Unit 2: Iterative algorithms

In this unit, we will design iterative algorithms for following problems and analyze their complexity.

- Algorithms for finding GCD and Fibonacci Number
- Searching Algorithms: Sequential Search
- Sorting Algorithms: Bubble sort, Selection sort, and Insertion Sort

Iterative algorithm for finding GCD of two Integers

- The GCD of a two integers a and b is the largest integer x that divides both a and b.
- A naïve approach to solve gcd problem would be to find prime factors of the numbers and note the common factors. However this approach is very complex.
- The most famous gcd algorithm is Euclid's algorithm.

Algorithm:

```
Gcd(int a, int b)          //a>b>0
  {
    while(a!=b)
       if(a>b)
          a = a-b
       else
          b = b-a
    end while
    return a
  }
```

- For some positive integers a and b, it works by repeatedly subtracting the smaller number from the larger one until they become equal.
- At this point, the value of either term is the greatest common divisor of our inputs.

Example: Trace the algorithm to find GCD(2100,980)

Analysis:

- We see that each iteration decreases either a or b by some amount.
- Since the smallest possible deduction for each step is 1, and since all positive integers are guaranteed to have a common divisor of 1, the worst case time complexity (upper bound) will be O(a+b).
- Space complexity is the negligible constant since there are only two variables to be stored.

Iterative algorithm for finding n^{th} Fibonacci number

- We know the famous Fibonacci series where each number is the sum of the two preceding ones, starting from 0 and 1.

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, .. .

- Applications of Fibonacci numbers include computer algorithms such as the *Fibonacci search technique*, the *Fibonacci heap data structure*, and graphs called *Fibonacci cubes* used for interconnecting parallel and distributed systems.
- They also appear in biology, such as branching in trees, the arrangement of leaves on a stem etc

Algorithm:

```
fibonacci(int n)
 {
   fib1 = 0
   Fib2 = 1
   i = 3
   while(i<=n)
     {
       fib = fib1 + fib2
       fib1 = fib2
       fib2 = fib
       i = i + 1
     }
   return fib
 }
```

Analysis:

- The first three statements take 1 step each
- The the four statements inside while loop run for n-3 times

Total steps = 3 + 4(n-3)

= 4n -9

= $\Theta(n)$

Sequential search algorithm

The Search problem can be stated as follows:

> *"Given an array(list) A of n elements and a search key K, find the position of K in A (if K is in A)."*

- A simple approach to solve search problem is the sequential search.

Idea:

- ✓ compare K with the first element a in array
- ✓ If K != a then compare with next element in array
- ✓ Continue until the required element is found.

Algorithm:

```
Seq-search(A,K)
  {
   for(i = 0 ; i < n ; i++)
    {
      if(K = A[i])    //element found!
        return(i)
    }
   return(-1)         //element not found
  }
```

Analysis:

- *Best case*: Θ(1) Best case occurs when the search element is in first position of array.
- *Worst case*: O(n) when the search element lies at the last position of array, the *'for'* loop runs n times performing n comparisons.
- Space complexity: Θ(n) How??

Sorting algorithms

Sorting problem:

"Given an array A of n elements, arrange the elements in ascending or descending order"

- There are numerous sorting algorithms with their own advantages, Limitations and applications.

- Here we will deal with following sorting techniques:
    1. *Bubble sort*
    2. *Selection sort*
    3. *Insertion sort*

