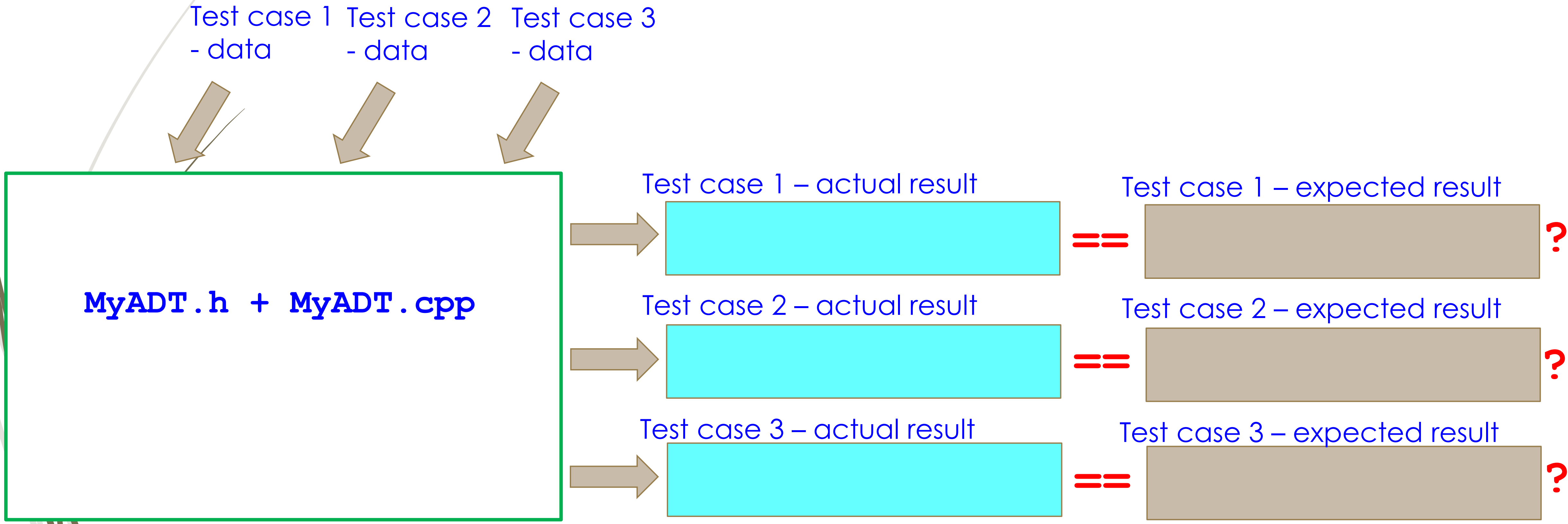
CMPT 225

Lecture 9 – Simple Sorting Algorithms

How are our Assignments marked?



Learning Outcomes

- At the end of the next few lectures, a student will be able to:
  - describe the behaviour of and implement simple sorting algorithms:
    - selection sort
    - insertion sort
  - describe the behaviour of and implement more efficient sorting algorithms:
    - quick sort
    - merge sort
  - analyze the best, worst, and average case running time (and space) of these sorting algorithms

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Last Lecture

- We saw how to ...
  - Describe Queue
  - Define public interface of Queue ADT
  - Design and implement Queue ADT using various data structures (CDTs)
  - Compare and contrast these various implementations using Big O notation
  - Give examples of real-life applications (problems) where we could use Queue to solve the problem
  - Solve problems using Queue ADT

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Today's menu

- Quick Review of Searching algorithms
- Simple sorting
  - Selection sort
  - Insertion sort
  - Analyze their best, worst, and average case running time and space efficiency of these sorting algorithms

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Review – Searching Algorithms

1. One of the worst case scenarios of the linear search algorithm would be: looking for **target** ______________ in this array

5 7 2 1 8 9 11 3 4 15 6

a. 15
b. 5
c. 9
d. 12
e. None of the above

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Review – Searching Algorithms

