1: Big O Notation Graph

This notation is upper bound means indicate what worse can happen [30].

4.1.2 Theta Notation (Θ)

- **Definition:** A theoretical measure of the execution of an algorithm usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = Θ (g(n)) means it is within a constant multiple of g(n). The equation is read, "f of n is theta g of n".
- **Formal Definition:** f(n) = Θ (g(n)) means there are positive constants c_1, c_2, and k, such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq k$. The values of c_1, c_2, and k must be fixed for the function f and must not depend on n.

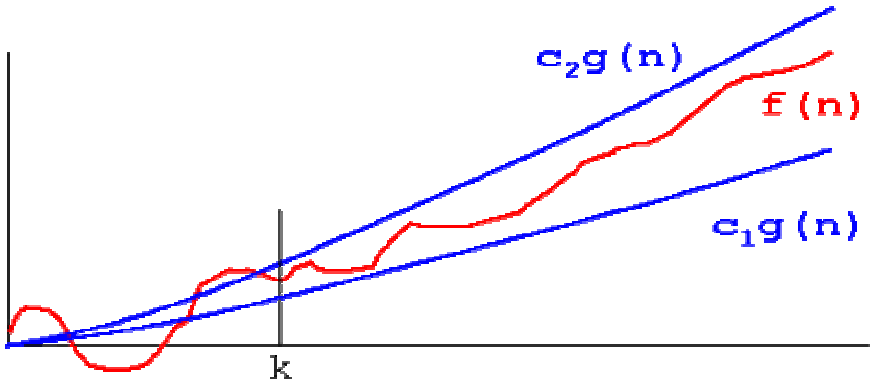

Figure 4.2: Theta Notation Graph

This notation is medium bound indicate what average can happen [31].

4.1.3 Omega Notation (ω)

- **Definition:** A theoretical measure of the execution of algorithms usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = ω (g(n)) means g(n) becomes insignificant relative to f(n) as n goes to infinity.
- **Formal Definition:** f(n) = ω (g(n)) means that for any positive constant c, there exists a constant k, such that $0 \leq cg(n) < f(n)$ for all $n \geq k$. The value of k must not depend on n, but may depend on c.

This notation is lower bound indicate what best can happen [32].

4.2 How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used [29].

1. Sequence of statements
2. statement 1;
3. statement 2;

```
4.          ...
5.      statement  k;
```

Total time = time (statement 1) + time (statement 2) + ... + time (statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

```
6.  if-then-else statements
7.      if (cond) {
8.          sequence of statements 1
9.      }
10.     else {
11.         sequence of statements 2
12.     }
```

Worst-case time is the slowest of the two possibilities: max (time (sequence 1) time (sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

```
13. for loops
14.     for (i = 0; i < N; i++) {
15.         sequence of statements
16.     }
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

```
17. Nested loops
18.     for (i = 0; i < N; i++) {
```

```
19.       for (j = 0; j < M; j++) {
20.           sequence of statements
21.       }
22.   }
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is `j < N` instead of `j < M` (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

When a loop is involved, the same rule applies. For example:

```
for (j = 0; j < N; j++) g(N);
```

has complexity (N^2). The loop executes N times and each time call `g(N)`.

4.3 Best-case and Average-case Complexity

Some methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, we know that if add before is called with a sequence of length N, it may require time proportional to N (to move all of the items and/or to expand the array). This is what happens in the worst case. However, when the current item is the last item in the sequence, and the array is not full, add Before will only have to move one item, so in that case its time is independent of the length of the sequence; i.e., constant time.

In general, we may want to consider the best and average time requirements of a method as well as its worst-case time requirements. Which is considered the most important will

depend on several factors. For example, if a method is part of a time-critical system like one that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average-case times for that computation are not relevant - the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases [29].

For add Before, for a sequence of length N, the worst-case time is O(N), the best-case time is O(1), and the average-case time (assuming that each item is equally likely to be the current item) is O(N), because on average, N/2 items will need to be moved.

Note that calculating the average-case time for a method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly.

4.4 When does Constants Matter

Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the same big-O time complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires $10 * N^2 + N$ time. For both algorithms, the time is $O(N^2)$, but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms do matter in terms of which algorithm is actually faster [29].

