CS627 - IP2 - Kirsten Reid - 5/30/2019

# 

[**Background**](#_n3j4cgagt9l2) **2**

[Function Flow:](#_tuowazhq108g) 2

[Pseudocode:](#_3ohe7kc76jct) 3

[**Actual Code:**](#_9udd26ub5fwb) **4**

[**Analysis of Time Complexity:**](#_j7j05miq9ywj) **4**

[**Conclusion:**](#_2cz1ze5m1zn) **4**

[**References:**](#_msklb5oolum7) **4**

# 

# Background

Upon initial inspection, the elements for this problem are broken into two parts:

1. How to break the input into 3 equal(ish) parts, and
2. Where to perform the comparison

Once these functional pieces were in place, there was the matter of the recursion. This means that within the search and compare function itself, there should be an embedded call to the function again. This naturally gave rise that after each array is in thirds, to **re-break** into thirds again. At a super high level, the function flow is as follows.

## Function Flow:

Input string array[length = n], variable string x

Break :

Part 1 = 0-n/3

Part 2 = (n/3 + 1) - 2n/3

Part 3 = (2n/3+1) - (n-1)

Is x < part 1?

Yes - break part 1 into 3 and repeat

No - is x > part 3?

Yes - break part 3 into 3 and repeat

No - break part 2 into 3 and repeat

When part is 3 or less, and x is not found, stop and return -1.

However, this isn’t recursive on it’s own. We need to variablize the left and right bounds, and use those index values back into the break function.

So instead of

Part 1 = 0 → n/3,

It should be:

**Part 1 = left bound → width remaining/3, or left bound to (right minus left)/3.**

Two points to add are:

* A counting function is needed to divide and find the boundaries.
* The for loop needs to end when there are no more ways to split what’s left in the input array (or total remaining index size n < 3)

A complication arose when I started thinking about: what if n was not divisible by 3? I started creating a series of mods to differentiate between array counts that were a multiple of 3,

and what i should do if it was NOT divisible by 3.

For example:

If nmod3 = 0, then BREAK into X pattern,

If nmod3 = 1, then BREAK into Y pattern,

If nmod3 = 2, then BREAK into Z pattern,

But, we don't need to calculate the actual third, because whatever it is, the index breaking through a simple division case would handle it. We don’t want to add an entire looping operation each time we want to recursively break the input down. A simple C++ test shows that just using n/3 would be sufficient:

#include <iostream>

using namespace std;

int main()

{

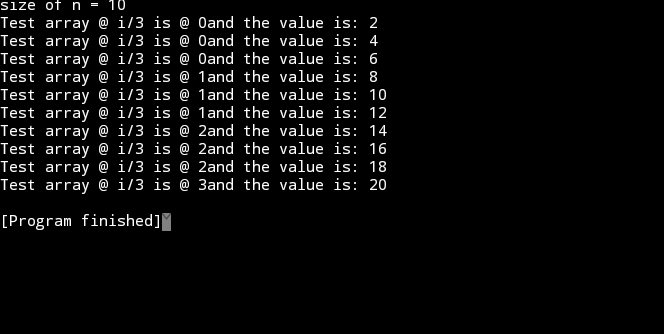
int testArray[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

int n = sizeof(testArray)/sizeof(testArray[0]);

cout << "size of n = "<< n <<"\n";

for (int i = 0; i < n; i++){

cout << "Test array @ i/3 is @ " << (i/3) << "and the value is: " << testArray[i] << "\n" ;}}



Even though the last index is position 3 (which makes us have FOUR partitions, that’s ok because our upper end comparator asks if the value is ABOVE the two thirds mark, or index i/3 >= 2, so we can discount it.