2. The best case scenario of the binary search algorithm would be: looking for **target** ______________ in the array

   2 5 8 9 11 23 24 35 56 78 89

a. 2
b. 23
c. 89
d. 100
e. None of the above

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Review – Searching Algorithms

3. A worst case scenario of the binary search algorithm would be: looking for **target** ______________ in the array

2 5 8 9 11 23 24 35 56 78 89

a. 2
b. 23
c. 89
d. 100
e. None of the above

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Review – Searching Algorithms

4. A worst case scenario of the binary search algorithm would be: looking for **target** _______________ in the array

5 7 2 1 8 9 11 3 4 15 6

a. 15
b. 5
c. 9
d. 12
e. None of the above

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Review – Searching Algorithms

5. Modify the binary search algorithm below such that it can quickly tell if **target** is not in the array, i.e., in **O(1)**?

```
PreCondition: data must be sorted

binarySearch(list, target)

  set position to value TARGET_NOT_FOUND
  set targetNotFound to value true

  if array not empty
  while targetNotFound AND have not looked or discarded every
                                                   element of array
      find middle element of array
      if middle element == target
         set position to position of target in original array
         set targetNotFound to false
      else
         if target < middle element
            array = first half of array
         else
            array = last half of array
  return position
```


Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Review – Searching Algorithms

6. How can we tweak the linear search algorithm such that it …
   - … finds the ***first*** occurrence of the target in the array?
   - … finds the ***last*** occurrence of the target in the array?
   - … finds ***all*** occurrences of the target in the array?
7. What else can it return aside from the target itself?

```
linearSearch(array, target)

  set result to value TARGET_NOT_FOUND
  set targetNotFound to value true

  if array not empty
    set currentElement to first element of array
    while targetNotFound AND have not looked at every element
                                                     of array
      if currentElement == target
        set result to current element
        set targetNotFound to false
      otherwise
        set currentElement to next element of array if there is one

  return result
```


Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time/Space Efficiency

- **Linear search**
  - Time efficiency of worst case scenario: $O(n)$
  - Space efficiency: $O(1)$
- **Binary search**
  - Time efficiency of worst case scenario: $O(\log_2 n)$
  - Space efficiency: $O(1)$

Binary Search vs Linear Search

- Advantages:
  1. Binary search is **much** faster than linear search especially when searching large data because it does not have to look at every element (at every iteration, it ignores ½ of data being searched).
- Disadvantages:
  1. A bit more complicated to implement and test.
  2. Data structure must be **sorted**.
     - Great if data is already sorted, but if this is not the case …
       - How much work does this sorting requires?

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Why Sorting?

- Definition: Process of placing elements in a particular sort order based on the value of a/some search key(s).
  - Ascending/descending sort order
- Why sorting?
  - Easier to deal with sorted data: easier to search (e.g. binary search)
  - Common operation but time consuming
- What can be sorted?
  - Internal data (data fits in memory)
  - External data (data that must reside on secondary storage)
- How to sort?

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Selection Sort

How Selection Sort works

- Initially, the array has **n** elements and the entire array is considered **unsorted**

- Start with first element (at index 0)
- Until the array is **sorted**
  1. Find (i.e., **select**) the smallest (or largest) element in the **unsorted section of array**
     - This is done by comparing one element with all other elements
  2. Swap it with the first element in the **unsorted section of array**
     - The **sorted section of array** has just grown by one element

So, the actual **sorting** is done when we **select** an element.

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Demo - Let's have a look at Selection Sort

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Selection Sort is an **in place** algorithm

- **in-place**: algorithm does not require additional space i.e., another array(s) aside from the original array
- Selection sort starts with an **unsorted array**
- The start: 
- As the array is being sorted, the **unsorted section** decreases and the **sorted section** increases:

- The end: 

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Selection Sort

Unsorted elements	Number of comparisons required to select either the smallest or the largest	Number of swapping
n	n-1	1
n-1	n-2	1
…	…	…
3	2	1
2	1	1
1	0	0
	$n(n$-1)/2	n-1

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Selection Sort

Note that comparing $n - 1$ times and iterating $n - 1$ time is the most amount of work selection sort does when sorting data

