Simple Sorting

Objectives

- Analyze simple sorting algorithms
  - Selection sort
  - Insertion sort
- Assertions

Sorting

- The goal of sorting is to put a collection of items in order from smallest to largest
  - The order depends on the type of the items
    - Numerical value for numbers
    - Alphabetical order for strings
- Reasons for sorting
  - Sometimes useful in its own right
  - Also part of other algorithms

Simple Sorting

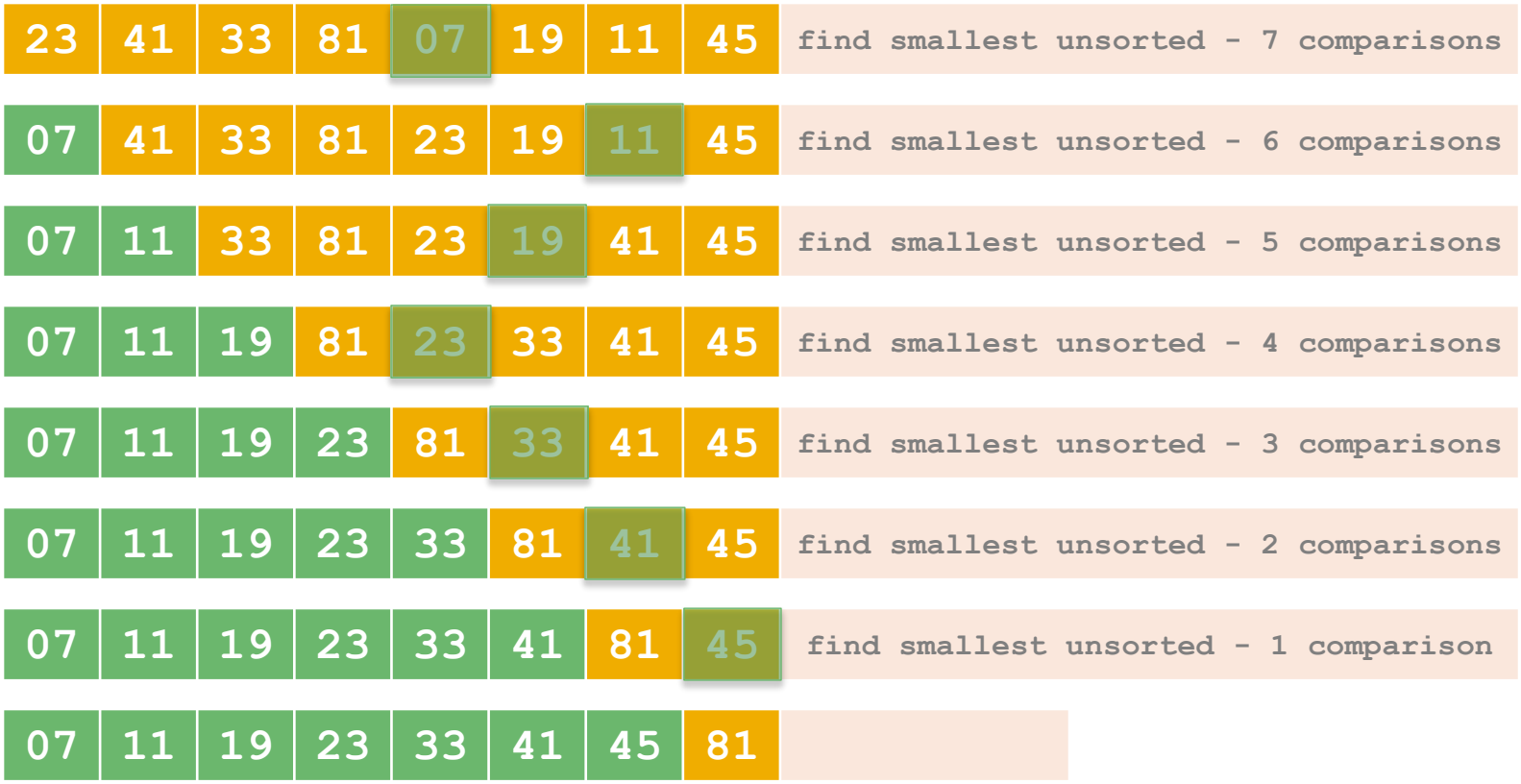
- As an example of algorithm analysis we will look at two simple sorting algorithms
  - Selection Sort and
  - Insertion Sort
- Calculate an approximate cost function for these two sorting algorithms
  - By analyzing how many operations are performed by each algorithm
  - This will include an analysis of how many times the algorithms' loops iterate

Selection Sort

Selection Sort

- Selection sort is a simple sorting algorithm that repeatedly finds the smallest item
  - The array is divided into a sorted part and an unsorted part
- Repeatedly swaps the first unsorted item with the smallest unsorted item
  - Starting with the element with index 0, and
  - Ending with last but one element (index $n - 1$)

