REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

Introduction to Procedural Programming - BLM 1012

Ternary Search

INSTRUCTOR: Doç.Dr. Fatih AMASYALI

18011906 MOUNES ZAWAL

June, 2021

Definition

Ternary search is a divide and conquer searching strategy for determining the position of a certain value in a sorted array. The sorted array is broken into three parts in ternary search, and then it identifies which part the element exists in.

Ternary search works similar to Binary search. The only difference is instead of dividing the array into 2 parts, the array is divided into three parts and 2 parts are rejected on each iteration. That is, the array is reduced to 1/3 of its size on each iteration.

Steps to perform Ternary Search on a sorted array:

- ***To begin, we compare the key to the mid1 element. If the results are equal, we return to mid1 element.***
- ***If not, the key is compared to the element at mid2. If the results are same, we return mid2.***
- ***If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.***
- ***If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.***
- ***If not, then we recur to the second (middle) part.***

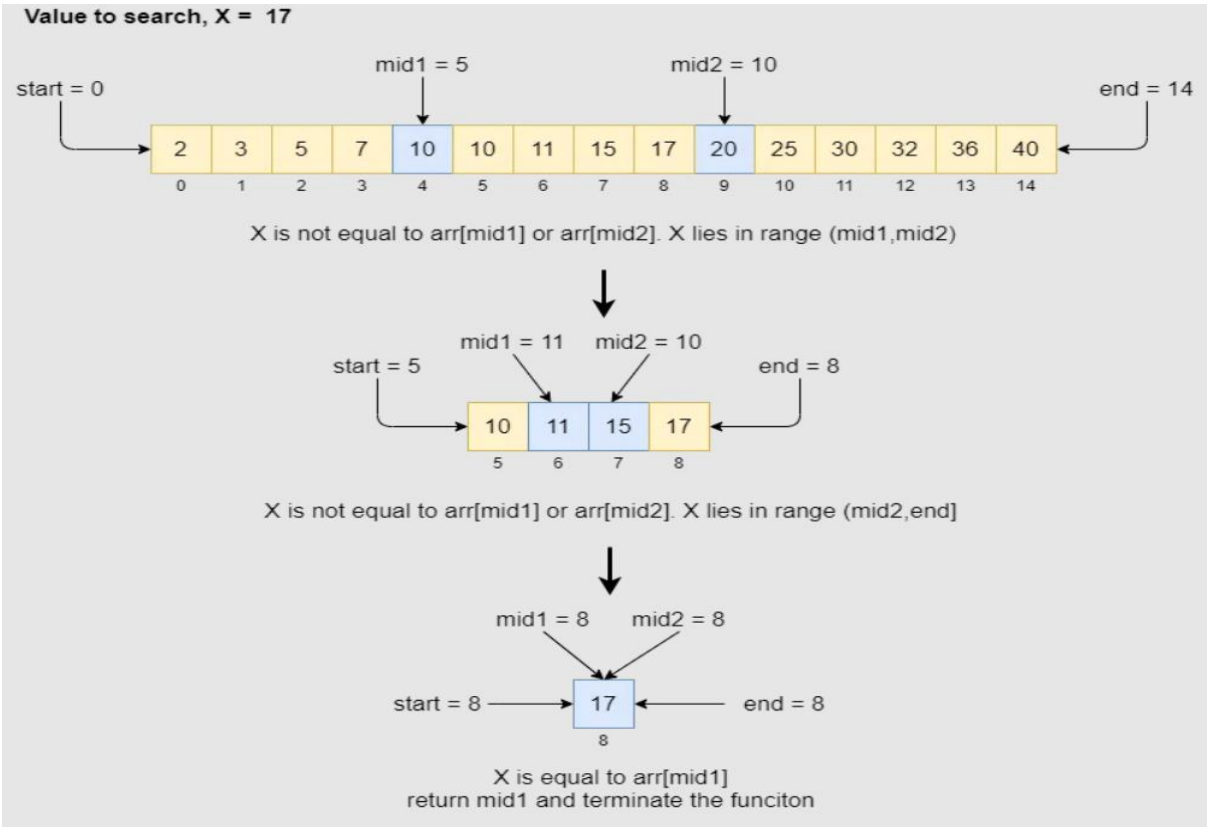


Figure 1-Ternary Search visual example

Pseudo Code

```
arr: Array we want to search
x:value we want to search in arr
Integer Ternary_Search( arr , x )
      start := 0
      end := arr.LENGTH – 1
      WHILE start <= end DO
            mid1 := (end - start)/3 + start
            mid2 := 2*(end - start)/3 + start
            IF x == arr[mid1] THEN // found at mid1
                  return mid1
            ELSE IF x == arr[mid2] THEN // found at mid2
                  return mid2
            ELSE IF x < arr[mid1] THEN // search in part1
                  end := mid1 – 1
            ELSE IF x < arr[mid2] THEN // search in part2
                  start := mid1 + 1
                  end := mid2 – 1
            ELSE //if target value is not in part1 or part2, go to part3
                  start := mid2 + 1
            END ELSEIF
      END WHILE
      return -1 // if value not found return -1
END FUNCTION
```

Implementation

The following is the C code implementation of Ternary search:

```
int ternarySearch(int startIndex, int EndIndex, int target, int *array,int
*number_of_moves){

    *number_of_moves=0; // variable to count the number of iterations
    while (EndIndex >= startIndex) {
        (*number_of_moves)++;
        // Find the mid1 and mid2
        int mid1 = startIndex + (EndIndex - startIndex) / 3;
        int mid2 = EndIndex - (EndIndex - startIndex) / 3;

        // Check if the target equals to any middle point
        // if this is the case, return the mid index
        if (array[mid1] == target) {
            printf("number of moves: %d\n",(*number_of_moves));
            return mid1;
        }
        if (array[mid2] == target) {
            printf("number of moves: %d\n",(*number_of_moves));
            return mid2;
        }

        // Since target is not present at mid,
        // check in which region it is present
        // then repeat the Search operation
        // in that region

        if (target < array[mid1]) {

            // The target is between startIndex and mid1
