**COMPARATIVE STUDY ON DIFFERENT SORTING ALGORITHMS**

**Dissertation submitted in fulfilment of the requirements for the degree of B.Tech.In Information Technology of**

**JIS College of Engineering**

**By**

**Susmita Nama**

**Univ.Roll No:09123002084**

**Kazi Habib Hasan**

**Univ.Roll No:09123002083**

**Jayeeta Das**

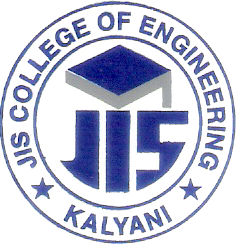
**Univ.Roll No:09123002081**

Under The Guidence of

Prof.Somsubhra Gupta

Department.of Information Tehnology (HOD)

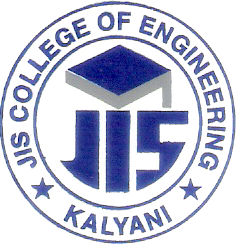
**JIS College of Engineering,**

****

JIS College of Engineering

Block-a,phase-III,kalyani,Nadia,Pin-741235

West Bengal,India

****

**Certificate**

This is to certify that Ankur Chakrabarti (Roll No-08123002032 & Registration No- 081230110126), Susmita Chakraborty (Roll No-08123002015 & Registration No- 081230110202), Sabnam Raha (Roll No-08123002043 & Registration No- 081230110180), are students of B.Tech. in Information Technology of 4th Year 8th Semester of JIS College of Engineering, Session 2008-2012, submitted the major thesis work in partial fulfillment of the requirements of the Degree of Bachelor of Technology in Information Technology, is a result of the bonfire work carried out on the dissertation titled *“Face Recognition sing heuristic training approach under the framework of Digital Image Processing”* during the academic session 2011-2012,under the guidance and supervision of mine. I hereby recommend that this thesis be accepted in partial fulfillment of the requirements for the Degree of Bachelor of Technology in Information Technology, of JIS College of Engineering. During the thesis work they were found to be sincere, regular and hard working and have successfully completed the thesis work assigned to them.

**Mr. Somsubhra Gupta**

(Project Supervisor)

HOD, Dept of Information Technology

**JIS College of Engineering**

ACKNOWLEDGEMENT

The analysis of the project work wishes to express my gratitude to Mr. Somsubhra Gupta for allowing the degree attitude and providing effective guidance in development of this project work. His conscription of the topic and all the helpful hints, he provided, contributed greatly to successful development of this work, without being pedagogic and overbearing influence.

I also express my sincere gratitude to Mr. Somsubhra Gupta, Head of the Department of Information Technology of JIS College of Engineering and all the faculties’ members of IT department for giving the scope of successfully carrying out the project work.

Finally, I take this opportunity to thank to Dr. U.B. Bandhopadhay, Principal of JIS College of Engineering and Dr.A.Guha, Director of JIS College of Engineering for giving us the scope of carrying out the project work.

Date:

**..…………………….………………………………….**

Ankur Chakrabarti (IT/**4TH** YEAR/**8TH** SEMESTER)

**..…………………….………………………………….**

Susmita Chakraborty (IT/**4TH** YEAR/**8TH** SEMESTER)

**..…………………….………………………………….**

Sabnam Raha (IT/**4TH** YEAR/**8TH** SEMESTER)

**CONTENTS**

**Preface………………………………………………………………. 5**

**1. Classification……………………………………………………….6**

**2. Stability………………………………….…………………………7**

**3. Comparative Study Of The Algorithms………………………….8**

**4. Conclusion………………………………………………………..79**

**5. Reference ………………………………………….80**

**Preface**

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);

2. The output is a permutation (reordering) of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.[1] Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2006). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

**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](http://en.wikipedia.org/wiki/Big_O_notation).) Ideal behavior for a sort is O(*n*), but this is not possible in the average case. [Comparison-based sorting algorithms](http://en.wikipedia.org/wiki/Comparison_sort), 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](http://en.wikipedia.org/wiki/Computational_complexity_theory) of swaps (for "in place" algorithms).
* Memory usage (and use of other computer resources). In particular, some sorting algorithms are "[in place](http://en.wikipedia.org/wiki/In-place_algorithm)". 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](http://en.wikipedia.org/wiki/Sorting_algorithm#Stability) maintain the relative order of records with equal keys (i.e., values).
* Whether or not they are a [comparison sort](http://en.wikipedia.org/wiki/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 quicksort. Selection sorts include shaker sort and heapsort.
* Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be [adaptive](http://en.wikipedia.org/wiki/Adaptive_sort).

### 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](http://en.wikipedia.org/wiki/Computational_complexity_theory).

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).

**Comparison of algorithms**

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](http://en.wikipedia.org/wiki/Comparison_sort). 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](http://en.wikipedia.org/wiki/Comparison_sort) | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Best** | **Average** | **Worst** | **Memory** | **Stable** | **Method** | **Other notes** |
| [Binary tree sort](http://en.wikipedia.org/wiki/Binary_tree_sort) | \mathcal{} n | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} n | Yes | Insertion | When using a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) |
| [Bogosort](http://en.wikipedia.org/wiki/Bogosort) | \mathcal{} n | \mathcal{} n \cdot n! | \mathcal{} {n \cdot n! \to \infty} | \mathcal{} {1} | No | Luck | Randomly permute the array and check if sorted. |
| [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging |  |
| [Comb sort](http://en.wikipedia.org/wiki/Comb_sort) | \mathcal{} n | \mathcal{} n \log n | \mathcal{} n^2 | \mathcal{} {1} | No | Exchanging | Small code size |
| [Cycle sort](http://en.wikipedia.org/wiki/Cycle_sort) | — | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | No | Insertion | In-place with theoretically optimal number of writes |
| [Gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {1} | No | Selection |  |
| [In-place](http://en.wikipedia.org/wiki/In-place) [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | \mathcal{} - | \mathcal{} - | \mathcal{} {n \left( \log n \right)^2} | \mathcal{} {1} | Yes | Merging | Implemented in Standard Template Library (STL): [[3]](http://www.sgi.com/tech/stl/stable_sort.html); can be implemented as a stable sort based on stable in-place merging: [[4]](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8381) |
| [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Insertion | O(*n* + *d*), where *d* is the number of [inversions](http://en.wikipedia.org/wiki/Permutation_groups#Transpositions.2C_simple_transpositions.2C_inversions_and_sorting) |
| [Introsort](http://en.wikipedia.org/wiki/Introsort) | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} \log n | No | Partitioning & Selection | Used in [SGI](http://en.wikipedia.org/wiki/Silicon_Graphics) [STL](http://en.wikipedia.org/wiki/Standard_Template_Library) implementations |
| [Library sort](http://en.wikipedia.org/wiki/Library_sort) | — | \mathcal{} {n \log n} | \mathcal{} n^2 | \mathcal{} n | Yes | Insertion |  |
| [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | Depends; worst case is  \mathcal{} n | Yes | Merging | Used to sort this table in Firefox [[2]](http://mxr.mozilla.org/seamonkey/source/js/src/jsarray.c). |
| [Patience sorting](http://en.wikipedia.org/wiki/Patience_sorting) | — | — | \mathcal{} n \log n | \mathcal{} n | No | Insertion & Selection | Finds all the [longest increasing subsequences](http://en.wikipedia.org/wiki/Longest_increasing_subsequence) within O(*n* log *n*) |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n^2 | \mathcal{} \log n | Depends | Partitioning | Quicksort is usually done in place with O(log(*n*)) stack space.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] Most implementations are unstable, as stable in-place partitioning is more complex. [Naïve](http://en.wikipedia.org/wiki/Na%C3%AFve_algorithm) variants use an O(*n*) space array to store the partition.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] |
| [Selection sort](http://en.wikipedia.org/wiki/Selection_sort) | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | No | Selection | Stable with O(n) extra space, for example using lists [[5]](http://www.algolist.net/Algorithms/Sorting/Selection_sort). Used to sort this table in Safari or other Webkit web browser [[6]](http://svn.webkit.org/repository/webkit/trunk/Source/JavaScriptCore/runtime/ArrayPrototype.cpp). |
| [Shell sort](http://en.wikipedia.org/wiki/Shell_sort) | \mathcal{} n | \mathcal{} n (\log n)^2  or  \mathcal{} n^{3/2} | Depends on gap sequence; best known is \mathcal{} n (\log n)^2 | \mathcal{} 1 | No | Insertion |  |
| [Smoothsort](http://en.wikipedia.org/wiki/Smoothsort) | \mathcal{} {n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {1} | No | Selection | An [adaptive sort](http://en.wikipedia.org/wiki/Adaptive_sort) - \mathcal{} {n} comparisons when the data is already sorted, and 0 swaps. |
| [Strand sort](http://en.wikipedia.org/wiki/Strand_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} n | Yes | Selection |  |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | \mathcal{} {n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} n | Yes | Insertion & Merging | \mathcal{} {n} comparisons when the data is already sorted or reverse sorted. |
| [Tournament sort](http://en.wikipedia.org/wiki/Tournament_sort) | — | \mathcal{} n \log n | \mathcal{} n \log n |  |  | Selection |  |

The following table describes [integer sorting](http://en.wikipedia.org/wiki/Integer_sorting) algorithms and other sorting algorithms that are not [comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort). As such, they are not limited by a \Omega\left( {n \log n} \right)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** |
| [Pigeonhole sort](http://en.wikipedia.org/wiki/Pigeonhole_sort) | — | \;n + 2^k | \;n + 2^k | \;2^k | Yes | Yes |  |
| [Bucket sort](http://en.wikipedia.org/wiki/Bucket_sort) (uniform keys) | — | \;n+k | \;n^2 \cdot k | \;n \cdot k | Yes | No | Assumes uniform distribution of elements from the domain in the array.[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-clrs-1) |
| [Bucket sort](http://en.wikipedia.org/wiki/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}\left( {n} \right)then Avg RT = \mathcal{O}\left( {n} \right)[[3]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-gt-2) |
| [Counting sort](http://en.wikipedia.org/wiki/Counting_sort) | — | \;n+r | \;n+r | \;n+r | Yes | Yes | r is the range of numbers to be sorted. If r = \mathcal{O}\left( {n} \right)then Avg RT = \mathcal{O}\left( {n} \right)[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-clrs-1) |
| [LSD Radix Sort](http://en.wikipedia.org/wiki/Radix_sort#Least_significant_digit_radix_sorts) | — | \;n \cdot \frac{k}{d} | \;n \cdot \frac{k}{d} | \mathcal{} n | Yes | No | [[3]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-gt-2)[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-clrs-1) |
| [MSD Radix Sort](http://en.wikipedia.org/wiki/Radix_sort#Most_significant_digit_radix_sorts) | — | \;n \cdot \frac{k}{d} | \;n \cdot \frac{k}{d} | \mathcal{} n + \frac{k}{d} \cdot 2^d | Yes | No | Stable version uses an external array of size n to hold all of the bins |
| [MSD Radix Sort](http://en.wikipedia.org/wiki/Radix_sort#Most_significant_digit_radix_sorts) | — | \;n \cdot \frac{k}{d} | \;n \cdot \frac{k}{d} | \frac{k}{d} \cdot 2^d | No | No | In-Place. k / d recursion levels, 2d for count array |
| [Spreadsort](http://en.wikipedia.org/wiki/Spreadsort) | — | \;n \cdot \frac{k}{d} | \;n \cdot \left( {\frac{k}{s} + d} \right) | \;\frac{k}{d} \cdot 2^d | No | No | Asymptotics are based on the assumption that n << 2k, but the algorithm does not require this. |

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](http://en.wikipedia.org/wiki/Bead_sort) | — | N/A | N/A | — | N/A | No | Requires specialized hardware |
| [Simple pancake sort](http://en.wikipedia.org/wiki/Pancake_sorting) | — | \mathcal{} n | \mathcal{} n | \mathcal{} {\log n} | No | Yes | Count is number of flips. |
| [Spaghetti (Poll) sort](http://en.wikipedia.org/wiki/Spaghetti_sort) | \mathcal{} n | \mathcal{} n | \mathcal{} n | \mathcal{} n^2 | Yes | Polling | This A linear-time, analog algorithm for sorting a sequence of items, requiring O(*n*) stack space, and the sort is stable. This requires nparallel processors. [Spaghetti sort#Analysis](http://en.wikipedia.org/wiki/Spaghetti_sort#Analysis) |
| [Sorting networks](http://en.wikipedia.org/wiki/Sorting_network) | — | \mathcal{} {\log n} | \mathcal{} {\log n} | \mathcal{} {n \cdot \log (n)} | Yes | No | Requires a custom circuit of size \mathcal{O}\left( n \cdot \log (n) \right) |

Additionally, theoretical computer scientists have detailed other sorting algorithms that provide better than \mathcal{O}\left( {n \log n} \right)time complexity with additional constraints, including:

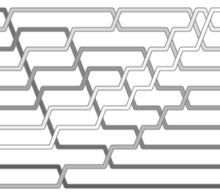
* Han's algorithm, a deterministic algorithm for sorting keys from a [domain](http://en.wikipedia.org/wiki/Domain_of_a_function) of finite size, taking \mathcal{O}\left( {n \log \log n} \right)time and \mathcal{O}\left( {n} \right)space.[[4]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-3)
* Thorup's algorithm, a randomized algorithm for sorting keys from a domain of finite size, taking \mathcal{O}\left( {n \log \log n} \right)time and \mathcal{O}\left( {n} \right)space.[[5]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-4)
* An [integer](http://en.wikipedia.org/wiki/Integer) sorting algorithm taking \mathcal{O}\left( {n \sqrt{\log \log n}} \right)expected time and \mathcal{O}\left( {n} \right)space.[[6]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-5)

Algorithms not yet compared above include:

* [Odd-even sort](http://en.wikipedia.org/wiki/Odd-even_sort)
* [Flashsort](http://en.wikipedia.org/wiki/Flashsort)
* [Burstsort](http://en.wikipedia.org/wiki/Burstsort)
* [Postman sort](http://en.wikipedia.org/wiki/Postman_sort)
* [Stooge sort](http://en.wikipedia.org/wiki/Stooge_sort)
* [Samplesort](http://en.wikipedia.org/wiki/Samplesort)
* [Bitonic sorter](http://en.wikipedia.org/wiki/Bitonic_sorter)

**Summaries of popular sorting algorithms**

**1. Bubble sort**

[](http://en.wikipedia.org/wiki/File:Bubblesort-edited.png)

[http://bits.wikimedia.org/skins-1.20wmf1/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Bubblesort-edited.png)

A bubble sort, a sorting algorithm that continuously steps through a list, [swapping](http://en.wikipedia.org/wiki/Swap_%28computer_science%29) items until they appear in the correct order.

