

Sorting and Searching

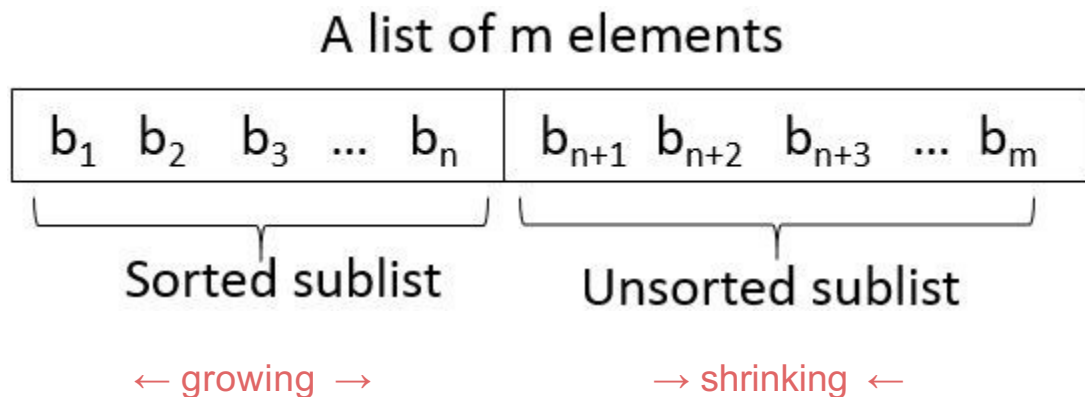
ACS-1904 LECTURE 10

Sorting & searching

- Sorting:
 - Putting lists into sequence
- Searching:
 - Finding things in a list

Sorting

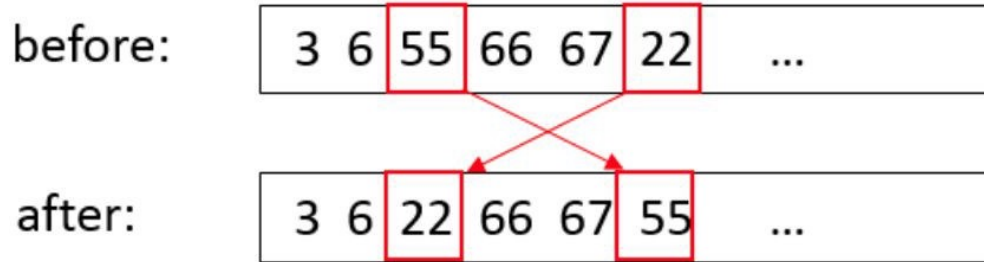
- many algorithms conceptually consider a list that comprises a sorted sublist and an unsorted sublist



Sorting - list operations

Swapping a pair of elements

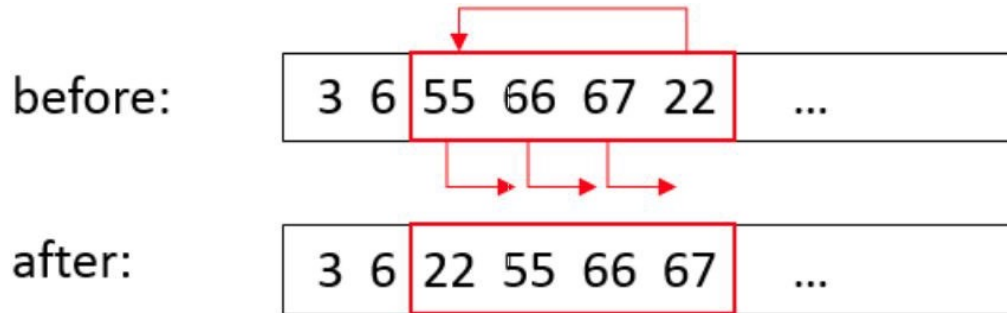
- e.g. 3rd and 6th elements swap positions



Sorting - list operations

Shifting an element into position

- e.g. 6th element moves into 3rd position; others shift right



Sorting algorithms

- Selection sort
- Insertion sort
- Bubble sort
- Quicksort

... many others

e.g. Merge sort, heap sort, bucket sort, radix sort...