1. *Bubble sort*

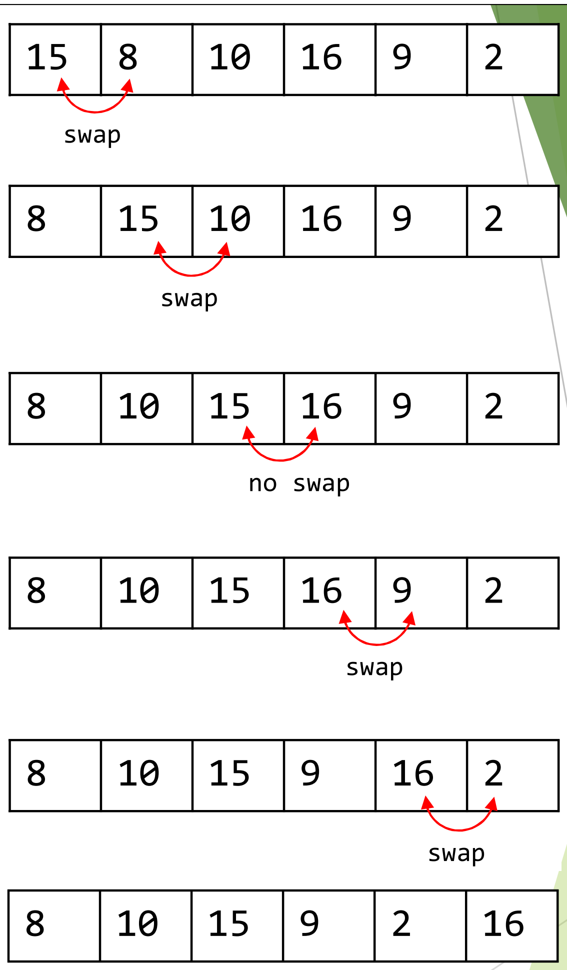
- Bubble sort is a simple comparison-based sorting algorithm.
- Each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- We iterate through several rounds. Just like the movement of air bubbles in the water that rise up to the surface, one element of the array move to the correct position in each iteration. Therefore, it is called a bubble sort.
- let us first see how bubble sort works:

A =

15	8	10	16	9	2

Round 1 : Start with first element.

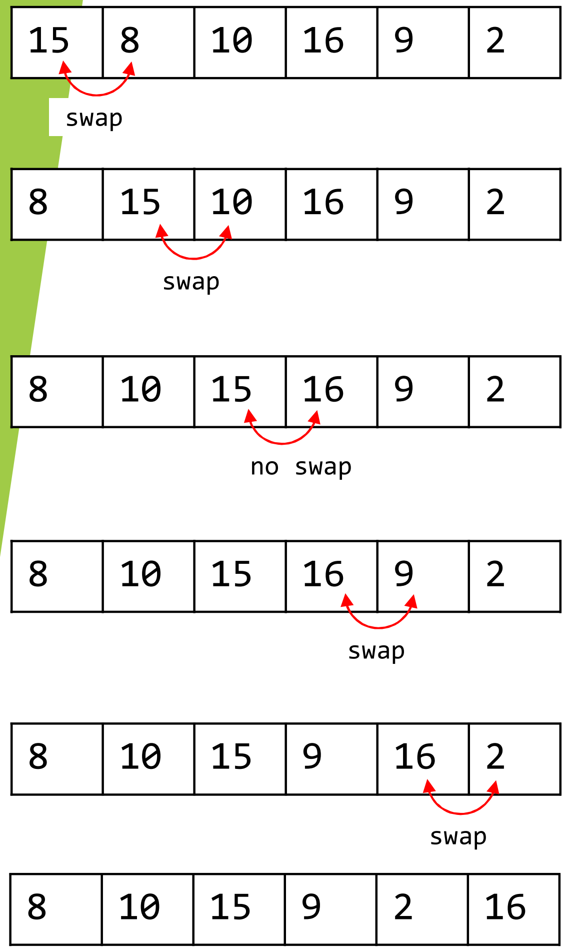
compare adjacent elements and swap if not in order.



Round 1 complete.

Now one (the largest) element 16 has bubbled up in correct position.

