CHAPTER 1

PROJECT INTRODUCTORY

IN THIS CHAPTER

1. 1. INTRODUCTION

From time to time people ask the ageless question: Which sorting algorithm is the fastest? This question doesn't have an easy or unambiguous answer, however. The speed of sorting can depend quite heavily on the environment where the sorting is done, the type of items that are sorted and the distribution of these items.

For example, sorting a database which is so big that cannot fit into memory all at once is quite different from sorting an array of 100 integers. Not only will the implementation of the algorithm be quite different, naturally, but it may even be that the same algorithm which is fast in one case is slow in the other. Also sorting an array may be different from sorting a linked list, for example.

In this study I will only concentrate on sorting items in an array in memory using comparison sorting (because that's the only sorting method that can be easily implemented for any item type, as long as they can be compared with the less-than operator).

1. 2. OBJECTIVES

In our modern era of organization of data and information into the memory is very important and keep the data into an organized manner so that when requited it could be fetched out from the location. This organization of data is done by using various sorting algorithms.

In this project the goal is to study & analysis different types of sorting algorithms and to take an unordered set of comparable data items and arrange them in order.

The overall scope of this project is to develop an algorithm which is better or effective as per as complexity concern.

1.3. PROJECT DESCRIPTION

This Project has been created by the students of 3rd Year **Diploma in Computer Science & Technology** as a part of their final year project.

This project has been created by a team of 7 members working together in a group to work for improvement and management of **CSICAISA (COMPARATIVE STUDY AND IMPLEMENTATION OF CONVENTIONAL ALGORITHMS & INNOVATION OF SORTING ALGORITHM)** for the detail about study and analysis of sorting algorithms.

Our aim is to study and analysis the different type of sorting algorithms and reduces complexity. The overall objective is to take an unordered set of comparable data items and arrange them in an ordered manner. And also try to develop an algorithm which is better or effective as per as complexity concern.

1. 3.1. PROJECT IN BRIEF

This is a **Comparative Study and Implementation of Conventional Algorithms & Innovation of Sorting Algorithm** to be developed in C. The user should be presented with a well defined login page where he/she will be allowed to give predefined password to get along with the system. There should be a proper verification process otherwise the user should be redirected to the login page. Once the login is successful there will be a welcome page and then options like list of particular known algorithms, then also some of the newly invented algorithm, comparison of different algorithms and lots more. The project may be enhanced by improving the user interface and inventing other algorithms.

1. 3.2. PROJECT IN DETAIL

Area : Sorting Algorithm Analysis.

Software Name : Comparative Study and Implementation of Conventional Algorithms & Innovation of Sorting Algorithm.

Group Strength : Seven(7).

Responsibility : Project development, testing and Implementation.

Front End : C Language.

Back End : No such data base ; Random input from the users.

Project Type : Analytical Project.

Under By : Mr. Taufique Ahmmad Gazi.

1. 4. FEASIBILITY ANALYSIS

Whatever we think need not be feasible. It is wise to think about the feasibility of any problem we undertake. Feasibility is the study of impact, which happens in the organization by the development of a system. The impact can be either positive or negative. When the positive dominates the negatives then the system is considered is feasible. Here the feasible study can be performed in two ways such as Technical Feasibility and Economical Feasibility.

1. 4.1. TECHNICAL FEASIBILITY

We can strongly say that it is technically feasible, since there will not be much difficulty in getting required resources for the development and maintaining the system as well. All the resources needed for the development of the software as well as the same is available in the organization itself.

1. 4.2. ECONOMICAL FEASIBILITY

Development of this application is highly economically feasible. The organization needed not spend much money for the development of the system already available. The only thing is to be done is making an environment for the development with an effective supervision. If we are doing so, we can attain the maximum usability of the corresponding resources. Even after the development, the organization will not be in a condition to invest more in the organization. Therefore, the system is economically feasible.

CHAPTER 2

SYSTEM ANALYSIS

IN THIS CHAPTER

2. 1. SCHEDULING CRITERIA

Proto type model are used in this project. The goal of prototyping based development is to counter the first two limitations of the waterfall model discussed earlier. The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding and testing. But each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.

2. 2. SYSTEM IMPLEMENTATION

The systems implementation is a process of construction and delivery phases of the life cycle. Systems implementation is the construction of the new system and the delivery of that system into production.

Stages of Implementation Are:

I. Analysis of the project

II. GUI Design

III. Coding (Individual testing method)

IV. Testing

V. System testing

VI. System maintenance.

2. 3. PLATFORM USE

2. 3.1. HARDWARE REQUIREMENTS

- Hard Disk (minimum 80 MB free space)
- RAM (128 MB or more)
- Processor (Intel, AMD minimum 1 GHz speed)
- Video memory(minimum 16MB)
- Keyboard

2. 3.1. SOFTWARE REQUIREMENTS

- Operating System – Windows XP
- Turbo C++ 3.0

CHAPTER 3

BASIC KNOWLEDGE

IN THIS CHAPTER

3. 1. BASIC IDEA ABOUT SORTING

The purpose of any sort is to permute some set of items that we'll call records so that they are sorted according to some ordering relation. In general, the ordering relation looks at only part of each record, the key. The records may be sorted according to more than one key, in which case we refer to the primary key and to secondary keys.

This distinction is actually realized in the ordering function: record A comes before B if either A's primary key comes before B's, or their primary keys are equal and A's secondary key comes before B's. One can extend this definition in an obvious way to hierarchy of multiple keys. For the purposes of these Notes, I'll usually assume that records are of some type Record and that there is an ordering relation on the records we are sorting. I'll write before (A, B) to mean that the key of A comes before that of B in whatever order we are using.

Although conceptually we move around the records we are sorting so as to put them in order, in fact these records may be rather large. Therefore, it is often preferable to keep around pointers to the records and exchange those instead. If necessary, the real data can be physically re-arranged as a last step. In Java, this is very easy of course, since \large" data items are always referred to by pointers.

3. 1.1. SORTING

Sorting algorithms are the basic foundations of practical Computer Science, and therefore, the analysis and design of useful sorting algorithms has remained one of the most important research area in the field. Despite the fact that, several new sorting algorithms being introduced, the large number of programmers in the field depends on one of the comparison-based sorting algorithms: Bubble, Insertion, Selection sort etc. The new algorithms are usually not being used as much because they are not as general purpose as comparison-based sorting algorithms, usually require more alteration to work with new classes and data types and in large cases, do not perform as well as expected due to poorer

locality of reference caused by linear passes through the array. It is necessary for a new algorithm to be acknowledged and used in the field of Computer Science, the process must be shown to have as good as performance to the abovementioned sorting algorithms and be easy to use, implement, and debug. Better performance will get programmers interested in the algorithm, and ease-of-use will be the final determinate in changing the programmer's sorting preference. The easiest, most time efficient way to accomplish a task is usually preferred, and the comparison-based sorts tinted here offer excellent performance and adaptability to any type of record or data type to be sorted.

