Intro Sorting and Searching Algorithm

Welcome to our fourth super skill :)
By now, we have learned the different data structures that exist in programming, also we have seen to divide our algorithm into procedure and function.

It's time to learn some famous algorithms of searching and sorting.
So during this chapter, we are going to see:

- What's BigO?
- What are the algorithms of sorting, and their types?
- What are the algorithms of searching, and their types?

Big O

Big O definition in Wikipedia is the next :

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

- Do not worry, I know that is ambiguous for you, here I will make it more simple, Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

- In other words, it is an approximation value that can gives us an idea about the performance of our algorithm. It helps us know about efficiency and complexity of an algorithm without the need to actually it with test inputs on a real machine.

Big O

- Big O is not a speed
-

, it does not allow you to compare the speed of two algorithms. Rather, it is used to determine what factor the number of operations will grow by as the amount of input increases.

- The notation is useful when comparing algorithms used for similar purposes. Seeing the Big O of multiplication algorithms would allow you to compare their relative complexity. It wouldn't be useful to see the Big O of a multiplication algorithm and an addition algorithm side-by-side.
- Big O is a simple approximation of the worst-case-scenario, it reduces an algorithm to the most significant portion of complexity. If an algorithm could be represented as $n^2 + n$, as the inputs became large, the + n would become meaningless in the overall complexity and could be dropped. The same logic

holds true for n3 + n² + n, as this would be simplified to n3. In the next skill, we will understand more about how to be calculated.

- ## How to calculate Big O.
- Remember, all calculations of the Big O is for the worst-case-scenario. This means it is possible that the algorithm would return results earlier than the approximation.
- Let's practice!
  In the code box below, we will find an algorithm that returns true if a given element exists in a given array, and false if not.

```
FUNCTION seek_elt(tab : ARRAY_OF INTEGER , val : INTEGER)
: BOOLEAN
VAR
   i : INTEGER;
BEGIN
  FOR i FROM 0 TO tab.length-1 STEP 1  DO
      IF (val = tab[i]) THEN
          RETURN TRUE
      END_IF
  END_FOR
  RETURN FALSE
END
```

- As we can see, the tab.length is our n (or input). Now, the main idea behind the BigO is to find the worst-case scenario. Returning to our example, the worst-case scenario is that we browse the whole array and we do not find the searched element. So our complexity is O(n)
  Let's take a second example. This time we will have a nested loop.

```
FUNCTION contain_dup(tab : ARRAY_OF INTEGER) : BOOLEAN
VAR
  i,j : INTEGER;
BEGIN
  FOR i FROM 0 TO tab.length-1 STEP 1  DO
       FOR j FROM 0 TO tab.length-1 STEP 1  DO
           IF (i<>j AND tab[i] = tab[j]) THEN
               RETURN TRUE
           END_IF
       END_FOR
  END_FOR
   RETURN FALSE ;
END
```

- in the example above, we have a nested loop with each one of these loops has a complexity of O(n). So the whole complexity of our algorithm will be O(n * n) which equal to O(n²)
  Our third example will be the same algorithm as before but with a little change. We'll make the second loop not starting from the beginning of the array, but it will start from the i position.

```
FUNCTION contain_dup(tab : ARRAY_OF INTEGER) : BOOLEAN
VAR
  i,j : INTEGER;
BEGIN
  FOR i FROM 0 TO tab.length-1 STEP 1  DO
```

```
•         FOR j FROM i+1 TO tab.length-1 STEP 1  DO
•             IF (i<>j AND tab[i] = tab[j]) THEN
•                 RETURN TRUE
•             END_IF
•         END_FOR
•    END_FOR
•     RETURN FALSE ;
•  END
```

- With this change, the first loop will keep the complexity of O(n), while the second loop will get a complexity of O(n - i).
  This is what we call the logarithmic format. So the complexity of this algorithm will be O(n log n)

Big O it an approximation of calculation of what ?

