Binary Search

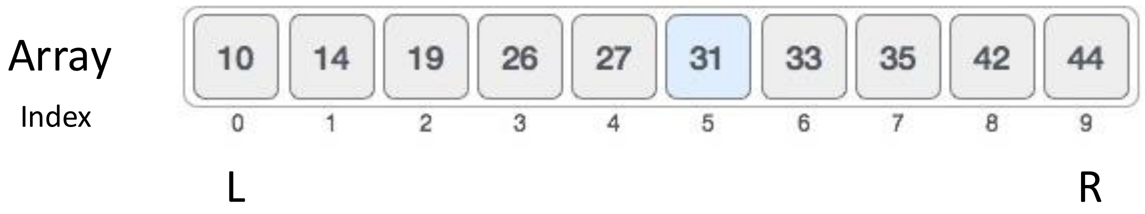
琪石职业发展俱乐部

Qishi CPC

Binary Search – How it works

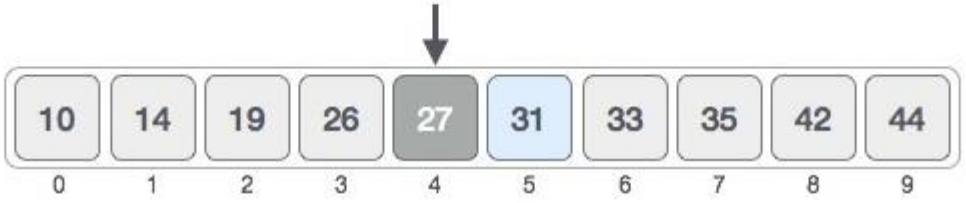
Problem to solve: Find a value in a sorted array

Search a sorted array by repeatedly dividing the search interval in half.
For Example: Search the location of "31" in the below sorted array

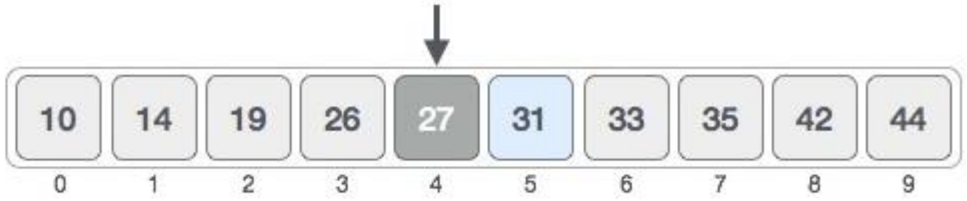


Step 1: Determine the middle of the array: mid = (L + R)/2 = (0 + 9)/2 = 4.5 (integer value of 4)

Step 2: check if array[4] == 31, if so, we find it.
if array[4] < 31, search right half of the array, else search left half of the array
In our example, array[4] = 27 < 31, the repeat step 1 and 2 on the right half of the array

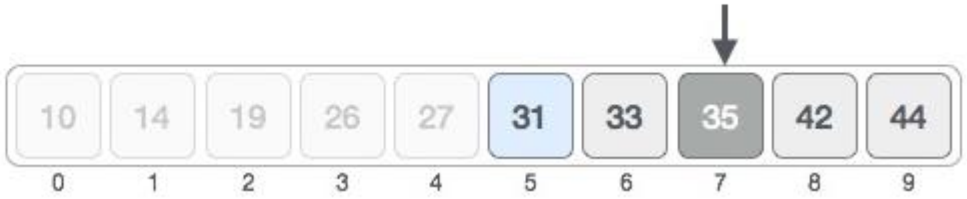


Binary Search – How it works

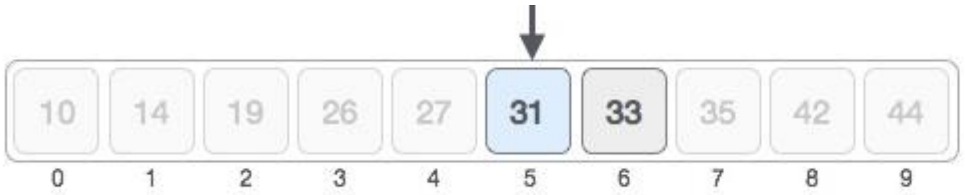


Step 1: Determine the new middle of the sub array: mid = (5 + 9)/2 = 7

Step 2: check if array[7] == 31, array[7] = 35 > 31, the repeat step 1 and 2 on the left half of the sub-array



Repeat Step 1: Determine the new middle of the sub array: mid = (5 + 6)/2 = 5
Repeat Step 2: array[5] match



Binary Search – Implementation

```
# Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l)/2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1
```

Termination of the loop

Mid = (l +r)/2

Binary Search – Implementation

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky ...

