



WCOM125/COMP125

Fundamentals of Computer Science

Lecture: Sorting

Gaurav Gupta

February 2, 2018

Contents

1 Overview	4
1.1 Why is sorting important?	4
2 Selection Sort	6
2.1 Selection Sort sample pseudo-code	7
2.2 Selection Sort sample source code	8
2.3 Sorting array of objects	10
2.4 Variations to sorting algorithm	11
3 Insertion Sort	13
3.1 Insertion sort sample pseudo-code	14
3.2 Insertion Sort sample source code	15
3.3 Insertion sort sample source code	16
4 Sample solutions for exercises	17

List of exercises

Exercise 1 <i>Trace selection sort execution</i>	10
Exercise 2 <i>Trace insertion sort execution</i>	16



Figure 1: Source: Nancy Hebert, licensed for reuse. https://c1.staticflickr.com/4/3153/2328295047_86f0fe9710.jpg

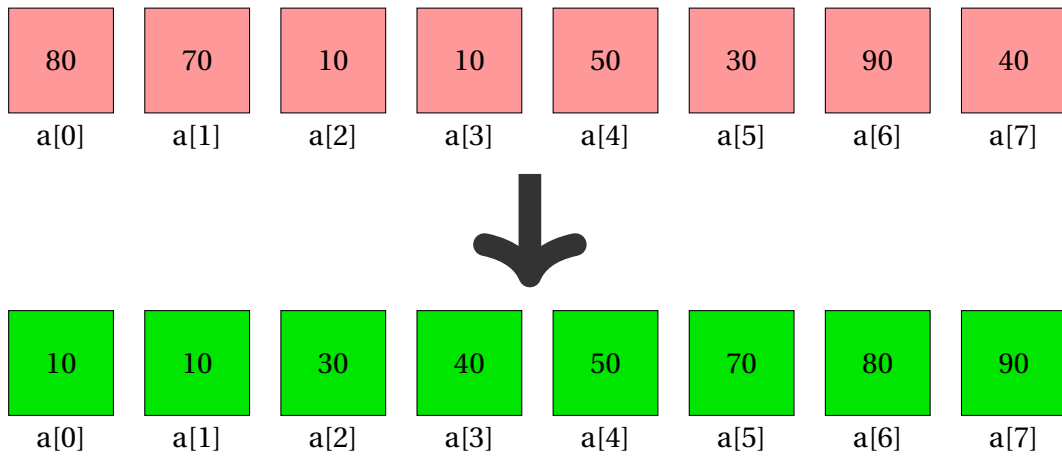
1 Overview

We explore the two standard sorting algorithms - insertion, and selection, along with analysis of the two. In addition, we also take a less detailed look at merge sort.

1.1 Why is sorting important?

Sorting a collection makes it easy to analyse data. Several tasks are made simpler if a collection is sorted, such as:

