Basic Sorting Algorithms (2)
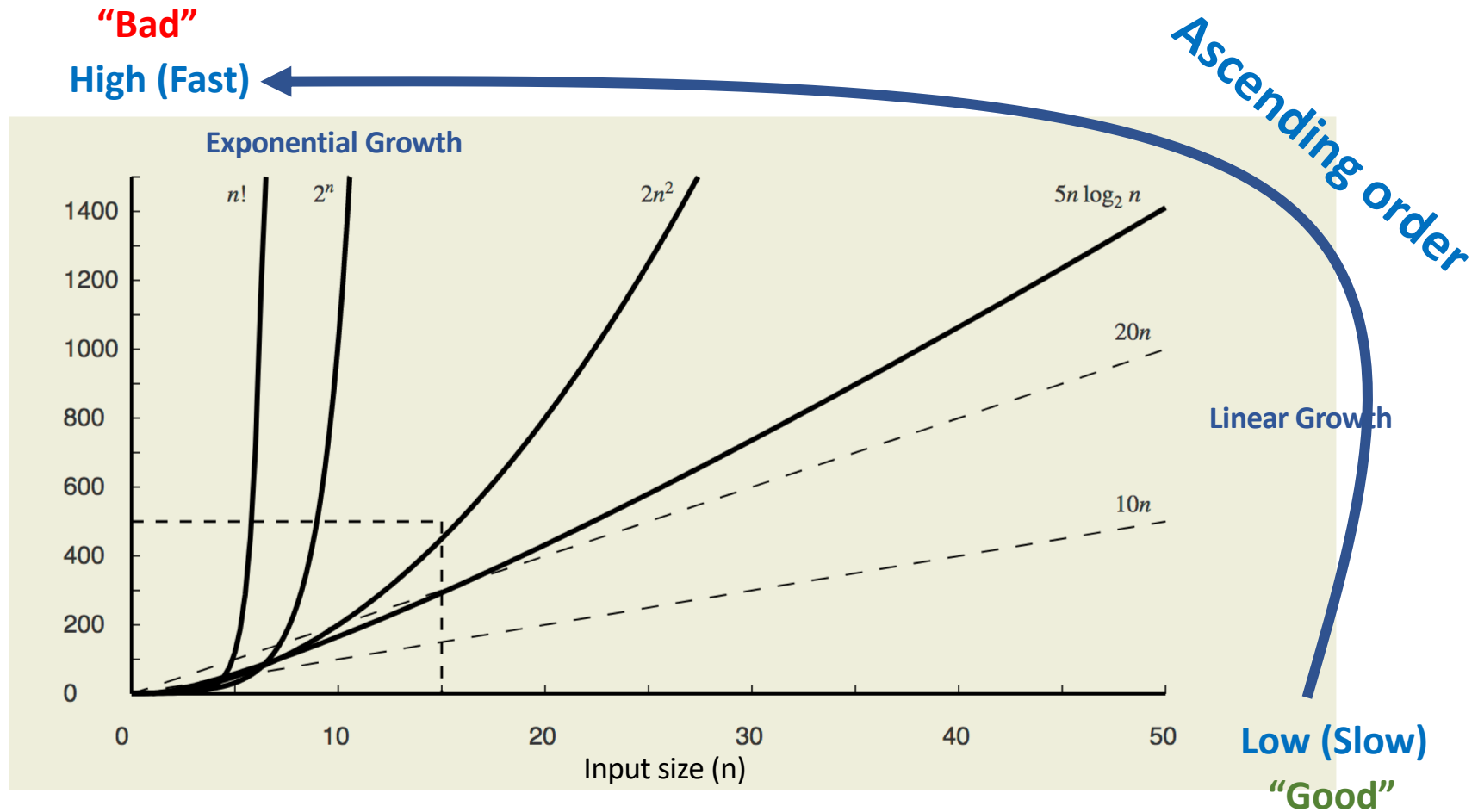
Instructor: Krishna Venkatasubramanian

CSC 212

Announcement

- Don't forget the office hours poll on Piazza.
  - Last day to do it today.
  - We will make our decision (to change or not to change) tomorrow.

- Who is Ming?

- Grade for Quiz 1 will be out soon.

