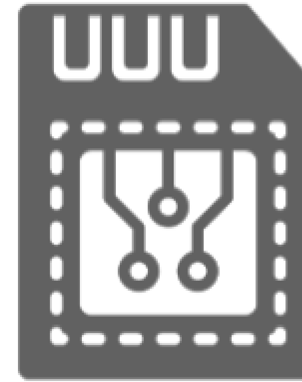
Time Complexity



Space Complexity

CS 2124: Data Structures
Spring 2024

Lecture 2 (Part – I)

Topics: Time and Space Complexity, Runtimes, Sorting, Searching

Topics

- Time and Space Complexity
- Introduction to Asymptotic Notations
- Big O notation
- Searching
  - Binary Search
  - Linear Search

- Assignment – 1
  - Assigned: 22nd Jan 2024
  - Due: 29th Jan 2024

Time and Space Complexity

- As a human we tend to improve/optimize or find efficient ways to perform a task.
  - i.e. making a cup of tea/coffee or reaching university for a class in both case we try to improve the time and effort require to complete the task.

- **Time Complexity:** Time take by an algorithm for execution.
  - Technically, it is the process of determining how the processing (execution) time increases as the size of the problem (input size) increase
  - Generally the time complexity is expressed by keeping only the values which affects most.

  - ***For example***, if the time complexity for a program needs to be calculated as a *function of* ***n (i.e.*** $n^4+n^3+n^2+n \approx n^4$***)*** as all the terms are small and they add lesser (too less) effect in the overall computation when compared with **n^4.**
  - i.e. The equation $5n^4 + 10n^3 + 100 + 100n$ has a time complexity of = ?

Time and **Space** Complexity

- As a human we tend to improve/optimize or find efficient ways to perform a task.
  - i.e. making a cup of tea/coffee or reaching university for a class in both case we try to improve the time and effort require to complete the task.

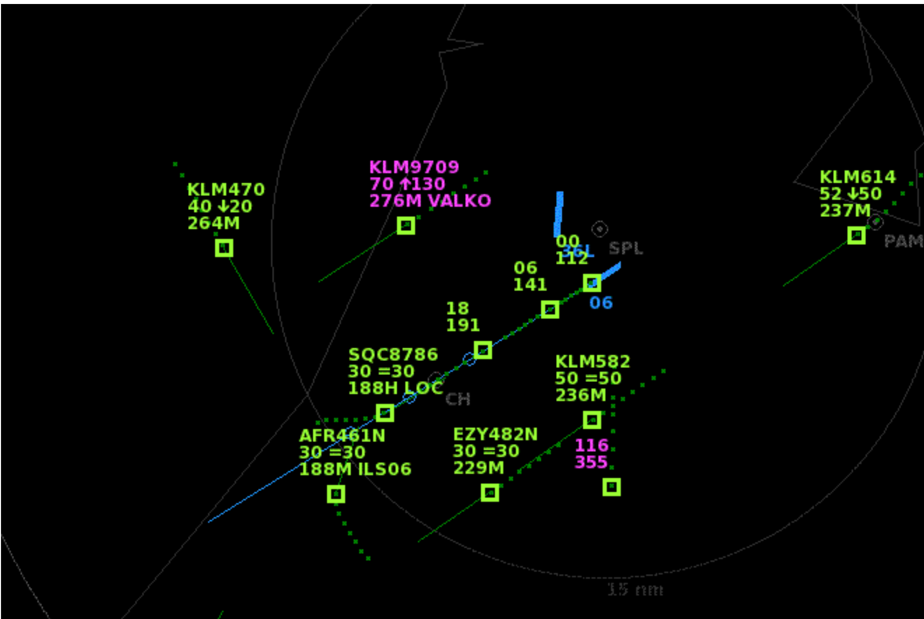
- **Space Complexity:** Memory require by an algorithm to execute a program.
  - Space complexity is the total amount of memory space used by an algorithm/program including the space of input values for execution.

Space Complexity *= Auxiliary space (Auxiliary space is just a temporary or extra space) + Space use by input values*

Time and Space complexity

Importance

- Importance of time and space complexity when it comes to programming.
  - i.e. Air traffic controlling and monitoring program, Intrusion detection program (IDS), fire alarm etc.