3. 1.2. THE PROBLEM

We are given a list *L* of *n* elements and we wish arrange the elements in order. Usually the elements to be sorted are part of a larger structure called a record. Each record contains a key which is the value to be sorted. As the keys are rearranged during the sorting process, the data associated with the keys is also rearranged.

Key	Additional data

3. 1.3. TERMINOLOGY

1. An **Internal Sort** assumes that all of the data is stored in the computers main memory.
2. An **External Sort** assumes that large sets are stored in slower, external storage devices.
3. A sorting algorithm is called **in-place** if the amount of extra space it uses does not change as the input size changes.
4. A sorting method is **Stable** if equal keys remain in the same relative order as they were in the original data.

3. 2. COMPLEXITY

Complexity has turned out to be very difficult to define. The dozens of definitions that have been offered all fall short in one respect or another, classifying something as complex which we intuitively would see as simple, or denying an obviously complex phenomenon the label of complexity. Moreover, these definitions are either only applicable to a very restricted domain, such as computer algorithms or genomes, or so vague as to be almost meaningless.

To find real complexity on the scale dimension, we may look at the human body: if we zoom in we encounter complex structures at least at the levels of complete organism, organs, tissues, cells, organelles, polymers, monomers, atoms, nucleons, and elementary particles. Though there may be superficial similarities between the levels, e.g. between organs and organelles, the relations and dependencies between the different levels are quite heterogeneous, characterized by both distinction and connection, and by symmetry breaking.

3. 3. COMPLEXITY MEASUREMENT

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

What effects run time of an algorithm?

(a) Computer used, the harware platform.

(b) Representation of abstract data types (ADT's).

(c) Efficiency of compiler.

(d) Competence of implementer (programming skills).

(e) Complexity of underlying algorithm.

(f) Size of the input.

We will show that of those above (e) and (f) are generally the most significant time for an algorithm to run t(n) a function of input. However, we will attempt to characterize this by the size of the input. We will try and estimate the **Worst Case**, and sometimes the **Best Case**, and very rarely the **Average Case**.

3. 3.1. WORST CASE

Worst case is the maximum run time, over all inputs of size n. That is, we only consider the "number of times the principle activity of that algorithm is performed".

3. 3.2. AVERAGE CASE

Average case is the most useful measure. It might be the case that worst case behaviour is pathological and extremely rare, and that we are more concerned about how the algorithm runs in the general case. Unfortunately this is typically a very difficult thing to measure. Firstly, we must in some way be able to define by what we mean as the "average input of size n". We would need to know a great deal about the distribution of cases throughout all data sets of size n. Alternatively we might make a possibly dangerous assumption that all

data sets of size n are equally likely. Generally, in order to get a feel for the average case we must resort to an empirical study of the algorithm, and in some way classify the input (and it is only recently with the advent of high performance, low cost computation, that we can seriously consider this option).

3. 3.3. BEST CASE

In this case we look at specific instances of input of size n. For example, we might get best behaviour from a sorting algorithm if the input to it is already sorted.

3. 4. ASYMPTOTIC NOTATION

A problem may have numerous algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run. Or, more accurately, you need to be able to judge how long two solutions will take to run, and choose the better of the two. You don't need to know how many minutes and seconds they will take, but you do need some way to compare algorithms against one another.

Asymptotic complexity is a way of expressing the main component of the cost of an algorithm, using idealized units of computational work. Consider, for example, the algorithm for sorting a deck of cards, which proceeds by repeatedly searching through the deck for the lowest card. The asymptotic complexity of this algorithm is the square of the number of cards in the deck. This quadratic behavior is the main term in the complexity formula, it says, e.g., if you double the size of the deck, then the work is roughly quadrupled.

The exact formula for the cost is more complex, and contains more details than are needed to understand the essential complexity of the algorithm. With our deck of cards, in the worst case, the deck would start out reverse-sorted, so our scans would have to go all the way to the end. The first scan would involve scanning 52 cards, the next would take 51, etc. So the cost formula is 52 + 51 +.....+ 2. Generally, letting N be the number of cards, the formula is 2 + ... + N, which equals

$$((N) \times (N+1)/2) - 1 = ((N^2 + N)/2) - 1 = (1/2)N^2 + (1/2)N - 1$$

But the N^2 term dominates the expression, and this is what is key for comparing algorithm costs. (This is in fact an expensive algorithm; the best sorting algorithms run in sub-quadratic time.)

Asymptotically speaking, in the limit as N tends towards infinity, 2 + 3 + .,,,.. + N gets closer and closer to the pure quadratic function (1/2)N^2. And what difference does the constant factor of 1/2 make, at this level of abstraction? So the behavior is said to be $O(n^2)$.

3. 4.1. BIG - O NOTATION

The Big-O is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

More formally, for non- negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) = cg(n)$, then f (n) is Big-O of $g(n)$.

This is denoted as "*f (n) = O (g (n))*". If graphed, *g (n)* serves as an upper bound to the curve you are analyzing, *f (n*).

Note that if f can take on finite values only (as it should happen normally) then this definition implies that there exists some constant *C* (potentially larger than *c*) such that for all values of *n*, *f (n)* $\max_{1 \le n \le n_0} f(n)/g(n)$priate value for *C* is the maximum of *c* and

3. 4.2. BIG - OMEGA NOTATION

For non-negative functions, *f (n)* and *g(n)*, if there exists an integer and a constant $c > 0$ such that for all integers $n > n_0$, *f (n) = cg(n)*, then *f (n)* is omega of *g(n)*. **This is denoted as "*f(n) = Ω(g(n))*"**.

This is almost the same definition as Big Oh, except that "*f(n) = cg(n*)", this makes g(n) a lower bound function, instead of an upper bound function. It describes the best that can happen for a given data size.

3. 4.3. THETA NOTATION

For non- negative functions, *f(n)* and *g(n)*, *f(n)* is theta of *g(n)* if and only if *f(n) = O(g(n))* and *f (n) = Ω(g(n))*. This is denoted as "***f (n) = Θ(g(n))***".

This is basically saying that the function, *f (n)* is bounded both from the top and bottom by the same function, *g(n)*.

3. 4.4. LITTLE - O NOTATION

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little *o* of *g(n)* if and only if *f(n)=O(g(n))*, but *f (n) ≠ Θ (g(n))*. This is denoted as "***f (n) = o(g(n))***".

This represents a loose bounding version of Big *O*. *g(n)* bounds from the top, but it does not bound the bottom.

3. 4.5. LITTLE - OMEGA NOTATION

For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little omega of *g(n)* if & only if *f(n)=Ω(g(n))* but *f (n) ≠ Θ (g(n))*. This is denoted as "***f (n) = ω(g(n))***".

Much like Little Oh, this is the equivalent for Big Omega. g(n) is a loose lower boundary of the function *f (n)*; it bounds from the bottom, but not from the top.

