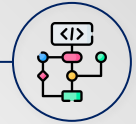


Data structure and Algorithms

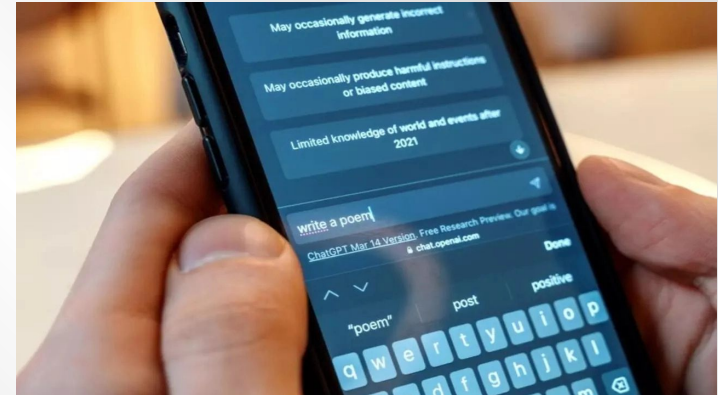


Presented by : Asmaa Ghonaim



Sample Real World **problem**

- **Keyboard** input on cell phones
 - After each letter typed, guess 3 words
 - Have you wondered how?
 - Also the system is given:
 - A list of possible English Words
 - A collection of messages of the user



Sample Real World **problem**

- **Keyboard** input on cell phones
 - What are the desirable characteristics for guessing 3 word
 - **Accuracy**
 - Algorithms
 - **Speed**
 - Algorithms and data structures
 - **Less memory space**
 - Data structures



What about **data structures**?

Structure : How data is organized ?

- ❑ Place data **continuously**
- ❑ Place data here and there with "**links**"
- ❑ Place data with "**formula**"
- ❑ Organization of a data structure may allow better algorithms to be applied



Algorithms : How to access data for a result/task ?

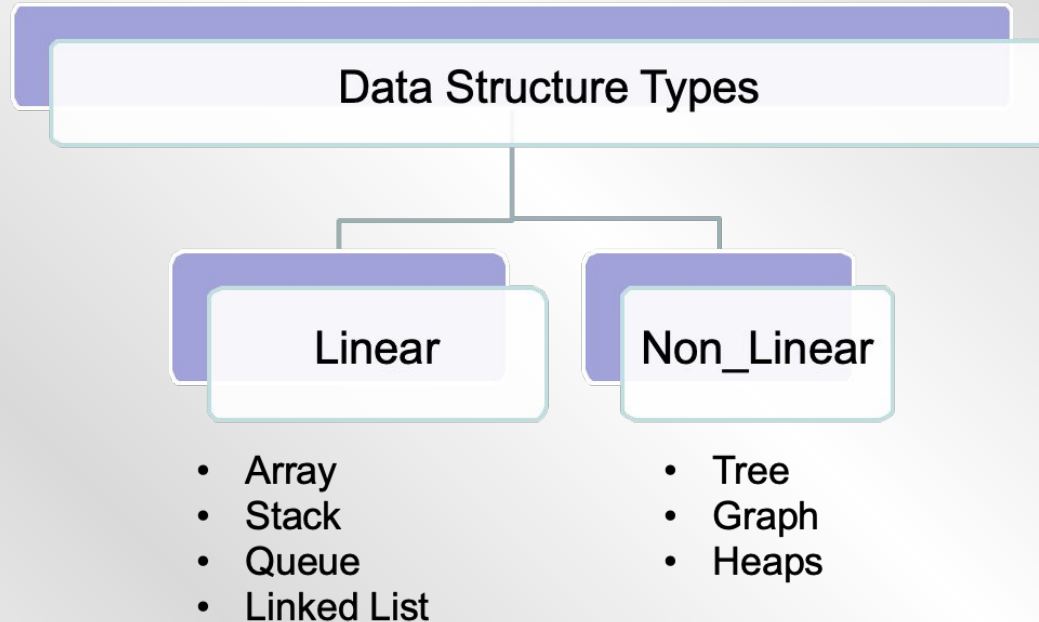
A high level, language independent, description of a step-by-step process

- ❑ Scan data sequentially
- ❑ Scan data according to the sequence of a structure
- ❑ Scan data with "**formula**"
- ❑ Algorithms can be smarter without a special organization of a data structure



What about **data structures**?

