Sorting

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.

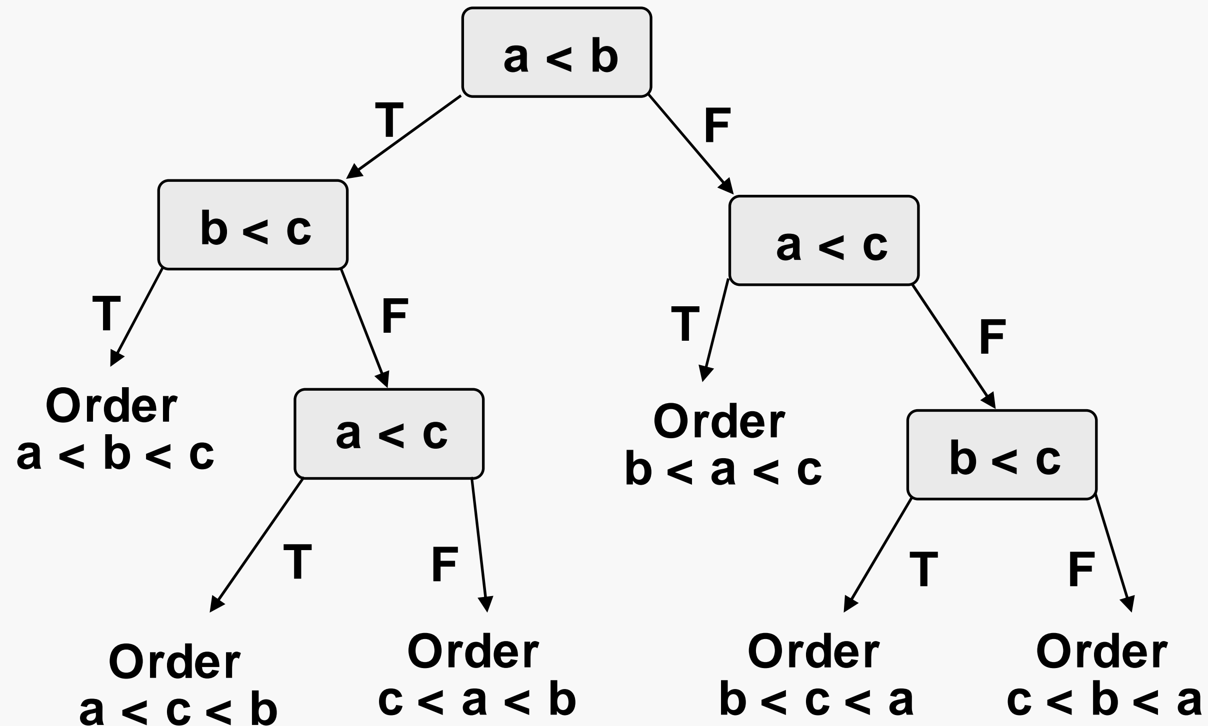
Generally the amount of work is measured by the number of comparisons of list elements and/or the number of swaps of list elements that are performed.

©1995-2002 Barnette ND & McQuain WD

Decision Trees

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.

