Algorithms and their Applications CS2004 (2020-2021)

Dr Mahir Arzoky

6.1 Classic Algorithms - Sorting

NOTICES

Note on Laboratory Worksheets

- Class tests
  - As of Friday (last week):
    - Mock Test: 174 attempts
    - Class Test 2: CR I: 71 attempts
  - Class Test 3 will be released in week 8
- Task 1 and Task 2 components are based on the content of laboratory worksheets
- Remember not to let these worksheets "pile up"
  - Do not leave any worksheet longer than 2 weeks
- Don't miss the introduction at the beginning of the laboratory

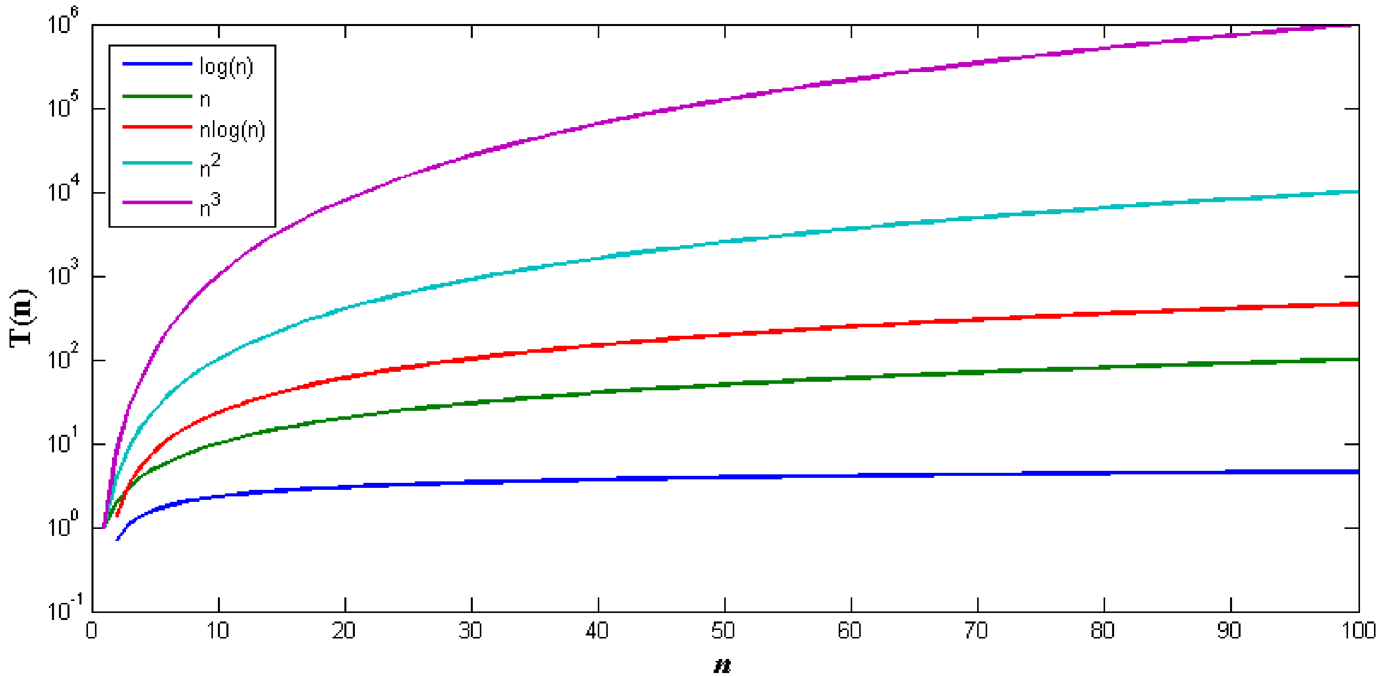
Previously On CS2004…

- So far we have looked at:
  - Concepts of Computation and Algorithms
  - Comparing algorithms
  - Some mathematical foundation
  - The Big-Oh notation
  - Computational Complexity
  - Last week we also looked at data structures…