Finally, a true Pseudocode can be developed:

## Pseudocode:

String array [length]

Input search string x

If r >= 1 // when there are at least 3 values to work with

Int firstthird = left + ((right - left)/3);

Int secondthird = right - ((right - left)/3);

If x = array[firstthird], return first third

If x = array[secondthird], return second third

If x < array[firstthird]

Return [recursive function where leftbound stays left, end is firstthird bound]

If x > array[secondthird]

Return [recursive function where rightbound is secondthird bound, right bound stays right]

Else

Return [recursive function between firsttthird and secondthird]

Return -1 //if length <3 and cannot be broken more

# Actual Code:

Please see:

1. Integerternarysearch.cpp
2. stringternarysearch.cpp

# Analysis of Time Complexity:

This method of dividing and conquering is an altered case from the traditional **binary search**, which breaks an argument into a binary tree, in half, so that it can break out each function on one half of a tree or the other, then re-make the tree. It seems logical that a **ternary tree**, or searching thirds rather than halves, would be more effective. The pseudocode for each is broken out:

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary** | **Time Complexity** | **Ternary** | **Time Complexity** |
| Array [x] | O(1) | Array [x] | O(1) |
| Count length | O(1) | Count length | O(1) |
| For X>1 | O(1) | For X>2 | O(1) |
| Half = x/2 | O(1) | Third 1 = x/3 | O(1) |
|  |  | Third 2 = 2x/3 | O(1) |
| Is X half? | O(1) | Is X third 1? | O(1) |
|  |  | Is X third 2? | O(1) |
| Is X > half? | O(1) | Is X < third 1? | O(1) |
| Yes? Break apart top half | RECURSIVE | Yes? Break apart first third | RECURSIVE |
|  |  | Is x < third 2? | O(1) |
|  |  | Yes? Break apart last third | RECURSIVE |
| else |  | else |  |
| Break apart bottom half | RECURSIVE | Break apart middle third | RECURSIVE |
| Array only has one piece left |  | Array only has two pieces left |  |
| Return -1 | O(1) | Return -1 | O(1) |
|  |  |  |  |

We can see just from the number of steps that even though the recursion only executes once per loop for either binary or ternary, the number of steps each loop is actually **more** with the ternary algorithm!

Mathematically, the equations for each algorithm are: (taken from Anmol, n.d.)

Binary: T(n) = T(n/2) + 2, T(1) = 1

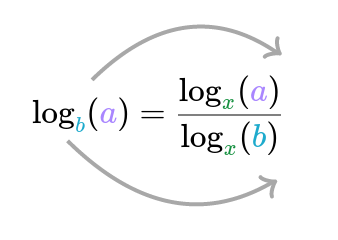
Ternary: T(n) = T(n/3) + 4, T(1) = 1

The number of comparisons is then:

Binary O(n) = 2c\*log2n + O(1)

Ternary O(n) = 4c\*log3n + O(1)

Using algorithmic rules to change bases: (Khan Academy, n.d.)



