Data Structures and Algorithms
CSE 2101

Class time:

Wednesday and Thursday – 11.10am

Google classroom code: mpsx2hl

Course Teacher:

Dr. Md. Mosaddek Khan

Associate Professor and Student Advisor

Department of Computer Science and Engineering

University of Dhaka

Email: mosaddek@du.ac.bd

Lectures	Topics
1	Complexity analysis
2	**Searching** • Linear search, binary Search, application of Binary Search- finding element in a sorted array finding nth root of a real number, solving equations.
3-6	**Recursion** • Basic idea of recursion (3 laws-base case, call itself, move towards base case by state change) • tracing output of a recursive function • applications- merge sort, permutation, combination. Memoization
	Sorting • Insertion sort, selection sort, bubble sort, • merge sort, quick sort (randomized quick sort) • distribution sort (counting sort, radix sort, bucket sort) lower bounds for sorting, external sort
7	**Linked List** • Singly/doubly/circular linked lists, • basic operations on linked list (insertion, deletion, and traverse), • dynamic array and its application.
8-9	**Stack Basic** • stack operations (push/pop/peek), • stack-class implementation using Array and linked list, • in-fix to post-fix expressions conversion and evaluation, • balancing parentheses using stack,
10-11	**Queue** • basic queue operations • (Enqueue, dequeue), circular queue/ dequeue, • queue-class implementation using array and linked list • application- Josephus problem, palindrome checker using stack and queue.
12	**Binary tree** • Binary tree representation using array and Pointer • traversal of Binary Tree (in-order, pre-order and post-order).
13-14	**Binary Search Tree** • BST representation • basic operations on BST (creation, insertion, deletion, querying and traversing), • application- searching, sets.
15-16	**Heap** • Min-heap, max-heap, • Fibonacci-heap • applications-priority queue • heap sort.
17	**General Tree** Implementation, application of general tree- file system
18	**Disjoint Set** Union finds, path compression.
20	**Huffman Coding: application- Compression.**
21 -22	**Graph representation (adjacency matrix/adjacency list), basic operations on graph (node/edge insertion and deletion), traversing a graph: breadth-first search (BFS), depth-first search (DFS), graph-bi-colouring.**
23	**Self-balancing Binary Search Tree:** AVL tree (Rotation, insertion).
24-25	**Set Operations: Set representation using** **bitmask, set/clear bit, querying the status of a bit, toggling bit values, LSB, application of set operations.**
26-27	**String ADT: The** **concatenation of two strings, the extraction of substrings, searching a string for a matching substring, parsing.**

Data Structures ?

- A data structure is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently.
- Different types of data structures are suited to different kinds of applications:
- One of the first recorded uses of a data structure was the Jacquard loom in 1801, which used a punched card to control the pattern of a woven textile.
    - some are highly specialized to specific tasks
    - Some common examples of data structures include
        - Arrays
        - Linked lists Stacks
        - Queues
        - Trees
        - graphs.

Algorithms?

- Algorithms are a set of instructions for carrying out a specific task or solving a specific problem
- A key part of an algorithm is the use of one or more data structures in order to store and organize the data that the algorithm operates on.

Relationship –

- data structures provide a way to store and organize data, and
- algorithms use the data structures to accomplish a specific task or solve a specific problem.

Searching Algorithms: Linear and Binary Search

Linear Search

- Sequential search
  - It traverses the array sequentially to locate the required element.
  - It searches for an element by comparing it with each element of the array one by one.

- Applicability
  - No information is given about the array.
  - The given array is unsorted or the elements are unordered.
  - The list of data items is smaller.

Linear Search

Best Case

- The element being searched may be found at the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in best case, linear search algorithm takes O(1) operations.