CHAPTER 4

DIFFERENT SORTINGS

IN THIS CHAPTER

4.1. THEORETICAL BACKGROUND

4.1.1. BUBBLE SORT

IDEA

For example, consider the array:

45, 67, 12, 34, 25, 39

In the first step, the focus is on the first two elements (in **bold**) which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45, **67**, 12, 34, 25, 39

Then the focus moves to the elements at index 1 and 2 which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, 67, 34, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:
45, 12, 34, 67, 25, 39
45, 12, 34, 25, 67, 39
45, 12, 34, 25, 39, 67

ALGORITHM

```
for i = 1:n,
swapped = false
   for j = n:i+1,
   if a[j] < a[j-1],
   swap a[j,j-1]
   swapped = true
   → invariant: a[1..i] in final position
   break if not swapped
end
```

PSEUDO CODE

```
func bubblesort( var a as array )
   for i from 1 to N
      for j from 0 to N - 1
        if a[j] > a[j + 1]
          swap( a[j], a[j + 1] )
end func
```

COMPLEXITY ANALYSIS

Best Case : n
Worst Case : n^2
Average Case : n^2

ADVANTAGE AND DISADVANTAGE

A bubble sort is a sort where adjacent items in the array or list are scanned repeatedly, swapping as necessary, until one full scan performs no swaps. Advantage is simplicity. Disadvantage is that it can take N scans, where N is the size of the array or list, because an out of position item is only moved one position per scan. This can be mitigated somewhat by starting with a swap gap of greater than one (typically N/2), scanning until no swaps occur, then halving the gap and repeating until the gap is one. This, of course, is no longer a bubble sort - it is a merge exchange sort.

4.1.2. BUCKET SORT

IDEA

Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

7, 5, 2, 4, 3, 9
5, **7, 2**, 4, 3, 9
5, 2, **7, 4**, 3, 9
5, 2, 4, **7, 3**, 9
5, 2, 4, 3, **7, 9**
5, 2, 4, 3, 7, 9

ALGORITHM

```
void bubbleSort(int ar[]) {
   for (int i = (ar.length - 1); i >= 0; i--) {
        for (int j = 1; j ≤ i; j++) {
        if (ar[j-1] > ar[j]) {
           int temp = ar[j-1];
           ar[j-1] = ar[j];
           ar[j] = temp;
 } } } }
```

PSEUDO CODE

```
function bucketSort(array, n) is
  buckets ← new array of n empty lists
  for i = 0 to (length(array)-1) do
   insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
   nextSort(buckets[i])
  return the concatenation of buckets[0], ...., buckets[n-1]
```

COMPLEXITY ANALYSIS

Best case :
Worst case : o(n+r)
Average case : o(n+r)

ADVANTAGE AND DISADVANTAGE

Bucket sort, or bin sort, is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, and is a cousin of radix sort in the most to least significant digit flavour. Bucket sort is a generalization of pigeonhole sort. Since bucket sort is not a comparison sort, the O(n log n) lower bound is inapplicable. The computational complexity estimates involve the number

of buckets. Bucket sort works as follows:

- Set up an array of initially empty "buckets."
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array.

4.1.3. COCKTAIL SORT

IDEA

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 , shuffle sort , shut t le 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 a bubble sort in that it sorts in both directions on each pass through the list. This sorting algorithm is only marginally more difficult to implement than a bubble sort, and solves the problem of turtles in bubble sorts.

ALGORITHM

1 for *i* = 1 to *k*

2 do *c[i]* = 0

3 for *j* = 1 to *length[A]*

4 do *C[A[j]] = C[A[j]]* + 1

5 >*C[i]* now contains the number of elements equal to *i*

6 for *i* = 2 to *k*

7 do *C[i] = C[i] + C[i-1]*

8 >*C[i]* now contains the number of elements less than or equal to *i*

9 for *j = length[A]* downto 1

10 do *B[C[A[j]]] = A[j]*

11 *C[A[j]] = C[A[J]]* – 1

PSEUDO CODE

The simplest form of cocktail sort goes through the whole list each time:

```
procedure cocktailSort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length( A ) - 2 do:
      if A[ i ] > A[ i + 1 ] then // test whether the two elements are in the wrong order
        swap( A[ i ], A[ i + 1 ] ) // let the two elements change places
        swapped := true
      end if
    end for
    if swapped = false then
      // we can exit the outer loop here if no swaps occurred.
      break do-while loop
    end if
```

COMPLEXITY ANALYSIS

Best Case : o(n)
Worst Case : o(n^2)
Average Case : o(n^2)

ADVANTAGE AND DISADVANTAGE

Another optimize action can be that the algorithm remembers where the last actual swap has been done. In the next iteration, there will be no swaps beyond this limit and the algorithm has shorter passes. As the Cocktail sort goes bidirectional, the range of possible swaps, which is the range to be tested, will reduce per pass, thus reducing the overall running time.

4.1.4. COMB SORT

IDEA

Having the following list, let's try to use comb sort to arrange the numbers from lowest to greatest:

Unsorted list:

5	7	9	10	3	1	4	8	2	6

Iteration 1, gap = 8. The distance of 8 from the first element 5 to the next element, leads to the element of 2, these numbers are not in the right order so they have to swap. Also, the same distance is between 7 and 6 which also are not in the right order, so again a swap is required:

2	7	9	10	3	1	4	8	5	6
2	6	9	10	3	1	4	8	5	7

Iteration 2, gap =6, the elements that were compare on each line are 2 with 4, 6 with 8, 5 with 9 and 7 with 10:

2	7	9	10	3	1	4	8	5	7
2	6	9	10	3	1	4	8	5	7
2	6	5	10	3	1	4	8	9	7
2	6	5	7	2	1	4	8	9	10

Iteration 3, gap = 4:

2	7	9	10	3	1	4	8	5	10
2	1	9	10	3	6	4	8	5	10
2	1	4	10	3	6	5	8	9	10
2	1	4	7	3	6	5	8	9	10
2	1	4	7	3	6	5	8	9	10
2	1	4	7	3	6	5	8	9	10

Iteration 4, gap = 3:

2	1	4	7	3	6	5	8	9	10
2	1	4	7	3	6	5	8	9	10
2	1	4	7	3	6	5	8	9	10
2	1	4	5	3	6	7	8	9	10
2	1	4	5	3	6	5	8	9	10
2	1	4	5	3	6	5	8	9	10
2	1	4	5	3	6	7	8	9	10

Iteration 5, gap =2:

2	1	4	5	3	6	7	8	9	10
2	1	4	5	3	6	7	8	9	10
2	1	3	5	4	6	7	8	9	10
2	1	3	5	4	6	7	8	9	10
2	1	3	5	4	6	7	8	9	10
2	1	3	5	4	6	7	8	9	10
2	1	3	5	4	6	7	8	9	10
2	1	3	5	4	6	7	8	9	10

Iteration 6, gap =1:

1	2	3	5	4	6	7	8	9	10
1	2	3	5	4	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Iteration 7 : since the items are sorted, no swaps will be made and the algorithm ends its execution.