- **Data Structure**



What about **algorithms**?

Algorithms:

- Is a finite set of instructions which , when followed, accomplishes a particular task.
- There can be more than one algorithm to solve a given problem.
- An algorithm can be implemented using different programming languages.
- we should employ mathematical techniques that analyze algorithms
- **independently** of specific **implementations, computers, or data.**

What about **algorithms**?

Algorithms:

→ There are two aspects of Algorithm performance:

- ◆ **Time**

- How to estimate the time required for an algorithm
- How to reduce the time required

- **Space**

- Data structures take space
- What kind of data structures can be used?
- How does choice of data structure affect the runtime?

What about **algorithms**?

Analysis of Algorithms:

→ The Execution Time of Algorithms:

- Cost is the amount of computer time required for a single operation in each line [1].

Times is the amount of computer time required by each operation for all its repeats.

Total is the amount of computer time required by each operation to execute.

- It requires 1 unit of time for Arithmetic and Logical operations
- It requires 1 unit of time for Assignment and Return value
- It requires 1 unit of time for Read and Write operations

`count = count + 1;`

→ take a certain amount of time, but it is constant say: 2 units



What about **algorithms**?

Example :

count = count + 1; Cost: 1 +1 Time : 1

sum = sum + count; Cost: 1+1 Time : 1

Total cost = 4 units

What about **algorithms**?

The Execution Time of Algorithms :

Example : Simple Loop

```
int i = 1;
int sum = 0;
while(i<=n)
{
    i=i+1;
    sum=sum+i;
}
```

Cost	Times
1	1
1	1
1	n+1
2	n
2	n

Total cost = $1 + 1 + (n+1) + 2n + 2n$

- The time required for this algorithm is proportional to **n**

What about **algorithms**?

The Execution Time of Algorithms :

Example : Simple Loop

```
int sum = 0, i;  
for(i = 0; i < n; i++)  
{  
    sum = sum + A[i];  
} return sum;
```

Cost	Times
1	1
1+1+1	1+(n+1)+n
2	n
1	n

Total cost = $1 + (2+2n) + 2n + 1 = 4n + 4$

- The time required for this algorithm is proportional to **n**
- If the amount of time required by an algorithm is **increased** with the **increase** of input value then that time complexity is said to be **Linear Time Complex it**

What about **algorithms**?

The Execution Time of Algorithms :

- General Rules for Estimation:
 - **Loops:** The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
 - **Nested Loops:** Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the size of all loops.
 - **Consecutive Statements:** Just add the running times of those consecutive statements.
 - **If/Else:** Never more than the running time of the test plus the larger of running times of S1 and S2.

What about **algorithms**?

The Execution Time of Algorithms :

- **Big O Notation:**
 - An algorithm's proportional time requirement is known as **growth rate**.
 - The function **$f(n)$** is called the algorithm's growth-rate function.
 - Since the capital **O** is used in the notation, this notation is called the Big O notation.
 - If Algorithm A requires time proportional to n^2 , it is $O(n^2)$.
 - If Algorithm A requires time proportional to n , it is $O(n)$.

What about **algorithms**?

What to Analyze:

- An algorithm can require different times to solve different problems of the same size.
- ◆ **Worst-Case Analysis**: The maximum amount of time that an algorithm require to solve a problem of size n .
 - This gives an upper bound for the time complexity of an algorithm.
 - Normally, we try to find worst-case behavior of an algorithm.
- **Best-Case Analysis**: The minimum amount of time that an algorithm require to solve a problem of size n .
 - The best case behavior of an algorithm is NOT so useful.