— Donald Knuth

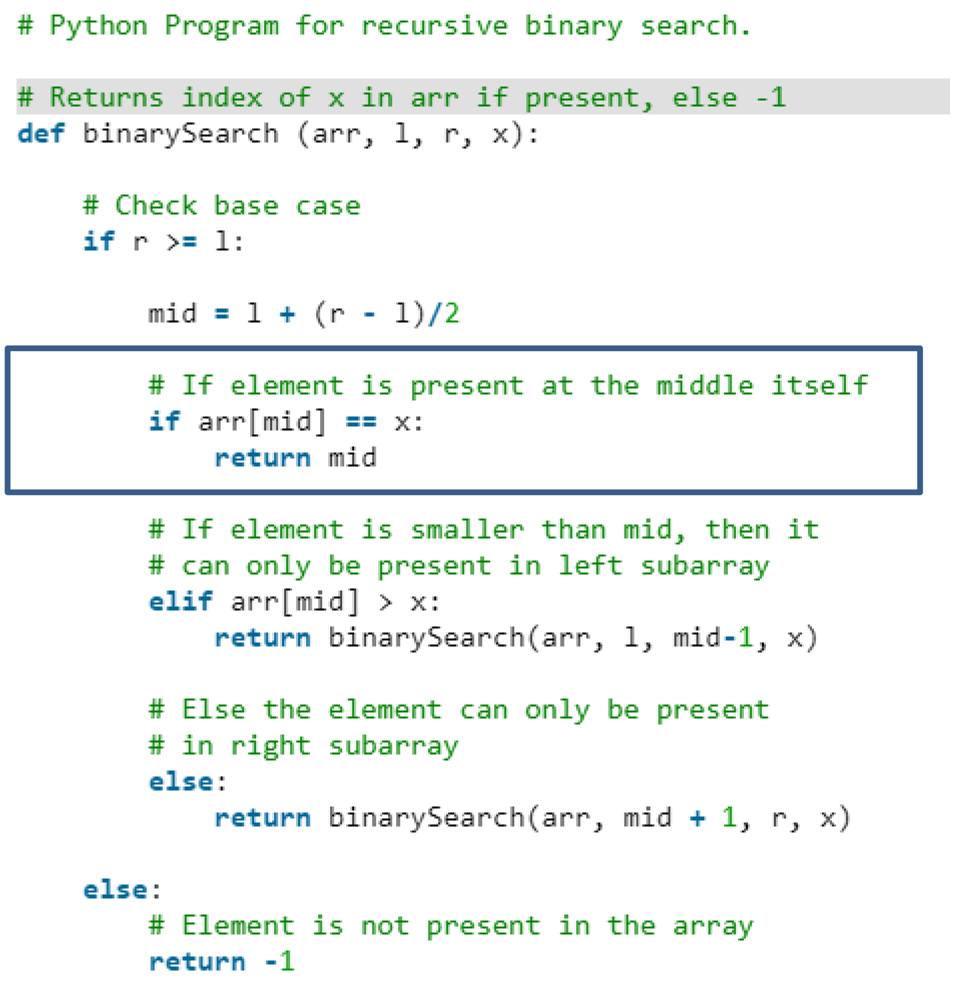
(Turing Award 1974, Author of "The Art of Computer Programming")

- **90%** of **professional** programmers failed to provide a correct solution after several hours of working on it when assigned by Jon Bentley
- mainly because the incorrect implementations failed to run or returned a wrong answer in rare **edge case**

edge case:

1. (L + R)/2, the value of L+R may exceed the range of the integer. L + (R-L)/2
2. Infinite loop occur if the exit condition for the loop are not defined correctly, if L >R, search has failed, must exit the loop. (most programmers made an error here)

Binary Search – Skip the check for the middle



```python
# Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l)/2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1
```

- The algorithm checks whether the middle element is equal to the target in every iteration.
- Some implementations leave out this check during each iteration.
- The algorithm would perform this check only when one element is left (L=R).
- This results in a faster comparison loop, as one comparison is eliminated per iteration. However, it requires one more iteration on average