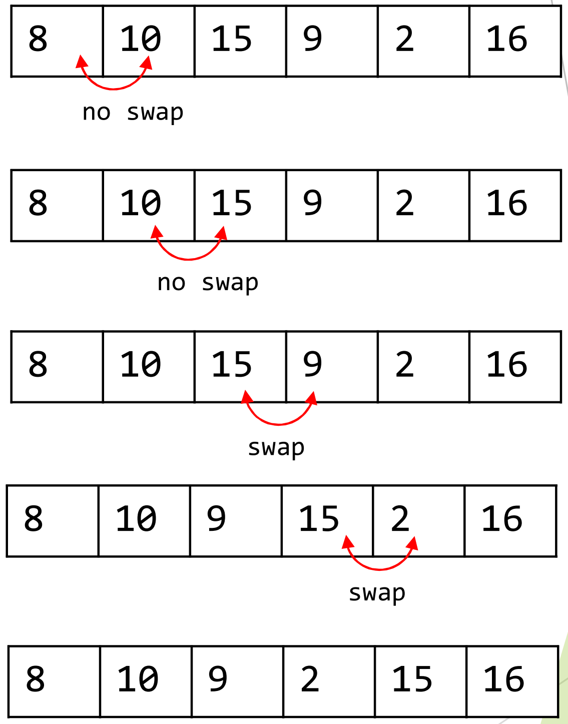
Round 1:



Round 1 complete.
Now one (the largest) element 16 has bubbled up in correct position.

Round 2:

Now we leave the last element *and again* start with first element comparing adjacent element and swapping if necessary.



Round 2 complete. Now second last element is in correct position.

we continue the process for n-1 rounds.

Algorithm:

```
Bubblesort(A)      //the input array is A
 {
   n = size(A)
   for(i=0;i<n-1;i++)
     for(j=0;j<n-i-1;j++)
       if(A[j]>A[j+1])
         swap(A[j],A[j+1])
 }
```

Analysis:

- There are n-1 rounds.

Number of comparisons in first round = n-1

Number of comparisons in second round = n-2

.. .. ..

Number of comparisons in $(n-1)^{th}$ round = 1

Therefore,

Total number of comparisons =

$1 + 2 + 3 + ... + (n-2) + (n-1)$

= ??

There is another way to analyze the complexity:

The outer 'for' loop runs for n-1 times
The inner 'for' loop runs at most n-1 times.
Therefore the comparison and swapping may be done at most $(n-1)(n-1) = O(n^2)$ times.

2. *Selection sort*

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.
- The algorithm works as follows:

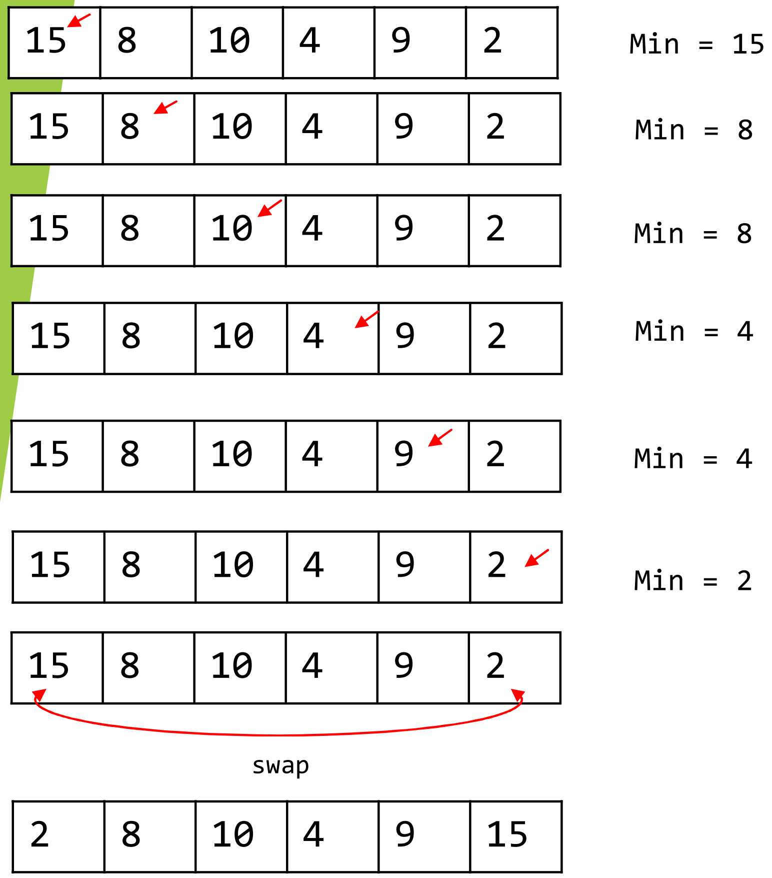
1. Let we have an array A with n elements.
2. Find the smallest element a_0 of A and swap it with A[0].
3. Then find the smallest element a_1 of A-{a_0} and swap it with A[1].
4. Find the smallest element a2 of A-{a_0,a_1} and swap with A[2].
5. And so on..

