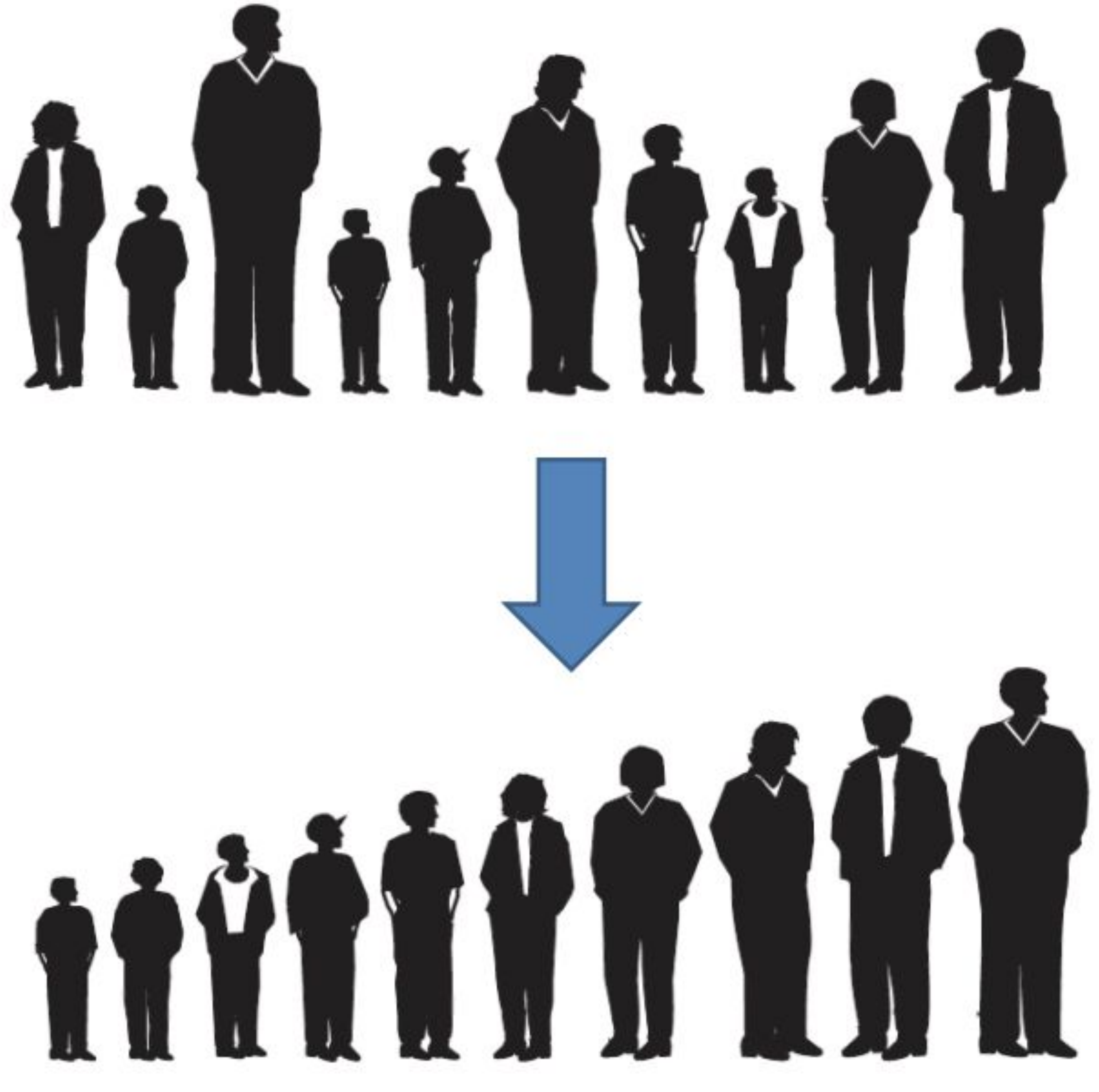
A Familiar Example

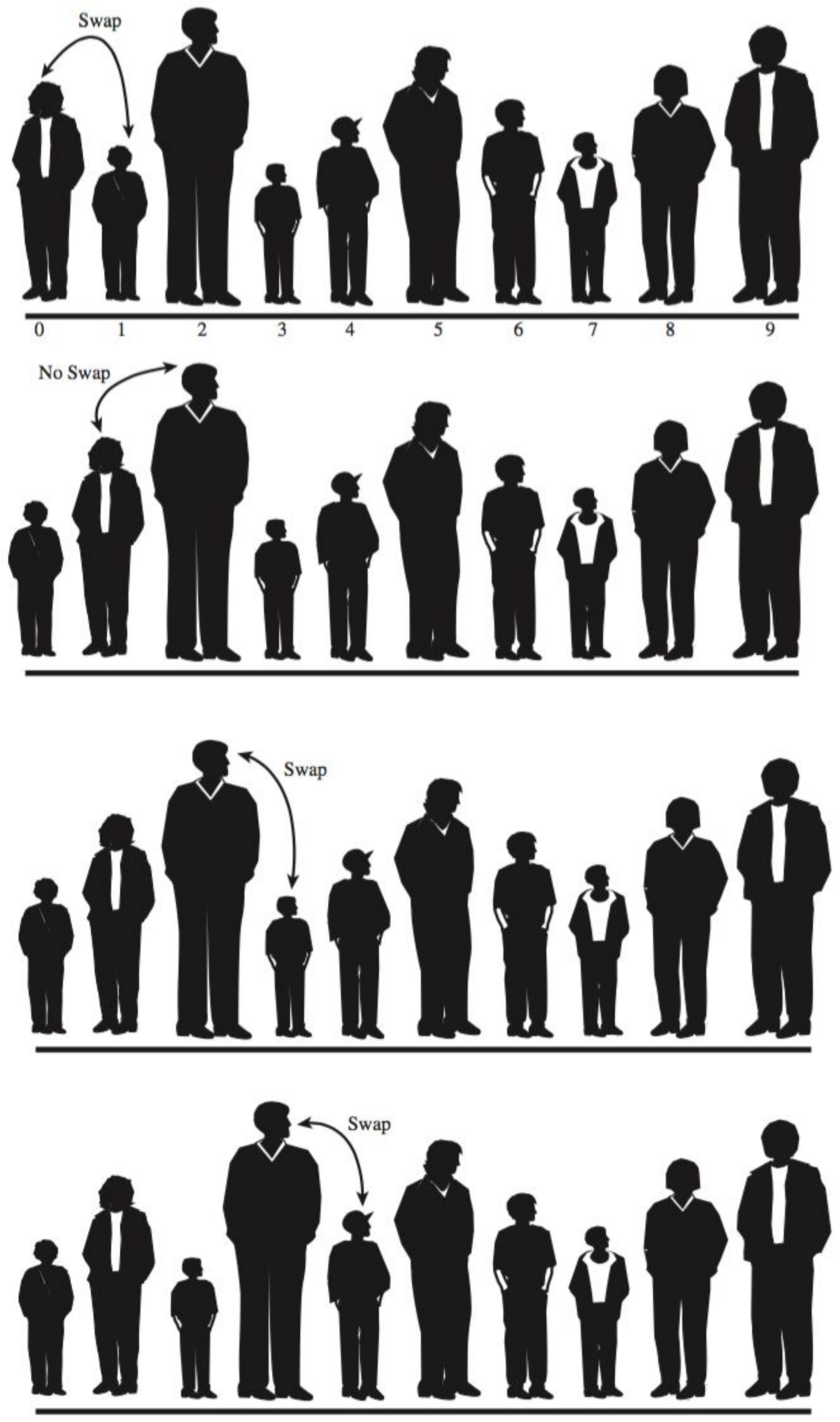


Sorting

- Given an array of comparable objects, sort them in ascending (default) or descending order.
- As a human being, we can perform this task quickly
    - Spot the tallest person / shortest person right away
    - Simultaneously compare multiple persons
- A computer cannot see the big picture all at once
    - It's limited to compare two numbers at a time, swap or copy them, and move on to the next pair
    - However, computers can do these really really fast!