Classic Algorithms - Sorting

- Within this lecture we are going to look at three sorting algorithms in detail
    - Bubble Sort
    - Quick Sort
    - Radix Sort
- We will also look [very] briefly at other sorting algorithms

Growth Rates of Algorithms



- A plot of n against a number of possible $T(n)$
- The y-axis is in logarithmic scale so that all the data can be viewed
  - I am using log base 10, but often log base 2 is used
- When $n = 100$
  - $\log(n) = 2$, $n = 100$, $n\log(n) = 200$, $n^2 = 10{,}000$ and $n^3 = 1{,}000{,}000$

Recap – Why Bother With Sorting?

- Sorting applications are one of the most common algorithms
- They are implemented in computer games, network routing software, operating systems, AI algorithms, bin packing algorithms, etc….
- The sorting problem itself is easily understood
    - It does not require a large amount of background knowledge before you can start implementing it
- The algorithms are quite small and manageable
    - They can be implemented in a short period of time
    - They are relatively simple to analyse

Formal Definition of Sorting

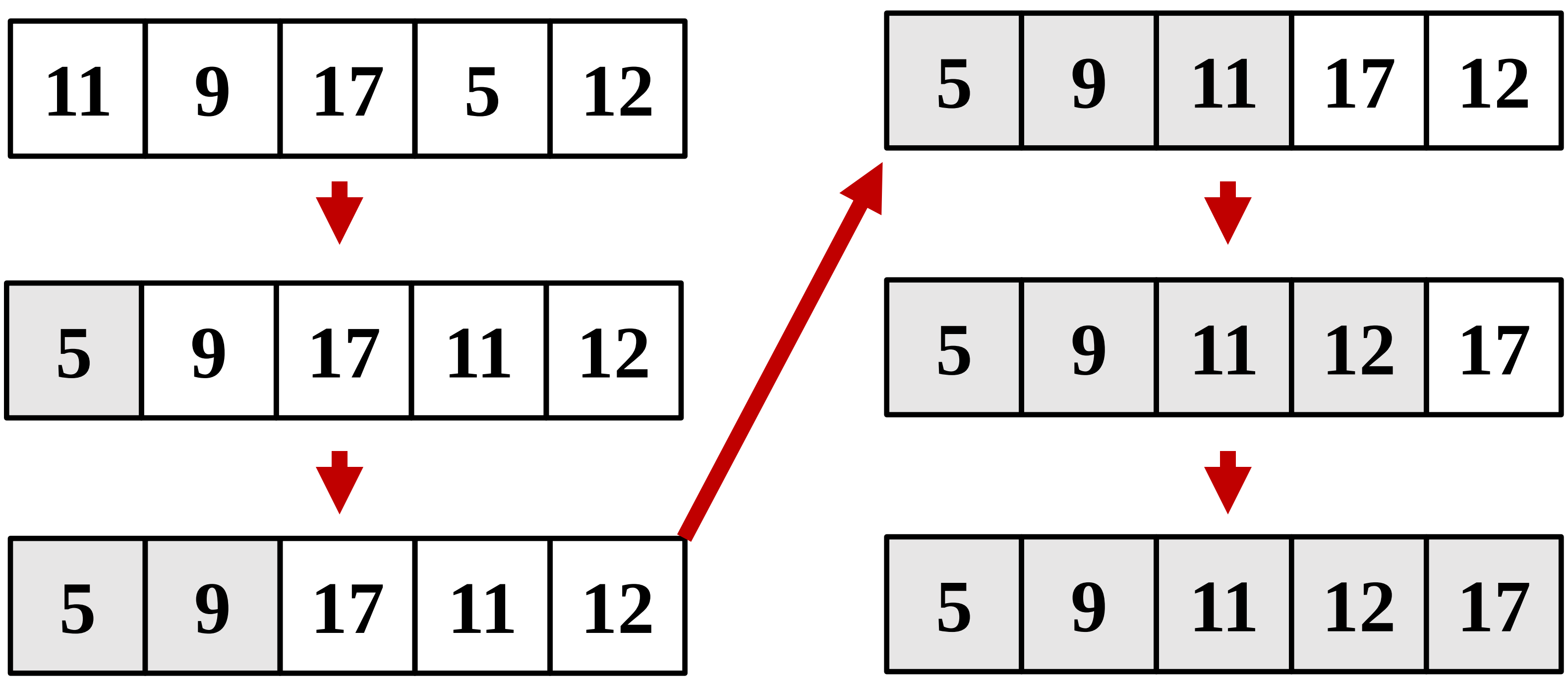
- The sorting problem is a mapping from x to y, where
  - x and y are both n length real vectors (lists and/or arrays)
  - y is a permutation of x (different ordering)
  - $y_i < y_{i+1}$ for all $i=0,\ldots,n-1$
- The problem of reordering items of an array in a certain order
- The algorithms can easily be applied to integers or character vectors
  - Sorting whole numbers
  - Sorting names and/or addresses
  - Essentially anything that can be ordered/compared…

