RIPHAH
INTERNATIONAL UNIVERSITY

NAME	**NASIR HUSSAIN**
SAP ID	**43913**
SUBMITTED TO	**Muhammad Usman Sharif**
ASSIGNMNET	**04**
DEPARTMENT	**COMPUTING CYB-05**
DATED	**10-NOV-2024**
COURSE	**Analysis of Algorithms**

GITHUB: https://github.com/leo1548/Hybrid-ALgo

Task:

Consider two algorithms, Algorithm A and Algorithm B, designed to solve the same problem. Your task is to design a hybrid algorithm that combines the best aspects of both Algorithm A and Algorithm B to achieve improved performance.

Searching for an element in a sorted array.

Algorithms Combined:

- Algorithm A: Binary Search
- Algorithm B: Jump Search

CODE:

```
import math


def hybrid_search(arr, target):
    n = len(arr)
    jump = int(math.sqrt(n))  # Optimal jump size (sqrt of array length)
    prev = 0  # Starting index of the current block

    # Jump Search Phase
    while prev < n and arr[min(jump, n) - 1] < target:
        prev = jump
        jump += int(math.sqrt(n))
        if prev >= n:
            return -1  # Target not found, we are past the array bounds

    # Binary Search Phase within the identified block
    left = prev
    right = min(jump, n) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Target not found


# Example array to test the hybrid search
test_array = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91, 105, 150, 190, 250]
targets = [23, 72, 105, 190, 300]  # The last target (300) is not in the array

# Testing the hybrid search with different targets
for target in targets:
    result = hybrid_search(test_array, target)
    if result != -1:
        print(f"Element {target} found at index {result}.")
    else:
        print(f"Element {target} not found in the array.")
```

Working:

```
C:\Users\Admin\PycharmProjects\pythonProject1\.venv\Scripts\python.exe C:\Users\Admin\PycharmProjects\pythonProject1\main.py
Element 23 found at index 5.
Element 72 found at index 8.
Element 105 found at index 10.
Element 190 found at index 12.
Element 300 not found in the array.

Process finished with exit code 0
```

To understand hybrid algorithm working we should first understand working of Binary and Jump search.

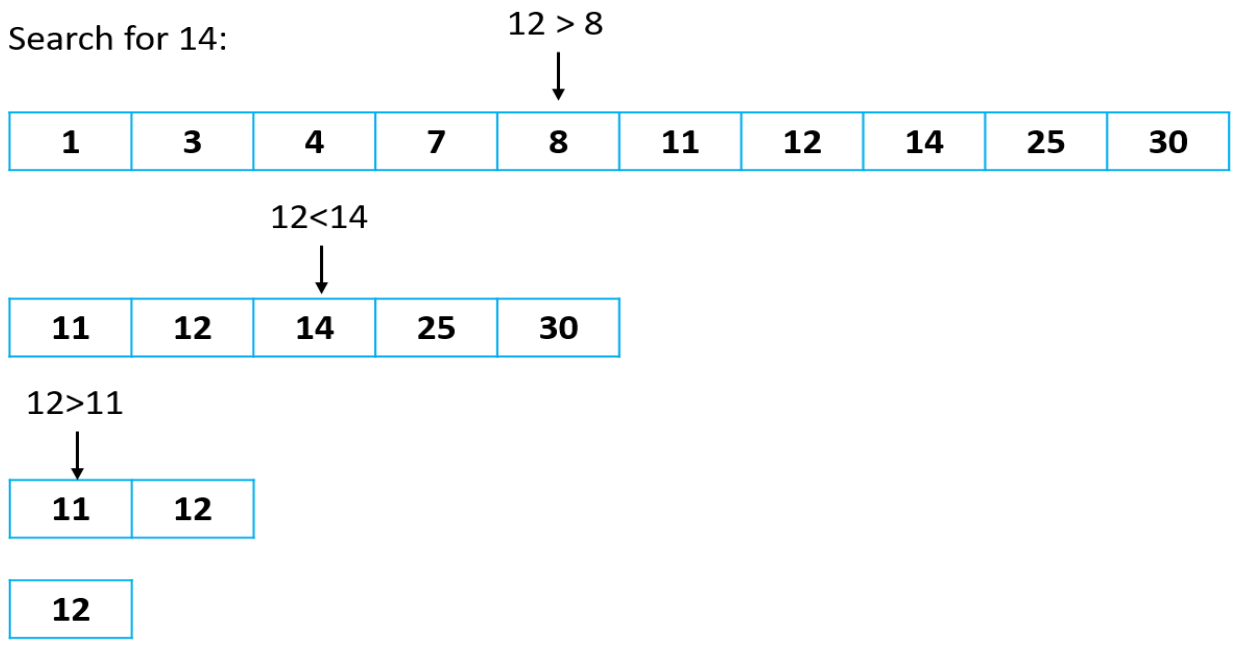
Binary Search:

How it works: Binary Search is an efficient algorithm for finding an element in a sorted array. It works by repeatedly dividing the search interval in half.