- In selection sort …
  1. … makes $n - 1$ comparisons
  2. … performs 1 swap (i.e., 3 assignments)

  $\Big\}$ Done in sequence

  -> **max[** $O(n - 1), O(1)$ **]** = **max[** $O(n), O(1)$ **]** = $O(n)$

- Then 1. and 2. are done in sequence $n - 1$ times

  -> $O(n - 1) * O(n) = O(n) * O(n) = O(n * n) = O(n^2)$

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Selection Sort

- Is $O(n^2)$ the time efficiency of the best, worst or average case scenario?
- Would the way the data is organized affect the number of operations selection sort perform (affect its time efficiency)?
  - For example:
    - If the data was already sorted (in the desired sort order, e.g., ascending)?
    - If the data was sorted but in the other sort order (e.g., descending)?
    - If the data was unsorted?
    - Let's check it out! ☺ https://www.toptal.com/developers/sorting-algorithms

Summary – Selection Sort Algorithm

- The way the data is organized **does not** affect the number of operations selection sort perform, i.e., does not affect its time efficiency
- Time efficiency
  - Best case scenario: $O(\boldsymbol{n}^2)$
  - Worst case scenario: $O(\boldsymbol{n}^2)$
  - Average case scenario: $O(\boldsymbol{n}^2)$
- Space efficiency
  - **in-place** sorting algorithm => $O(1)$

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Insertion Sort

How Insertion Sort works

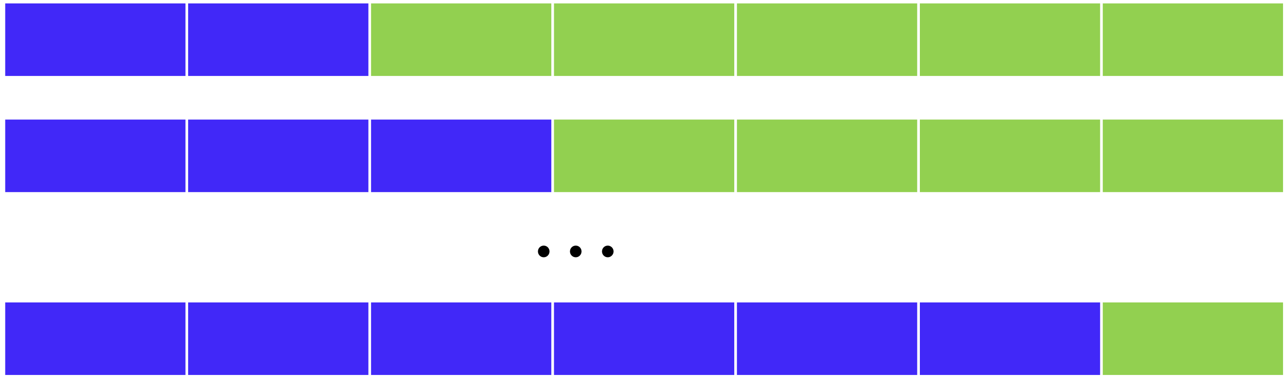
- Array has **n** elements
- At the start, insertion sort considers the first element of the array to be already sorted -> **sorted section** 
- Starts with the second element (at index 1)
- Repeat until the array is **sorted**
  1. Pick the first element of the **unsorted section** and store it in a `temp` variable
  2. Comparing it with each element of the **sorted section** (starting with the last element and moving towards the first element), looking for where in this **sorted section** it should be placed (i.e., in its proper sort order)
  3. Shift the elements in the **sorted section** up one position (starting with the last element of **sorted section** all the way down to the desired place) to make space for this element (if needed)
  4. **Insert** it in this newly vacated place in the **sorted section**

So, the actual **sorting** is done when we **insert** an element.

