Searching

Created	@July 11, 2022 8:13 PM
Class	
Type	
Materials	
Reviewed	☐

`Searching algorithms` as the name suggests that it is trying to search for something.

Why do we even want to learn searching ?

Starting from ecommerce websites to ott platforms like netflix hotstar, everywhere we have the use case of searching. Somewhere we are searching for products, sometimes we are searching for movies and shows etc. Searching is one of the most fundamental problems in computer science.

To solve the problem of searching there is no dedicated one or two algorithms. Based on different situations we use different algorithms and data structures to solve the problem of searching in an optimised fashion.

Search Space

Search space is defined as the set in which we are going to search for a corresponding element.

For example:

- if we want to search in an array, then our search space will be the array.
- Let's say we need to search for square root of x, then we can say square root of x will lie somewhere in the range 1 - x/2 , so the search space becomes all the elements in the range [1, x/2]

Motivation problem:

Given an array in an unsorted fashion, and an element x, find whether the element is present in the array or not ?

Ex: `[4,5,2,7,0,1,3,5,6]` → x = 3 → ans - index 6 we have 3

```
function search(arr, x) {
  // this function searches for the index on which x is present
  for(let i = 0; i < arr.length; i++) {
    if(arr[i] == x) return i;
  }
  return undefined;
}
```

So in the above algorithm, we just went on each index and checked whether the current element is equal to the element `x` or not. This seems to be like the most brute force way to solve the problem of searching.

In the above solution, we have the array as our search space, and in every iteration we are reducing the search space by 1. This process of reducing our search space by one in every iteration is called as `Linear Search`. So in the, linear search algorithm we have to just go to every single index of the search space and check if we found the value or not.

It is called linear because we are reducing the search space in linear fashion.

Binary Search

Binary search is one of the most efficient searching algorithm. A lot of people think that binary search can be only applied on a search space which is sorted, but that is **not** correct. In different situation binary search can also be applied on unsorted search spaces.

Binary search works efficiently because in every iteration it reduces the search space into half of it's original size.

n → n/2 → n/4 → n/8 …..

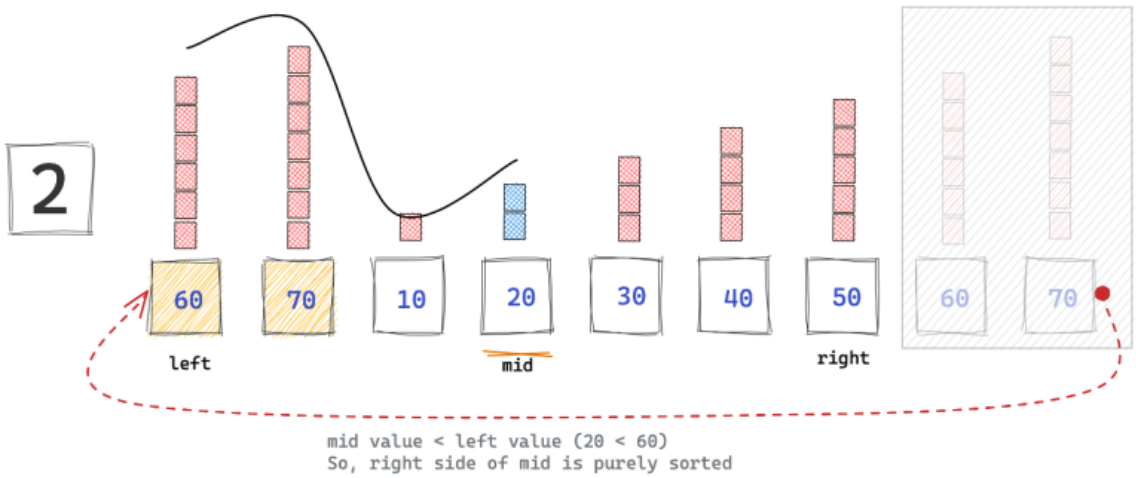
If the search space is of size n, then in the first iteration it reduces the search space to size n/2 then in the next iteration it reduces it to n/4 and so on and so forth.

That means in every iteration it will discard half of the elements in the search space.

This is the reason it is called as Bi-n-ary → Bi stands here for the half division of search space.

Where can we apply binary search ?

Any search space that can be divided into two halfs, such that one half is different than other half based on some property.



The above array is unsorted, still we can apply binary search here.

Ex: 60,70,80,90,100,110,3,5,10

In both of the above search spaces i.e. in the example and the diagram, we can divide the search space into two parts such that one part is always sorted and other is always unsorted.

So these properties can vary from problem to problem, but the main point is that we should not restrict our thinking about binary search to only sorted situation .

How to proceed with a searching problem ?

To solve a problem using searching algorithms, first of all try to see that the output is a part of some search space or not, if yes then can we partition the search space into two part such that one part is diff from other based on some property. If yes then apply binary search.

Simple problem where binary search can be applied:

Given an array which is arranged in ascending order (sorted) and an element x, try to find on which index this element x lies.