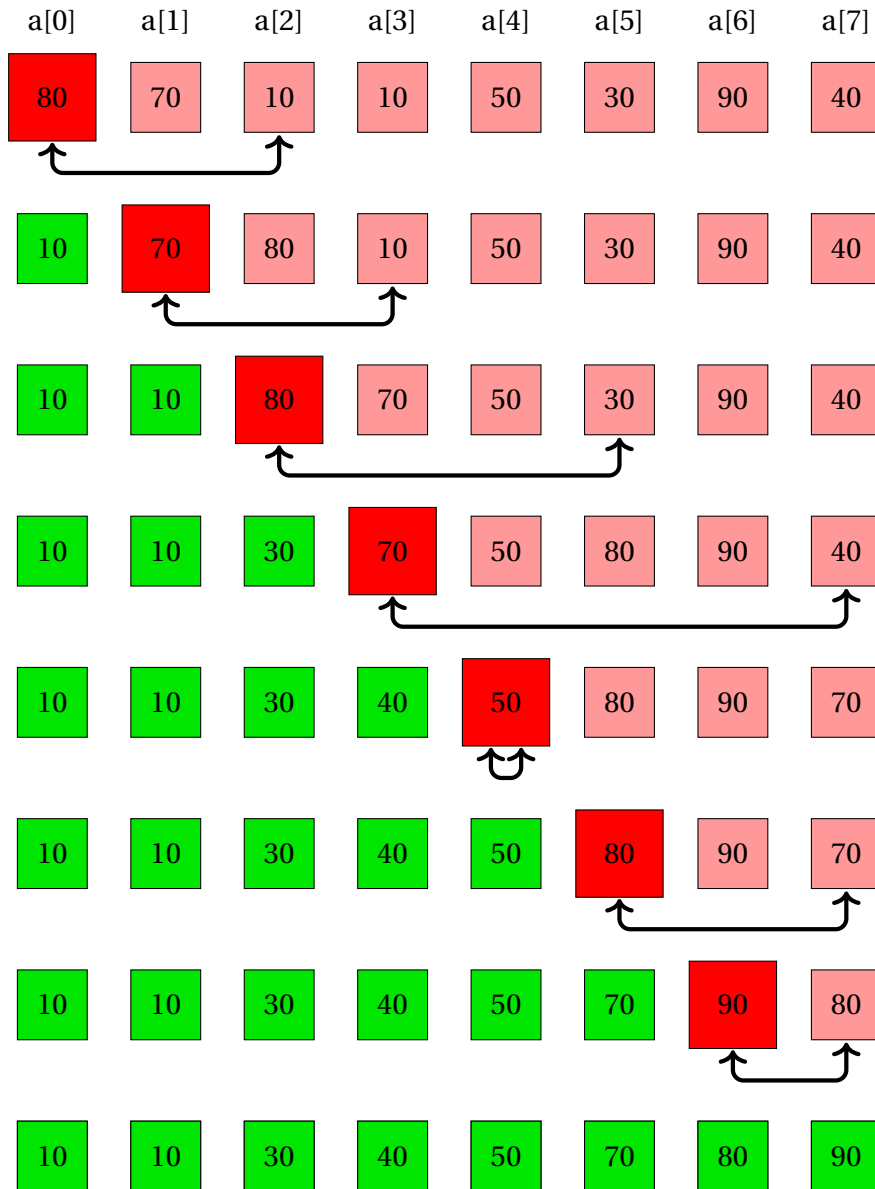
- finding the lowest value (first value in a collection sorted in ascending order)
- finding the highest value (last value in a collection sorted in ascending order)
- finding the median value (item at `arr.length/2`)
- checking if the array contains any negative value (it does if the lowest value is less than zero)
- checking if the array contains only negative value (it does if the highest value is less than zero)
- faster search (using binary search algorithm)



2 Selection Sort

The principle behind selection sort is:

Swap the smallest item in the unsorted part of the array with the first item of the unsorted part of the array



2.1 Selection Sort sample pseudo-code

```
Parameter(s): int[] arr  
Return: none (array is sorted in ascending order at the end)  
1 set i to 0;  
2 while i is not the last index do  
3   set minIndex to i;  
4   set k to i+1 ;  
5   while k is a valid index do  
6     if item at index k < item at index minIndex then  
7       | update minIndex to hold value of k ;  
8     end  
9   end  
10  swap item at index minIndex with item at index i ;  
11 end
```

