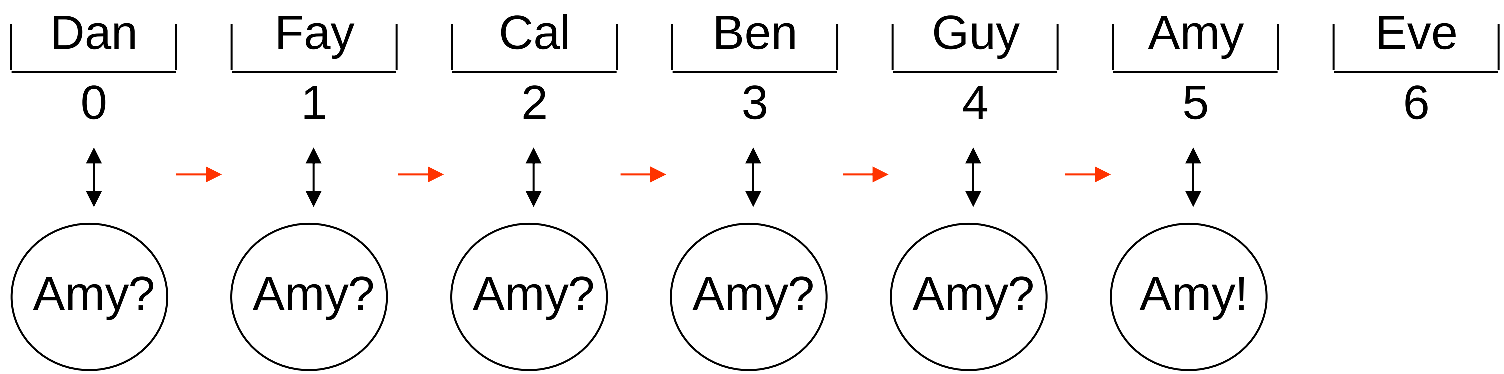
Searching and Sorting

Sequential Search

- Scans the list comparing the target value to each element.



Sequential Search (cont'd)

```
public int sequentialSearch(Object [ ] arr, Object value)
{
    for (int i = 0;  i < arr.length ;  i++)
    {
        if  (value.equals(arr [i]))
            return i;
    }
    return –1;
}
```

For primitive data types it is

if (value == arr [i])

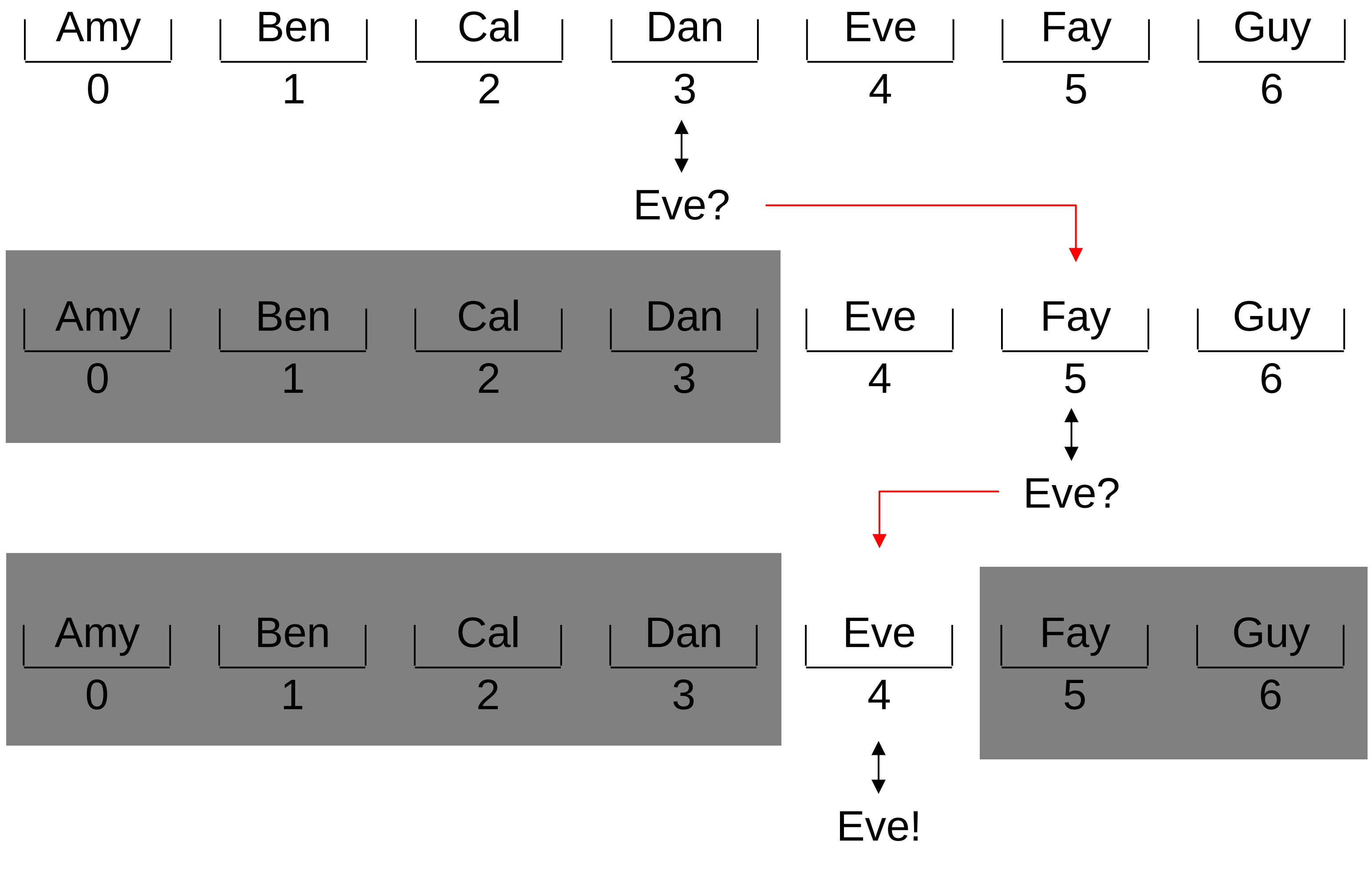
Sequential Search (cont'd)