            EndIndex = mid1 - 1;
        }
        else if (target > array[mid2]) {
            // The target is between mid2 and EndIndex
            startIndex = mid2 + 1;

        }
        else {
            // The target is between mid1 and mid2
            startIndex = mid1 + 1;
            EndIndex = mid2 - 1;
        }
    }

    printf("number of moves: %d\n",(*number_of_moves));
    // target not found
    return -1;
}
```

Time and Space Complexity

Keeping in mind that the array is reduced to 1/3 of its size on every iteration. Thus, the recurrence relation of Ternary search could be constructed as:

$$T(N) = T\left(N/3\right) + C \text{ where } C \text{ is a constant}$$

$$T(N) = T\left(N/3\right) + C \quad (1)$$

$$T\left(N/3\right) = T\left(N/3^2\right) + C \quad (2)$$

$$T\left(N/3^2\right) = T\left(N/3^3\right) + C \quad (3)$$

.....

.....

$$T(9) = T(3) + C \quad (k\text{-}1)$$

$$T(3) = T(1) + C \quad (k)$$

On adding equation 1 to k, we have:

$$T(N) = T(1) + K * C$$

Since N is divided by 3 each iteration:

$$N/3^k = 1$$

$$k = \log_3(N)$$

Therefore:

$$T(N) = T(1) + \log_3(N) * C$$

Since $T(1)$ and C are constants we ignore them:

$$T(N) = \log_3(N)$$

Therefore, the time complexity with Big-O notation is: $\mathbf{O(\log_3(N))}$

For Space Complexity:

with iterative Ternary search algorithm, the space complexity is $O(1)$

Ternary Search VS Binary Search and Linear Search

Advantages:

- ***Compared to linear search (checking each element in the array starting from the first), ternary search is much faster. Linear search takes, on average N/2 comparisons (where N is the number of elements in the array), and worst-case N comparisons. ternary search takes an average and worst-case $log_3(N)$ comparisons. So, for a million elements, linear search would take an average of 500,000 comparisons, whereas ternary search would take 13.***
- ***It's a fairly simple algorithm, though people get it wrong all the time.***
- ***Ternary search can solve all problems that are solvable using Binary Search, whereas Binary search cannot solve some problems that are solvable using Ternary search.***
- ***Ternary Search concept could be used to solve unimodal functions.***

Disadvantages:

- ***It works only on lists that are sorted and kept sorted (except if the list represents a unimodal function). That is not always feasible, especially if elements are constantly being added to the list.***
- ***Even though its time complexity is $log_3(N)$ and Binary Search's is $log_2(N)$, Binary Search is faster and preferred over ternary search when the list represents a linear function.***
- ***There is a great loss of efficiency if the list does not support random-access. You need, for example, to immediately jump to the second middle of the list. If your list is a plain array, that's great. If it's a linked-list, not so much. Depending on the cost of your comparison operation, the cost of traversing a non-random-access list could dwarf the cost of the comparisons.***

There are even faster search methods available, such as hash lookups. However, a hash lookup requires the elements to be organized in a much more complicated data structure (a hash table, not a list).

