Binary Search Application In Exploring and Calculating n^th Root Value

Harry Setiawan Hamjaya/13516079
Program Studi Teknik Informatika
Institut Teknologi Bandung
Bandung, Indonesia
hamjaya_harry@yahoo.co.id

***Abstract*—Mathematical computational can't be denied from not only Informatics Engineering but also from Computer Science study program. In mathematical computational, sometimes we didn't have an exact value such what were expected, that's why we need to calculate and make some formula for the worst case what we could have from the computation. Some of the mathematical computational we sometimes found is about calculate the value of the square root and cubic root which we can done easily by our hand. But for bigger root of course we need a calculator or we can use the computer to help us to count and find out the value. In this paper, I would like to investigate and analyze binary search approaches for exploring the method, and also counting the value of a nth root of a decimal value.**

Keywords— Mathmematical; computational; square; cubic; decimal; root;

I. INTRODUCTION (*HEADING 1*)

Root, in mathematics, a solution to an equation, usually expressed as a number or an algebraic formula. There are two categories of a root in Mathematics. The first one is the root of a number. The root of a number x is another number, which when multiplied by itself a given number of times equals x. The other one is the root of a polynomial are the values of the variable that cause the polynomial to evaluate to zero. In this paper, I would like to explain more about the root of a number which can be vary various

About the history why we are using "*root*", In the 9th century, Arab writers usually called one of the equal factors of a number *jadhr* (or in English we pronounce it "*root*"), and their medieval European translators used the Latin word *radix* (from which derives the adjective *radical*). If a is a positive real number and n a positive integer, there exists a unique positive real number x such that $x^n = a$. This number is the (principal) n^{th} root of a, now days, we have it written in some forms such as:

$$\sqrt[n]{a} \text{ or } a^{\frac{1}{n}}$$

The integer n is called the index of the root. For $n = 2$, the root is called the square root and is written $\sqrt[2]{a}$, and for $n = 3$, the root is called the cube root and is written $\sqrt[3]{a}$. If a is negative and n is odd, the unique negative nth root of a is termed principal. For example, the principal cube root of – 27 is – 3.

If a whole number (positive integer) has a rational n^th root—i.e., one that can be written as a common fraction—then this root must be an integer. Thus, 5 has no rational square root because 2^2 is less than 5 and 3^2 is greater than 5. Exactly n complex numbers satisfy the equation $x^n = 1$, and they are called the complex n^th roots of unity. If a regular polygon of n sides is inscribed in a unit circle centred at the origin so that one vertex lies on the positive half of the x-axis, the radii to the vertices are the vectors representing the n complex nth roots of unity. If the root whose vector makes the smallest positive angle with the positive direction of the x-axis is denoted by the Greek letter omega, ω, then $\omega, \omega^2, \omega^3, \omega^4, \ldots, \omega^n = 1$ constitute all the nth roots of unity. For example,

$$\omega = -\frac{1}{2} + \sqrt{\frac{-3}{2}}$$

$$\omega^2 = -\frac{1}{2} - \sqrt{\frac{-3}{2}}$$

$$\omega^3 = 1$$

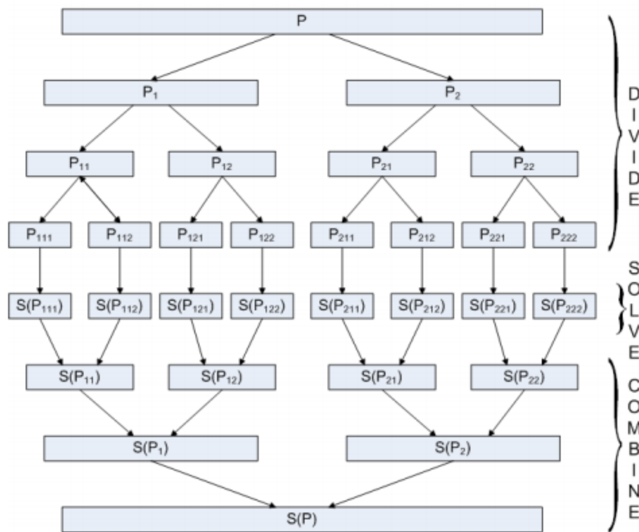
are all the cube roots of unity. Any root, symbolized by the Greek letter epsilon, ε, that has the property that $\varepsilon, \varepsilon^2, \varepsilon^3 \ldots, \varepsilon^n = 1$ give all the nth roots of unity is called primitive. Evidently the problem of finding the nth roots of unity is equivalent to the problem of inscribing a regular polygon of n sides in a circle. For every integer n, the nth roots of unity can be determined in terms of the rational numbers by means of rational operations and radicals; but they can be constructed by ruler and compasses (i.e., determined in terms of the ordinary operations of arithmetic and square roots) only if n is a product of distinct prime numbers of the form $2^h + 1$, or 2^k times such a product, or is of the form 2^k. If a is a complex number not 0, the equation $x^n = a$ has exactly n roots, and all the nth roots of a are the products of any one of these roots by the nth roots of unity.

