# Sorting Algorithms

## Rearranging data into a particular order

Alwin Tareen

# Selection Sort

## The “search and swap” algorithm

- Sorting algorithms take data in an array and rearrange it into a particular order.
- The selection sort algorithm is commonly known as a “search and swap” algorithm, due to its specific behavior.
- It works by selecting the smallest unsorted item remaining in the array, and then swapping it with the item in the next position to be filled.

# Selection Sort

## A description of the selection sort algorithm

- Similar to a linear search, selection sort will first loop through the array, and look for the lowest value.
- Once it has found the lowest value, it will swap this element with the element at index 0.
- Now, the first element is sorted.

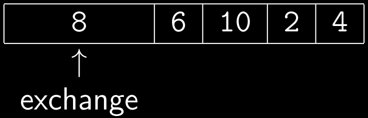
# Selection Sort

## A description of the selection sort algorithm, continued

- Then, the process repeats with the element at index 1.
- Starting from this position, it will search for the lowest value in the rest of the array.
- Once the lowest value has been found, this element is swapped with the element at index 1.
- Now, the first two elements are sorted.
- This process is repeated until the end of the array is reached, and all the elements are sorted.
- Note that if the lowest value is already in its correct position during the search, it will stay there.

# Running the Selection Sort Algorithm

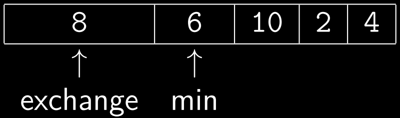
- Consider the following array. We want to sort it from smallest to largest.
- Remember, our strategy is to first find the smallest element in the array, and place it in the first position.



| 8 | 6 | 10 | 2 | 4 |
|---|---|---|---|---|

↑
exchange

- The first element we consider is the 6. It is smaller than 8, so it becomes the new minimum value.

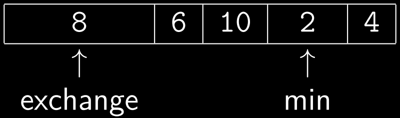


| 8 | 6 | 10 | 2 | 4 |
|---|---|---|---|---|

↑ exchange ↑ min

# Running the Selection Sort Algorithm

- Then, compare the 10 to the 6. The 10 is larger, so we move on to the next element.
- Compare the 2 to the 6. The 2 is smaller, so it becomes our new minimum.



| | | | | |
|---|---|---|---|---|
| 8 | 6 | 10 | 2 | 4 |

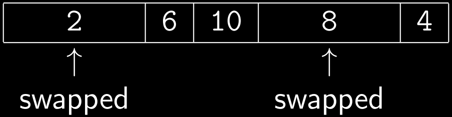
exchange

min

- Then compare the 4 to the 2. The 4 is larger, so 2 is still the minimum.

# Running the Selection Sort Algorithm

- We have reached the end of the array, so we must swap the 2 and the 8.

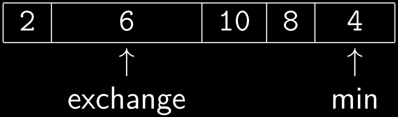


|   |   |    |   |   |
|---|---|----|---|---|
| 2 | 6 | 10 | 8 | 4 |

swapped

swapped

- The first element is now sorted.
- We repeat this process, starting with the second element.

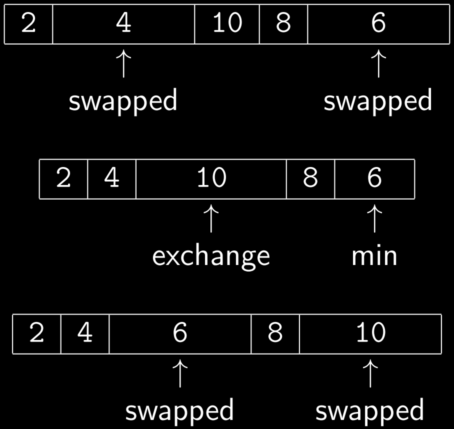


|   |   |    |   |   |
|---|---|----|---|---|
| 2 | 6 | 10 | 8 | 4 |

exchange

min

# Running the Selection Sort Algorithm



| 2 | 4 | 10 | 8 | 6 |
|---|---|---|---|---|
|  | ↑<br>swapped |  |  | ↑<br>swapped |

| 2 | 4 | 10 | 8 | 6 |
|---|---|---|---|---|
|  |  | ↑<br>exchange |  | ↑<br>min |

| 2 | 4 | 6 | 8 | 10 |
|---|---|---|---|---|
|  |  | ↑<br>swapped |  | ↑<br>swapped |

# Running the Selection Sort Algorithm

- Once we reach the end of the array, we see that it is sorted.

| | | | | |
|---|---|---|---|---|
| 2 | 4 | 6 | 8 | 10 |

# The SelectionSort Class

```
public class SelectionSort
{
    public static void selectionSort(int[] arr)
    {
        int min = 0;
        int temp = 0;
        for (int i = 0; i < arr.length-1; i++)
        {
            min = i;
            for (int j = i+1; j < arr.length; j++)
            {
                if (arr[j] < arr[min])
                {
                    min = j;
                }
            }
            temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }
```

# The SelectionSort Class, Continued

```
    public static void main(String[] args)
    {
        int[] values = {12, 3, 8, 7, 9, 1, 23, 18};
        selectionSort(values);
        for (int item : values)
        {
            System.out.print(item + " ");
        }
    }
}
```

# Insertion Sort

## Applying a strategy of partial sortedness

- The array of elements is separated into two parts: a partially sorted part, and an unsorted part.
- Partially sorted means that the elements are sorted amongst themselves.
- However, the elements aren't necessarily in their final resting positions, because they may still need to be moved, when other elements are inserted between them.