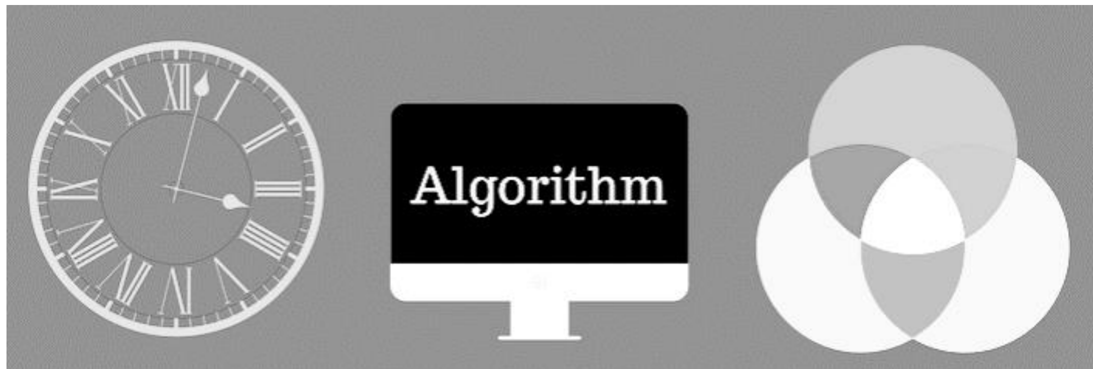


Air Traffic Controller Radar

Data Structures

Time and Space complexity

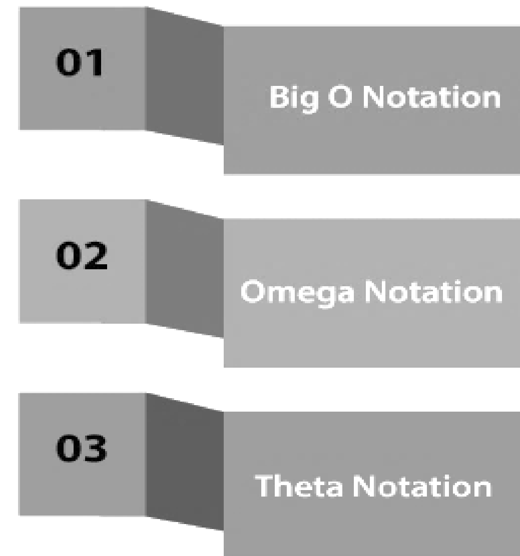
- Data Structures are necessary for designing efficient algorithms.
  - It provides reusability and abstraction.
  - Using appropriate data structures can help programmers save a good amount of time while performing operations such as storage, retrieval, or processing of data.
  - Helps in optimizing data manipulation (i.e. add, remove, edit large amounts of data).



Introduction to Asymptotic Notations

- In simple words, it tells us how good an algorithm is when compared with another algorithm.
- Parameters can play a part i.e. hardware used for implementation, Operating System, CPU model, processor generation, etc.
- Therefore, we use Asymptotic analysis to compare space and time complexity.
- It analyzes two algorithms based on changes in their performance concerning the increment or decrement in the input size

- **Big O (O())** describes the upper bound of the complexity.
- **Omega (Ω())** describes the lower bound of the complexity.
- **Theta (Θ())** describes the exact bound of the complexity.



