

CS 162FZ: Introduction to Computer Science II

Lecture 09

Sorting and Searching

Dr. Chun-Yang Zhang

Introduction

Sorting is an everyday task of arranging items in a particular format. Common examples of sorting include:

Arranging the contacts in your phone

Arranging the money in your wallet in order

Arranging the cards in order in a deck of cards

Arranging your family in order of ages

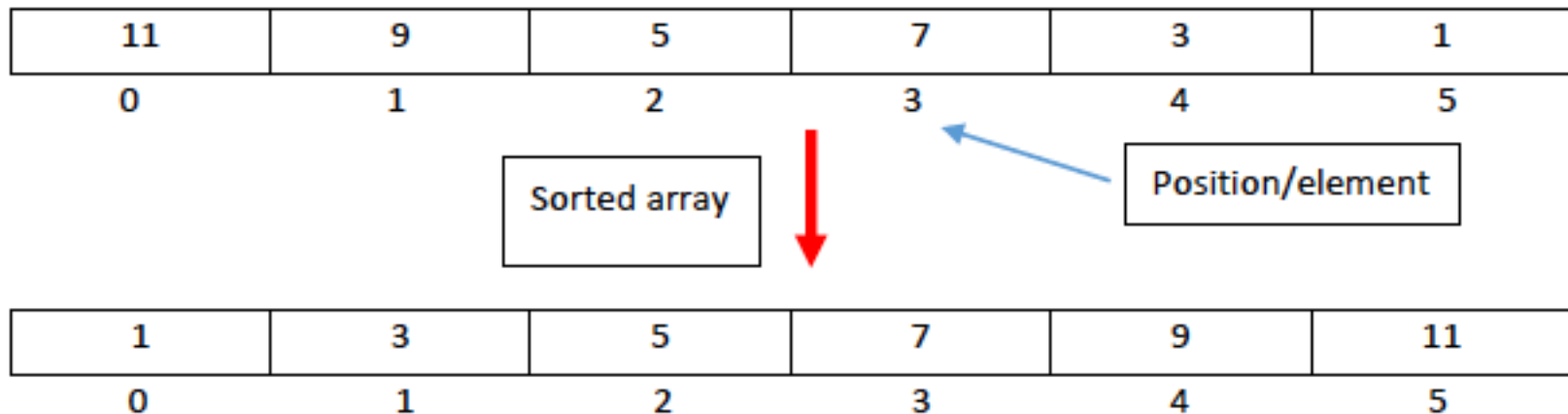
Arranging students in a class by their student number

There are many techniques used to sort data. Sorting techniques mainly depend on two factors.

- The first is the length of the **execution time** of a program
- The second is the **space** required to execute the program.

Sorting and Array

- When dealing with arrays it can be beneficial to sort them into a particular order before working with them. This sorting could be from smallest to biggest value or biggest to smallest value. When sorting we take an **unordered list** and put it in a **particular order**.
- As an example, consider the following array and its sorted version below:



Bubble Sort

- There are many different algorithms that we can use to sort an array. One way to sort an array is to use a **bubble sort** (it's not the most efficient but it's a relatively simple algorithm). The idea is that we bubble the largest element to the right hand side (end) of the array.
- To do this we traverse all of the elements in the array – let us refer to the array as `array1`.
- Then, using **pair-wise comparisons** we compare two elements with each other and put the larger of the two values in the right most position, if it is not already there. This creates our bubbling effect.
- We are just comparing each element to its right hand neighbour.

Bubble Sort

The steps of the Bubble Sort are as follows:

1. Compare the **current element** to the **element on its right**. If the current element on the left is the larger then we need to **swap** the values:

```
if array1[currentPos] > array1  
[posToTheRight] then swap
```

2. Move on to the next element in the array and compare this element's value with the element to its right and repeat the swapping process if required.

3. This process is repeated until the end of the array is reached – at this point the largest item is in the last position of the array.

Bubble Sort

4. We must now return to the first position and repeat the above process starting at Step 1. This time we position the **second largest number** in the second last element of the array. In this run through the array there is no need to go to the last element as this is already sorted.

5. We **repeat** this until we have all values in their correct position of the array.

For each iteration of the Bubble sort, the **effective size** of the array is reduced, as there is one less element to sort.

