EE2331 Data Structures and Algorithms

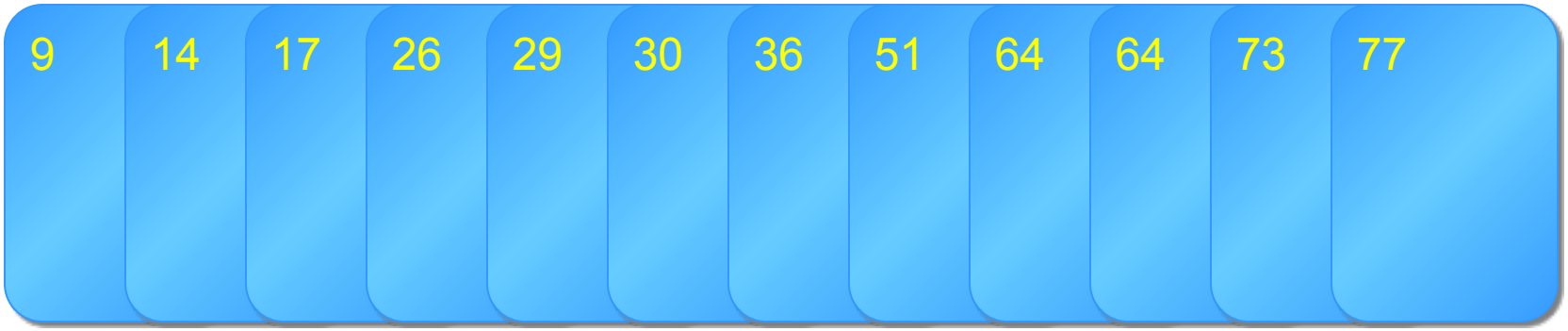
Sorting

Given a List in Random Order

- How to find the largest number?
- How to find the smallest number?
- How to determine if an arbitrary number exists in the list?

Given a List in Ascending Order

- How to determine the largest / smallest / any arbitrary number now?
  - Remember binary search?



  - The numbers can be also in descending order
- Or at least in some proper order (such as BST and heap)

Sorting

- To rearrange the order (ascending or descending) of data for **ease of searching**
- In this notes, discuss the various ways to sort a large amount of data and compare them by time/space efficiency.
  - O(n), O($n$log$n$), O(n^2) …
- Efficiency of a sorting method is usually measured by the number of comparisons and data movements required.

Outline

- Terminologies
- 6 sorting algorithms
  - Bubble Sort, Insertion Sort, Merge Sort
  - Heapsort, Quicksort, Radix Sort
- Sorting using Queues
- Sorting using Stacks
- Indirect Sorting

Terminologies

Stable vs. Unstable

Internal vs. External

Stable & Unstable Sort

- Sequence before sorting: 5, 3, $8^{\#}$, 6, 8^{*}
- Sequence after sorting: 3, 5, 6, $8^{\#}$, 8^{*}
  - Stable sort
- Sequence after sorting: 3, 5, 6, 8^{*}, $8^{\#}$
  - Unstable sort

- Stable: if it always leaves elements with equal keys in their original order

Internal & External Sort

- Internal sort
  - Small data volume
  - Process in main memory
- External sort
  - Large amount of data
  - Need external or secondary storage in processing (e.g. disk storage)

Internal Sorting Algorithms

- In this course, we shall only discuss internal sorting algorithms. To simplify discussion, sorting of an integer array is used in our examples.
  1. Bubble Sort
  2. Insertion Sort
  3. Heap Sort
  4. Radix Sort
  5. Quick Sort
  6. Merge Sort (also good for external sort)

- How to choose the sorting algorithm?