[1,2,4,6,8,10,12,14,19] x = 14

- In this problem our search space is the whole array, so in binary search we have to mark starting and ending point of the search space.
- In this case starting point of the search space is index `0` and ending point is index `arr.length - 1`
- Then we calculate the mid point of the search space so that we can divide it into two parts: `mid = (start + end)/2`
- Now we will try to find a property based on which the left half is always different than the right half.
- Property:
  - If the element on the `mid` index is greater than `x` then everything to the right of mid will also be greater than `x` so `x` can never be present on the right side, it will always be on the left side.
  - If the element on the `mid` index is lesser than `x` then everything to the left of the mid will also be lesser than `x` so `x` can never be present on the left side, it will always be on the right side. If we have to reduce the search space to the right we can do `start = mid+1`
  - So overall property says that, `x` will always lie on one half of the search space, so we can discard the other half
- If the element at index mid is equal to x, then mid is the answer

Dry Run

[1,2,4,6,8,10,12,14,19] x = 14

start	end	mid	arr[mid]	arr[mid] == x	arr[mid] > x	arr[mid] < x
0	8	4	8	No	No	Yes
5	8	6	12	No	No	Yes
7	8	7	14	Yes		

Code Of Binary Search

```
function binarySearch(arr, x) {
  let start = 0, end = arr.length - 1;
  while(start <= end) {
    //let mid = Math.floor((start + end) / 2);
    let mid = Math.floor(start + (end - start)/2);
    if(arr[mid] == x) return mid;
    if(arr[mid] > x) {
      // reduce to the left
      end = mid - 1;
    } else {
      // reduce to the right
      start = mid + 1;
    }
  }
  return undefined;
}
```

Currently mid can overflow, due to the fact that we might add very large numbers.

Can we modify mid ?

```
mid = (start + end) / 2
// add and subtract start in the numerator
mid = (start + end + start - start) / 2
mid = (2*start + end - start ) / 2
mid = start + (end - start)/2
```

Given an array of integers, arranged in ascending order (sorted). Also given an element x, find the index of the first element greater than x. (UPPER BOUND)

Ex: [1,2,3,3,3,4,5,5,6,7] x = 5 → ans - index 8

[1,2,3,3,3,4,5,5,6,7] x = 3, ans - index 5

[1,2,3,3,3,4,4,4,6,7] x = 5, ans - index 8

Let's assume all the elements ≤ x to be True and all the elements remaining as false.

[1,2,3,3,3,4,5,5,6,7] x = 5

[T,T,T,T,T,T,T,T,F,F]

And now we can visualise the problem as finding the first False in the array.

Now if we divide our search space into two parts, then one part will be having all same elements and other part will be having mixed elements.

[T,T,T,T,T,T,T,T,F,F] → Then after dividing this into two parts, one part is having all true's and the right part is having mix of true and false

[T,T,F,F,F,F,F,F,F,F] → After dividing this into two parts, one part is having all false's and the left part is having mix of true and false

That means we can apply binary search here. Now you might be thinking why assigning true and false makes sense ? Because we have to find first element greater than x, then all the elements having value > x are potential candidates to become the answer. So we marked potential candidates with same label and remaining with other label.

We will try to divide our search space, if the middle element is less than or equal to x, then we can say that all elements greater than are always on right side. The left side has **no** potential answers. [TTTTTTTFF]

But if the middle element is greater than x, then this element is part of one of potential candidates of our answer. So we can store mid as answer and we can try to find better answer to the left of it, as on the right side no element will be the answer and mid is better than them. [TTTFFFFFFFF]

So we will always discard the same element side and always chose mixed side hence we can apply binary search.

```
function upperBound(arr, x) {
  let start = 0, end = arr.length - 1;
  let ans = arr.length;
  while(start <= end) {
    let mid = Math.floor(start + (end - start) / 2);
    if(arr[mid] <= x) {
      // discard left half
      start = mid + 1;
    } else {
      // element at mid can be a potential answer
      ans = mid;
```

```
      end = mid - 1; // go and find something even better on left side
    }
  }
  return ans;
}
```

Time complexity of Binary Search

In every iteration we reduce the search space by half. In every iteration we are doing constant task. So time complexity will be no_of_iterations * constant .

Iteration	Size of search space
0	n
1	n / 2
2	n / (2^2) = n / 4
3	n / (2^3) = n / 8
4	n / (2^4) = n / 16
.	.
.	.
K	n / (2^k)

So let's say in the kth iteration algorithm stops, so total iterations are K + 1. If we can get value of K then that will be denoting time complexity.

Now we know if K is the last iteration, that means that in the last iteration the search space becomes 1.

```
n / (2^k) = 1
n = 2^k
taking log on both sides
log2(n) = log2(2^k)
logn2(n) = klog2(2)
k = log2(n)
Time complexity will be O(k+1) -> O(logn)
```

NOTE: In writing logarithmic complexities, we can avoid writing base because we can convert any one base to other by multiplying a constant, and constants are avoided in Big O notation.

So, finally Time complexity of Binary Search is `O(logn)`

Given an array of integers, arranged in ascending order (sorted). Also given an element x, find the index of the first element greater than or equal to x. (LOWER BOUND) - HW

Example: [1,2,3,3,4,5,6,6,6,7] x = 3, ans - index 2

[1,2,3,3,4,6,6,6,7] x = 5, ans - index 5