

Many computer applications involve sorting the items in a list into some specified order.

For example, we have seen that a list may be searched more efficiently if it is sorted.

To sort a group of items, the following relationships must be clearly defined over the items to be sorted:

a	<	b
a	>	b
a	=	b

Ascending order: smallest ... largest

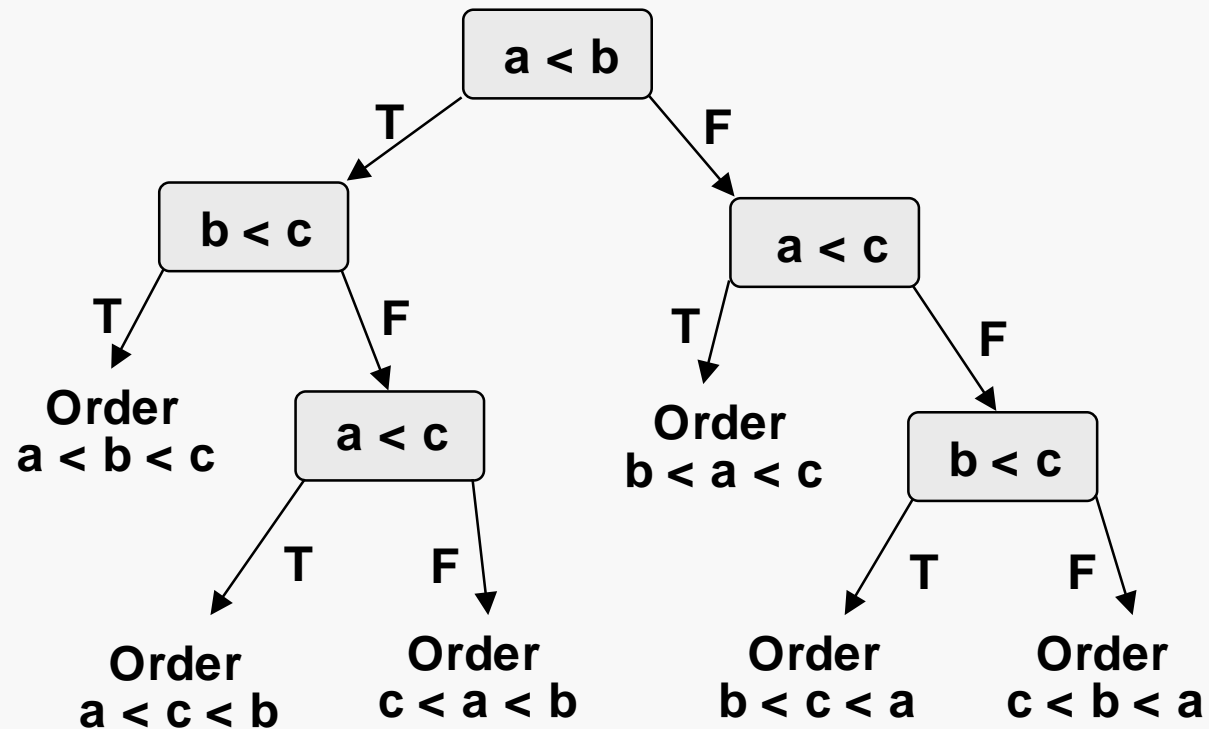
Descending order: largest ... smallest

When designing or choosing an algorithm for sorting, one goal is to minimize the amount of work necessary to sort the list of items.

Generally the amount of work is measured by the number of comparisons of list elements and/or the number of swaps of list elements that are performed.

To sort three items (a, b, c), how many comparisons must be made?

Naively we may consider the question by drawing a diagram representing the decisions that must be made to arrive at a solution:



The decision tree above assumes that the three items (a, b, c), are unique.

One of the simplest sorting algorithms proceeds by walking down the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is continued until the list is sorted.

More formally:

1. Initialize the size of the list to be sorted to be the actual size of the list.
2. Loop through the list until no element needs to be exchanged with another to reach its correct position.
 - 2.1 Loop (i) from 0 to size of the list to be sorted - 2.
 - 2.1.1 Compare the i^{th} and $(i + 1)^{\text{st}}$ elements in the unsorted list.
 - 2.1.2 Swap the i^{th} and $(i + 1)^{\text{st}}$ elements if not in order (ascending or descending as desired).
 - 2.2 Decrease the size of the list to be sorted by 1.

Each pass "bubbles" the largest element in the unsorted part of the list to its correct location.