Bubble Sort

Consider the following array:

```
int [] data= {12,11,232,666,1433,0,-  
34,14,43,554};
```

How would we use a bubble sort algorithm to sort this array?

What value should be in the last position after one pass through it?

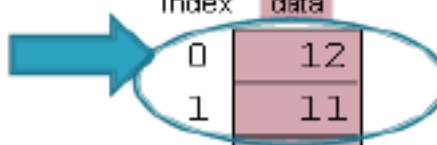
We will look in detail now at how the bubble sort would sort this array.

Bubble Sort

Step 1: Start with the first two elements of the array and compare their value.

If the value at index 0 is greater than the element at position 1 then we need to swap these values. In this case 12 is greater than 11, so we need to swap them.

data[0] and data[1]
are not in the correct
order.
So swap them



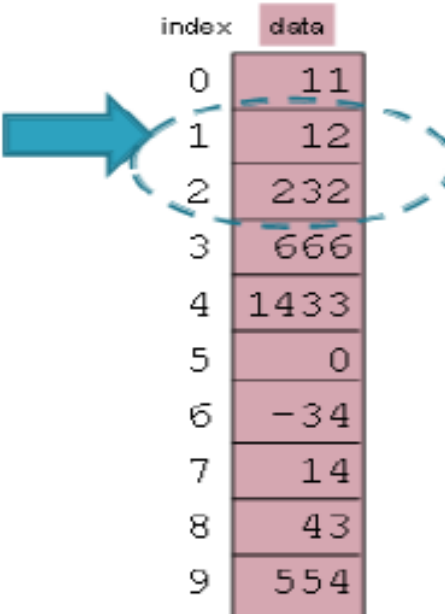
Index	data
0	12
1	11
2	232
3	666
4	1433
5	0
6	-34
7	14
8	43
9	554

Bubble Sort

Step 2: We now compare the elements at index 1 and index 2 of the array and swap if needed. In this case 12 is not greater than 232 so we do not need to swap these values.

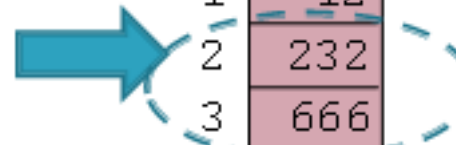
Move onto the next two values in data

index	data
0	11
1	12
2	232
3	666
4	1433
5	0
6	-34
7	14
8	43
9	554



Bubble Sort

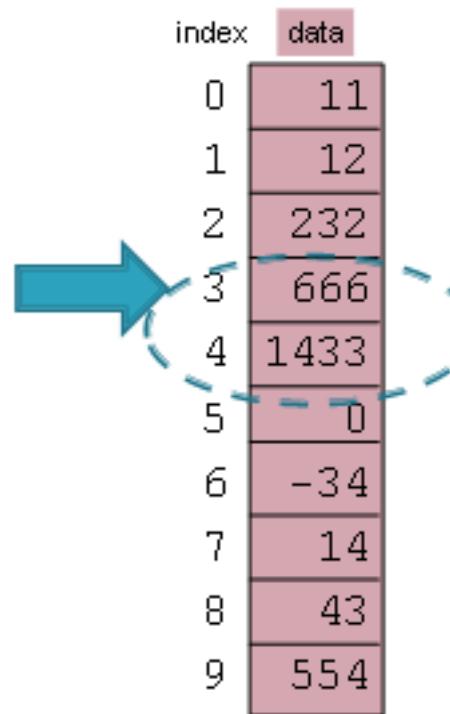
Step 3: We now compare the elements at index 2 and index 3 of the array and swap if needed. In this case 232 is not greater than 666 so we do not need to swap these values.



index	data
0	11
1	12
2	232
3	666
4	1433
5	0
6	-34
7	14
8	43
9	554

Bubble Sort

Step 4: We now compare the elements at index 3 and index 4 of the array and swap if needed. In this case 666 is not greater than 1433 so we do not need to swap these values.