ALGORITHM

```
comb_sort(list of t)
   gap = list.count
   temp as t
   swapped = false
   while gap > 1 or not swapped
      swapped = false
      if gap > 1 then
         gap = floor(gap/1.3)
         i = 0
    while i + gap < list.count
         if list(i) > list(i + gap)
         temp = list(i) // swap
         list(i) = list(i + gap)
         list(i + gap) = temp
         i += 1
```

PSEUDO CODE

```
function combsort(array input)
   gap := input.size //initialize gap size
   loop until gap = 1 and swaps = 0
//update the gap value for a next comb. Below is an example
   gap := int(gap / 1.25)
      if gap < 1
 //minimum gap is 1
      gap := 1
      end if
      i := 0
      swaps := 0 //see Bubble Sort for an explanation
//a single "comb" over the input list
      loop until i + gap >= input.size //see Shell sort for similar idea
         if input[i] > input[i+gap]
            swap(input[i], input[i+gap])
            swaps := 1 // Flag a swap has occurred, so the
// list is not guaranteed sorted
         end if
         i := i + 1
      end loop
   end loop
end function
```

COMPLEXITY ANALYSIS

Best Case : o(n)
Worst Case : o(n^2)
Average Case : o(nlogn)

ADVANTAGE AND DISADVANTAGE

Comb sort can be made more effective if the sorting method is changed once the gaps reach numbers small enough. For example, once the gap reaches a size of about 10 or smaller, stopping the comb sort and doing a simple gnome sort or Cocktail Sort, or, even better, an Insertion Sort will increase the sort's overall efficiency.
Another advantage of this method is that there is no need to keep track of swaps during the sort passes to know if the sort should stop or not.

4.1.5. COUNTING SORT

IDEA

Counting sort is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set. Integers which lie in a fixed interval, say k1 to k2, are examples of such items. The algorithm proceeds by defining an ordering relation between the items from which the set to be sorted is derived (for a set of integers, this relation is trivial).Let the set to be sorted be called A. Then, an auxiliary array with size equal to the number of items in the superset is defined, say B. For each element in A, say e, the algorithm stores the number of items in A smaller than or equal to e in B(e). If the sorted set is to be stored in an array C, then for each e in A, taken in reverse order, C[B[e]] = e. After each such step, the value of B(e) is decremented.

The algorithm makes two passes over A and one pass over B. If size of the range k is smaller than size of input n, then time complexity=O(n). Also, note that it is a stable algorithm, meaning that ties are resolved by reporting those elements first which occur first.

This visual demonstration takes 8 randomly generated single digit numbers as input and sorts them. The range of the inputs are from 0 to 9.

ALGORITHM

COUNTING SORT(A,B,k) algorithm

1 for i = 1 to k

2 do c[i] = 0

3 for j = 1 to length[A]

4 do C[A[j]] = C[A[j]] + 1

5 >C[i] now contains the number of elements equal to i

6 for i = 2 to k

7 do C[i] = C[i] + C[i-1]

8 >C[i] now contains the number of elements less than or equal to i

9 for j = length[A] downto 1

10 do B[C[A[j]]] = A[j]

11 C[A[j]] = C[A[J]] – 1

PSEUDO CODE

```
countingsort(A[], B[], k)
        for i = 1 to k do
          C[i] = 0
        for j = 1 to length(A) do
          C[A[j]] = C[A[j]] + 1
        for 2 = 1 to k do
          C[i] = C[i] + C[i-1]
        for j = 1 to length(A) do
          B[C[A[j]]] = A[j]
          C[A[j]] = C[A[j]] – 1
```

COMPLEXITY ANALYSIS

Best Case : -
Worst Case : o(n+r)
Average Case : o(n+r)

ADVANTAGE AND DISADVANTAGE

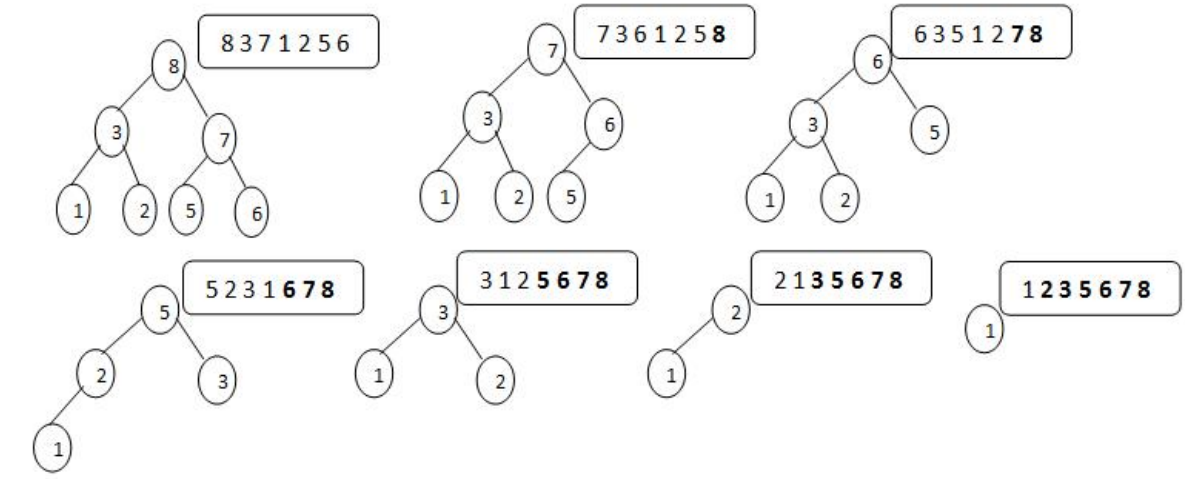
The biggest advantage of counting sort is its complexity –where is the size of the sorted array and is the size of the helper array (range of distinct values).

It has also several disadvantages – if non-primitive (object) elements are sorted, another helper array is needed to store the sorted elements. Second and the major disadvantage is that counting sort can be used only to sort discrete values (for example integers), because otherwise the array of frequencies cannot be constructed.

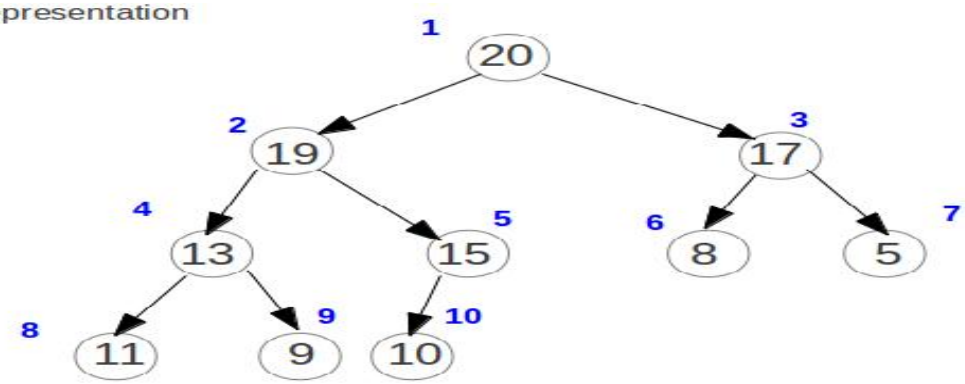
4.1.6. HEAP SORT

IDEA

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



Tree Representation



Array Representation

ALGORITHM

1. Build a heap with the sorting array, using recursive insertion.
2. Iterate to extract n times the maximum or minimum element in heap and heapify the heap.
3. The extracted elements form a sorted subsequence.

