CS111
Introduction to Computer Science

Fall 2015

- Sorting
  - Insertion sort
  - Selection sort

Sorting

- If we keep our data sorted in an array we can use Binary search O(log n) instead of Linear search O(n) to find an item in the array
- How to keep the data sorted?
  - We'll learn two algorithms to sort arrays
    - Insertion sort
    - Selection sort

Insertion Sort

One way of thinking of inserting sort:

- Imagine you are playing a card game.
- The cards you are holding on your hand are sorted.
- The dealer hands you one card.
- You have to put it into the correct place so that the cards you're holding are still sorted

Insertion Sort: Insert into sorted array

- If we keep the cards in an array
  - sub-array from index 0 through index 6 is sorted
  - insert the element currently at index 7

0	1	2	3	4	5	6	7
2	5	6	7	10	11	12	8

sorted

  - To move 8 into index 3, shift elements 10, 11, and 12 right by one position

Insertion Sort: Insert into sorted array

0	1	2	3	4	5	6	7
2	5	6	7	10	11	12	8

8

0	1	2	3	4	5	6	7
2	5	6	7	10	11	12	12

8

0	1	2	3	4	5	6	7
2	5	6	7	10	11	11	12

8

0	1	2	3	4	5	6	7
2	5	6	7	10	10	11	12

8

0	1	2	3	4	5	6	7
2	5	6	7	8	10	11	12

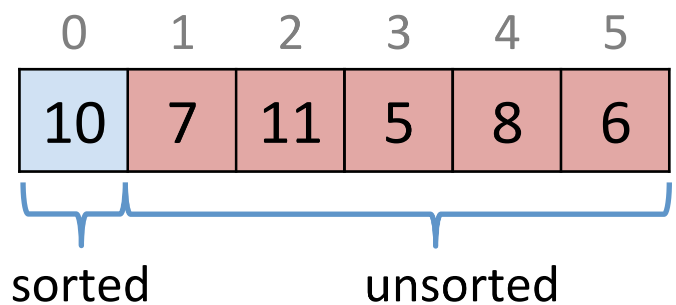
8

Compare 8 with each item from right to left until a smaller item is found

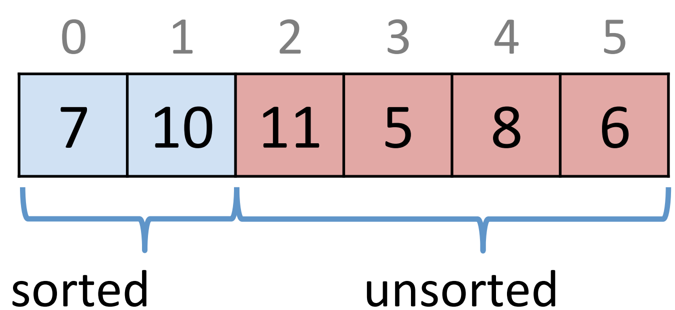
Insertion Sort: sort entire array

- Now you are given an unsorted array

  mark two regions sorted (only the first item) and unsorted (rest of the array)



  take the first element from the unsorted region and insert into the sorted region



Insertion Sort: sort entire array

- Do the same for the rest of the unsorted region
  - take the first element from the unsorted region and insert into the sorted region

0	1	2	3	4	5
7	10	11	5	8	6

sorted unsorted

0	1	2	3	4	5
7	10	11	5	8	6

0	1	2	3	4	5
5	7	10	11	8	6

0	1	2	3	4	5
5	7	8	10	11	6

0	1	2	3	4	5
5	6	7	8	10	11

Insertion Sort: Efficiency Analysis

```
void InsertionSort(int[] a, int n) {
    for (int i = 1; i < n; i++) {

        int itemToInsert = a[i]; // start of unsorted region
        int loc = i – 1; //end of sorted region

        while (loc >= 0 && a[loc] > itemToInsert) {
            a[loc+1] = a[loc];
            loc--;
        }
        a[loc+1] = itemToInsert;
    }
}
```

Basic Operation

Done *loc* times for each number

Insertion Sort: Efficiency Analysis

- Best case: whole array sorted

0	1	2	3	4	5
5	6	7	8	10	11

  - count the number of comparisons

1^{st} insertion	0 compares
2^{nd} insertion	1 compares
3^{rd} insertion	1 compares
n^{th} insertion	1 compares

  - $0+1+1+...+1 = n-1 = O(n)$

Insertion Sort: Efficiency Analysis

- Worst case: whole array unsorted
  - count the number of comparisons

1^{st} insertion	0 compares
2^{nd} insertion	1 compares
3^{rd} insertion	2 compares worst case
n^{th} insertion	n-1 compares worst case

$$-0+\underbrace{\boxed{1+2+3+\ldots+n-1}}_{\text{Arithmetic Series}} = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = O(n^2)$$