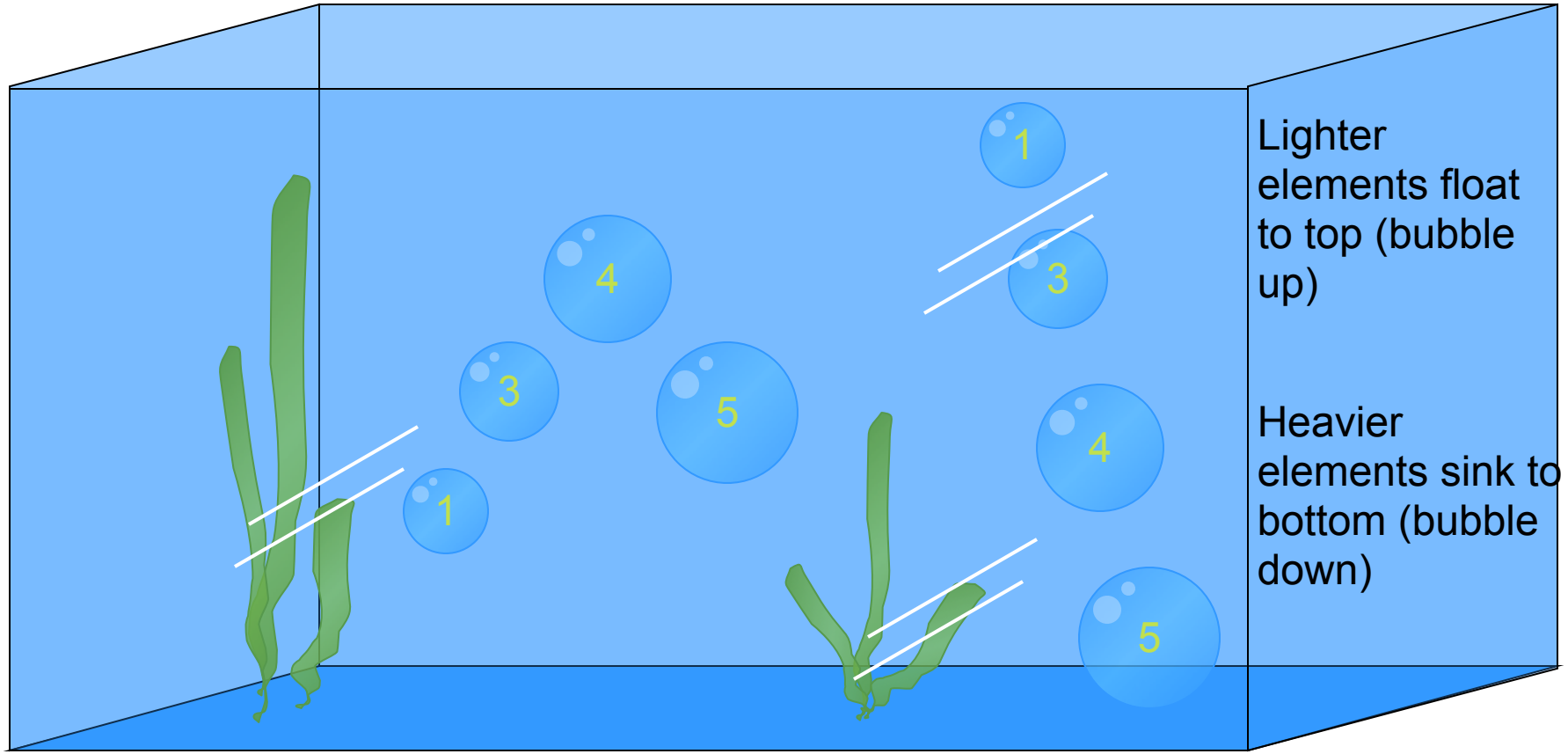
Bubble Sort

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Daily Life Example

- Consider the goldfish bowl

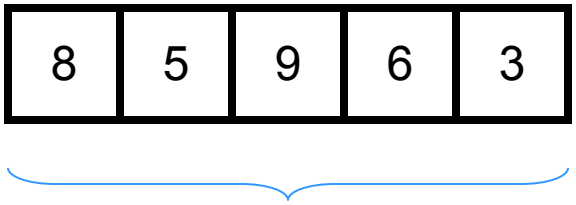


Bubble Sort

- The easiest sorting algorithm
- The most time consuming algorithms
- Another name: **interchange sort**
- The idea:
  - Scanning the list from one end to the other
  - When a pair of adjacent keys is found to be out of order, swap those entries
  - In each pass, the largest key in the list will be bubbled to the end, but the earlier keys may still be out of order

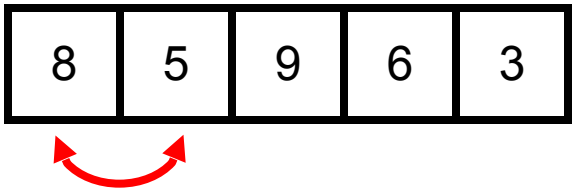
Bubble Sort Example

- Sort the sequence {8, 5, 9, 6, 3} in ascending order
- The final result should be {3, 5, 6, 8, 9}



Bubble Sort: 1st Pass

- 1st pass, 1st comparison



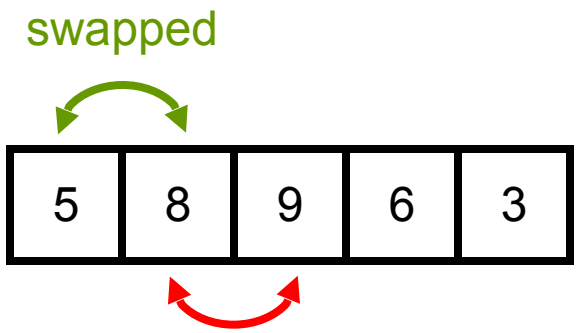
Compare 1st element with 2nd element

i.e. 8 vs. 5

if left hand side > right hand side, swap them!