Best case scenario

Normal case scenario

Worst case scenario

Look at the loop, how many steps for the following snippet code?

```
FOR i FROM 0 TO N-1 STEP 2  DO
            sum := sum + tab[i];
        END_FOR
```

$O(N^2)$

O(N)

O(N/2)

O(log N)

What is the runtime of the following snippet code?

```
    WHILE (j < M) DO
            FOR j FROM 0 TO N  DO
```

```
                Write(i*j)
            END_FOR
            j := j+1;
        END_WHILE
```

$O(N^2)$

O(N * M)

O(M log N)

O(log N)

Simple Sorting

Simple Sorting

Simple Sort, also known as Bubble Sort, is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. The swapping is repeatedly passing through the list until it is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Recommended: Please take 20 minutes to solve it on your own first, write the algorithm before moving on to the solution at the end of this skill.

Simple Sorting

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 maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element from the unsorted subarray is picked up and moved to the sorted subarray.

Simple Sort Algorithms

Let’s see the example below.
We are going to implement the algorithm of the simple sort and bubble sort.

```
PROCEDURE swap(VAR xp, VAR yp : INTEGER)

VAR

   tmp : INTEGER;

BEGIN

  tmp := xp;

  xp := yp;

  yp := tmp;

END

/* *** Bubble sort *** */

PROCEDURE bubble_sort(VAR tab : ARRAY_OF INTEGER)

VAR

   i,j,n : INTEGER;

BEGIN

   n := tab.length;

   FOR i FROM 0 TO n- 1 STEP 1  DO

       // Last i elements are already in place

       FOR j  FROM 0 TO n-i-1 STEP 1  DO

           IF (tab[j] > tab[j+1]) THEN

               swap(tab[j], tab[j+1])

           END_IF

       END_FOR

   END_FOR

END
```

And this is the implementation of the selection sort algorithm.

```
PROCEDURE swap(VAR xp, VAR yp : INTEGER)

VAR

   tmp : INTEGER;
```

```
BEGIN
  tmp := xp;
  xp := yp;
  yp := tmp;
END
/* *** selection sort *** */
PROCEDURE selection_sort(VAR tab : ARRAY_OF INTEGER)
VAR
   i,j,min_idx,n : INTEGER;
BEGIN
   n := tab.length;
   // one by one move boundary of sub-array
   FOR i FROM 0 TO n-2 STEP 1  DO
       min_idx := i;
       FOR j  FROM i  TO n-i-2 STEP 1  DO
           IF (tab[j]<tab[min_idx]) THEN
               min_idx := j;
           END_IF
       END_FOR
   // swap the found minimum element with the first element
   swap(arr[min_idx],arr[i])
   END_FOR
END
```

What is the worst case complexity of bubble sort?

O(nlogn)

O(logn)

O(n)

O(n^2)

What is the advantage of selection sort over other sorting techniques?

It requires no additional storage space

It is scalable

It works best for inputs which are already sorted

It is faster than any other sorting technique

The given array is arr = {3,4,5,2,1}. The number of iterations in bubble sort and selection sort respectively are,

5 and 4

4 and 5

2 and 4

2 and 5

Merge sort

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists, until each sublist consists of a single element, merging then those sublists that will become a sorted list.