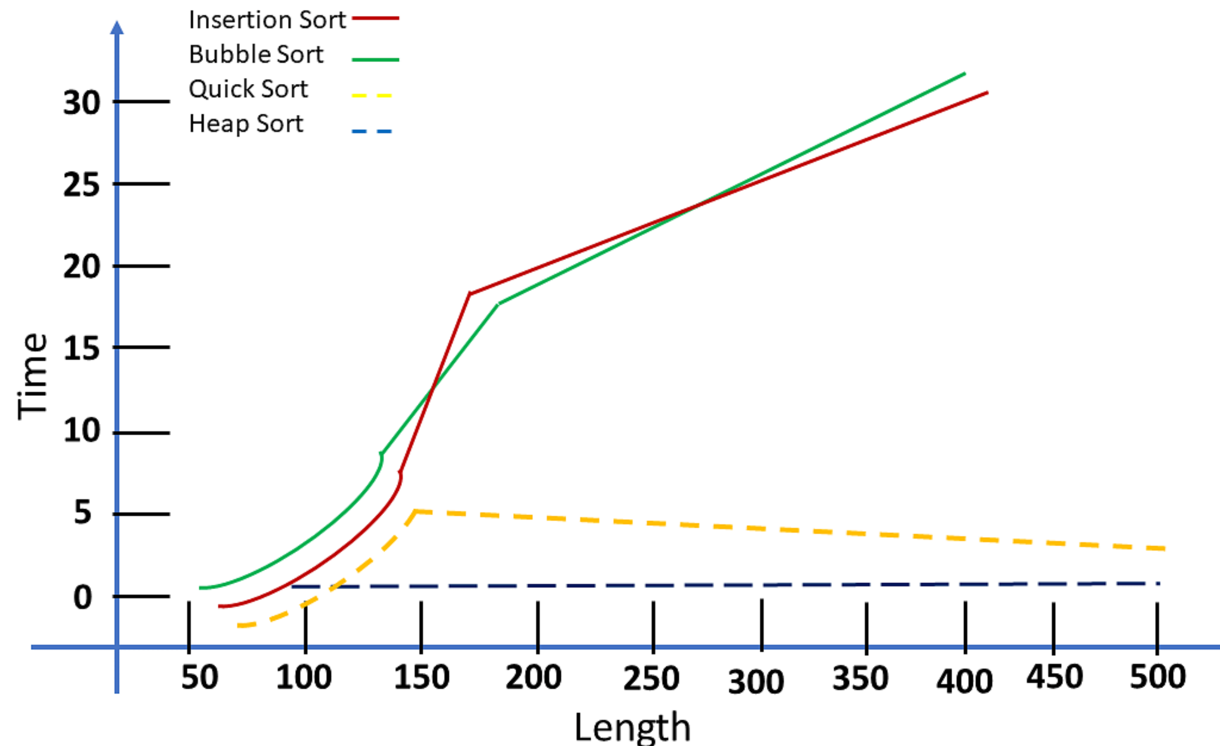
Asymptotic = Approaching a value or curve arbitrarily closely (i.e. as some sort of limit is taken)

Introduction to Asymptotic Notations

- Run time usually depends on the size of the input

$$\boldsymbol{T(n)}: the\ time\ taken\ on\ an\ imput\ of\ size\ n.$$

- Asymptotic analysis consider the growth of $T(n)$ as n goes to infinity.



Introduction to Asymptotic Notations

- **Big O** is also known as the algorithm's upper bound since it analyses the worst-case situation.

- The best-case scenario are not considered to be used in a comparative analysis. That’s why we employ worst-case scenarios to get meaningful input.

- The algorithm in data structure while programming code is critical. Big O notation makes it easier to compare the performance of different algorithms and figure out which one is best for your code.

Big O notation

- The big O notation, $\boldsymbol{O(g(n))}$, is a collection of functions.
- A function $\boldsymbol{f(n)}$ is a member of that collection only if it fits the following criteria:

$$\boldsymbol{Constant\ c\ and\ n_0\ exist\ where\ f(n) \leq c.g(n)\ for\ all\ n \geq n_0}$$

- So, when an algorithm performs a computation on each item in an array of size $\boldsymbol{n}$, it takes $\boldsymbol{O(n)}$ time and performs $\boldsymbol{O(1)}$ work on each item
- This can be written as:

$\boldsymbol{f(n)} = \boldsymbol{O(g(n))}$, where $\boldsymbol{n}$ tends to infinity ($\boldsymbol{n} \rightarrow \infty$)

- We can simply write the above expression as:

$$\boldsymbol{f(n)=O(g(n))}$$

Later we will look in to some examples to make it clear

But why do we need Big O?

- The world we live in today consists of complicated apps and software, each running on various devices and each having different capabilities.
- Some devices like desktops can run heavy machine learning software, but others like phones can only run apps.
- So when you create an application, you’ll need to optimize your code so that it runs smoothly across devices to give you an edge over your competitors.