Main article: [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort)

*Bubble sort* is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average and worst case performance is O(*n*2), so it is rarely used to sort large, unordered, data sets. Bubble sort can be used to sort a small number of items (where its inefficiency is not a high penalty). Bubble sort may also be efficiently used on a list that is already sorted except for a very small number of elements. For example, if only one element is not in order, bubble sort will take only *2n* time. If two elements are not in order, bubble sort will take only at most *3n* time...

Algorithm in c:

#include "stdio.h"

#include "conio.h"

void main( )

{

int arr[5] = { 25, 17, 31, 13, 2 } ;

int i, j, temp ;

clrscr();

printf ( "\n\nArray before sorting:\n") ;

for ( i = 0 ; i <= 4 ; i++ )

printf ( "%d\t", arr[i] ) ;

for ( i = 0 ; i <= 3 ; i++ )

{

for ( j = 0 ; j <= 3 - i ; j++ )

{

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

{

temp = arr[j] ;

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

arr[j + 1] = temp ;

}

}

}

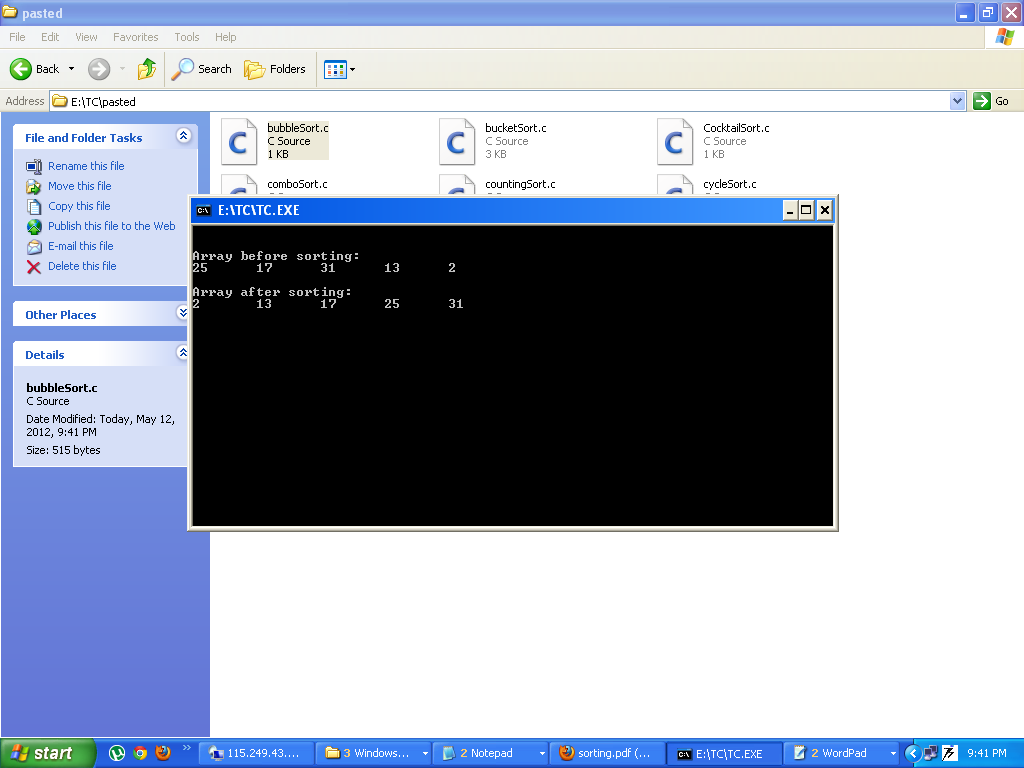
printf ( "\n\nArray after sorting:\n") ;

for ( i = 0 ; i <= 4 ; i++ )

printf ( "%d\t", arr[i] ) ;

getch();

}



**2. Selection sort**

Main article: [Selection sort](http://en.wikipedia.org/wiki/Selection_sort)

*Selection sort* is an [in-place](http://en.wikipedia.org/wiki/In-place_algorithm)[comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). It has [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n*2) complexity, making it inefficient on large lists, and generally performs worse than the similar [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than *n* swaps, and thus is useful where swapping is very expensive.

Algoritham in c:

#include "stdio.h"

#include<conio.h>

void main( )

{

int arr[5] = { 25, 17, 31, 13, 2 } ;

int i, j, temp ;

clrscr();

printf ( "\n\nArray before sorting:\n") ;

for ( i = 0 ; i <= 4 ; i++ )

printf ( "%d\t", arr[i] ) ;

for ( i = 0 ; i <= 3 ; i++ )

{

for ( j = i + 1 ; j <= 4 ; j++ )

{

if ( arr[i] > arr[j] )

{

temp = arr[i] ;

arr[i] = arr[j] ;

arr[j] = temp ;

}

}

}

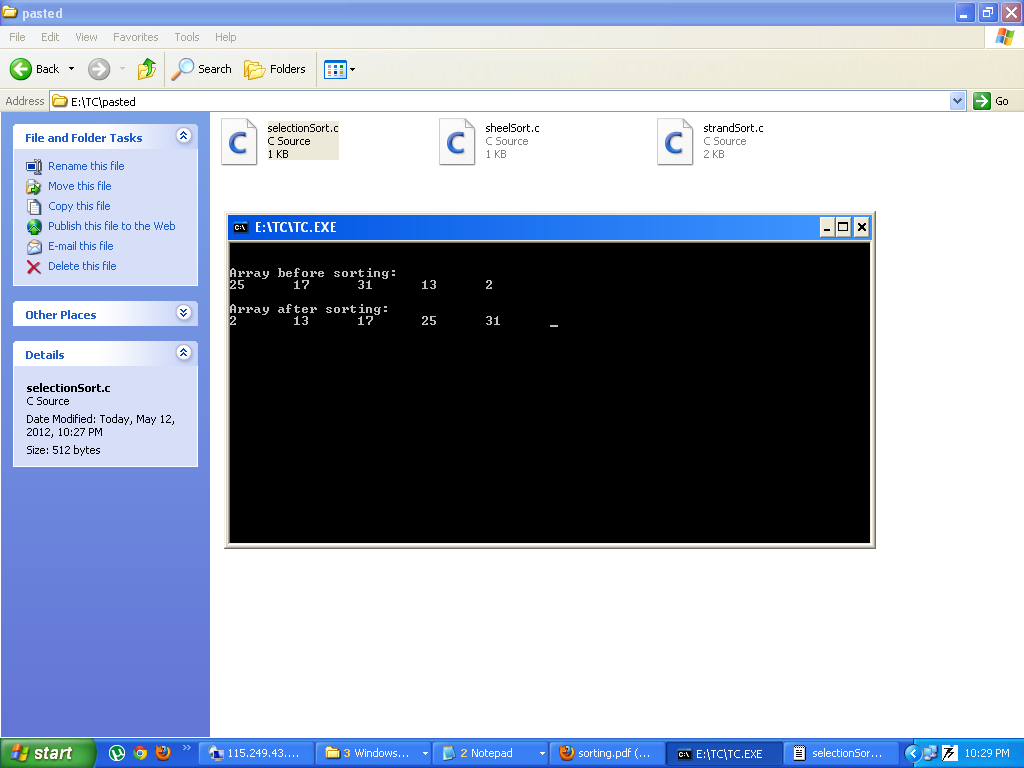
printf ( "\n\nArray after sorting:\n") ;

for ( i = 0 ; i <= 4 ; i++ )

printf ( "%d\t", arr[i] ) ;

getch();

}



**3. Insertion sort**

Main article: [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort)

*Insertion sort* is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. [Shell sort](http://en.wikipedia.org/wiki/Shell_sort) (see below) is a variant of insertion sort that is more efficient for larger lists.

Algoritam in C:

#include "stdio.h"

#include<conio.h>

void main( )

{

intarr[5] = { 25, 17, 31, 13, 2 } ;

int i, j, k, temp ;

clrscr();

printf ( "Insertion sort.\n" ) ;

printf ( "\nArray before sorting:\n") ;

for ( i = 0 ; i <= 4 ; i++ )

printf ( "%d\t", arr[i] ) ;

for ( i = 1 ; i <= 4 ; i++ )

{

for ( j = 0 ; j < i ; j++ )

{

if ( arr[j] >arr[i] )

{

temp = arr[j] ;

arr[j] = arr[i] ;

for ( k = i ; k > j ; k-- )

arr[k] = arr[k - 1] ;

arr[k + 1] = temp ;

}

}

}

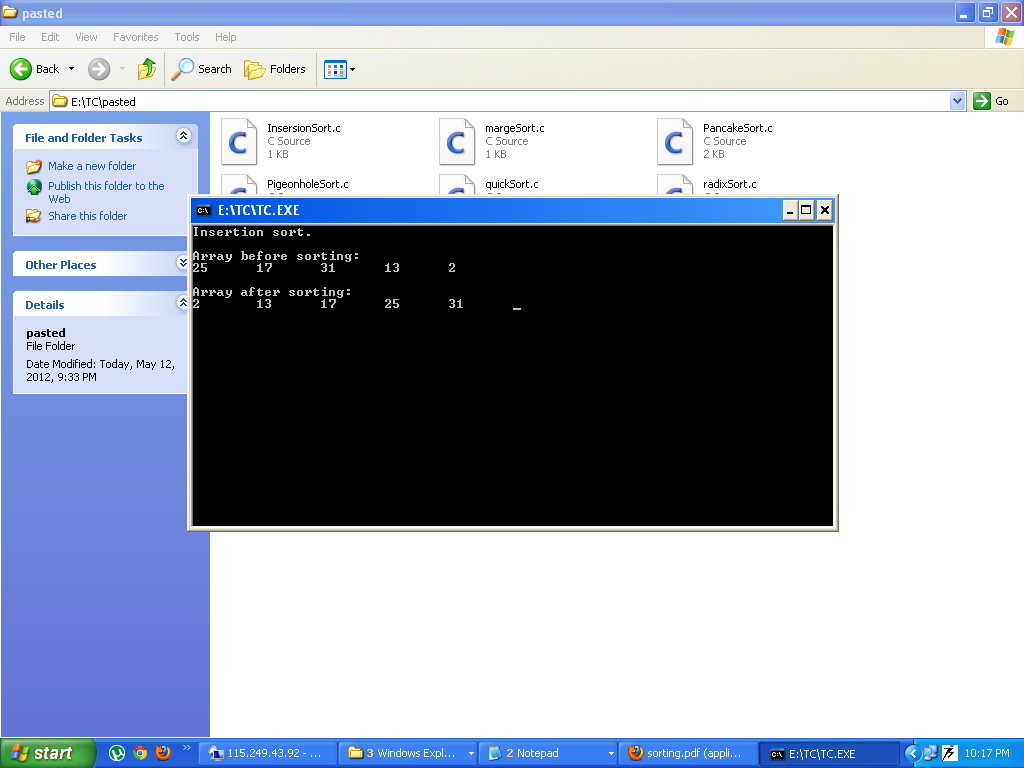
printf ( "\n\nArray after sorting:\n") ;

for ( i = 0 ; i <= 4 ; i++ )

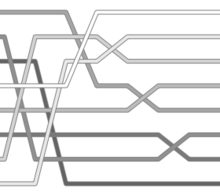
printf ( "%d\t", arr[i] ) ;

getch();

}



**4. Shell sort**

[](http://en.wikipedia.org/wiki/File:Shellsort-edited.png)

[http://bits.wikimedia.org/skins-1.20wmf1/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Shellsort-edited.png)

A Shell sort, different from bubble sort in that it moves elements numerous positions [swapping](http://en.wikipedia.org/wiki/Swap_%28computer_science%29)

Main article: [Shell sort](http://en.wikipedia.org/wiki/Shell_sort)

*Shell sort* was invented by [Donald Shell](http://en.wikipedia.org/wiki/Donald_Shell) in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Algorithm in C:

#include<stdio.h>

#include<conio.h>

voidshellsort(int a[],int n)

{

intj,i,k,m,mid;

for(m = n/2;m>0;m/=2)

{

for(j = m;j<n;j++)

{

for(i=j-m;i>=0;i-=m)

{

if(a[i+m]>=a[i])

break;

else

{

mid = a[i];

a[i] = a[i+m];

a[i+m] = mid;