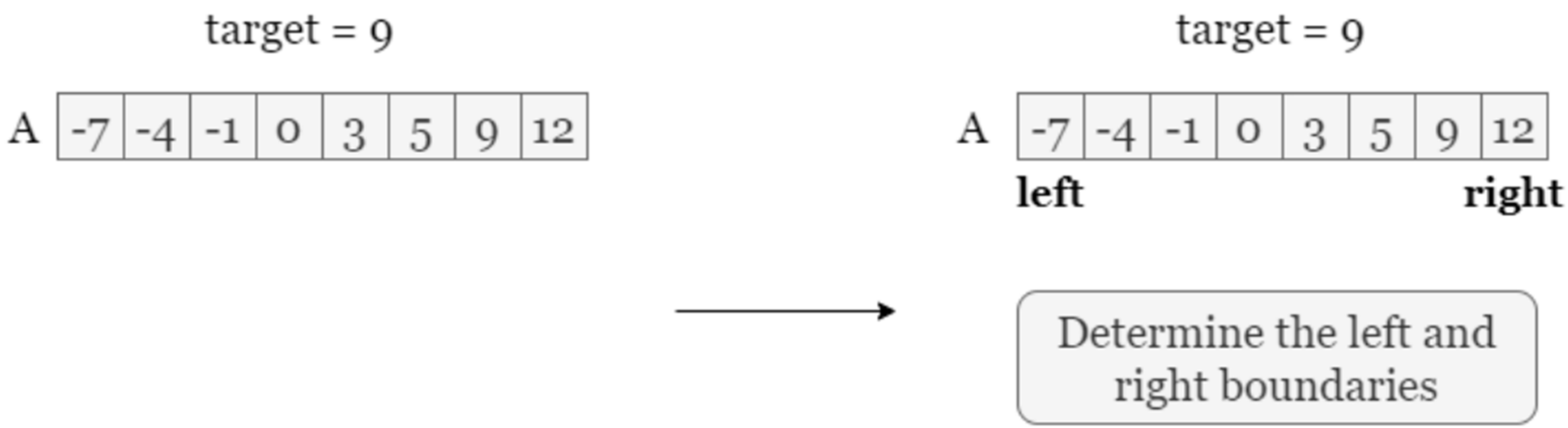
Worst Case

- The element being searched may be present at the last position or not present in the array at all.
- In the former case, the search terminates in success with n comparisons.
- In the later case, the search terminates in failure with n comparisons.
- Thus in worst case, linear search algorithm takes O(n) operations.