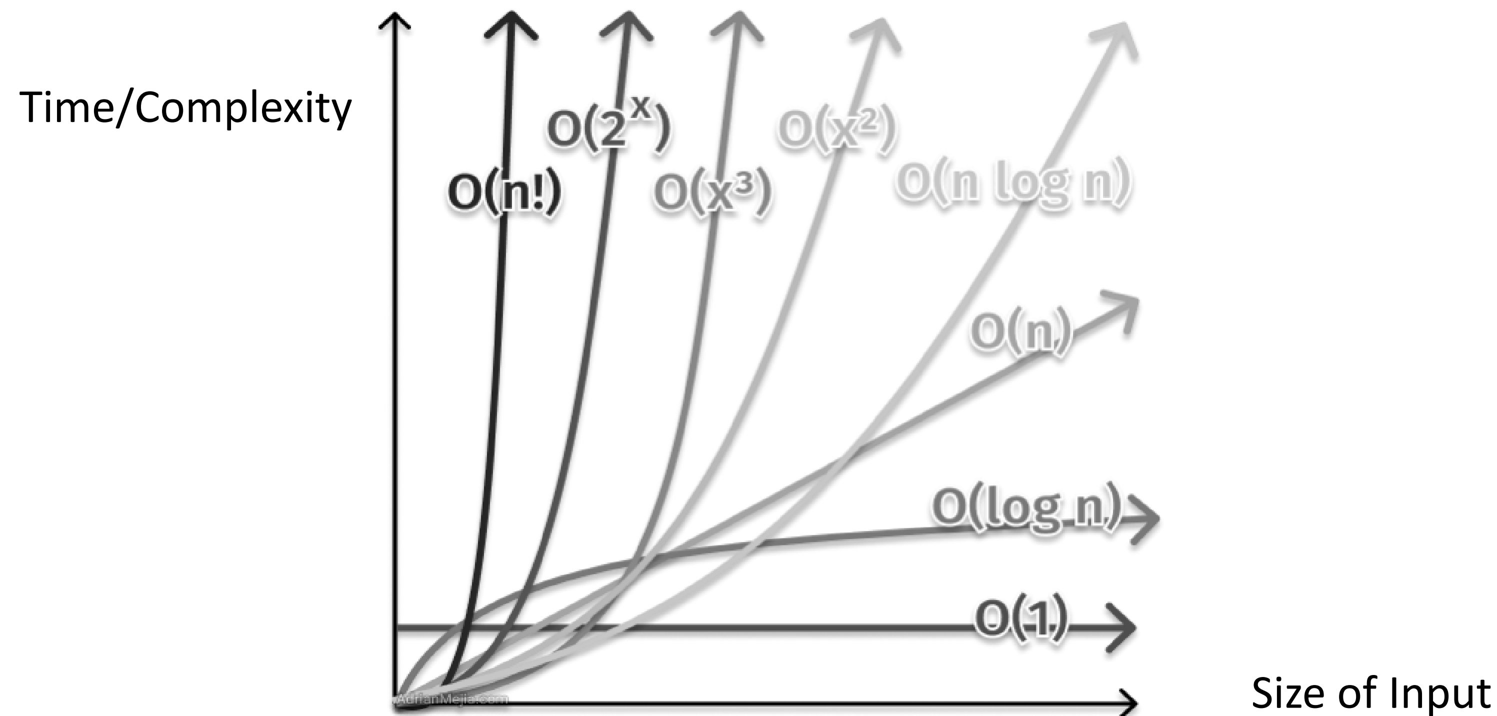
- As a result, programmers should inspect and evaluate their code thoroughly.

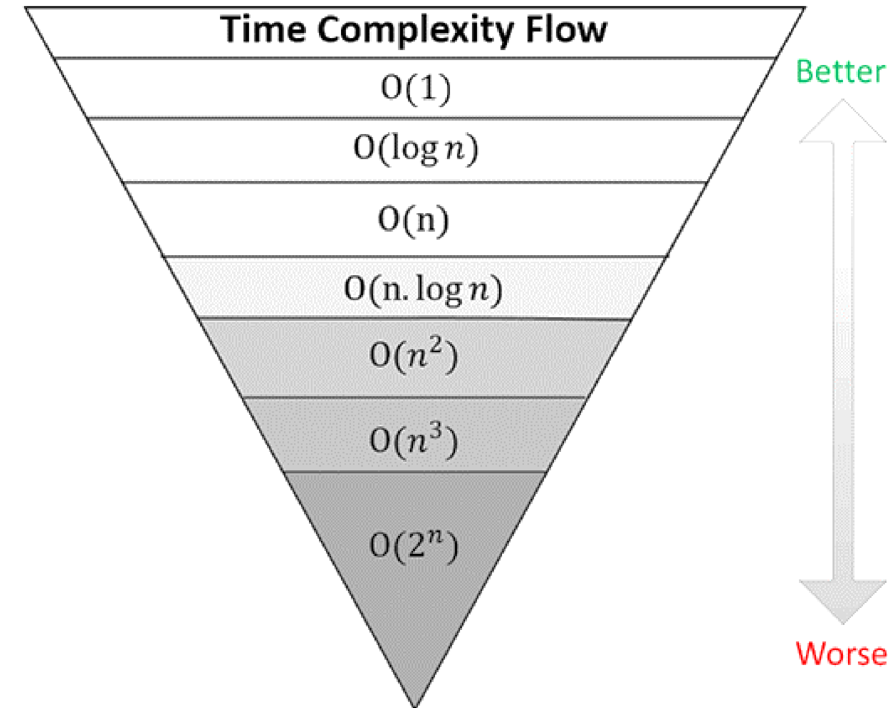
Big O notation (Steps)

