Data Structures and Algorithms

Searching Algorithms

Prepared by:

Eng. Malek Al-Louzi

School of Computing and Informatics– Al Hussein Technical University

Spring 2021/2022

Outlines

- Introduction
- Brute Force Strategy
- Linear Search
- Binary Search

Introduction

- Algorithms is a common subject in computer science.
- It is a very core and important subject.
- In Algorithms subject, you will learn how to solve different types of problems and the strategies or approaches for solving a problem.
- First, what is an algorithm?

Introduction

- The formal common definition is:

It is a step-by-step procedure for solving a computational problem.

- So basically, you have a problem you want to solve.
- For solving the problem, you will have a set of systematic instructions.

Introduction

- There are strategies for solving problem.
- What is a strategy?

Strategy is an approach for solving a problem.

- For solving computational problem, we can adopt a specific strategy for solving the problem if the strategy is suitable for the problem.

Brute Force Strategy

- Brute Force Strategy is exactly what they sound like straightforward.
- Approach of solving a problem by trying every possibility available, no matter how much the time it will take.
- It is a straightforward technique of problem-solving, in which trying all the possible ways or all the possible solutions to find the desired solution.

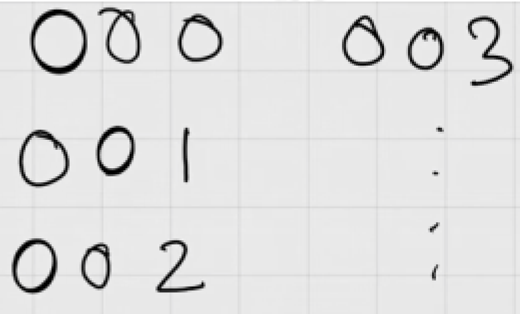
Brute Force Strategy

- Many problems solved in day-to-day life using the brute force Approach, for example:
- Suppose you have a small padlock with 3 digits, each from 0-9.

- You forgot your combination, but you don't want to buy another padlock.

Brute Force Strategy

- Since you can't remember any of the digits, you have to use a brute force method to open the lock.
- So, you set all the numbers back to 0 and try them one by one: 001, 002, 003, and so on until it opens.



- In the worst-case scenario, it would take 1000 tries to find your combination.

Linear Search

- **Let's define the problem:**
- We are given a sorted array of integers.
- A sorted array means that the elements in the array are arranged either in increasing order or decreasing order.
- We are given an integer **x**; we want to find out whether **x** exists in the array or not.

A

A	2	6	13	21	36	47	63	81	97
	0	1	2	3	4	5	6	7	8

Linear Search

- If **x** exists in the array, then we want to find out the position at which **x** exists in the array as shown below.

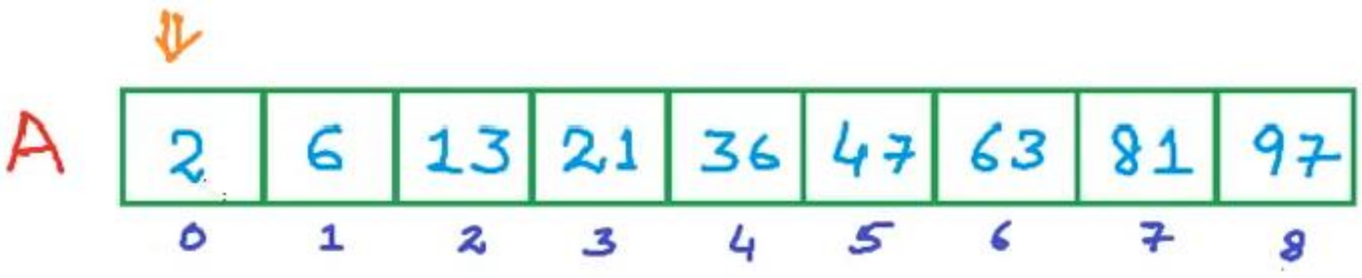
A

2	6	13	21	36	47	63	81	97
0	1	2	3	4	5	6	7	8

x	Exists?
81	yes (7)
25	NO
21	yes (3)