A

13	7	43	5	3	19	2	23	29	??	??	??	??	??
----	---	----	---	---	----	---	----	----	----	----	----	----	----

Here is an ascending-order implementation of the bubblesort algorithm for integer arrays:

```
void BubbleSort(int List[] , int Size) {  
    int tempInt;    // temp variable for swapping list elems  
    for (int Stop = Size - 1; Stop > 0; Stop--) {  
        for (int Check = 0; Check < Stop; Check++) { // make a pass  
            if (List[Check] > List[Check + 1]) { // compare elems  
                tempInt          = List[Check];    // swap if in the  
                List[Check]      = List[Check + 1]; // wrong order  
                List[Check + 1] = tempInt;  
            }  
        }  
    }  
}
```

Bubblesort compares and swaps adjacent elements; simple but not very efficient.

Efficiency note: the outer loop could be modified to exit if the list is already sorted.

Bubble Sort Trace

Trace the given implementation on the array below. Try to keep track of how many comparisons and swaps are performed.

A	13	7	43	5	3	19	2	23	29	??	??	??	??	??
---	----	---	----	---	---	----	---	----	----	----	----	----	----	----

	0	1	2	3	4	5	6	7	8	
A	13	7	43	5	3	19	2	23	29	Original array
										Pass 1
										Pass 2
										Pass 3
										Pass 4
										Pass 5
										Pass 6
										Pass 7
										Pass 8
										Pass 9

Another simple sorting algorithm proceeds by walking down the list, and finding the smallest (or largest) element, and then swapping it to the beginning of the unsorted part of the list. The process is continued until the list is sorted.

More formally:

1. Loop (i) from 0 to the (number of elements to be sorted - 2)
 - 1.1 Assume the smallest remaining item is at the i^{th} position, call this location smallest.
 - 1.2 Loop (j) through the remainder of the list to be sorted ($i+1 .. \text{size}-1$).
 - 1.2.1 Compare the j^{th} & smallest elements in the unsorted list.
 - 1.2.2 If the j^{th} element is $<$ the smallest element then
reset the location of the smallest to the j^{th} location.
 - 1.3 Move the smallest element to the head of the unsorted list,
(i.e. swap the i^{th} and smallest elements).

After sorting all but 1 element the remaining element must be in its correct position.

Here is an ascending-order implementation of the selection sort algorithm for integer arrays:

```
void SelectionSort(int List[], int Size) {  
  
    int Begin, SmallSoFar, Check;  
    void Swap(int& Elem1, int& Elem2);           // see previous slide  
  
    for (Begin = 0; Begin < Size - 1; Begin++) {  
  
        SmallSoFar = Begin;                       // set head of tail  
  
        for (Check = Begin + 1; Check < Size; Check++) { // scan current tail  
  
            if (List[Check] < List[SmallSoFar])  
                SmallSoFar = Check;  
        }  
  
        Swap(List[Begin], List[SmallSoFar]); // put smallest elem at front  
                                            // of current tail  
    }  
}  
  
void Swap(int& Elem1, int& Elem2) {  
    int tempInt;  
    tempInt = Elem1;  
    Elem1 = Elem2;  
    Elem2 = tempInt;  
}
```

Selection Sort Trace

Trace the given implementation on the array below. Try to keep track of how many comparisons and swaps are performed.

A

13	7	43	5	3	19	2	23	29	??	??	??	??	??
----	---	----	---	---	----	---	----	----	----	----	----	----	----

	0	1	2	3	4	5	6	7	8	
A	13	7	43	5	3	19	2	23	29	Original array
										Pass 1
										Pass 2
										Pass 3
										Pass 4
										Pass 5
										Pass 6
										Pass 7
										Pass 8
										Pass 9

Here is an ascending-order implementation of the selection sort algorithm for integer arrays:

```
// insert code from running example here
```

Suppose the array to be sorted contains N elements. The cost of sorting may be measured by the number of array elements that must be compared to each other, or by the number of times two array elements must be swapped.

Using those measures (approximately):

	Comparisons		Swaps	
	Worst	Average	Worst	Average
Bubble Sort	N^2	N^2	N^2	N^2
Selection Sort	N^2	N^2	$N-1$	$N-1$

In some cases, comparisons are more expensive than swaps; in other cases, swaps may be more expensive than comparisons. In any case, there is no cost advantage to using Bubble Sort.

There are other, more complex, sorting algorithms whose costs are on the order of $N \log_2 N$ — those are introduced in CS 1704.