Start with the middle element of the array:

- If it matches the target, return the index.
- If the target is smaller, repeat the process on the left subarray.
- If the target is larger, repeat the process on the right subarray.

- **Time Complexity**: O (log n)
- **Space Complexity**: O (1)

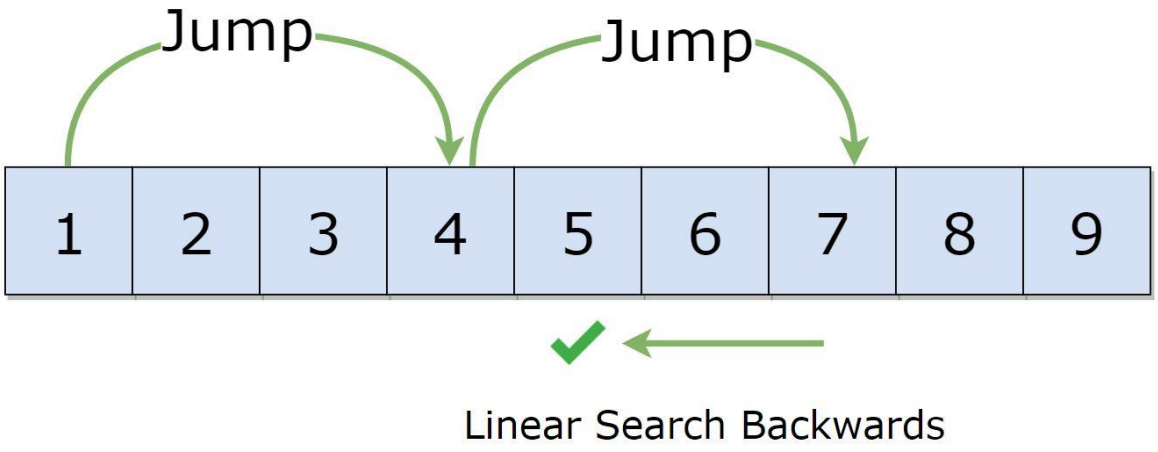


Jump Search:

How it works: Jump Search is designed for sorted arrays, where you "jump" ahead by a fixed number of elements (typically the square root of the array size, n\sqrt {n} n) instead of checking each element sequentially.

Once a jump goes past the target element, a linear search is performed in the previous block.

- **Time Complexity**: O(√n)
- **Space Complexity:** O(1)
- **When it's useful:** Jump Search is beneficial for searching in large datasets stored on media where sequential access is faster than random access (like tapes or SSDs).
- **Need of algorithm:**
- Binary Search is very efficient with O (log n) complexity but can be suboptimal for very large datasets stored in slower, non-RAM storage, where repeated midpoint accesses may introduce latency.
- Jump Search can quickly skip through large portions of the dataset but is inefficient when the target is located near the middle or end of the array due to its O (√n) complexity.
- Hybrid Search



NEED FOR ALGORITHMS:

- Binary
- Jump
- Hybrid

- ## Binary Search:

- **Efficiency**: It's highly efficient with a time complexity of O (log⁡n)O(\log n)O(logn), making it one of the best options for sorted arrays.
- **Limitation**: However, for very large datasets stored on non-RAM storage (like HDDs or even SSDs), Binary Search can be suboptimal. Accessing the midpoint repeatedly can introduce latency, as each access incurs a potential delay due to the nature of storage access times.

- ## Jump Search:

- **Fast Skipping**: Jump Search can skip through large portions of the dataset in a single step, ideal for cases where random access may be slower.
- **Limitation**: Despite being useful for large datasets, Jump Search is inefficient when the target element is near the middle or end of the array due to its O (n) O (\sqrt{n})O(n) complexity.

- ## Hybrid Search:

- **Combined Benefits**: The hybrid search leverages the fast-skipping capability of Jump Search to quickly narrow down the search range and then applies the precision of Binary Search within the identified block.
- **Optimized Process**: By combining both algorithms, Hybrid Search reduces the total number of comparisons and improves performance for large, sorted datasets, especially when random access speed is a concern.