PSEUDO CODE

```
Heapify(A, 1)
BuildHeap(A as array)
n = elements_in(A)
for i = floor(n/2) to 1
Heapify(A,i)
Heapify(A as array, i as int)
left = 2i
right = 2i+1
if (left <= n) and (A[left] > A[i])
max = left
else
max = i
if (right<=n) and (A[right] > A[max])
max = right
if (max != i)
swap(A[i], A[max])
Heapify(A, max)
```

COMPLEXITY ANALYSIS

Best Case : o(n logn)
Worst Case : o(n logn)
Average Case : o(n logn)

ADVANTAGE

1. **Efficiency:** The Heap sort algorithm is very efficient. While other sorting algorithms may grow exponentially slower as the number of items to sort increase, the time required to perform Heap sort increases logarithmically.
2. **Memory Usage:** The Heap sort algorithm can be implemented as an in-place sorting algorithm. This means that its memory usage is minimal.
3. **Simplicity:** The Heap sort algorithm is simpler to understand.
4. **Consistency:** The Heap sort algorithm exhibits consistent performance. This means it performs equally well in the best, average and worst cases.

DISADVANTAGE

Heap sort is an in- place algorithm, but is not a stable sort.

4.1.7. INSERTION SORT

IDEA

The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the item under consideration is underlined. The item that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1
37 4 9 5 2 6 1
3 74 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 7 95 2 6 1
3 4 5 7 9 2 6 1
2 3 4 5 7 9 6 1
2 3 4 5 6 7 9 1
1 2 3 4 5 6 7 9

ALGORITHM

```
function insertionSort(array A)
   for i from 1 to length[A]-1 do
      value := A[i]
      j := i-1
      while j >= 0 and A[j] > value do
         A[j+1] := A[j]
         j := j-1
      done
      A[j+1] = value
   done
```

PSEUDO CODE

Pseudocode of the complete algorithm follows, where the arrays are zero-based:

```
for i ← 1 to i ← length(A)-1
  {
 //The values in A[ i ] are checked in-order, starting at the second one
// save A[i] to make a hole that will move as elements are shifted
// the value being checked will be inserted into the hole's final position
valueToInsert ← A[i]
holePos ← i
// keep moving the hole down until the value being checked is larger than
// what's just below the hole <!-- until A[holePos - 1] is <= item -->
whileholePos> 0 and valueToInsert< A[holePos - 1]
{ //value to insert doesn't belong where the hole currently is, so shift
       A[holePos] ← A[holePos - 1] //shift the larger value up
holePos ← holePos - 1      //move the hole position down
     }
   // hole is in the right position, so put value being checked into the hole
   A[holePos] ← valueToInsert }
```

COMPLEXITY ANALYSIS

Best Case **:** o(n)

Worst Case : o(n^2)

Average Case : o(n^2)

ADVANTAGE AND DISADVANTAGE

The insertion sort repeatedly scans the list of items, each time inserting the item in the unordered sequence into its correct position.

The main advantage of the insertion sort is its simplicity. It also exhibits a good performance when dealing with a small list. The insertion sort is an in-place sorting algorithm so the space requirement is minimal. The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms. With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list. Therefore, the insertion sort is particularly useful only when sorting a list of few items.

4.1.8. MERGE SORT

The sorting algorithm Merge sort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$, Mergesort is optimal.

IDEA

The Mergesort algorithm is based on a divide and conquer strategy. First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*) (Figure).

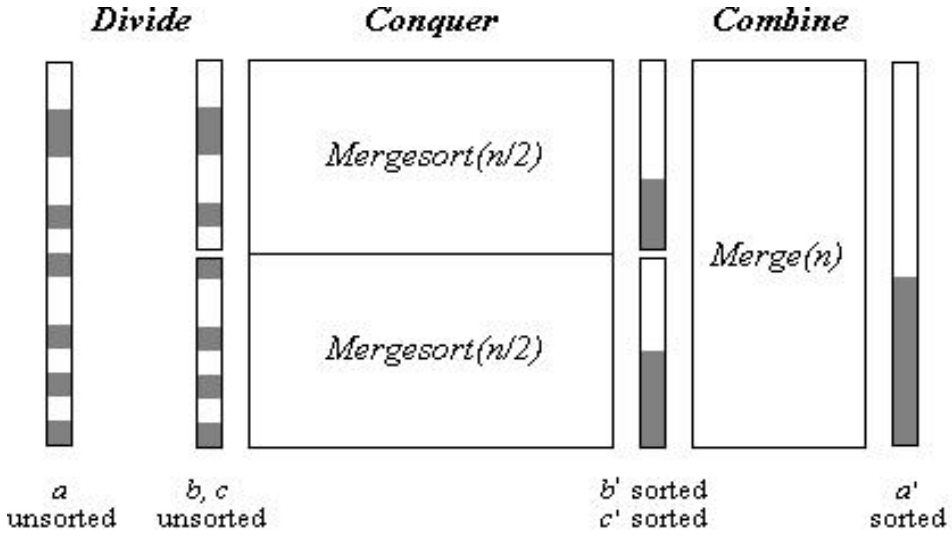


Figure: Mergesort(n)

The following procedure *mergesort* sorts a sequence *a* from index *lo* to index *hi*.

```
void mergesort(int lo, int hi)
{
   if (lo<hi)
   {
int m=(lo+hi)/2;
mergesort(lo, m);
mergesort(m+1, hi);
merge(lo, m, hi);
   }
}
```

First, index *m* in the middle between *lo* and *hi* is determined. Then the first part of the sequence (from *lo* to *m*) and the second part (from *m*+1 to *hi*) are sorted by recursive calls

of *mergesort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when *lo* = *hi*, i.e. when a subsequence consists of only one element.

The main work of the Mergesort algorithm is performed by function *merge*. There are different possibilities to implement this function.