- We work through n-1 rounds. In each round we find smallest element in unsorted array and put it in sorted position.

Example:

15	8	10	4	9	2

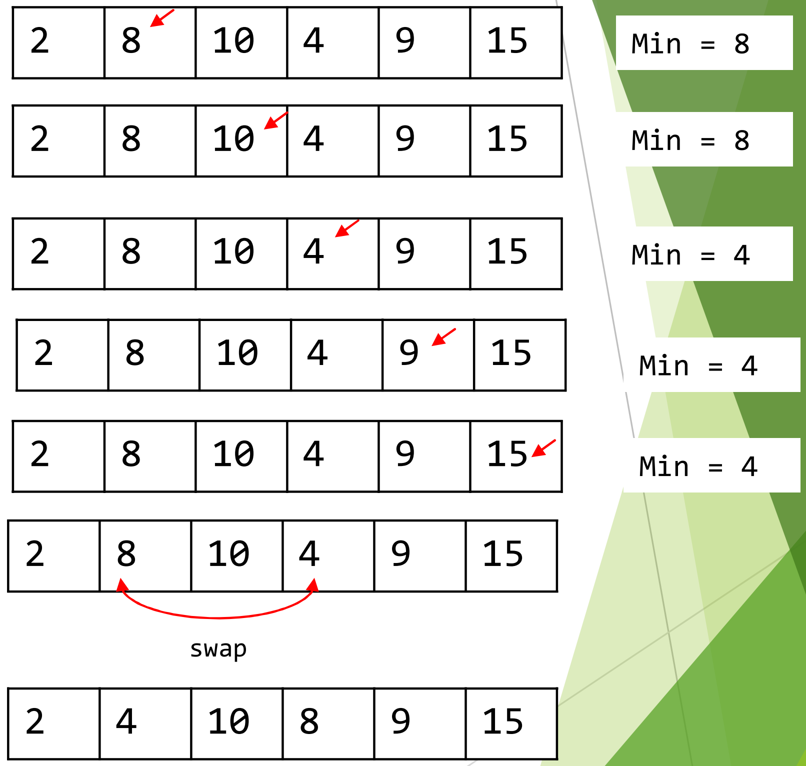
Round 1:



Round 1 complete.
Now the minimum element 2 is selected and put in first position.

Round 2:

Now we leave the first element *and* again start with second element to find next smallest element.



Round 2 complete. Now second smallest element is in correct(second) position.

we continue the process for n-1 rounds.

Algorithm:

```
selectionsort(A)
 {
   n = size(A)
   for (i = 0; i < n-1; i++)
       {
         min_idx = i;
         for (j = i+1; j < n; j++)
          {
           if (A[j] < A[min_idx])
               min_idx = j;
          }

         swap(A[min_idx], A[i])
       }

   }
```

Analysis:

- There are n-1 rounds.
- In each round, comparisons are performed to find min element. Only one swap is done in each round

Number of comparisons in first round = n-1

Number of comparisons in second round = n-2

.. .. ..