Selection Sort



Selection Sort Algorithm

```
void selectionSort(int arr[], int n){
    repeat for all i from 0 to n-2

        find smallest, the index of the smallest
            element of arr [i … n-1]
        Algorithm: linear scan

        swap the smallest element with the current item
        arr[smallest] ↔ arr[i]

}
```

Selection Sort Algorithm

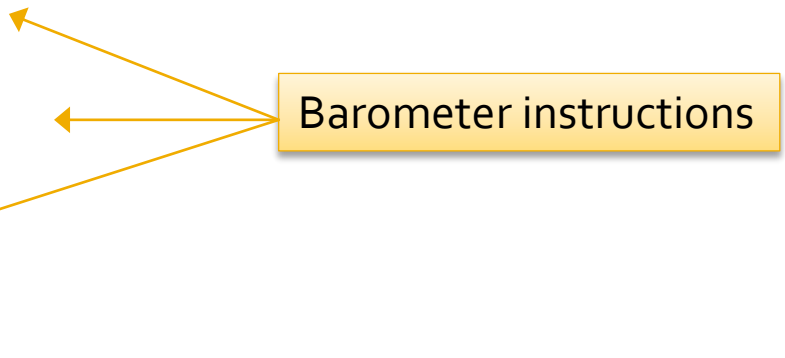
```
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = i;
        // Find the index of the smallest element
        for(int j = i + 1; j < n; ++j){
            if(arr[j] < arr[smallest]){
                smallest = j;
            }
        }
        // Swap the smallest with the current item
        temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
    }
}
```

outer loop
n-1 times

inner loop body
$n(n-1)/2$ times

Barometer Operation

- The barometer operation for selection sort must be in the inner loop
  - Since operations in the inner loop are executed the greatest number of times
- The inner loop contains four operations
  - Compare *j* to array length
  - Compare *arr*[*j*] to *smallest*
  - Change *smallest*
  - Increment *j*



Selection Sort Comparisons

Unsorted elements	Comparisons
n	n-1
n-1	n-2
…	…
3	2
2	1
1	0
	$n(n-1)/2$

Selection Sort Summary

- Ignoring the actual number of executable statements selection sort
  - Makes $n*(n-1)/2$ comparisons, regardless of the original order of the input
  - Performs $n-1$ swaps
- Neither of these operations are substantially affected by the organization of the input
- O Notation running time?

n*(n-1)/2 barometer instructions: | $O(n^2)$

Assertions

- An assertion is a statement that is expected to always be true at a particular point in a function
  - Expressed as a predicate
    - A function that returns true or false
- Assertions have two benefits
  - Help to reason about an algorithm
  - Assist in debugging

Selection Sort Assertion

- The first *i* elements of the input array always hold the smallest *i* elements in sorted order
  - If we can write an assertion and prove it holds throughout the algorithm we can prove the algorithm correct
  - Referred to as a loop invariant
- Loop invariant for selection sort
  - *arr*[0 … *i*-1] is sorted and
  - *arr*[*i*-1] < all elements in *arr*[*i* … *n*-1]

Assertions and Debugging

- Assertions can be inserted in code to assist with debugging
  - C has an assert statement that checks assertions and halts a program if the assertion fails
    - `assert(<condition>);`
- Be careful that assertions do not cause side-effects
  - i.e. change the data in the program

