Sorting

This lecture we will learn
- Insertion Sort
- Selection Sort
- Bubble Sort

Sorting as you known is a method for arranging data in ascending or descending order. Ascending order means starting from the lowest value and moving to the highest value. Descending order is opposite of ascending order, i.e. Starting from highest value and moving to the lowest value.

Sorting algorithms that we will discuss will arrange data in ascending order.

Selection Sort

We start with an unsorted array. Our objective is to sort elements in the array in an ascending order.

In selection sort, after each iteration, we find the smallest(a[0],a[1],a [2], a[3], a[4]), and store the smallest value at a[0].

Then we try to find the smallest of (a[1],a[2],a[3],a[4]) and store the smallest value in a[1].

Then we try to find the smallest of (a[2],a[3],a[4]) and store the smallest value in a[2].

Then we try to find the smallest of (a[3],a[4]) and store the smallest value in a[3].

And we are finished.

Watch the flash presentation.

Summary of Selection Sort.
In the first iteration, we compared a[0] with a[1], a[2], a[3], a[4]. i.e. We first checked if condition (a[0] > a[1]) is true, then swap. This swap ensures that smallest data among a[0] and a[1] is stored in a[0].

Then we checked if condition (a[0] > a[2]) is true, then swap. This swap ensures that smalelst data among a[0] and a[2] is stored in a[0].

We repeat the above procedure by comparing a[0] with a[3] and a[4]. This ends the first iteration. After the end of first iteration, we have smallest data value stored in a[0].

In second iteration, we need to find the smallest value among a[1], a[2], a[3], a[4]. TO find the smallest value, we compare a[1] with a[2], a[3], and a[4]. If a[1] > a[2], swap. If a[1] > a[3] swap. If a[1] > a[4] swap. After these three comparison, we were able to find the smallest value and a[1] contains that smallest value.

Then comes the 3rd iteration, and finally 4th iteration.

So, how many iterations we had? 4

This means, for an array of size 5, we need 4 iteration.
For an array of size 10, we need 9 iterations.
For an array of size n, we need n-1 iterations.

In this example, size of array is 5, so we need 4 iterations.
In 1st iteration i=0.
In 2nd iteration i=1.
In 3rd iteration i=2.
In 4th iteration i=3.

In 1st iteration, we compare a[0] with a[1], a[2],a[3], a[4]. This means 1st iteration contains in itself 4 iterations or Inside 1st iteration there are 4 iterations.
1.1compare a[0] with a[1]
1.2compare a[0] with a[2]
1.3compare a[0] with a[3]
1.4compare a[0] with a[4].

This means, when i=0, we should start a loop from 1 to 4.

In 2nd iteration, we compare a[1] with a[2], a[3], a[4]. This means 2nd iteration contains in itself 3 iterations, or Inside 2nd iteration there are 3 iterations.
2.1compare a[1] with a[2].
2.2compare a[1] with a[3]
2.3compare a[1] with a[4]

This means, when i=1, we should start a loop from 2 to 4.

In 3rd iteration, we compare a[2] with a[3], a[4]. This means 3rd iteration contains in itself 2 iterations, or Inside 3rd iteration there are 2 iterations.
3.1compare a[2] with a[3]
3.2compare a[2] with a[4]

This means, when i=2, we should start a loop from 3 to 4.

In 4th iteration, we compare a[3] with a[4]. This means 4th iteration contains in itself 1 iteration, or Inside 4th iteration there is 1 iteration.
4.1compare a[3] with a[4].

This means, when i=3, we should start a loop from 4 to 4.

Outer for loop iterates starts from 0 until size_of_array – 1.
In this examle size_of_array=4, so for loop iterates from 0 to 3.

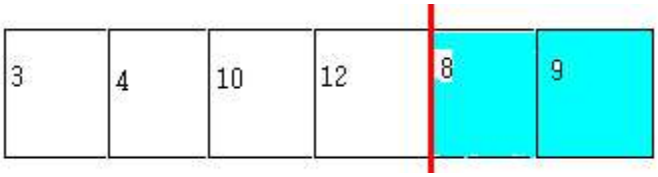
```
for(i=0;i<4;i++) // this loop executed 4 times from i=0, 1, 2, 4
        for(j=i+1;j<=4;j++) //i=0, j = 1 to 4. i=1, j=2 to 4. i=2, j=3 to 4. i=3, j=4 to 4
                if (a[i] > a[j])
                        swap(a[i],a[j])
```

Insertion Sort

Below is an array and we are in the middle of insertion sort. I have taken one step from the middle of insertion sort. Lets see what happens here.

As you can see, there are 6 elements in the array. There is red line which logically partitions array into 2 parts. Left part (unshaded) represents sorted array. Right part (shaded) represents unsorted array.

We want to find the correct position of 8 in sorted array. The correct position of 8 is after 4 and before 10. This is what insetion sort do. **Insertion sort finds the correct position of a value (in this case 8) in the sorted part of the array**.



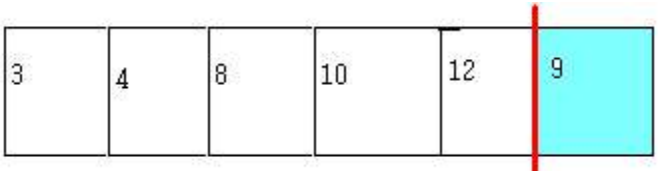
How does insertion sort finds the correct position of 8 in sorted part of array?

- Compare 8 with a[0]. If 8 < a[0], then shift ahead all the elements from position 0 until 3 by 1 position. After shifting store 8 in position 0. In this case, 8 < a[0] is false.