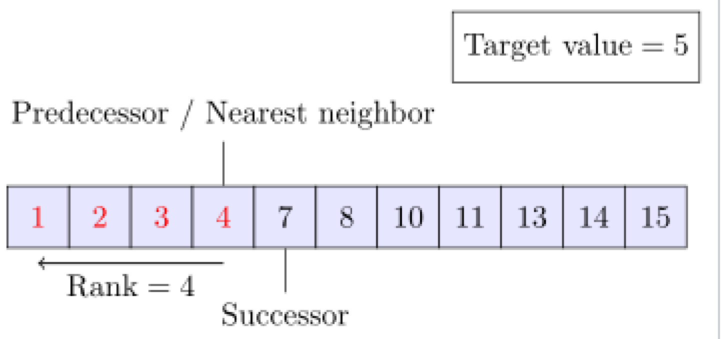
Binary Search – Duplicated Elements

If the array to be searched was [1,2,3,3,4,4,4,5,5,5,5,5,6] and the target is 5

How to find the left most element or the right most element?

Binary Search – Approximate Match

- It is trivial to extend binary search to perform approximate matches because binary search operates on sorted arrays.
- Binary search can be used to compute, for a given value:
    a. its rank (the number of smaller elements)
    b. predecessor (next-smallest element)
    c. Successor (next-largest element)
    d. nearest neighbor.
    e. Range queries seeking the number of elements between two values can be performed with two rank queries.



Binary search can be adapted to compute approximate matches. In the example above, the rank, predecessor, successor, and nearest neighbor are shown for the target value 5, which is not in the array.

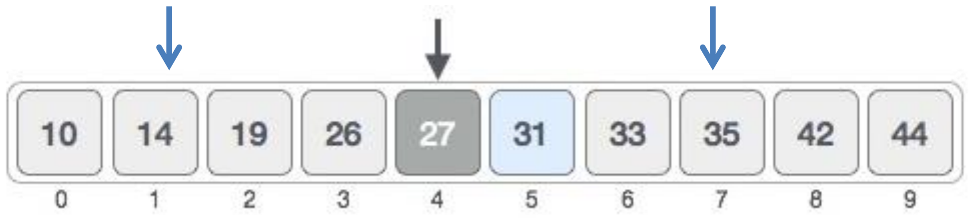
Binary Search – Compared to other schemes

Binary Search Run Time: O(log n)

- Sorted arrays with binary search are a very inefficient solution when insertion and deletion operations are interleaved with retrieval
- Sorted arrays can complicate memory use especially when elements are often inserted into the array.
- There are other data structures that support much more efficient insertion and deletion as well as faster exact matching and set membership
- **Binary search can be used for efficient approximate matching.**
- **There are some operations, like finding the smallest and largest element, that can be performed efficiently on a sorted array.**

Binary Search Variations – Uniform Binary Search

Use a lookup table that contains the mid points instead of calculating



Lookup_table = [4,3,1]

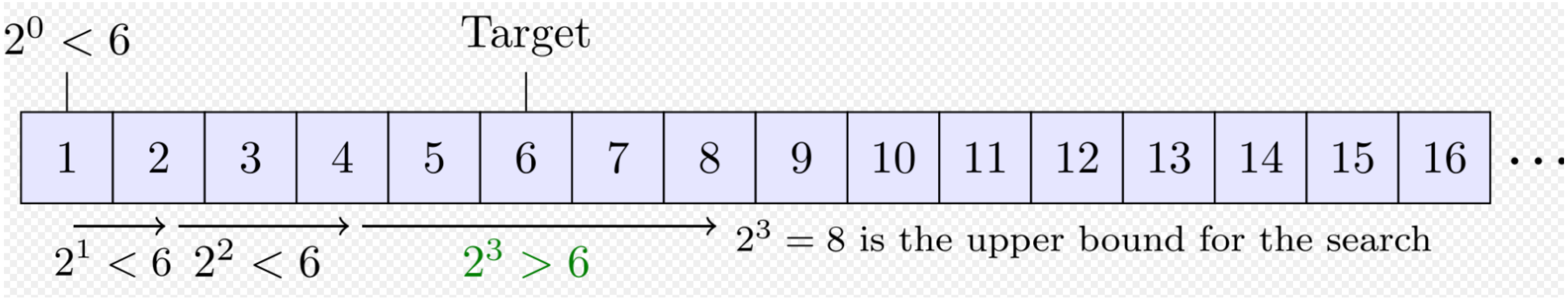
Start with x = lookup_table[0],
if Target < array[x] then x = x - lookup_table[0]
if Target > array[x] then x = x + lookup_table[0]