Quick Sorting

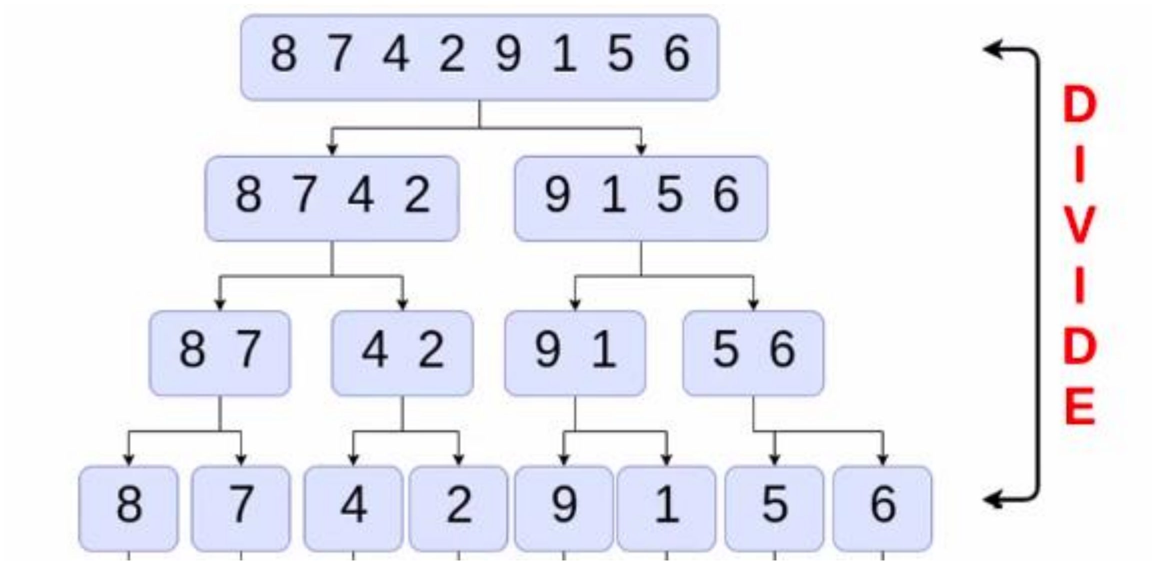
Like Merge Sort, Quick Sort is a Divide and Conquer algorithm, but even more efficient as we do not use the extra space to work with. It is the most commonly used algorithm in various programming languages. It picks an element that we call a pivot, then it rearranges them as such: as the smallest items are pivoted to the left, and the largest items are pivoted to the right, this is what we call partitioning. The array is then divided into two partitions separated by a pivot.

It does not matter how the elements are ordered in each partition, what matters is positioning all smaller elements for the pivot element and all the largest elements right after that. A partition with only one element is considered a sorted partition.

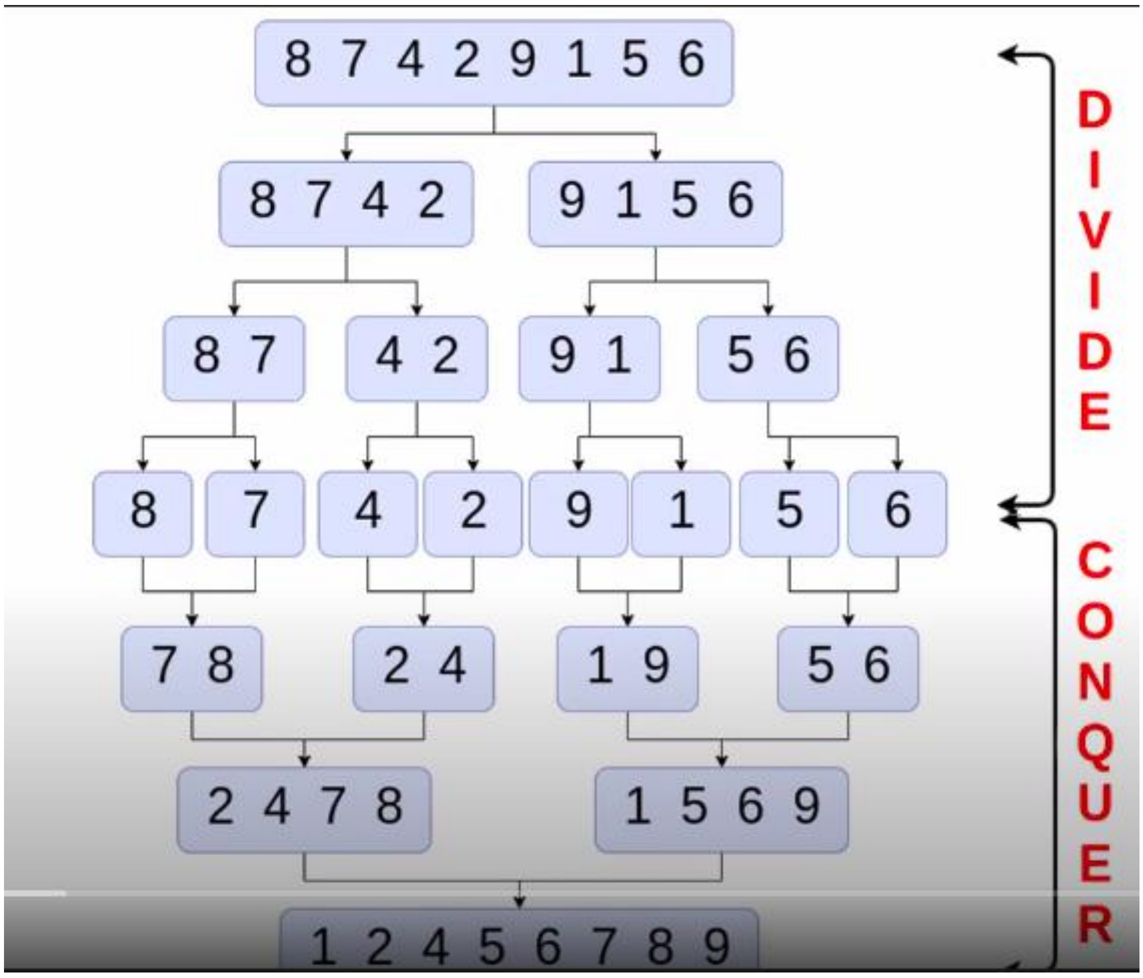
1. Always pick last element as a pivot (shown in the example)
2. Always pick first element as a pivot.
3. Pick a random element as a pivot.
4. Pick median as a pivot.