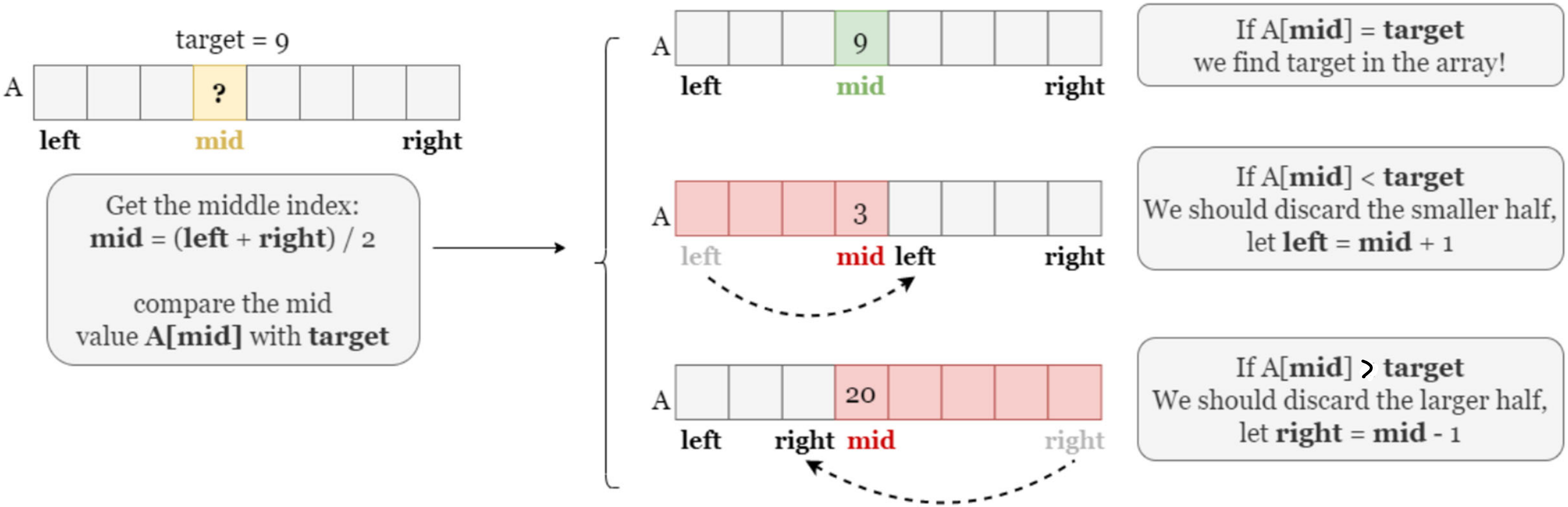
Binary Search

First, we define the search space using two boundary indexes, `left` and `right`

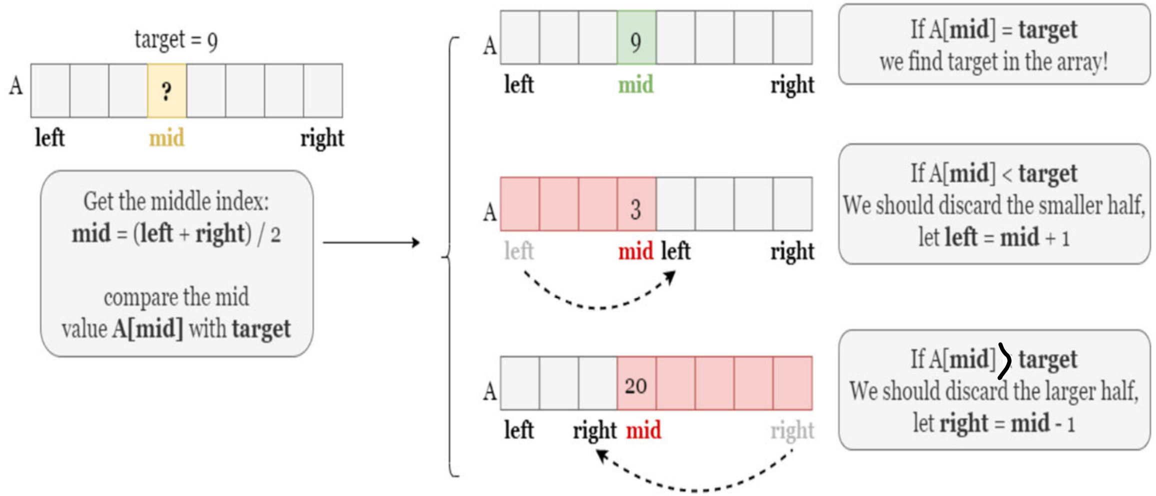
- We shall continue searching over the search space as long as it is not empty.
- A while loop with a condition: `left <= right`



Binary Search



Binary Search



Algorithm

1. Initialize the boundaries of the search space as `left = 0` and `right = nums.size - 1`.
2. If there are elements in the range `[left, right]`, we find the middle index `mid = (left + right) / 2` and compare the middle value `nums[mid]` with `target`:
   - If `nums[mid] = target`, return `mid`.
   - If `nums[mid] < target`, let `left = mid + 1` and repeat step 2.
   - If `nums[mid] > target`, let `right = mid - 1` and repeat step 2.
3. We finish the loop without finding `target`, return `-1`.