However, it is important to note that constants do not matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an

algorithm that requires $10*N^2$ time, for both algorithms, if the problem size doubles, the actual time will quadruple.

When two algorithms have different big-O time complexity, the constants and low-order terms only matter when the problem size is small.

4.5 Comparison of Various Sorting

Here we are going to compare among various sorting algorithms on the basis of various important factor. From the given comparison we can easily find out which algorithm is best according to my problem [7]. The various factor of the comparison are as follow -

- Complexity
- Memory
- Stability
- Method

The table given below tell all the specific feature of all sorting which we have studied. By seeing the table in one sight we can know the various important factor of the sorting like worst complexity, average complexity, stability, method, and extra memory required etc.

Name	**Average**	**Worst**	**Memory**	**Stable**	**Method**	**Other notes**
Bubble sort	—	O(n^2)	O(1)	Yes	Exchanging	Oldest sort
Cocktail	—	O(n^2)	O(1)	Yes	Exchanging	Variation of bubble sort
Comb sort	—	—	O(1)	No	Exchanging	Small code size
Gnome sort	—	O(n^2)	O(1)	Yes	Exchanging	Tiny code size

Selection sort	$O(n^2)$	$O(n^2)$	O(1)	No	Selection	Can be implemented as a stable sort
Insertion sort	$O(n^2)$	$O(n^2)$	O(1)	Yes	Insertion	Average case is also $O(n + d)$, where d is the number of inversion
Shell sort	—	$O(n \log^2 n)$	O(1)	No	Insertion	No extra memory required
Binary Tree sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Insertion	When using a self-balancing binary search tree
Library sort	$O(n \log n)$	$O(n^2)$	$O(n)$	Yes	Insertion	Also know as gap insertion sort
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging	Recursive nature, extra memory required
Heap sort	$O(n \log n)$	$O(n \log n)$	O(1)	No	Selection	Recursive , extra memory required
Smooth sort	—	$O(n \log n)$	O(1)	No	Selection	
Quick sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Partitioning	Recursive, based on divide conquer technique
Patience sorting	—	$O(n^2)$	$O(n)$	No	Insertion	

Strand sort	O(n log n)	O(n^2)	O(n)	Yes	Selection	

Table 4.1: Comparison of Various Sortings [7]

The following table describes sorting algorithms that are not comparison sorts. As such, they are not limited by a O(n log n) lower bound. Complexities below are in terms of n, the number of items to be sorted, k, the size of each key, and s, the chunk size used by the implementation. Many of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that $n << 2^k$, where << means "much less than."

Name	**Average**	**Worst**	**Memory**	**Stable**	***n***	**Notes**
Pigeonhole sort	$O(n+2^k)$	$O(n+2^k)$	$O(2^k)$	Yes	Yes	
Bucket sort	$O(n \cdot k)$	$O(n^2 \cdot k)$	$O(n \cdot k)$	Yes	No	Assumes uniform distribution of elements from the domain in the array.
Counting sort	$O(n+2^k)$	$O(n+2^k)$	$O(n+2^k)$	Yes	Yes	
Radix sort	$O(n \cdot k/s)$	$O(n \cdot (k/s) \cdot 2^s)$	$O((k/s) \cdot 2^s)$	No	No	

Table 4.2: Comparison of Various Sortings [7]

Chapter 5

Problem Statement

Sorting is important data structure operation. Every sorting algorithm have some advantage and disadvantage .How many algorithms we have read until fall generally in two category. First one is numerical sort and another one is alphabetical or lexicographical sort. In numeric sorting we arrange the element of data in ascending or descending order and in alphabetical or in lexicographical sort arrange the record in alphabetical order but no such sorting is there which can sort the record or data element on the basis of their priority's. This is well defined problem that if we want to sort data that is more important to us or we want to see the important data first, in comparison to non important data then what we should do?. All sorting algorithm which we have study until does not provide any priority to important data in the sorting, they sort all the data in equal speed .The main problem occur in the various sorting algorithms, which we have read when memory space is less and we want to sort my some important data.

There is some advantage and some disadvantage in every sorting algorithm, which we have read, and this disadvantage lead to discover new sorting algorithm. Some sorting algorithm are problem specific means they well suited for some specific problem and have disadvantage against another problem e.g. Insertion sort well suited for small number of data but heap sort well suited for large number of data. The problem of fundamental sorting algorithm like bubble, insertion, selection, quick etc have removed till some condition but have not removed for all condition so research work will go on until we will not get optimum solution for all sorting algorithms.