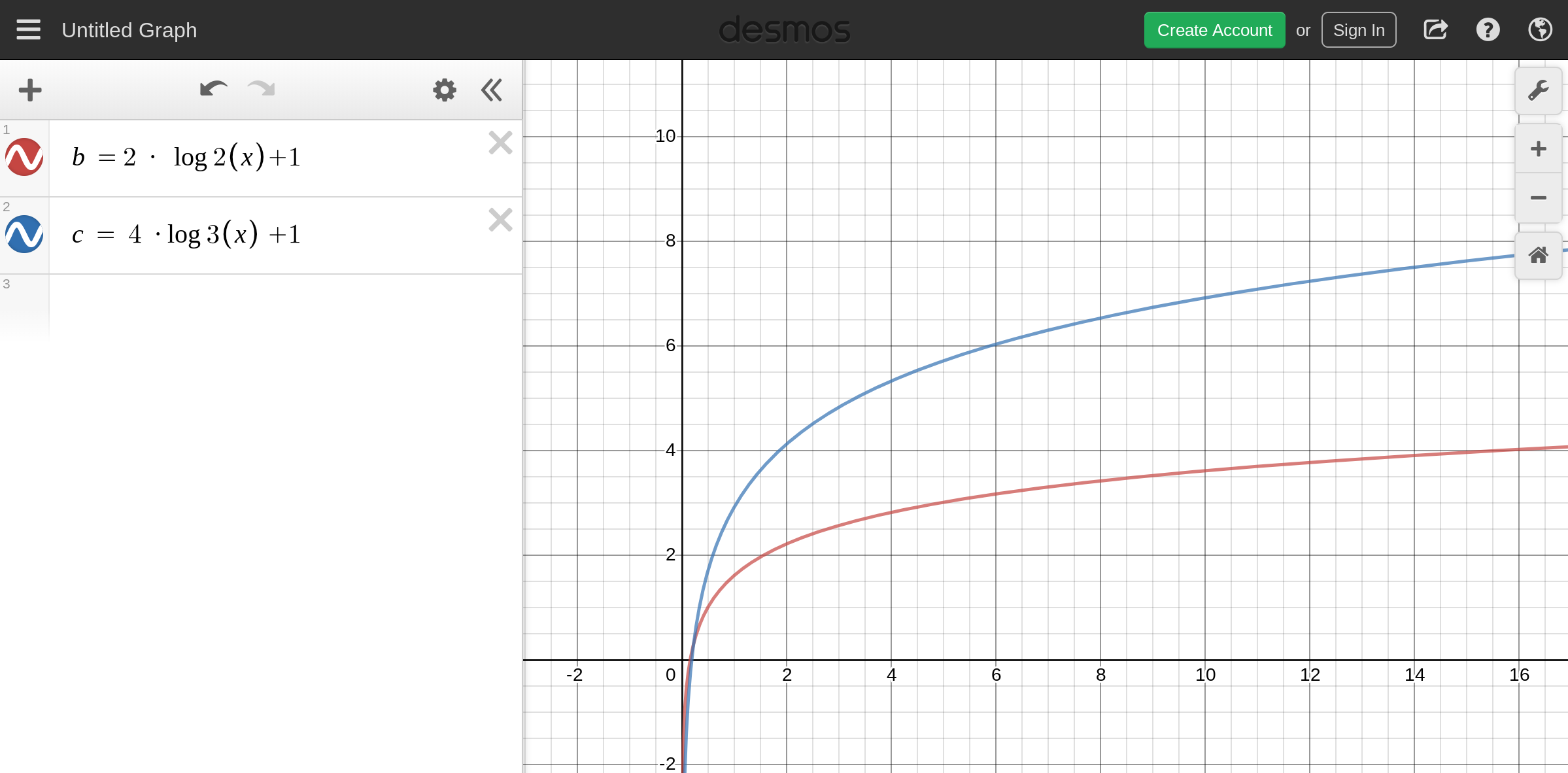
So for ternary:

4c log3 (n) = 4c \* (log2 (n) / log2 (3)) = 4c \* (1/ log2 (3)) \* log2 (n).

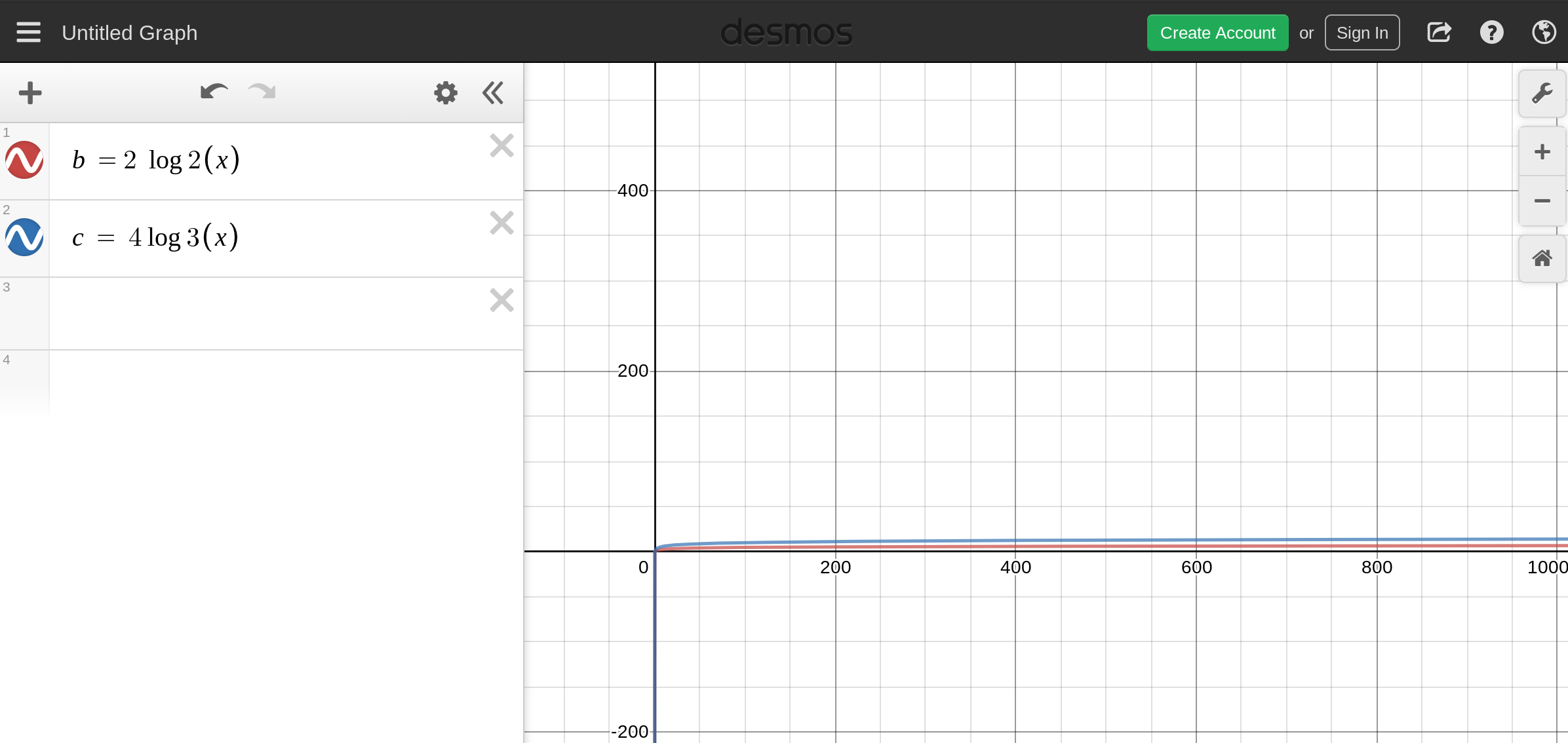
4c (1/ log2 (3)) log2 (n) is greater than 2c \* log2 (n), it is actually **more** comparisons in the worst case for ternary over binary.

