Lecture 7: Binary Search, N-ary Search

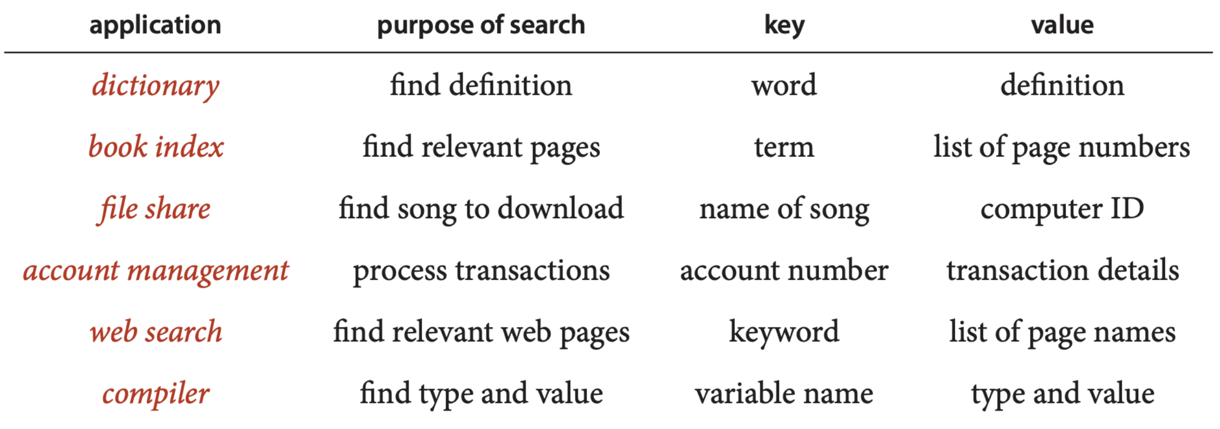
Date: 09/08/2023

Searching

Modern computing and the internet have made accessible a vast amount of information. The ability to efficiently search through this information is fundamental to processing it.

⇒ We use the term **symbol table** to describe an abstract mechanism where we save information (a value) that we can later search for and retrieve by specifying a key. The nature of the keys and the values depends upon the application.

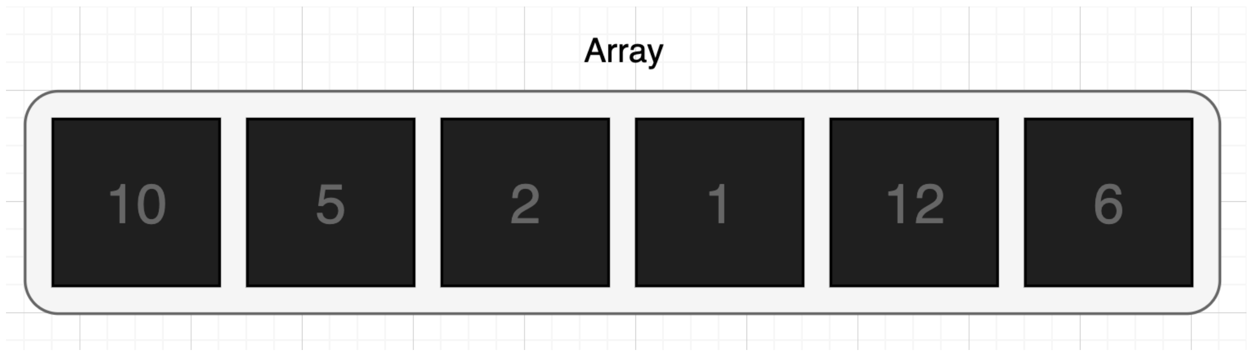
Reference: Algorithms 4e. Page 375.