The two well-defined problems are as follow:

1. How we will display our important data prior to general data.
2. How we can perform sorting if there is less memory and we want to see my important data.

Chapter 6

Proposed Solution &Implementation

We have read various sorting algorithm and these sorting algorithms have some advantage and some disadvantage. Here we are going to solve two type problem mentions in the problem statement chapter; first one is that if we want to display important data prior to general data in sorted form and second is if there is limited memory then how we will perform sorting on important data.

The purposed algorithm for first problem is as follow:

6.1 Proposed Algorithm

Step1: Insert all the Record.
Step2: Assign priority to the specific Record.
Step3: Sort the specific Record on the basis of priority first.
Step 4: Display the specific Record.
Step5: Sort The remaining Record.
Step6: Display The remaining Record.
Step7: Exit

Table 6.1: Proposed Algorithm

The above mention sorting algorithm work well when there is requirement of display specific information prior to general information. The Proposed algorithm focuses mainly to display specific information prior to non-specific information and does not care about the memory. But in the sub part of algorithm we can display specific information and

also care about memory utilization .In the sub part of algorithm, we only store specific record not general, so problem is this that, it show only specific information not general information in sorting form.(Sub part of algorithm is subset of main algorithm given above and it will insert only specific record and display the record on the basis of priority and save memory)

6.1.1 Advantage: Saving memory, saving time, display data according requirement.

6.1.2 Disadvantage: Problem Oriented.

6.2 Case Study for Implementing Proposed Sorting

For the implementing to my proposed sorting algorithm we will consider one case. In this case we consider that we visit any university and we want the information of various employees in the staff and condition is that, we want to see some specific information prior to another information according to my choice.

Name	Department	Designation	Salary	Age	Highest Degree
RAM	CSE	LECT	25000	28	ME
SHYAM	CSE	PROF	40000	40	PHD
SURESH	CSE	PROF	40000	38	PHD
SHYAM	CSE	LECT	25000	45	BE
RAM	CSE	LECT	25000	30	M.TECH
MOHAN	CSE	LECT	25000	25	BE
RAM	CSE	PROF	40000	42	PHD
SURESH	CSE	LECT	25000	27	ME
AMAR	CSE	LECT	25000	25	BE

Table 6.2: CSE Staff Member

From above table we see that if we want to see the name or salary detail of professor first and later to the lecturer then simple sorting will sort name in alphabetical order and can

not differentiate to sort the first professor name and next lecturer name. To achieving this we will provide priority to the professor and apply priority sorting to sort the employee information. For implementing priority sorting an integer value will be given as priority to the essential data. Integer value 1 will indicate highest priority, 2 will indicate less priority than 1 and 3 will indicate less priority than 2and so on.

When we will provide the priority professor in the given table then from all the data first only those data will be sorted which have priority information. After sorting the priority data remaining will be sorted. The given table is very precise table, Priority sorting has much scope when there is vast amount of data and we want to see my specific data. Data warehouse is collection of vast amount of historical data .So getting important information from data warehouse this sorting help the data mining technique. From the given precise table, my aim is to show how important is priority sorting.

The table can have thousand records, so at that time if we want some specific staff detail first like suppose we want computer science department professor detail first and later to another department professor detail. To implementing this we have assign priority to the department and in this case priority will be assign to the department and highest priority will be given to the cse department. Priority sort not work alone it take help of another sorting algorithm according to requirements. The main purpose of priority sorting algorithm is to provide priority to the specific data, which we want to see first. After selecting specific data, which we want to sort first, the remaining sorting is done by either fundamental sorting algorithm or advance sorting algorithm.From above table if we want to see the professor detail first after that lecturer detail then we will assign priority to the professor record like given table in the below.

DESIGNATION	PRIORTY
PROF	1
LECT	0

Table 6.3: Assign Priority

Now seeing the priority the sorting algorithm display the result as follow

Name	Department	Designation	Salary	Age	Highest degree
RAM	CSE	PROF	40000	42	PHD
SHYAM	CSE	PROF	40000	40	PHD
SURESH	CSE	PROF	40000	38	PHD

Table 6.4: Sorted professor Record

The output table shows the details of professor. Here priority-sorting algorithm does only simple work, that is to find out priority related data and sort them by taking help of another sorting algorithm. Priority sorting sorts the important data according to our requirement. . In the above table, we have sort the detail of professor using name key.