Binary Search Variations – Exponential Search

For the list does not have a upper limit

Step 1: loop through the sorted array by 2^n incremental, and stop when array[2^n] > Target

Step 2: Use binary search where 2^n is the upper bound

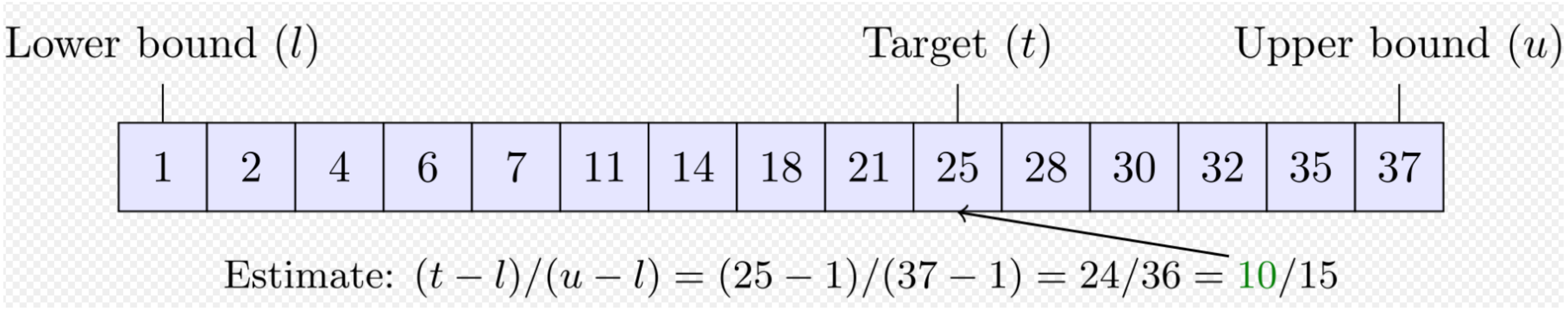


Exponential search becomes an improvement over binary search only if the target value lies near the beginning of the array

Binary Search Variations – Interpolation Search

Instead of using L + (R-L)/2 to estimate mid point, it use linear interpolation to estimate the mid point

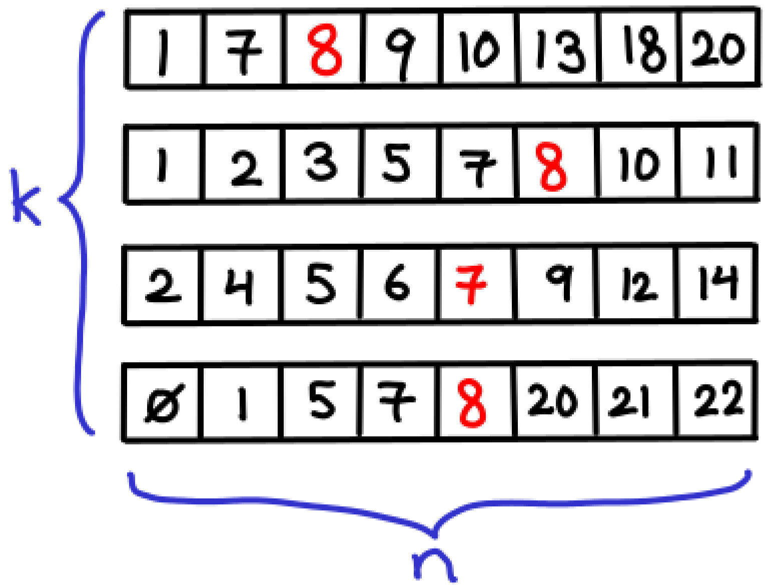
In the array below, Target is 25. the length of the array is 15.
Array[0] = 1, Array [15] = 37, if Target = 25, what is its linear interpolation estimated index?
15 x (25-1)/(37-1) = 10



- When linear interpolation is used, and the distribution of the array elements is uniform or near uniform, interpolation search makes O(log log n) comparisons.
- In practice, interpolation search is **slower** than binary search for small arrays, as interpolation search requires extra computation. Its time complexity grows **more slowly than binary search, but this only compensates for the extra computation for large arrays**

Binary Search Variations – fractional cascading

If we have K shorted arrays, each of size n, we need to search one element in each of the k arrays (or its predecessor if it doesn't exist