Algorithm 1: Selection Sort

2.2 Selection Sort sample source code

```
1 //helper 1
2 public static void swap(int[] a, int idx1, int idx2) {
3     if(a == null) nothing to do
4         return;
5     if(idx1 < 0 || idx1 >= a.length) //invalid index 1
6         return;
7     if(idx2 < 0 || idx2 >= a.length) //invalid index 2
8         return;
9     int temp = a[idx1];
10    a[idx1] = a[idx2];
11    a[idx2] = temp;
12 }
13
14 //helper 2
15 public static int indexSmallestItem(int[] a, int start) {
16     if(a == null)
17         return -1; //error code
18     if(start < 0 || start >= a.length) //invalid index
19         return -1;
20     int result = start;
21     for(int k=start+1; k < a.length; k++) {
22         if(a[k] < a[result]) {
23             result = k;
24         }
25     }
26     return result;
27 }
28
29 //sorting method
30 public static void selectionSort(int[] arr) {
31     if(arr == null) //nothing to do
32         return;
33     for(int i=0; i < arr.length - 1; i++) {
34         int minIndex = indexSmallestItem(arr, i);
35         swap(arr, i, minIndex);
36     }
37 }
```


The helpers can be written inline as well, with which the method is:

```
1      //sorting method
2  public static void selectionSort(int[] arr) {
3      if(arr == null) //nothing to do
4          return;
5      for(int i=0; i < arr.length - 1; i++) {
6          int minIndex = i;
7          for(int k=i+1; k < arr.length; k++) {
8              if(arr[k] < arr[minIndex]) {
9                  minIndex = k;
10             }
11         }
12         int temp = arr[i];
13         arr[i] = arr[minIndex];
14         arr[minIndex] = temp;
15     }
16 }
```

Exercise 1 Trace selection sort execution

Trace the status of the array at the end of each iteration of the loop controlled by variable **i** in selection sort for the following cases:

1. **arr = {4, 3, 6, 5, 2, 1}**

2. **arr = {1, 8, 2, 7, 3, 6}**

(To see the answer click here: [1](#))

Write your answer here

2.3 Sorting array of objects

Since objects cannot be compared using the primitive comparison operators ($>$, $<$, \geq , \leq), we must use the method **compareTo** to compare them.

Essentially,

```
1 obj1 < obj2
2 //is same as
3 obj1.compareTo(obj2) == -1
```

Similarly,

```
1 obj1 > obj2
2 //is same as
3 obj1.compareTo(obj2) == 1
```

The only two statements in the sorted algorithm that are affected are:

```
1 if(arr[k] < arr[minIndex]) //on line 8
2 int temp = arr[i]; //on line 12
```

The sorting algorithm applied on array of objects changes to:

```
1 //sorting method
2 public static void selectionSort(Circle[] arr) {
3     if(arr == null) //nothing to do
4         return;
5     for(int i=0; i < arr.length - 1; i++) {
6         int minIndex = i;
7         for(int k=i+1; k < arr.length; k++) {
8             if(arr[k].compareTo(arr[minIndex]) == -1) {
9                 minIndex = k;
10            }
11        }
12        Circle temp = arr[i];
13        arr[i] = arr[minIndex];
14        arr[minIndex] = temp;
15    }
16 }
```

2.4 Variations to sorting algorithm

Sometimes, the basis of sorting might be a bit more complex than simple numerical comparison. For example, I might want to sort an array of integers in ascending order of number of divisors. For example, if the array is **{14, 5202, 12, 121, 36}**, the different states of the array sorted on different criteria are below:

1. Based on numerical value: **{12, 14, 36, 121, 5202}**
2. Based on number of digits: **{14, 12, 36, 121, 5202}**
3. Based on number of divisors: **{121, 5202, 14, 12, 36}**

This **only** affects the comparison statement.

In each of the above situations, the comparison statements would be:

1. Based on numerical value:

```
1 if(arr[k] < arr[minIndex])
```

2. Based on number of digits:

```
1 if(nDigits(arr[k]) < nDigits(arr[minIndex]))
```

3. Based on number of divisors:

```
1 if(nDivisors(arr[k]) < nDivisors(arr[minIndex]))
```

In general, when comparing on a function of the items of the array, we should be using,

```
1 if(someFunction(arr[k]) < someFunction(arr[minIndex]))
```

Where the simplest function is the identity function (the item itself), reducing the statement to,