- The Big-O Asymptotic Notation gives us the Upper Bound Idea. The general step wise procedure for Big-O runtime analysis is as follows:
    1. Figure out what the input is and what **‘n’** represents (i.e. $\boldsymbol{f}(\boldsymbol{n})$=$\boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n}))$.
    2. Then identify the maximum number of operations, the algorithm performs in terms of ‘**n**’. (i.e. An addition of two numbers is just 1 operation)
    3. Eliminate all excluding the highest order terms. (i.e. if you have n^4 and n^3 consider only n^4)
    4. Remove all the constant factors. As constants will remain constant regardless of the user input

- Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically. For any algorithm, the Big-O analysis should be straightforward as long as we correctly identify the operations that are dependent on ‘**n’**, the input size.
- In general cases, we mainly consider the worst-case (theoretical running time complexities) of algorithms for the performance analysis.

Big O notation	Example
Constant:O(c)	O(1)
Logarithmic time: O(log n)	n=20, means log (20) = 2.996
Linear time: O(n)	n=20, means 20
Logarithmic time: O(n log n)	n=20, means 20 log(20) = 59.9
Quadratic time: O(n^2)	n=20 mean, 20^2 = 400
Exponential time: O(2^n)	n=20 means, 2^{20} = 1084576
Factorial time: O(n!)	n=20 means, 20!





Big O notation (Example)

```c
#include <stdio.h>

int main()
{
    int n;
    printf("N = ");
    scanf("%d", &n); //input 3
    int a[n];
for(int i=0; i<n; i++)
    printf("a[%d] = %d \n", i, a[n]=i-1);
}
```

Line	Count
Line 5:	1
Line 6:	1
Line 7:	1
Line 8:	1
Line 9:	1+(n+1)+n
Line 10:	n+1

Big O = ?

Output ?

Ignore constant multiplier so O(n)

Big O notation

```c
int main()
{
    int arr[] = {0};
    printf("First element of array = %d",arr[0]);
}
```

This function runs in $\boldsymbol{O(1)}$ time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

```c
#include <stdio.h>

int main()
{
    int arr[]= {1,2,3,4,5};
    for (int i = 0; i < 5; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

This function runs in $\boldsymbol{O(n)}$ time (or "linear time"), where **n** is the number of items in the array (i.e $\boldsymbol{O(5)}$).

Try to implement the code

Big O notation

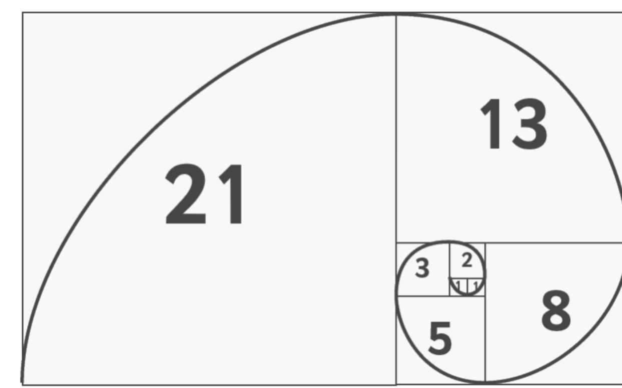
```c
#include <stdio.h>

int main()
{
    int arr[]= {1,2,3,4,5};
    int t=0;
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
            printf("%d\n", t++);
        }
    }
}
```

Here we're nesting two loops. If our array has **n** items, our outer loop runs **n** times and our inner loop runs **n** times for each iteration of the outer loop, giving us $\boldsymbol{n^2}$ total prints. Thus this function runs in $\boldsymbol{O(n^2)}$ time (or "quadratic time")

Try to implement the code

Big O notation