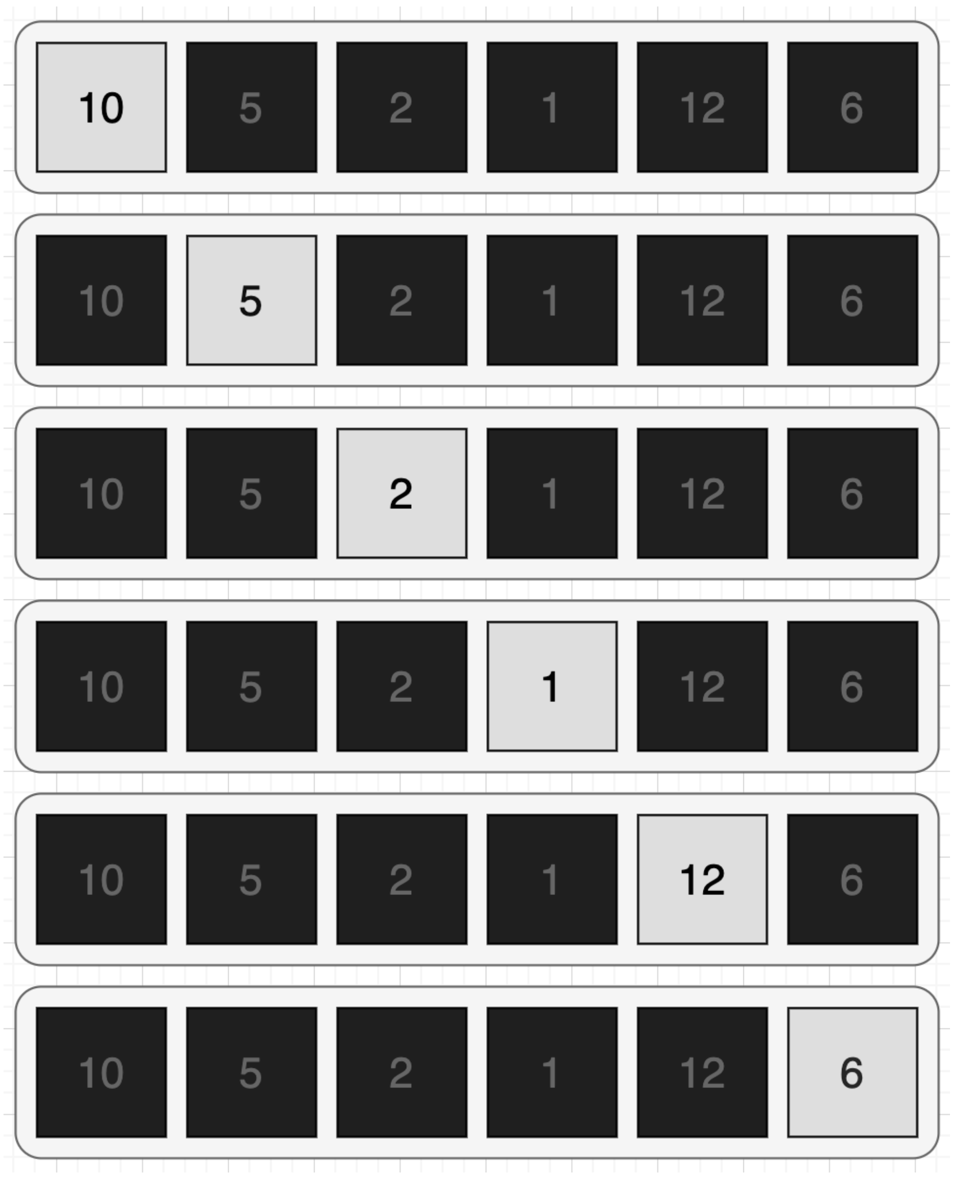
application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
account management	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find type and value	variable name	type and value

Typical symbol-table applications

Linear Search



Computers can not see the whole array in the memory. It can only access one element at a time.
⇒ So, we have to iterate through all the elements to see if we have our target element.



Implementation

1_linear_search.cpp

```cpp
bool find(vector<int> nums, int target){
    for(int i=0; i<nums.size(); i++){
        if(nums[i] == target)
        return true;
    }
    return false;
}
```

Complexity Analysis

Time Complexity

- Best Case Time Complexity: θ(1)
- Worst Case Time Complexity: θ(n)

Space Complexity

- Input Space Complexity: θ(n)
- Output Space Complexity: θ(1)
- Auxiliary Space Complexity: θ(1)

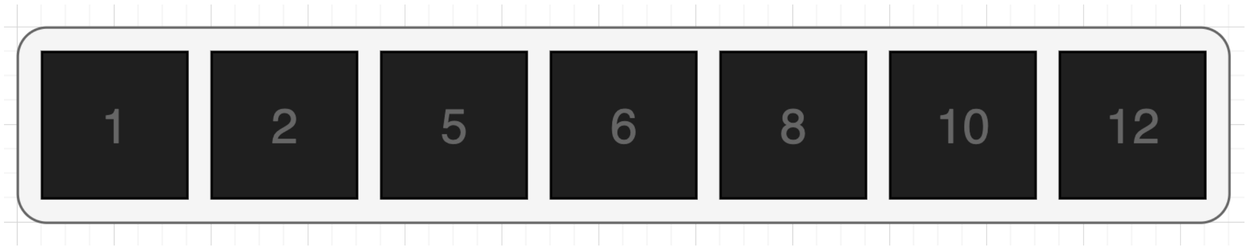
Binary Search

Divide and Conquer approach that repeatedly divides the search space in half until the target element is found or determined to be absent.
⇒ This approach will only work with **sorted** arrays.