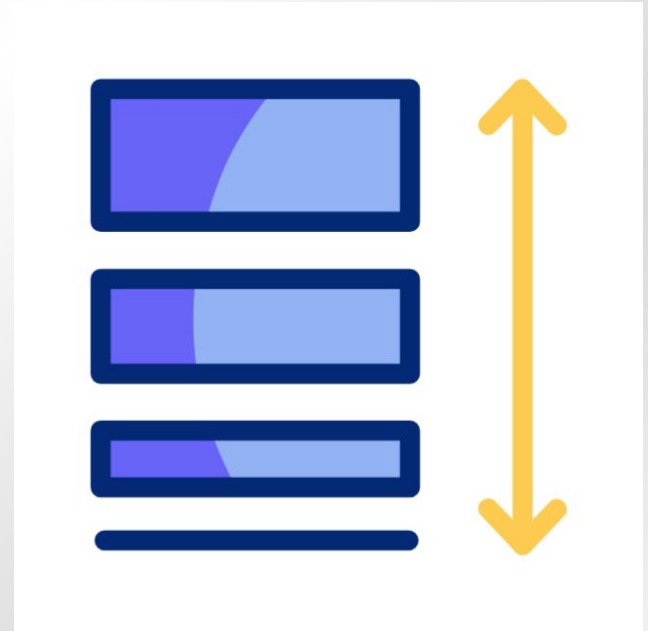
What about **algorithms**?

What to Analyze:

- **Average-Case Analysis:** The average amount of time that an algorithm require to solve a problem of size n .
 - Sometimes, it is difficult to find the average-case behavior of an algorithm.
 - We have to look at all possible data organizations of a given size n , and their distribution probabilities of these organizations.

Worst-case analysis is more common than average-case analysis.

Sorting Algorithms



Sorting Algorithms

Sorting:

- ❑ Is the process of arranging a set of similar information into an increasing or decreasing order.
- ❑ Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.

Sorting Algorithms:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort

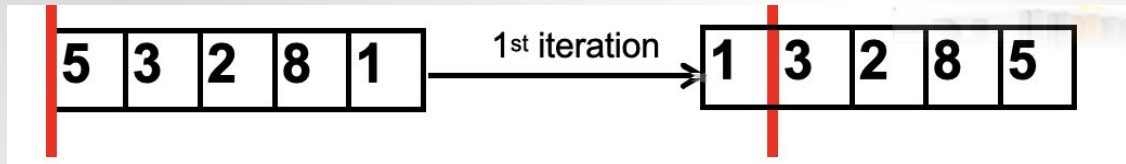


- ❑ The first three are the foundations for faster and more efficient algorithms.

Sorting Algorithms

Selection Sorting :

- ❑ The list is divided into two sub lists, **sorted** and **unsorted**.
 - We find the smallest element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.
 - After each selection and swapping, the wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
 - A list of n elements requires $n-1$ passes to completely rearrange the data.



Sorting Algorithms

Selection Sorting :

Algorithm Procedure

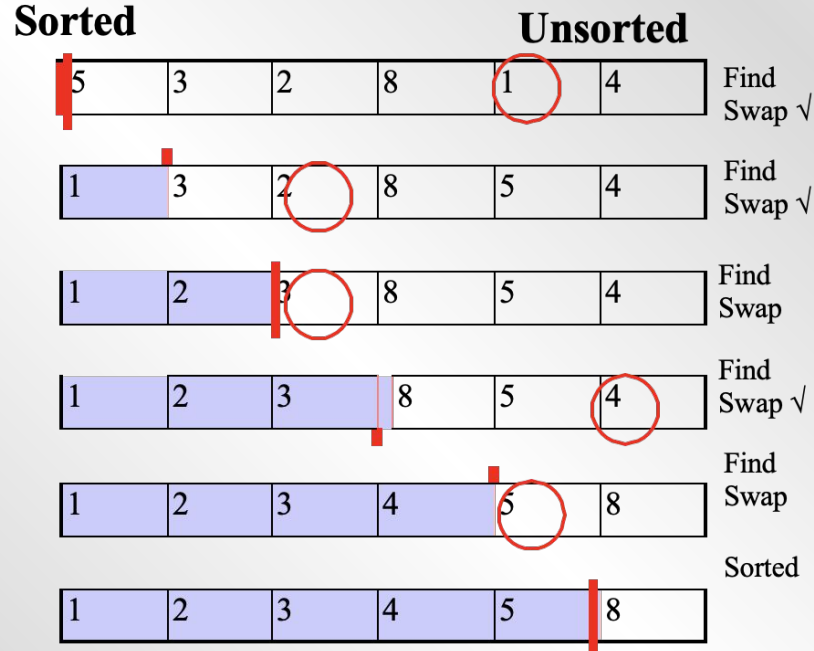
1. Find the minimum value in the list.
2. Swap it with the value in the first position.
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