Here let me clarify two points:

- ***Ternary Search can solve problems Binary Search cannot.***
- ***Binary Search Considered faster than Ternary Search.***

Ternary Search advantage Over Binary Search:

There's no advantage when solving a linear function, however Ternary Search has its own property that Binary Search cannot handle. Ternary Search is used for finding maximum or minimum of a unimodal function.

A function is said to be unimodal if it obeys one of the following properties:

- *The function strictly increases first, reaches maximum and then strictly decreases.*
- *The function strictly decreases first, reaches minimum and then strictly increases.*

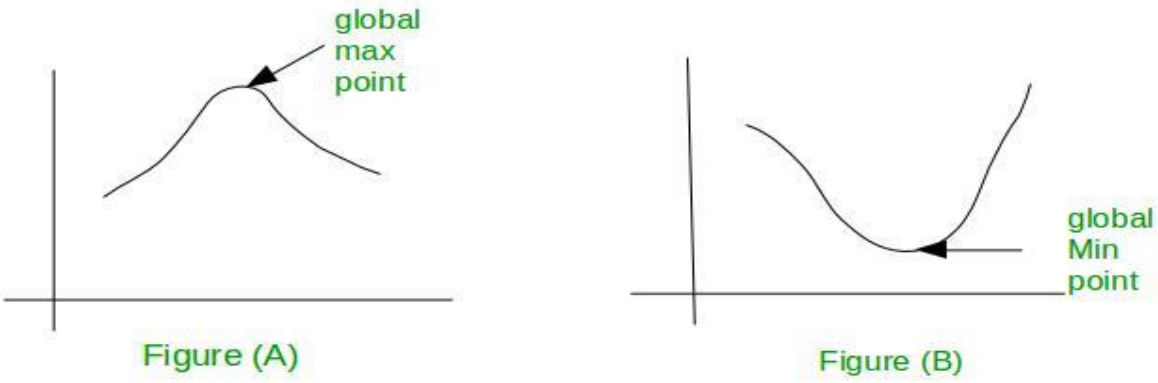


Figure 2-Visual representation of the Unimodal function (source: GeeksForGeeks)

Let's consider a simple example:

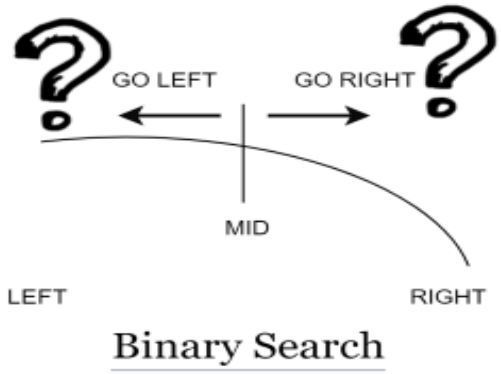
Given the following bitonic sequence:

1 2 3 3 3 4 4 5 5 7 6 5 5 5 4 2 2 2

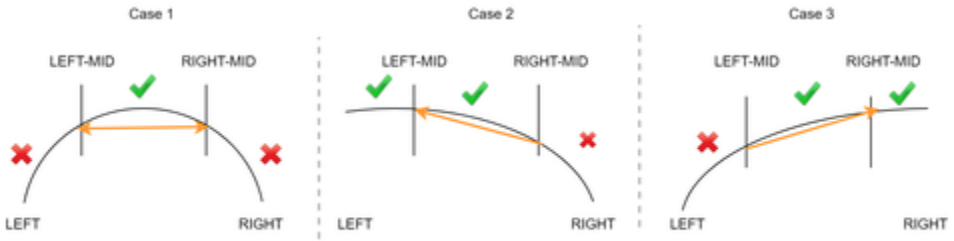
where numbers first appear in increasing order then suddenly started to decrease.

Can we find the maximum number in $log(N)$ with binary search?

In this array, value[low] = 1, value[high] = 2, let's say value[mid] = 5, now how you chose where the actual answer belongs, left? or right? you can't. Because this problem paradigm is not linear. This is a parabolic function to solve.



But in the case of Ternary Search, it will divide the curve into 3 segments using left-mid and right-mid. And based on the slope of left-mid and right-mid it can converge towards the maximum value.