Selection Sort Analysis

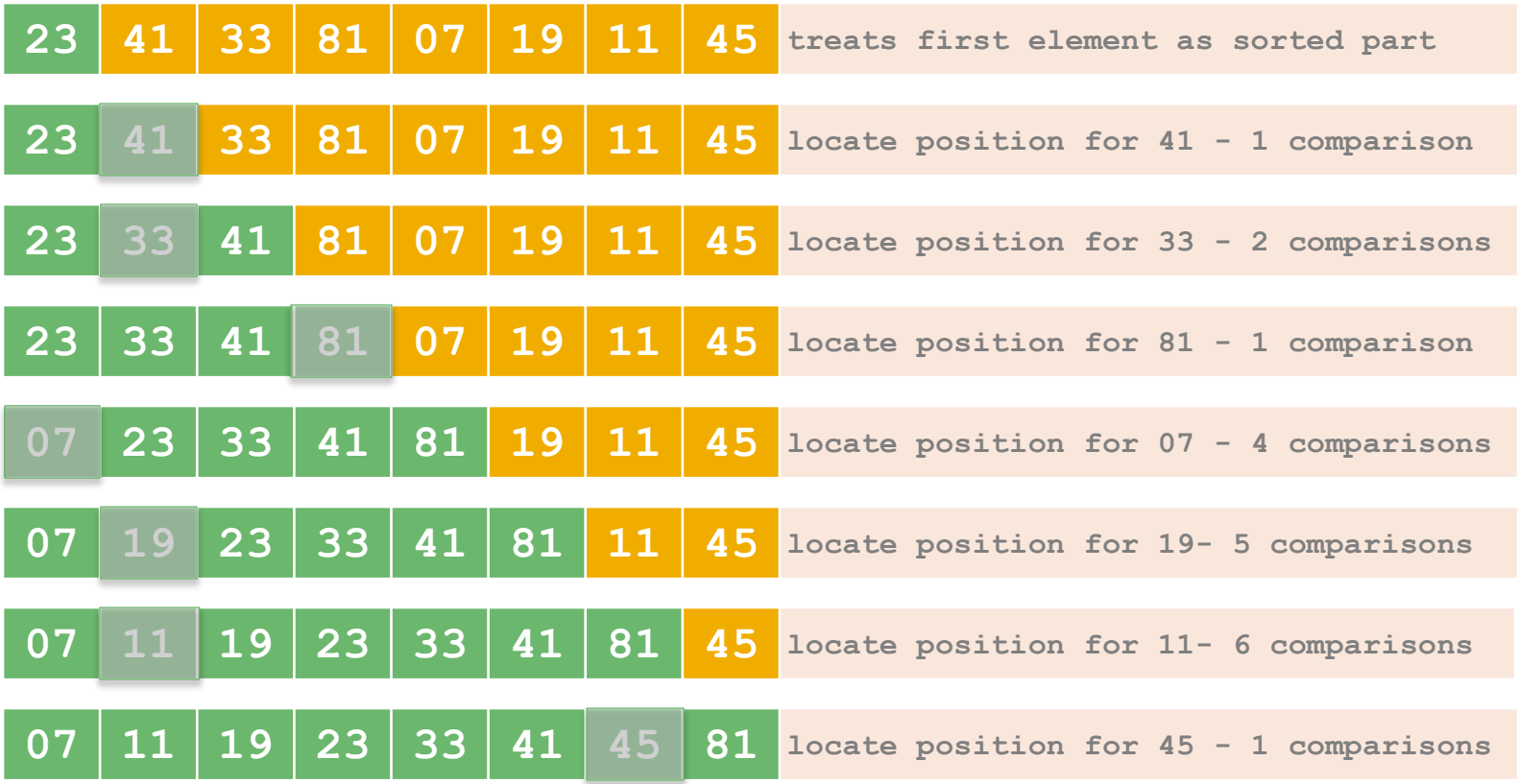
- Worst case running time
  - $O(n^2)$
- Average case running time
  - Not discussed in CMPT 125, but
  - The variations in running time of selection sort are small
  - Note that the number of times the for loops iterate is dependent only on the size of the input
    - Not the organization of the input

Insertion Sort

Insertion Sort

- Another simple sorting algorithm
  - Divides array into sorted and unsorted parts
- The sorted part of the array is expanded one element at a time
  - Find the correct place in the sorted part to place the 1st element of the unsorted part
    - By searching through all of the sorted elements
  - Move the elements after the insertion point up one position to make space

Insertion Sort



Insertion Sort Algorithm

```
void insertionSort(int arr[], int n){
    repeat for all i from 1 to n-1

        slide elements to the right to make space for
        the current element, arr[i]
        Algorithm:
            copy arr[i] to temp
            linear scan from right to left
            slide while temp < array element

        place new element in position

}
```

Insertion Sort Algorithm

```
void insertionSort(int arr[], int n){
    for(int i = 1; i < n; ++i){
        temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while(pos > 0 && arr[pos - 1] > temp){
            arr[pos] = arr[pos – 1];
            pos--;
        } //while
        // Insert the current item
        arr[pos] = temp;
    }
}
```

outer loop
n-1 times

inner loop body
how many times?

minimum: just the test for each outer loop iteration, *n*

maximum: *i* – 1 times for each iteration, *n* * (*n* – 1) / 2

Insertion Sort Cost

Sorted Elements	Worst-case Search	Worst-case Shuffle
0	0	0
1	1	1
2	2	2
…	…	…
n-1	n-1	n-1
	$n(n-1)/2$	$n(n-1)/2$

Insertion Sort Best Case

- The efficiency of insertion sort *is* affected by the state of the array to be sorted
- In the best case the array is already completely sorted!
  - No movement of array elements is required
  - Requires *n* comparisons

Insertion Sort Worst Case

- In the worst case the array is in reverse order
- Every item has to be moved all the way to the front of the array
  - The outer loop runs *n*-1 times
    - In the first iteration, one comparison and move
    - In the last iteration, *n*-1 comparisons and moves
    - On average, *n*/2 comparisons and moves
  - For a total of *n* * (*n*-1) / 2 comparisons and moves

Insertion Sort Analysis

- The cost of insertion sort is dependent on the organization of the input
  - Worst case is $O(n^2)$ when input is in reverse order
  - Best case is $O(n)$ when input is sorted
- Selection sort and insertion sort
  - Are both incremental sorts
  - Have the same worst case Big O running times
- There are better sorting algorithms