Applications

- When an array is sorted, many problems involving this array becomes easier, for example:
  - To search for a specific value
  - To find the smallest and/or largest value (min/max)
  - To count how many times a specific value appear in array
  - To find out how unique are the values and delete duplicates
  - To do intersection and/or union between two different arrays
  - etc…

Recap: Selection Sort

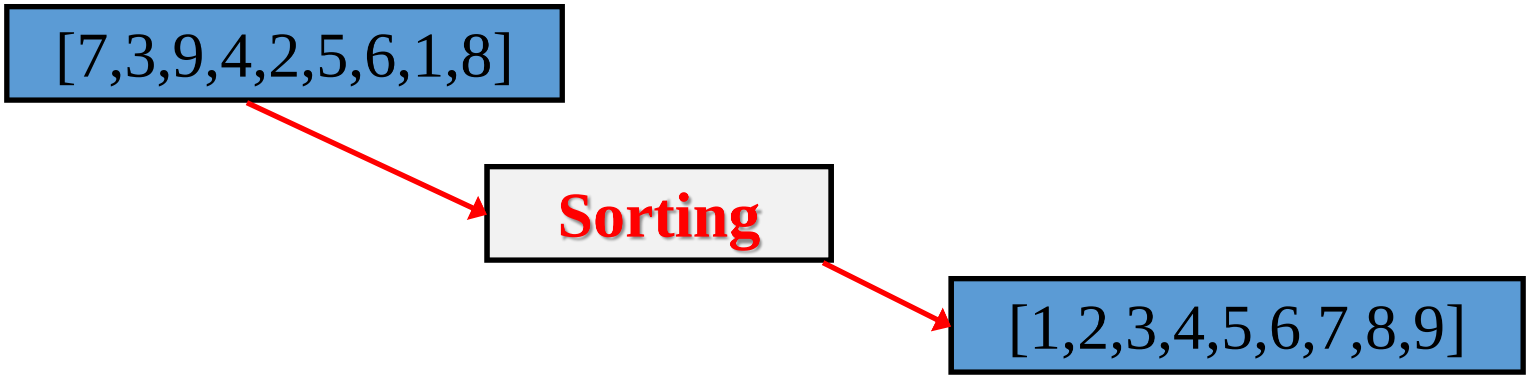
- ❑ So far you only looked at Selection Sort…
- ❑ It sorts an array by repeatedly finding the minimum element from unsorted part of the array and puts it at the beginning…



Monkey Sort!