©1995-2002 Barnette ND & McQuain WD

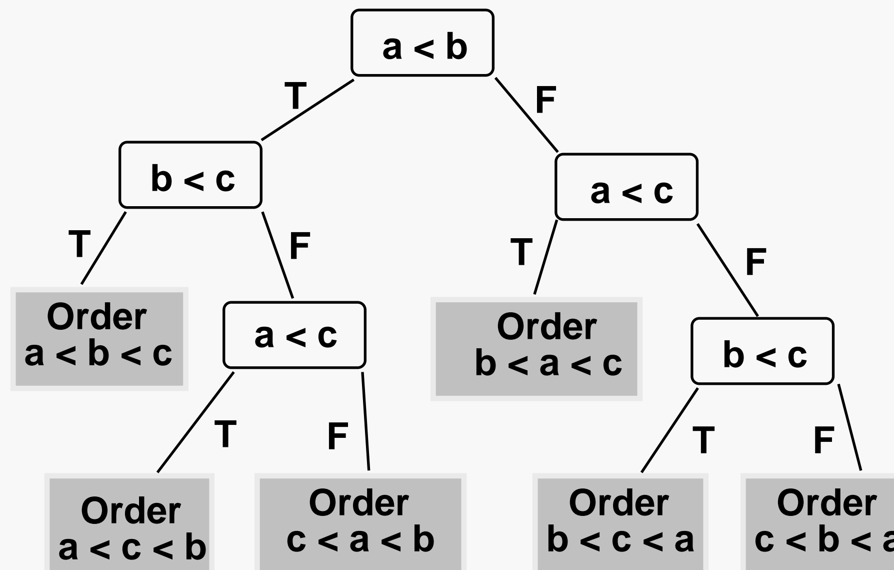
Comparison-Based Sorting

3 – Any of the 3 elements (a, b, c) could be first in the final order. Thus there are 3 distinct ways the final sorted order could start.

2 – After choosing the first element, there are two possible selections for the next sorted element.

1 – After choosing the first two elements there is only 1 remaining selection for the last element.

– Therefore selecting the first element one of 3 ways, the second element one of 2 ways and the last element 1 way, there are 6 possible final sorted orderings = **3 * 2 * 1 = 3!**



©1995-2002 Barnette ND & McQuain WD

Bubble Sort Algorithm

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 ith and $(i + 1)^{st}$ elements in the unsorted list.
   - 2.1.2 Swap the ith and $(i + 1)^{st}$ elements if not in order (ascending or descending as desired).

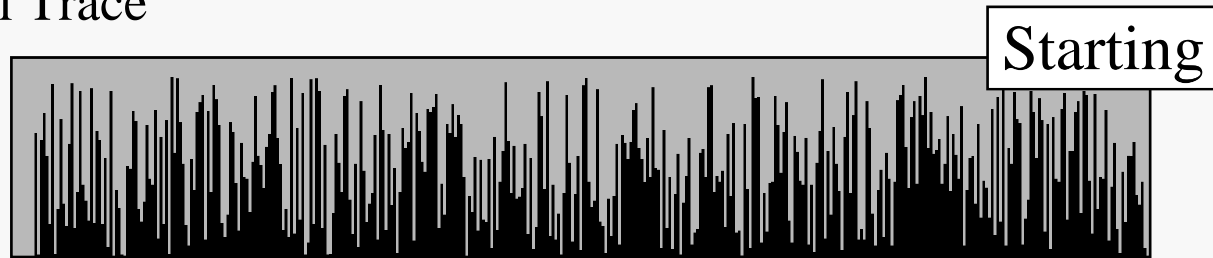
   2.2 Decrease the size of the list to be sorted by 1.

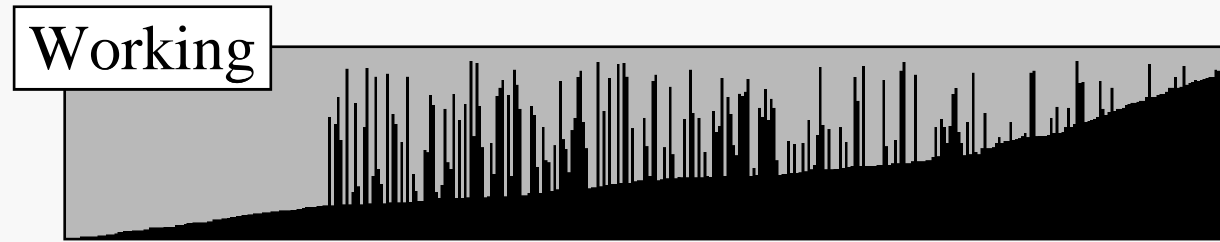
Each pass "bubbles" the largest element in the unsorted part of the list to its correct location.

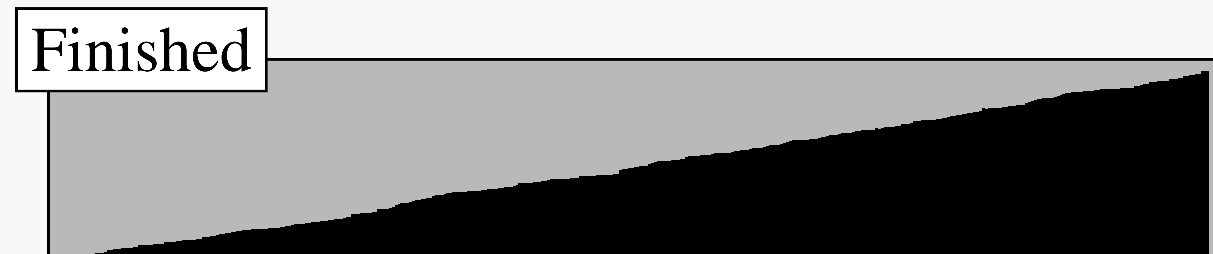
A													
13	7	43	5	3	19	2	23	29	??	??	??	??	??