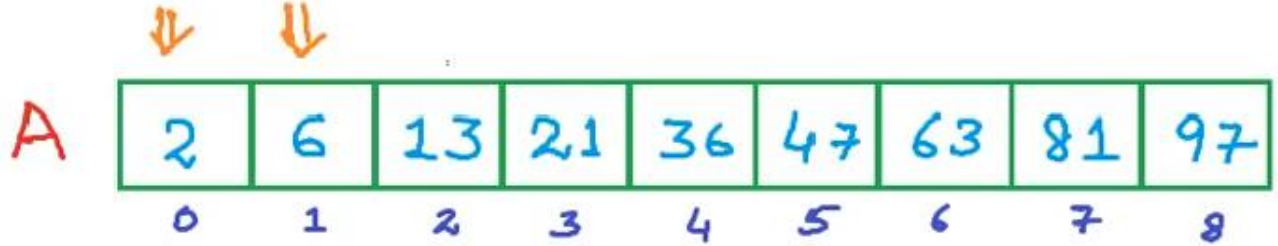
Linear Search

- What would be the logic to find out whether **x** exists in the array or not?
- One simple approach that we can scan the whole array to find out the desired number.
- So, we start at index 0 and compare this element with **x**.
- If it is equal to **x** then we are done with our search, we have found the element in the array.

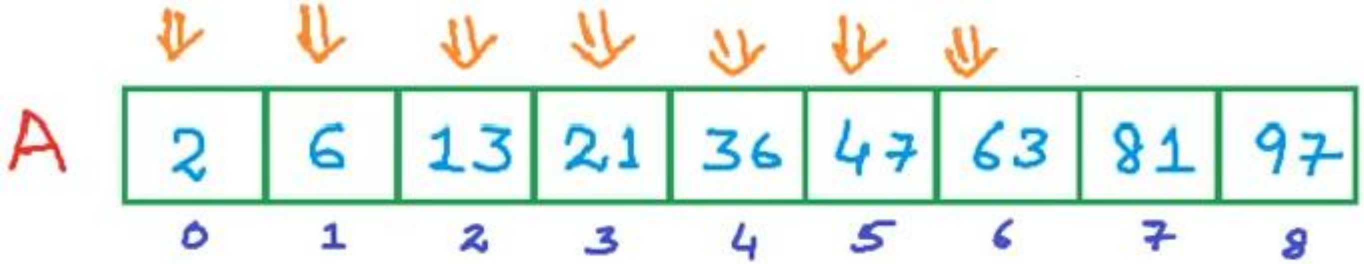


Linear Search

- If not, we go to the next element and compare again.

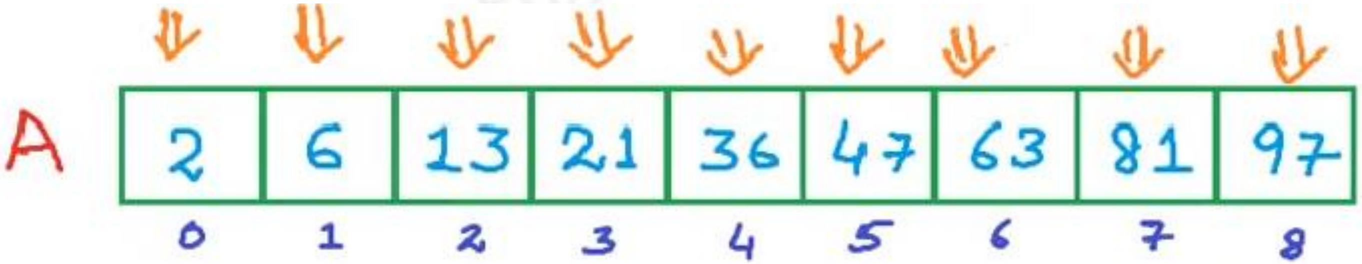


- We keep on comparing with the next element until either we find the number, or we reached the end of the array.
- For example, if we wanted to find 63 in array **A**, we start at index 0 and our search will be over at index 6.



Linear Search

- If we wanted to find out 25, our search will be over at index 8 with the conclusion that 25 doesn't exist in the array.



- This approach will work irrespective of whether the array is sorted or not.

Linear Search

- Let's write the code for this approach.
- We want to write a method called **Search** that takes an array **A**, it's size **n**, and the number **x** to be searched for.

```
Search(A, n, x)
{
    for i ← 0 to n-1
        if A[i] == x
            return i
    return -1
}
```

Linear Search

- Now, with this algorithm, if we are lucky, we will find **x** at the first position.
- So, in the best case we will make only one comparison and we will be able to find the result.