- The average number of comparisons (assuming the target value is equal to one of the elements of the array, randomly chosen) is about *n* / 2 (where `n = arr.length`).
- Worst case: *n* comparisons.
- Also *n* comparisons are needed to establish that the target value is not in the array.
- We say that this is an *O*(*n*) (order of *n*) algorithm.

Binary Search

- The elements of the list must be arranged in ascending (or descending) order.
- The target value is always compared with the middle element of the remaining search range.
- We must have random access to the elements of the list (an array or ArrayList are OK).

Binary Search (cont'd)



Binary Search (cont'd)

- Recursive implementation:

```
public int binarySearch (int [ ] arr, int value, int left, int right)
{
   if (right < left)
      return –1;        // Not found

   int middle = (left + right) / 2;

   if (value == arr [middle] )
      return middle;

   else if (value < arr[middle])
      return binarySearch (arr, value, left, middle – 1);

   else     //  if ( value > arr[middle])
      return binarySearch (arr, value, middle + 1, right);
}
```

Binary Search (cont'd)

- Iterative implementation:

```
public int binarySearch (int [ ] arr, int value, int left, int right)
{
   while (left <= right)
   {
      int middle = (left + right) / 2;

      if ( value == arr [middle] )
         return middle;
      else if ( value < arr[middle] )
         right = middle - 1;
      else     //  if ( value > arr[middle] )
         left = middle + 1;
    }
    return –1;  // Not found
}
```

Binary Search (cont'd)

- A "divide and conquer" algorithm.
- Works very fast: only 20 comparisons are needed for an array of 1,000,000 elements; (30 comparisons can handle 1,000,000,000 elements; etc.).
- We say that this is an $O(\log n)$ algorithm.