```c
#include <stdio.h>

int main() {
  //Fibonacci sequence = each number is the sum of the two preceding numbers
  int t1 = 0, t2 = 1, nextTerm = 0, n;
  printf("Enter a positive number: ");
  scanf("%d", &n);

  // displays the first two terms which is always 0 and 1
  printf("Fibonacci Series: %d, %d, ", t1, t2);
  nextTerm = t1 + t2;

  while (nextTerm <= n) {
    printf("%d, ", nextTerm);
    t1 = t2;
    t2 = nextTerm;
    nextTerm = t1 + t2;
  }

  return 0;
}
```

Exponential time: $\boldsymbol{O}(2^n)$

- Recursive calculation of Fibonacci numbers. $\boldsymbol{O}(2^n)$ denotes an algorithm whose growth doubles with each addition to the input data set.
- The growth curve of an $\boldsymbol{O}(2^n)$ function is exponential.
- It starts off very shallow, then rising exponentially.

- What will be the output if input is 3?

Big O notation

```c
#include <stdio.h>

int main() {
{
    int arr[]={1,2};
    int size=2;
    for (int i = 0; i < size; i++)
    {
        printf("Loop one: %d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        printf("Loop two: %d\n", arr[i]);
    }
}
  return 0;
}
```

When you're calculating the big O complexity of something, you just throw out the constants:
$\boldsymbol{O(2n)} = \boldsymbol{O(n)}$

Try to implement the code

Big O notation

```c
#include <stdio.h>

int main()
{
int n=4;
while(n >1)
{
printf("Big Problems with Big O notations!\n" );
n = n/2;
}
}
```

As the program is following n/2 (i.e. n is the input)
Logarithmic time: $\boldsymbol{O}(\boldsymbol{log_2}\ \boldsymbol{n})$

Try to implement the code

Searching

- Searching in data structure refers to the process of finding location of an element in a list.
- This is one of the important parts of many data structures algorithms, as one operation can be performed on an element if and only if we find it.
- We do not want searching to take **'n' steps for searching an array of 'n' number of elements**.
  - In some cases we are bound to take 'n' steps
  - But different algorithms try to minimize the number of steps to search an element

- Why optimize searching algorithms ?

Website Statistics (Top Picks)

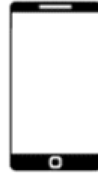
As of 2023, the number of websites worldwide has exceeded two billion

U.S. internet users view 138.1 webpages per day on an average.

Video strategy is seen by 96% of marketers as a way to enhance user comprehension

The vast majority (93%) of global web traffic originates from Google.

Website Traffic Statistics



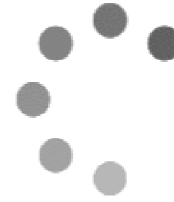
As of April 2023, more than half of the traffic is from mobile devices

Google is ranked as the most popular website, having 88.6 billion visitors

YouTube has over 32 billion visitors, making it the 2nd most visited website

Website User Experience Statistics

Internet users expect a website to load in less than 2 seconds



Retailers lose $2.6 billion annually due to slow website loading speeds

The avg time spent looking at the website's navigation menu is 6.44 sec

Images Source: [Link]

Binary Search

- In the sequential search, when we compare against the first item, there are at most more items to look through if the first item is not what we are looking for.

  - Instead of searching the list in sequence, a binary search will start by examining the middle item.
  - If that item is the one we are searching for, we are done.
  - If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items.
  - If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration.
  - The item, if it is in the list, must be in the upper half.

Search = 7
Sorted Array

Mid Value = 6
$6 < 7$
Search the Right side

Mid Value = 8
$7 < 8$
Search the Left side

Binary Search

Algorithm

1. binarySearch(arr, x, low, high)
2. repeat till low = high
3. mid = (low + high)/2
4. if (x == arr[mid])
5. return mid
6. else if (x > arr[mid]) // x is on the right side
7. low = mid + 1
8. else // x is on the left side
9. high = mid - 1

Pseudocode

1. Set two variables. min = 0 and max = n - 1.
2. Find the mid value between min and max by averaging min and max and rounding it down.
3. If array[mid] === target, return mid.
4. If array[mid] < target, set min = mid + 1.
5. Otherwise set max = mid - 1.
6. Go back to step 2.

Double the size of the array, need at most one more guess.

Search for 47

Iterative Approach:

0	4	7	10	14	23	45	47	53

Important: For Binary search, array must be sorted

Binary Search