Best case:
1 Comparison

Linear Search

- In the worst case, when **x** would not even be present in the array:
  - We will scan the whole array and we will make **n** comparisons with all the elements in the array.
  - Then we will be able to giveback a result that **x** doesn't exist in the array.

Worst case:
n Comparisons

  - So, the time taken in the worst case is proportional to the size of the array.
  - In other words, the time complexity is $O(n)$

Linear Search

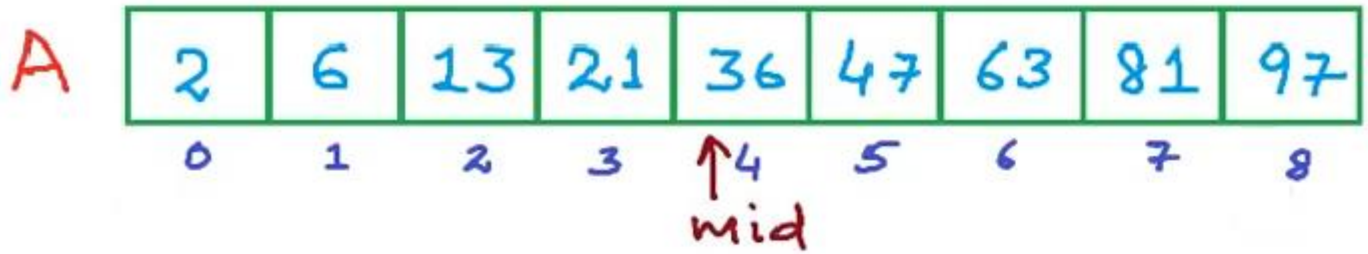
- We call this search **Linear search**.
- In linear search, we are not using any property like the array sorted or not.
- Linear search is following a brute force approach, because we are exhausting all the possibilities to find a number in the array.

Binary Search

- Now, lets try to improve this algorithm using the extra property of the array that it is sorted.
- Let's say we want to find out whether number 13 exists in the array A, so **x** is 13.
- We will use different approach this time.
- Instead of comparing **x** with the first element (as we do in linear search), we will compare it with the middle element in the array.

Binary Search

- The size of this array is 9, so the middle element will be at index 4.



- Now, there can be three cases here:

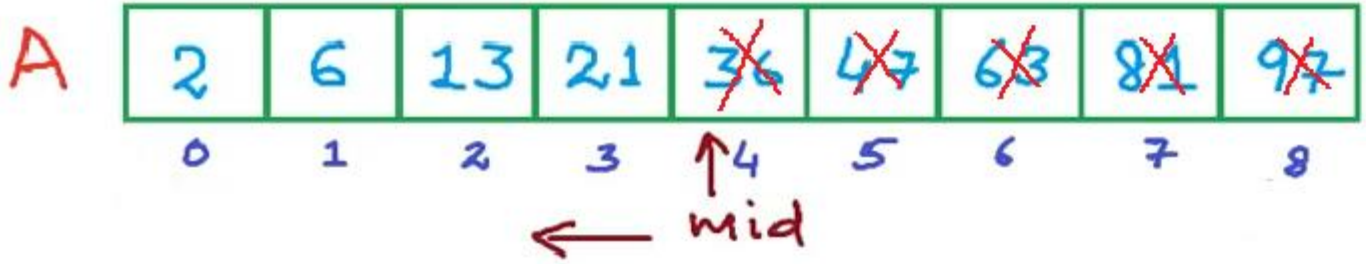
Case 1 : $x == A[mid]$

Case 2 : $x < A[mid]$

Case 3 : $x > A[mid]$

Binary Search

- Clearly if **x** is equal to the middle element, our search is over because we have found **x** in the array.
- If **x** is less than the middle element, then because the array is sorted, it lies before the middle element.
- Also, we can discard the middle element and all the elements after middle element.



Binary Search

- So, in case 2 and case 3 we reduce our search space and discard half the elements from our search space.