Quick Sorting Algorithms.

Now let's see the implementation of the merge sort, but before let's take a look at this array in the image below:



Do you remember the theory of divide and conquer, here we are going to use it again.
First, we are going to divide our array into smaller arrays to make sure that we get a single element in each array.
The next step in our algorithm is to merge each sub-array with one next to it in an ordered way.
We will take the image below into consideration :



Well, this is the implementation of the merge sort algorithm:

```
PROCEDURE merge_sort(VAR arr : ARRAY_OF INTEGER)

VAR

   i, m, mid, from, to, high, low : INTEGER;

BEGIN

   low := 0;

   high := arr.length-1;

   // divide the array into blocks of size m

   // m ={1,2,4,8,16,...}

   m := 1;

   WHILE (m<= high - low) DO

       // for m = 1; i={0,2,4,6,8,...}

       // for m = 2; i={0,4,8,...}
       // for m = 4; i={0,8,...}

       //....

       FOR i FROM low TO high-1 STEP 2*m  DO

           from := i;

           mid := i+m-1;

           to := min(i+2*m-1,high);

           merge(arr,from,mid,to);

       END_FOR

       m := 2*m;

   END_WHILE


END
```

This the merge procedure:

```
PROCEDURE merge(VAR arr : ARRAY_OF INTEGER, left, mid, right :
INTEGER)

VAR

   i,j,k : INTEGER;

   n1 : INTEGER := mid - left + 1;

   n2 : INTEGER := right - mid;

   L : ARRAY_OF INTEGER[n1];

   R : ARRAY_OF INTEGER[n2];

BEGIN

   // copy data to temp arrays L[] and R[]

  FOR i FROM 0 TO n1-1   DO

      L[i] := arr[left+i];

  END_FOR

  FOR j FROM 0 TO n2-1 DO

      R[j] := arr[mid+1+j];
  END_FOR
  // Merge the temp arrays back into arr[left .. right]
  i := 0;
  j := 0;
  k := left;
  WHILE (i<n1 AND j<n2) DO
      IF (L[i] <= R[j]) THEN
          arr[k] := L[i];
          i := i+1;
       ELSE
          arr[k] := R[j];
          j := j+1;
      END_IF
      k := k+1;
  END_WHILE
  /* Copy the remaining elements of R[], if there are any */
  WHILE (j < n2) DO
      arr[k] := R[j];
      j := j+1;
      k := k+1;
  END_WHILE
END
```

This is the quick sort algorithm:

```
/*
arr -> Array to be sorted
l-> starting index
h-> ending index
```

```
*/


PROCEDURE quick_sort(VAR arr : ARRAY_OF INTEGER)

VAR

  // Create an auxiliary stack

  stack : STACK;

  p : INTEGER;

BEGIN

   // pushing initial values of l and h to stack

   stack.push(l);

   stack.push(h);

   // keep popping from the stack while is not empty

   WHILE (NOT stack.isEmpty()) DO

       // pop h and l

       h := stack.pop();

       l := stack.pop();

       // set pivot element at its correct position

       // in a sorted array

       p := partition(arr,l,h);

       // if there are elements on the right side of the
pivot,

       // then push right side to stack

       IF (p+1 < h) THEN

           stack.push(p+1);

           stack.push(h);

       END_IF

   END_WHILE
```

```
END
```