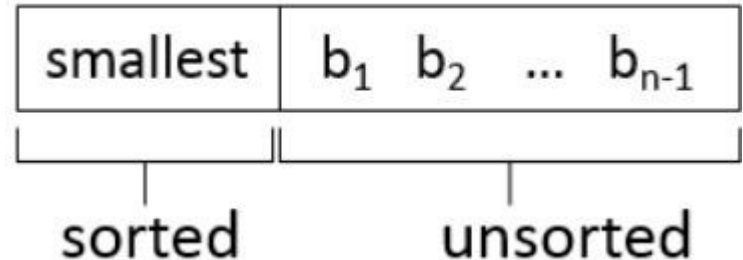
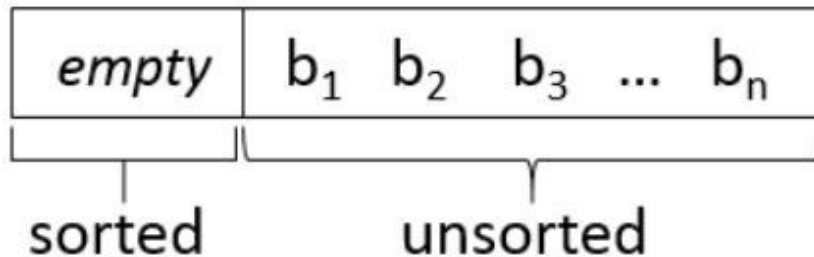
→ ACS-2947

