SORTING ALGORITHMS

Implementation in C language

General Information

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 (alphabetic order).

Sorting is the first step in solving a host of other algorithm problems. For example efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly, or it is often useful for canonicalizing data (a process for converting data that has more than one possible representation into a "standard" canonical representation) and for producing human-readable output.

Classification

Sorting algorithms used in computer science are often classified by:

- **Computational complexity of element** comparisons in terms of the size of the list.
- **Computational complexity of swaps** (for "in place" algorithms).
- **Computational time**.
- **Memory usage** (and computer resources).
- **Recursion**. Some algorithms are either recursive or non recursive, while others may be both

Sorting Methods

Here are the simplest Sorting Algorithms:

- **Bubble Sort**
- **Insertion Sort**
- **Linear Selection Sort**
- **Selection and Change Sort**

Bubble Sort

Bubble sort is a straightforward and simplistic method of sorting data.

The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

Because it only uses comparisons to operate on elements, it is a *comparison sort.*

- **The algorithm starts at the beginning of the data set.**
- **It compares the first two elements**
- **If the first and second are in wrong order, 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.**

Bubble Sort

While simple, this algorithm is highly inefficient.

It has worst-case complexity $O(n^2)$, where n is the number of items being sorted.

Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students.

Code in C language

```
void BubbleSort(int a[], int n)
{
 int  i,  j,  temp,  flag;
 for(i=n-1; i>0; i--)
 {
  flag = 1;
  for(j=0; j<i; j++)
  {
   if(a[j]>a[j+1])
   {
    flag = 0;
    temp = a[j];
    a[j] = a[j+1];
    a[j+1] = temp;
   }
  }
  //out this block when flag is true
  //i.e. inner loop performed no swaps, so the list is already sorted
  if(flag)
   break;
 }
}
```

Insertion Sort

Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time.

It is much less efficient on large lists than more advanced algorithms, but it has various advantages:

- *Simple to implement*
- *Efficient on (quite) small data sets*
- *Efficient on data sets which are already substantially sorted*
- *More efficient in practice than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort*
- *Stable (does not change the relative order of elements with equal keys)*
- *It is an online algorithm, in that it can sort a list as it receives it.*

Insertion Sort

In the best case of an already sorted array, this implementation of insertion sort takes O(n) time: in each iteration, the first remaining element of the input is only compared with the last element of the sorted subsection of the array.

The worst case is an array sorted in reverse order, as every execution of the inner loop will have to scan and shift the entire sorted section of the array before inserting the next element. Insertion sort takes $O(n^2)$ time in this worst case as well as in the average case, which makes it impractical for sorting large numbers of elements.

Code in C language

```
void insertSort(int a[ ], size_t length) {
   int i, j, value;

   for(i = 1; i < length; i++) {
      value = a[i];
      for (j = i - 1; (j >= 0) && (value< a[j]); j--) {
         a[j + 1] = a[j];
      }
      a[j+1] = value;
   }
}
```

Linear Selection Sort

Selection sort is a simple sorting algorithm that improves on the performance of bubble sort.

The algorithm works as follows:

- **Find the minimum value in the list**
- **Swap it with the value in the first position**
- **Repeat the steps above for remainder of the list (starting at the second position)**

Effectively, we divide the list into two parts: the sublist of items already sorted, which we build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Linear Selection Sort

Selecting the lowest element requires scanning all n elements (this take n - 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n - 1 elements and so on, for (n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 = $\Theta(n^2)$ comparisons. Each of these scans requires one swap for n - 1 elements (the final element is already in place). Thus, the comparisons dominate the running time, which is $\Theta(n^2)$.

Among simple average-case $\Theta(n^2)$ algorithms, selection sort always outperforms bubble sort, but is generally outperformed by insertion sort.

Linear Selection Sort

```
void selectionSort(int A[], int n)
{
  int i, j, min, imin;
  for (i = 0; i < n-1; i++)
  {
    min = A[i];
    imin = I;
    for (j = i+1; j < n; j++)
    {
      if (A[j] < min)
       {
        min = A[j];
        imin = j;
       }
     }
    A[imin] = A[i];
    A[i] = min;
  }
}
```

Selection and Change Sort

Selection and Change sort is a simple sorting algorithm that is more like Linear Selection sort.

The algorithm works as follows:

- **Fix the first position as minimum**
- **Check for a smaller current value**
- **Every time a smaller value will be found, it will be changed (swapped) with fixed minimum from first position.**
- **When all the elements were processed repeat the steps above for remainder of the list (starting with the next position)**

Selection and Change

```
void selectionChange(int A[ ], int n)
{
  int i, k. t;
  for (i = 0; i < n-1; i++)
  {
    for (k = i+1; k < n; k++)
    {
      if (A[k] < A[i])
       {
        t = A[k];
        A[k] = A[i];
        A[i] = t;
       }
     }
   }
}
```