Number of comparisons in (n-1)th round = 1

Therefore,

Total number of comparisons =

$1 + 2 + 3 + ... + (n-2) + (n-1)$

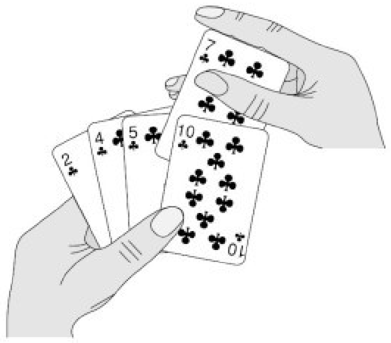
$= \Theta(n^2)$

There is another way to analyze the complexity:
The outer ‘for’ loop runs for n-1 times
The inner ‘for’ loop runs at most n-1 times.
Therefore the comparisons and swapping may be done at most $(n-1)(n-1) = O(n^2)$ times.

Advantage: selection sort makes less swaps O(n) than bubble sort (1 swap in each round). It can be useful when memory write is a costly operation.

3. *Insertion sort*

- Insertion works the way we sort a hand of playing cards.
- We start with empty left hand and the cards face down on the table.
- We pick up one card.
- We then pick up other cards (one at a time) and insert it into correct position among the cards in hand. To find the correct position for new card, we compare with each of the card already in the hand.
-



Example:

15	8	10	4	9	2

Round 1:
Pickup second element and find its correct position in on the left side.

15	8	10	4	9	2

key = 8 (second element)

15		10	4	9	2

move 15 to right

	15	10	4	9	2

8	15	10	4	9	2

Insert key = 8

Round 1 complete.
Now the first two elements are sorted.

Round 2:
Now we pickup third element *as key and* move elements elements on the left side of key to find correct position of the key.

8	15	10	4	9	2

key = 10 (third element)

8	15		4	9	2

move 15 to right

8		15	4	9	2

8	10	15	4	9	2

Insert key = 10

Round 2 complete.
Now the first 3 elements are sorted.

Round 3:

8	10	15	4	9	2

key = 4 (fourth element)

8	10	15		9	2

move

8	10		15	9	2

move

8		10	15	9	2

move

	8	10	15	9	2

4	8	10	15	9	2

Insert 4

Round 3 complete. we continue the process for n-1 rounds.

Algorithm:

```
insertionsort(A)
 {
   n = size(A)
   for (i = 1; i < n; i++)
       {
          key = A[i];

          //insert A[i] into sorted array A(0,1, .. i-1)
          j = i-1

          while(j ≥ 0 and A[j] > key)
            {
             A[j+1] = A[j]
             j = j-1
            }

          A[j+1] = key
       }
 }
```

Analysis:

- There are n-1 rounds.
- In each round, comparisons are performed and elements are moved to find position for key element.

Max^{m} comparisons and moves in first round = 1

Max^{m} comparisons and moves in second round = 2

.. .. ..

Max^{m} comparisons and moves in (n-1)th round = n-1

Therefore,

Maximum total number of comparisons =

$1 + 2 + 3 + ... + (n-2) + (n-1)$

$= O(n^2)$

There is another way to analyze the complexity:
The outer 'for' loop runs for n-1 times
The "while" loop inside 'for' loop runs at most n-1 times.
Therefore the comparisons and moves may be done at most $(n-1)(n-1) = O(n^2)$ times.

Worst case and best case:

- By nature of its working , insertion sort has best and worst case time complexity.
- Insertion sort takes maximum time $O(n^2)$ to sort if elements are sorted in reverse order.
- And it takes minimum time (O(n)) when elements are already sorted.

Advantage: Insertion sort is useful when Array is small. It can also be useful when input array is almost sorted and only few elements are misplaced in complete big array.

End of unit 2

Thank You!