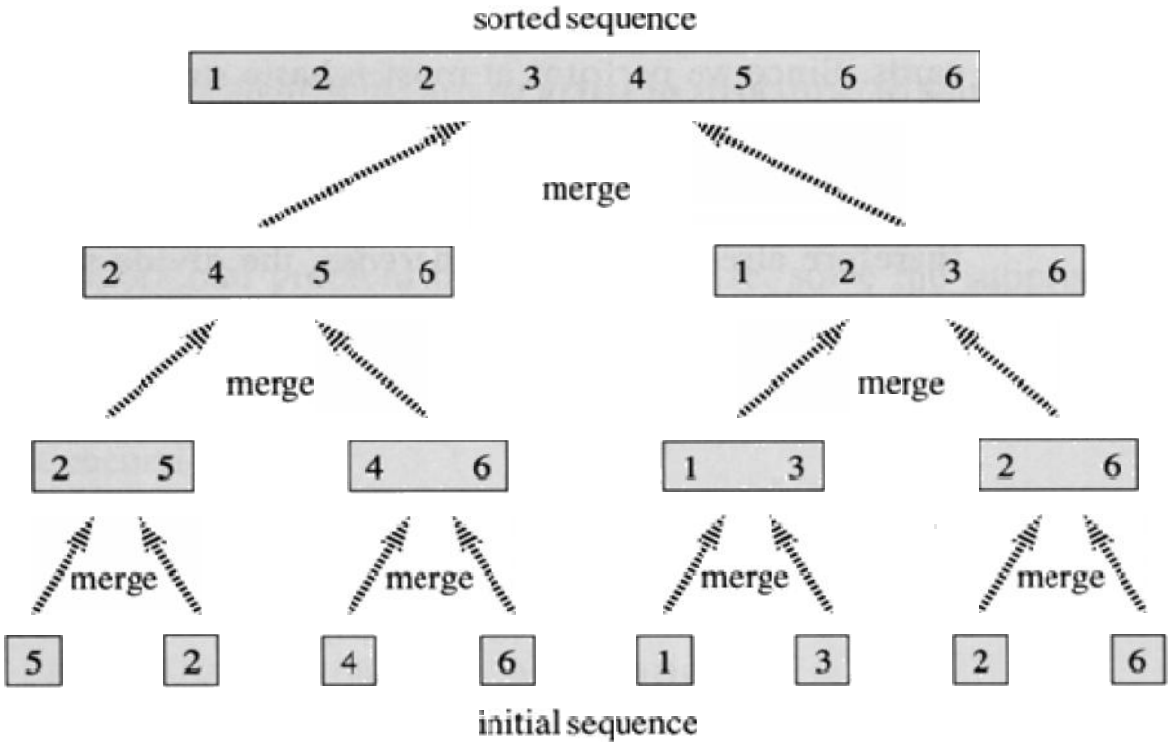
ALGORITHM

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (*A*, 1, *n*).

MERGE-SORT (*A*, *p*, *r*)

1. IF $p<r$ // Check for base case
2. THEN q = FLOOR[$(p + r)/2$] // Divide step
3. MERGE (A, p, q) // Conquer step.
4. MERGE (A, $q + 1$, r) // Conquer step.
5. MERGE (A, p, q, r) // Conquer step.

Example: Bottom-up view of the above procedure for $n = 8$.



Input: Array *A* and indices p, q, r such that $p \le q \le r$ and subarray$A[p .. q]$ is sorted and subarray$A[q + 1 .. r]$ is sorted. By restrictions on p, q, r, neither subarray is empty.
Output: The two subarrays are merged into a single sorted subarray in $A[p .. r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$, which is the number of elements being merged.

PSEUDO CODE

MERGE (*A*, *p*, *q*, *r*)

1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. Create arrays L[1 . . $n_1 + 1$] and R[1 . . $n_2 + 1$]
4. **FOR** $i \leftarrow 1$ **TO** n_1
5. **DO** L[i] ←A[$p + i - 1$]
6. **FOR** $j \leftarrow 1$ **TO** n_2
7. **DO** R[j] ←A[$q + j$]
8. L[$n_1 + 1$] ← ∞
9. R[$n_2 + 1$] ← ∞
10. $i \leftarrow 1$
11. $j \leftarrow 1$
12. **FOR** $k \leftarrow p$ **TO** r
13. **DO IF** L[i] ≤ R[j]
14. **THEN** A[k] ← L[i]
15. $i \leftarrow i + 1$
16. **ELSE** A[k] ← R[j]
17. $j \leftarrow j + 1$

COMPLEXITY ANALYSIS

Best Case **:** O(nlogn)

Worst Case **:** O(nlogn)

Average Case : O(nlogn)

ADVANTAGES

- Merge sort requires only half as much additional space.
- It is faster than the other variants, and it is stable.

DISADVANTAGES

- Merge sort needs an additional space of Θ(n) for the temporary array.

4.1.9. QUICK SORT

IDEA

Quick sort is an efficient sorting algorithm invented by C.A.R. Hoare. Its average-case running time is O(nlogn). Unfortunately, Quicksort's performance degrades as the input list becomes more ordered. The worst-case input, a sorted list, causes it to run in O(n2) time. An improvement upon this algorithm that detects this prevalent corner case and guarantees O(nlogn) time is Introsort.

ALGORITHM

1. Pick a "pivot point". Picking a good pivot point can greatly affect the running time.
2. Break the list into two lists: those elements less than the pivot element, and those elements greater than the pivot element.
3. Recursively sort each of the smaller lists.
4. Make one big list: the 'smallers' list, the pivot points, and the 'biggers' list.

Picking a random pivot point will not eliminate the O(n2) worst-case time, but it will usually transform the worst case into a less frequently occuring permutation. In practice, the sorted list usually comes up more often than any other permutation, so this improvement is often used.

PSEUDO CODE

```
Quicksort(A as array, low as int, high as int)
if (low < high)
pivot_location = Partition(A,low,high)
Quicksort(A,low, pivot_location - 1)
Quicksort(A, pivot_location + 1, high)
Partition(A as array, low as int, high as int)
pivot = A[low]
leftwall = low
for i = low + 1 to high
if (A[i] < pivot) then
leftwall = leftwall + 1
```

swap(A[i], A[leftwall])

swap(A[low],A[leftwall])

return (leftwall)

COMPLEXITY ANALYSIS

Best Case : o(nlogn)

Worst Case : o(n^2)

Average Case : o(nlogn)

ADVANTAGE AND DISADVANTAGE

The quick sort works on the divide-and-conquer principle. First, it partitions the list of items into two sub-lists based on a pivot element. All elements in the first sublist are arranged to be smaller than the pivot, while all elements in the second sublist are arranged to be larger than the pivot. The same partitioning and arranging process is performed repeatedly on the resulting sub-lists until the whole list of items are sorted.

The quick sort is regarded as the best sorting algorithm. This is because of its significant advantage in terms of efficiency because it is able to deal well with a huge list of items. Because it sorts in place, no additional storage is required as well. The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts. In general, the quick sort produces the most effective and widely used method of sorting a list of any item size.