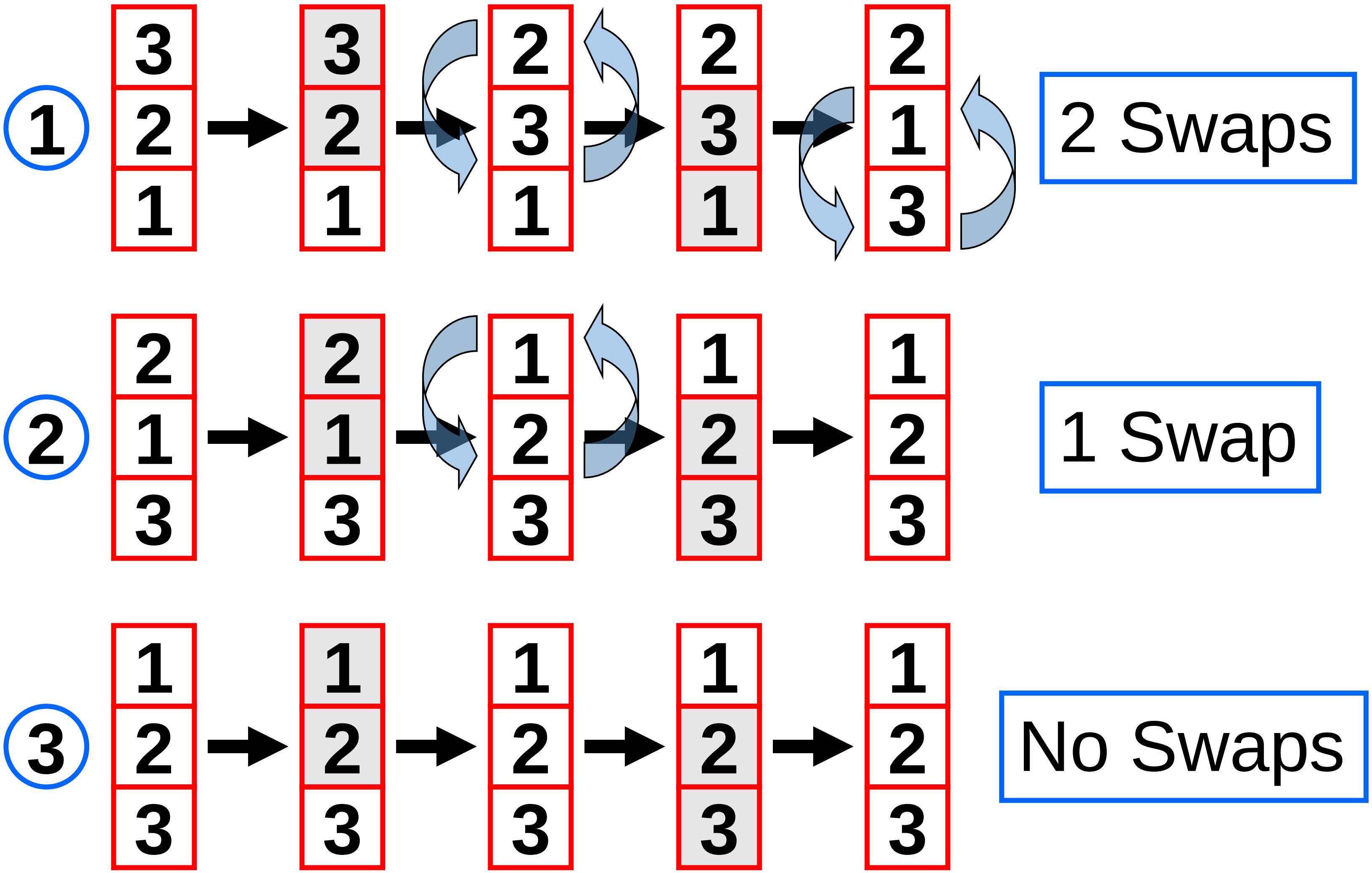
- Bogosort, stupid sort, slowsort, shotgun sort, permutation sort…
- Highly inefficient sorting algorithm
- Successively generates permutations of its input until it finds one permutation that is sorted!
- Best case:
  - $O(n)$
- Worst case:
  - $O(n!)$

Bubble Sort



- It is one of the simplest sorting algorithm
- Repeatedly compare adjacent pairs of elements
- Swap the elements in pair putting the smaller element first
- When it reaches end of list, starts over
- It stops when no more swaps can be made