Now we are going to develop source code for such a problem where only data will display according priority basis:

Consider another staff table and we want highest priority data display first.

JOB DESIGNATION	PRIORITY
CLERK	5
LECT	4
APROF	3
PROF	2
DEAN	1

Table 6.5: Designation Priority

NAME	JOB	DEPARTME	SALARY	MATERNA	AGE

	DESIGNATION	NT		L STATUS	
AJAY	LECT	CSE	25000	M	30
RAVINDER	LECT	CSE	25000	M	30
ROHIT	LECT	EC	20000	U	25
RAINA	DEAN	-	50000	M	50
OEM PAKESH	PROF	PHY	40000	M	50
RAJU	APROF	CSE	35000	U	24
NK VERMA	DEAN	--	50000	U	52
RAJU	CLERK	CSE	15000	U	22
MANINDER	APROF	CSE	35000	M	45
HARVINDER	DEAN	-	55000	M	60
ARUN	LECT	EC	20000	U	25
ARUN	DEAN	--	55000	M	52
PUNAM	LECT	CSE	27000	M	28
ROMI	PROF	EC	35000	U	25
RICHA	APRO	CSE	28000	U	25

Table 6.6: University Staff detail

6.3 Source Code for Implementing Proposed Sorting

```
#include<stdio.h>
#include<conio.h>
# define max 5
struct data
{
char job[max];
int prno;
int ord;
};
```

```
struct pque
{
struct data d[max];
int front;
int rear;
};
void initpque(struct pque*)
void add(struct pque*,struct data);
struct data delete(struct pque *);
void main()
{
struct pque q;
struct pque q;
struct data dt,temp;
int i ,j=0;
clrscr();
initpque(&q);
printf("enter the job description(max 4 char ) and its pr4iorty\n");
printf("lower priorty no and higher priorty number\n");
printf("job priorty\n");
for(i=0;i<max;i++);
{
scanf("%s%d",&dt.job,&dt.prno);
dt.ord=j++;
add(&q,dt);
}
printf("\n');
printf("process the job prioty wise\n");
printf("job\tpriorty\n");
for(i=0;i<max;i++)
{
```

```
temp=delete(&q);
printf("%s\t%d\n",temp.job,temp.prno);
}


printf("\n");
getch();    }
/* initalise data member*/
void initque(struct pque *pq)
{
int i;
pq->front=pq->rear=-1;
for(i=0;i<max;i++)
{
strcpy(pq->d[i].job,'\0') ;
pq->d[i].prno=pq->d[i].ord=0;
}
}
/* add item to for priorty sorting*/
void add(struct pque *pq,struct data dt)
{
struct data temp;
int i , j;
if(pq->rear==max-1)
{
printf("\nqueue is full ');
return;
}
pq->rear++;
pq->d[pq->rear]=dt;
if (pq->front==-1)
```

```
pq->front=0;
for(i=pq->front;i<=pq->rear;i++)
{
for(j=i+1;j<=pq->rear;j++)
{
if(pq->d[i].prno>pq->d[j].prno)
{
temp =pq->d[i];
pq->d[i]=pq->d[j];
pq->d[j]=temp;
}
else
{
if(pq->d[i].prno==pq->d[j].prno)
{

if(pq->d[i].ord>pq->d[j].ord)
{
temp=pq->d[I];
pq->d[I]=pq->d[j];
pq->d[j]=temp;
}}}}}}
/*Perform sorting*/
struct data delete(struct pque * pq)
{
struct data t;
struct(t.job,"");
t.prno=0;
t.ord=0;
if(pq->front==-1)
{
```

```
printf"\n queue is empty\n");
return t;
}
t=pq->d[pq->front];
pq-.d[pq->front]=t;
if(pq->front==pq->rear)
pq->front=pq->rear=-1;

else
pq->front++;
return t;
}
```

output:

Enter job description and its priority

lower the priority number ,higher the priority

Job	**Priority**
Prof	2
Lect	4
Dean	1
Aprof	3
Clerk	5

Process jobs priority wise

Job	**Priority**
Dean	1
Prof	2
Aprof	3
Lect	4
Clerk	5

6.4 Results

The result of proposed sorting algorithm on the basis of priority is given below. In this result, we are seeing that we have gotten most prior data first in comparison of less prior data. The priority sorting only pickup those data that have priority tag if there will be record that do not have any priority tag then by default that will get lowest priority entry. The result shows that dean have most priority show it will display first and for sorting among various record of dean, priority sorting take help of fundamental sorting algorithm or advance sorting algorithm.