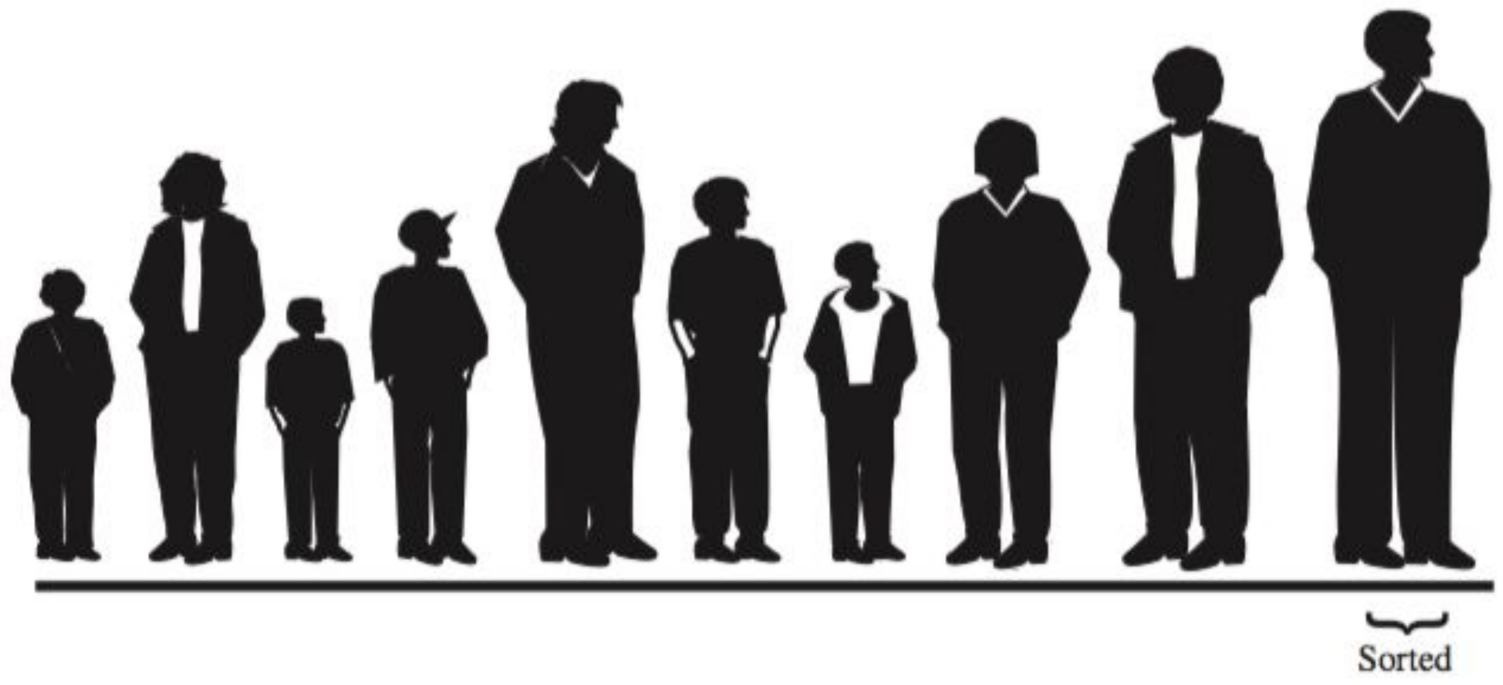
Bubble Sort

- **Intuition**
  - Round 1: find the largest number
  - Round 2: find the second largest number
  - Round 3: find the third largest number
  - …
- Bubble sort finds the largest number in each round by repeatedly comparing every two adjacent numbers, and swapping them if the one on the left is larger.


Swap
0
1
2
3
4
5
6
7
8
9
No Swap
Swap
Swap

Bubble Sort

- **After one pass, we find the largest number in that round**



- It’s like the biggest ‘bubble’ floats to the top of the surface, hence the name ‘bubble sort’.
- Let’s work together on this example: `13 1 5 10 7`

Bubble Sort

- In the second pass, we repeat the same process, but have only N-1 numbers to work on
- In the third pass, we have only N-2 numbers to work on
- …
- Repeat until we are left with just 1 number.
- How many comparisons in total do we have to perform?
  - How many comparisons in the 1st, 2nd, 3rd round?

$$\mathbf{(N-1) + (N-2) + (N-3) + \ldots\ 1 = N\ (N-1)\ /\ 2 = O(N^2)}$$

Bubble Sort

```
// assume elements are stored in T[] a;
// number of elements in variable nElems
public void bubbleSort() {
  int out, in;
  for(out=nElems-1; out>0; out--)  // outer loop
    for(in=0; in<out; in++)        // inner loop
      if(a[in].compareTo(a[in+1]) > 0)
        swap(in, in+1);
}
```

Bubble Sort

```
// swap elements stored at i and j
public void swap(int i, int j) {
  T temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

```
public void bubbleSort() {
  int out, in;
  for(out=nElems-1; out>0; out--)  // outer loop
    for(in=0; in<out; in++)        // inner loop
      if(a[in].compareTo(a[in+1]) > 0)
        swap(a[in], a[in+1]); // how about this?
}
// swap elements x and y
public void swap(T x, T y) {
  T temp = x;
  x = y;
  y = temp;
}
```

Wrong! This only swaps the local variables x and y, leaving the elements in the array intact!

Clicker Question #1

- In Bubble Sort, what's the minimum and maximum difference between number of swaps and number of comparisons? (hint: think about how many times the if statement is true, in the best case and the worst case)