II. DIVIDE AND CONQUER

Divide and Conquer, like Greedy and Dynamic Programming, Divide and Conquer (wellknown as DnC) is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide:* Break the given problem into subproblems of same type.
2. *Conquer:* Recursively solve these subproblems
3. *Combine:* Appropriately combine the answers.

Majority the implementation of this algorithm is applying the recursive method, which is calling the function by it self when still dividing it into subproblems, until the subproblems become trivial and more ease to be solved, Finish solving the subproblems than it will be combined into the bigger problems.



Picture I: Illustration of apply the Divide and Conquer Algorithm[1]

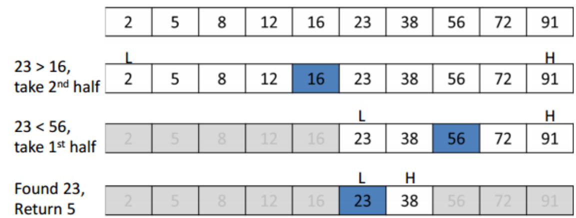
Following are some standard algorithms that are Divide and Conquer Algorithms:

1. Binary Search
2. Quick Sort
3. Merge Sort
4. Closest Pair of Points
5. Strassen's Algorithm
6. Cooley-Tukey Fast Fourier Transform (FFT) Algorithms
7. Karatsuba Algorithm for Fast Multiplication

Since from the title and the explanation in the abstraction, we only will focus discussing about the 1st topic which is ***Binary Search*** in the next chapter.

III. Binary Search

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.



Picture I: Illustration of apply the Divide and Conquer Algorithm[5]

One of the most common ways to use binary search is to find an item in an array. For example, the Tycho-2star catalog contains information about the brightest 2,539,913 stars in our galaxy. Suppose that you want to search the catalog for a particular star, based on the star's name. If the program examined every star in the star catalog in order starting with the first, an algorithm called **linear search**, the computer might have to examine all 2,539,913 stars to find the star you were looking for, in the worst case. If the catalog were sorted alphabetically by star names, **binary search** would not have to examine more than 22 stars, even in the worst case.

When describing an algorithm to a fellow human being, an incomplete description is often good enough. Some details may be left out of a recipe for a cake; the recipe assumes that you know how to open the refrigerator to get the eggs out and that you know how to crack the eggs. People might intuitively know how to fill in the missing details, but computer programs do not. That's why we need to describe computer algorithms completely.

In order to implement an algorithm in a programming language, you will need to understand an algorithm down to the details. What are the inputs to the problem? The outputs? What variables should be created, and what initial values should they have? What intermediate steps should be taken to compute other values and to ultimately compute the output? Do these steps repeat instructions that can be written in simplified form using a loop?

Let's look at how to describe binary search carefully. The main idea of binary search is to keep track of the current range of reasonable guesses. Let's say that I'm thinking of a number between one and 100, just like the guessing game. If you've already guessed 25 and I told you my number was higher, and you've already guessed 81 and I told you my number was lower, then the numbers in the range from 26 to 80 are the only reasonable guesses. Here, the red section of the number line contains the reasonable guesses, and the black section shows the guesses that we've ruled out:

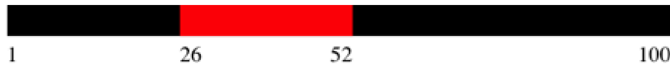
Picture II: The Step and Illustration of apply the Binary Search Algorithm[6]

In each turn, you choose a guess that divides the set of reasonable guesses into two ranges of roughly the same size. If your guess is not correct, then I tell you whether it's too high or too low, and you can eliminate about half of the reasonable

guesses. For example, if the current range of reasonable guesses is 26 to 80, you would guess the halfway point,

$$\frac{(26 + 80)}{2} = 53$$

If I then tell you that 53 is too high, you can eliminate all numbers from 53 to 80, leaving 26 to 52 as the new range of reasonable guesses, halving the size of the range.