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Insertion Sort is an **in place** algorithm

- **in-place**: algorithm does not require additional space i.e., another array(s) aside from the original array
- Insertion sort starts with a **sorted section** of 1 element:
- The start: 
- As the array is being sorted, the **unsorted section** decreases and the **sorted section** increases:



- The end: 

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Demo - Let's have a look at Insertion Sort

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Insertion Sort

Number of elements in Sorted section	Worst case		Best case	
	Comparison	Shift	Comparison	Shift
1	1	1	1	0
2	2	2	1	0
…	…	…	1	0
n-1	n-1	n-1	1	0
	$n(n-1)/2$	$n(n-1)/2$	n-1	0

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Insertion Sort

- Time efficiency of insertion sort **is** affected by the way data is organized in the array to be sorted

- As we saw on the previous slide, the best case scenario
  - Requires a total of **n - 1** comparisons
  - Requires no shifting

- Activity:
  - How should the array be organized in order to achieve the best case scenario of the time efficiency of insertion sort?
  - Give an example of such array:

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Insertion Sort

- As we saw on the slide (two slides ago), in the worst case scenario
  - Every element in **sorted section of array** is compared with *the element currently being sorted*
  - Every element in **sorted section of array** has to be shifted to make space for *the element currently being sorted*

  - The outer loop runs ***n*-1** times
    - In the first iteration, one comparison and one shift is executed
    - …
    - In the last iteration, **n-1** comparisons and **n-1** shifts are executed

    Nested
  - For every element: (**n-1**) * (**n-1** comparisons and **n-1** shifts)

Nested: O((**n-1**) * max(O(**n-1**), O(**n-1**)))

= O((**n**) * max(O(**n**), O(**n**))

= O(**n**) * O(**n**)

= O(n^2)

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Time Efficiency Analysis of Insertion Sort

- Activity:
  - How should the array be organized in order to achieve the worst case scenario of the time efficiency of insertion sort?
  - Give an example of such array:

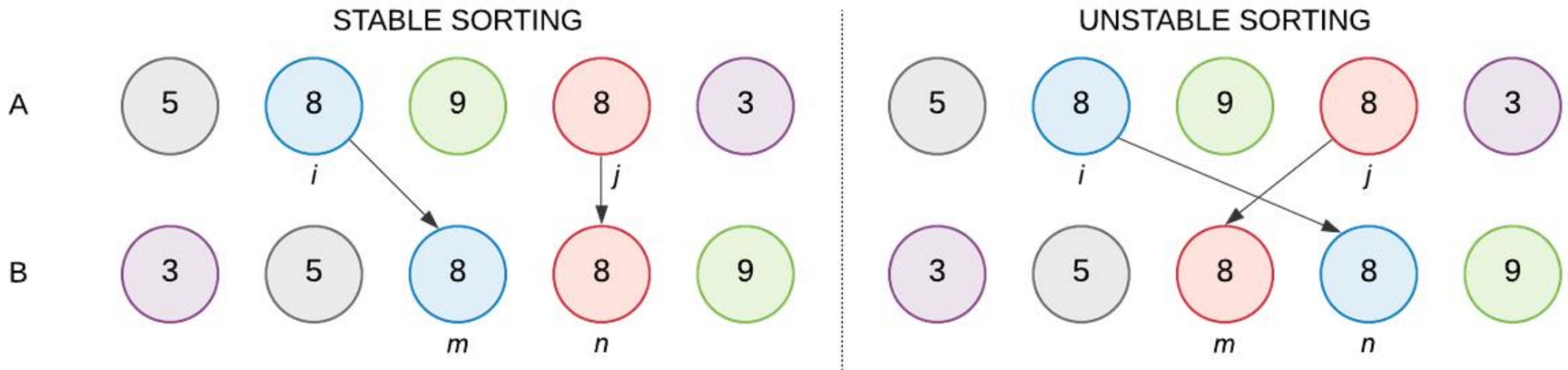
Summary – Insertion Sort Algorithm

- Time efficiency
  - Best case scenario: $O(\boldsymbol{n})$
  - Worst case scenario: $O(\boldsymbol{n}^2)$
  - Average case scenario: $O(\boldsymbol{n}^2)$
- Space efficiency
  - **in-place** sorting algorithm => $O(1)$