```
public void bubbleSort() {
  int out, in;
  for(out=nElems-1; out>0; out--)
    for(in=0; in<out; in++)
      if(a[in].compareTo(a[in+1])>0)
        swap(in, in+1);
}
```

a) 0 and N(N-1)/2

b) 0 and N/2

c) 1 and N(N-1)/2

d) (N-1) and N^2

e) (N-1) and N(N-1)/2

Bubble Sort

- Number of swaps:
  - Best case: 0 (already in ascending order)
  - Worst case: N(N-1)/2 (in descending order)
  - Average: O(N^2)

Selection Sort

- As we’ve seen on the previous slide, Bubble Sort can lead to a large number of swaps (each 3 assignments).
- We can improve it by doing only one swap per outer loop, reducing the number of swaps to O(N).
- Idea:
    - Track the **<u>index</u>** of the smallest element in each round (previously we are finding the largest element. Now we are tracking the smallest, just to see some variety).
    - Swap the smallest element towards the beginning of the array after each round.
    - Repeat the above two steps.

Selection Sort

```
// assume elements are stored in T[] a;
public void selectionSort() {
  int out, in, min;
  for(out=0; out<nElems-1; out++){ // outer loop
    min = out;
    for(in=out+1; in<nElems; in++){// inner loop
      if(a[in].compareTo(a[min])<0)  min = in;
    }
    swap(out, min);
  }
}
```

Selection Sort

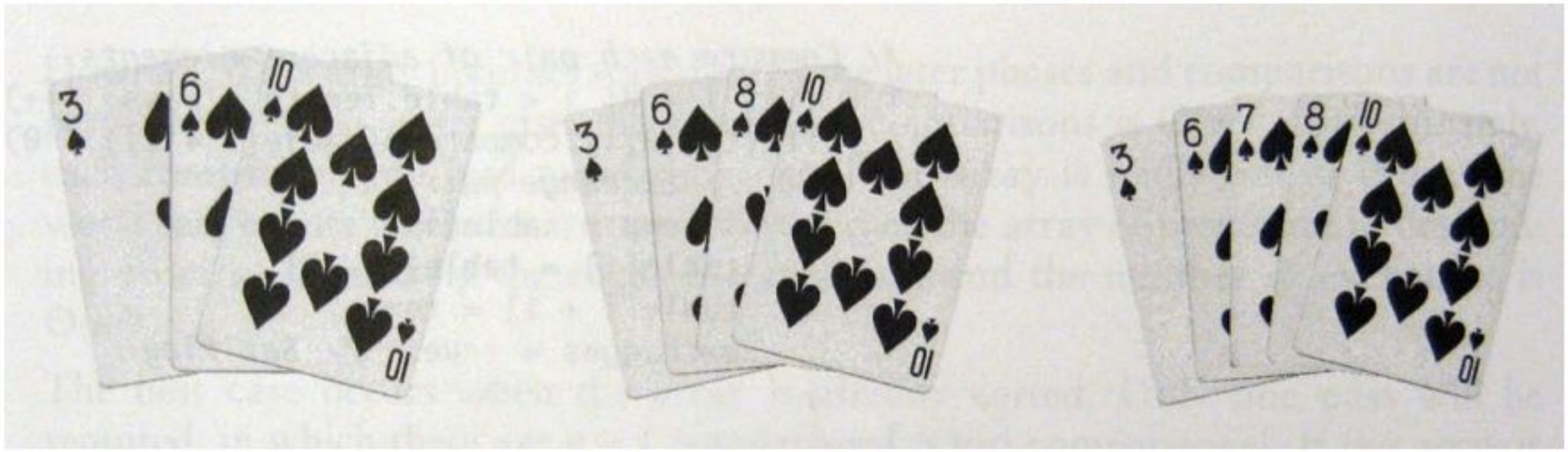
- Work together on this example: `13 10 1 5 7`
- This is slight different from Bubble Sort as you build up the sorted array from the left (smallest elements)
- Number of swaps?
  - O(N) in all cases (0 best case if you check out!=min)
- Number of comparisons?
  - Same as before: $N(N-1)/2 = O(N^2)$

So while selection sort reduces the number of swaps,

it does not reduce the number of comparisons.

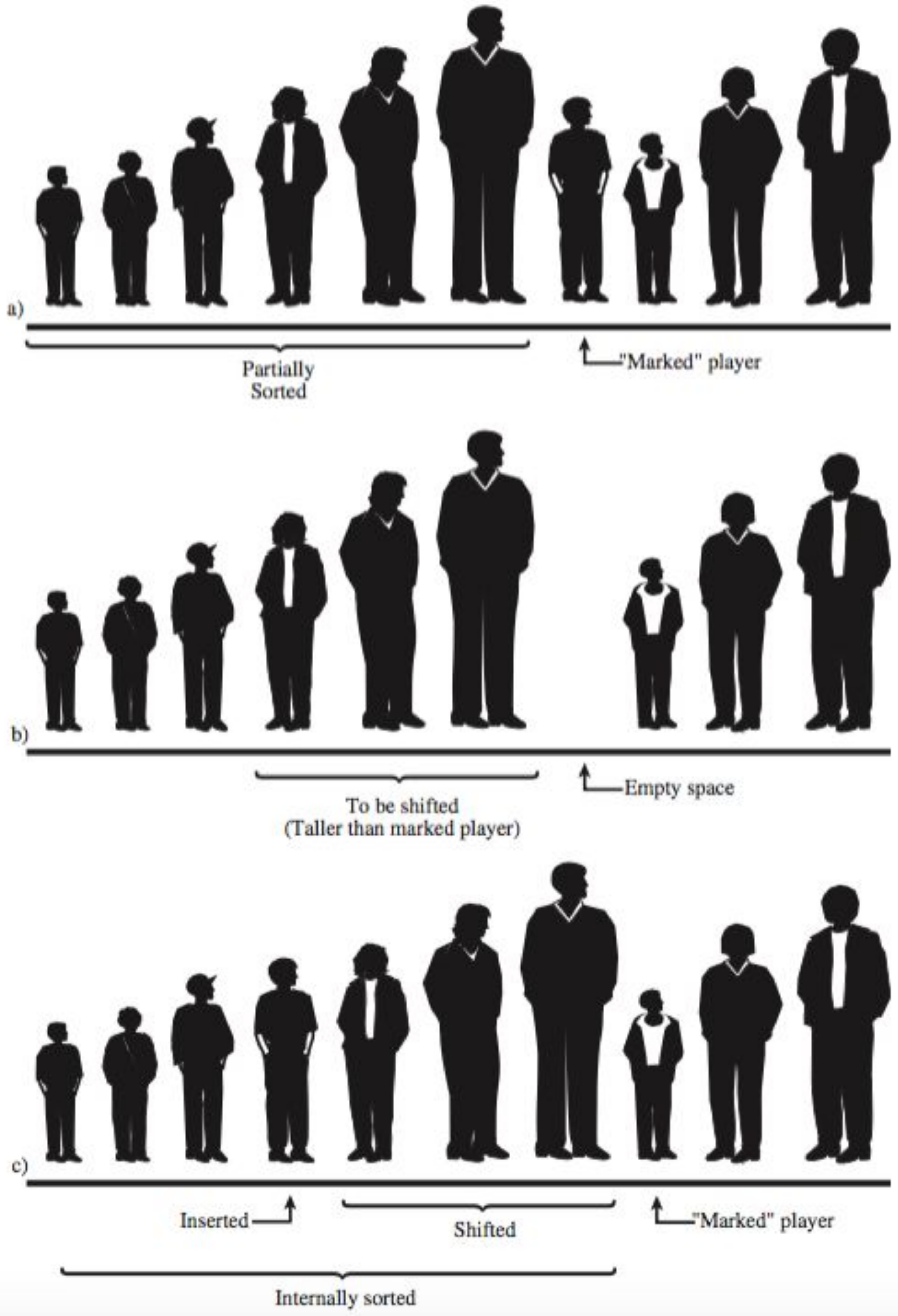
Insertion Sort

- How do you sort a set of poker cards?



Insertion Sort

- Idea
  - Build up the sorted array from the left (similar to selection sort).
  - Assume the left portion of the array is partially sorted, take the first element in the right portion, insert it to the left portion at the correct position (just like inserting an element to a sorted array).
  - Repeat until done.


a)
Partially Sorted
"Marked" player
b)
To be shifted (Taller than marked player)
Empty space
c)
Inserted
Shifted
"Marked" player
Internally sorted

Insertion Sort