Picture III: The Step and Illustration of apply the Binary Search Algorithm[6]

For the guessing game, we can keep track of the set of reasonable guesses using a few variables. Let the variable min be the current minimum reasonable guess for this round, and let the variable max be the current maximum reasonable guess. The input to the problem is the number n, the highest possible number that your opponent is thinking of. We assume that the lowest possible number is one, but it would be easy to modify the algorithm to take the lowest possible number as a second input.

Here's a step-by-step description of using binary search to play the guessing game:

1. Let $min = 1$ and $max = n$.
2. Guess the average of max and min rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set min to be one larger than the guess.
5. If the guess was too high, set max to be one smaller than the guess.
6. Go back to step two.

We could make that description even more precise by clearly describing the inputs and the outputs for the algorithm and by clarifying what we mean by instructions like "guess a number" and "stop." But this is enough detail for now.

We know that linear search on an array of n elements might have to make as many as n guesses. You probably already have an intuitive idea that binary search makes fewer guesses than linear search. You even might have perceived that the difference between the worst-case number of guesses for linear search and binary search becomes more striking as the array length increases. Let's see how to analyze the maximum number of guesses that binary search makes.

The key idea is that when binary search makes an incorrect guess, the portion of the array that contains reasonable guesses is reduced by at least half. If the reasonable portion had 32 elements, then an incorrect guess cuts it down to have at most 16. Binary search halves the size of the reasonable portion upon every incorrect guess.

If we start with an array of length 8, then incorrect guesses reduce the size of the reasonable portion to 4, then 2, and then 1. Once the reasonable portion contains just one element, no further guesses occur; the guess for the 1-element portion is either correct or incorrect, and we're done. So with an array of length 8, binary search needs at most four guesses.

What would happen with an array of 16 elements? If you said that the first guess would eliminate at least 8 elements, so that at most 8 remain, you're getting the picture. So with 16 elements, we need at most five guesses.

By now, you're probably seeing the pattern. Every time we double the size of the array, we need at most one more guess. Suppose we need at most m guesses for an array of length n. Then, for an array of length $2n$ the first guess cuts the reasonable portion of the array down to size n, and at most m guesses finish up, giving us a total of at most $m + 1$ guesses.

Let's look at the general case of an array of length n. We can express the number of guesses, in the worst case, as "the number of times we can repeatedly halve, starting at n, until we get the value 1, plus one." But that's inconvenient to write out.

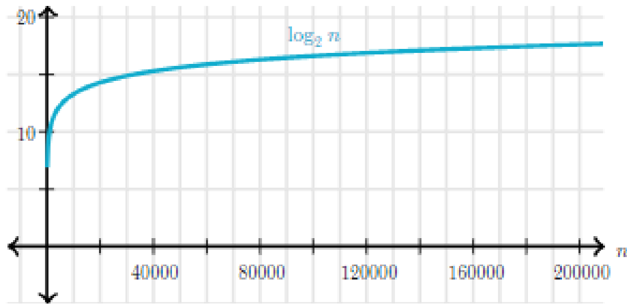
Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at n nn, until we get the value 1: **the base-2 logarithm** of n. That's most often written as $\log_2 n$, but sometimes in computer science we have it written in $\log n$.

Also we can plot a table to compare the number of executions needed when we are comparing a *linear search algorithm* and a *binary search algorithm.*

n	$\log_2 n$
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21

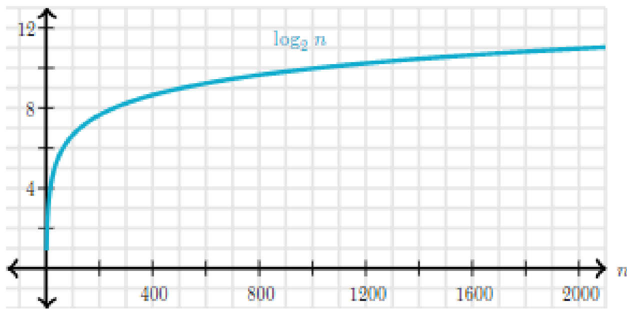
Table I: The Comparison of Linear Search and Binary Search

We can view this same table as a graph:



Picture IV: The Curve for Binary Search Apply [6]

Or zooming in on smaller values of n:



Picture V: The Curve for Binary Search Apply [6]

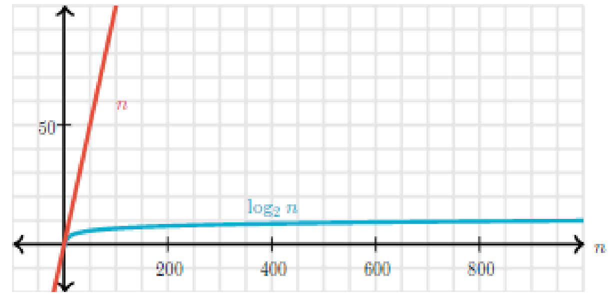
The logarithm function grows very slowly. Logarithms are the inverse of exponentials, which grow very rapidly, so that if$\log_2 n = x$ then $n = 2^x$. For example, because$\log_2 128 = 7$, and we know $2^7 = 128$.