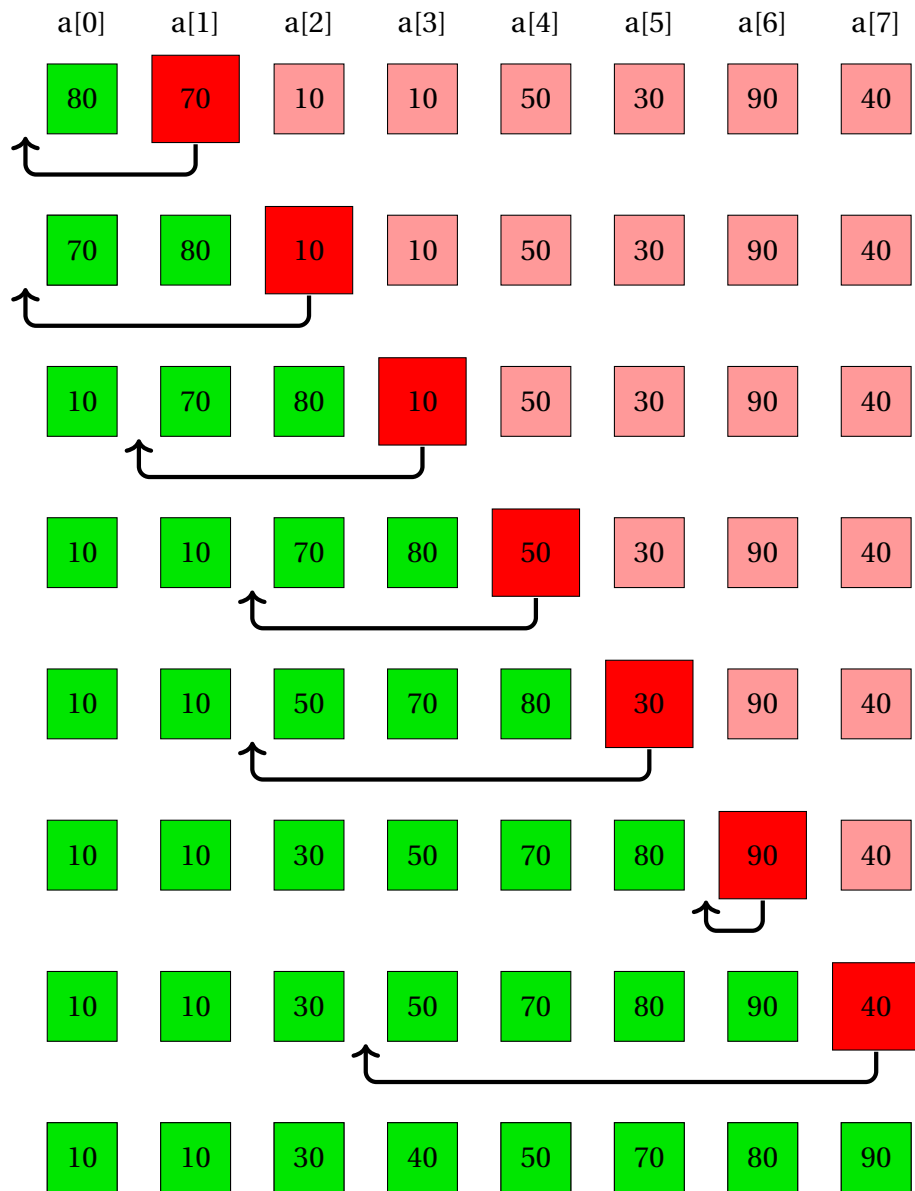
```
1 if(arr[k] < arr[minIndex])
```

3 Insertion Sort

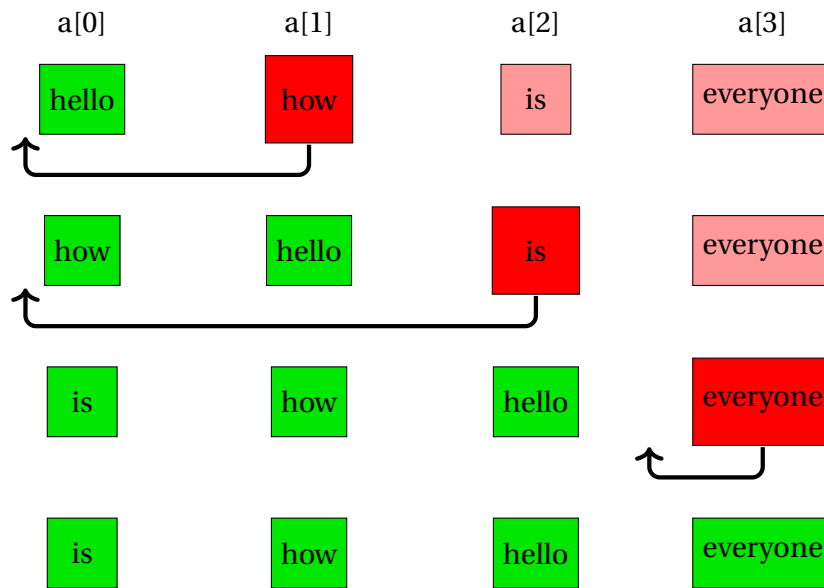
The principle behind insertion sort is:

Put the first item of the unsorted part at the right place in the sorted part.

Example 1



Example 2



3.1 Insertion sort sample pseudo-code

```
Parameter(s): int[] arr = {arr[0], ..., arr[arr.length - 1]}  
Return: none (array is sorted in ascending order at the end)  
1 set i to 1;  
2 while i is a valid index do  
3   set backup to arr[i] ;  
4   set k to i-1 ;  
5   while k ≥ 0 and arr[k] > backup do  
6     set arr[k+1] to arr[k] ;  
7     k = k - 1 ;  
8   end  
9   set arr[k+1] to backup ;  
10 end
```

Algorithm 2: Insertion Sort

3.2 Insertion Sort sample source code

```
1 public static void insertionSort(int[] a) {  
2     if(a == null)  
3         return;  
4     for(int i=1; i < a.length; i++) {  
5         //keep a copy of item to insert into  
6         //its rightful place  
7         int backup = a[i];  
8         int k = i - 1;  
9         //all higher items  