Selection sort

During a series of passes over the unsorted sublist

- the sorted sublist grows larger and
- the unsorted sublist shrinks to nothing

Basic idea: find smallest & move it to beginning of unsorted



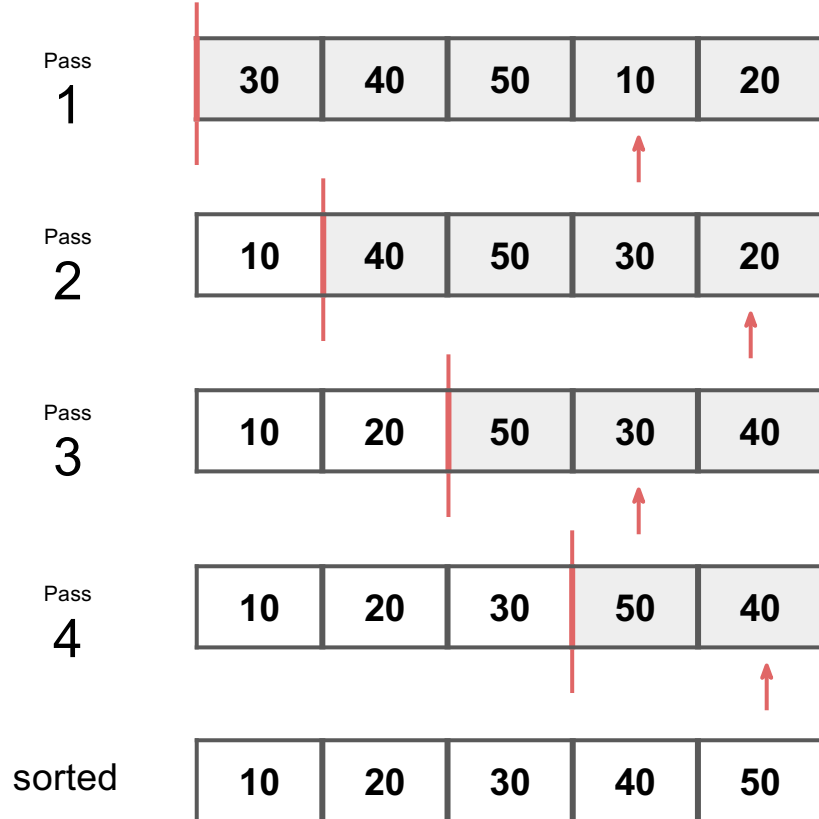
Selection sort

1. Sorted sublist \leftarrow empty.
2. Unsorted sublist \leftarrow list to be sorted.
3. Iteration:
 - a. find the smallest entry (X) in the unsorted sublist
 - b. swap X and the first entry of the unsorted sublist
 - c. sorted sublist grows by 1 element & unsorted sublist shrinks by 1 element

Selection sort

Each pass:

- find smallest of unsorted
- swap with first of unsorted

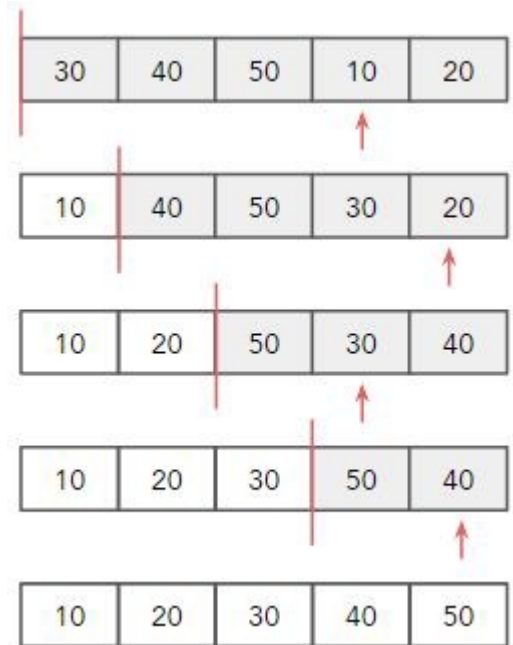


5 elements → 4 passes

Selection sort

SelectionSort.java

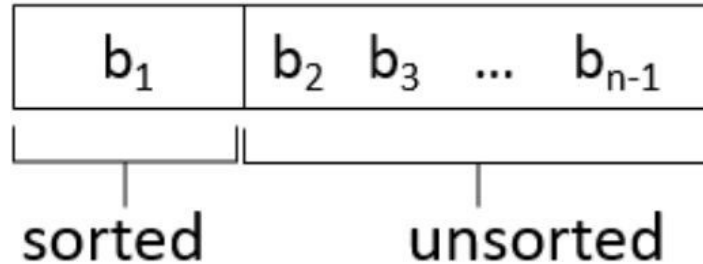
```
public static void selectionSort(int[] a){
    int n = a.length;
    for (int i=0; i<n-1; i++) {
        int iSmallest = i;
        for (int j=i+1; j<n; j++) {
            if (a[j] < a[iSmallest]) {
                iSmallest = j;
            }
        }
        if(iSmallest != i) {
            int temp = a[i];
            a[i] = a[iSmallest];
            a[iSmallest] = temp;
        }
    }
}
```



Insertion sort

Initially first element is part of sorted sublist

Basic idea: move first of the unsorted into proper position of sorted elements ... may require some shifting