Binary Search

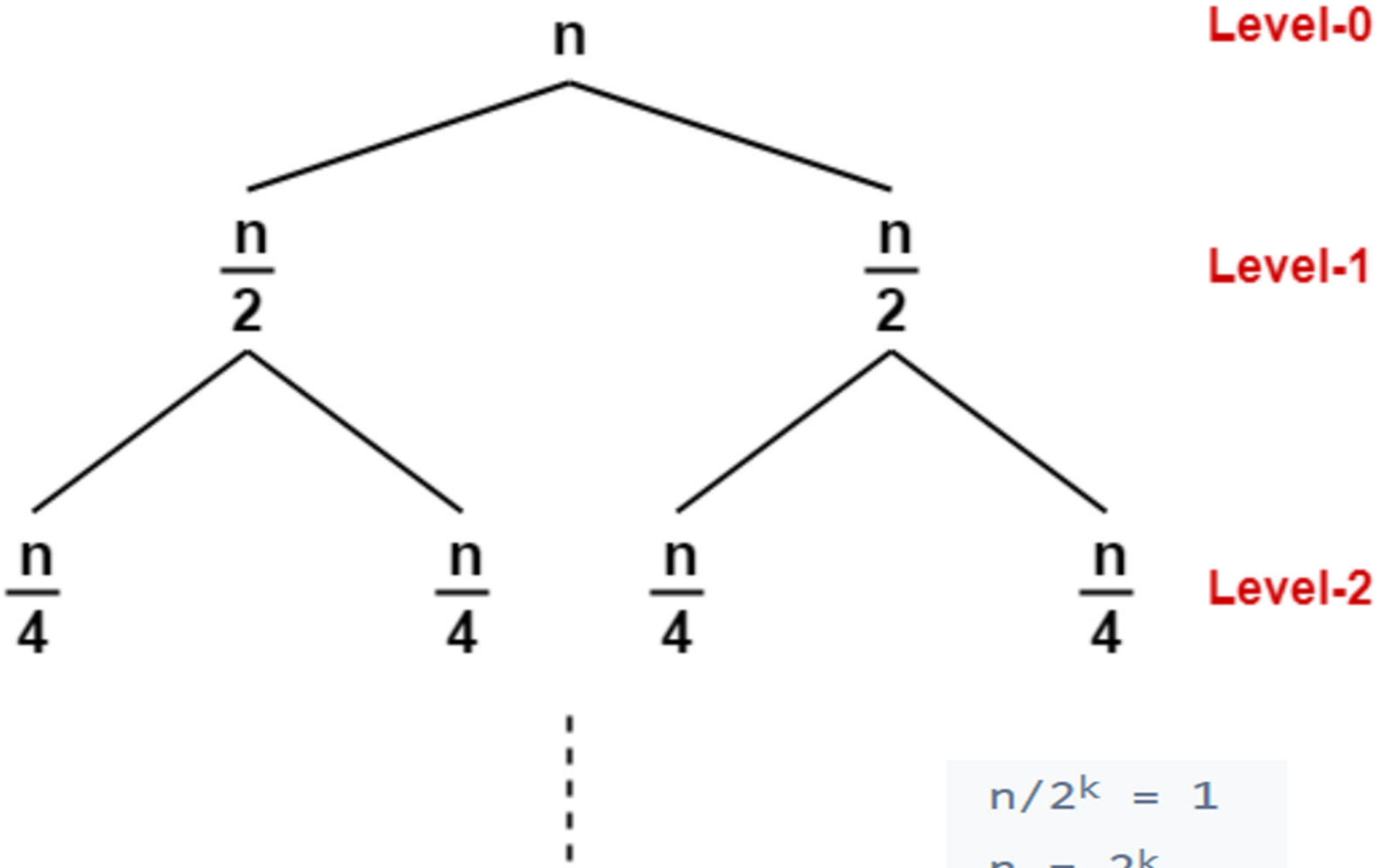
```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # Set the left and right boundaries
        left = 0
        right = len(nums) - 1

        # Under this condition
        while left <= right:
            # Get the middle index and the middle value.
            mid = (left + right) // 2

            # Case 1, return the middle index.
            if nums[mid] == target:
                return mid
            # Case 2, discard the smaller half.
            elif nums[mid] < target:
                left = mid + 1
            # Case 3, discard the larger half.
            else:
                right = mid - 1

        # If we finish the search without finding target, return -1.
        return -1
```