And this is the implementation of the partition function:

```

/*
This function takes the last element as a pivot,
places its element at its correct position in sorted
array, and places all smaller (than the pivot) to left
of pivot and all the greater elements to right of pivot
*/


FUNCTION partition(arr : ARRAY_OF INTEGER, low, high :
INTEGER) : INTEGER

VAR

   b,i,pivot : INTEGER;

BEGIN

  pivot := arr[high]; //pivot

  i := low-1; // index of smaller element

  FOR i FROM low TO high-1 DO

       // if the current element is smaller than the pivot

       IF (arr[i] < pivot) THEN

           i := i+1;

           swap(arr[i],arr[i])

       END_IF

  END_FOR

   swap(arr[b+1],arr[i])


   RETURN b+1 ;

END
```

Which of the following sorting algorithms is the fastest?

Merge sort

Quick sort

Insertion sort

Shell sort

Find the pivot element from the given input using median partitioning method. 8, 1, 4, 9, 6, 3, 5, 2, 7, 0.

5

6

7

8

What is the average running time of a quick sort algorithm?

$O(N^2)$

$O(N)$

$O(N \log N)$

$O(\log N)$

Simple linear search

Not even a single day pass, when we do not have to search for something in our day to day life, car keys, books, pen, mobile charger and what not. Same is the life of a computer, there is so much data stored in it that whenever a user asks for some data, computer has to search its memory to look for the data and make it available to the user.

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. Interval Search: These algorithms are specifically designed for searching in sorted data structures. These types of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Simple linear search

Not even a single day pass, when we do not have to search for something in our day to day life, car keys, books, pen, mobile charger and what not. Same is the life of a computer, there is so much data stored in it that whenever a user asks for some data, computer has to search its memory to look for the data and make it available to the user.
Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. Interval Search: These algorithms are specifically designed for searching in sorted data structures. These types of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

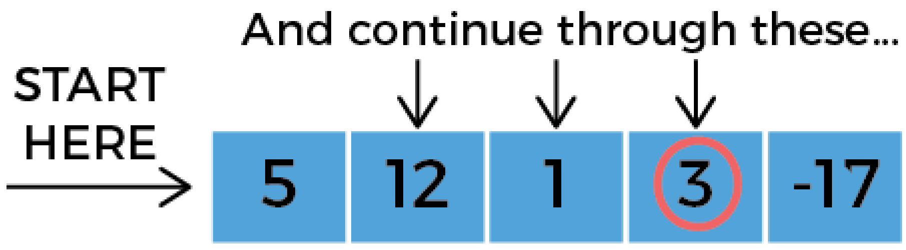
Linear search

Linear search is a very basic and simple search algorithm. In linear search, we search an element or value in a given array by traversing the array from the starting,

till the desired element or value is found.
It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.
Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good either.
3. It has a very simple implementation.



Linear search

The time complexity of the linear search algorithm is O(n).
This is its implementation:

```
FUNCTION linear_search(tab : ARRAY_OF INTEGER, elt : INTEGER)
: INTEGER

VAR

  j : INTEGER;

BEGIN

   j := 0;

   WHILE (j< tab.length) DO

      IF (tab[j] = elt) THEN

          RETURN j; // element is found let's break the loop
and return the index

      END_IF
```

```
        j := j+1; // update the index
    END_WHILE
    // we reached the end of array without finding the element
    RETURN -1 ;// means that we did not find the element
END
```

Where is linear searching used?

When the list has only a few elements

When performing a single search in an unordered list

Used all the time

When the list has only a few elements and When performing a single search in an unordered list

What is the best case and worst case complexity of ordered linear search?

O(nlogn), O(logn)

O(logn), O(nlogn)

O(n), O(1)

O(1), O(n)

Which of the following is a disadvantage of linear search?