Case 1 : $x == A[mid]$

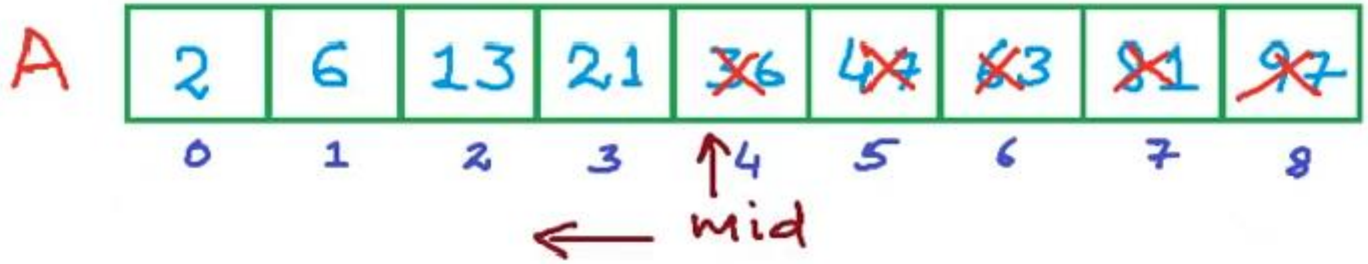
✓ Case 2 : $x < A[mid]$

✓ Case 3 : $x > A[mid]$

- In this example when **x** is 13, initially our search space is the whole array, **x** can exist any where in the array.

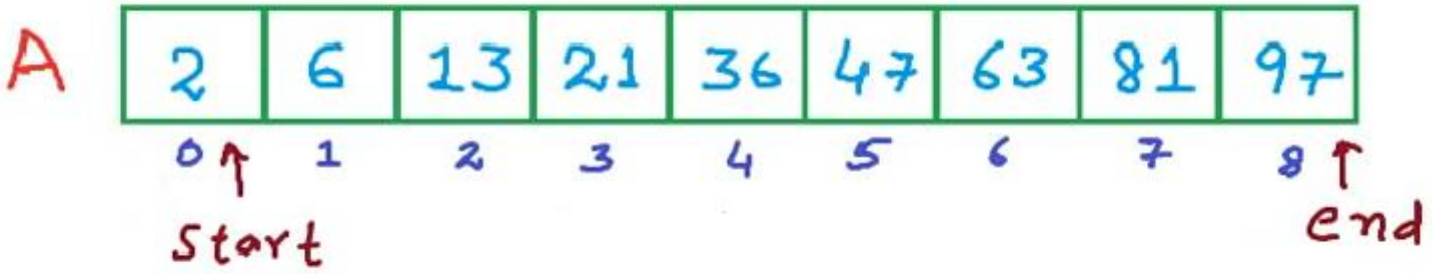
Binary Search

- In this example when **x** is 13, initially our search space is the whole array, **x** can exist any where in the array.
- We compare it with the middle element which is 36.
- Now, **x** is less than 36, so it should exist somewhere before 36.
- So, we discard all the elements after 36 and 36 as well.



Binary Search

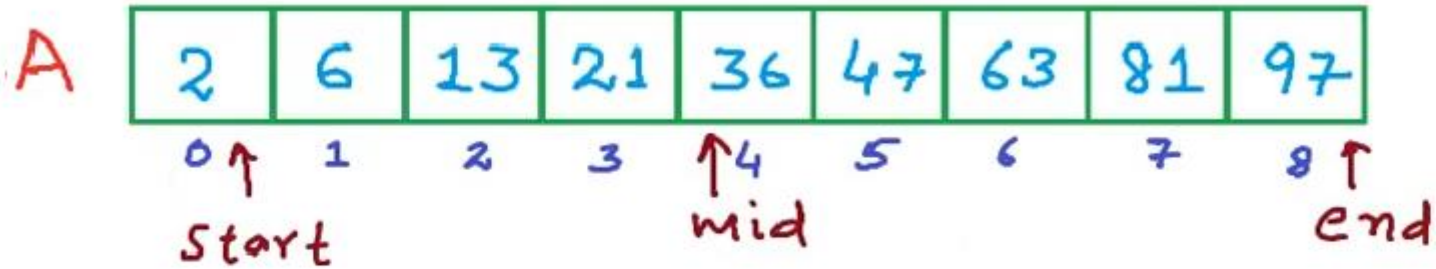
- Now, the problem gets redefined, we need to search **x** only between index 0 and index 3.
- How do we keep track of the search space?
- We keep track of the search space using two indices, **start** and **end**.
- Initially, the **start** would be 0 and **end** would the last index of the array, in this case **end** is 8, because initially the whole array is our search space.