Clarification: Asymptotic Growth Rates

Sorting: Problem Definition

- **Sorting takes an unordered collection and makes it an ordered one.**

	1	2	3	4	5	6
Input:	77	42	35	12	101	5

	1	2	3	4	5	6
Output:	5	12	35	42	77	101
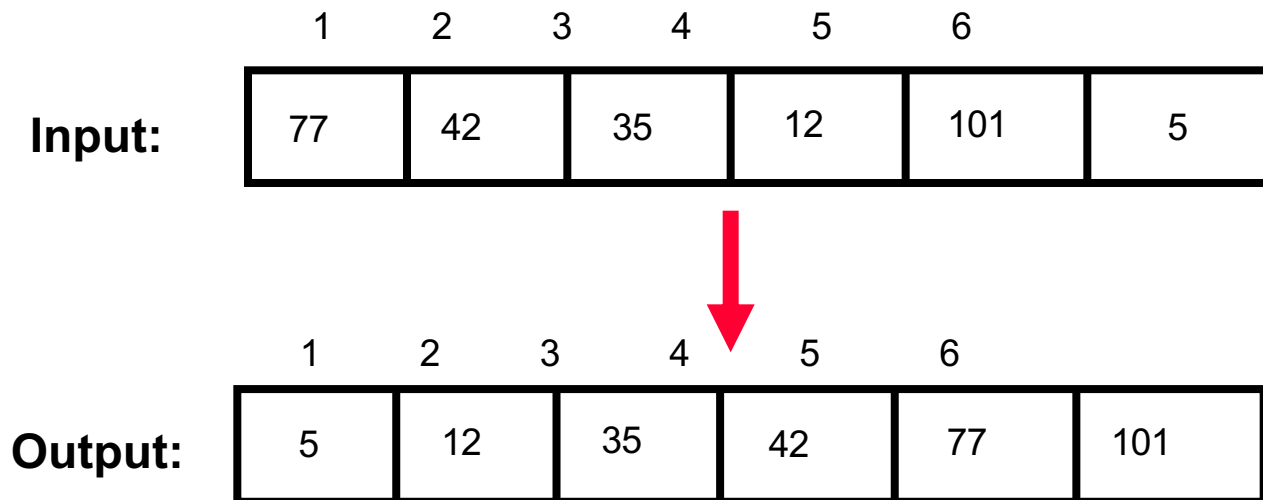
Sorting Algorithms

- *Insertion Sort --- covered already*

- *Bubble Sort --- covered already*

- **Selection Sort**

- Heap Sort

- Merge Sort

- Quick Sort

- …

Selection Sort

- Simple Algorithm --- has similarities to both Insertion-Sort and Bubble-Sort
  - Array divided into two parts – sorted and unsorted (like Insertion-Sort)
  - Stack the array from smallest to the largest (inverse of Bubble-Sort)

- **Invariants**
  - Elements in the sorted array are fixed
  - No element in the sorted array is greater than element in the unsorted array

