ALGORITHMS AND DATA STRUCTURES
LECTURE 7 - SEARCHING

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

CONTENT

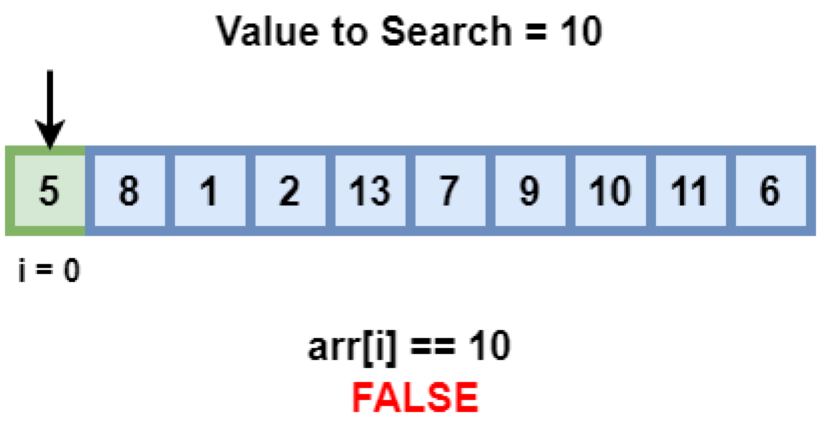
1. Linear Search
2. Binary search
3. Hashing
4. Find a pair with the given sum

LINEAR SEARCH

The *traditional* search method (Brute force)

Time complexity is O(N)

Does NOT need an array to be sorted



If we are given an array of integers A without any further information and have to decide if an element x is in A, we just have to search through it, element by element

LINEAR SEARCH

```java
// Linked list example
public boolean hasItem(T item) {
    MyNode<T> current = head;
    while (current != null) {
        if (current.data.equals(item))
            return true;

        current = current.next;
    }

    return false;
}
```

```java
// Array based example
public boolean hasItem(T item) {
    int current = 0;
    while (current < size) {
        if (array[current].equals(item))
            return true;

        current++;
    }

    return false;
}
```

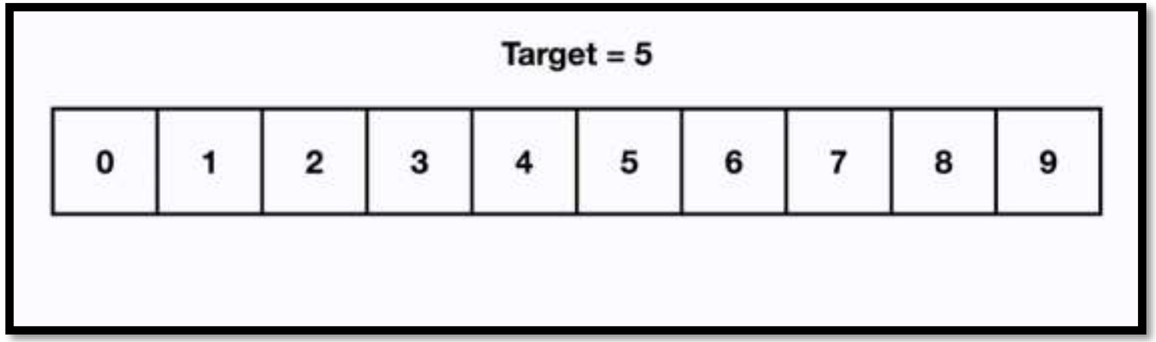
BINARY SEARCH

The *bisection* search method

Time complexity is O(logN)

List **must be sorted** to give correct answer

We start by examining the middle element of the array



If it smaller than x, then x must be in the upper half of the array (if it is there at all); if is greater than x then it must be in the lower half

*Now we continue by **restricting** our attention to either the upper or lower half, again finding the middle element and proceeding as before*

BINARY SEARCH

```java
public boolean hasItem(T item) {
    return binarySearch(item, start: 0, end: size-1);
}

// Binary Search example (Recursive)
private boolean binarySearch(T item, int start, int end) {
    if (start > end) return false;

    int mid = (start + end) / 2;

    int cmp = array[mid].compareTo(item);

    if (cmp == 0) { // if we found the item
        return true;
    } else if (cmp > 0) { // if middle element is more
        return binarySearch(item, start, end: mid - 1);
    } else {
        return binarySearch(item, start: mid + 1, end);
    }
}
```

```java
// Binary Search example (Iterative)
private boolean binarySearch(T item) {
    int start = 0, end = size - 1;

    while (start <= end) {
        int mid = (start + end) / 2;

        int cmp = array[mid].compareTo(item);

        if (cmp == 0) {
            return true;
        } else if (cmp > 0) {
            end = mid - 1;
        } else {
            start = mid + 1;
        }
    }

    return false;
}
```

BINARY SEARCH

Binary search gives better performance

Is it good to sort before we search to apply binary search instead of linear?