Binary Search

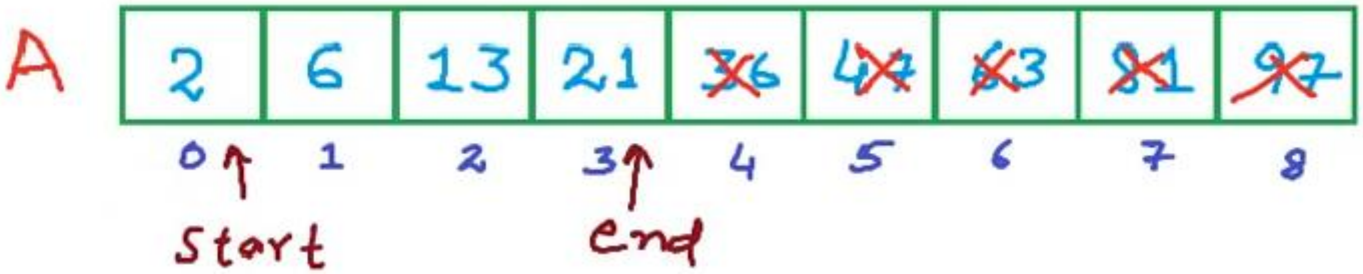
- We calculate **mid** as:

$$mid = \left\lfloor \frac{(start + end)}{2} \right\rfloor$$



Binary Search

- Now, once we find out our reduced search space, we adjust **start** and **end** accordingly.
- In our case, after comparing 13 with 36, and discarding half of the array, our **end** now becomes index 3, which is less than **mid** element by 1.



Binary Search

- Now, again we will find out the **mid** element in this reduced search space.

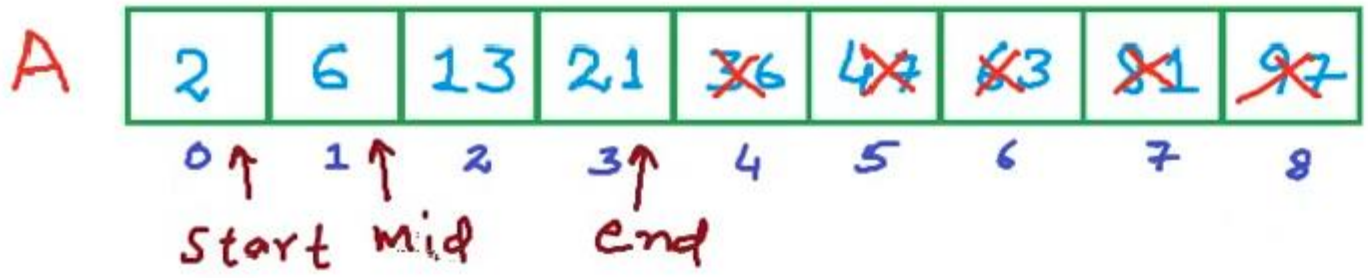
$$mid = \left\lfloor \frac{0+3}{2} \right\rfloor$$

$$= \lfloor 1.5 \rfloor$$

$$= 1$$

Binary Search

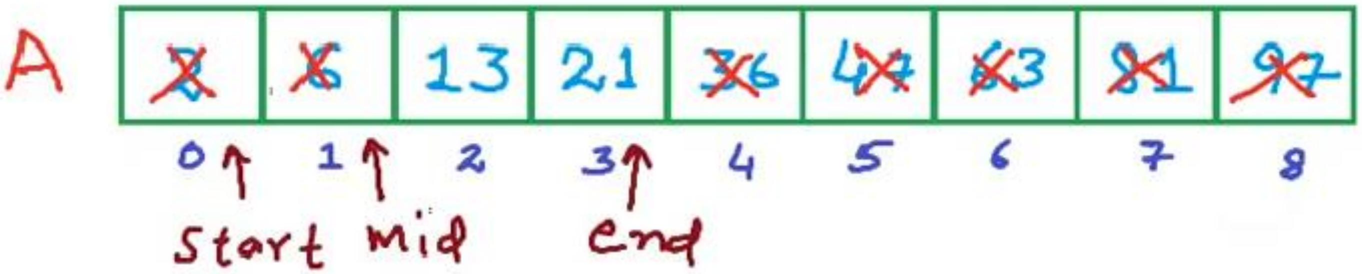
- So, **mid** is index 1.