⇒ Idea behind this algorithm is reducing the number of elements to search from using logical reasoning.

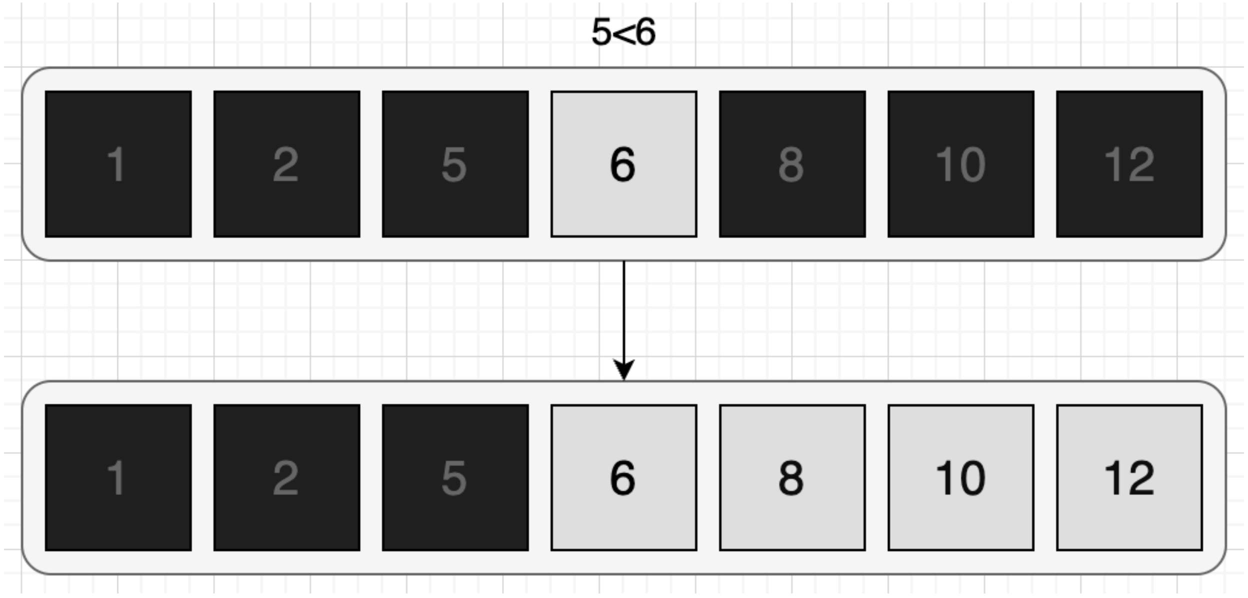
Example & Visualization

Target element is **5.**



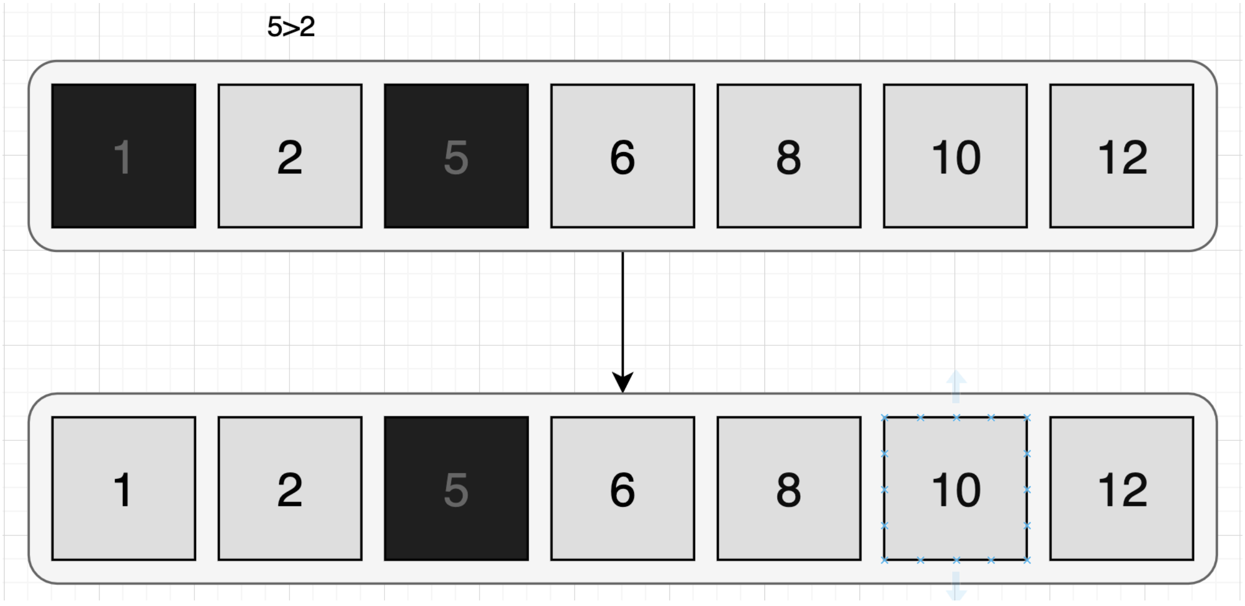
Let’s compare the target element with the middle element in the whole array.