Bubble Sort: 1st Pass

- 1st pass, 2nd comparison



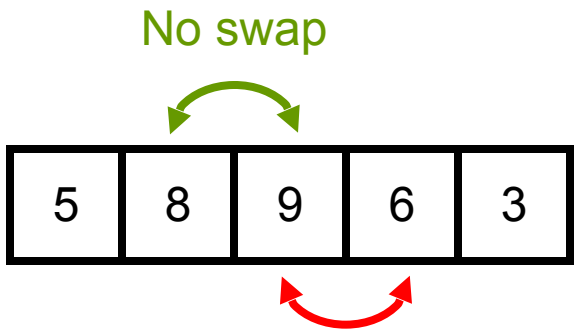
Compare 2nd with 3rd element

i.e. 8 vs. 9

Since left hand side < right hand side, do nothing!

Bubble Sort: 1st Pass

- 1st pass, 3rd comparison



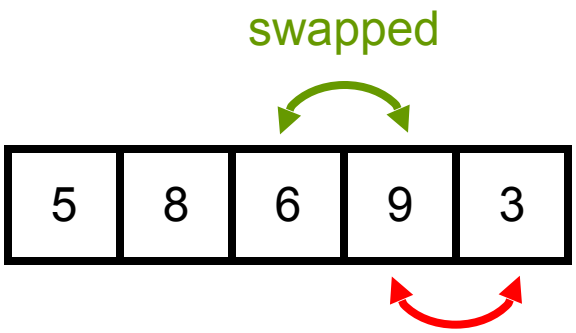
Compare 2nd with 3rd element

i.e. 9 vs. 6

Since left hand side < right hand side, swap them!

Bubble Sort: 1st Pass

- 1st pass, 4th comparison



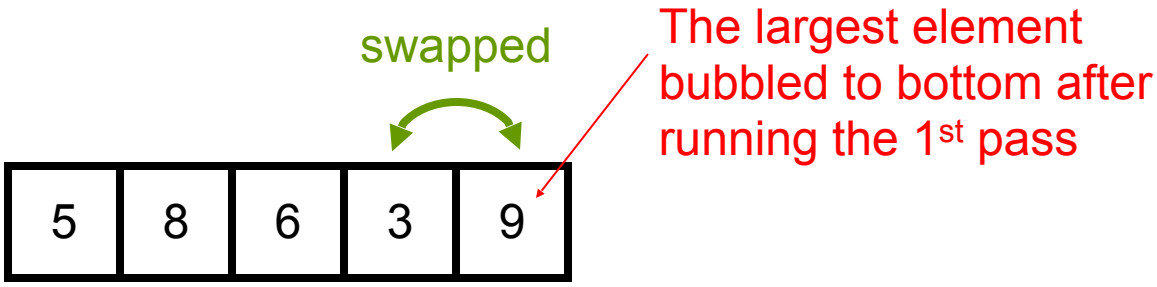
Compare 2nd with 3rd element

i.e. 9 vs. 3

Since left hand side < right hand side, swap them!