Binary Search



$n \rightarrow n/2 \rightarrow n/4 \rightarrow \dots \rightarrow 1$

$n/2^k = 1$

$n = 2^k$

$k = \log_2 n$

Assuming our search space is exhausted after k level

Binary Search

Complexity Analysis

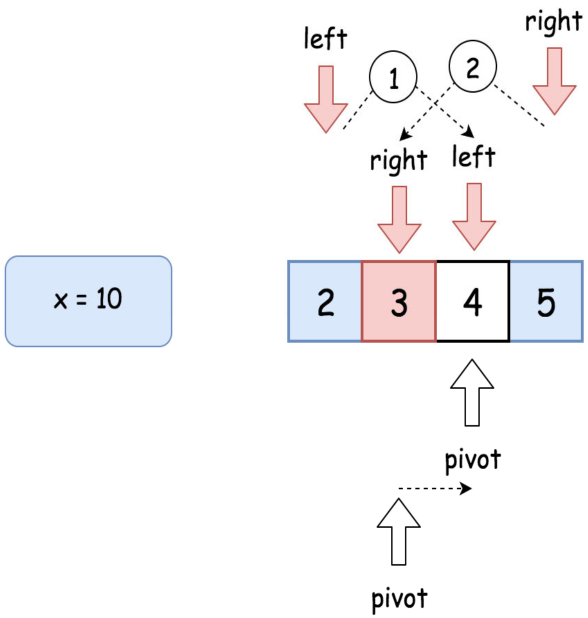
Let n be the size of the input array `nums`.

- Time complexity: $O(\log n)$
  - `nums` is divided into half each time. In the worst-case scenario, we need to cut `nums` until the range has no element, and it takes logarithmic time to reach this break condition.
- Space complexity: $O(1)$
  - During the loop, we only need to record three indexes, `left`, `right`, and `mid`, they take constant space.

Binary Search – finding the square root of a real number x

Let's go back to the interview context. For $x \geq 2$ the square root is always smaller than $x/2$ and larger than 0 : $0 < a < x/2$.

Since a is an integer, the problem goes down to the iteration over the sorted set of integer numbers. Here the binary search enters the scene.



- If x < 2, return x.
- Set the left boundary to 2, and the right boundary to x / 2.
- While left <= right:
  - Take num = (left + right) / 2 as a guess. Compute num * num and compare it with x:
    - If num * num > x, move the right boundary right = pivot -1
    - Else, if num * num < x, move the left boundary left = pivot + 1
    - Otherwise num * num == x, the integer square root is here, let's return it
- Return right

Binary Search – finding the square root of a real number x

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # Set the left and right boundaries
        left = 0
        right = len(nums) - 1

        # Under this condition
        while left <= right:
            # Get the middle index and the middle value.
            mid = (left + right) // 2

            # Case 1, return the middle index.
            if nums[mid] == target:
                return mid
            # Case 2, discard the smaller half.
            elif nums[mid] < target:
                left = mid + 1
            # Case 3, discard the larger half.
            else:
                right = mid - 1

        # If we finish the search without finding target, return -1.
        return -1
```

```python
class Solution:
    def mySqrt(self, x):
        if x < 2:
            return x

        left, right = 2, x // 2

        while left <= right:
            pivot = left + (right - left) // 2
            print("This is the pivot value:", pivot)
            num = pivot * pivot
            if num > x:
                right = pivot -1
            elif num < x:
                left = pivot + 1
            else:
                return pivot

        return right

sol = Solution()
print(sol.mySqrt(23))
```

```
This is the pivot value: 6
This is the pivot value: 3
This is the pivot value: 4
This is the pivot value: 5
4
```

Binary Search – Recursive Implementation

```python
if __name__ == '__main__':

    nums = [2, 5, 6, 8, 9, 10]
    target = 5

    (left, right) = (0, len(nums) - 1)
    index = binarySearch(nums, left, right, target)

    if index != -1:
        print('Element found at index', index)
    else:
        print('Element found not in the list')
```

```python
# Recursive implementation of the binary search algorithm to return
# the position of `target` in subarray nums[left…right]
def binarySearch(nums, left, right, target):

    # Base condition (search space is exhausted)
    if left > right:
        return -1

    # find the mid-value in the search space and
    # compares it with the target

    mid = (left + right) // 2

    # overflow can happen. Use below
    # mid = left + (right - left) / 2

    # Base condition (a target is found)
    if target == nums[mid]:
        return mid

    # discard all elements in the right search space,
    # including the middle element
    elif target < nums[mid]:
        return binarySearch(nums, left, mid - 1, target)

    # discard all elements in the left search space,
    # including the middle element
    else:
        return binarySearch(nums, mid + 1, right, target)
```