```
1  #include <stdio.h>
2  int main()
3  {
4  int i, low, high, mid, n, key, array[100];
5      printf("Enter number of elements:");
6      scanf("%d",&n);
7      printf("Enter %d integers:", n);
8  for(i = 0; i < n; i++)
9      scanf("%d",&array[i]);
10 printf("Key value to find:");
11 scanf("%d", &key);
12     low = 0;
13     high = n - 1;
14     mid = (low+high)/2;
15 while (low <= high) {
16     if(array[mid] < key)
17 low = mid + 1;
18     else if (array[mid] == key) {
19 printf("%d found at location %d.", key, mid+1);
20     break;
21 }
22 else
23     high = mid - 1;
24     mid = (low + high)/2;
25 }
26 if(low > high)
27 printf("Not found! %d isn't present in the list.", key);
28 }
```

Binary Search (For better understanding)

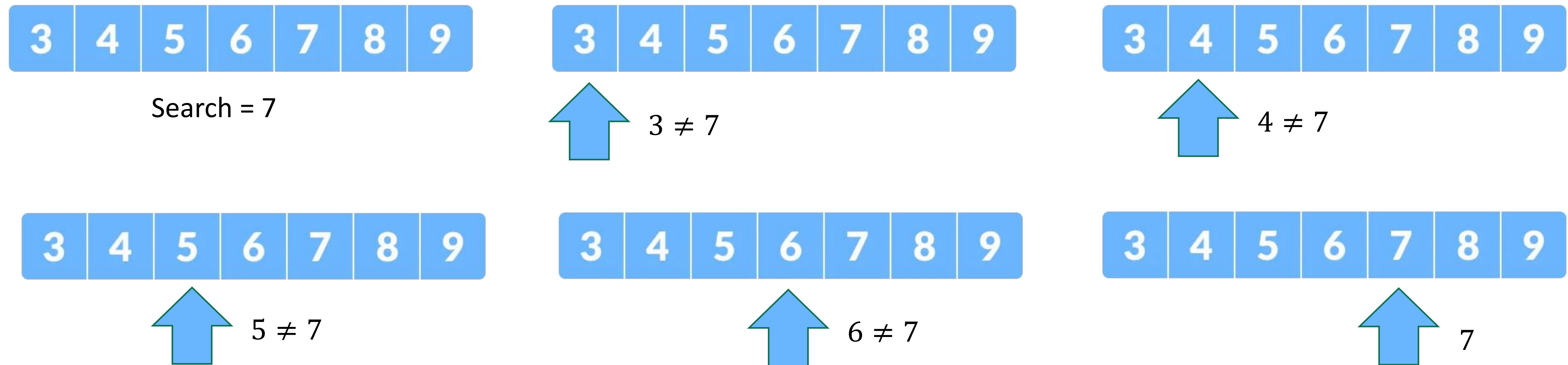
```
#include <stdio.h>
int main()
{
int i, low, high, mid, n, key, array[100];
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Enter %d integers:", n);
for(i = 0; i < n; i++)
    scanf("%d",&array[i]);
printf("Key value to find:");
scanf("%d", &key);
    low = 0;
    high = n - 1;
    mid = (low+high)/2;
       printf("\n High:%d [%d], Mid: %d [%d] \n", high, array[high], mid, array[mid]);
low = mid + 1;
    else if (array[mid] == key) {
printf("%d found at location %d.", key, mid+1);
    break;