Bubble Sort Worked Example



Bubble Sort: Summary

- If we compare pairs of adjacent elements and none are out of order, the list is sorted
- If any are out of order, we must have to swap them to get an ordered list
- Bubble sort will make passes though the list swapping any elements that are out of order

Bubble Sort: An Example – Part 1

Pass 1:

[7, 3, 9, 4, 2, 5, 6, 1, 8]

↓

[**3**, **7**, 9, 4, 2, 5, 6, 1, 8]

↓

[3, 7, 9, 4, 2, 5, 6, 1, 8]

↓

[3, 7, **4**, **9**, 2, 5, 6, 1, 8]

↓

[3, 7, 4, **2**, **9**, 5, 6, 1, 8]

↓

[3, 7, 4, 2, **5**, **9**, 6, 1, 8]

↓

[3, 7, 4, 2, 5, **6**, **9**, 1, 8]

↓

[3, 7, 4, 2, 5, 6, **1**, **9**, 8]

↓

[3, 7, 4, 2, 5, 6, 1, **8**, **9**]

Observations:

- ❑ After the first pass, the list has not been sorted yet!
- ❑ 8 (or n-1) comparisons have been made!
- ❑ The largest element (the bubble), 9, has been moved to the end!
- ❑ The smaller elements have been shifted towards to the beginning

Questions:

- ❑ How many elements do we need to sort in the next pass?
- ❑ When will the sort algorithm stop?
- ❑ What about the best case? How many passes?
- ❑ What about the worst case? How many passes?

Bubble Sort: An Example – Part 2

Overall result:

Original	7	3	9	4	2	5	6	1	8
Pass 1	3	7	4	2	5	6	1	8	9
Pass 2	3	4	2	5	6	1	7	8	9
Pass 3	3	2	4	5	1	6	7	8	9
Pass 4	2	3	4	1	5	6	7	8	9
Pass 5	2	3	1	4	5	6	7	8	9
Pass 6	2	1	3	4	5	6	7	8	9
Pass 7	1	2	3	4	5	6	7	8	9
Pass 8	1	2	3	4	5	6	7	8	9

In each pass, how many operations (comparisons) are needed?

Bubble Sort

```
Algorithm 1. BubbleSort(x)
Input: x - a list of n numbers
 1)  Let NoSwaps = False
 2)  While NoSwaps = False
 3)     Let NoSwaps = True
 4)     For i = 0 to n-2
 5)        If x_i > x_{i+1} then
 6)           Swap x_i and x_{i+1}
 7)           Let NoSwaps = False
 8)        End If
 9)     End For
10)  End While
Output: x – sorted (ascending)
```

- The algorithm works by running through the list of numbers, swapping pairs that are out of order
- The algorithm terminates when no swaps need to be made
- Smaller numbers “bubble” to the top whilst larger numbers sink to the bottom
- Line 5 needs changing to < in order to get the algorithm to sort the list in descending order
- **Note: to fit the pseudo code onto one slide, I have not reduced the list size index by one each iteration…**

Best-Case Analysis

- If the elements are in sorted order at the start, the for loop will compare the adjacent pairs but not make any changes
- This means that the `NoSwaps` variable will remain `true`
  - The `While` loop is only done once
- Thus, comparisons are done but not the swaps…
- There are n-1 comparisons in the best case
  - $B(n) \equiv O(n)$

Worst-Case Analysis

- If in the best case the while loop is done once, in the worst case the while loop must be done as many times as possible
  - This will be when the data is in **reverse order**
- Each pass of the `For` loop must make at least one swap of the elements
- The number of comparisons will be:

$$W(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \equiv O\left(n^2\right)$$