©1995-2002 Barnette ND & McQuain WD

- Graphical Trace







©1995-2002 Barnette ND & McQuain WD

Bubble Sort Implementation

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.

©1995-2002 Barnette ND & McQuain WD

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

©1995-2002 Barnette ND & McQuain WD

Selection Sort Algorithm

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 ith position, call this location smallest.

    1.2 Loop (j) through the remainder of the list to be sorted (i+1 .. size-1).

    1.2.1 Compare the jth & smallest elements in the unsorted list.

    1.2.2 If the jth element is < the smallest element then
    reset the location of the smallest to the jth location.

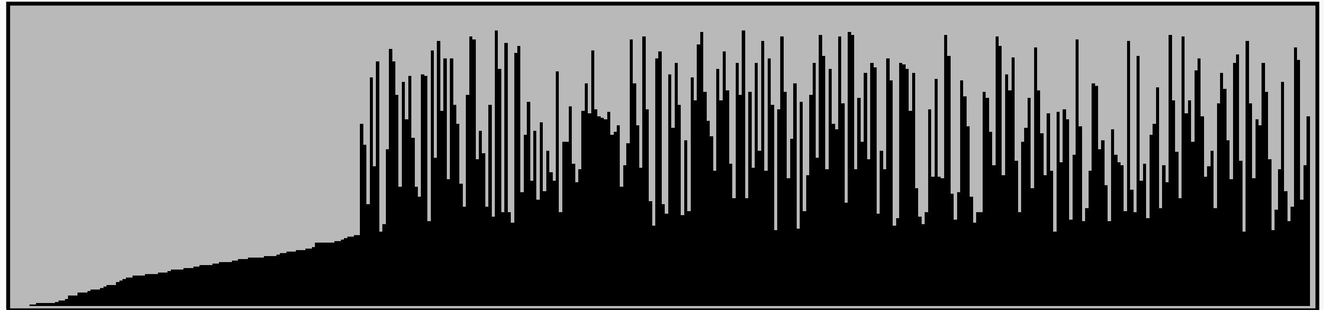
    1.3 Move the smallest element to the head of the unsorted list,
    (i.e. swap the ith and smallest elements).