```
// assume elements are stored in T[] a;
public void insertionSort() {
  int out, in;
  for(out=1; out<nElems; out++){ // dividing line
    T temp = a[out];
    in = out;        // start shifts at out
    while(in>0 && a[in-1].compareTo(temp)>0) {
      a[in] = a[in-1]; // shift right until the
      in--;  // element that’s no longer larger
    }
    a[in] = temp;    // copy
  }
}
```

Insertion Sort

- Work together on this example: `7 1 10 13 5`
- Note that our Insertion Sort code does not use swap (unlike Bubble Sort and Selection Sort). Instead, it uses a copy at the end of the outer loop. How is this a benefit?

  Swap involves 3 copies, so triples the cost!
- Number of copies:
  - Best case (while condition never true): N
  - Worst case: N(N-1)/2

Clicker Question #2

- In insertion sort, what's the minimum and maximum number of comparisons?

```
public void insertionSort() {
  int out, in;
  for(out=1; out<nElems; out++){
    T temp = a[out];
    in = out;
    while(in>0 &&
         a[in-1].compareTo(temp)>0) {
      a[in] = a[in-1];
      in--;
    }
    a[in] = temp;
  }
}
```

a) 0 and N(N-1)/2

b) 0 and N/2

c) 1 and N(N-1)/2

d) (N-1) and N^2

e) (N-1) and N(N-1)/2

Summary

- **Bubble Sort** uses repeated comparisons and swaps to find the biggest element in each pass, and positions it towards the end of the array.
- **Selection Sort** reduces the number of swaps by only performing one swap at the end of each pass.
- **Insertion Sort** eliminates swaps (vs bubble sort) and replaces them with copies, which are 3 times faster
- They are all **quadratic cost: $O(N^2)$ in the worst and average cases.**

Merge Sort

- You've already seen Merge Sort in the Queue project.
- Idea
    - **Divide and conquer.**
    - Can be implemented recursively or iteratively.
    - **Cost is O(N log N)**, much faster than simple sorting.
    - Requires additional memory space
        - A temporary array as large as the input array
        - So it's **not an in-place** sorting algorithm.

Merging Two Sorted Arrays

- This is a key step in Merge Sort.
- Assume two subsets of the array (Left and Right) are already sorted
- Merge them into array `temp` such that `temp` contains all elements from Left and Right, and remains sorted.
- Note the two subsets may have different sizes. In fact, one of them may be empty! Must correctly handle all cases! Example:

```
A: 4 6 10 13

B: 1 3 5 7 8
```

Merging Two Sorted Arrays

- Summary:

1. Start from the first elements of Left part of A and Right part of A.
2. Compare and **copy the smaller of the two** to `temp`.
3. Increment index of `temp` as well as the array where you picked the smaller element from.
4. Repeat.
5. If reaching the end of either Left or Right, quit loop and append the remaining elements to temp.
6. Copy all of temp back to A

```
public void merge(T[] A, T[] temp,
       int low, int mid, int high) {
  int li=low, ri=mid, ti=low;
  while((li < mid) && (ri <= high))
    T left=A[li++], right=A[ri++];
    if(left.compareTo(right) < 0) {
      temp[ti++] = left;
      temp[ti++] = right;
    } else {
      temp[ti++] = right;
      Temp[ti++] = left;
    }
  }  // end of loop

  for (i = low; i <= high;
    i++){A[i] = temp[i];}
}
```

Clicker Question #3

What's wrong of this algorithm? (Assume the parameters are correct passing value)

a) It works just fine
b) Doesn't change array
c) Doesn't sort it correctly
d) Sort in descending order
e) NullPointerException

```
public void merge(T[] A, T[] temp,
       int low, int mid, int high) {
  int li=low, ri=mid, ti=low;
  while((li < mid) && (ri <= high))
    if(A[li].compareTo(A[ri]) < 0)
      temp[ti++] = A[li++];
    else
      temp[ti++] = A[ri++];
  }  // end of loop
  while(li < mid)
     temp[ti++] = A[li++];
  while(ri <= high)
     temp[ti++] = A[ri++];
  }
```

Clicker Question #4

What's wrong of this algorithm? (Assume the parameters are correct passing value)

a) It works just fine
b) Doesn't change array
c) Doesn't sort it correctly
d) Sort in descending order
e) NullPointerException

```
public void merge(T[] A, T[] temp,
        int low, int mid, int high) {
   int li=low, ri=mid, ti=low;
   while((li < mid) && (ri <= high))
      if(A[li].compareTo(A[ri]) < 0)
         temp[ti++] = A[li++];
      else
         temp[ti++] = A[ri++];
   }  // end of loop
   while(li < mid)
        temp[ti++] = A[li++];
   while(ri <= high)
        temp[ti++] = A[ri++];
   }
   for (i = low; i <= high;
         i++){A[i] = temp[i];}
}
```

Clicker Question #5

What's wrong of this algorithm? (Assume the parameters are correct passing value)

a) It works just fine
b) Doesn't change array
c) Doesn't sort it correctly
d) Sort in descending order
e) NullPointerException

```
public void merge(T[] A, T[] temp,
                  int low, int mid, int high) {
  int li=low, ri=mid, ti=low;
  while((li < mid) && (ri <= high))
    if(A[li].compareTo(A[ri]) < 0)
      temp[ti++] = A[li++];
    else
      temp[ti++] = A[ri++];
  }  // end of loop
  while(li < mid) temp[ti++] = A[li++];
  while(ri <= high) temp[ti++] = A[ri++];
  for (i = low; i <= high; i++){A[i] = temp[i];}
}
```

Clicker Question #6

- To merge the following two sorted arrays:

```
A = [11, 50];

B = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100];
```

How many comparisons (compareTo) will be performed?