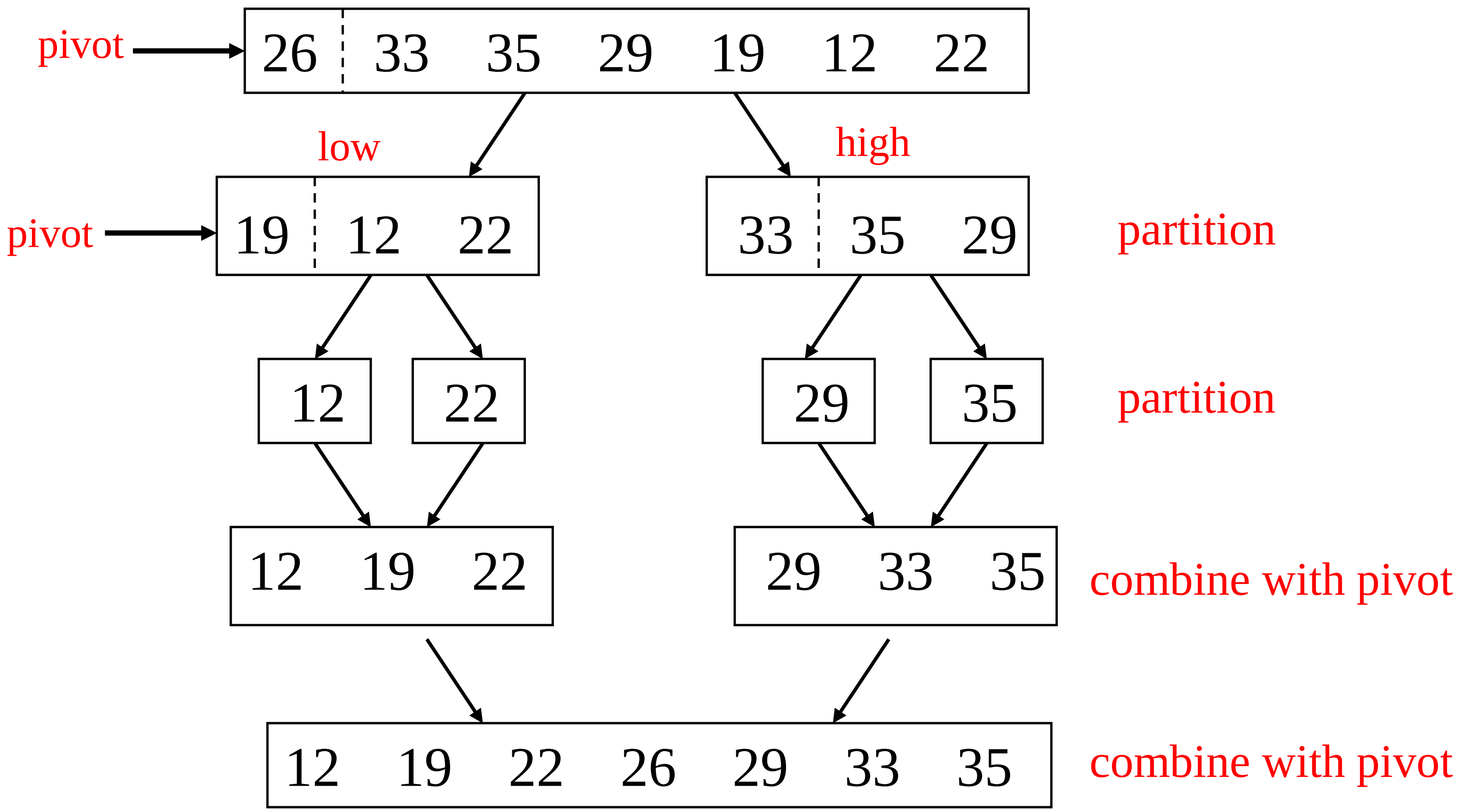
Quick Sort

- Quicksort is a divide and conquer algorithm
- Quicksort picks an element from the list as the **pivot**, and partitions the list into two pieces:
  - Those elements smaller than the pivot value (not necessarily in order)
  - Those elements larger than the pivot value (not necessarily in order)
- Quicksort is then called **recursively** on both pieces