}

}

}

}

}

main()

{

int a[10],i,n;

clrscr();

printf("Enter The number Of Elements\t: ");

scanf("%d",&n);

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

{

printf("\nElement %d\t: ",i+1);

scanf("%d",&a[i]);

}

printf("\nArrayBefor Sorting : ");

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

printf("%5d",a[i]);

shellsort(a,n);

printf("\nArray After Sorting : ");

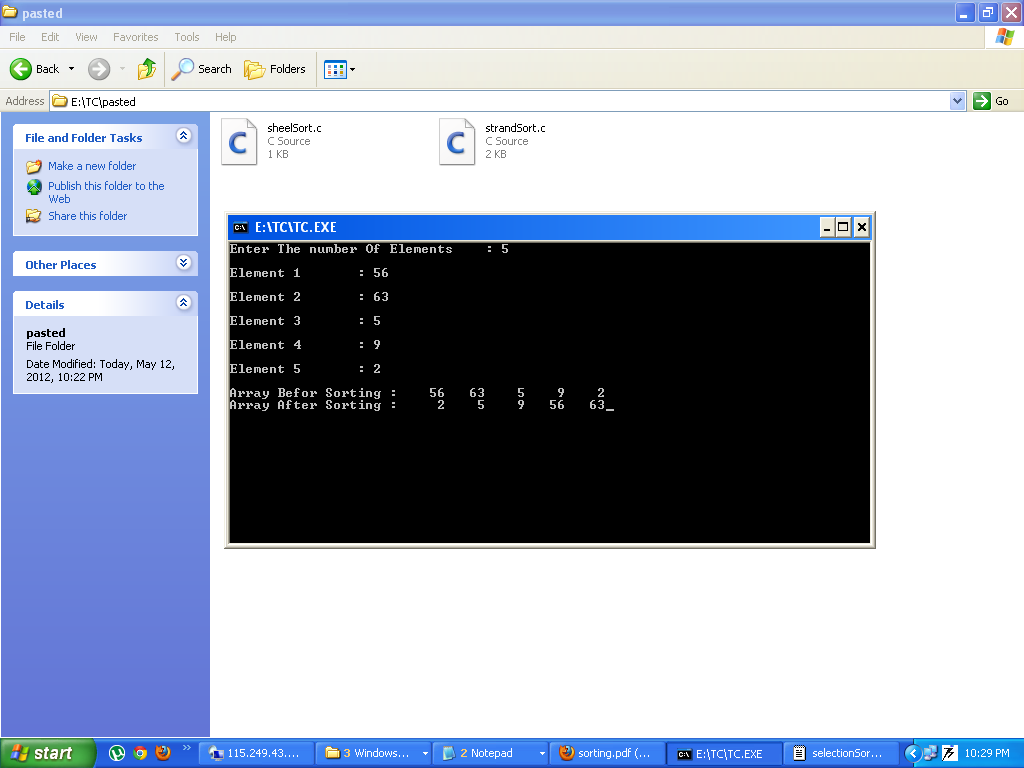
for(i=0;i<n;i++)

printf("%5d",a[i]);

getch();

return 0;

}



**5. Comb sort**

Main article: [Comb sort](http://en.wikipedia.org/wiki/Comb_sort)

*Comb sort* is a relatively simplistic sorting algorithm originally designed by [WlodzimierzDobosiewicz](http://en.wikipedia.org/w/index.php?title=Wlodzimierz_Dobosiewicz&action=edit&redlink=1) in 1980. Later it was rediscovered and popularized by [Stephen Lacey](http://en.wikipedia.org/w/index.php?title=Stephen_Lacey&action=edit&redlink=1) and [Richard Box](http://en.wikipedia.org/w/index.php?title=Richard_Box&action=edit&redlink=1) with a [Byte Magazine](http://en.wikipedia.org/wiki/Byte_Magazine) article published in April 1991. Comb sort improves on [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), and rivals algorithms like [Quicksort](http://en.wikipedia.org/wiki/Quicksort). The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (*Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort)

Algorithm in C:

#include<stdio.h>

#include<conio.h>

void Combsort11(double a[], intnElements)

{

int i, j, gap, swapped = 1;

double temp;

gap = nElements;

while (gap > 1 || swapped == 1)

{

gap = gap \* 10 / 13;

if (gap == 9 || gap == 10) gap = 11;

if (gap < 1) gap = 1;

swapped = 0;

for (i = 0, j = gap; j <nElements; i++, j++)

{

if (a[i] > a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

swapped = 1;

}

}

}

}

void main()

{

double a[5]={10,5,4,2,100};

int i=0;

clrscr();

Combsort11(a,5);

for(i=0;i<5;i++)

{

printf("%d\n",a[i]);

}

getch();

}

**6. Merge sort**

Main article: [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) [Heapsort](http://en.wikipedia.org/wiki/Heapsort)

*Merge sort* takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(*n* log *n*). Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages [Perl](http://en.wikipedia.org/wiki/Perl),[[7]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-6)[Python](http://en.wikipedia.org/wiki/Python_%28programming_language%29) (as [timsort](http://en.wikipedia.org/wiki/Timsort)[[8]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-7)), and [Java](http://en.wikipedia.org/wiki/Java_%28programming_language%29) (also uses timsort as of [JDK7](http://en.wikipedia.org/wiki/JDK7)[[9]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-8)), among others. Merge sort has been used in Java at least since 2000 in JDK1.3.[[10]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-mergesort_in_jdk13-9)[[11]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-jdk13_since_2000-10)

**Algorithm In c:**

#include<stdio.h>

#include <conio.h>

voidgetdata(intarr[],int n)

{

int i;

printf("enter the data:");

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

{

scanf("%d",&arr[i]);

}

}

void display(intarr[],int n)

{

int i;

printf("");

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

{

printf("%d ",arr[i]);

}

getchar();

}

void sort(intarr[],intlow,intmid,int high)

{

inti,j,k,l,b[20];

l=low;

i=low;

j=mid+1;

while((l<=mid)&&(j<=high))

{

if(arr[l]<=arr[j])

{

b[i]=arr[l];

l++;

}

else

{

b[i]=arr[j];

j++;

}

i++;

}

if(l>mid)

{

for(k=j;k<=high;k++)

{

b[i]=arr[k];

i++;

}

}

else

{

for(k=l;k<=mid;k++)

{

b[i]=arr[k];

i++;

}

}

for(k=low;k<=high;k++)

{

arr[k]=b[k];

}

}

void partition(intarr[],intlow,int high)

{

int mid;

if(low<high)

{

mid=(low+high)/2;

partition(arr,low,mid);

partition(arr,mid+1,high);

sort(arr,low,mid,high);

}

}

void main()

{

intarr[20];

int n;

clrscr();

printf("Enter number of data:");

scanf("%d",&n);

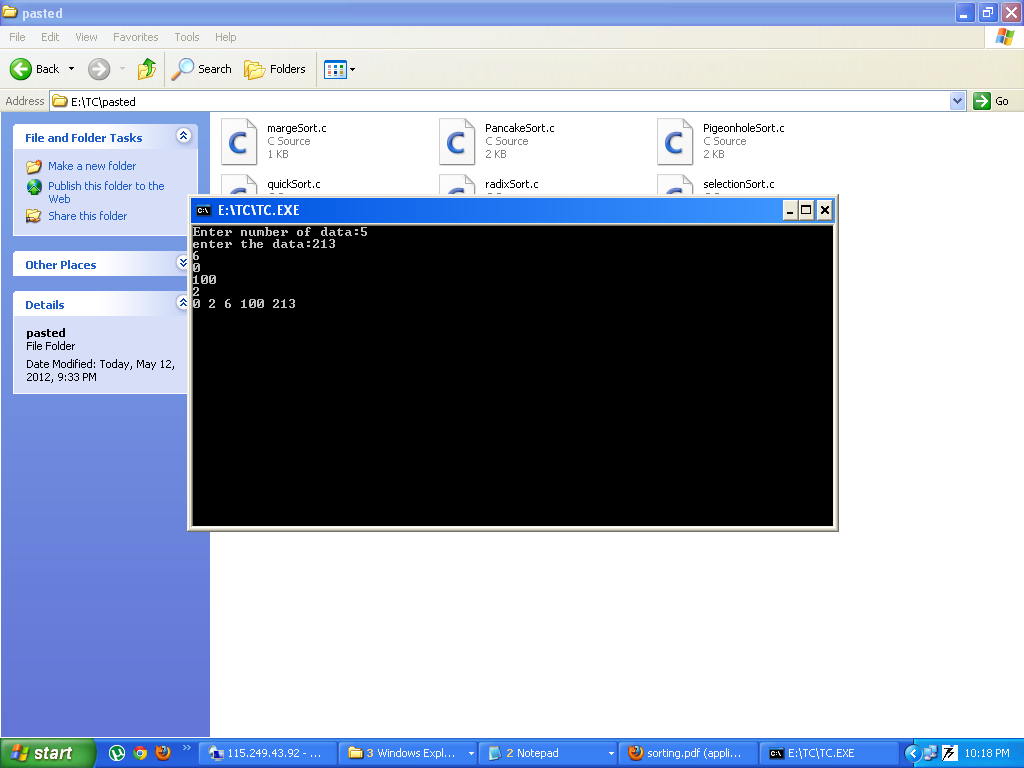
getdata(arr,n);

partition(arr,0,n-1);

display(arr,n);

getchar();

}



**7. Heapsort**

Main article:

*Heapsort* is a much more efficient version of [selection sort](http://en.wikipedia.org/wiki/Selection_sort). It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a [heap](http://en.wikipedia.org/wiki/Heap_%28data_structure%29), a special type of [binary tree](http://en.wikipedia.org/wiki/Binary_tree). Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes *O(*log *n)* time, instead of *O(n)* for a linear scan as in simple selection sort. This allows Heapsort to run in *O(n* log *n)* time, and this is also the worst case complexity.

Algorithm in C:

#include<stdio.h>

#include <conio.h>

voidgetdata(intarr[],int n)

{

int i;

printf("enter the data:");

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

{

scanf("%d",&arr[i]);

}

}

void display(intarr[],int n)

{

int i;

printf("");

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

{

printf("%d ",arr[i]);

}

getchar();

}

void sort(intarr[],intlow,intmid,int high)

{

inti,j,k,l,b[20];

l=low;

i=low;

j=mid+1;

while((l<=mid)&&(j<=high))

{

if(arr[l]<=arr[j])

{

b[i]=arr[l];

l++;

}

else

{

b[i]=arr[j];

j++;

}

i++;

}

if(l>mid)

{

for(k=j;k<=high;k++)

{

b[i]=arr[k];

i++;

}

}

else

{

for(k=l;k<=mid;k++)

{

b[i]=arr[k];

i++;

}

}

for(k=low;k<=high;k++)

{

arr[k]=b[k];

}

}

void partition(intarr[],intlow,int high)

{

int mid;

if(low<high)

{

mid=(low+high)/2;

partition(arr,low,mid);

partition(arr,mid+1,high);

sort(arr,low,mid,high);

}

}

void main()

{

intarr[20];

int n;

clrscr();

printf("Enter number of data:");

scanf("%d",&n);

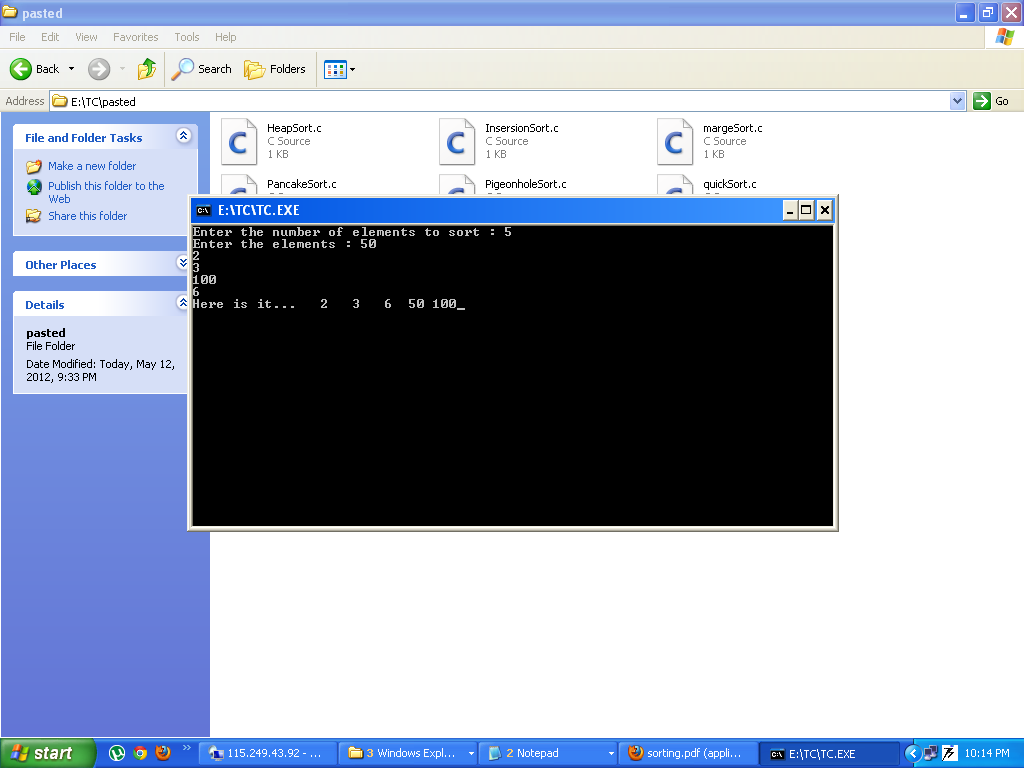
getdata(arr,n);

partition(arr,0,n-1);

display(arr,n);

getchar();

}



**8. Quicksort**

Main article: [Quicksort](http://en.wikipedia.org/wiki/Quicksort)

*Quicksort* is a [divide and conquer](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm)[algorithm](http://en.wikipedia.org/wiki/Algorithm) which relies on a *partition* operation: to partition an array an element called a *pivot* is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and [in-place](http://en.wikipedia.org/wiki/In-place_algorithm). The lesser and greater sublists are then recursively sorted. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log *n*) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower O(*n*²) performance, if at each step the [median](http://en.wikipedia.org/wiki/Median) is chosen as the pivot then the algorithm works in O(*n* log *n*). Finding the median however, is an O(n) operation on unsorted lists and therefore exacts its own penalty with sorting.