4.1.10. RADIX SORT

IDEA

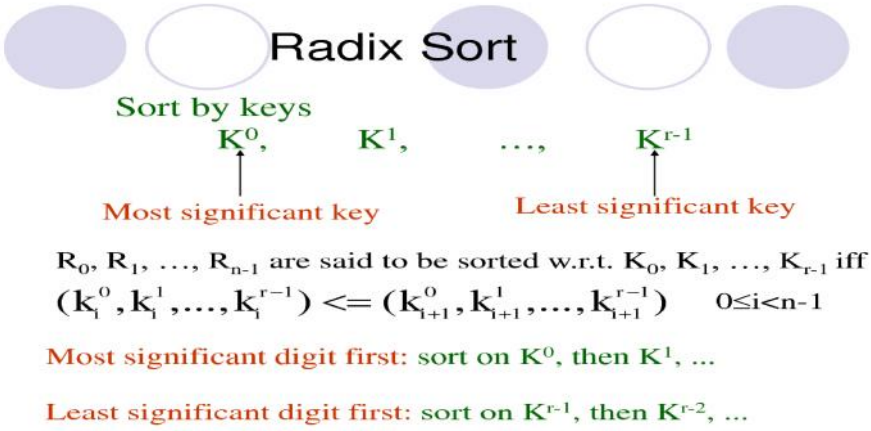
Radix sort is a stable sorting algorithm used mainly for sorting strings of the same length.

Description:

The fundamental principle of radix sort stems from the definition of the stable sort – sorting algorithm is stable, if it maintains the order of keys, which are equal.

Radix sort iteratively orders all the strings by their n-th character – in the first iteration, the strings are ordered by their last character. In the second run, the strings are ordered in respect to their penultimate character. And because the sort is stable, the strings, which

have the same penultimate character, are still sorted in accordance to their last characters. After n-th run the strings are sorted in respect to all character positions.



ALGORITHM

```
Radix (item [], N) /* item is an array of size N */
{   Initialize array Q; /*Q is an array of Circular Queue of size 10;*/
    Find out the maximum element among the elements of the input array;
    Count the digit of the maximum element and store the value into pass;
    Initialize div by 1;
    For (I=1; I<=pass; I++)      {
    /*inserting the elements into queue*/
    For (J=0; J<N; J++) {
        Set remainder is equals to item [J] %( div*10);
        Update remainder by (remainder/div);
        Insert (&Q [remainder], item [J]);
    }   Update div by (div*10);
    /*deleting the elements from the queue and place them into the array*/
For (J=0, index=0; J<10; J++) While (! Isempty (&Q [J])) item [index++] = Delete (&Q [J]);   } }
```

PSEUDO CODE

```
1.functionradixSort(String s)
2.fori in (s.length - 1)  -> 0do
3.stableSort(s[i])
java
/**
   * Radix sort (ascending)
   * Inner stable sort: counting sort
```

```
    * @param array array to be sorted
    * @param dimension fixed length of sorted strings
    * @return sorted array
    */
public static String[] radixSort(String[] array, int dimension){
for(int i = dimension - 1; i >= 0 ; i--){
array = countingSortForRadix(array, i); //order strings by characters at their i-th position
      }
return array;
   }
   /**
    * Counting sort for radix sort
    * @param array array to be sorted
    * @param position position (key) used for the sorting
    * @return sorted array
    */
public static String[] countingSortForRadix(String[] array, int position){
String[] aux = new String[array.length];
char min = array[0].charAt(position);
char max = array[0].charAt(position);
for(int i = 1; i <array.length; i++){
if(array[i].charAt(position) < min) min = array[i].charAt(position);
else if(array[i].charAt(position) > max) max = array[i].charAt(position);
      }
int[] counts = new int[max - min + 1];
for(int i = 0;  i <array.length; i++){
counts[array[i].charAt(position) - min]++;
      }
counts[0]--;
for(int i = 1; i <counts.length; i++){
counts[i] = counts[i] + counts[i-1];
      }
for(int i = array.length - 1; i >=0; i--){
aux[counts[array[i].charAt(position) - min]--] = array[i];
      }
return aux;
   }
```

COMPLEXITY ANALYSIS

Best Case :
Worst Case : O (n)
Average Case :

ADVANTAGE

Radix Sort is stable, meaning it preserves existing order of equal keys. It works in linear time, unlike most other sorts. In other words, it does not bog down when you have large numbers of items to sort. The time per item to sort is constant. With other sorts, the time to sort per time increases with the number of items. Mathematicians would put it that most sorts run in O(n log(n))or $O(n^2)$ time, where Radix Sort runs in O(n) time.

DISADVANTAGE

- Radix Sort does not work well when you have very long keys, because the total sorting time is proportional to key length and to the number of items to sort. If you insisted that all records have unique keys, necessarily the keys would have to be at least $\log_2$ (n) bits long, which would tend to defeat the advantage of Radix Sort. Radix Sort shines when you have large numbers of records to sort with short keys.
- Unfortunately, in this particular implementation, 16-bit characters count as two key slots, unless you can guarantee you never user the high order parts. In other words Unicode takes twice as long to sort as byte arrays
- The biggest drawback with Radix Sort is that you have to write an unconventional compare routine.
- The other drawback is Radix Sort also uses somewhat more working storage than a traditional sort, because it copies references to the elements to sort back and forth between two arrays for each pass. Many other sorts make do with one array and some stack space**.**

4.1.11. SELECTION SORT

IDEA

The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

For example, consider the following array, shown with array elements in sequence separated by commas:

63, 75, 90, 12, 27

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in **bold**. We then swap the element at index 2 with that at index 4. The result is:

63, 75, 27, 12, 90

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in **bold**):

63, 12, 27, 75, 90

The next two steps give us:

27, 12, 63, 75, 90
12, 27, 63, 75, 90

The last effective array has only one element and needs no sorting.

ALGORITHM

```
Selection (item [], N) /* item is an array of size N*/
{
For (I=0; I<N−1; I++){
Initialize MIN by item [I] and p by I;
/*finding minimum element*/
For (J=I+1; J<N; J++)
IF item [J] <MIN then set MIN is equals to item [J] and p is equals to J;
/*swapping the minimum element with the target element*/
Set item[p] by item [I] and item [I] by MIN.}}
```

PSEUDO CODE

```
SelectionSort(A)
// GOAL: place the elements of A in ascending order
1  n := length[A]
2  for i := 1 to n
3     // GOAL: place the correct number in A[i]
4     j := FindIndexOfSmallest( A, i, n )
5     swap A[i] with A[j]
      // L.I. A[1..i] the i smallest numbers sorted
6  end-for
7  end-procedure
FindIndexOfSmallest( A, i, n )
// GOAL: return j in the range [i,n] such
//       that A[j]<=A[k] for all k in range [i,n]
1  smallestAt := i ;
2  for j := (i+1) to n
3     if ( A[j] < A[smallestAt] ) smallestAt := j
      // L.I. A[smallestAt] smallest among A[i..j]
4  end-for
5  return smallestAt
5      end-procedure
```

COMPLEXITY ANALYSIS

Best Case : o(n^2)
Worst Case : o(n^2)
Average Case : o(n^2)

ADVANTAGE AND DISADVANTAGE

The selection sort works by repeatedly going through the list of items, each time selecting an item according to its ordering and placing it in the correct position in the sequence. The main advantage of the selection sort is that it performs well on a small list. Furthermore, because it is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list. The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items. Similar to the bubble sort, the selection sort requires n-squared number of steps for sorting n elements. Additionally, its performance is easily influenced by the initial ordering of the items before the sorting process. Because of this, the selection sort is only suitable for a list of few elements that are in random order.