Quick Sort: Example – Part 1

4	Pivot:4	2	Pivot:2	1		1
2		3		2		2
6		1		3		3
5		4		4		4
3		6		6	Pivot:6	5
7		5		5		6
1		7		7		7

Quick Sort: Example – Part 2



The Quick Sort Algorithm

```
Algorithm 2. QuickSort(List,First,Last)
Input: List, the elements to be put into order
       First, the index of the first element
       Last, the index of the last element
1) If First < Last Then
2)    Let Pivot = PivotList(List,First,Last)
3)    Call QuickSort(List,First,Pivot-1)
4)    Call QuickSort(List,Pivot+1,Last)
5) End If
Output: List in a sorted order
```

Two Algorithms: ***QuickSort*** (recursive), *PivotList* (defined later)

The PivotList Algorithm

```
Algorithm 3. PivotList(list,first,last)
Input: List- the elements to be put into order
       First- the index of the first element
       Last- the index of the last element
1)  PivotValue = list[first]
2)  PivotPoint = first
3)  For index = first+1 to last
4)     If list[index] < PivotValue Then
5)        PivotPoint=PivotPoint+1
6)        Swap(list[PivotPoint],list[index])
7)     End if
8)  End For
9)  Swap(list[first],list[PivotPoint])
Output: PivotPoint
```

Pick the first element as the pivot

Set the pivot point as the first location of the list

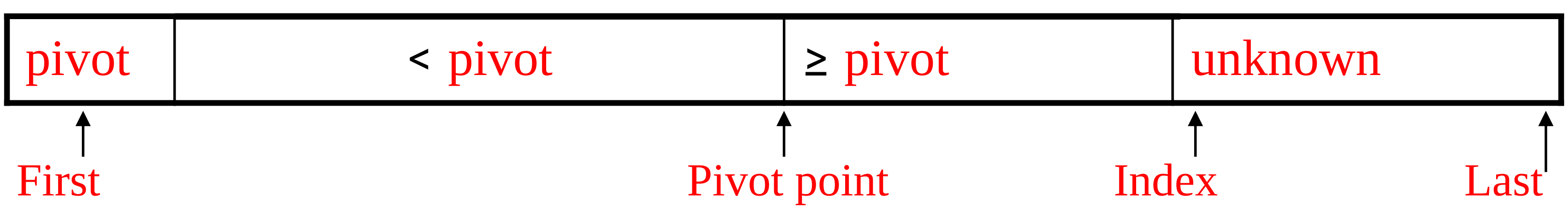
Move through the list comparing the pivot element to the rest

If an element is smaller

Increase the pivot point

Swap this element into the new pivot point location

Move pivot value into correct place



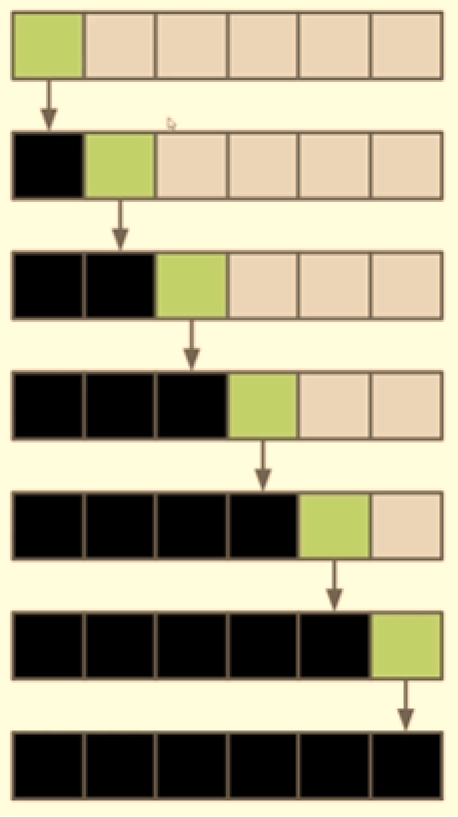
Variations of Quick Sort

- There are many different versions of Quick Sort, based on the different ways of picking the pivot:
  - Pivot always being the first element
  - Pivot always being the last element
  - Pivot is selected randomly
  - The median is chosen as the pivot