index	data
0	11
1	12
2	232
3	666
4	1433
5	0
6	-34
7	14
8	43
9	554

Bubble Sort

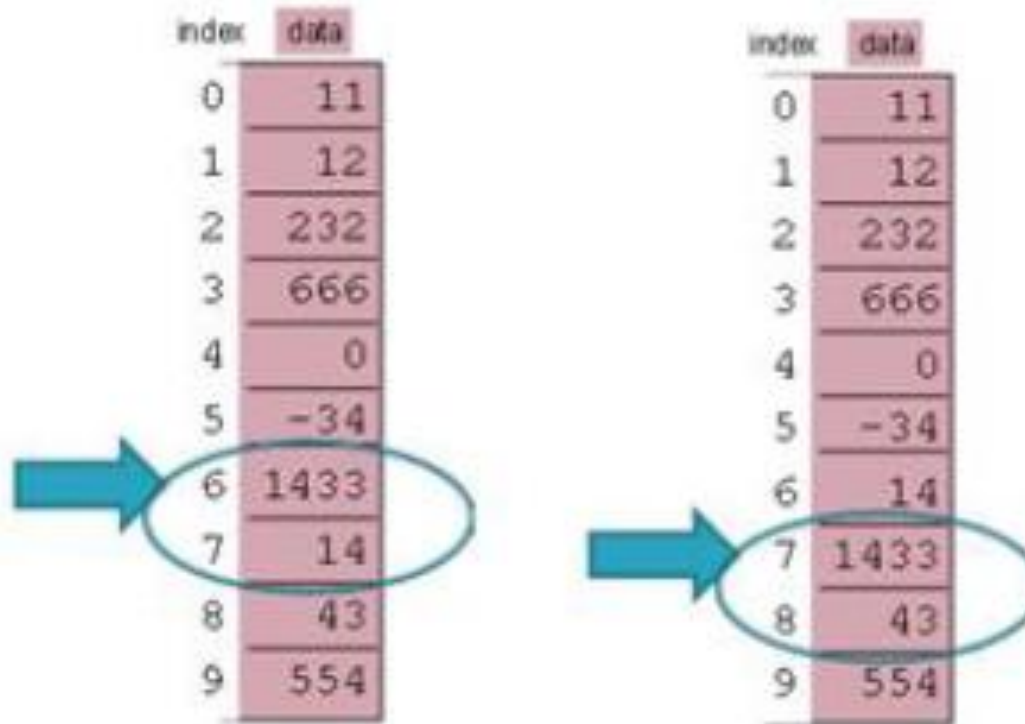
Continuing on with this approach we repeat this process for all element pairs until we **reach the end of the array**. The state of the array at each step is as follows:

index	data
0	11
1	12
2	232
3	666
4	1433
5	0
6	-34
7	14
8	43
9	554

index	data
0	11
1	12
2	232
3	666
4	0
5	1433
6	-34
7	14
8	43
9	554



Bubble Sort : Continued

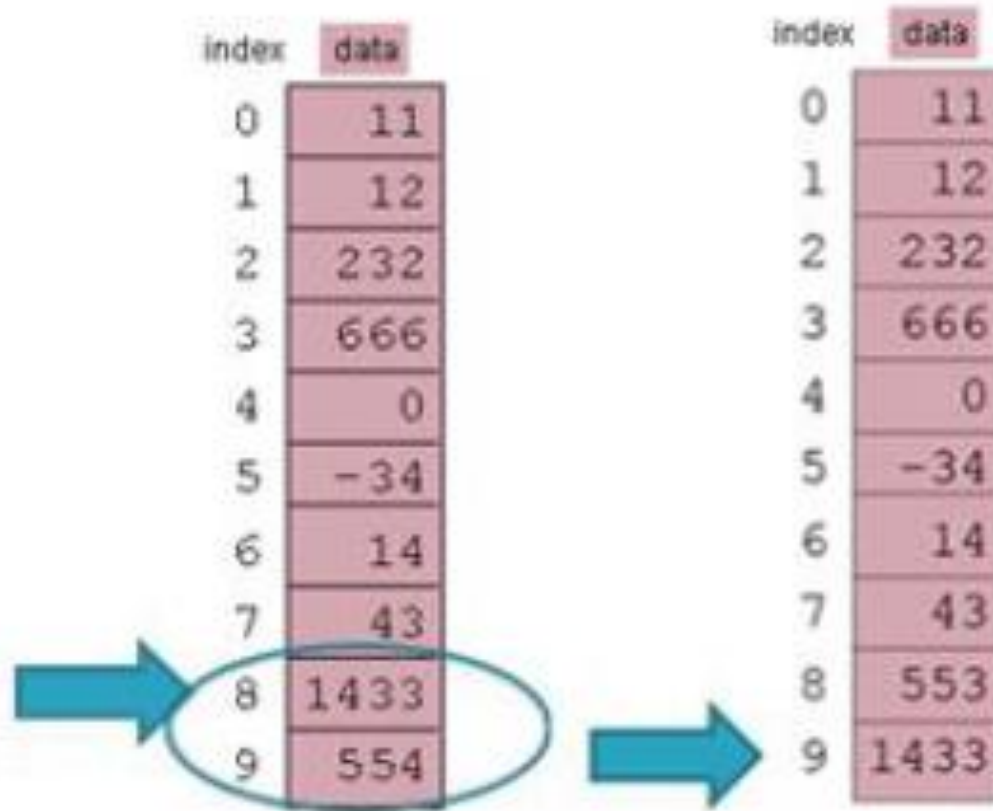


index	data
0	11
1	12
2	232
3	666
4	0
5	-34
6	1433
7	14
8	43
9	554

index	data
0	11
1	12
2	232
3	666
4	0
5	-34
6	14
7	1433
8	43
9	554



Bubble Sort : Continued



Bubble Sort : Continued

- Once we have completed this run through the array, **comparing all elements to its neighbour on its right**, the largest value is in the last location of the array.
- At this stage we **go back to the start and get the next largest value to the second last position in the array** and so on for each element of the array.

Bubble Sort : Code

- Let us now look at writing Java code to complete this Bubble sort process:

```
int[] data = {12,11,232,666,1433,0,-34,14,43,544};

for(i=0;i<data.length-1;i++)
{
    if (data[i] > data[i+1])
    {
        temp = data[i];
        data[i] = data[i+1];
        data[i+1] = temp;
    }
}
```


Bubble Sort : Code

- This code only works for one pass and at the end of this pass we have the largest number in the correct position but we need to keep going until all numbers are in the correct position.

Bubble Sort : Code

In order to get the bubble sort working for all elements we need to add another loop as follows:

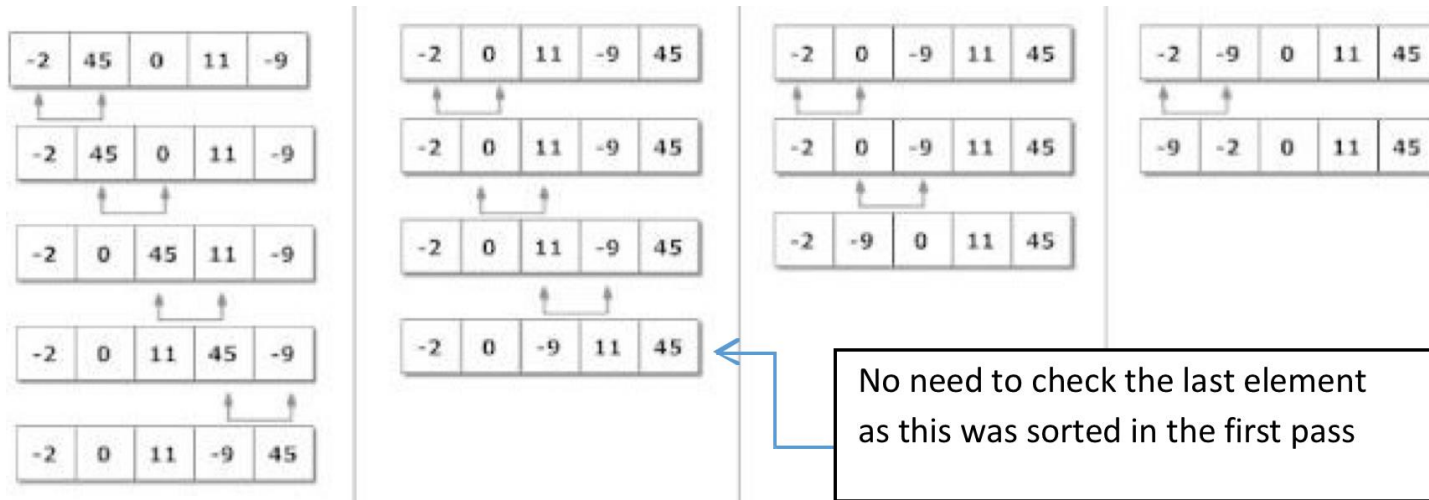
```
int [] data= {12,11,232,666,1433,0,-34,14,43,554};

int temp=0;
for(int pass=1; pass <data.length; pass++)
{
    for(int i=0;i<data.length-1;i++)
    {
        if(data[i]>data[i+1])
        {
            temp=data[i];
            data[i]=data[i+1];
            data[i+1]=temp;
        }
    }
    for(int i = 0; i < data.length; i++)
    {
        System.out.print(data[i] + " ");
    }
}
```