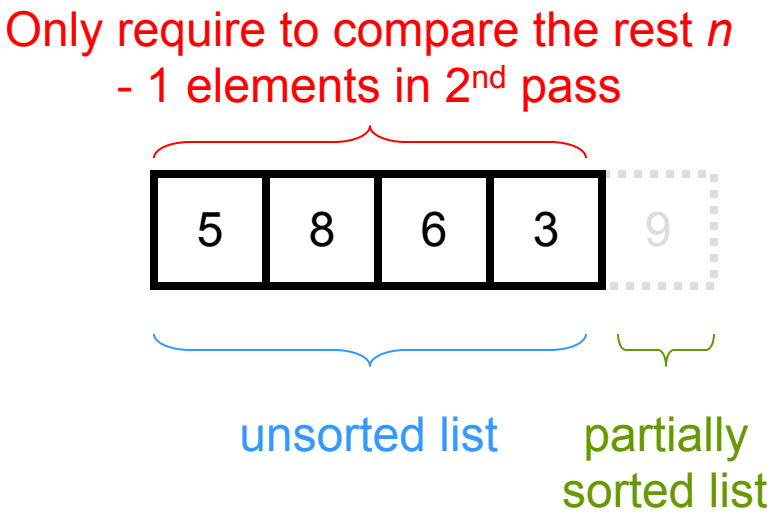
Bubble Sort: 1st Pass

- After 1st pass



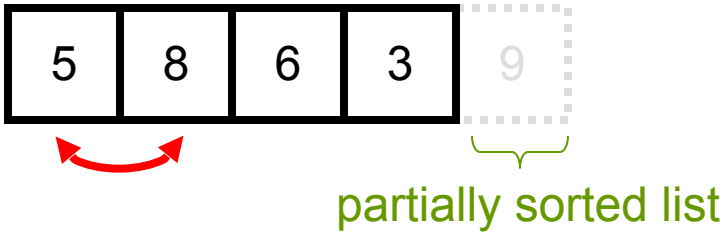
Bubble Sort: 2nd Pass

- Start from 2nd pass, no need to consider the largest element (the last element)

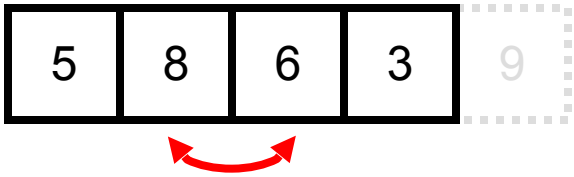


Bubble Sort: 2nd Pass

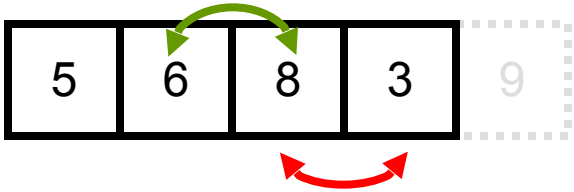
2nd pass, 1st comparision



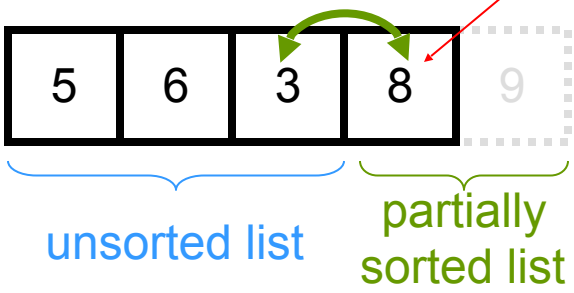
2nd pass, 2nd comparision



2nd pass, 3rd comparision

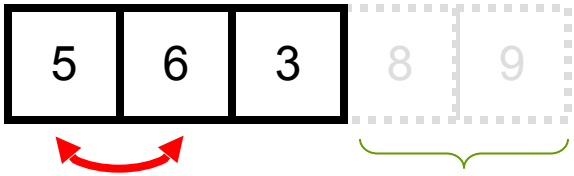


After 2nd pass



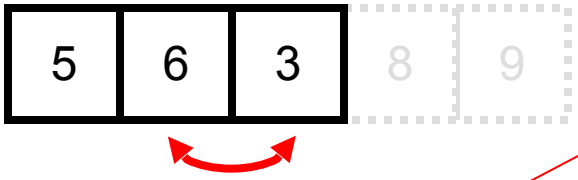
Bubble Sort: 3rd Pass

3rd pass, 1st comparision



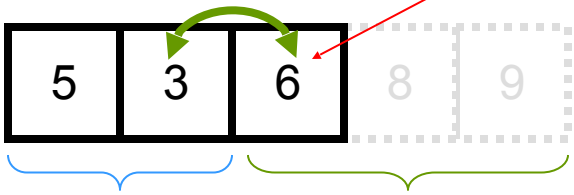
partially sorted list

3rd pass, 2nd comparision



The 3rd largest element fall to 3rd bottom after running the 3rd pass

After 3rd pass

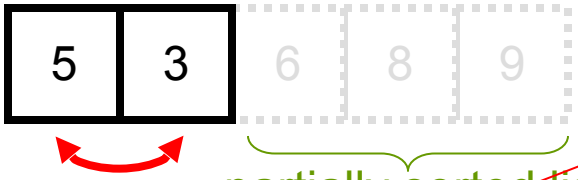


unsorted list

partially sorted list

Bubble Sort: 4th Pass

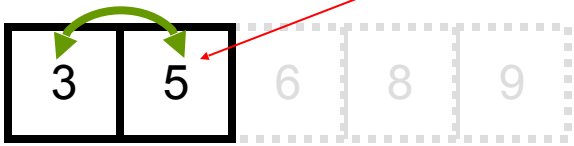
4th pass, 1st comparision



partially sorted list

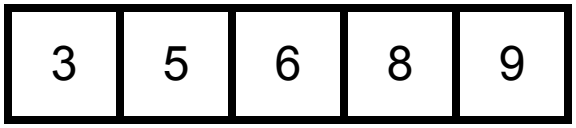
The 4th largest element fall to 4th bottom after running the 4th pass