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

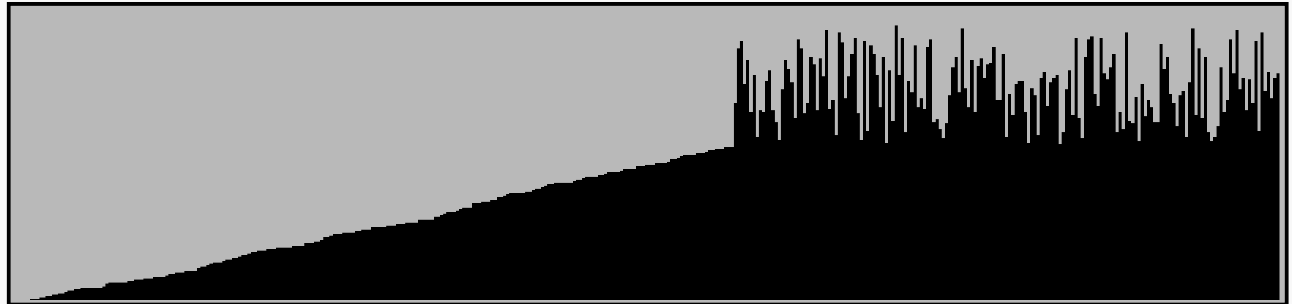
©1995-2002 Barnette ND & McQuain WD

- Graphical Trace

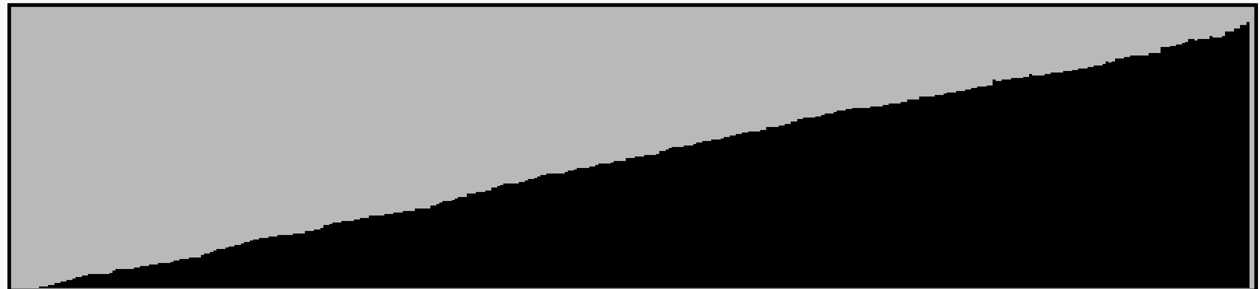
Working



Working



Finished



©1995-2002 Barnette ND & McQuain WD

Selection Sort Implementation

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;
}
```


©1995-2002 Barnette ND & McQuain WD

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

©1995-2002 Barnette ND & McQuain WD

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

```
void sortByOrigin(Trip dB[], int numTrips) {

   Trip tempTrip;       // temp variable for swapping list elems

   for (int Stop = numTrips - 1; Stop > 0; Stop--) {

      for (int Check = 0; Check < Stop; Check++) { // make a pass

         // compare Origin fields of array elements
         if (dB[Check].Origin > dB[Check + 1].Origin) {