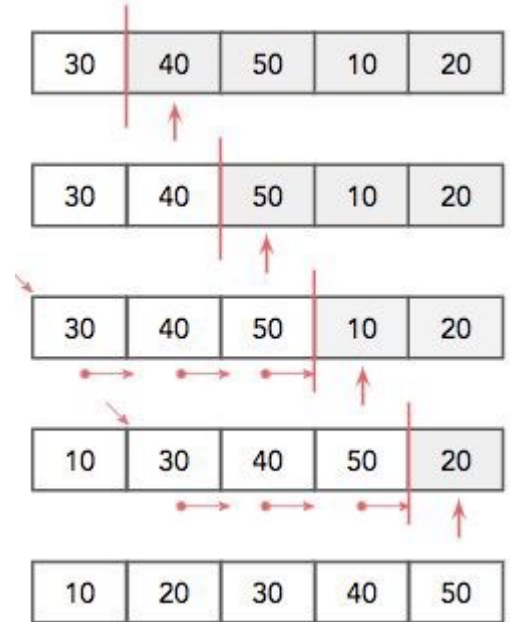
Insertion sort

1. Sorted sublist \leftarrow first element of list.
2. Unsorted sublist \leftarrow remaining $n-1$ elements.
3. Iteration: Repeat until the unsorted sublist is empty.
 - a. Let X be the first element in the unsorted list
 - b. Shift entries that are greater than X to the right making room
 - c. for X in its proper position in the sorted sublist.
 - d. X is now part of the sorted sublist and the unsorted sublist is 1 element shorter

Insertion sort

InsertionSort.java

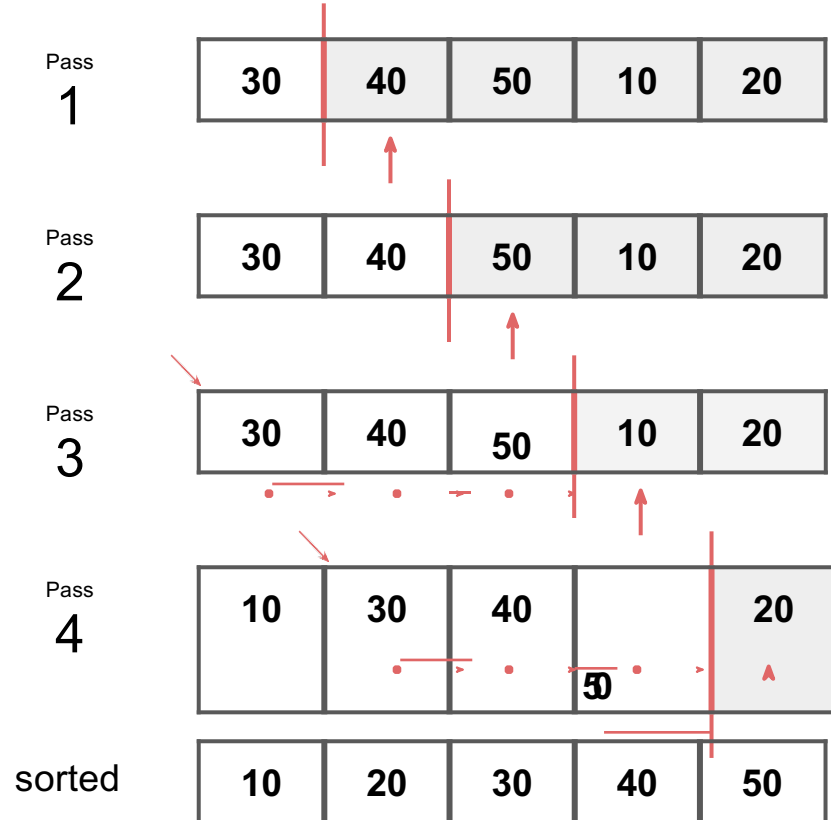
```
public static void insertionSort(int[] a){
    int n = a.length;
    for (int i=1; i<n; i++) {
        int x = a[i];
        int j=i-1;
        while (j>=0 && a[j]>x){
            a[j+1] = a[j];
            j--;
        }
        a[++j]=x;
    }
}
```



Insertion sort

Each pass:

- move first unsorted element to its proper position in sorted sublist
- shift elements as is necessary