Requires more space

Greater time complexities compared to other searching algorithms

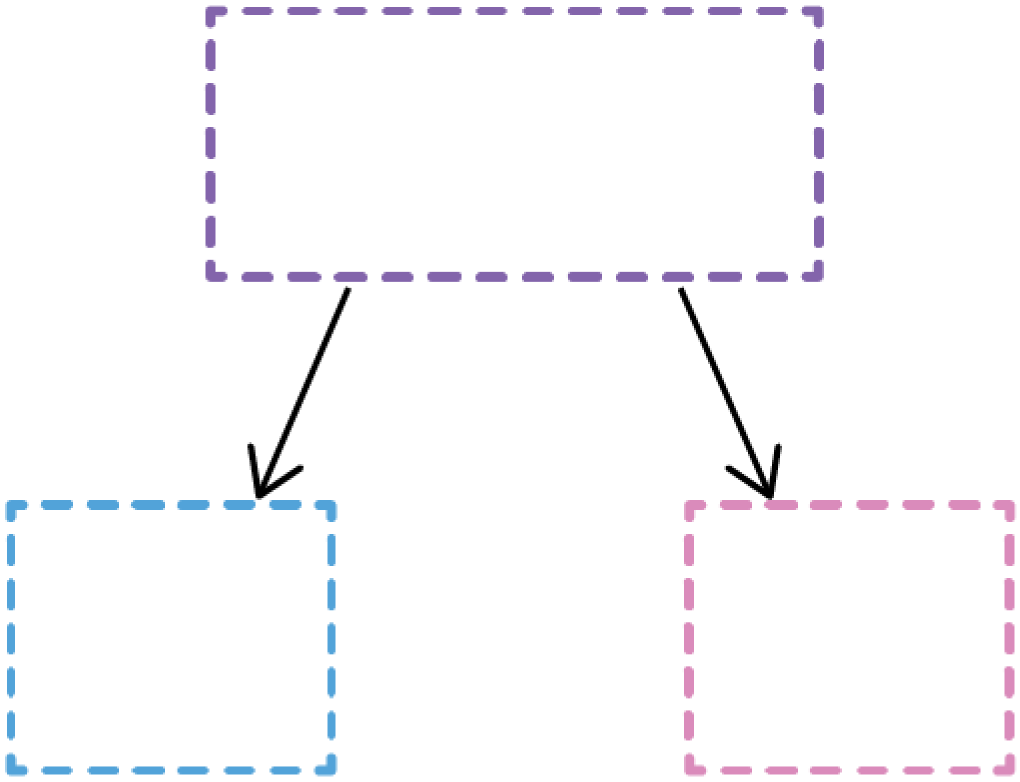
Not easy to understand

Not easy to implement

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched has a greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.
   Binary Search is useful when there are large number of elements in an array and they are sorted. So a necessary condition for the Binary search to work is that the list/array should be sorted.



Binary Search

Binary search is In this example we are looking for the value 6 in a array of integer.

Features of Binary Search Algorithm

1. It is great to search through large sorted arrays.
2. It has a time complexity of O(log n) which is a very good time complexity. We will discuss this in detail in the Binary Search tutorial.
3. It has a simple implementation.

Binary search

This is the iterative version of the binary search:

```
FUNCTION binary_search(arr : ARRAY_OF INTEGER) : INTEGER
VAR
   left, right, mid : INTEGER;
BEGIN
   left := 0;
   right := arr.length-1;
   WHILE (left < right) DO
       mid := left + (right - left)/2;
       // check if x is present in the mid
       IF (arr[mid] = x) THEN
           RETURN mid;
       END_IF
       // if x grater, ignore the left half
       IF (arr[mid] < x) THEN
           left := mid+1;
       ELSE
       // if x is smaller, ignore the right half
       right := mid-1;
       END_IF
   END_WHILE
   // if we reached here then the element is not present
```

```
    RETURN -1   ;

END
```

Which of the following is not an application of binary search?

To find the lower/upper bound in an ordered sequence

Union of intervals

To search in unordered list

What is the time complexity of binary search with iteration?

O(nlogn)

O(logn)

O(n)

Given an array arr = {5,6,77,88,99} and key = 88; How many iterations are done until the element is found?

1

2

3

4

What you should know by now :

Recapitulation

Congratulations, we salute you for this achievement!
You have seen the most known algorithm of sorting and searching