            tempTrip       = dB[Check];      // swap if in the
            dB[Check]      = dB[Check + 1]; //   wrong order
            dB[Check + 1] = tempTrip;
         }
      }
   }
}
```

 ©1995-2002 Barnette ND & McQuain WD

Cost of Sorting

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 $N\log_2 N$ — Quicksort.

©1995-2002 Barnette ND & McQuain WD

Algorithm

- Select an item in the array as the pivot key.
- Divide the array into two partitions: a left partition containing elements < the pivot key and a right partition containing elements ≥ the pivot key.

Trace

Start with i and j pointing to the first & last elements, respectively.

Select the pivot (3): [3 1 4 1 5 9 2 6 5 8]

L R

Swap the end elements, then move L, R inwards.

[8 1 4 1 5 9 2 6 5 3]

L R

Swap, and repeat: [2 1 4 1 5 9 8 6 5 3]

L R

Swap, and repeat: [2 1 1 | 4 5 9 8 6 5 3]

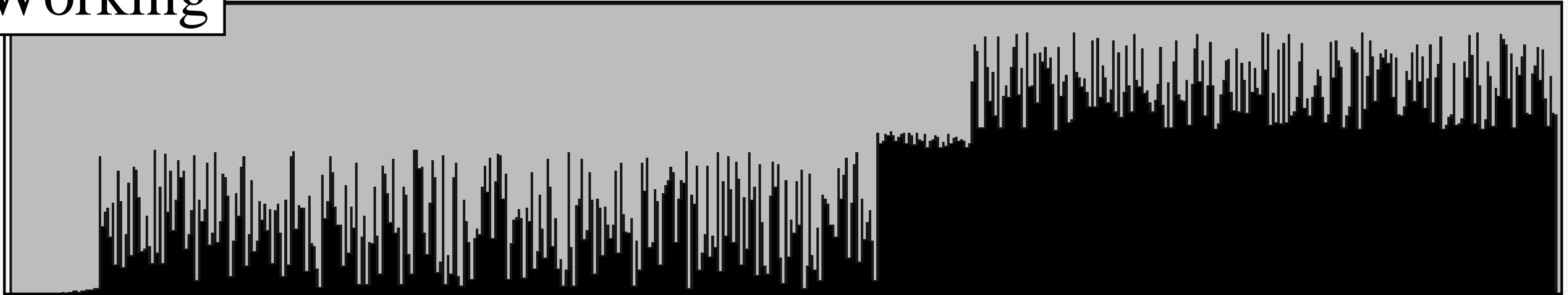
R L

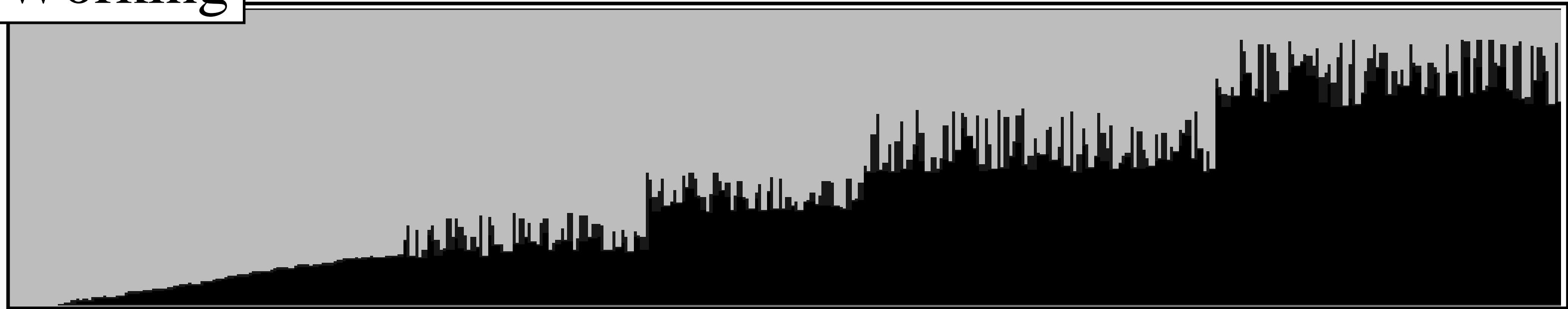
Algorithm

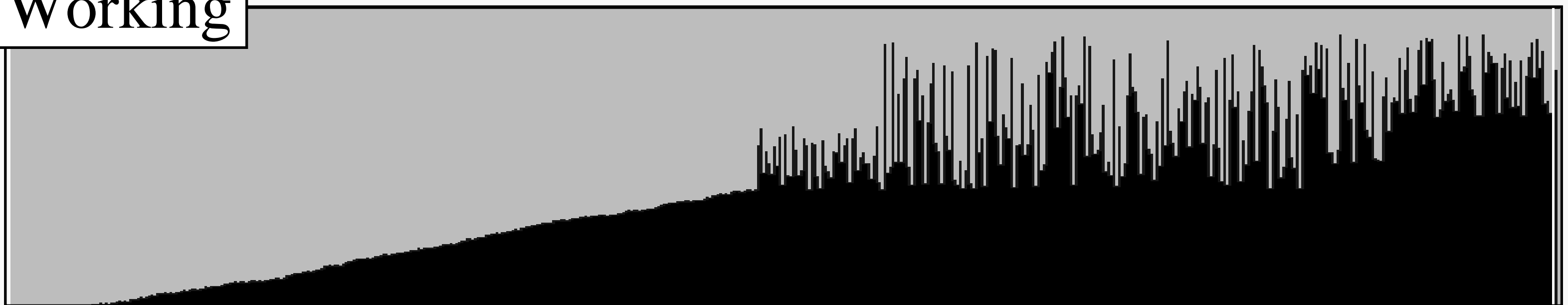
- Partition the left & right partitions, continuing the partitioning until partitions are of size 1 or less.

©1995-2002 Barnette ND & McQuain WD

- Graphical Trace







©1995-2002 Barnette ND & McQuain WD

FindPivot

- Partitioning test **requires** at least 1 key with a value < that of the pivot, and 1 value ≥ to that of the pivot, to execute correctly.
- Therefore, pick the greater of the first two distinct values (if any).

Improving FindPivot

- Try and pick a pivot such that the list is split into equal size sublists, (a speedup that should cut the number of partition steps to about 2/3 that of picking the first element for the pivot).
  - † Choose the middle (median) of the first 3 elements.
  - † Pick k elements at random from the list, sort them & use the median.
- There is a trade-off between reduced number of partitions & time to pick the pivot as k grows.

©1995-2002 Barnette ND & McQuain WD