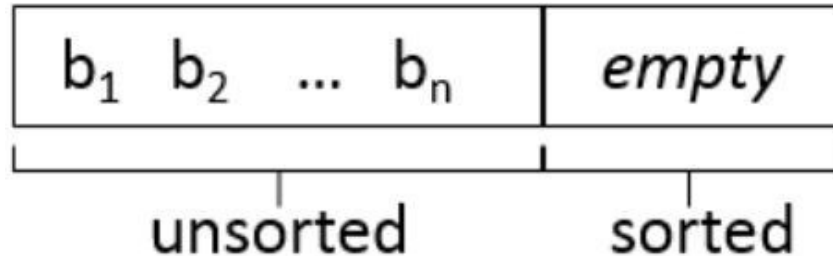
5 elements → 4 passes

Bubble sort

Larger elements bubble to the right

- largest element moved to end of unsorted
- sorted sublist grows by 1, unsorted shrinks by 1

Initially:



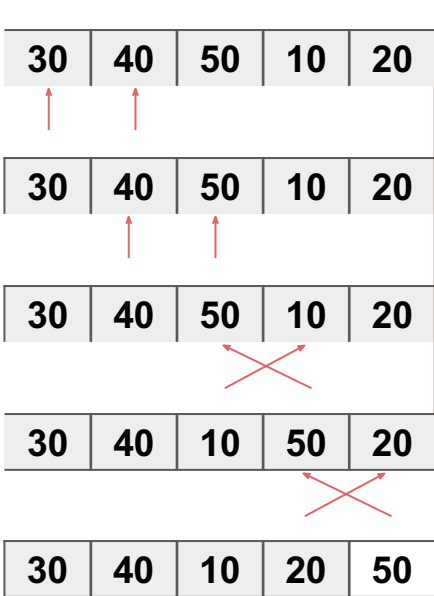
Bubble sort

1. Unsorted sublist \leftarrow the original list.
2. Sorted sublist \leftarrow an empty list.
3. Iteration:
 - a. Scanning forward from the beginning of the unsorted list, consider the elements pairwise: a_1 and a_2 , then a_2 and a_3 , ...
For each pair where $a_i > a_{i+1}$, swap a_i and a_{i+1} .
After a scan the largest element has moved to the end of the unsorted list.
 - b. The unsorted sublist shrinks by 1 element; sorted sublist increases by 1 element.

Bubble sort

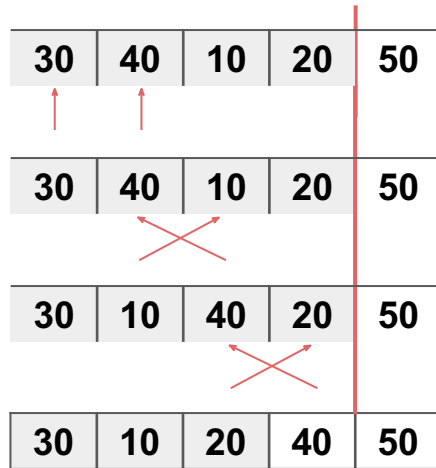
Each iteration: bubble larger elements to the right