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Stability and Sorting Algorithm

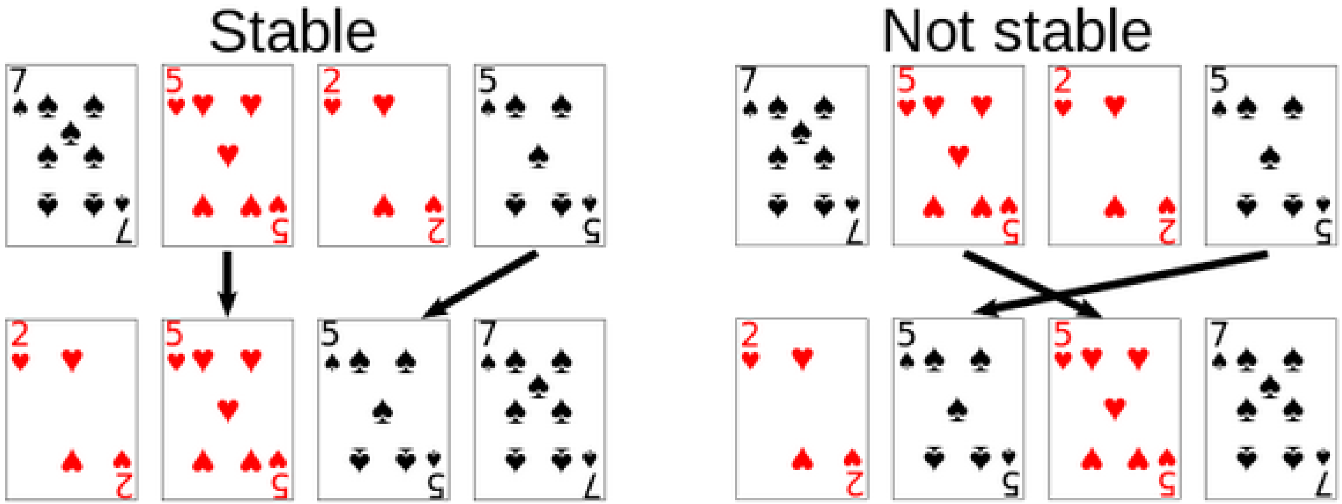
- The stability of a sorting algorithm is concerned with **how the algorithm treats equal (or duplicated) elements**.
- Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equals elements relative to each other.



Source: https://www.baeldung.com/cs/stable-sorting-algorithms

Stability - Example from wiki

- Here is an example of **stability of sorting algorithm** using playing cards. When the cards are sorted by rank using a **stable sorting algorithm**, the two 5's remain in the same order (in relation to each other) in the sorted output as they were in originally. When they are sorted using a **non-stable sorting algorithm**, the 5's may end up in the opposite order (in relation to each other) in the sorted output:



Source: https://en.wikipedia.org/wiki/Sorting_algorithm

When does stability matter?

- Data sorted using a **stable sort algorithm**:

BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Earl	B
Fabian	B
Gill	B
Dave	C

- Data sorted using an **unstable sort algorithm**:

BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Gill	B
Fabian	B
Earl	B
Dave	C

Source: https://www.freecodecamp.org/news/stability-in-sorting-algorithms-a-treatment-of-equality-fa3140a5a539/

Conclusion – Simple Sorting Algorithms

- Insertion sort versus Selection sort
  - Efficient: for small **n**'s
    - More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms
  - Stable: does not change the relative order of elements with equal keys
- Both sorting algorithms: Selection sort and Insertion sort
  - **In-place**: only requires a constant amount of additional memory space, i.e., O(1)
  - Both are from a class of sorting algorithm called **Comparison sort**

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

√ Learning Check

- Quick Review of Searching algorithms
- Simple sorting
  - Selection sort
  - Insertion sort
  - Analyze their best, worst, and average case running time and space efficiency of these sorting algorithms

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University

Next Lecture

- More efficient sorting algorithm -> **Quick sort**

Copyright © Anne Lavergne, School of Computing Science, Simon Fraser University