Basic Comparison Sort in Data Structures

Dr.Haitham A. El-Ghareeb

Information Systems Department
Faculty of Computers and Information Sciences
Mansoura University

helghareeb@gmail.com

November 11, 2012

Sorting

Sorting is any process of arranging items in some sequence and/or in different sets, and accordingly, it has two common, yet distinct meanings:

- Ordering: arranging items of the same kind, class, nature, etc. in some ordered sequence,
- Categorizing: grouping and labeling items with similar properties together (by sorts).

Order Theory

Order theory is a branch of mathematics which investigates our intuitive notion of order using binary relations. It provides a formal framework for describing statements such as "this is less than that" or "this precedes that".

Sorting Information / Data

Sorting n-tuples (depending on context also called e.g. records consisting of fields) can be done based on one or more of its components. More generally objects can be sorted based on a property. Such a component or property is called a sort key. For example, the items are books, the sort key is the title, subject or author, and the order is alphabetical.

Weak Order

If the sort key values are totally ordered, the sort key defines a weak order of the items: items with the same sort key are equivalent with respect to sorting. If different items have different sort key values then this defines a unique order of the items.

Standard Order

A standard order is often called ascending (corresponding to the fact that the standard order of numbers is ascending, i.e. A to Z, 0 to 9), the reverse order descending (Z to A, 9 to 0). For dates/times ascending means that earlier values precede later ones e.g. 1/1/2012 will sort ahead of 1/1/2013.

Sorting in Computer Science

- Sorting is one of the most extensively researched subjects because of the need to speed up the operation on thousands or millions of records during a search operation. The main purpose of sorting information is to optimise its usefulness for specific tasks.
- Most common sorting purposes are Name, by Location and by Time (these are actually special cases of category and hierarchy). Together these give the acronym LATCH (Location, Alphabetical, Time, Category, Hierarchy) and can be used to describe just about every type of ordered information.

Opposite of Sorting

The opposite of sorting, rearranging a sequence of items in a random or meaningless order, is called **Shuffling**.

Sorting Algorithm

- 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 (generalization of Alphabetical 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.

Sorting Algorithm (Cont.)

- More formally, the output must satisfy two conditions:
  - The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
  - 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.
- 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 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(n2). 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 usage (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 is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator) that determines which of two elements should occur first in the final sorted list.
- The only requirement is that the operator obey two of the properties of a total order:
  - if a b and b c then a c (transitivity)
  - for all a and b, either a b or b a (totalness or trichotomy).

  It is possible that both a b and b a; in this case either may come first in the sorted list. In a stable sort, the input order determines the sorted order in this case.

Comparison Sort Disadvantages

There are fundamental limits on the performance of comparison sorts. A comparison sort must have a lower bound of (n log n) comparison operations. This is a consequence of the limited information available through comparisons alone or, to put it differently, of the vague algebraic structure of totally ordered sets. In this sense, mergesort, heapsort, and introsort are asymptotically optimal in terms of the number of comparisons they must perform, although this metric neglects other operations. The non-comparison sorts above achieve O(n) performance by using operations other than comparisons, allowing them to sidestep this lower bound (assuming elements are constant-sized).

Comparison Sort Advantages

Comparison sort offers the notable advantage that control over the comparison function allows sorting of many different datatypes and fine control over how the list is sorted. For example, reversing the result of the comparison function allows the list to be sorted in reverse; and one can sort a list of tuples in lexicographic order by just creating a comparison function that compares each part in sequence.

Comparison Sort Examples

Some of the most well-known comparison sorts include:

- Quick sort
- Heap sort
- Merge sort
- Intro sort
- Insertion sort
- Selection sort
- Bubble sort
- Odd-even sort
- Cocktail sort
- Cycle sort
- Merge insertion (Ford-Johnson) sort
- Smoothsort
- Timsort

Non-Comparison Sorts

There are many integer sorting algorithms that are not comparison sorts; they include:

- Radix sort (examines individual bits of keys)
- Counting sort (indexes using key values)
- Bucket sort (examines bits of keys)

Stability

- Stable sorting algorithms maintain the relative order of records with equal keys.
- A key is that portion of the record which is the basis for the sort; it may or may not include all of the record.
- 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.

Stability (Cont.)

- However, assume that the following pairs of numbers are to be sorted by their first component:
  - (4, 2) (3, 7) (3, 1) (5, 6)
  - (3, 7) (3, 1) (4, 2) (5, 6) (order maintained)
  - (3, 1) (3, 7) (4, 2) (5, 6) (order changed)
- 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.

Bubble Sort

- Simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- 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.
- Although the algorithm is simple, most other algorithms are more efficient for sorting large lists.