Algorithm in C:

#include<stdio.h>

#include<conio.h>

void quicksort(int [10],int,int);

int main(){

int x[20],size,i;

clrscr();

printf("Enter size of the array: ");

scanf("%d",&size);

printf("Enter %d elements: ",size);

for(i=0;i<size;i++)

scanf("%d",&x[i]);

quicksort(x,0,size-1);

printf("Sorted elements: ");

for(i=0;i<size;i++)

printf(" %d",x[i]);

getch();

return 0;

}

void quicksort(int x[10],intfirst,int last)

{

intpivot,j,temp,i;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(x[i]<=x[pivot]&&i<last)

i++;

while(x[j]>x[pivot])

j--;

if(i<j){

temp=x[i];

x[i]=x[j];

x[j]=temp;

}

}

temp=x[pivot];

x[pivot]=x[j];

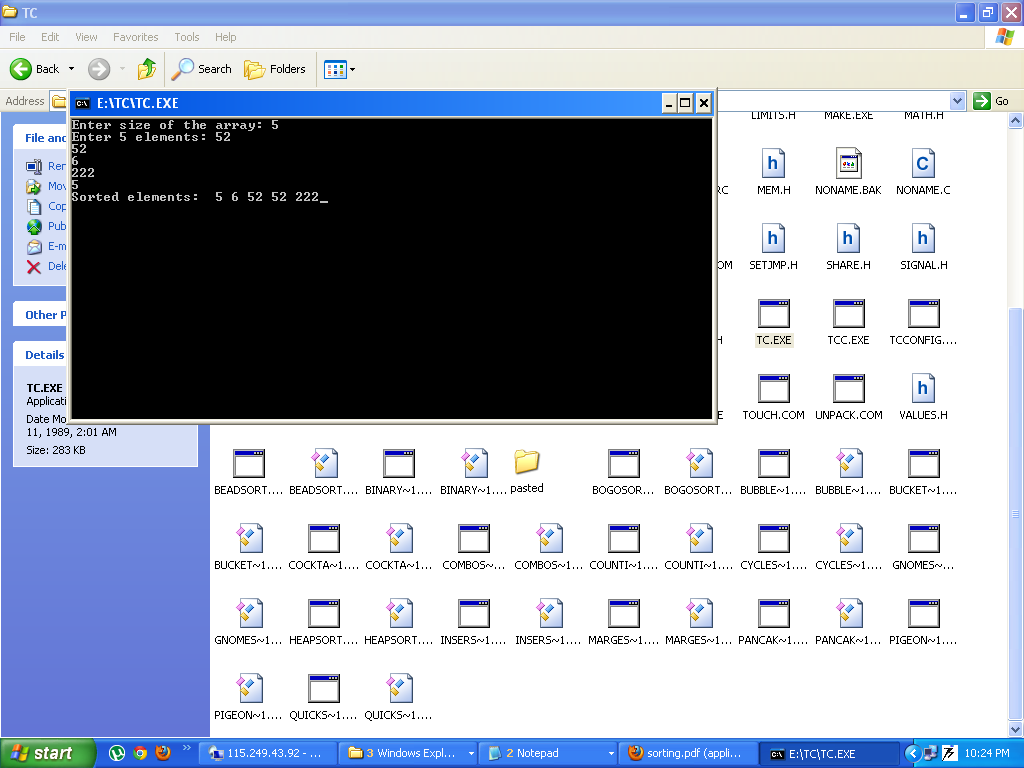
x[j]=temp;

quicksort(x,first,j-1);

quicksort(x,j+1,last);

}

}



**9. Counting sort**

Main article: [Counting sort](http://en.wikipedia.org/wiki/Counting_sort)

Counting sort is applicable when each input is known to belong to a particular set, *S*, of possibilities. The algorithm runs in O(|*S*| + *n*) time and O(|*S*|) memory where *n* is the length of the input. It works by creating an integer array of size |*S*| and using the *i*th bin to count the occurrences of the *i*th member of *S* in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm cannot often be used because *S* needs to be reasonably small for it to be efficient, but the algorithm is extremely fast and demonstrates great asymptotic behavior as *n* increases. It also can be modified to provide stable behavior.

Algorithm in C:

#include<stdio.h>

#include<conio.h>

/\* end is the last index + 1 \*/

void csort(int \*array,int end,int max, int min)

{

int i;

int range = max-min+1;

int \*count=(int\*)malloc(sizeof(int)\*(range+1));

int \*scratch=(int\*)malloc(sizeof(int)\*end);

int c;

int s;

for(i=0; i<range+1; i++)

count[i] = 0;

/\* Set the value of count[i] to the number of

\* elements in array with value i+min-1. \*/

for(i=0; i<end; i++) {

c = array[i]+1-min;

count[c]++;

}

/\* Update count[i] to be the number of

\* elements with value less than i+min. \*/

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

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

/\* Copy the elements of array into scratch in

\* stable sorted order. \*/

for(i=(end-1); i>=0; i--) {

c = array[i]-min;

s = count[c];

scratch[s] = array[i];

/\* Increment count so that the next element

\* with the same value as the current element

\* is placed into its own position in scratch. \*/

count[c]++;

}

for(i=0; i<end; i++)

array[i] = scratch[i];

}

void main()

{

int a[5]={5,6,1,2,3};

int i;

clrscr();

printf("\nBefore Sort\n");

for(i=0;i<5;i++)

{

printf("%d\n",a[i]);

}

csort(a,5,6,1);

printf("\nAfter Sort\n");

for(i=0;i<5;i++)

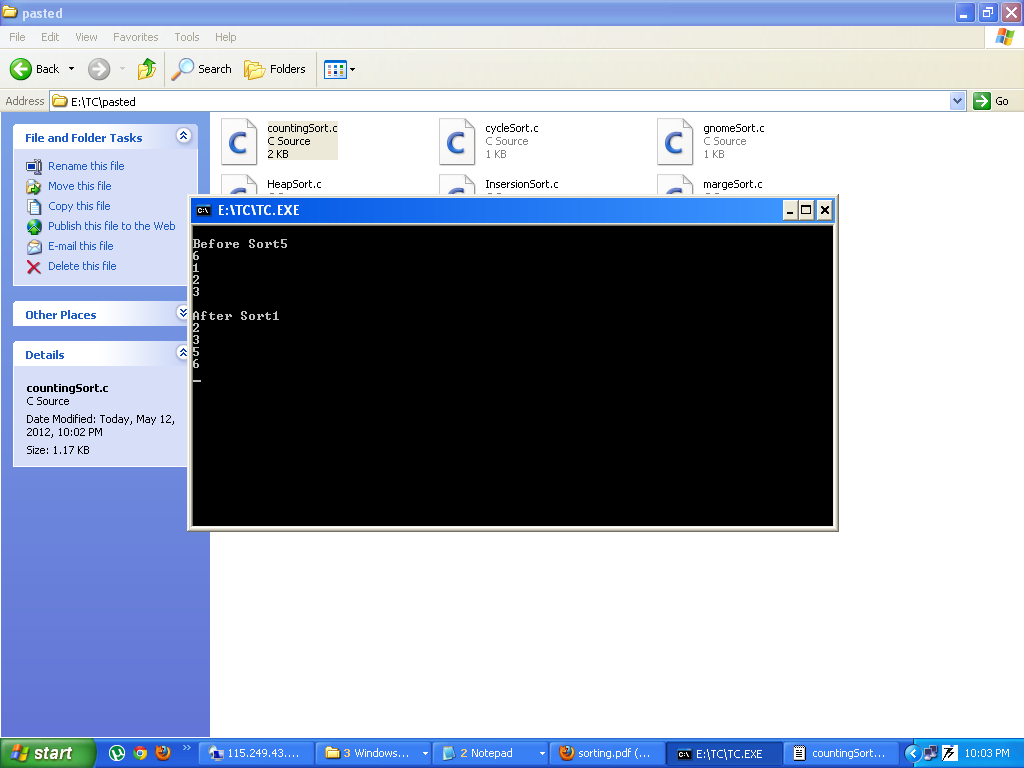
{

printf("%d\n",a[i]);

}

getch();

}



**10. Bucket sort**

Main article: [Bucket sort](http://en.wikipedia.org/wiki/Bucket_sort)

Bucket sort is a [divide and conquer](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) sorting algorithm that generalizes [Counting sort](http://en.wikipedia.org/wiki/Counting_sort) by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. A variation of this method called the single buffered count sort is faster than quicksort.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

Due to the fact that bucket sort must use a limited number of buckets it is best suited to be used on data sets of a limited scope. Bucket sort would be unsuitable for data such as social security numbers - which have a lot of variation.

Algorithm in C:

/\*Bucket sort \*/

/\*Bucket sort \*/

#include<stdio.h>

#include<conio.h>

/\*Defining the strcuture for each node in the list\*/

typedef struct node

{

float value;

struct node \*link;

} node;

void main()

{

/\*An array of structures,pointers to the structure\*/

node counter[10], \*n2,\*n1;

float ar[10] = {0.79,0.13,0.16,0.64,0.39,0.20,0.89,0.53,0.71,0.42};

float fa[10],temp;

int i,j,k=0;

float n,a;

clrscr();

printf("\Before sort: \n\n");

for (i=0;i<10;i++) printf("%f\t",ar[i]);

/\*Initializing the elements of the counter arrray to zero\*/

for(i=0;i<10;i++)

{

counter[i].value = 0;

counter[i].link = 0;

}

/\*Redcuing the value equal to index of an array\*/

for(i=0;i<10;i++)

{

n = ar[i];

j = n \* 100;

j = j/10;

/\*Moving the values in to the appropriate bucket\*/

/\*If there are no elements in the bucket\*/

if(counter[j] . value ==0 && counter[j] . link == 0)

counter[j] . value = ar[i];

else

{

/\*If there is only one element at that index\*/

if(counter[j].link==0 && counter[j] .value != 0)

{

counter[j].link=(node \*) malloc(sizeof(node));

n2 = counter[j].link;

n2 -> link = 0;

n2 -> value =ar[i];

continue;

}

/\*If there is already an node in the list at this index\*/

n2 = counter[j].link ;

while(n2 -> link !=0 )

{

n2 = n2 -> link;

}

n2 -> link =(node \*) malloc(sizeof(node));

n2 = n2 -> link;

n2 -> link=0;

n2 -> value = ar[i];

}

}

/\*Sorting of all the buckets in order\*/

printf("\nThe sorted values after merging all buckets in order are: \n\n");

for(i=0;i<10;i++)

{

/\*No nodes at that index\*/

if(counter[i] . link ==0 && counter[i] . value == 0)

continue;

else

{

n1 = &counter[i];

n2 = &counter[i] ;

/\*If there is more than one node at this Index\*/

if(n2 -> link != 0)

{

while(n1!=0)

{

while(n2!= 0)

{

if(n1 -> value > n2 -> value)

{

temp =n1 -> value;

n1 -> value =n2 -> value;

n2 -> value =temp;

}

n2 = n2 -> link;

}

n2 = n1 -> link;

n1 = n1 -> link;

}

n1 = &counter[i];

for(; n1!=0; k++)

{

fa[k] = n1 -> value;

n1 = n1 -> link;

}

}

/\*If there is only one node at this Index\*/

else

{

fa[k] = counter[i].value;

k=k+1;