Pass 1



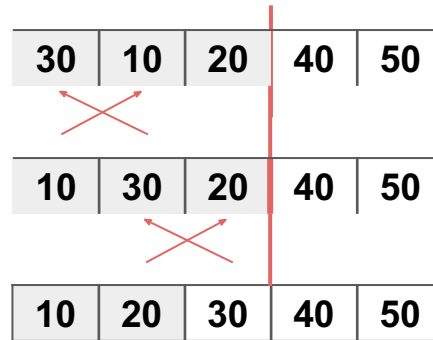
50 in its final position

Pass 2



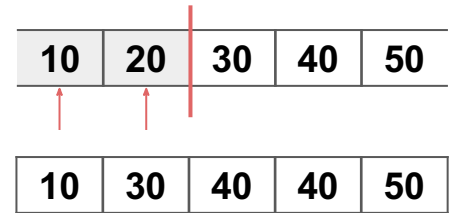
40 & 50 in final positions

Pass 3



30, 40, 50 in final positions

Pass 4

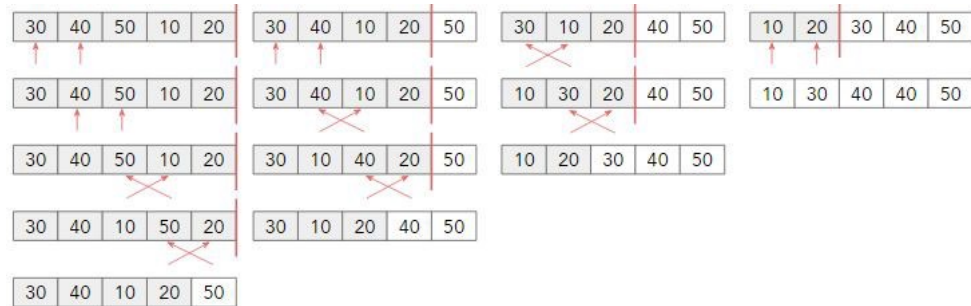


sorted

Bubble sort

BubbleSort.java

```
public static void bubbleSort(int[] a) {  
    int n = a.length;  
    for (int i=0; i<n-1; i++) {  
        for (int j=0; j<n-i-1; j++){  
            if (a[j] > a[j+1]) {  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```



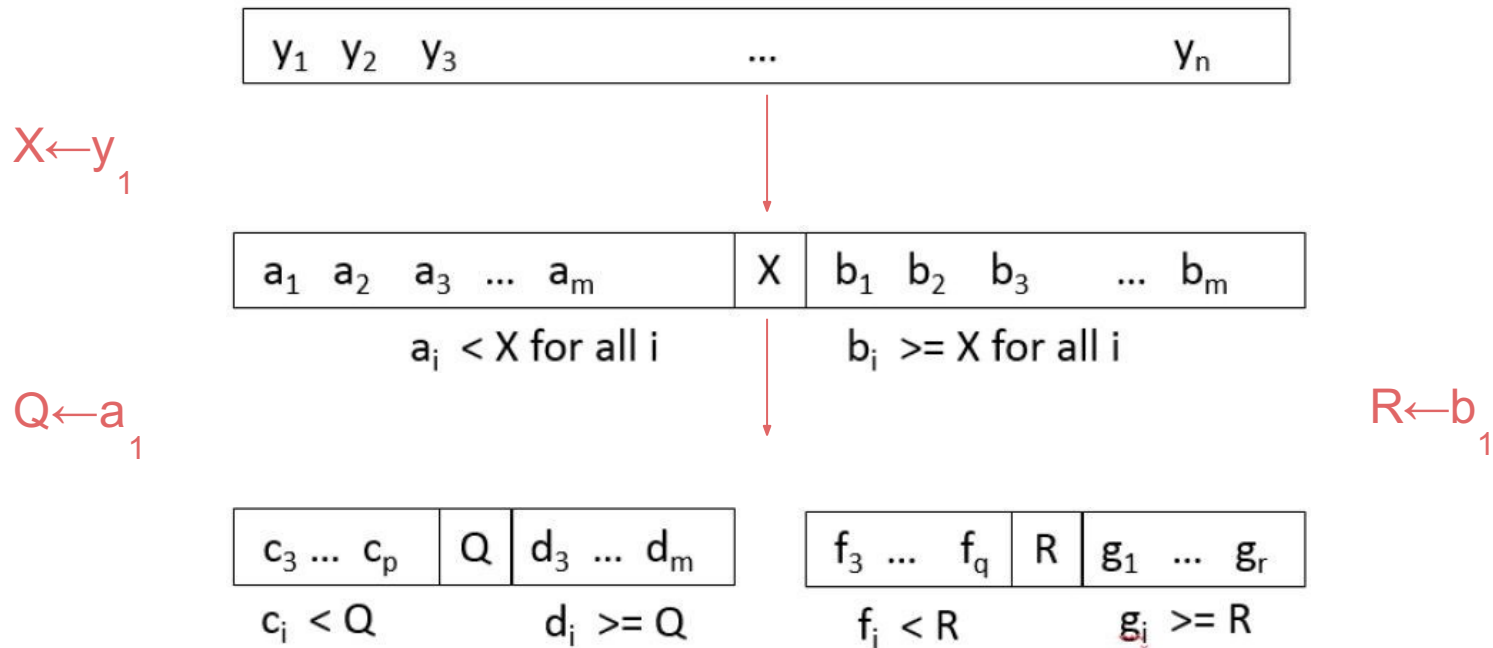
Quick sort

Different: Recursive

- Continuously subdivide a list into 2 sublists based on a ***pivot*** value
- Variations on how the pivot value is chosen
 - we will always choose the first element of a sublist as the pivot
- Left sublist has elements with values $<$ pivot
- Right sublist has elements with values \geq pivot

Quick sort

A list of n elements



Quick sort

QuickSort (list):

1. If list of length ≤ 1 return.
2. Choose a pivot value X .
3. Rearrange the elements of list:
 Left sublist L with elements $< X$;
 Right sublist R with elements $\geq X$.
4. **QuickSort** (L)
 QuickSort (R)

Quick sort

Unsorted → [6, 5, 9, 0, 11, 8, 4, 10, 2, 1, 12, 7, 3]

Pivot=6 [3, 5, 1, 0, 2, 4] [6] [10, 8, 11, 12, 7, 9]

Pivots 3, 10 [2, 0, 1] [3] [5, 4] [9, 8, 7] [10] [12, 11]

Pivots 2, 5, 9, 12 [1, 0] [2] [4][5] [7, 8] [9] [11][12]

Pivots 1, 7 [0] [1] [7] [8]

Final result sorted → [0][1][2][3][4][5][6][7][8][9][10][11][12]