- Compare 8 with a[1]. If 8 < a[1], then shift ahead all the elements from position 1 until 3 by 1 position. After shifting store 8 in position 1. In this case 8 < a[1] is false.

- Compare 8 with a[2]. If 8 < a[2], then shift ahead all the elements from position 2 until 3 by 1 position. After shifting store 8 in position 2. In this case 8 < a[1] is true, therefore shift a[2] and a[3] by 1 position thus creating space for 8.

- After shifting and storing 8 in correct place, our array looks like this.



Now watch the flash presentation.
Explanation of insertion sort

25, 17, 31, 13, 2

In the above list, red part represents sorted list, and yellow part represents unsored list. In the beginning we assume that 1st element (a[0]) in the array is part of sorted list and 2nd, 3rd, 4th, 5th are part of unsorted list.

In the 1st iteration, we need to find the correct postion of a[1] (17) in sorted list. If a[1] is less than a[0], then shift ahead all the elements from position 0 until 0 by 1 position. Shifting a[0] ahead by 1 position will create an empty space at position 0. Store value of a[1] at position 0.
After 1st iteration, our list looks like :

17, 25, 31, 13, 2
(Red part shows sorted list and yellow part shows unsorted list.)

In the 2nd iteration, we need to find the correct position of a[2] (31) in sorted list.
We compare a[2] with a[0]. If a[2] < a[0], then shift ahead all the elements from position 0 until 1 by 1 position. Shifting will create a vacant space at position 0. Store value of a[2] in position 0. In this example a[2] < a[0] is false, therefore no shifting.

Then compare a[2] with a[1]. If a[2] < a[1], then shift ahead all the elements from position 1 until 1 by 1 position. Shifting will create a vacant space at position 1. Store value of a[2] in position 0. In this example a[2] < a[1] is false, therefore no shifting.
After 1st iteration our list looks like :

17, 25, 31, 13, 2
(Red part shows sorted list and yellow part shows unsorted list.)

In the 3rd iteration, we need to find the correct position of a[3] (13) in sorted list.
We compare a[3] with a[0]. If a[3] < a[0], then shift ahead all the elements from position 0 until 2 by 1 position. Shifting will create a vacant space at position 0. Store value of a[3] at position 0. In this example a[3] < a[0] is true, therfore shift.

If the above condition is false, i.e. a[3] < a[0] is false,
then check for condition a[3] < a[1] is true, if it is true, then shift.

If the above condition is false, i.e. a[3] < a[1] is false,
then check for condition a[3] < a[2] is true, if it is true, then shift.
After 3rd iteration, our list looks like,

13, 17, 25, 31, 2
(Red part shows sorted list and yellow part shows unsorted list.)

In the 4th iteration, we need to find the correct position of a[4] (2) in sorted list.
We compare a[4] with a[0]. If a[4] < a[0] is true, then shift ahead all the elements from

position 0 until 3 by 1 position. Shifting will create empty space at position 0. Store value of a[4] at position 0. In this case, a[4] < a[0] is true, hence shift.

If the above condition is false, i.e. a[4] < a[0] is false,
then check for condition a[4] < a[1] is true, if it is true, then shift.

If the above condition is false, i.e. a[4] < a[1] is false,
then check for condition a[4] < a[2] is true, if it is true, then shift.

If the above condition is false, i.e. a[4] < a[2] is false,
then check for condition a[4] < a[3] is true, if it is true, then shift.

After 4th iteration, our list looks like

2, 13, 17, 25, 31.
(Red part shows sorted list and yellow part shows unsorted list.)

Q. Sort the following list of numbers?

8, 9, 3, 5, 6, 4, 2, 1, 7, 0
(Red parts represent sorted list and yellow part represent unsorted part.)

```
/*
Let ne be the number of elements. Outer loop iterates n-1 times.
For each value of i, we try to find the position of a[i] in list a[0] to a[i-1].
*/
for(i=1;i<n;i++)
{
	temp=a[i]; // store the element (whose position we wan to find) in temp.
/*
in the inner for loop, we are trying to find the position a[i] in list a[0] to a[i-1]. Therefore we start a loop from 0 to i-1.

In eatch iteration of the loop, we check if a[i] <  a[j] is true. If it is true, we shift  and store a[i] in position j.

After finding correct position of a[i], we break inner loop and start with next iteration of outer loop.
*/
	for (j=0;j<=i -1; j++)
	{
		if ( temp < a[j])
		{
			shift(a, j,i-1) // shift all the elements starting form position j until i-1.
			a[j] = temp;
			break;
		}
	}
}
```

```
void shift(int *arr, int x, int y)
{
	for (int i=y; i>=x;i--)
		arr[i + 1] = arr[i];
}
```

Bubble Sort

In bubble sort, we compare adjacent elements, and finally at the end of each iteration biggest element is at right most position.

See the flash presentation.

```
/*
n is number of elements in the array.
 outer loop controls the number of iteration, if n=5, then k=3, 2, 1, 0.
In 1st iteration, k=3, inner for loop iterates from i=0 ,1, 2, 3
In 2nd iteration, k=2, inner for loop iterates from i=0, 1, 2
In 3rd iteration, k=1, inner for loop iterates from i=0 , 1
In 4th iteration, k=0, inner for loop iterates from i=0
*/
for(k=n-2;k>=0;k--)
{
	for(int i=0;i<=k;i++) //  i  = 0, 1,2, ..., n-1
	{
		if (a[i] > a[i+1])
			swap(a, i, i+1);
	}
}
```