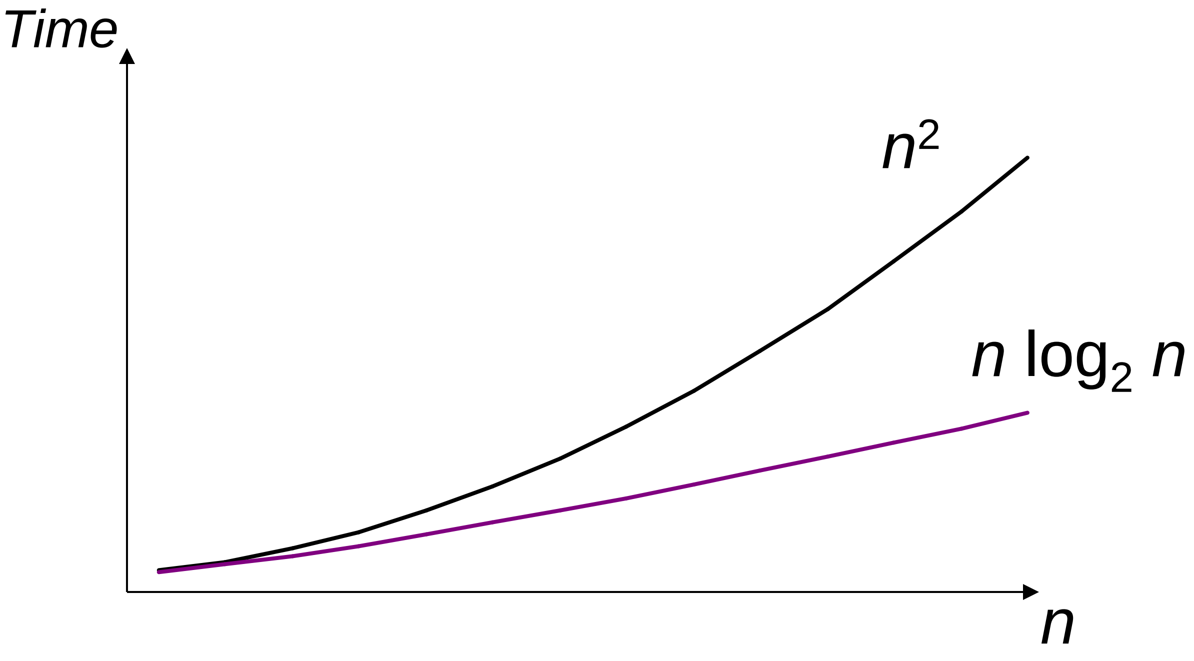
Sorting

- To sort means to rearrange the elements of a list in ascending or descending order.
- Examples of sorting applications:
  - a directory of files sorted by name or date
  - bank checks sorted by account #
  - addresses in a mailing list sorted by zip code
  - hits found by a search engine sorted by relevance
  - credit card transactions sorted by date

Sorting (cont'd)

- The algorithms discussed here are based on "honest" comparison of values stored in an array. No tricks.
- How fast can we sort an array of *n* elements?
  - If we compare each element to each other we need $n(n\text{-}1) / 2$ comparisons (that is, n^2 by the "order of magnitude.")
  - Faster "divide and conquer" sorting algorithms need approximately $n \cdot \log_2 n$ comparisons (much better).

Sorting (cont'd)



n	10	100	1000
n^2	100	10,000	**1,000,000**
$n \log_2 n$	35	700	**10,000**

Selection Sort

1. Select the max among the first *n* elements:

$$\underbrace{1 \quad \mathbf{13} \quad 8 \quad 5 \quad 2 \quad 1 \quad 3}_{n}$$

2. Swap it with the *n*-th element :

$$\underbrace{1 \quad 3 \quad 8 \quad 5 \quad 2 \quad 1 \quad \mathbf{13}}_{n}$$

3. Decrement *n* by 1 and repeat from Step 1 (while *n* > 1)

$$\underbrace{1 \quad 3 \quad 8 \quad 5 \quad 2 \quad 1}_{n} \quad 13$$

Selection Sort (cont’d)

- Iterative implementation:

```
public void selectionSort (double [ ] arr, int n)
{
    while  (n > 1)
    {
        int maxPos = 0;
        for (int k = 1; k < n; k++)
            if (arr [k] > arr [maxPos] )
                maxPos = k;
        double temp = arr [maxPos];
        arr [maxPos] = arr [n–1];
        arr [n–1] = temp;
        n––;
    }
}
```

swap **a[maxPos]** and **a[n-1]**

Selection Sort (cont'd)

- The total number of comparisons is always

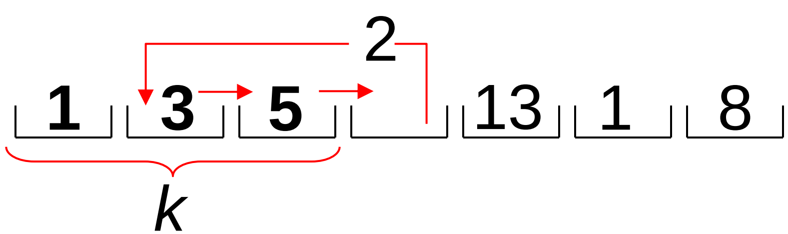
$$(n\text{-}1) + (n\text{-}2) + \ldots + 1 = n(n\text{-}1) / 2$$

- No average, best, or worst case — always the same.
- An $\boxed{O(n^2)}$ algorithm.