Ternary search

By considering the above 3 cases, the ternary search will decide which segments to through away and which to dig in for the final result.

So, if you can define any function that forms a parabolic curve, you can find minima/maxima using a ternary search where a binary search will fail. The ternary search can be used to solve many problems that involve unimodular behaviour in them.

Let's take a look at C code implementation of ternary search algorithm with solving unimodular functions:

```
double ternary_search(double l, double r) {

    double eps = 0.0001;    //set the error limit here
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1); //evaluates the function at m1
        double f2 = f(m2); //evaluates the function at m2
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l);  //return the maximum of f(x) in [l, r]
}
```

Here eps is in fact the absolute error (not taking into account errors due to the inaccurate calculation of the function).

We can use a constant number of repetitions as a halting condition instead of the requirement r - l > eps. To provide the needed precision, the number of iterations should be chosen. Typically, the error limit in most programming challenges is 10^{-6} *and thus 200-300 iterations are sufficient. Furthermore, because the number of iterations is independent of the values of l and r, the number of iterations is equal to the needed relative error.*

Run Time Analysis: $T(N) = T\left(\frac{2N}{3}\right) + C = \theta(\log_3(N))$

It can be visualized as follows: every time after evaluating the function at points $m1$ and $m2$, we are essentially ignoring about one third of the interval, either the left or right one. Thus, the size of the search space is $\frac{2N}{3}$ of the original one.

Why Binary Search is faster than Ternary Search?

Binary Search reduces the array by $\frac{1}{2}$ on each iteration which makes its time complexity $\log_2(N)$, whereas Ternary Search reduces array size by $\frac{1}{3}$ on each iteration.

Ternary Search should be faster than Binary Search since $\log_2(N) \geq \log_3(N)$, but this is not the case. When calculate the time complexity of any algorithm, we generally ignore the constants. But the constants in Ternary Search are relatively larger than Binary Search, and that's what makes it slower.

Let's proof this with mathematics:

$For\ Binary\ Search, T_b(N) = C_b * \log_2(N)$ ...1

$For\ Ternary\ Search, T_t(N) = C_t * \log_3(N)$ ...2

Using the property $\log_b(N) = \frac{\log_e(N)}{\log_e(b)}$ ***in equation 1 and 2, we get:***

$$T_b(N) = C_b * \frac{\log_e(N)}{\log_e(2)} = \frac{C_b}{\log_e(2)} * \log_e(N) = 1.4426 * C_b * \log_e(N) \quad ...3$$

$$T_t(N) = C_t * \frac{\log_e(N)}{\log_e(3)} = \frac{C_t}{\log_e(3)} * \log_e(N) = 0.9102 * C_t * \log_e(N) \quad ...4$$

C_b ***= Number of Comparisons in each iteration of Binary Search = 2***
C_t ***= Number of Comparisons in each iteration of Ternary Search = 4***

Substituting C_t ***and*** C_b ***in equation 3 and 4, we get***

$$T_b(N) = 1.4426 * 2 * \log_e(N) = 2.885 * \log_e(N) \quad ...5$$

$$T_t(N) = 0.9102 * 4 * \log_e(N) = 3.6408 * \log_e(N) \quad ...6$$

On Comparing 5 and 6 equations:

$$T_b(N) < T_t(N)$$

Therefore, we can state that Binary Search is faster than Ternary search.

This happens because of the increase in the number of comparisons in Ternary search. In simple words, the reduction in the number of iterations in Ternary search is not able to compensate for the increase in comparisons. Hence, Binary search algorithm is preferred over the Ternary search when searching a sorted array.

Implementation Results

For calculating the time complexity in order to plot the results, I decided to do that by counting the number of iterations made by Ternary Search algorithm when given different array Sizes.

Since the algorithm is of $\log_3(N)$ ***degree:***

Array Size	*Expected iterations (worst case)*
10	*2*
100	*4*
1000	*6*
10000	*8*
100000	*10*
1000000	*13*
10000000	*15*
100000000	*17*