```
a)  10
b)  11
c)  12
d)  20
e)  22
```

```
public void merge(T[] A, T[] temp,
                  int low, int mid, int high) {
  int li=low, ri=mid, ti=low;
  while((li < mid) && (ri <= high))
    if(A[li].compareTo(A[ri]) < 0)
      temp[ti++] = A[li++];
    else
      temp[ti++] = A[ri++];
  }  // end of loop
  while(li < mid) temp[ti++] = A[li++];
  while(ri <= high) temp[ti++] = A[ri++];
  for (i = low; i <= high; i++){A[i] = temp[i];}
}
```

Merging Two Sorted Arrays

- Is it possible that both Left and Right have remaining elements after the first while loop? → No!
- What happens if Left (A) is empty to begin with?
- How many copy (assignment) instructions? This is the cost of the merge step:

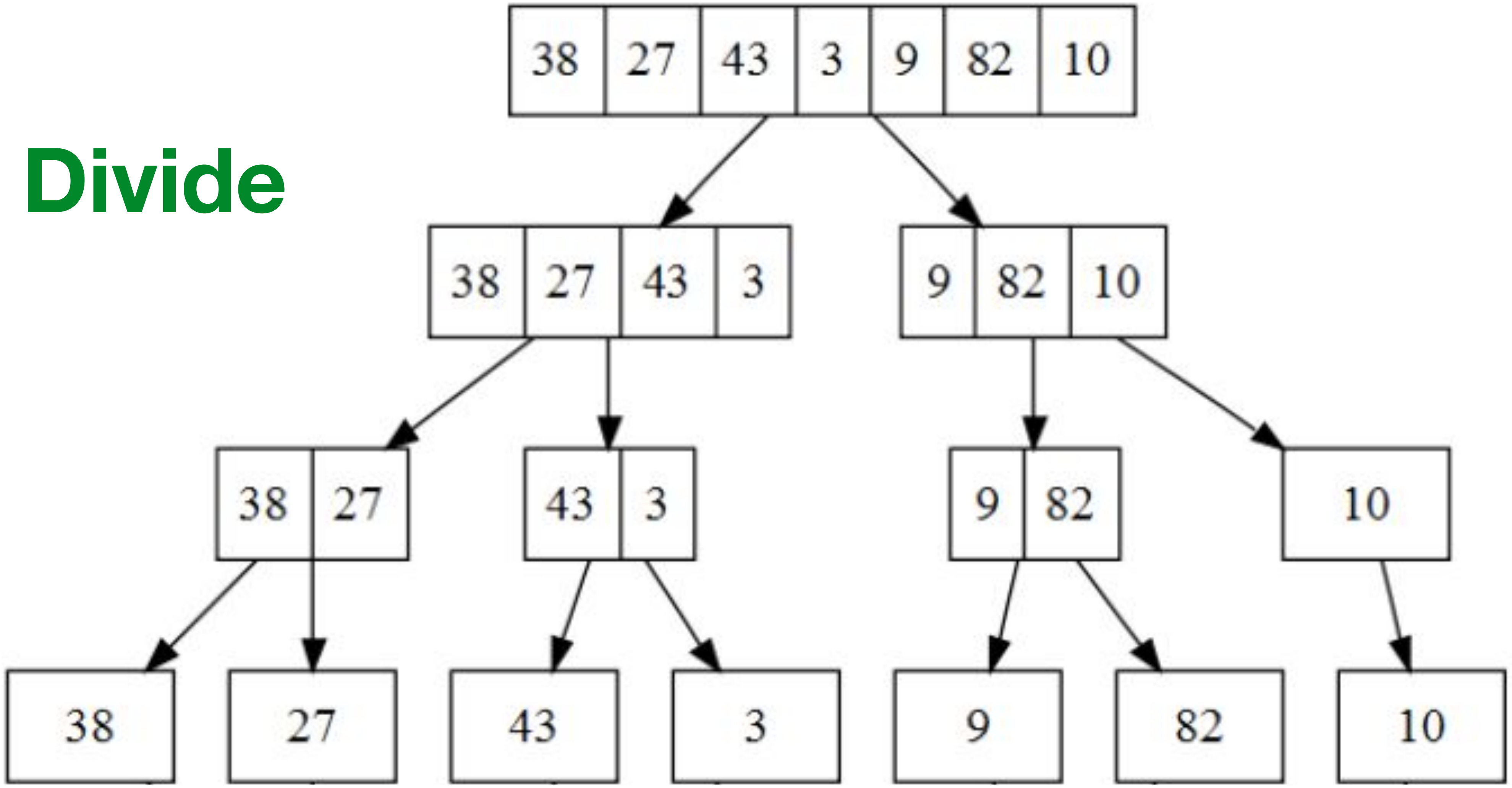
```
size of A + size of B
```

Merge Sort

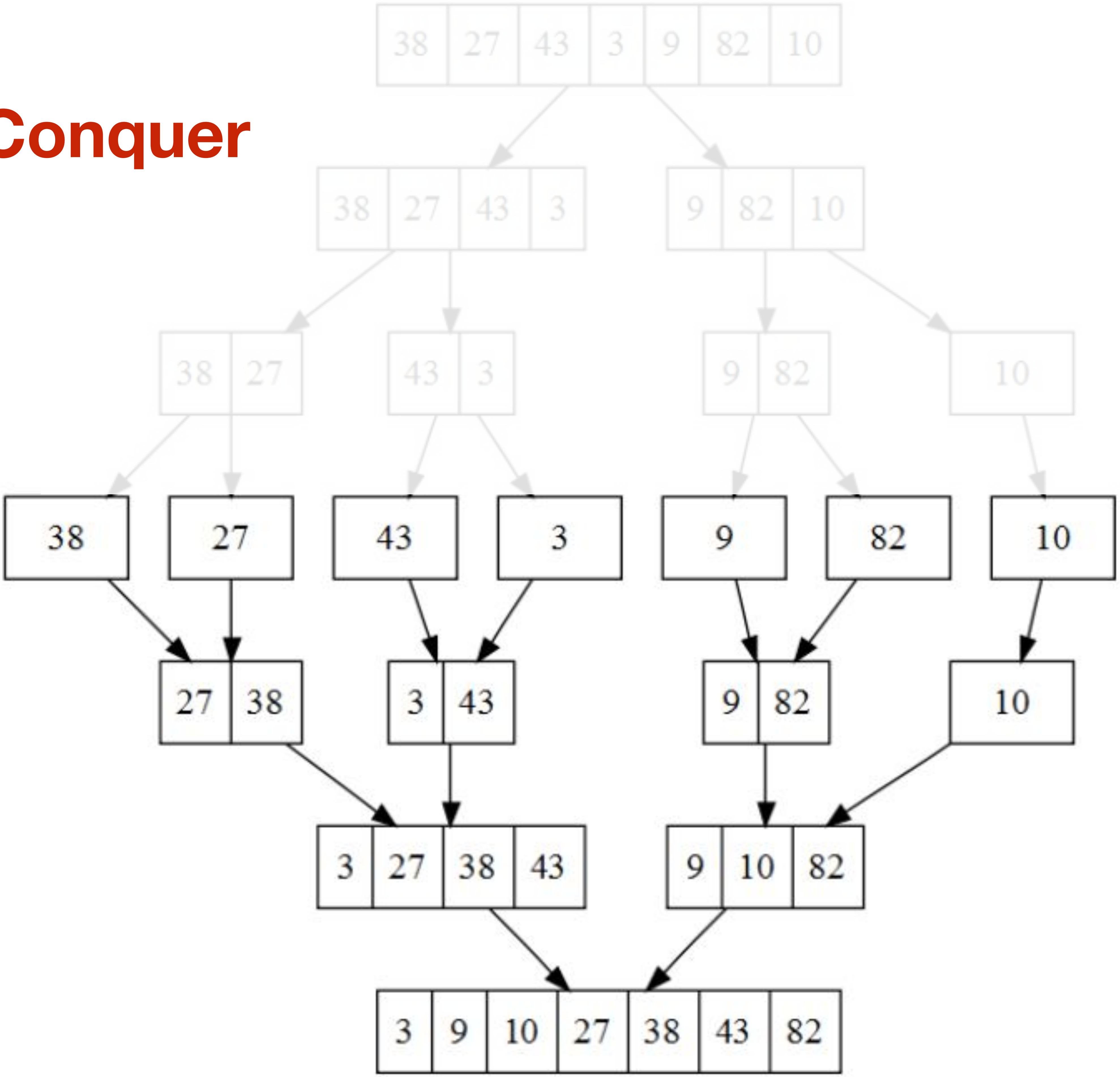
- With the `merge()` method, Merge Sort is quite simple:
  - **Divide** the array into two halves
  - Sort each half (**Conquer**). How? **Recursion!**
  - Call `merge()` to merge the two halves.
  - What’s the base case of this recursion?

When there is only 1 element left to sort, it’s trivially sorted, so return immediately.

Divide



Conquer



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

```
// assume elements are stored in T[] array ‘a’
// temp: scratch buffer to stored merged elements
public void mergeSort() {
  T[] temp = (T[])new Object[nElems];
  recMergeSort(a, temp, 0, nElem-1);
}
private void recMergeSort(T[] a, T[] temp,
                          int low, int high) {
  if(low >= high)  return;
  int mid = (low + high) / 2;
  recMergeSort(a, temp, low, mid);
  recMergeSort(a, temp, mid+1, high);
  merge(a, temp, low, mid+1, high);
}
```