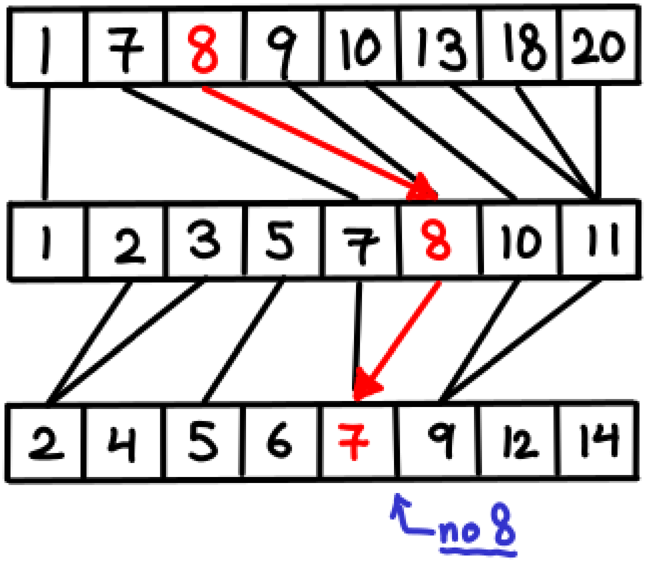


1	7	8	9	10	13	18	20
1	2	3	5	7	8	10	11
2	4	5	6	7	9	12	14
Ø	1	5	7	8	20	21	22

Option 1: perform binary search in each of the array. O(k log n) run time

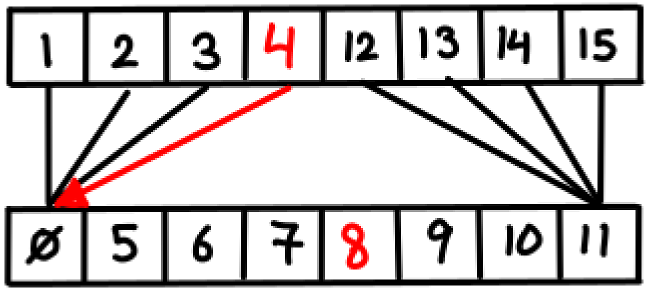
Binary Search Variations – fractional cascading

Option 2: for every element in the first array, let's give it a pointer to the element with the same value in the second array.
Then once we've found the item in the first array, we can just follow these pointers down in order to figure out where the item is in all the other arrays.



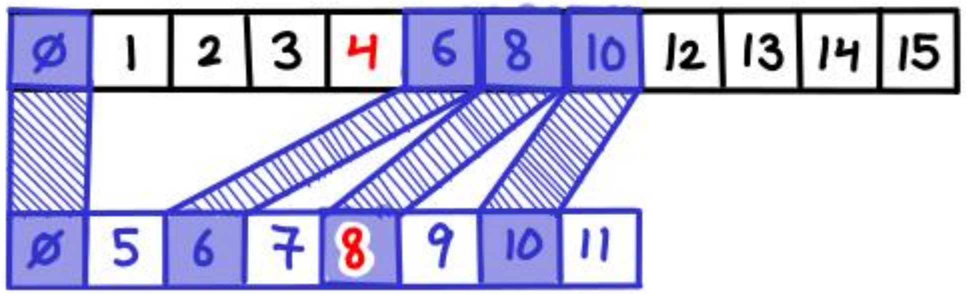
Binary Search Variations – fractional cascading

But there's a problem in Option 2: sometimes, these pointers won't help us at all. In particular, if a later lists is completely "in between" two elements of the first list, we have to redo the entire search, since the pointer gave us no information that we didn't already know.

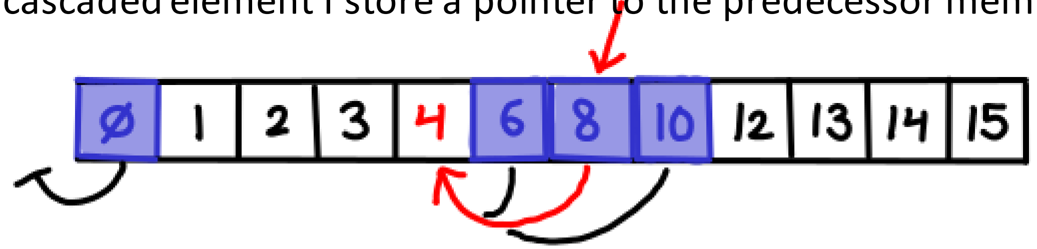


Binary Search Variations – fractional cascading

- Consider the case where *k = 2*; everything would be better if only we could guarantee that the first list contained the right elements to give you useful information about the second array.
- We could just merge the arrays, but if we did this in the general case we'd end up with a totally merged array of size , which is not so good if *k* is large.
- But we don't need all of the elements of the second array; **every other item will do!**