C implementation of creating number_of_iterations[] array :

```
int main(){
    int startIndex=0, EndIndex, targetIndex, target,byTen;
    int *array = (int*)(malloc(sizeof(int) * SIZE)); //SIZE= 100000000
    // Storing the number of iterations made by ternary search algorithm
    // to find the target
    int number_of_iterations[ITERATION_SIZE],number_of_moves,i;
    //pupulates array with 100000000 numbers
    populateData(array);

    byTen=1;
    i=0;
    while(i<8){
        byTen*=10;
        target = generateRandomTarget(byTen);
        EndIndex = byTen -1;
        targetIndex = ternarySearch(startIndex, EndIndex , target,
                                    array,&number_of_moves);
        number_of_iterations[i] = number_of_moves;

        printf("EndIndex: %d\n",EndIndex+1);
        targetIndex != -1 ?
        printf("Target %d found at index: %d with %d iterations\n\n",
                target, targetIndex,number_of_iterations[i])
        : printf("Target %d was not found with %d iterations\n\n",
                target,number_of_iterations[i]);

        i++;
    }
    printf("\n\n\nnumber of iteration plot:\n");
    plot(number_of_iterations);

    return 0;
}
```

Screenshots:

```
number of moves: 2
EndIndex: 10
Target 6 found at index: 5 with 2 iterations

number of moves: 4
EndIndex: 100
Target 98 found at index: 97 with 4 iterations

number of moves: 7
EndIndex: 1000
Target 369 found at index: 368 with 7 iterations

number of moves: 8
EndIndex: 10000
Target 4213 found at index: 4212 with 8 iterations

number of moves: 10
EndIndex: 100000
Target 20763 found at index: 20762 with 10 iterations

number of moves: 12
EndIndex: 1000000
Target 527419 found at index: 527418 with 12 iterations

number of moves: 15
EndIndex: 10000000
Target 9528085 found at index: 9528084 with 15 iterations

number of moves: 17
EndIndex: 100000000
Target 39528151 found at index: 39528150 with 17 iterations




number of iteration plot:
data_size       number_of_iterations
10              **
100             ****
1000            *******
10000           ********
100000          **********
1000000         ***********
10000000        ***************
100000000       ******************
```

```
number of moves: 1
EndIndex: 10
Target 7 found at index: 6 with 1 iterations

number of moves: 4
EndIndex: 100
Target 59 found at index: 58 with 4 iterations

number of moves: 6
EndIndex: 1000
Target 216 found at index: 215 with 6 iterations

number of moves: 9
EndIndex: 10000
Target 8573 found at index: 8572 with 9 iterations

number of moves: 10
EndIndex: 100000
Target 75555 found at index: 75554 with 10 iterations

number of moves: 12
EndIndex: 1000000
Target 80253 found at index: 80252 with 12 iterations

number of moves: 14
EndIndex: 10000000
Target 2080723 found at index: 2080722 with 14 iterations

number of moves: 15
EndIndex: 100000000
Target 22080770 found at index: 22080769 with 15 iterations




number of iteration plot:
data_size       number_of_iterations
10              *
100             ****
1000            ******
10000           *********
100000          **********
1000000         ************
10000000        **************
100000000       ***************
```

```
number of moves: 2
EndIndex: 10
Target 8 found at index: 7 with 2 iterations

number of moves: 4
EndIndex: 100
Target 68 found at index: 67 with 4 iterations

number of moves: 6
EndIndex: 1000
Target 580 found at index: 579 with 6 iterations

number of moves: 9
EndIndex: 10000
Target 6846 found at index: 6845 with 9 iterations

number of moves: 11
EndIndex: 100000
Target 11426 found at index: 11425 with 11 iterations

number of moves: 12
EndIndex: 1000000
Target 516885 found at index: 516884 with 12 iterations

number of moves: 12
EndIndex: 10000000
Target 6517431 found at index: 6517430 with 12 iterations

number of moves: 17
EndIndex: 100000000
Target 6517485 found at index: 6517484 with 17 iterations




number of iteration plot:
data_size       number_of_iterations
10              **
100             ****
1000            ******
10000           *********
100000          ***********
1000000         ************
10000000        ************
100000000       ******************
```

YouTube Video Link:

https://www.youtube.com/watch?v=DdtaQgEseKs

References:

[1] Arora, Nitin, Mamta Martolia Arora, and Esha Arora. "A Novel Ternary Search Algorithm." International Journal of Computer Applications 144.11 (2016).

[2] M. S. Bajwa, A. P. Agarwal and S. Manchanda, "Ternary search algorithm: Improvement of binary search," 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), 2015, pp. 1723-1725.

[3] https://www.geeksforgeeks.org/binary-search-preferred-ternary-search/

[4] https://www.codementor.io/@svenkatadileepkumar/ternary-search-algorithm-explained-with-example-1amtce20ao

[5] https://slaystudy.com/ternary-search/