}

}

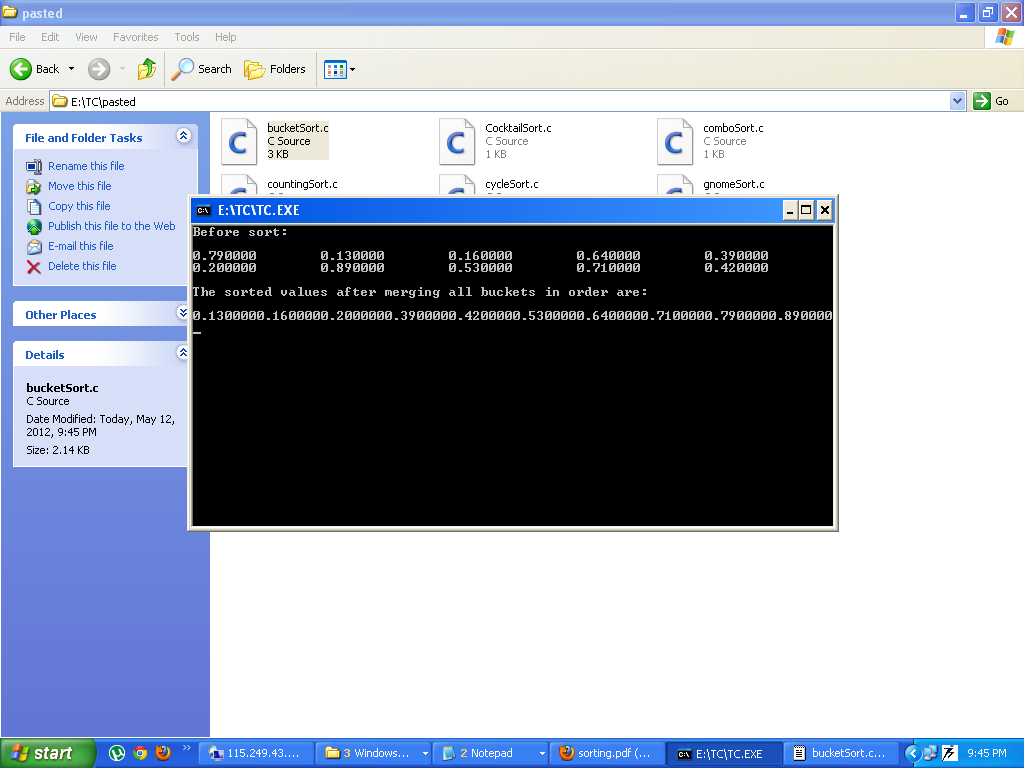
}

for(i=0;i<10;i++)

printf("%f",fa[i]);

getch();

}



**11. Radix sort**

Main article: [Radix sort](http://en.wikipedia.org/wiki/Radix_sort)

*Radix sort* is an algorithm that sorts numbers by processing individual digits. *n* numbers consisting of *k* digits each are sorted in O(*n* · *k*) time. Radix sort can process digits of each number either starting from the [least significant digit](http://en.wikipedia.org/wiki/Least_significant_digit) (LSD) or starting from the [most significant digit](http://en.wikipedia.org/wiki/Most_significant_digit) (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the [counting sort](http://en.wikipedia.org/wiki/Counting_sort) algorithm to be used internally by the radix sort. Hybrid sorting approach, such as using [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) for small bins improves performance of radix sort significantly.

Algorithm in C:

#include "stdio.h"

#include "conio.h"

#define MAX 100

#define SHOWPASS

void print(int \*a, int n) {

int i;

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

printf("%d\t", a[i]);

}

voidradix\_sort(int \*a, int n) {

int i, b[MAX], m = 0, exp = 1;

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

if (a[i] > m)

m = a[i];

}

while (m / exp> 0) {

int box[10] = { 0 };

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

box[a[i] / exp % 10]++;

for (i = 1; i < 10; i++)

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

for (i = n - 1; i >= 0; i--)

b[--box[a[i] / exp % 10]] = a[i];

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

a[i] = b[i];

exp \*= 10;

#ifdef SHOWPASS

printf("\n\nPASS : ");

print(a, n);

#endif

}

}

int main() {

intarr[MAX];

int i, num;

clrscr();

printf("\nEnter total elements (num< %d) : ", MAX);

scanf("%d", &num);

printf("\nEnter %d Elements : ", num);

for (i = 0; i <num; i++)

scanf("%d", &arr[i]);

printf("\nARRAY : ");

print(&arr[0], num);

radix\_sort(&arr[0], num);

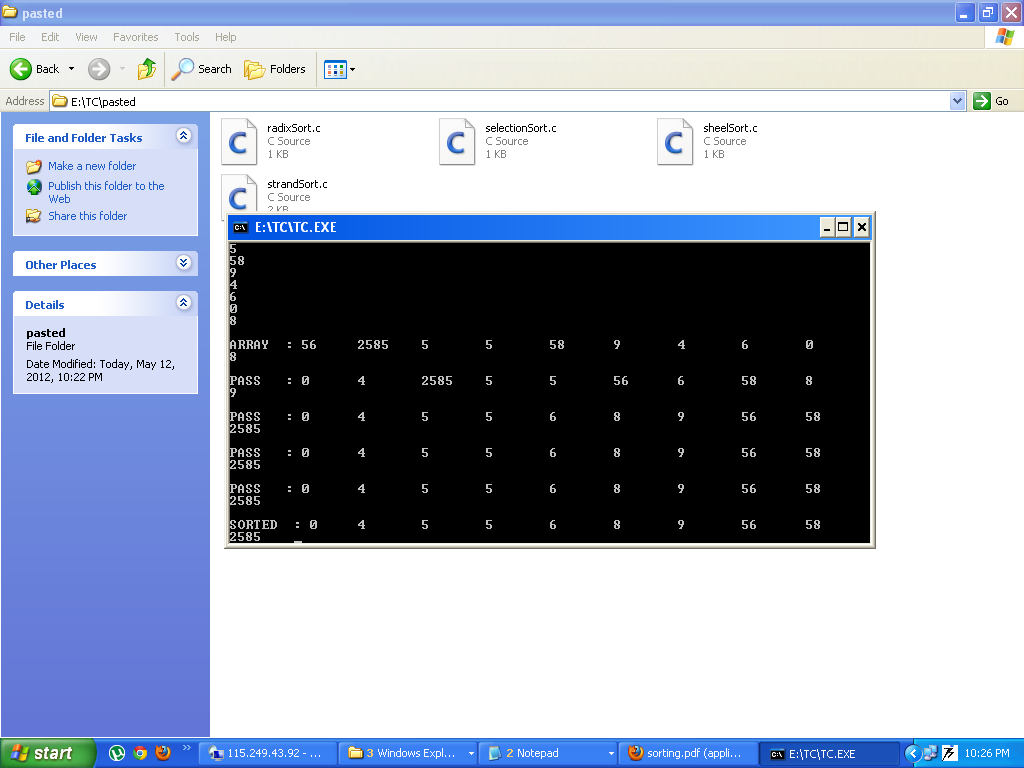
printf("\n\nSORTED : ");

print(&arr[0], num);

getch();

return 0;

}



12. Bead Sort

The bead sort operation can be compared to the manner in which beads slide on parallel poles, such as on an [abacus](http://en.wikipedia.org/wiki/Abacus). However, each pole may have a distinct number of beads. Initially, it may be helpful to imagine the beads suspended on vertical poles. In Step 1, such an arrangement is displayed using *n=5* rows of beads on *m=4* vertical poles. The numbers to the right of each row indicate the number that the row in question represents; rows 1 and 2 are representing the positive integer 3 (because they each contain three beads) while the top row represents the positive integer 2 (as it only contains two beads).[[1]](http://en.wikipedia.org/wiki/Bead_sort#cite_note-rowconventions-0)

If we then allow the beads to fall, the rows now represent the same integers in sorted order. Row 1 contains the largest number in the set, while row *n* contains the smallest. If the above-mentioned convention of rows containing a series of beads on poles 1..*k* and leaving poles *k*+1..*m* empty has been followed, it will continue to be the case here.

The action of allowing the beads to "fall" in our physical example has allowed the larger values from the higher rows to propagate to the lower rows. If the value represented by row *a* is smaller than the value contained in row *a+1*, some of the beads from row *a+1* will fall into row *a*; this is certain to happen, as row *a* does not contain beads in those positions to stop the beads from row *a+1* from falling.

The mechanism underlying bead sort is similar to that behind [counting sort](http://en.wikipedia.org/wiki/Counting_sort); the number of beads on each pole corresponds to the number of elements with value equal or less than the index of that pole.

Algorithm in C:

#include <stdio.h>

#include <stdlib.h>

#include<conio.h>

void bead\_sort(int \*a, int len)

{

int i, j, max, sum;

unsigned char \*beads;

# define BEAD(i, j) beads[i \* max + j]

for (i = 1, max = a[0]; i < len; i++)

if (a[i] > max) max = a[i];

beads = calloc(1, max \* len);

/\* mark the beads \*/

for (i = 0; i < len; i++)

for (j = 0; j < a[i]; j++)

BEAD(i, j) = 1;

for (j = 0; j < max; j++) {

/\* count how many beads are on each post \*/

for (sum = i = 0; i < len; i++) {

sum += BEAD(i, j);

BEAD(i, j) = 0;

}

/\* mark bottom sum beads \*/

for (i = len - sum; i < len; i++) BEAD(i, j) = 1;

}

for (i = 0; i < len; i++) {

for (j = 0; j < max && BEAD(i, j); j++);

a[i] = j;

}

free(beads);

}

int main()

{

int i, x[] = {5, 3, 1, 7, 4, 1, 1, 20};

int len = sizeof(x)/sizeof(x[0]);

clrscr();

printf("Before Sort");

for ( i=0;i<len;i++)

{

printf("%d\t",x[i]);

}

bead\_sort(x, len);

printf ("\nAfter Sort");

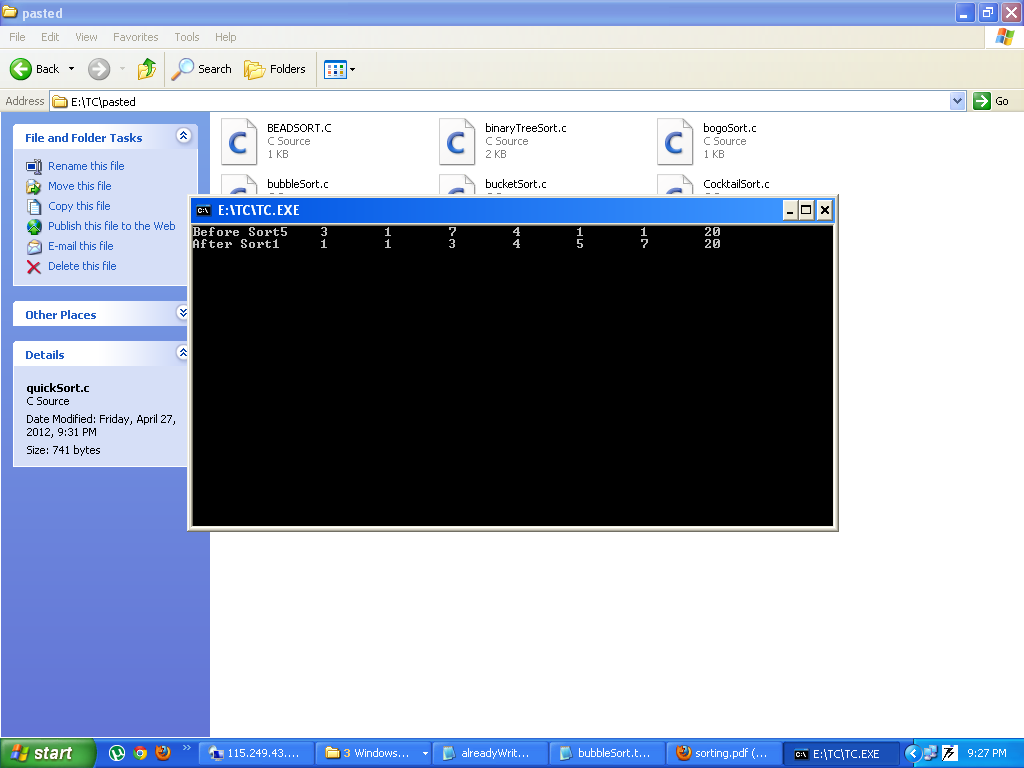
for (i = 0; i < len; i++)

printf("%d\t", x[i]);

getch();

return 0;

}



13. Binary Tree Sort:

A tree sort is a [sort algorithm](http://en.wikipedia.org/wiki/Sort_algorithm) that builds a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree) from the keys to be sorted, and then traverses the tree ([in-order](http://en.wikipedia.org/wiki/Tree_traversal)) so that the keys come out in sorted order. Its typical use is when sorting the elements of a stream from a file. Several other sorts would have to load the elements to a temporary data structure, whereas in a tree sort the act of loading the input into a data structure is sorting it.