Bubble Sort (Cont.)

Graphical Illustration

An example on bubble sort. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.
`http://upload.wikimedia.org/wikipedia/commons/c/c8/Bubble-sort-example-300px.gif`

Bubble Sort (Step-by-step example)

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

- First Pass:
  - (**5** **1** 4 2 8) (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.
  - (1 **5** **4** 2 8) (1 4 5 2 8), Swap since $5 > 4$
  - (1 4 **5** **2** 8) (1 4 2 5 8), Swap since $5 > 2$
  - (1 4 2 **5** **8**) (1 4 2 5 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.
- Second Pass:
  - (**1** **4** 2 5 8) (1 4 2 5 8)
  - (1 **4** **2** 5 8) (1 2 4 5 8), Swap since $4 > 2$
  - (1 2 **4** **5** 8) (1 2 4 5 8)
  - (1 2 4 **5** **8**) (1 2 4 5 8)

Bubble Sort (Step-by-step example)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

- Third Pass:
  - (**1** **2** 4 5 8) (1 2 4 5 8)
  - (1 **2** **4** 5 8) (1 2 4 5 8)
  - (1 2 **4** **5** 8) (1 2 4 5 8)
  - (1 2 4 **5** **8**) (1 2 4 5 8)

Pseudocode implementation

```
procedure bubbleSort( A : list of sortable items )
   repeat
     swapped = false
     for i = 1 to length(A) - 1 inclusive do:
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

Bubble Sort Implementation in C#

```
public void BubbleSort( ) {
int temp;
for(int outer = upper; outer >= 1; outer--) {
for(int inner = 0; inner <= outer-1;inner++)
if ((int)arr[inner] > arr[inner+1]) {
          temp = arr[inner];
          arr[inner] = arr[inner+1];
          arr[inner+1] = temp;
}
}
}
```

Bubble Sort Properties

- Stable
- O(1) extra space
- O(n2) comparisons and swaps
- Adaptive: O(n) when nearly sorted

Selection Sort

- Selection sort is a sorting algorithm, specifically an in-place comparison sort.
- It has O(n2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.
- Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

Selection Sort in Steps

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 the remainder of the list (starting at the second position and advancing each time)

Effectively, the list is divided into two parts: the sublist of items already sorted, which is built 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.

Selection Sort Algorithm

```
for i = 1:n,
    k = i
    for j = i+1:n, if a[j] < a[k], k = j
    swap a[i,k]
end
```

Graphical Illustration

http://upload.wikimedia.org/wikipedia/commons/9/94/Selection-Sort-Animation.gif

Selection Sort Step by Step

- 64 25 12 22 11
- 11 25 12 22 64
- 11 12 25 22 64
- 11 12 22 25 64
- 11 12 22 25 64

Selection Sort Implementation

```
public void SelectionSort( ) {
int min, temp;
for(int outer = 0; outer <= upper; outer++) {
      min = outer;
      for(int inner = outer + 1; inner <= upper; inner++)
        if (arr[inner] < arr[min])
          min = inner;
      temp = arr[outer];
      arr[outer] = arr[min];
      arr[min] = temp;
} }
```

Selection Sort Properties

- Not stable
- O(1) extra space
- (n2) comparisons
- (n) swaps
- Not adaptive

Insertion Sort

When humans manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort.

Insertion Sort in Words

The Insertion sort has two loops. The outer loop moves element by element through the array whereas the inner loop compares the element chosen in the outer loop to the element next to it in the array. If the element selected by the outer loop is less than the element selected by the inner loop, array elements are shifted over to the right to make room for the inner loop element.

Graphical Illustration

http://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

Insertion Sort Step by Step

To Sort the sequence 3, 7, 4, 9, 5, 2, 6, 1 we need 8 steps. In each step, the item under consideration is in Bold.

- **3** 7 4 9 5 2 6 1
- 3 **7** 4 9 5 2 6 1
- 3 7 **4** 9 5 2 6 1
- 3 4 7 **9** 5 2 6 1
- 3 4 7 9 **5** 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

Insertion Sort Implementation in C#

```
public void InsertionSort() {
int inner, temp;
for(int outer = 1; outer <= upper; outer++) {
temp = arr[outer];
inner = outer;
while(inner > 0 && arr[inner-1] >= temp) {
         arr[inner] = arr[inner-1];
 inner -= 1;
}
       arr[inner] = temp;
} }
```

Insertion Sort Properties

- Stable
- O(1) extra space
- O(n2) comparisons and swaps
- Adaptive: O(n) time when nearly sorted
- Very low overhead