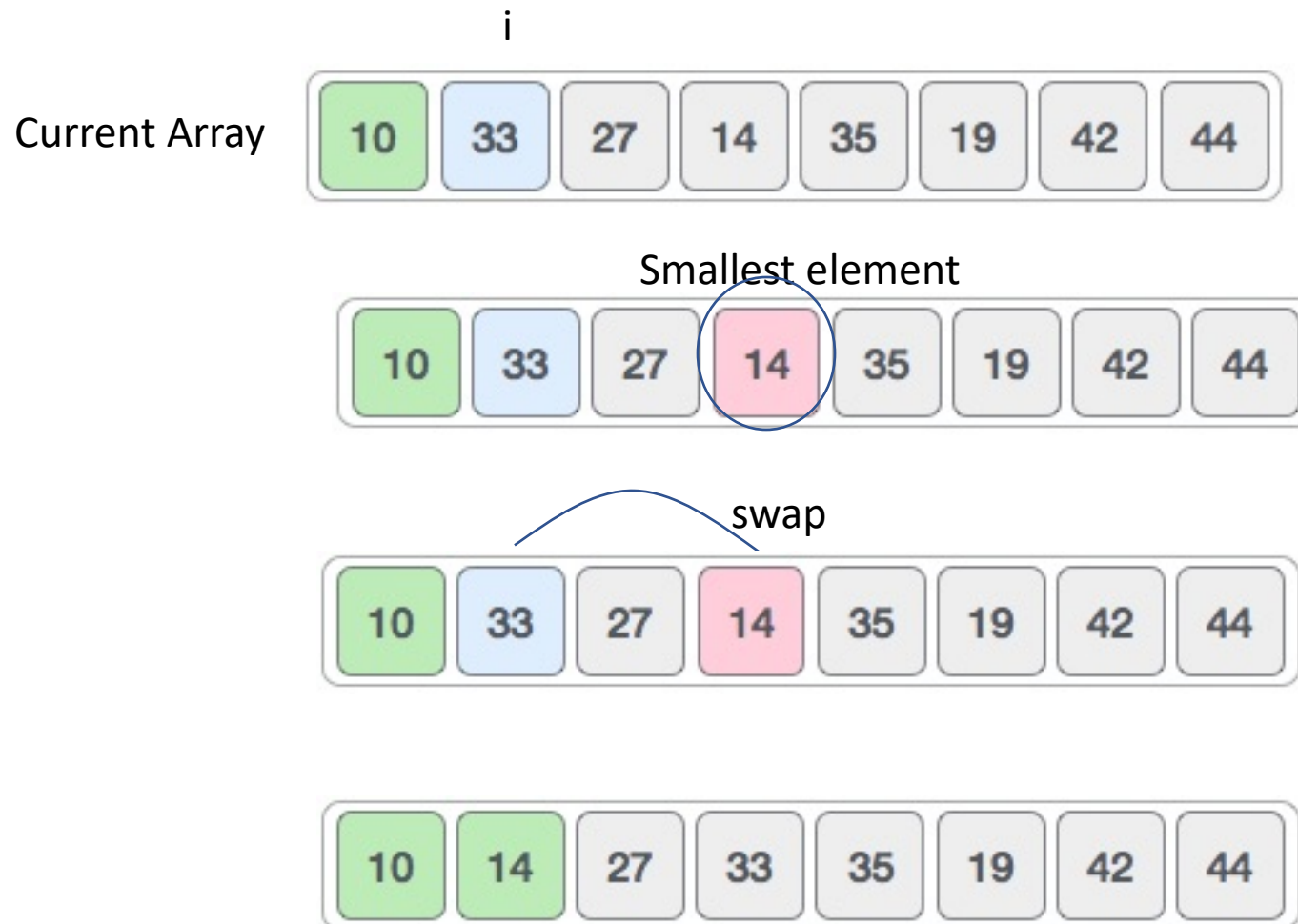| 1	2	4	5	18	9	10	7

SORTED ARRAY   UNSORTED ARRAY

Example (of basic steps)
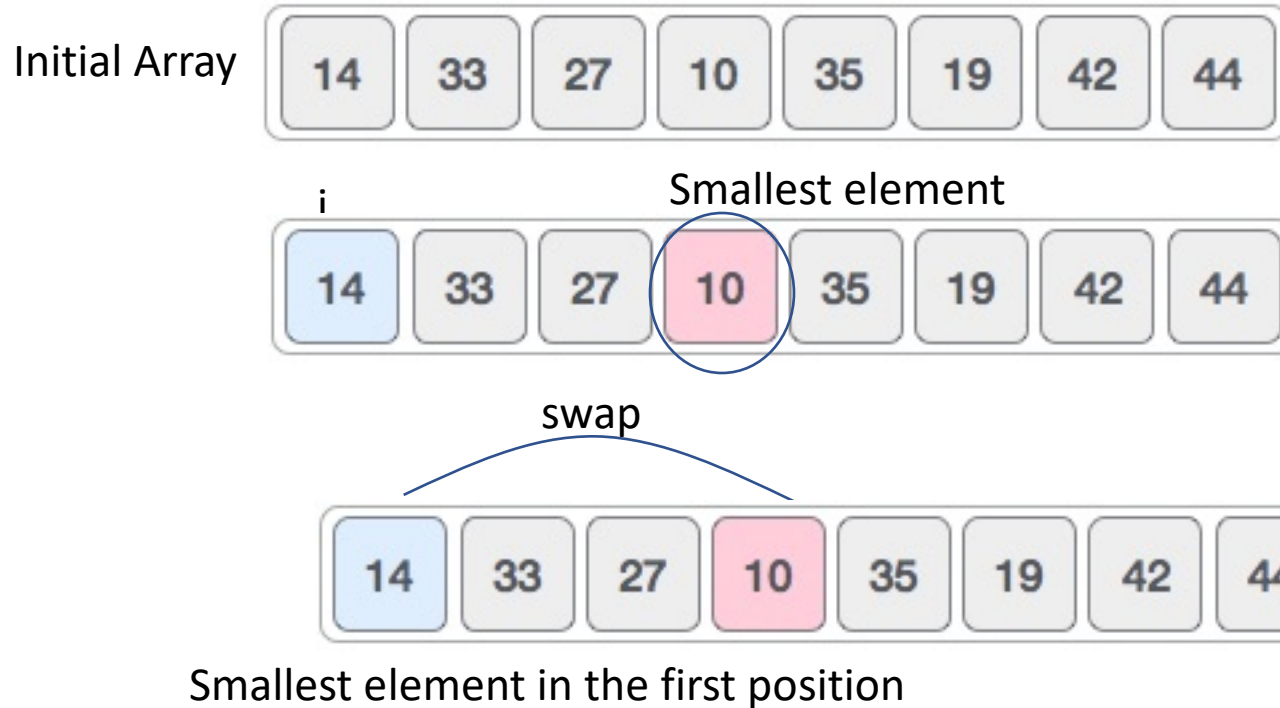
i = denotes position which is to be brought into the sorted array (initially the first element, as the entire array is unsorted)