⇒ Target = 5, Middle Element = 6



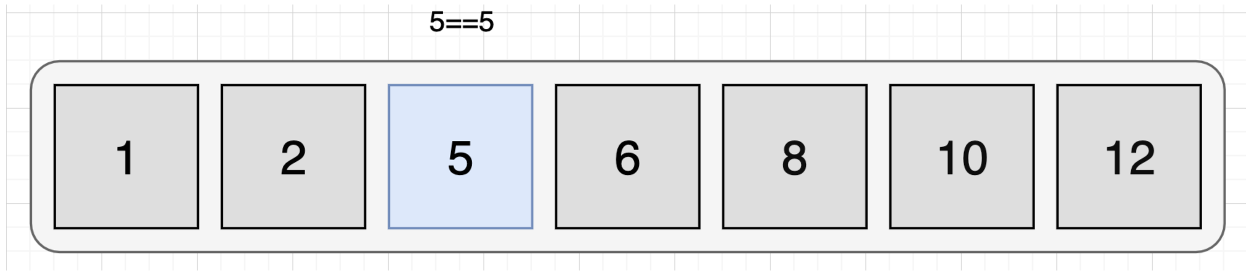
as target < mid element, by just one comparison we can remove all elements after mid element from search space. Because, the array is sorted so none of the elements after the mid element will be the target number.

⇒ Next, pick the next mid element from the elements remaining in the search space. Which will be 2.



as target > mid element, we can remove all the elements before the mid element from our search space. Because the array is sorted so none of the elements before the mid element will be the target number.

⇒ Next, pick the next mid element from the elements remaining in the search space. Which will be 5.



We have found our target element in just 3 comparisons. Whereas, if instead we did linear search, it would have taken 7 comparisons.

Pseudocode

```
function find(array, target){
    if len(array) == 0:
        return false;

    if target < mid_element of array
        find(array[0:n/2])
    if target > mid_element of array
        find(array[n/2:n])

    return true;
}
```

Implementation

2_binary_search.cpp

```
bool find(vector<int> nums, int target, int low, int high){
    int n = nums.size();

    // check if we have exhausted the search space
    if (low > high) return false;

    // find the middle element's index
    int mid_ind = (low+high)/2;

    if (target < nums[mid_ind])
        return find(nums, target, low, mid_ind-1);

    else if (target > nums[mid_ind])
        return find(nums, target, mid_ind+1, high);

    return true;
}
```

Complexity Analysis

Time Complexity

⇒ The implementation of binary search above is recursive.

Let's write a time complexity expression for that implementation.

T(find in array of size n) = T(find in array of size n/2) + θ(1) 1

The θ(1) part comes from the constant number of comparisons per iteration. (in binary search, in every iteration, the algorithm will have to do 2 comparisons at max. which is constant.)

This is analogous to other divide and conquer approaches. Let's use the **substitution** method to solve this recurrence.

⇒ Substitute n = n/2 in expression 1,
T(n/2) = T((n/2) / 2) + θ(1)
       = T(n/4) + θ(1)

Now, replace the T(n/2) value in expression 1.
T(n) = T(n/4) + θ(1) + θ(1)

⇒ Substitute n = n/4 in expression 1, expression becomes
$T(n/4) = T(n/8) + \theta(1)$

Now, replace the T(n/4) value in expression 1.
$T(n) = T(n/8) + \theta(1) + \theta(1) + \theta(1)$

⇒ In every step, we are dividing n by 2. So, this process will stop when we can not further divide n by 2.