After 4th pass



partially sorted list

The final sequence



sorted list

Not necessary to run the 5th pass (why?)

Time Complexity

- The amount of time to compare two numbers is constant O(1)
- The amount of time to swap two numbers is also constant O(1)
- The amount of time require to sort the sequence is proportional to the **number of comparisons** (or **swaps**)

How Many Comparisons?

- If there are *n* elements in total
  - No. of passes?
    - $n - 1$
  - How many comparisons in each pass?
    - i^{th} pass: $n - i$ comparisons
  - How many comparisons in total?
    - $$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$
  - Therefore, the time complexity is
    - $O(n^2)$

Simple Version

```
void bubble(int data[], int n) {
    int i, j;
    //sort in ascending order
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - 1 - i; j++)
            if (data[j] > data[j+1])
                swap(&data[j], &data[j+1]);
}
```

Mind the for-loop indexes here

i control the no. of passes

j control the no. of comparisons in each pass

1 pass

Each pass consists of comparing each element with its successor

Swap these two elements if they are not in proper order

After each pass *i*, the elements from ***data[n - i - 1]*** to ***data[n - 1]*** are sorted

Improved Version

```
void bubble(int data[], int n) {
    int i, j, no_swap;
    //sort in ascending order
    for (i = 0; i < n - 1; i++) {
        no_swap = true;
        for (j = 0; j < n – 1 - i; j++)
            if (data[j] > data[j+1]) {
                swap(&data[j], &data[j+1]);
                no_swap = false;
            }
        if (no_swap) break;
    }
}
```

1 pass

The algorithm breaks the for-loop and stops at once if there is no swap in one of the pass

How Many Swaps in Total?

- # of swaps is at most # of comparisons
  - $\leq \frac{1}{2} n(n-1)$
- The worst case: the algorithms has to run all the $n - 1$ passes
- The best case: (already sorted list) the algorithms stops after running the 1st pass (i.e. O(n))

Drawback of Bubble Sort

- Slow
  - Worst case: O(n^2)
  - Average case: O(n^2)
    - Half the number of comparisons: $\sum_{i=1}^{n-1} (\frac{n-i}{2}) = \frac{n(n-1)}{4}$
  - Best case: O(n)

- Need one temporary space for swapping?

```
temp = a;
a = b;
b = temp;
```

Swap Without Using Extra Var

```
//sort in ascending order
int a = 13, b = 7;
if (a > b) {
    a ^= b;  //^ means XOR, a = a ^ b;
    b ^= a;
    a ^= b;
}
```

e.g. a: 13 $(1101)_2$, b: 7 $(0111)_2$

a: 1010_2, b: 0111_2

a: 1010_2, b: 1101_2

a: 0111_2, b: 1101_2

The values of *a* & *b* have been interchanged

Note:

1) this trick applicable to primitive data types only (e.g. char, integer, float, pointers, etc). Cannot use this code to swap two structures, arrays.

2) *a* and *b* cannot be the same. Otherwise both a and b will reset to 0 after running this code.

Bubble Up and Down

- The previous algorithm bubble down the largest element in each pass
- The alternative way to implement bubble sort is:
  - Bubble up the smallest element to the front of the sublist in each pass
  - Their time and space complexities are the same

Bubble Up

```
void bubble(int data[], int n) {
    int i, j, no_swap = 0;
    //sort in ascending order
    for (i = 0; i < n – 1 && !no_swap; i++) {
        no_swap = true;
        for (j = n - 1; j > i; j--)
            if (data[j] < data[j-1]) {
                swap(&data[j], &data[j-1]);
                no_swap = false;
            }
    }
}
```

Mind the changes in red
j starts from end of the list up to *i*
If the right element is smaller, bubble up to the left of the list

1 pass

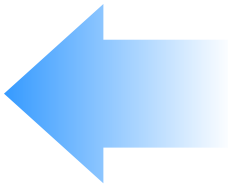
Insertion Sort

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Daily Life Example

- The idea of insertion is like playing cards



Insert into the cards

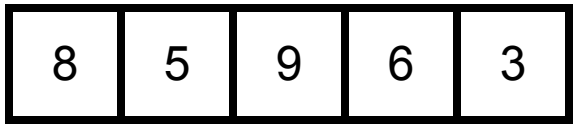
Any pre-condition?