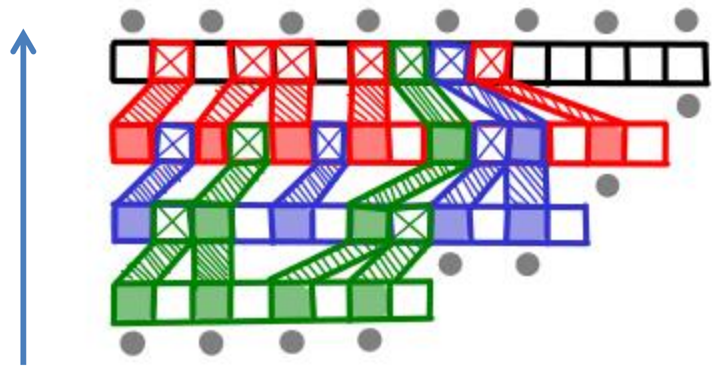


- When I follow a pointer down, I might end up on an element which is not actually a member of the current array (it was one that was cascaded up).
- For every cascaded element I store a pointer to the predecessor member element.



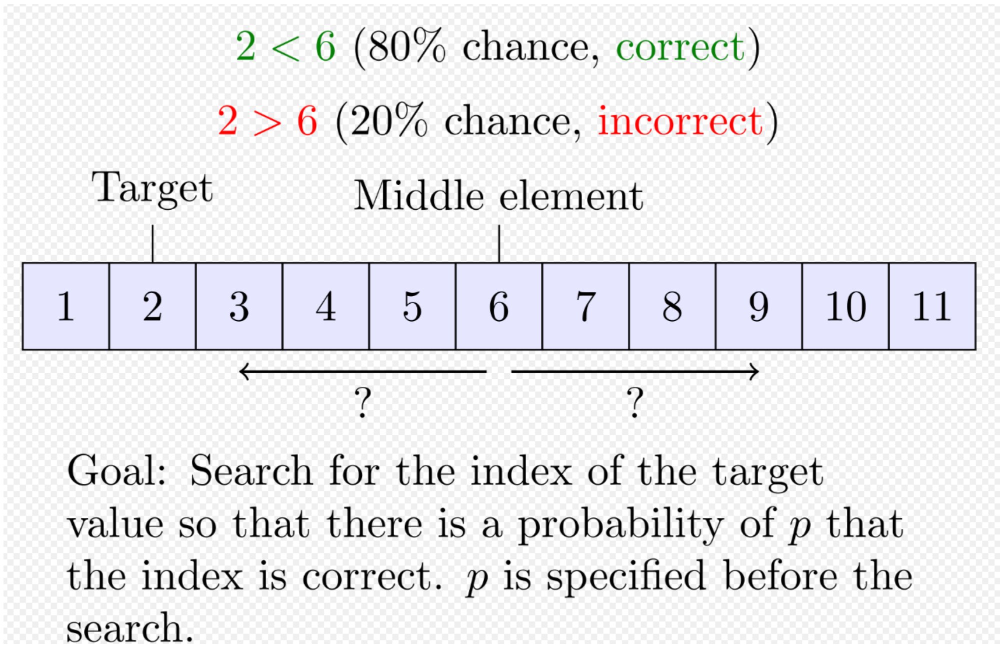
Binary Search Variations – fractional cascading

- If we take every other element and merge it with the previous array, the final merged array will be n + n/2 + n/4 + n/8 … = 2n



Binary Search Variations – Noisy Binary Search

- **Ulam's game**, or the **Rényi–Ulam game**, is a mathematical game similar to the popular game of twenty questions where one attempts to guess an unnamed object with yes–no questions, but where some of the answers may be wrong.
- For each pair of elements, there is a certain probability that the algorithm makes the wrong comparison. Noisy binary search can find the correct position of the target with a given probability that controls the reliability of the yielded position.


$2 < 6$ (80% chance, correct)
$2 > 6$ (20% chance, incorrect)
Target
Middle element
1 2 3 4 5 6 7 8 9 10 11
?
?

Goal: Search for the index of the target value so that there is a probability of p that the index is correct. p is specified before the search.

Reference

- https://en.wikipedia.org/wiki/Binary_search_algorithm
- http://blog.ezyang.com/2012/03/you-could-have-invented-fractional-cascading/