JOB	PRIORITY
DEAN	1
PROF	2
APROF	3
LECT	4
CLERK	5

Table 6.7: Sorted job on the basis of priority

Chapter 7
Conclusion and Future Work

7.1Conclusion

In this thesis we have study about various sorting algorithms and their comparison. There is advantage and disadvantage in every sorting algorithmic .The fundamental sorting algorithm are basic sorting algorithm and we have try to show this how disadvantage of fundamental sorting algorithm have removed in advance sorting algorithm. We have compared the various sorting algorithm on the basis of various factors like complexity, memory required, stability etc. After the study of all various sorting algorithm we find that there is no such algorithm, which work in this way that to sort the specific or important detail first and after that remaining. So solve this problem we have proposed sorting algorithm, which work on the basis of the priority. For implementing to the proposed algorithm we have consider the case study of the university staff member. In this case study we see that after applying proposed sorting we get the specific data first then general data if required.

7.2Future work

In this thesis we have study about various sorting algorithms and each sorting algorithm have some advantage and some disadvantage, so my first aim is to remove the disadvantage of various sorting algorithms. We have also seen that many algorithm are problem oriented so we try to make it global oriented. In my proposed sorting we have implemented it using static data structure but we want to implement it using data dynamic data structure. So we can say that there are many future works as follow.

- Remove disadvantage of various fundamental sorting and advance sorting
- Make problem oriented sorting to global oriented.
- Implementation of proposed sorting algorithm using dynamic data structure.

At the last we want to say that there is huge scope of the sorting algorithm in the future, and to find optimum-sorting algorithm, the work on sorting algorithm will go on forever.

References

[1] Y. Han "Deterministic sorting in O(nlog log n) time and linear space", Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, Montreal, Quebec, Canada, 2002, p.602-608.

[2]. M. Thorup "Randomized Sorting in O (n log log n) Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations", Journal of Algorithms, Volume 42, Number 2, February 2002, p. 205-230 .

[3] Y. Han, M. Thorup, "Integer Sorting in O(n √(log log n) Time and Linear Space", Proceedings of the 43rd Symposium on Foundations of Computer Science, 2002, p. 135-144.

[4] P. M. McIlroy, K. Bostic and M. D. McIlroy, "Engineering radix sort", Computing Systems, 2004, p.224-230.

[5] M. D. McIlroy, "A killer adversary for quick sort", Software--Practice and Experience, 1999, p.123-145.

[6]http://209.85.141.104/search?q=cache:iBhfoE92zZoJ:www.softpanorama.org/Algorithms/sorting.shtml+introduction+of+sorting+algorithm&hl=en&ct=clnk&cd=9&gl=in, accessed on 20 March, 2007.

[7].http://209.85.141.104/search?q=cache:aEKNhmWd5XEJ:en.wikipedia.org/wiki/Sorting_algorithm+sorting+algorithm&hl=en&ct=clnk&cd=3&gl=inclassification recursion and comparison, accessed on 30 March, 2007.

[8] http://www.nist.gov/dads/HTML/bidirectionalBubbleSort.html, accessed on 2Aprial, 2007

[9]http://209.85.175.104/search?q=cache:yO0E_OpfQ6MJ:www.nist.gov/dads/HTML/combSort.html+comb+sort&hl=en&ct=clnk&cd=3&gl=in, accessed on 10 April, 2007.

[10]http://209.85.175.104/search?q=cache:4G6ZbnNtL8AJ:www.nist.gov/dads/HTML/gnomeSort.html+gnome+sort&hl=en&ct=clnk&cd=3&gl=in, accessed on 20 April, 2007.

[11]http://stochastix.wordpress.com/2007/11/10/smooth-sorting/, accessed on 22 April, 2007.