Initial Array

| 14 | 33 | 27 | 10 | 35 | 19 | 42 | 44 |

i   Smallest element

| 14 | 33 | 27 | 10 | 35 | 19 | 42 | 44 |

swap

| 14 | 33 | 27 | 10 | 35 | 19 | 42 | 44 |

Smallest element in the first position

| 10 | 33 | 27 | 14 | 35 | 19 | 42 | 44 |

The first element is in it's correct spot

Next Step

i

Current Array

| 10 | 33 | 27 | 14 | 35 | 19 | 42 | 44 |

Smallest element

| 10 | 33 | 27 | 14 | 35 | 19 | 42 | 44 |

swap

| 10 | 33 | 27 | 14 | 35 | 19 | 42 | 44 |

| 10 | 14 | 27 | 33 | 35 | 19 | 42 | 44 |

The first two element is in it's correct spot

Continuing On…

Green = already sorted (numbers in correct position)
Blue = i[th] position
Red = smallest element, which swaps with the ith position element

The Selection Sort Algorithm

```
Selection Sort( A ):

 for i in range(len(A)-1):

    # Find the minimum element in remaining unsorted array
    min_id = i
    for j in range(i+1, len(A)):
        if A[min_id] > A[j]:
            min_id = j

    # Swap the found minimum element with the first element
    A[i], A[min_id] = A[min_id], A[i]
```

Why len(A) − 1?

Why len(A) here ?

Selection Sort Time Complexity

- **Best-Case Time Complexity**
  - **The scenario under which the algorithm will do the least amount of work (finish the fastest)**

- **Worst-Case Time Complexity**
  - **The scenario under which the algorithm will do the largest amount of work (finish the slowest)**

Selection Sort Time Complexity

- **Best-Case Time Complexity**
  - **Array is already sorted**
  - **Need N-1 iterations**
  - **(N-1) + (N-2) + …. + 1 = (N-1)* N / 2 comparisons**

Called Quadratic Time
$O(N^2)$
Order-of-N-square

Called Quadratic Time
$O(N^2)$
Order-of-N-square

- **Worst-Case Time Complexity**
  - **Array reverse sorted**
  - **Still need N-1 iterations**
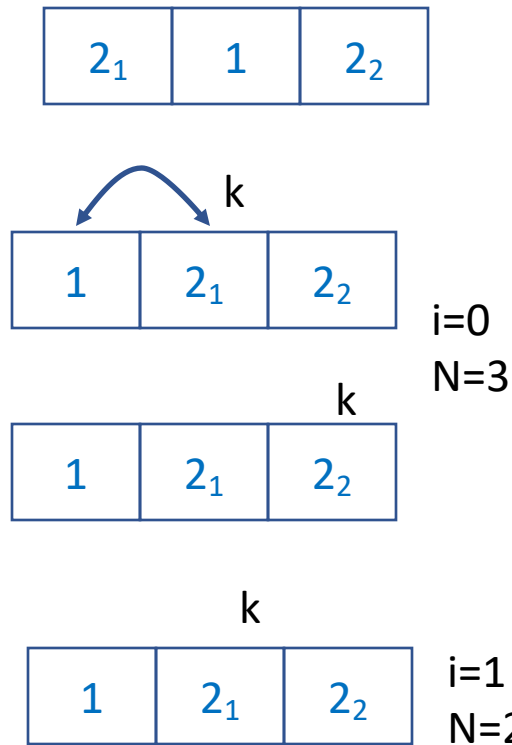 - **(N-1) + (N-2) + …. + 1 = (N-1)* N / 2**