10        while(k >= 0 && a[k] > backup) {  
11            //must be pushed towards  
12            //the back of the array  
13            a[k+1] = a[k];  
14            k--;  
15        }  
16        //copy backup into its rightful place  
17        a[k+1] = backup;  
18    }  
19 }
```

3.3 Insertion sort sample source code

Exercise 2 Trace insertion sort execution

Trace the status of the array at the end of each iteration of the loop controlled by variable **i** in insertion sort for the following cases:

1. **arr = {4, 3, 6, 5, 2, 1}**
2. **arr = {1, 8, 2, 7, 3, 6}**

(To see the answer click here: [2](#))

Write your answer here

4 Sample solutions for exercises

Solution: Exercise 1

1. 4 3 6 5 2 1
1 3 6 5 2 4
1 2 6 5 3 4
1 2 3 5 6 4
1 2 3 4 6 5
1 2 3 4 5 6
2. 1 8 2 7 3 6
1 8 2 7 3 6 (no change as smallest item already at front)
1 2 8 7 3 6
1 2 3 7 8 6
1 2 3 6 8 7
1 2 3 6 7 8

Solution: Exercise 2

1. 4 3 6 5 2 1
3 4 6 5 2 1
3 4 6 5 2 1
3 4 5 6 2 1
2 3 4 5 6 1
1 2 3 4 5 6
2. 1 8 2 7 3 6
1 8 2 7 3 6
1 2 8 7 3 6
1 2 7 8 3 6
1 2 3 7 8 6
1 2 3 6 7 8