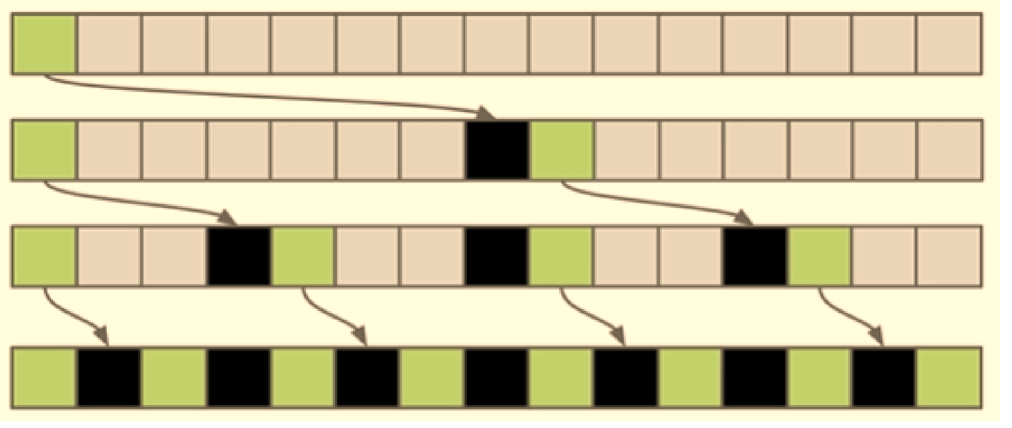
Worst-case Analysis

- In the worst case, `PivotList` will do n-1 comparisons, but creates one partition that has n-1 elements and the other will have no elements
  - When will this happen?
- Because we wind up just reducing the partition by one element each time
- The worst case is given by:

$$W(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2} \equiv O(n^2)$$



Best-case Analysis



That leaves only size 1 problems – so we are done!

- `PivotList` creates two parts that are the same size
- And then all subsequent parts are the same size as the algorithm calls itself, this can be modelled as a binary tree
- Summing up over the partitions we get $B(n)=B(nh)=O(n\mathrm{Log}_2(n))$

*h=number of levels

Comparison of Sorting Algorithms

Method	Best	Average	Worse
Bubblesort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Mergesort	$O(n\mathrm{Log}_2(n))$	$O(n\mathrm{Log}_2(n))$	$O(n\mathrm{Log}_2(n))$
Quicksort	$O(n\mathrm{Log}_2(n))$	$O(n\mathrm{Log}_2(n))$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

- ❑ We are interested in the worse case performance
- ❑ Useful to consider the average
- ❑ Can we do $O(n)$ for the worst case?

Radix Sort – Part 1

- The Radix sort method works only on binary or integer data
- Radix sort works by using a binary bucket sort for each binary digit
- We first “sort” by the least significant bit
- Split input into 2 sets based on the bit – those that have a 0 or those that have a 1
  - Otherwise maintain the order…
- Then proceed to the next least significant bit and repeat until we run out of bits…

Radix Sort – Part 2

Bits in the middle

Int	Binary rep					Sorted!	
0	000	0	000	0	000	000	0
7	111	6	110	1	001	001	1
3	011	7	111	5	101	011	3
1	001	3	011	6	110	101	5
6	110	1	001	7	111	110	6
5	101	5	101	3	011	111	7

Least significant order

Most significant order

Radix Sort – Part 3

- The Radix sort takes $O(nb)$ time complexity
  - n is the numbers items
  - b is the numbers of bits (in the representation)
  - Best, worse and average case
- It is very, very fast!
- Can be used for alphabetical sorting, i.e. strings

This Week's Laboratory

- This laboratory is also one of the worksheets you may be assessed on for **Task #1 and Task #2**
- We will look at testing three sorting algorithms

Next Lecture

- We will be looking at classic graph based algorithms
- Next week is ASK/Reading week
  - There will be no lectures or laboratories for this module…