Algorithm in C:

#include<stdio.h>

#include<conio.h>

#define TRUE 1

#define FALSE 0

struct btreenode

{

struct btreenode \*rightchild;

int data;

struct btreenode \*leftchild;

};

insert(struct btreenode \*\*sr,int num)

{

if(\*sr==NULL)

{

\*sr=malloc(sizeof(struct btreenode));

(\*sr)->leftchild=NULL;

(\*sr)->data=num;

(\*sr)->rightchild=NULL;

return;

}

else

{

if(num< (\*sr)->data)

insert(&((\*sr)->leftchild),num);

else

insert(&((\*sr)->rightchild),num);

}

return;

}

inorder(struct btreenode \*sr)

{

if(sr!=NULL)

{

inorder(sr->leftchild);

printf("%d ",sr->data);

inorder(sr->rightchild);

}

else

return;

}

postorder(struct btreenode \*sr)

{

if(sr!=NULL)

{

postorder(sr->rightchild);

printf("%d ",sr->data);

postorder(sr->leftchild);

}

else

return;

}

void main()

{

struct btreenode \*bt;

int req,i=0,num,a[10],no;

bt=NULL;

clrscr();

while(i < 5)

{

printf("\nEnter value to be inserted: ");

scanf("%d",&a[i]);

insert(&bt,a[i]);

i++;

}

clrscr();

printf("\n\nSorted Binary tree in ascending order== > \n\n");

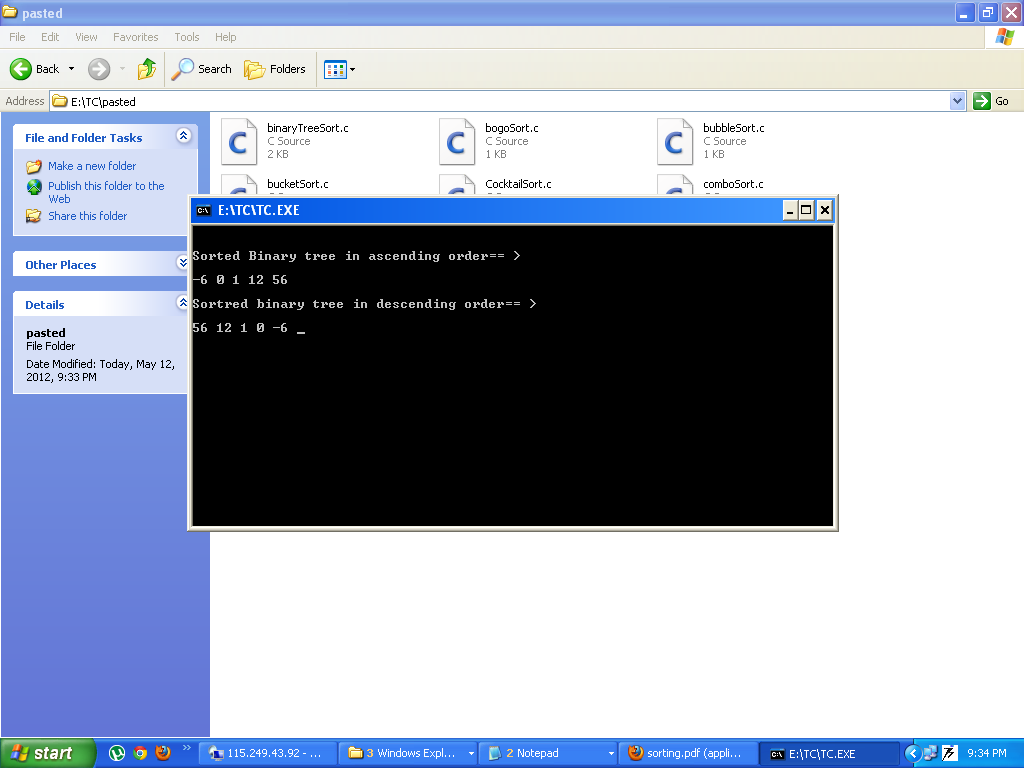
inorder(bt);

printf("\n\nSortred binary tree in descending order== >\n\n");

postorder(bt);

getch();

}



14. Bogo Sort:

In [computer science](http://en.wikipedia.org/wiki/Computer_science), **bogosort** is a particularly ineffective [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) based on the [generate and test](http://en.wikipedia.org/wiki/Trial_and_error) paradigm. It is not useful for sorting, but may be used for educational purposes, to contrast it with other more realistic algorithms; it has also been used as an example in [logic programming](http://en.wikipedia.org/wiki/Logic_programming).[[2]](http://en.wikipedia.org/wiki/Bogosort#cite_note-KSFS-1)[[4]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Naish86-3)[[5]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Naish95-4) If bogosort were used to sort a [deck of cards](http://en.wikipedia.org/wiki/Deck_of_cards), it would consist of checking if the deck were in order, and if it were not, throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted. Its name comes from the word *bogus*.

Algorithm in C:

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

int is\_sorted(int \*a, int n)

{

while ( --n >= 1 ) {

if ( a[n] < a[n-1] ) return 0;

}

return 1;

}

void shuffle(int \*a, int n)

{

int i, t, r;

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

t = a[i];

r = rand() % n;

a[i] = a[r];

a[r] = t;

}

}

void bogosort(int \*a, int n)

{

while ( is\_sorted(a, n) != 1 ) shuffle(a, n);

}

int main()

{

int numbers[] = { 1, 10, 9, 7, 3, 0 };

int i;

clrscr();

printf("Before Sort\n");

for (i=0; i < 6; i++) printf("%d ", numbers[i]);

bogosort(numbers, 6);

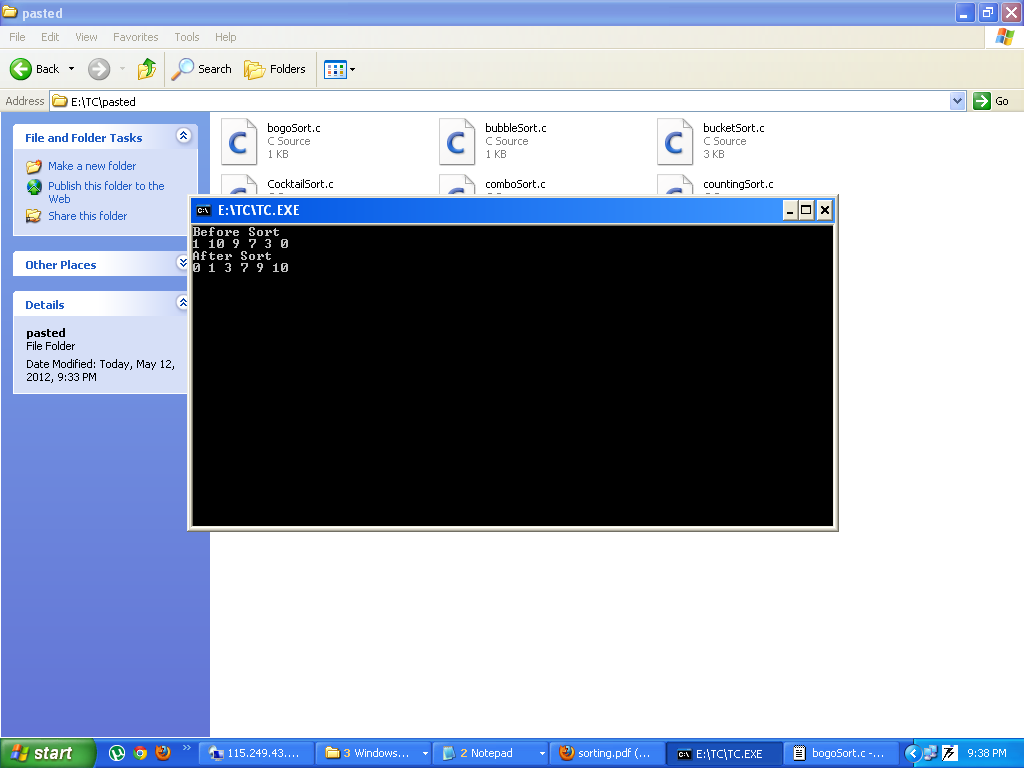
printf("\nAfter Sort\n");

for (i=0; i < 6; i++) printf("%d ", numbers[i]);

printf("\n");

getch();

}



15. Cocktail Sort:

**Cocktail sort**, also known as **bidirectional bubble sort**, **cocktail shaker sort**, **shaker sort** (which can also refer to a variant of [selection sort](http://en.wikipedia.org/wiki/Selection_sort)), **ripple sort**, **shuffle sort**, **shuttle sort** or **happy hour sort**, is a variation of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) that is both a [stable](http://en.wikipedia.org/wiki/Stable_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) and a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). The algorithm differs from a [bubble sort](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles) in bubble sorts.

Algorithm In C:

#include <stdio.h>

#include<conio.h>

#define try\_swap { if (a[i] < a[i - 1])\

{ t = a[i]; a[i] = a[i - 1]; a[i - 1] = t; t = 0;} }

void cocktailsort(int \*a, size\_t len)

{

size\_t i;

int t = 0;

while (!t) {

for (i = 1, t = 1; i < len; i++) try\_swap;

if (t) break;

for (i = len - 1, t = 1; i; i--) try\_swap;

}

}

int main()

{

int x[] = { 5, -1, 101, -4, 0, 1, 8, 6, 2, 3 };

size\_t i, len = sizeof(x)/sizeof(x[0]);

clrscr();

printf("Before Sorting\n");

for (i = 0; i < len; i++)

printf("%d\n", x[i]);

cocktailsort(x, len);

printf("\nAfter Sorting\n");

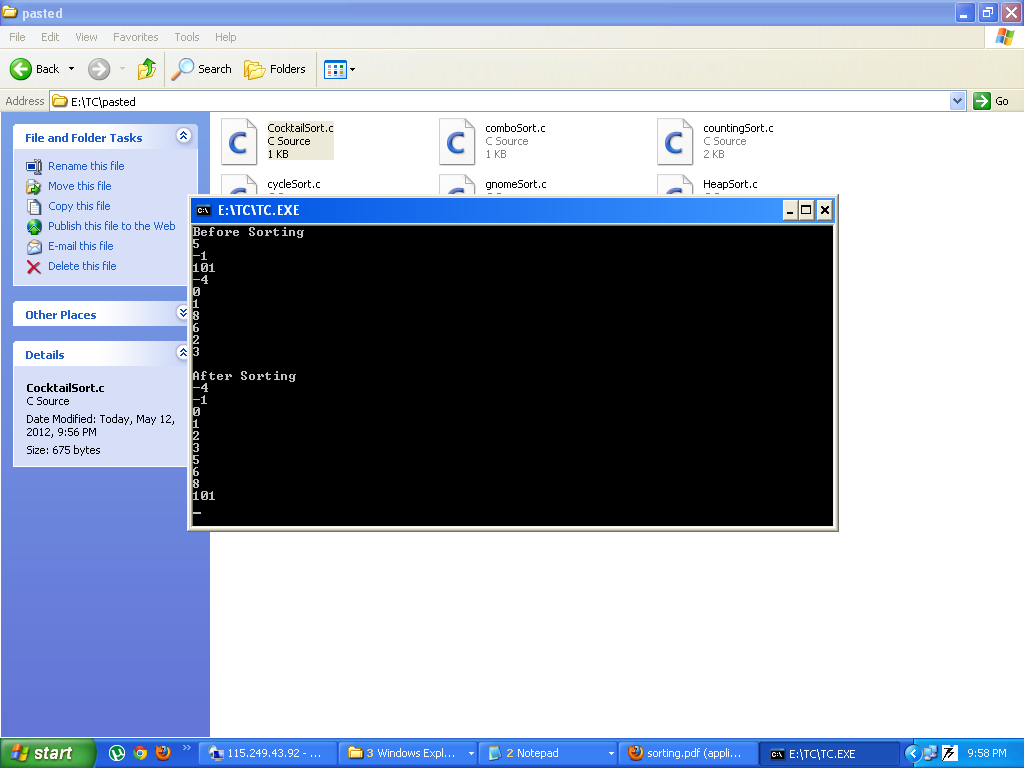
for (i = 0; i < len; i++)

printf("%d\n", x[i]);

getch();

return 0;

}



16. Cycle Sort:

**Cycle sort** is an in-place, unstable [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) that is theoretically optimal in terms of the total number of writes to the original [array](http://en.wikipedia.org/wiki/Array_data_structure), unlike any other in-place sorting algorithm. It is based on the idea that the [permutation](http://en.wikipedia.org/wiki/Permutation) to be sorted can be factored into [cycles](http://en.wikipedia.org/wiki/Cycle_notation), which can individually be rotated to give a sorted result.

Unlike nearly every other sort, items are *never* written elsewhere in the array simply to push them out of the way of the action. Each value is either written zero times, if it's already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with [EEPROMs](http://en.wikipedia.org/wiki/EEPROM) or [Flash memory](http://en.wikipedia.org/wiki/Flash_memory) where each write reduces the lifespan of the memory.

Algorithm In C:

#include <stdio.h>

#include <conio.h>

int cycleSort(int\* array,int len) {

int writes = 0;

int item;

int cycleStart;

int pos;

int i;

int temp;

for (cycleStart = 0; cycleStart < len-1; cycleStart++) {

item = array[cycleStart];

pos = cycleStart;

for (i = cycleStart + 1; i < len; i++)

if (array[i]< item ) pos++;

if (pos == cycleStart) continue;

while (item==array[pos]) pos++;

{

temp = array[pos];

array[pos] = item;

item = temp;

}

writes++;

while (pos != cycleStart) {

pos = cycleStart;

for (i = cycleStart + 1; i < len; i++)

if (array[i]<item) pos++;

while (item==array[pos]) pos++;

{

temp = array[pos];

array[pos] = item;

item = temp;

}

writes++;

}

}

return writes;

}

void main ()

{

int arr[5]={10,2,3,0,20};

int len=5;

int i;

clrscr();

printf("\nBefore Sort\n");

for ( i=0;i<len;i++)

{

printf("%d\n",arr[i]);

}

cycleSort(arr,len);

printf("\nAfter Sort\n");

for ( i=0;i<len;i++)