[12]http://209.85.175.104/search?q=cache:B3vHb9UHcAgJ:www.nist.gov/dads/HTML/strandSort.html+strand+sort&hl=en&ct=clnk&cd=3&gl=in, accessed on 25 April, 2007.

[13]http://209.85.175.104/search?q=cache:wBLyDu9hepcJ:www.nist.gov/dads/HTML/bucketsort.html+bucket+sort&hl=en&ct=clnk&cd=3&gl=in, accessed 28 April, 2007.

[14]http://en.org/Library_sort, accessed on 30 April, 2007.

[15]http://en.wikipedia.org/wiki/Topological_sohttp://en.wikipedia.org/wiki/Patience_so, accessed on 2 May,2007.

[16]rtingrtinghttp://en.org//Pigeonhole_sort, accessed on 8 May, 2007.

[17]http://en.org/Tally _sort, accessed on 9 May, 2007.

[18]http://209.85.175.104/search?q=cache:lGRcZjOxFikJ:www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/postmanSort/postman.html+postman+sort&hl=en&ct=clnk&cd=1&gl=in, accessed on 10 May, 2007.

[19]http://72.14.235.104/search?q=cache:EILa0seO7xEJ:linux.wku.edu/~lamonml/algor/sort/bubble.html+bubble+sort&hl=en&ct=clnk&cd=4&gl=in.accessed on 15 May, 2007.

[20]http://72.14.235.104/search?q=cache:m1tGbHhCYG0J:linux.wku.edu/~lamonml/algor/sort/insertion.html+insertion+sort&hl=en&ct=clnk&cd=3&gl=in, accessed on 16 May, 2007.

[21]http://72.14.235.104/search?q=cache:oRR7XxyvYMJ:linux.wku.edu/~lamonml/algor/sort/selection.html+selection+sort&hl=en&ct=clnk&cd=3&gl=in, accessed on 20 May, 2007.

[22]http://72.14.235.104/search?q=cache:PIMYa5D2gvgJ:linux.wku.edu/~lamonml/algor/sort/merge.html+merge+sort&hl=en&ct=clnk&cd=9&gl=in, accessed on 22 May, 2007.

[23]http://209.85.141.104/search?q=cache:9fSMRQf7t0J:linux.wku.edu/~lamonml/algor/sort/shell.html+shell+sort&hl=en&ct=clnk&cd=3&gl=in, accessed on 22 May, 2007.

[24]http://72.14.235.104/search?q=cache:dAIbrxx7MsJ:www.nist.gov/dads/HTML/heapSort.html+heap+sort&hl=en&ct=clnk&cd=8&gl=in, accessed on 26 May, 2007.

[25]http://www.google.co.in/search?hl=en&q=bucket+sort&btnG=Search&meta=, accessed on 28 May, 2007.

[26]http://72.14.235.104/search?q=cache:KFnlEPMC1VIJ:www.brpreiss.com/books/opus4/html/page518.html+radix+sort&hl=en&ct=clnk&cd=6&gl=in accessed on 22 March, 2007.

[27]http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/countingSort.htm, accessed on 24 March, 2007.

[28]http://209.85.175.104/search?q=cache:PYgTbyrtlVIJ:www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK/NODE30.HTM+application+of+sorting&hl=en&ct=clnk&cd=1&gl=in, accessed on 10 Feb, 2007.

[29]http://pages.cs.wisc.edu/~hasti/cs367-common/notes/COMPLEXITY.html, accessed on 16 Feb, 2007.

[30]http://www.nist.gov/dads/HTML/bigOnotation.html, accessed on 18 Feb,2007.

[31]http://www.nist.gov/dads/HTML/theta.html theta, accessed on 20 Feb, 2007.

[32] http://www.nist.gov/dads/HTML/omegaCapital.html , accessed on 28 Feb,2007

List of Publications

1. Ramesh Chand Pandey, Shivani Goel "Introduction of various sorting algorithms with their comparisons and their advantage in data warehouse and data mining" ,accepted in PIMR Third National IT Conference- 2008, Indore.

2. Ramesh Chand Pandey, Shivani Goel "Study and comparison of various advance sorting Algorithms" communicated with National conference on Emerging Trends In Information Technology, NCEIT-08 organized by the The Institute of engineering & Information Technology**,** Solan, Himachal Pradesh.