Find & Swap



Sorting Algorithms

Selection Sorting :



Sorting Algorithms

Selection Sorting :

```
void main ()
{
    int i, n;
    int a[6]={5,3,2,8,1,4};
    selection_sort (a, 6);
    for (i = 0; i < 6; i++)
        cout << a[i] << endl;
}

void swap (int &x, int &y)
{
    int temp;
    temp = x; x = y;
    y = temp;
}
```

C++ code implementation

Sorting Algorithms

Selection Sorting :

```
void selection_sort (int *a, int n)
{
    // *a: the array to sort, n: the array length
    int i, j, min;
    for (i = 0; i < (n-1); i++)
    {
        min = i;
        // assume that the first element is the min at the
        // beginning
        for (j = (i+1); j < n; j++)
        {
            // find the index of the min element
            // swap if needed
        }
    }
}
```

C++ code implementation

Sorting Algorithms

Selection Sorting :

Performance Analysis

Selection sort is not difficult to analyze compared to other sorting algorithms since **none of the loops depend on the data in the array.**

Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on $n(n - 1) / 2$

1. Best Case $O(n^2)$
2. Worst Case $O(n^2)$
3. Average Case $O(n^2)$

Where, n is the number of items being sorted

Sorting **Algorithms**

Bubble Sorting:

Algorithm Procedure

1. Compare the first two elements of the array and swap them if they are out-of-order.
2. Continue doing to swap the array for each pair of elements, until you reach the last entry. the array for each two adjacent elements until you reach the last entry.
3. At this point the last entry is the largest element in the array.
4. Continue this procedure for each next largest element until the array is fully sorted.

Step-by-Step Example:

6	8	3	5	1
---	---	---	---	---

Sorting Algorithms

Bubble Sorting:

Performing 1st Iteration:

Comparing 1st & 2nd numbers
No Swapping, since $8 > 6$

6	8	3	5	1
---	---	---	---	---

6	8	3	5	1
---	---	---	---	---

6	8	3	5	1
---	---	---	---	---

Comparing 2nd & 3rd numbers
Swapping, since $3 < 8$

6	8	3	5	1
---	---	---	---	---

6	3	8	5	1
---	---	---	---	---

Comparing 3rd & 4th numbers
Swapping, since $5 < 8$

6	3	8	5	1
---	---	---	---	---

6	3	5	8	1
---	---	---	---	---

Sorting Algorithms

Bubble Sorting:

Comparing 4th & 5th numbers
Swapping, since $1 < 8$

6	3	5	8	1
6	3	5	1	8

Now last position carries the right element.

Performing 2nd Iteration:

Comparing 1st & 2nd numbers
Swapping, since $3 < 6$

6	3	5	1	8
6	3	5	1	8
3	6	5	1	8
3	6	5	1	8

Sorting Algorithms

Bubble Sorting:

Comparing 2nd & 3rd numbers
Swapping, since $5 < 6$

3	6	5	1	8
3	5	6	1	8

Comparing 3rd & 4th numbers
Swapping, since $1 < 5$

3	5	6	1	8
3	5	1	6	8

Now last-1
position carries
the right element.

Check the list if one swap at least was done run the 3rd iteration.

Sorting Algorithms

Bubble Sorting:

Performing 3rd Iteration:

Comparing 1st & 2nd numbers
No Swapping, since $5 > 3$

3	5	1	6	8
---	---	---	---	---

3	5	1	6	8
---	---	---	---	---

3	5	1	6	8
---	---	---	---	---

Comparing 2nd & 3rd numbers
Swapping, since $1 < 5$

3	5	1	6	8
---	---	---	---	---

3	1	5	6	8
---	---	---	---	---

Now last-2
position carries
the right element.

Check the list if one swap at least was done run the 3rd iteration.

Sorting Algorithms

Bubble Sorting:

Performing 4th Iteration:

Comparing 1st & 2nd numbers
Swapping, since $1 < 3$

3	1	5	6	8
3	1	5	6	8
1	3	5	6	8

Now last-3
position carries
the right element.