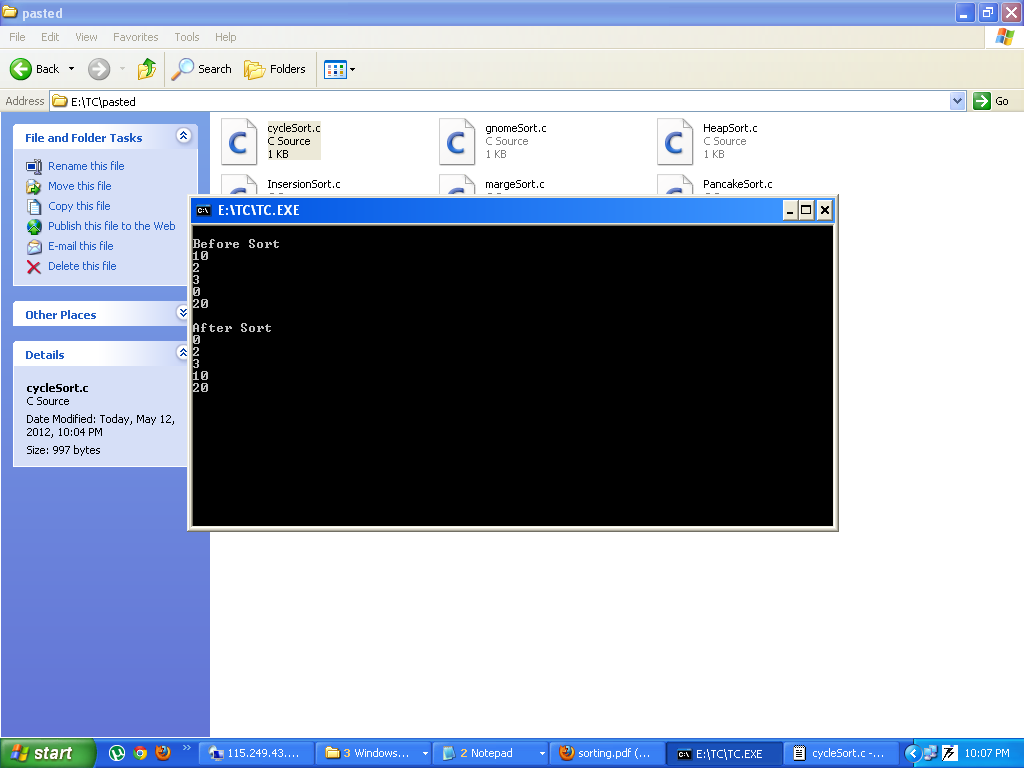
{

printf("%d\n",arr[i]);

}

getch();

}



17. Gnome Sort:

**Gnome sort**, originally proposed by [Hamid Sarbazi-Azad](http://en.wikipedia.org/w/index.php?title=Hamid_Sarbazi-Azad&action=edit&redlink=1) in 2000 and called [Stupid sort](http://sina.sharif.edu/%7Eazad/stupid-sort.PDF) (not to be confused with [Bogosort](http://en.wikipedia.org/wiki/Bogosort)), and then later on described by [Dick Grune](http://en.wikipedia.org/wiki/Dick_Grune) and named "Gnome sort", is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) which is similar to [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), except that moving an element to its proper place is accomplished by a series of swaps, as in [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort). It is conceptually simple, requiring no nested loops. The running time is [O](http://en.wikipedia.org/wiki/Big_O_notation)(n^2), but tends towards O(*n*) if the list is initially almost sorted. In practice the algorithm can run as fast as [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort)[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]. The average runtime is O(n^2).

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.

The name comes from the supposed behavior of the Dutch [garden gnome](http://en.wikipedia.org/wiki/Garden_gnome) in sorting a line of flowerpots and is described on Dick Grune's [Gnome sort page](http://www.cs.vu.nl/%7Edick/gnomesort.html):

Gnome Sort is based on the technique used by Dutch [Garden Gnomes](http://en.wikipedia.org/wiki/Garden_Gnome) (Du.: tuinkabouter). 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.

Algorithm In C:

#include <stdio.h>

#include <conio.h>

void gnome\_sort(int \*a, int n)

{

int i=1, j=2, t;

# define swap(i, j) { t = a[i]; a[i] = a[j]; a[j] = t; }

while(i < n) {

if (a[i - 1] > a[i]) {

swap(i - 1, i);

if (--i) continue;

}

i = j++;

}

# undef swap

}

void main ()

{

int a[10]={1,2,30,0,-2,8,6,8,6,3};

int count=10;

int i;

clrscr();

printf("\nBefore Sort\n");

for (i=0;i<count;i++)

{

printf ("%d\n",a[i]);

}

gnome\_sort(a,count);

printf("\nAfter Sort\n");

for (i=0;i<count;i++)

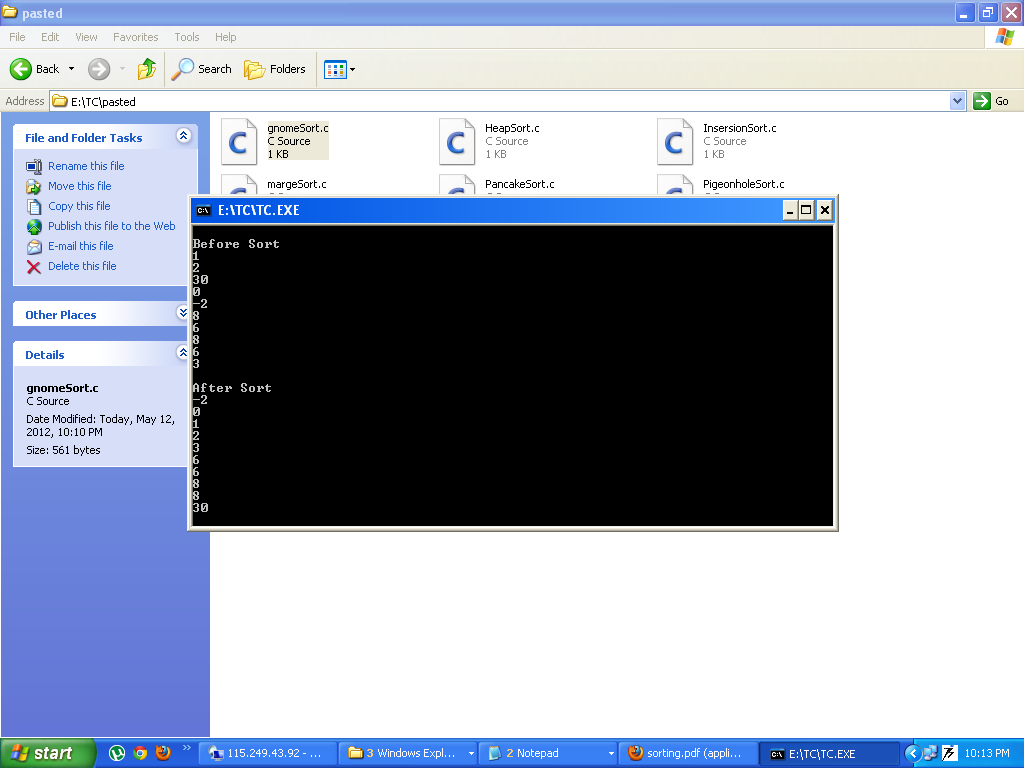
{

printf ("%d\n",a[i]);

}

getch();

}



18. Pancake Sort:

**Pancake sorting** is a variation of the [sorting](http://en.wikipedia.org/wiki/Sorting_algorithm) problem in which the only allowed operation is to reverse the elements of some *prefix* of the sequence. Unlike a traditional sorting algorithm, which attempts to sort with the fewest comparisons possible, the goal is to sort the sequence in as few reversals as possible. This operation can be visualized by thinking of a stack of [pancakes](http://en.wikipedia.org/wiki/Pancake) in which one is allowed to take the top *k* pancakes and flip them. A variant of the problem is concerned with *burnt* pancakes, where each pancake has a burnt side and all pancakes must, in addition, end up with the burnt side on top.

Algorithm In C:

#include<stdio.h>

#include<conio.h>

void do\_flip(int \*list, int length, int num)

{

int swap;

int i=0;

for(i;i<--num;i++)

{

swap=list[i];

list[i]=list[num];

list[num]=swap;

}

}

int pancake\_sort(int \*list, int length)

{

int i,a,max\_num\_pos,moves;

moves=0;

if(length<2)

return 0;

for(i=length;i>1;i--)

{

max\_num\_pos=0;

for(a=0;a<i;a++)

{

if(list[a]>list[max\_num\_pos])

max\_num\_pos=a;

}

if(max\_num\_pos==i-1)

continue;

if(max\_num\_pos)

{

moves++;

do\_flip(list, length, max\_num\_pos+1);

}

moves++;

do\_flip(list, length, i);

}

return moves;

}

int main(int argc, char \*\*argv)

{

int list[9];

int i;

int moves;

clrscr();

srand(time(NULL));

for(i=0;i<9;i++)

list[i]=rand()%100;

printf("\nOriginal: ");

for (i=0;i<9;i++)

{

printf("%d\t",list[i]);

}

moves = pancake\_sort(list, 9);

printf("\nSorted: ");

for (i=0;i<9;i++)

{

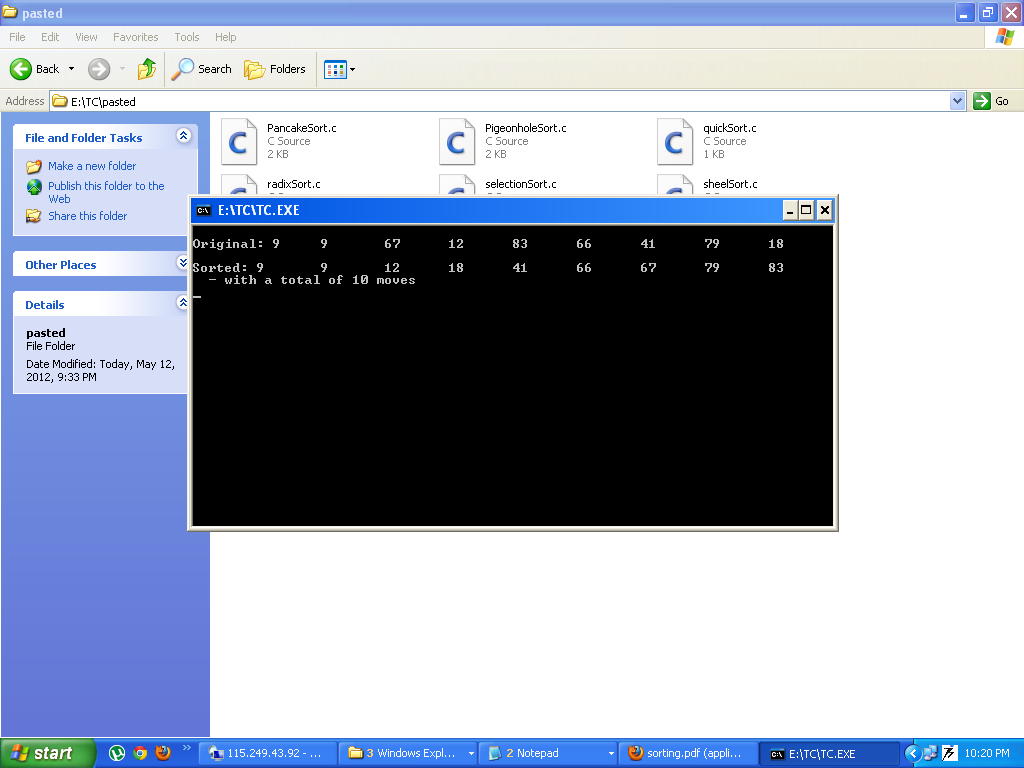
printf("%d\t",list[i]);

}

printf(" - with a total of %d moves\n", moves);

getch();

}



19. Pigeonhole Sort:

**Pigeonhole sorting**, also known as **count sort** (not to be confused with [counting sort](http://en.wikipedia.org/wiki/Counting_sort)), is a [sorting algorithm](http://en.wikipedia.org/wiki/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.[[1]](http://en.wikipedia.org/wiki/Pigeonhole_sort#cite_note-0) It requires [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n* + *N*) time.

The pigeonhole algorithm works as follows:

1. Given an array of values to be sorted, set up an auxiliary array of initially empty "pigeonholes," one pigeonhole for each key through the [range](http://en.wikipedia.org/wiki/Range_%28computer_science%29) of the original array.
2. Going over the original array, put each value into the pigeonhole corresponding to its key, such that each pigeonhole eventually contains a list of all values with that key.
3. Iterate over the pigeonhole array in order, and put elements from non-empty pigeonholes 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, 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, moving items only once. Pigeonhole sort, in contrast, moves items twice: once onto the pigeonhole/bucket array and again onto the destination array.

For arrays where *N* is much larger than *n*, [bucket sort](http://en.wikipedia.org/wiki/Bucket_sort) is a generalization that is more efficient in space and time.