Insertion Sort

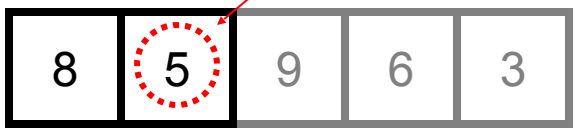
- Similar to bubble sort, consists of *n* - 1 passes
- Instead of bubbling the largest (or smallest) element, insertion sort successively inserts a new element into a (sorted) sublist in each pass

Insertion Sort Example

The unsorted list:



The 1st pass

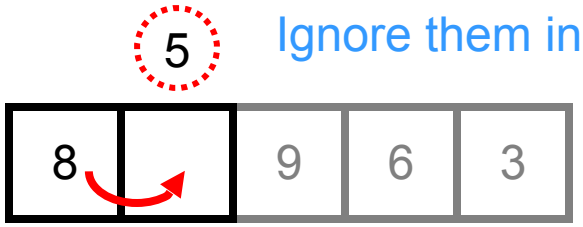


Compare with this sublist only

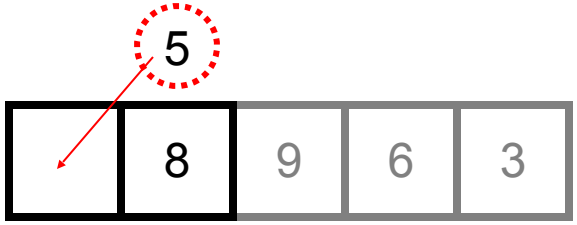
Insert this element into the left sublist such that they maintain a proper order

Ignore them in current pass

Pick up “5”. Move “8” to right

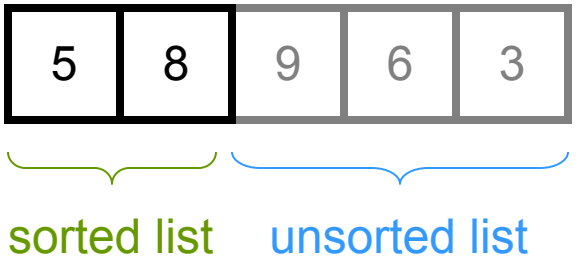


Insert “5” to the appropriate position



Insertion Sort Example

After 1st pass

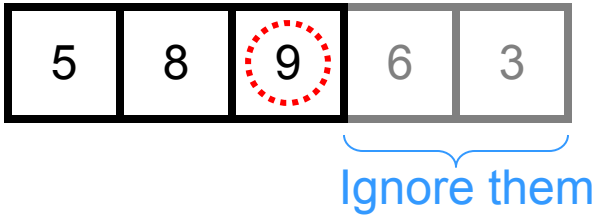


1 Move!

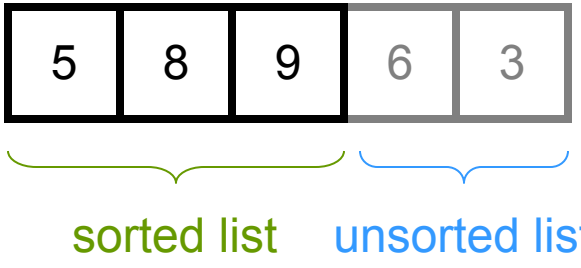
Compare with this sublist only

Insert this element into the left sublist such that they maintain a certain order