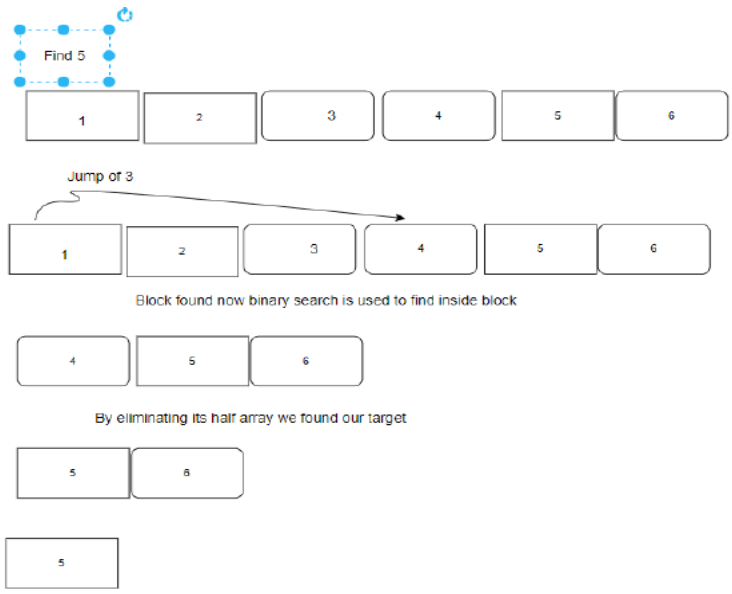
Hybrid algo Working:

Jump Search Phase:

- Use Jump Search to quickly narrow down the search interval.
- The jump size is chosen as n\sqrt {n} n because it balances the cost of jumping and the cost of the subsequent linear search.
- This phase identifies the block (of size n\sqrt {n} n) where the target element might be located.

Binary Search Phase:

- Once the correct block is identified, switch to Binary Search within that smaller block.
- This phase ensures fast convergence to the target element within the reduced search space.



Performance Analysis
Time Complexity: Jump Phase: O(√n)

Binary Search Phase: O(log√n)

Overall: O(√n + log √n) which simplifies to O(√n)

Space Complexity: O (1)Osince the algorithm uses a fixed amount of extra space.

Improved things:

- For large datasets, the initial Jump Search quickly reduces the search space, allowing Binary Search to efficiently finish within a smaller segment.
- This can be particularly useful for data stored in slower, sequential-access storage systems where Binary Search alone might incur a high overhead due to non-sequential access.

CODE:

```python
import math
import time


# Binary Search Implementation
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1


# Jump Search Implementation
def jump_search(arr, target):
    n = len(arr)
    jump = int(math.sqrt(n))
    prev = 0
    # Finding the block where the element may be present
    while prev < n and arr[min(jump, n) - 1] < target:
        prev = jump
        jump += int(math.sqrt(n))
        if prev >= n:
            return -1
    # Linear search within the identified block
    for i in range(prev, min(jump, n)):
        if arr[i] == target:
            return i
    return -1


# Hybrid Search Implementation
def hybrid_search(arr, target):
    n = len(arr)
    jump = int(math.sqrt(n))
    prev = 0
    # Step 1: Jump Search Phase
    while prev < n and arr[min(jump, n) - 1] < target:
        prev = jump
        jump += int(math.sqrt(n))
        if prev >= n:
            return -1
    # Step 2: Binary Search Phase within the identified block
    left = prev
    right = min(jump, n) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
```

```
        else:
            right = mid - 1
    return -1


# Function to measure the execution time of different search algorithms
def measure_search_time(search_func, arr, target):
    start_time = time.time()
    result = search_func(arr, target)
    end_time = time.time()
    elapsed_time = end_time - start_time
    return result, elapsed_time


# Example Array
test_array = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91, 105, 150, 190, 250, 300, 350, 400, 450, 500]
targets = [23]

# Testing and Timing the Searches
for target in targets:
    print(f"\nSearching for {target}:")

    # Measure Binary Search Time
    result, time_taken = measure_search_time(binary_search, test_array, target)
    if result != -1:
        print(f"Binary Search: Found at index {result} in {time_taken:.6f} seconds")
    else:
        print(f"Binary Search: Not found in {time_taken:.6f} seconds")

    # Measure Jump Search Time
    result, time_taken = measure_search_time(jump_search, test_array, target)
    if result != -1:
        print(f"Jump Search: Found at index {result} in {time_taken:.6f} seconds")
    else:
        print(f"Jump Search: Not found in {time_taken:.6f} seconds")

    # Measure Hybrid Search Time
    result, time_taken = measure_search_time(hybrid_search, test_array, target)
    if result != -1:
        print(f"Hybrid Search: Found at index {result} in {time_taken:.6f} seconds")
    else:
        print(f"Hybrid Search: Not found in {time_taken:.6f} seconds")
```

OUTPUT:

```
Searching for 23:
Binary Search: Found at index 5 in 0.000000 seconds
Jump Search: Found at index 5 in 0.000000 seconds
Hybrid Search: Found at index 5 in 0.000000 seconds

Process finished with exit code 0
```