Let's say after k steps, we have to stop the process.
$n / 2^k = 1$

If we solve this expression, **$k = \log_2(n)$**

⇒ So, the expression 1 becomes,
$T(n) = \theta(1) + \theta(1) + \ldots k\text{-times} \ldots + \theta(1)$

$T(n) = \log_2(n)$

- Best Case Time Complexity: $\theta(1)$
- Worst Case Time Complexity: $\theta(\log(n))$

Space Complexity

- Input Space Complexity: $\theta(n)$
- Output Space Complexity: $\theta(1)$
- Auxiliary Space Complexity: $\theta(1)$

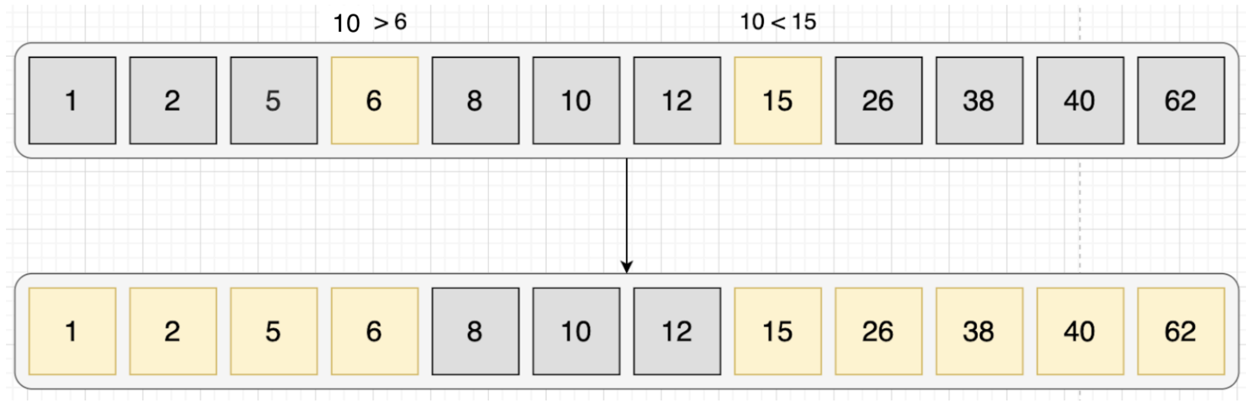
Ternary Search

Similar to binary search, instead of dividing the search space by a factor of 2, divide it in 3 parts.

⇒ So, in every iteration we can remove ⅔ (67%) part of the search space.

Example & Visualization

Target Element = **10**



Pseudocode

```
function find(array, target){
    if array length is zero
        return false

    mid1 = ⅓ rd element of array
    mid2 = ⅔ rd element of array

    if target < mid1 and target < mid2
        find(array[0: mid1], target)

    if target > mid1 and target < mid2
        find(array[mid1: mid2], target)

    if target > mid1 and target > mid2
        find(array[mid2: n], target)

    return true
}
```

Complexity Analysis

The time complexity expression will be similar to binary search,

```
T(n) = T(n/3) + θ(1)
```

The θ(1) part comes from the constant number of comparisons per iteration. (in ternary search, in every iteration, the algorithm will have to do 3 comparisons at max. which is constant.)

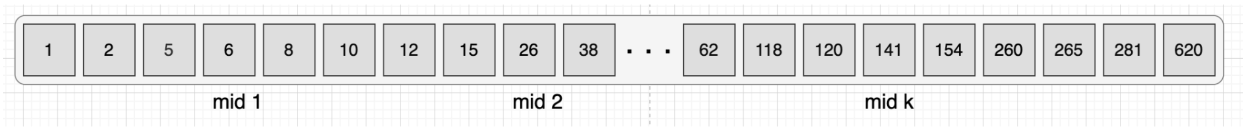
If we solve this recurrence,

⇒ Time Complexity: $\theta(\log_3(n))$

N-ary Search

Now, the greedy mind of us might think that the more search space we can remove the better, right?

Let’s divide the whole array into k parts,



Pseudocode

```
function find(array, target){
    if array length is zero
        return false

    mid1 = 1/k th element of array
    mid2 = 2/k th element of array
    mid3 = 3/k th element of array
    .
    .
    .

    if target < mid1
        find(array[0: mid1], target)
    if target < mid2
        find(array[mid1: mid2], target)
    .
    .
    .
    if target > midk
        find(array[midk: n], target)

    return true
}
```

Complexity Analysis

Now, if we want to write expression for time complexity,
We can **not** write,
$T(n) = T(n/k) + \theta(1)$

Because the number of comparisons per iteration is not constant now, the more partitions you do of the array, more comparisons have to take place.

$T(n) = T(n/k) + \theta(k)$

If we solve this recurrence, we get $T(n) = k * \log_k(n)$

⇒ The larger the value of k (number of partitions), the bigger the time complexity.

⇒ That's why it's not a good idea to divide the array into more parts.

Note: Binary Search is often chosen because of its simplicity.

Resources

- Algorithms 4e, Section 3.1
- Binary Search Visualization

- Polls