```
public static void quicksort(int[] a, int start, int end){
    if (start >= end) return;
    int pivot = a[start];
    int left = start;
    int right = end;
    while (left < right){
        while (right > left && a[right]>=pivot){
            right--;
        }
        if(left==right) break;
        a[left]=a[right];
        left++;
        while (right > left && a[left] <= pivot){
            left++;
        }
        if(left == right) break;
        a[right]=a[left];
        right--;
    }
    a[right]=pivot;
    quicksort(a, start,    right-1);
    quicksort(a, right+1, end    );
}
```

Quicksort.java
QuickSortAlternate.java

Sorting Objects

Example

- Objects (ducks) must implement `Comparable`
i.e. `compareTo` method
- Array passed in to `sort` method is of type `Comparable`
- Object holding a temporary is of type `Comparable`

Searching

Unordered list?

Ordered list?

i.e. sorted

Searching - unordered list

To determine that a search value is not present must scan the whole array

Searching an **Unordered** List

Search(list, X):

1. Let i reference the start of the list
2. While $i \leq$ end of list
 - a. If $list_i$ equals X then return *found at position i*
 - b. Increment i to reference the next element of the list
3. Return *not found* *Searched to the end of list*

Searching – ordered list (Sequential Search)

Searching an **Ordered** List

SequentialSearch (list, X):

1. Let i reference the first element of the list.
2. While $i \leq$ end of list
 - a. If $list_i > X$ then return *notfound*
 - b. If $list_i$ equals X then return *found at position i*
 - c. Increment i to reference the next element of the list
3. Return *not found*

Don't always need to go to end of list to determine the search value is not there

Searching – ordered list (Binary Search)

BinarySearch.java

Searching an Ordered List

BinarySearch (list, X):

1. If the list L is empty then return *not found*
2. Determine the midpoint of the list
3. Determine the value at the midpoint of the list
4. if $X < \text{midpoint value}$ call BinarySearch (left sublist, X)
else if $X > \text{midpoint value}$ call BinarySearch (right sublist, X)
else if $X = \text{midpoint value}$ then return *found at position midpoint*