A plotted graph shows the differences:

For small n:



For larger n:



We can see that if we truly treat the input as an infinite series, that the difference between log 2 and log 3 is virtually indistinguishable. Only the shape of the curves matter (logarithmic curve), and how fast or slow has little effect for large volumes traditionally seen in programs.

So why was a ternary search even developed? The simple answer is: when you need something broken into thirds, like finding a min or max of a unimodal function. There are also cases of *modified* triple search functions that improve upon the efficiency of ternary *and* binary search functions.

# Conclusion:

Although ternary seems like a better choice, binary is actually slightly better than the ternary search, most significantly in small arrays with large computing complexity time constants. There are a number of improvements that can be made, and being able to split a search into k-nary equal searching realms is a critical need of divide and conquer applications.

# References:

1. Techie Delight (n.d.) *Ternary Search vs Binary Search*. Retrieved from <https://www.techiedelight.com/ternary-search-vs-binary-search/>
2. Rai, A. (n.d.) *Ternary Search*. Geeks for Geeks. Retrieved from <https://www.geeksforgeeks.org/ternary-search/>
3. <https://www.ijcaonline.org/archives/volume181/number8/gupta-2018-ijca-917630.pdf>
4. Garg, P. (n.d.) *Ternary Search*. HackerEarth. Retrieved from <https://www.hackerearth.com/practice/algorithms/searching/ternary-search/tutorial/>
5. Shamim, A. (2017, Apr 30) *Ternary Search*. GitHub. Retrieved from <https://github.com/Algorithm-archive/Learn-Data_Structure-Algorithm-by-PHP/tree/master/Searching/Ternary%20Search>
6. Cave of Programming (n.d.) *Working with Arrays in C++*. Retrieved from <https://www.caveofprogramming.com/c-plus-plus-tutorial/c-array-working-with-arrays-in-c.html>
7. Anmol (n.d.) *Why is Binary Search Preferred over Ternary Search?* Retrieved from <https://www.geeksforgeeks.org/binary-search-preferred-ternary-search/>
8. Khan Academy (n.d.) *Logarithm Change of Base Rule*. Retrieved from <https://www.khanacademy.org/math/algebra2/exponential-and-logarithmic-functions/change-of-base-formula-for-logarithms/a/logarithm-change-of-base-rule-intro>