Improving Bubble Sort

- After each pass the highest number is in the correct place.
- As a result, subsequent passes need not check this position!
- If no swaps are made, then we assume the array must be sorted.



Code for Bubble Sort

```
public static void bubbleSort(int[] a) {  
    for (int pass = 1; i < a.length; i++) {  
        for (int j = 0; j < a.length - pass; j++) {  
            if (a[j] > a[j+1]) { // the larger item bubbles down (swap)  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```

BubbleSort.java

Notice `a.length - pass`:

This is a variation on the code in order to avoid redundant comparisons with the the last element in the array which is always the largest after each inner loop so we are reducing the search space by `n - pass` each loop

Analysis of Bubble Sort

- 1 iteration of the inner loop (test and swap) requires time bounded by a **constant c**
- Doubly nested loops:
 - **Outer loop:** exactly $n-1$ iterations
 - **Inner loop:**
 - When $i=1$, $(n-1)$ iterations
 - When $i=2$, $(n-2)$ iterations
 - ...
 - When $i=(n-1)$, 1 iteration
- Total number of iterations = $(n-1) + (n-2) + \dots + 1$
 $= n \times (n-1)/2$
- Total time = $c \times n \times (n-1)/2 = O(n^2)$

```
public static void bubbleSort(int[ ] a) {  
    for (int pass = 1; i < a.length; i++) {  
        for (int j = 0; j < a.length - pass; j++) {  
            if (a[j] > a[j+1]) { // (swap)  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```

Bubble Sort is inefficient

- Given a sorted input, Bubble Sort still requires $O(n^2)$ to sort.
- It does not make an effort to check whether the input has been sorted.
- Thus it can be improved by using a **flag**, **isSorted**, as follows (next slide):

Code of Bubble Sort (Improved version)

```
public static void bubbleSort2(int[] a) {  
    for (int i = 1; i < a.length; i++) {  
        boolean isSorted = true; // isSorted = true if a[] is sorted  
        for (int j = 0; j < a.length-i; j++) {  
            if (a[j] > a[j+1]) { // the larger item bubbles up  
                int temp = a[j]; // and isSorted is set to false,  
                a[j] = a[j+1]; // i.e. the data was not sorted  
                a[j+1] = temp;  
                isSorted = false;  
            }  
        }  
        if (isSorted) return; // why?  
    }  
}
```

BubbleSortImproved.java

Bubble Sorting- recursive version

How to write a recursive method to implement Bubble sorting?



Recursive Bubble Sorting

```
public static void sortArray(int[] array, int m, int n) {  
    if (m > 0) {  
        if (array[n] < array[n - 1]) {  
            int temp = array[n];  
            array[n] = array[n - 1];  
            array[n - 1] = temp;  
        }  
        if (n >= m) {  
            sortArray(array, m - 1, 1);  
        } else {  
            sortArray(array, m, n + 1);  
        }  
    }  
}
```

Bubble Sort : Recap

- Sorting an array in a particular order may be beneficial in order to find a value in the array in an effective method.
- Bubble sort is an algorithm which will sort an array from values high to low or low to high.
- Bubble sort algorithm
 - To do this we traverse a collection of elements
 - Then using pair-wise comparisons we swap the larger of the two values to the right most side.
 - This creates our bubbling effect



Introduction to Searching

- From time to time you will have to search through items to look for a specific value.
- There are many different ways which we can do this. You have already examined a **linear search** in Chapter 19.
- This method of searching is extremely inefficient. We could use an alternative search algorithm called a binary search which is more efficient.
- Binary search is a search algorithm designed to find a value in a **sorted** array.
- It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