Check the list if one swap at least was done run the 3rd iteration.

Sorting Algorithms

Bubble Sorting:

Performing 5th Iteration:

1	3	5	6	8
---	---	---	---	---

Now last-4 (array 1st)position carries the right element.

There is no swap done. So, the list is sorted

Sorting Algorithms

Bubble Sorting :

```
#include <iostream>
// stopping condition is no more swapping
void bubble_sort (int *a, int n){
    int swapped; int i, j;
    for (i = 1; i < n; i++) {
        bbswapped = 0; // this flag is to check if the array
                        // is already sorted
        for(j = 0; j < n - i; j++) {
            if(a[j] > a[j+1]) {
                swap(a[j],a[j+1]);
                swapped = 1; }
        } if(!swapped) break; // if it is sorted then stop
    }
} // use the early main for Test
```

C++ code implementation

Sorting Algorithms

Bubble Sorting :

Performance Analysis

No. of Comparisons = $(n - 1) + (n - 2) + \dots + 1$

$$\frac{1}{2} (n^2 - n)$$

1. Worst Case $O(n^2)$
2. Average Case $O(n^2)$

Where, n is the number of items being sorted

Sorting Algorithms

Insertion Sorting:

The **Insertion Sort Algorithm** is the simplest sorting algorithm that is appropriate for **small inputs**

- Most common sorting technique used by card players.
- Insertion Sort takes advantage of **presorting**.
- It requires fewer comparisons than bubble sort.
- Insert the number in **its location** and shift all of the next elements.

Sorting Algorithms

Insertion Sorting:

Performance Analysis

1. Insertion Sort is somewhat similar to the Bubble Sort in that we compare adjacent elements and swap them if they are out-of order.
2. Unlike the Bubble Sort however, we do not require that we find the next largest or smallest element.
3. Instead, we take the next element and insert it into the sorted list that we maintain at the beginning of the array.

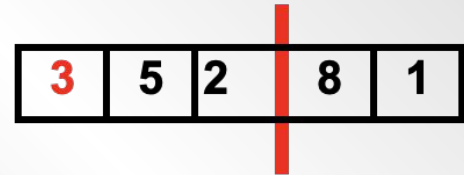
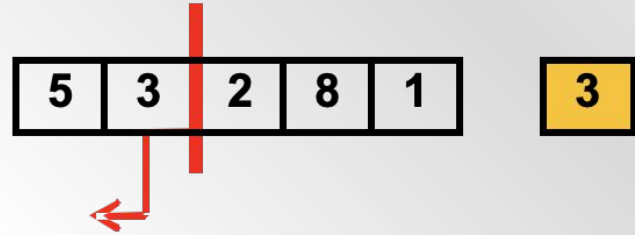
.

Sorting Algorithms

Insertion Sorting:

– 1st Iteration

1. Compare with 5
2. Swap
3. Go to next iteration

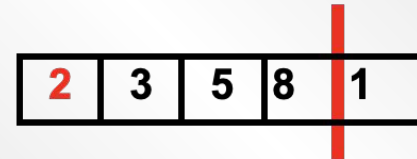
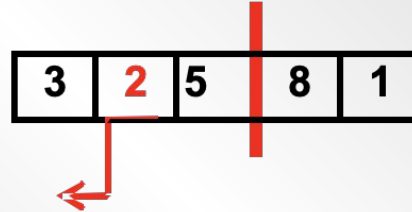
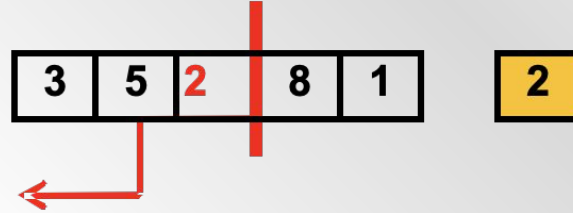


Sorting Algorithms

Insertion Sorting:

– 2nd Iteration

1. Compare with 5
2. Swap
3. Compare with 3
4. Swap
5. Go to next iteration

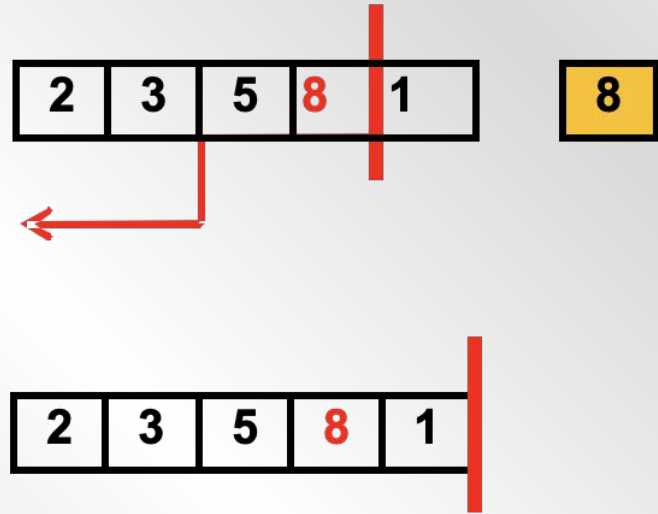


Sorting Algorithms

Insertion Sorting:

– 3rd Iteration

1. Compare with 5
2. No Swap
3. Go to next iteration

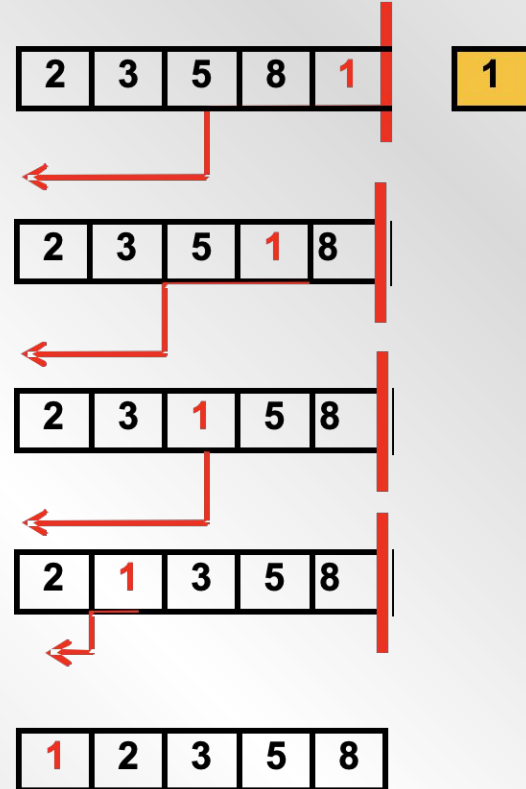


Sorting Algorithms

Insertion Sorting:

– 4th Iteration

1. Compare with 8
2. Swap
3. Compare with 5
4. Swap
5. Compare with 3
6. Swap
7. Compare with 2
8. Swap



Sorting **Algorithms**

Insertion Sorting :

Performance Analysis

Running time depends on not only the size of the array but also the contents of the array.

What is the Big O Notation ?

Searching **Algorithms**



searching **Algorithms**

Searching Algorithms:

Searching is to find the location of specific value in a list of data elements.

- Searching Methods can be divided into two categories:
- Searching methods for unsorted as well as sorted lists.
 - e.g., Sequential Search.
- Searching methods for sorted lists.
 - e.g., Binary Search.
- Direct access by key value (hashing)

searching **Algorithms**

Sequential Search Algorithm :

Is also known as Linear Search.

- can be used to search both sorted and unordered lists.
- operates by checking every element of a list one at a time in sequence until a match is found.
- Algorithm Procedure:
 1. For each item in the list, check if the item we are looking for matches the item in the list as follows:
 1. If it matches, return the location where the item is found(i.e., the item index) and end the search.
 2. Otherwise, continue searching until a match is found or the end of the list is reached.
 2. If the end of the list is reached without finding a match, then the Item does not exist in the list.

searching Algorithms

Sequential Search Algorithm:

Initial state : find 9

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Unsorted list

Is 1st element 6 == 9 ? False

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Is 2nd element 8 == 9 ? False

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Is 3rd element 3 == 9 ? False

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Is 4th element 5 == 9 ? False