Sorting Algorithm Additional Properties

- Memory Requirement:
  - **Denotes the amount of auxiliary storage** needed beyond that used by the list itself, under the same assumption.
    - No extra storage required. **Sorting in place.**

- Stability:
  - **Stable sorting algorithms sort repeated elements in the same order that they appear** in the **input**.
    - When sorting some kinds of data, only part of the data is examined when determining the sort order.
    - If two items compare as equal if one came before the other in the input, it will also come before the other in the output.
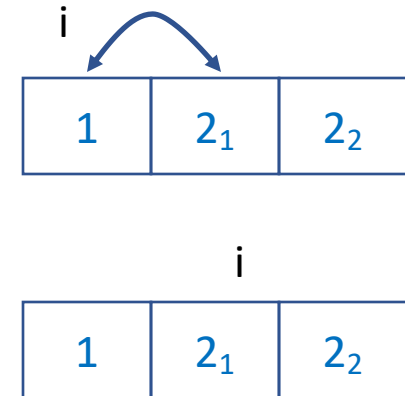
Stability Example

2_1 | 1 | 2_2

Insertion Sort

1 | 2_1 | 2_2 — i

1 | 2_1 | 2_2 — i

Bubble Sort

k

1 | 2_1 | 2_2 i=0 N=3

k

1 | 2_1 | 2_2

k

1 | 2_1 | 2_2 i=1 N=2

Selection Sort

i

1 | 2_1 | 2_2

i

1 | 2_1 | 2_2
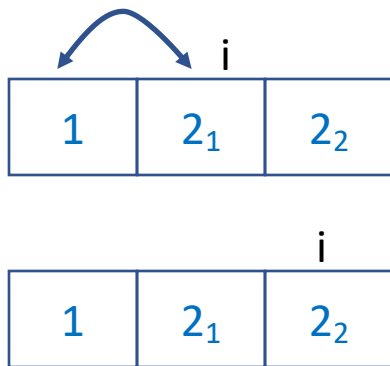
Memory Use and Stability of Basic Sorting Algorithms

- Insertion Sort
  - Memory = O(1) *# in place sorting (small amount of auxiliary space allowed, like to temp swap a variable)*
  - Stability = Yes
  - Best-case Running Time = O(n)
  - Worst-case Running Time = O(n^2)

- Bubble Sort
  - Memory = O(1) *# in place sorting*
  - Stability = Yes
  - Best-case Running Time = O(n^2)
  - Worst-case Running Time = O(n^2)

- Selection Sort
  - Memory = O(1) *# in place sorting*
  - Stability = Yes
  - Best-case Running Time = O(n^2)
  - Worst-case Running Time = O(n^2)

Exchange Sorts

- All three sorting algorithms seen thus far are examples of **Exchange sorting** algorithms
  - They swap (exchange) elements to sort

- Other algorithms exist that are faster that these for typical conditions

- So why are these algorithms slow?

General Slowness of Exchange Sort

- The crucial bottleneck is that only *adjacent* records are compared.
  - Thus, comparisons and moves (for Insertion and Bubble Sort) are by single steps.

- The cost of any exchange sort can be *at best* **the total number of inversions**
  - The number of elements with key value greater than the current record's key value appears before it that need to be moved.

Quiz

- How many **inversions** are in the following array?

| 14 | 5 | 81 | 92 | 101 | 7 |

5 inversions

| 4 | 5 | 1 | 9 | 11 | 12 |

2 inversions

| 63 | -51 | -9 | 17 | 1 | 0 |

8 inversions

N elements

Worst Case

| X_n | X_{n-1} | X_{n-2} | .... | 1 |

$1+2+3...+n-1 = n(n-1)/2$
$= O(n^2)$ inversions

$X_n > X_{n-1} > X_{n-2}....> 1$

Basic Sorting algorithms

- Insertion, selection and bubble sort have quadratic worst-case performance
  - That is $O(n^2)$

- The faster comparison based algorithm ?
  - $O(n*\log_2 n)$ or $O(n\lg n)$ complexity
  - REMEMBER: $O(n\lg n)$ more efficient than $O(n^2)$

- Examples Mergesort and Quicksort