Binary Search

- With the linear search it could take up to n steps to find this specific value (n = array length).
 - With binary search however it can take considerably less time.
 - With Binary search we can say how many steps it will take to find the item at most, for example:
 - *If the SORTED list is made up of numbers between 1 and 1000, it should be possible to find it in 10 goes or less...*
 - Why? Because $2^{10} = 1024$, just above the upper boundary (e.g. the array only holds up to 1000)
-

Binary Search

- It's always 2 to the power of something as we halve it (divide it in two).
- So how about between **1 and 20**, how many steps will it take at most?

If the array is SORTED, it should be possible to find it in 6 goes or less.

Why? Because $2^6 = 32$ and is above our upper boundary of 20

Binary Search Code

```
int array [] ={12,34,44,51,103,210,217,230,291,300,312,1024};

int target = 1024, high = array.length-1, low = 0, middle=0;

boolean found = false;

while(low<=high&&!found)
{
    middle= (high+low)/2;
    if(array[middle] >target)// search in the lower space
    {
        high = middle-1;
    }
    else if (array[middle] <target)// search the higher space
    {
        low = middle+1;
    }
    else {
        System.out.println(target + " found at position " + middle);
        found = true;
    }
}
if(!found)
{
    System.out.println( target + " was not found in the list");
}
```



Binary Search Code

Finding the Number

Given the following declarations

```
int array [] =  
{12, 34, 44, 51, 103, 210, 217, 230, 291, 300, 312, 1024};
```

```
int target = 1024; // the number we are looking for
```

The initial value of high will be the length of the array:

```
high=11
```

The initial value of low will be the first position of the array:

```
low =0
```

The initial value of middle will be $(high + low)/2$:

```
middle=high+low/2
```

```
middle=5
```

array	
0	12
1	34
2	44
3	51
4	103
5	210
6	217
7	230
8	291
9	300
10	312
11	1024



Binary Search Code

Finding the Number

Is the value stored at `array[middle]`, greater than the value stored in target (i.e. 1024)?

No! $1024 > 210$, so what are the new values of high and low now?

`low = middle+1` (i.e. 6), and high stays the same!

Let us look again.

`high = 11`

`low = 6`

`middle = high + low / 2`

`middle = 8`

Is the value stored at `array[middle]`, greater than the value stored at target (i.e. 1024)?

No! $1024 > 291$, so what are the new values of high and low now?

`low = middle+1` (i.e. 9), and high stays the same!

	array
0	12
1	34
2	44
3	51
4	103
5	210
6	217
7	230
8	291
9	300
10	312
11	1024

Binary Search Code

Finding the Number

Let us look again.

- `high = 11`
- `low = 9`
- `middle = high + low / 2`
- `middle = 10`

Is the value stored at `array[middle]`, greater than the value stored at target (i.e. 1024)?

No! $1024 > 312$, so what are the new values of high and low now?

`low = middle+1` (i.e. 11), and high stays the same!

Let us look again.

```
high = 11
low = 11
middle = high + low / 2
middle = 11
```

Is the value stored at `array[middle]`, greater than the value stored at target (i.e. 1024)?

• The values are equal. Therefore, we've found it.

array	
0	12
1	34
2	44
3	51
4	103
5	210
6	217
7	230
8	291
9	300
10	312
11	1024

Binary Search Recap

- Binary search is a search algorithm designed to find a value in a sorted array
- It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.
- If we look at an array containing the values between 1 and 20, how many steps will it take at most.
 - If the array is SORTED, it should be possible to find it in 6 goes or less
 - Why? Because $2^6 = 32$ and if above our upper boundary of 20.

How to give recursive binary searching

```
private static int recursiveFind(int[] arr,int start,int end,int searchKey){  
    if (start <= end) {  
        int middle = (start+end)/2;  
        if (searchKey == arr[middle]) {  
            return middle;  
        } else if (searchKey < arr[middle]) {  
            return recursiveFind(arr, start, middle - 1, searchKey);  
        } else {  
            return recursiveFind(arr, middle + 1, end, searchKey);  
        }  
    } else {  
        return -1;  
    }  
}
```