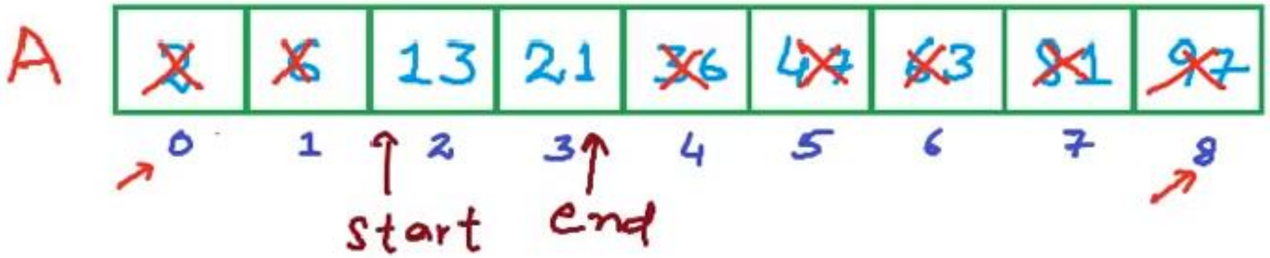
- Once again, is the **mid** element equal to **x**?
- No, 6 is not equal to 13.
- Is **x** less than the middle element, is it case 2?
- No, it is not.

Binary Search

- **x** is greater than the middle element.
- So, this time we discard the middle element and all the elements towards it's left.

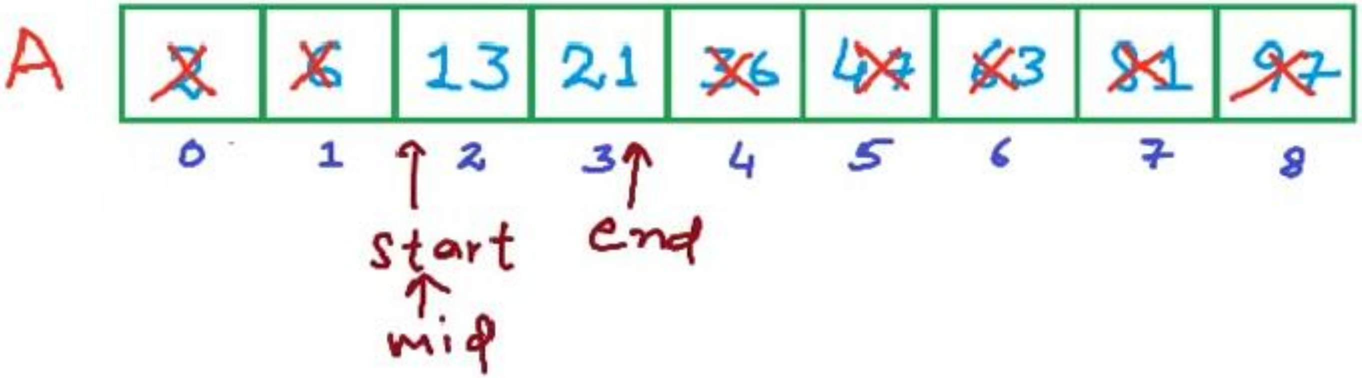


- Also, we shift **start** to mark our new search space.