# Insertion Sort

## A description of the insertion sort algorithm

- The algorithm begins with the first element as the partially sorted section, the second element as the item under consideration, and the rest of the array as the unsorted section.
- The goal is to insert the item under consideration into the appropriate place in the partially sorted group.
- To achieve this goal, we may need to shift some elements to the right, to make room for the insert.
- To provide a space for this shift, we place the item under consideration into a temporary variable. This leaves an empty space in the array.

# Insertion Sort

## A description of the insertion sort algorithm

- Then, we compare the item under consideration to its left-hand neighbor in the partially sorted group.
- If this neighbor is larger, it gets shifted to the right, and the item under consideration is inserted into index 0.
- If this neighbor is smaller, then the item under consideration is placed back where it was.
- Now, the partially sorted group contains two elements.

# Insertion Sort

## A description of the insertion sort algorithm

- In general, you keep shifting partially sorted elements to the right, until you find the proper position for the item under consideration, and you insert the item at that spot.
- This process is repeated until all the unsorted items have been inserted.

# The InsertionSort Class

```java
public class InsertionSort
{
    public static void insertionSort(int[] arr)
    {
        int j = 0;
        int index = 0;
        for (int i = 1; i < arr.length; i++)
        {
            index = arr[i];
            j = i;
            while ((j > 0) && arr[j-1] > index)
            {
                arr[j] = arr[j-1];
                j = j - 1;
            }
            arr[j] = index;
        }
    }
```

# The InsertionSort Class, Continued

```
    public static void main(String[] args)
    {
        int[] values = {12, 3, 8, 7, 9, 1, 23, 18};
        insertionSort(values);
        for (int item : values)
        {
            System.out.print(item + " ");
        }
    }
}
```

# MergeSort

## A “divide and conquer” algorithm

- MergeSort is a good example of the **divide and conquer** principle.

## Generally, we do the following:

- Break the problem into smaller sub-problems of the same type.
- Solve those sub-problems recursively.
- Combine the solutions found for the individual sub-problems into a solution for the entire problem.

# MergeSort

## The divide and conquer principle

- Divide the problem size into more comprehensible pieces.
- Conquer, or resolve the smaller pieces recursively.
- Combine, or put the pieces back together to create the final solution.

# MergeSort

## A description of the MergeSort algorithm

- **Divide step:** divide the array in half.
- **Conquer step:** sort each half of the array. Note that the base case of an one-element array is considered sorted.
- **Combine step:** merge the two halves into a single sorted array.

## A memory tradeoff

- Note that a disadvantage of the MergeSort is the need of a temporary array, similar in size to the one being sorted.
- This means that MergeSort requires more memory than the other sorts.

# Running the MergeSort Algorithm

- We begin with an array of size 8:

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 64 | 21 | 33 | 70 | 12 | 85 | 44 | 3 |

- Divide this array in half:

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 64 | 21 | 33 | 70 |  | 12 | 85 | 44 | 3 |

- Divide each subarray in half:

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 64 | 21 |  | 33 | 70 |  | 12 | 85 |  | 44 | 3 |

- Then, divide each of those subarrays in half:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 64 |  | 21 |  | 33 |  | 70 |  | 12 |  | 85 |  | 44 |  | 3 |

# Running the MergeSort Algorithm

- We cannot divide the subarrays any further. We have arrived at the base case, where it is assumed that an array with one element can be considered sorted.
- Now, we must apply the **combine** step.
- Each of the one-element arrays are merged into a sorted array of two elements.
- The smaller element is placed into the merged array first, then the larger element goes in.

| | |
|---|---|
| 21 | 64 |

| | |
|---|---|
| 33 | 70 |

| | |
|---|---|
| 12 | 85 |

| | |
|---|---|
| 3 | 44 |

# Running the MergeSort Algorithm

- Each of these two-element arrays are merged into a four-element array, using the same technique.

| | | | |
|---|---|---|---|
| 21 | 33 | 64 | 70 |

| | | | |
|---|---|---|---|
| 3 | 12 | 44 | 85 |

- Finally, these four-element arrays are merged into a single eight-element array, and the entire array is now sorted.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 12 | 21 | 33 | 44 | 64 | 70 | 85 |

## Analysis of the MergeSort algorithm

- Given an array of size `n`, the average runtime is: $n \log_2 n$

# Sorting Algorithms: End of Notes