- only if it satisfies the inequality: $Sort + O(logN) < O(N)$
- $Sort < O(N) - O(logN)$
- no such sorting (**never true**)

Multiple Searches Case (search k times)

- $Sort + kO(logN) < kO(N) \quad \Rightarrow \quad Sort < k[O(N) - O(logN)]$
- for large k, $Sort$ time becomes irrelevant

HASHING

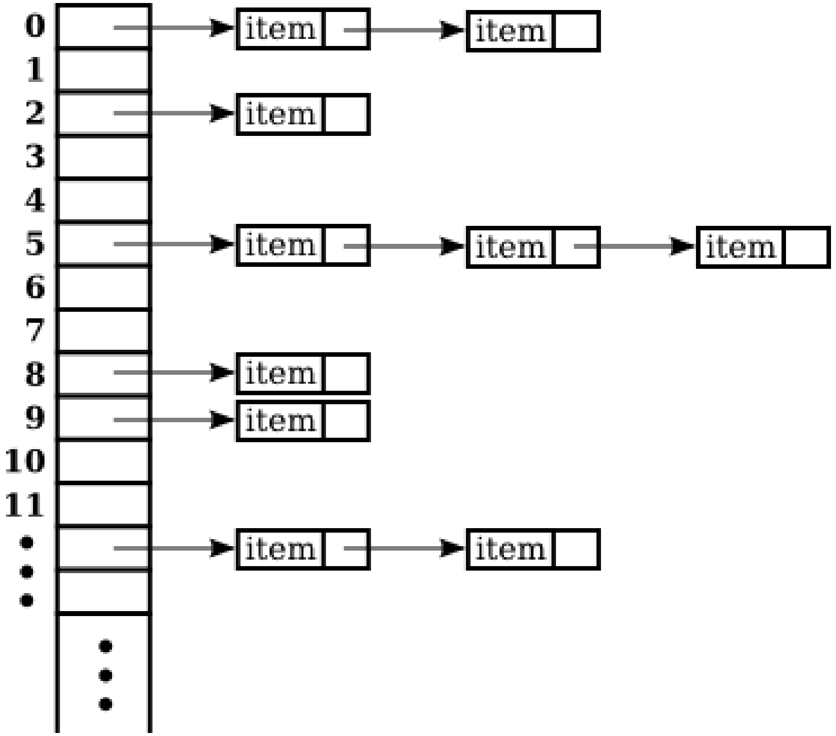
The *HashTable* is another good solution for searching

Searching Time complexity:

- **Average case: O(1)**
- Worst case: O(N)

Needs well-designed **hashing** method for making **less collisions (see Lecture 5)**

- Hash tables become quite inefficient when there are many collisions (Time complexity tends to O(N) as the number of collisions increases)



HASHING

```java
public boolean has(K key) {
    int index = hash(key);
    HashNode<K, V> temp = chainArray[index];
    while (temp != null) {
        if (temp.key.equals(key)) {
            return true;
        }
        temp = temp.next;
    }

    return false;
}

private int hash(K key) {
    return (key.hashCode() & 0x7fffffff) % M;
}
```

This can be accepted as O(1) for small number of collisions (it occurs in average case)

The number of buckets (chains)

FIND A PAIR WITH THE GIVEN SUM

Straightforward solution $O(N^2)$:

```
for (int i = 0; i < arr.length - 1; i++)
    for (int j = i + 1; j < arr.length; j++)
        if (arr[i] + arr[j] == sum)
            return true;

return false;
```

Using Binary search < O(NlogN) (only for sorted list):

```
for (int i = 0; i < arr.length; i++)
    if (binarySearch( item: sum - arr[i],  start: i + 1,  end: arr.length - 1))
        return true;

return false;
```

Using HashTable O(N) (we use HashSet<K>, since we are not dealing with key-value pairs):

```
MyHashSet<Integer> previousItems = new MyHashSet<>();
for (int i = 0; i < arr.length; i++) {
    if (previousItems.has( key: sum - arr[i])) // O(1)
        return true;

    previousItems.put(arr[i]);  // O(1)
}

return false;
```

LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 3

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 5

GOOD LUCK!