}
else
    high = mid - 1;
    mid = (low + high)/2;
}
if(low > high)
printf("Not found! %d isn't present in the list.", key);
}
```

Index Value

Element at that index value

Linear Search

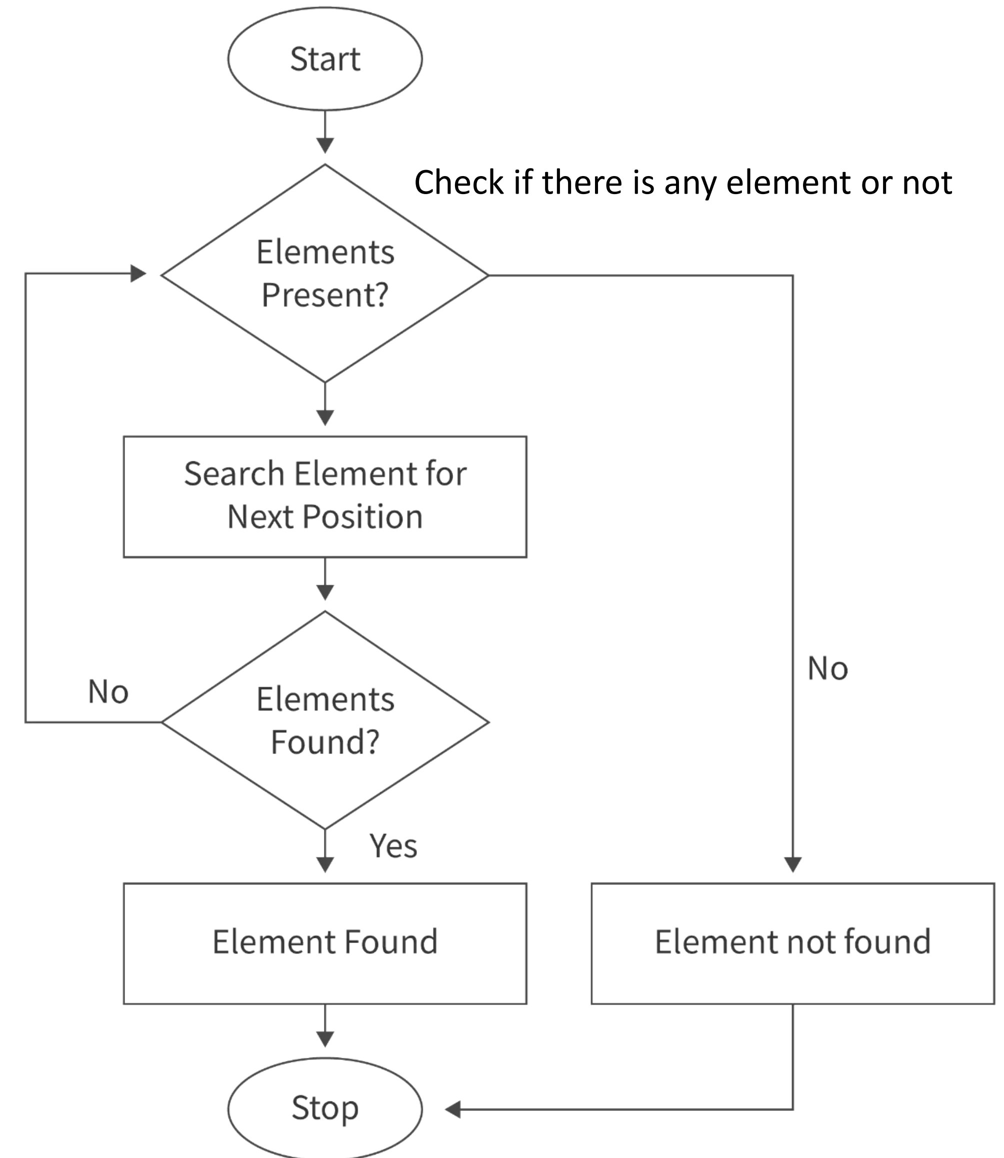
- Linear Search (sequential search) algorithm starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



Linear search ***do not require*** *the elements to be sorted*

Linear Search

1. linear_search (list, value)
2. for each item in the list
3. if match item == value
4. return the item's location
5. end if
6. end for
7. end procedure



Linear Search

```c
#include <stdio.h>
int search(int array[], int n, int x)
{
  for (int i = 0; i < n; i++)
    if (array[i] == x)
      return i;
  return -1;
}
int main() {
  int array[] = {2, 4, 0, 1, 9};
  int x;
  printf("Enter Number to search:");
  scanf("%d",&x);
  //int x = 1;
  int n = sizeof(array) / sizeof(array[0]);
  int result = search(array, n, x);
 if (result == -1)
 printf("Element not found");
 else
 printf("Element found at index: %d", result);
}
```

End of Lecture 2 (Part – I)

Do try to implement the codes by your self to better understand the working

- Assignment – 1 (Plagiarism check is enable on the canvas)
  - Assigned: 22nd Jan
  - Due: 29th Jan (End of day as per Canvas)