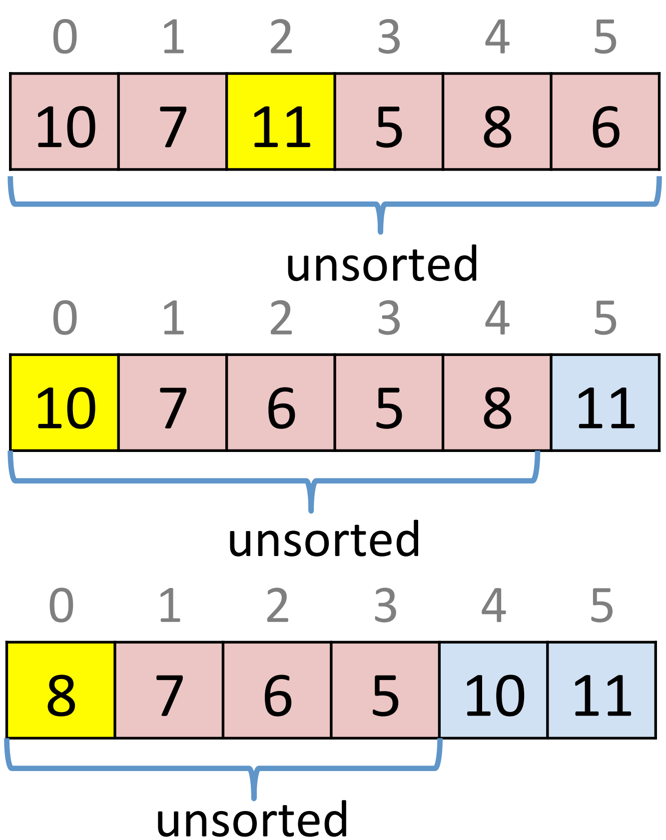
  - Plus the cost of moving data

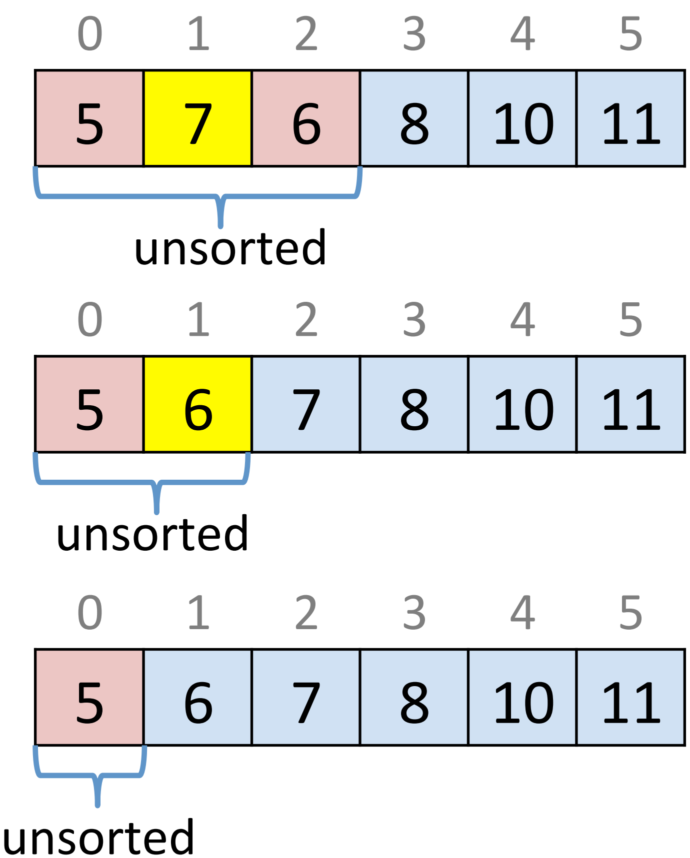
Selection Sort

- The second sorting algorithm we'll study
- The idea is to repeatedly find the biggest item in the array and move it to the end
  - to keep the array sorted in increasing order

Selection Sort

- Find the biggest item in the array and swap it with the last one





Selection Sort: Efficiency Analysis

```
void SelectionSort(int[] a, int n) {
    for (int i = n-1; i > 0; i--) {

        int maxLoc = 0; //Location of the largest value

        for (int j = 1; j <= i; j++) {
            if (a[j] > a[maxLoc]) {
                maxLoc = j;
            }
        }
        int temp = a[maxLoc];
        a[maxLoc] = a[i];
        a[i] = temp;
    }
}
```

Done *i-1* times for each number

Basic Operation

Selection Sort: Efficiency Analysis

- Worst case and Best case are the same
  - count number of comparisons to find

1^{st} largest	n-1 compares
2^{nd} largest	n-2 compares
3^{rd} largest	n-3 compares
n^{th} largest	0 compares

$$-0+\underbrace{\boxed{1+2+3+\ldots+n-1}}_{\text{Arithmetic Series}}=\sum_{i=1}^{n-1} i=\frac{(n-1)((n-1)+1)}{2}=O(n^2)$$

Sorting Algorithms

- Insertion sort $O(n^2)$
  - extra assignment for moving data
  - best if data is partially sorted

- Selection sort $O(n^2)$
  - best case the same as the worst case