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Is 5th element 1 == 9 ? False

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Is 6th element 4 == 9 ? False

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

Is 7th element 9 == 9 ? True [stop here]

6	8	3	5	1	4	9	2	7
---	---	---	---	---	---	---	---	---

searching **Algorithms**

Sequential Search Algorithm:

```
#include <iostream>

int main() {
    int i,n; int i, n;
    int a[9]={6,8,3,5,1,4,9,2,7};
    cout << "Found at : "<< sequential_search (a, 9,9);
}
```

C++ code implementation

searching **Algorithms**

Sequential Search Algorithm:

```
int sequential_search (int *a, int n, int num)
{
    int i = 0, found = 0;
    while ((!found) && (i < n)) {
        if ( num == a [i])
            found = 1;
        else
            i++;
    }
    if (found)
        return i;
    else
        return -1;
}
```

searching **Algorithms**

Sequential Search Algorithm:

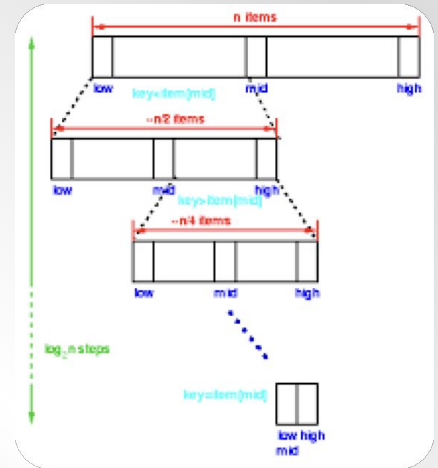
Performance Analysis

- Best Case = 1 (Only, one element will be tested).
- Worst Case = n (All n elements will be tested).
- Time Complexity: $O(n)$.

searching **Algorithms**

Binary Search Algorithm:

- Binary Search is fast searching algorithm, but it can be used only to search sorted lists.
- The Binary Search method uses the “divide-and-conquer” approach.



Searching **Algorithms**

Binary Search Algorithm:

Algorithm Procedure:

- we need three indexes:
 1. high: index of the last element in the array.
 2. low: index of the first element in the array.
 3. mid: index of the middle element $(\text{high} + \text{low})/2$

Searching **Algorithms**

Binary Search Algorithm:

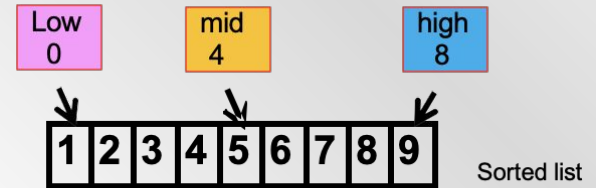
Algorithm Procedure:

- First test the middle element:
 1. If the element equals the key , stop and return mid (index of middle).
 2. If the element is larger than the key, then test the middle element of the first half;
 3. Otherwise, test the middle element of the second half.
- Repeat this process until either a match is found, or there are no more elements to test.

Searching Algorithms

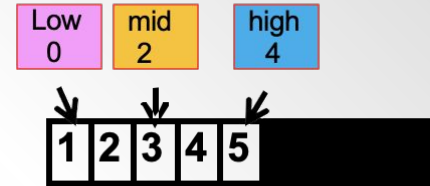
Binary Search Algorithm:

Initial state : find 4



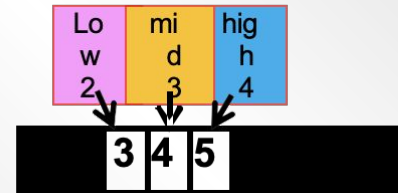
mid value 5 > 4

Consider only **right half**
mid value 3 < 4



Consider only **left half**

mid value 4 == 4 **found stop**



Lab **Exercise**

Assignments :

Performance Analysis

1. Implement insertion Sort on array of Employees
2. Calculate the time complexity to the following code
- 3. Implement Binary search on array of Employees**
4. **Search:** Merge Sort & Quick Sort

```
int i = 1;
int sum = 0;
while(i<=n)
{
    int j = 1;
    while(j <= n)
    {
        sum = sum + i;
        j = j + 1;
    }
    i = i + 1;
}
```