Algorithm in C:

#include<stdio.h>

#include<conio.h>

void pigeonsort( int max, int \*random, int pigeon[] )

{

typedef struct node

{

int data;

struct node\* next;

}NODE;

NODE \*\*hashtable, \*temp, \*p1, \*p2;

int i;

int j;

int hashnum;

int hashed;

float tempdata;

int counter[2] = {0,0};

memcpy( pigeon, random, (sizeof(int)\*max));

hashnum= (int)(max\*1.5);

hashtable = (NODE\*\*) malloc(sizeof(NODE\*)\*hashnum);

for(i = 0; i < hashnum; i++)

{

hashtable[i] = NULL;

}

for (i = 0; i < max; i++)

{

hashed = (int)((long)pigeon[i]\*(long)hashnum/1000);

if (hashtable[hashed]==NULL)

{

counter[0]++;

hashtable[hashed] = (NODE\*) malloc(sizeof(NODE));

hashtable[hashed]->data = pigeon[i];

hashtable[hashed]->next=NULL;

}

else

{

temp = (NODE\*)malloc(sizeof(NODE));

p2 = hashtable[hashed];

p1 = p2;

counter[0]++;

if(pigeon[i]<= p2->data)

{

hashtable[hashed] = temp;

temp->data = pigeon[i];

temp->next = p1;

}

else

{

while(pigeon[i] > p2->data )

{

counter[0]++;

p1 = p2;

p2 = p2->next;

if(p2 == NULL) break;

}

counter[1]++;

p1->next = temp;

temp->data = pigeon[i];

temp->next = p2;

}

}

}

for (i = 0, j = 0; i < hashnum; i++)

{

if(hashtable[i] != NULL)

{

p1 = hashtable[i];

while(p1 != NULL)

{

pigeon[j] = p1->data;

p2 = p1;

p1 = p1->next;

j++;

free(p2);

}

}

}

printf(" Unsorted || Sorted\n");

printf("----------||----------\n");

for(i = 0; i < max; i++)

{

printf(" %4d || %4d \n", random[i], pigeon[i]);

}

printf("Press Any Key to Continue");

getch();

free(hashtable);

}

void main()

{

int arr[5]={10,2,20,3,0};

int i;

int pgn[]={2,3};

clrscr();

pigeonsort( 20, arr , pgn );

for(i=0;i<5;i++)

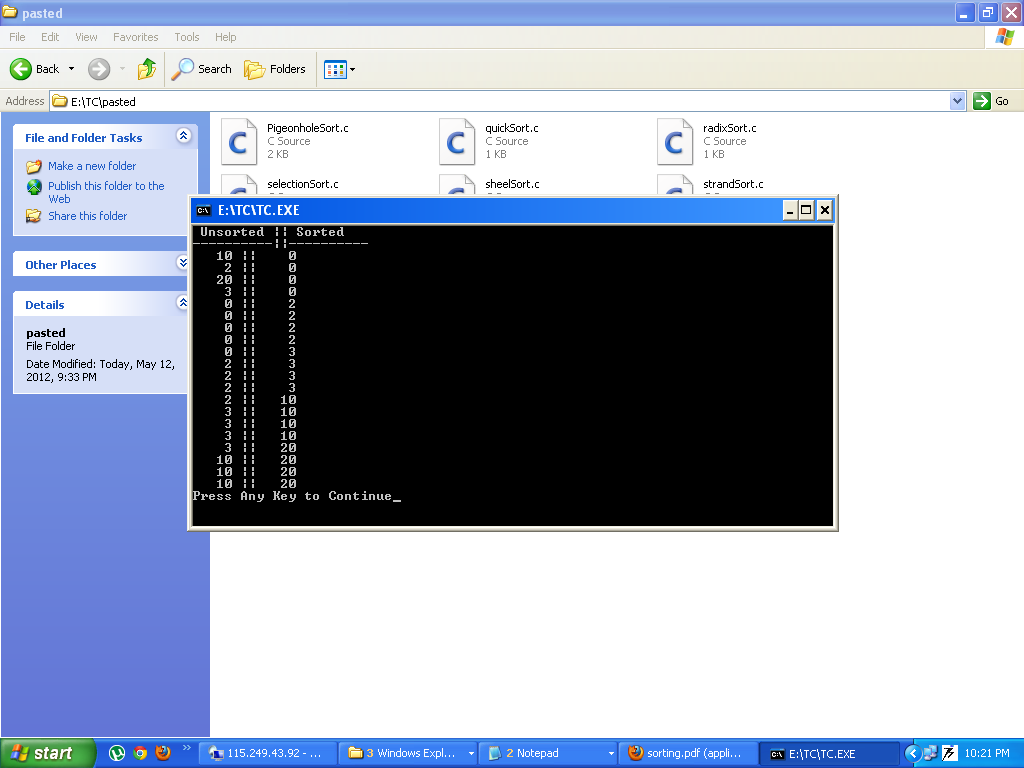
{

printf("%d\n",arr[i]);

}

getch();

}



20. Strand Sort:

**Strand sort** is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm). It works by repeatedly pulling sorted sublists out of the list to be sorted and merging them with a result array. Each iteration through the unsorted list pulls out a series of elements which were already sorted, and [merges](http://en.wikipedia.org/wiki/Mergesort) 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](http://en.wikipedia.org/wiki/Comparison_sort) due to its use of comparisons when removing strands and when merging them into the sorted array.

The strand sort algorithm is [O](http://en.wikipedia.org/wiki/Big-O_notation)(*n*2) 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.

Algorithm in C:

#include <stdio.h>

#include<conio.h>

typedef struct node\_t \*node, node\_t;

struct node\_t { int v; node next; };

void sort(int \*ar, int len)

{

node\_t \*all, head, shead, merged, \*cur, \*next, \*stail;

int i;

all = (node\_t\*)malloc(sizeof(node\_t)\*len);

/\* linkify \*/

for (i = 0; i < len; i++) {

all[i].v = ar[i];

all[i].next = all + i + 1;

}

all[len - 1].next = 0;

head.next = all;

shead.next = merged.next = 0;

while (head.next) {

/\* take strand \*/

cur = &head;

stail = shead.next = head.next;

while (next = cur->next) {

if (next->v >= stail->v) {

cur->next = next->next;

stail = stail->next = next;

} else

cur = next;

}

stail->next = 0;

/\* merge \*/

cur = merged.next;

next = shead.next;

stail = &merged;

/\* while both lists contain elements, append the smaller one \*/

while (next && cur) {

if (next->v <= cur->v) {

stail = stail->next = next;

next = next->next;

} else {

stail = stail->next = cur;

cur = cur->next;

}

}

/\* append the rest of the survivor to the end of merged \*/

stail->next = next ? next : cur;

}

cur = &merged;

len = 0;

while (cur = cur->next) ar[len++] = cur->v;

}

int main()

{

int x[] = {-2,0,-2,5,5,3,-1,-3,5,5,0,2,-4,4,2};

int i;

#define SIZE sizeof(x)/sizeof(int)

clrscr();

printf("before sort:");

for (i = 0; i < SIZE; i++) printf(" %3d", x[i]);

sort(x, sizeof(x)/sizeof(int));

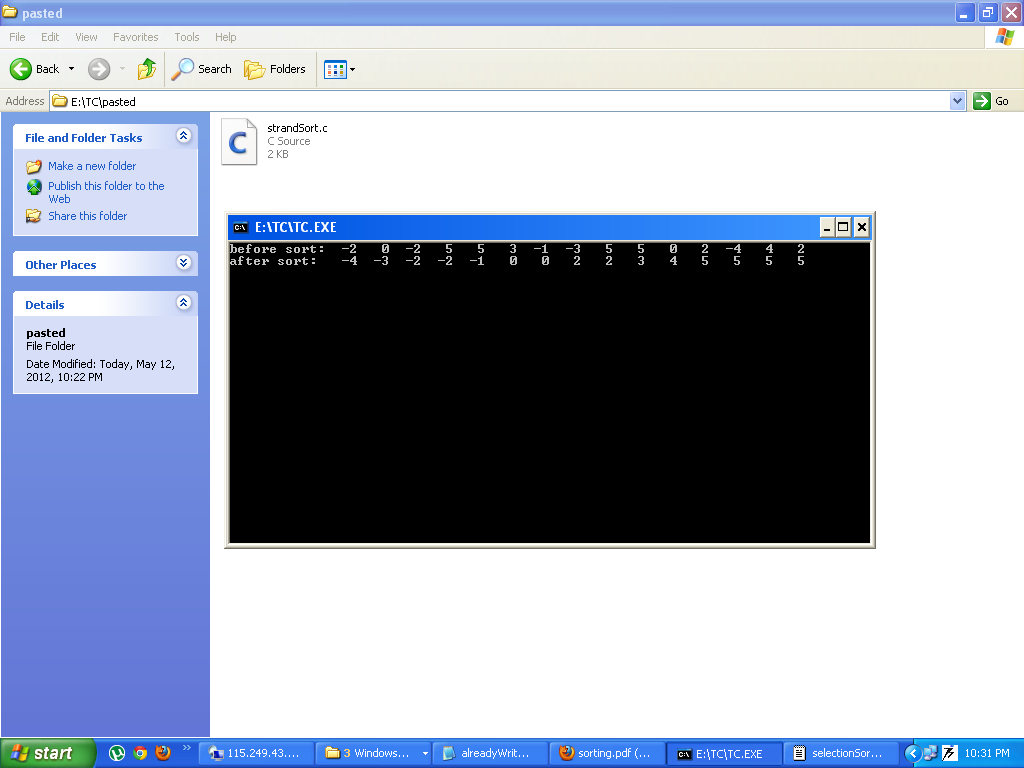
printf("\nafter sort: ");

for (i = 0; i < SIZE; i++) printf(" %3d", x[i]);

printf("\n");

getch();

}



Conclusions

By analyzing an algorithm, we mean to study the performance of an algorithm including the assertion of its correctness and a determination of the cost of its execution. Although a given algorithm is often analyzed in a particular way that is most suitable for such an algorithm, we are more interested in general procedures and techniques that can be used to study the performance of classes of algorithms. To be able to talk about general analysis techniques will not only add to our understanding of the behavior of a class of algorithms but will also, in many cases, lead to useful synthesis procedures. A good example illustrating these points is the various techniques that can be used to analyze a class of sorting algorithms which can be modelled as networks made up of comparator modules. In this paper, we discuss several approaches to such an analysis problem. Moreover, synthesis procedures suggested by these analysis techniques will also be presented.

12. References

[1] B. Brej¶ova, Analyzing variants of Shellsort, *Information Processing Letters,* 79:5

(2001), 223{227.

[2] H. Buhrman, T. Jiang, M. Li and P. Vitanyi, New applications of the incompress-

ibility method: Part II, *Theoretical Computer Science,* 235:1 (2000), 59{70.

[3] W. Dobosiewicz, An e±cient variant of bubble sort, *Information Processing Letters,*

11:1 (1980), 5{6.

[4] R. W. Floyd, Algorithm 245: Treesort 3. *Communications of the ACM,* 7 (1964),

701.

[5] J. Incerpi and R. Sedgewick, Improved upper bounds on Shellsort, *Journal of*

*Computer and System Sciences,* 31 (1985), 210{224.

[6] J. Incerpi and R. Sedgewick, Practical variations of Shellsort, *Information Process-*

*ing Letters,* 26:1 (1980), 37{43.

[7] S. Janson and D. E. Knuth, Shellsort with three increments, *Random Struct. Alg.,*

10 (1997), 125{142.

[8] S. V. Kerov and A. M. Versik, Asymptotics of the Plancherel measure on symmetric

group and the limiting form of the Young tableaux, *Soviet Math. Dokl.,* 18 (1977),

527{531.

[9] J. F. C. Kingman, The ergodic theory of subadditive stochastic processes, *Ann.*

*Probab.,* 1 (1973), 883{909.

[10] A. N. Kolmogorov, Three approaches to the quantitative de¯nition of information,

*Problems Inform. Transmission,* 1:1 (1965), 1{7.

[11] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,*

Addison-Wesley, 1973 (1st Edition), 1998 (2nd Edition).

[12] T. Jiang, M. Li and P. Vitanyi, Average complexity of Shellsort (preliminary

version), *Proc. ICALP99,* Lecture Notes in Computer Science, Vol. 1644, Springer-

Verlag (Berlin, 1999), pp. 453{462.

[13] T. Jiang, M. Li and P. Vitanyi, A lower bound on the average-case complexity of

Shellsort, *J. Assoc. Comp. Mach.,* 47:5 (2000), 905{911.

[14] T. Jiang, M. Li and P. Vitanyi, Average-case analysis of algorithms using Kol-

mogorov complexity, *Journal of Computer Science and Technology,* 15:5 (2000),

402{408.

232 *P. Vit¶anyi*

[15] T. Jiang, M. Li and P. Vit¶anyi, The average-case area of Heilbronn-type triangles,

*Random Structures and Algorithms,* 20:2 (2002), 206{219.

[16] M. Li and P. M. B. Vit¶anyi, *An Introduction to Kolmogorov Complexity and its*

*Applications,* Springer-Verlag, 2nd Edition (New York, 1997).

[17] B. F. Logan and L. A. Shepp, A variational problem for random Young tableaux,

*Advances in Math.,* 26 (1977), 206{222.

[18] I. Munro, Personal communication, 1992.

[19] A. Papernov and G. Stasevich, A method for information sorting in computer

memories, *Problems Inform. Transmission,* 1:3 (1965), 63{75.

[20] B. Poonen, The worst-case of Shellsort and related algorithms, *J. Algorithms,* 15:1

(1993), 101{124.

[21] C. G. Plaxton, B. Poonen and T. Suel, Improved lower bounds for Shellsort, in:

*Proc. 33rd IEEE Symp. Foundat. Comput. Sci.* (1992), pp. 226{235.

[22] V. R. Pratt, *Shellsort and Sorting Networks,* Ph.D. Thesis, Stanford Univ. (1972).

[23] R. Scha®er and R. Sedgewick, *J. Algorithms,* 15 (1993), 76{100.

[24] R. Sedgewick, Analysis of Shellsort and related algorithms, presented at the *Fourth*

*Annual European Symposium on Algorithms* (Barcelona, September, 1996).

[25] R. Sedgewick, Open problems in the analysis of sorting and searching algorithms,

Presented at *Workshop on Prob. Analysis of Algorithms* (Princeton, 1997).

[26] D. L. Shell, A high-speed sorting procedure, *Commun. ACM,* 2:7 (1959), 30{32.

[27] R. E. Tarjan, Sorting using networks of queues and stacks, *Journal of the ACM,*

19 (1972), 341{346.

[28] M. A. Weiss and R. Sedgewick, Bad cases for Shaker-sort, *Information Processing*

*Letters,* 28:3 (1988), 133{136.

[29] J. W. J. Williams *Comm. ACM,* 7 (1964), 347{348.

[30] A. C. C. Yao, An analysis of (*h; k;* 1)-Shellsort, *J. of Algorithms,* 1 (1980), 14{50.