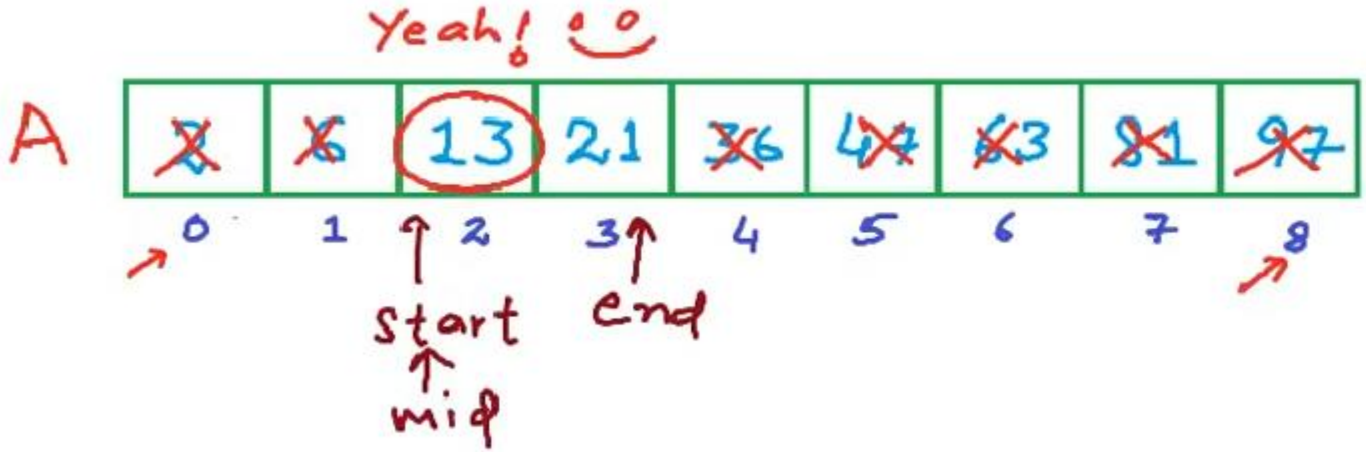
Binary Search

- Now, the new search space is starting at index 2 and ending at index 3.
- What is the middle element? **mid** = 2.



Binary Search

- Now, **x** is equal to the middle element, we have found our element.
- So, we are done with our search.



Binary Search

- This kind of search, where we reduce the search space into half at each comparison is called **Binary Search**.
- Binary search is one of the most famous and fundamentals algorithms in Computer Science.
- We can reduce the search space into half **only** because the array is sorted.
- Array being sorted is a **precondition** for binary search.

Binary Search

- Let's now write the pseudocode for this algorithm.
- We will write a method called **BinarySearch** that will take as argument an array **A**, it’s size **n**, and a number **x** to searched for in the array.

Binary Search

```
BinarySearch(A, n, x)
{  start ← 0
   end ← n-1
   while(start <= end)
   {
     mid ← (start+end)/2
     if A[mid] == x
         return mid
     else if x < A[mid]
     {
       end ← mid-1
     }
     else
     {
       start ← mid+1
     }
   }
   return -1
}
```

Binary Search

- Why the while statement with a condition (**start** <= **end**)?

while (start <= end)

- What we are doing basically in binary search is reducing our search space repeatedly, by adjusting the **start** and **end** values.
- There must be an exit condition for this repetition, the exit condition can be:
  - Either we find the element in the array, so we return and exit.

    if A[mid] == x
    return mid

  - Or we exhaust the whole search space.

Binary Search

- In binary search, in the best case, we can find the element x in one comparison.
- When the first middle element itself will be the element x.

Best case:
1 comparison

Binary Search

- In the worst case, we will keep reducing the search space till the search space becomes one element.
- So, from n we reduce to n/2, and from n/2 we reduce to n/4, and we go on till our search space becomes one.

$$\begin{array}{c} n \\ \downarrow \\ n/2 \\ \downarrow \\ n/4 \\ \downarrow \\ \vdots \\ 1 \end{array}$$

Binary Search

- How many steps does it take?
- Let's say, it takes **k** steps to reduce n to 1.

$$\begin{array}{c} n \\ \downarrow \\ n/2 \\ \downarrow \\ n/4 \\ \downarrow \\ \vdots \\ 1 \end{array} \quad k$$

Binary Search

- So:

$$\frac{n}{2^k} = 1$$

- If we solve this equation, then:

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$$

- So, in the worst case, binary search will take:

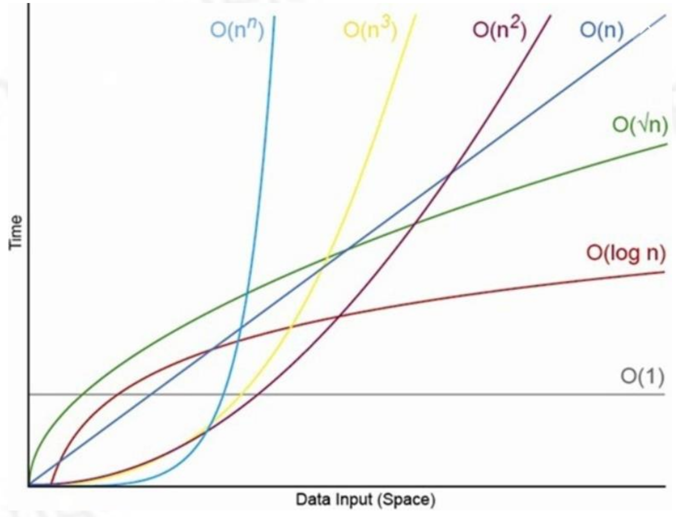
Worst case:
$\log n$ comparisons

Binary Search

- So, the time complexity of binary search is:

  $O(\log n)$

- $O(\log n)$ is a lot more efficient than $O(n)$



Any Questions???...