The 2nd pass

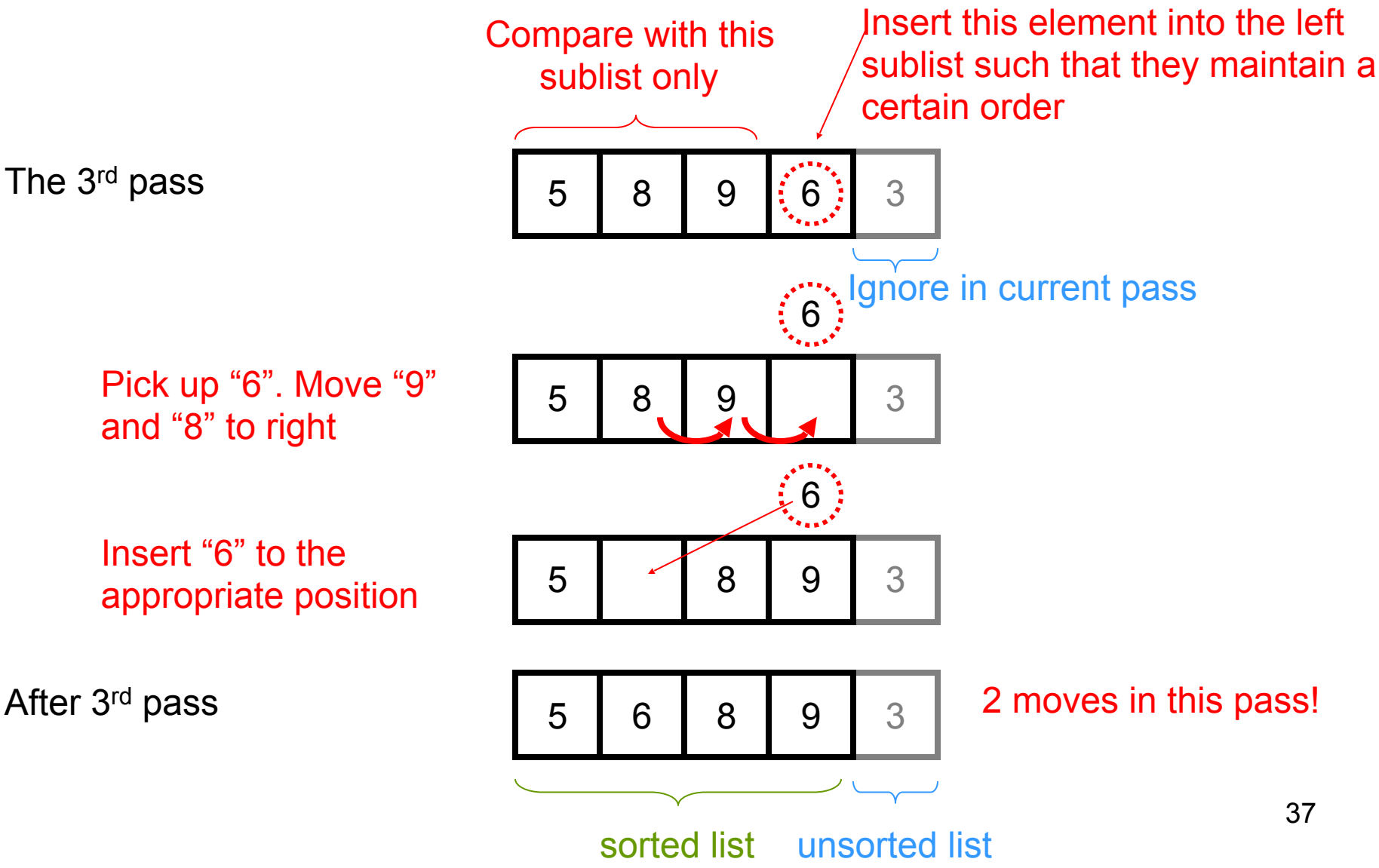


After 2nd pass

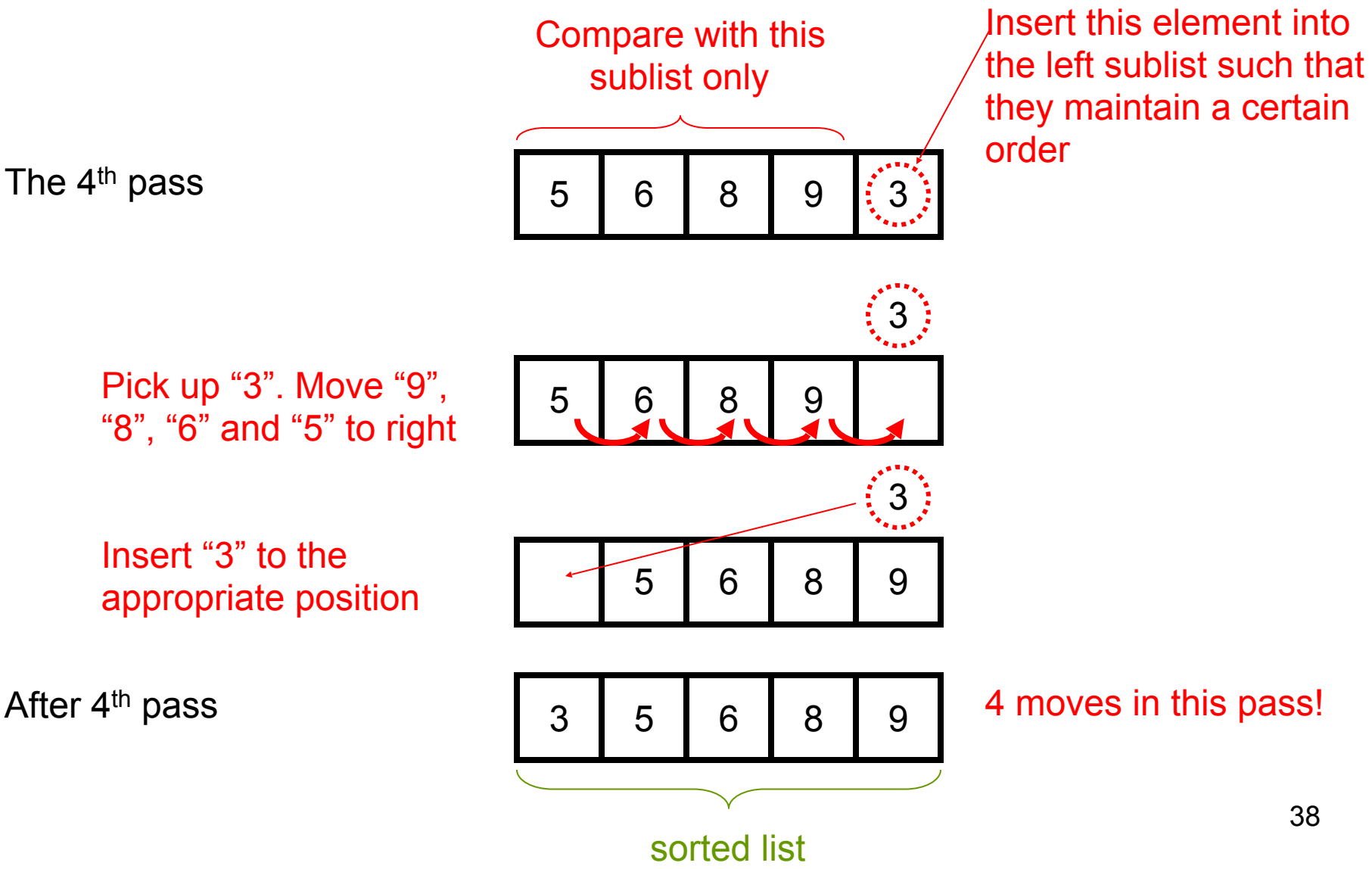


no move in this pass

Insertion Sort Example



Insertion Sort Example



Insertion Sort (correct version)

```
void insertion(int data[], int n) {
    int j, temp;
    for (int i = 1; i < n; i++) {   // n -1 passes
        temp = data[i];             // element to be inserted
        // shift the elements in the sublist if they are not in order.
        // the sublist is from data[0] to data[i]
        j = i-1;
        while( j >= 0 && data[j] > temp)
            data[j+1] = data[j];
            j--;
        data[j+1] = temp;     // j+1 is the location for insertion
    }
}
```

1 pass

Generic Version (correct version)

- Generic sorting function for any data type

```
template<class Type>
void insertionSort(Type *x, unsigned N,
                   int (*compare)(const Type&, const Type&)) {
   int j;

   for (int i = 1; i < N; i++) {
      Type t = x[i];
      j = i-1;
      while (j = i-1; j >= 0 && compare(x[j], t) > 0)
         x[j+1] = x[j];
         j--;

      x[j+1] = t;
   }
}
```

function pointer

Insertion Sort

- Initially *data*[0] may be thought of as a sorted list of one element
- After each loop *i* (from 2 to *n)*, the elements *data*[0] through *data*[*i*] are in proper order
- Insertion sort makes use of the fact that elements in positions 1 through *i*-1 are already known to be in sorted order

Complexity Analysis

- Like bubble sort, need an extra temporary memory
  - Space complexity: O(1)
  - Bubble sort: the temp. variable is used for **swapping**
  - Insertion sort: the temp. variable is used to **hold** the element that going to be inserted into the sublist

Complexity Analysis

- The best case
  - The list is already sorted; scan it once!
  - O(*n*)
- The worst case
  - *n-1* items to be inserted
  - At most *i* comparisons at *i-th* insertion
  - The total no. of comparisons = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
  - O(n^2)
- The average case
  - Half the number of comparisons
  - O(n^2)

Summary (Average Performance)