4.1.12. SHELL SORT

IDEA

We can now slice the array into 10:

5	3
1	2
7	4
8	10
9	6
12	14
11	13
17	16
15	18
19	20

Sorting the columns gives us:

1	2
5	3
7	4
8	6
9	10
11	13
12	14
15	16
17	18
19	20

Reassembling we get:

1	2	5	3	7	4	8	6	9	10	11	13	12	14	15	16	17	18	19	20

The majority of the elements are now near to where they should be, and the last pass of the algorithm is a conventional insertion sort.

ALGORITHM

This sorting algorithm is conceived by D. L. Shell (that's where it gets its name), and is inspired by the Insertion Sort's ability to work very fast on an array that is almost in order. It is also called diminishing increment sort.

Unlike Insertion Sort, Shell Sort does not sort the entire array at once. Instead, it divides the array into noncontiguous segments, which are separately sorted by using Insertion Sort. Once all of the segments are sorted, Shell Sort re-divides the array into less segments and repeat the algorithm until at last that the number of segment equals one, and the segment is sorted. There are two advantages of Shell Sort over Insertion Sort.

When the swap occurs in a noncontiguous segment, the swap moves the item over a greater distance within the overall array. Insertion Sort only moves the item one position at a time. This means that in Shell Sort, the items being swapped are more likely to be closer to its final position then Insertion Sort.

Since the items are more likely to be closer to its final position, the array itself becomes partially sorted. Thus when the segment number equals one, and Shell Sort is performing basically the Insertion Sort, it will be able to work very fast, since Insertion Sort is fast when the array is almost in order.

PSEUDO CODE

```
input: an array a of length n with array elements numbered 0 to n − 1
inc ← round(n/2)
while inc > 0 do:
        for i = inc .. n − 1 do:
                temp ← a[i]
                j ← i
        while j ≥ inc and a[j − inc] > temp do:
                 a[j] ← a[j − inc]
                 j ← j – inc
                a[j] ← temp
                inc ← round(inc / 2.2)
```

COMPLEXITY ANALYSIS

Best Case **:** o(n)

Worst Case : o(n(logn)^2)

Average Case : o(n^2/3)

ADVANTAGE

- Advantage of shell sort is that it's only efficient for medium size lists. For bigger lists, the algorithm is not the best choice fastest of all o(n^2) sorting algorithms.
- Five times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

DISADVANTAGE

- Disadvantage of shell sort is that it is a complex algorithm and it's not nearly as efficient as the merge sort, heap sort & quick sort.
- The shell sort is still significantly slower than the merge sort, heap sort & quick sort, but its relatively simple algorithm makes it a good choice for sorting lists of less than five thousand items unless speed important. It's also an excellent choice for repetitive sorting of smaller lists.

4.2. CLASSIFICATION

Sorting algorithms used in computer science are often classified by:

- **Computational complexity** (worst, average and best behavior) **of element comparisons in terms of the size** of the list (*n*). For typical sorting algorithms good behavior is O(*n* log *n*) and bad behavior is O(n^2). (See Big O notation.) Ideal behavior for a sort is O(*n*), but this is not possible in the average case. Comparison-based sorting algorithms, which evaluate the elements of the list via an abstract key comparison operation, need at least O(*n* log *n*) comparisons for most inputs.
- **Computational complexity of swaps** (for "in place" algorithms).
- **Memory usag**e (and use of other computer resources). In particular, some sorting algorithms are "in place". Strictly, an in place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(*n*)) additional memory is considered "in place".
- **Recursion.** Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- **Stability:** Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- **General Method:** insertion, exchange, selection, merging, *etc.*. Exchange sorts include bubble sort and quick sort. Selection sorts include shaker sort and heap sort.
- **Adaptability:** Whether or not the pre sortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

4.3. STABILITY

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first component:

(4, 2) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

(3, 7) (3, 1) (4, 2) (5, 6) (Order maintained)
(3, 1) (3, 7) (4, 2) (5, 6) (Order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional computational cost.

Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple times, each time with one sort key. In that case the keys need to be applied in order of increasing priority.

Example: sorting pairs of numbers as above by second, then first component:

(4, 2) (3, 7) (3, 1) (5, 6) (Original)

(3, 1) (4, 2) (5, 6) (3, 7) (After sorting by second component)

(3, 1) (3, 7) (4, 2) (5, 6) (After sorting by first component)

On the other hand:

(3, 7) (3, 1) (4, 2) (5, 6) (After sorting by first component)

(3, 1) (4, 2) (5, 6) (3, 7) (After sorting by second component, order by first component is disrupted).

4.4. COMPARISONS

In this table, ***n*** is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. These are all comparison sorts. The run time and the memory of algorithms could be measured using various notations like theta, omega, Big-O, small-o, etc. The memory and the run times below are applicable for all the 5 notations.

Comparison sorts							
Name	**Best**	**Average**	**Worst**	**Memory**	**Stable**	**Method**	**Other notes**
Bogosort	n	$n \cdot n!$	$n \cdot n! \rightarrow \infty$	1	No	Luck	Randomly permute the array and check if sorted.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	
Comb sort	n	$n \log n$	n^2	1	No	Exchanging	Small code size
Cycle sort	—	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O(n + d), where d is the number of inversions
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in SGISTL implementation
Merge sort	$n \log n$	$n \log n$	$n \log n$	**Depends; worst case is n**	Yes	Merging	Used to sort this table in Firefox.

Comparison sorts							
Name	**Best**	**Average**	**Worst**	**Memory**	**Stable**	**Method**	**Other notes**
Quicksort	\\$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning	Quicksort is usually done in place with O(log(*n*)) stack space. Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an O(*n*) space array to store the partition.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists. Used to sort this table in Safari or other Webkit web browser.
Shell sort	n	$n(\log n$ **or** $n^{3/2}$	**Depends on gap sequence; best known is** $n(\log n)^2$	1	No	Insertion	

The following table describes integer sorting algorithms and other sorting algorithms that are not comparison sorts. As such, they are not limited by a Ω(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 *d*, the digit 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."

Non-comparison sorts							
Name	**Best**	**Average**	**Worst**	**Memory**	**Stable**	$n << 2^k$	**Notes**
Bucket sort (integer keys)	—	$n+r$	$n+r$	$n+r$	Yes	Yes	r is the range of numbers to be sorted. If r = $\mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$
Counting sort	—	$n+r$	$n+r$	$n+r$	Yes	Yes	r is the range of numbers to be sorted. If r = $\mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$

The following table describes some sorting algorithms that are impractical for real-life use due to extremely poor performance or a requirement for specialized hardware.

Name	**Best**	**Average**	**Worst**	**Memory**	**Stable**	**Comparison**	**Other notes**
Bead sort	—	N/A	N/A	—	N/A	No	Requires specialized hardware