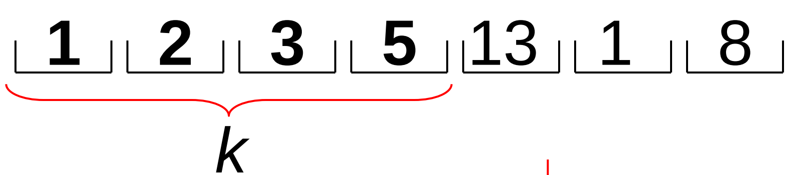
Insertion Sort

1. $k = 1$; keep the first k elements in order.

2. Take the (k+1)-th element and insert among the first k in the right place.



3. Increment k by 1; repeat from Step 2 (while $k < n$)



Insertion Sort (cont'd)

- Iterative implementation:

```
public void insertionSort (double [ ] arr, int n)
 {
   for (int k = 1 ; k < n;  k++)
   {
     double temp = arr [ k ];
     int i = k;
     while (i > 0  &&  arr [i-1]  >  temp)
     {
       arr [i] = arr [i - 1];
       i --;
     }
     arr [i] = temp;
   }
 }
```

shift to the right

Insertion Sort (cont'd)

- The average number of comparisons is roughly half of the number in Selection Sort.
- The best case is when the array is already sorted: takes only (n-1) comparisons.
- The worst case is $n(n-1) / 2$ when the array is sorted in reverse order.
- On average, an $\boxed{O(n^2)}$ algorithm.

Mergesort

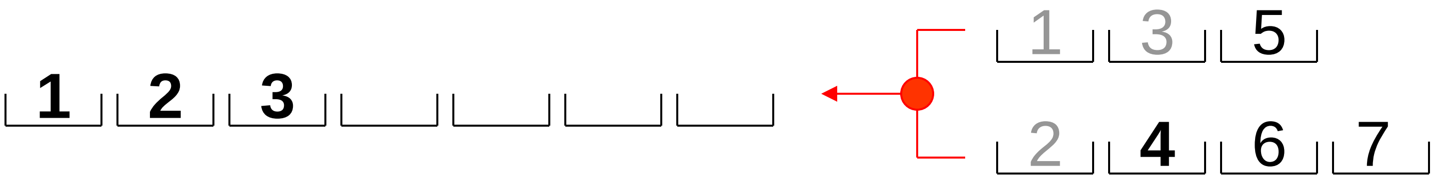
1. Split the array into two roughly equal “halves.”

5 1 3   2 4 7 6

2. Sort (recursively) each half using... Mergesort.

1 3 5   2 4 6 7

3. Merge the two sorted halves together.



The smaller value goes first

Mergesort (cont'd)

```
public void mergesort (double[ ] arr,
                                      int from,  int to)
{
   if  (from <= to)
       return;

   int middle = (from + to ) / 2;
   mergesort (arr, from, middle);
   mergesort (arr, middle + 1, to);

   if (arr [middle] > arr [middle + 1])
   {
      copy (arr, from, to, temp) ;
      merge (temp, from, middle, to, arr);
   }
}
```

Base case

Optional shortcut: "if not yet sorted"...

double[ ] temp is initialized outside the **mergesort** method

Mergesort (cont'd)

- Takes roughly $n \cdot \log_2 n$ comparisons.
- Without the shortcut, there is no best or worst case.
- With the optional shortcut, the best case is when the array is already sorted: takes only $(n-1)$ comparisons.
- An $\boxed{O(n \log n)}$ algorithm.

Quicksort

1. Pick one element, called "pivot"

5	1	6	2	4	7	3

2. Partition the array, so that all the elements to the left of pivot are ≤ pivot; all the elements to the right of pivot are ≥ pivot.

4	1	3	2	**5**	7	6

3. Sort recursively the left and the right segments using... Quicksort.

Quicksort (cont'd)

- Takes roughly $n \cdot \log_2 n$ comparisons.
- May get slow if pivot consistently fails to split the array into approximately equal halves.
- An $\boxed{O(n \log n)}$ algorithm.