That makes it easy to calculate the runtime of a binary search algorithm on an n that's exactly a power of 2. If n is 128, binary search will require at most 8 $(\log_2 128 + 1)$ guesses.

What if n isn't a power of 2? In that case, we can look at the closest lower power of 2. For an array whose length is 1000, the closest lower power of 2 is 512, which equals 2^9. We can thus estimate that $\log_2 1000$ is a number greater than 9 and less than 10, or use a calculator to see that its about 9.97. Adding one to that yields about 10.97. In the case of a decimal number, we round down to find the actual number of guesses. Therefore, for a 1000-element array, binary search would require at most 10 guesses.

For the Tycho-2 star catalog with 2,539,913 stars, the closest lower power of 2 is 2^{21} (which is 2,097,152), so we would need at most 22 guesses. Much better than linear search!

Another graphic when we need to show the comparison of linear search and binary search is below:



Picture VI: The Comparison between Linear Search and Binary Search [6]

IV. Solving the NTH Root using Binary Search Approach

As Written in the title and described well in the abstract, we will have to analysis, evaluate and calculate the value of the nth root using binary search approach in order to make the computation is fast for all possible decimal value in the root form.

A. *Method Analysis*

Before using the binary search approach in some problems, we have to know how the step of decomposition the problems into small parts such the Divide and Conquer algorithm teach us. At first we have to define an *Error* e in our program, in my experiments I would like to use 0.0000001. Hence we can step to the next step which is the main of our algorithm method for calculating the nth root of a number:

1) *Initialize The border of our searching range.*

   Let says it is Start = 0 and End = n

2) *Calculate the guess which will be tried.*

   In the experiment I used to use a variable

$$mid = \frac{Start + End}{2}$$

3) *Check The mid Value.*

   The next step is checking whether the mid is the answer or we need to do more recursive in case it doesn't satisfy:

$$n - pangkat(p, mid) < e$$

   But if the condition is fulfilled and satisfied, then we just need return the value of mid

4) *The First Condition from 3)*

   If we have such this condition:

$$n > pangkat(p, mid)$$

   Then we need to do a removal border which is:

$$End = mid$$

5) *The Second Condition from 3)*

   If we have such this condition:

$$n < pangkat(p, mid)$$

   Then we need to do a removal border which is:

$$Start = mid$$

6) Repetition

If we have either the first condition or the second condition, then we will need to repeat the recursive steps from step *2)*, until we have the condition where we only need to return the value.

B. Source Code

1. Powers Function

```
double powers(int n, double p)
{
    double res=1;
    for(int i=0; i<n; i++){
        res*=p;
    }
    return res;
}
```

In this function, we have two input parameters which is integer n and double p which return a value of p^n and evaluated using a looping *for*

2. Diff Function

```
double diff(double n,int p, double
mid)
{
    if (n > powers(p,mid))
        return (n-powers(p,mid));
    else
        return (powers(p,mid) - n);
}
```

In this function, we have three input parameters which is double n, integer p, and double mid which return the absolute value of the difference between n and the powers(p,mid) which is mid^p

3. PowerRoot Function

```
double powerRoot(int sqrt, double
n)
{
    double start = 0, end = n;
    double e = 0.0000001;
    while (true)
    {
        double mid = (start +
end)/2;
        double error = diff(n,
sqrt, mid);
        if (error <= e)
            return mid;
        if (powers(sqrt, mid) > n)
            end = mid;
        else
            start = mid;
    }
```

In this function, we have two input parameters which is int $sqrt$ or as known as the root power and the double value n which is needed to be found. Set the border with $start = 0$ and $end = n$ also the precision number $e = 0.0000001$ and then do the iterations while the different between n and mid is greater than e then repeat the iteration, while if the different less than e then we will return the value mid.

4. Main Program

```
int main()
{
    double n, p;
    cout<<"The Base Root:";
    cin>>n;
    cout<<"\nThe Main Number:";
    cin>>p;
    auto start =
chrono::steady_clock::now();
    cout<<"\nThe Result Is
"<<powerRoot(n,p);
    auto end =
chrono::steady_clock::now();
    auto diff = end - start;
    cout << "\nTime Execution in
NanoSeconds: "<<chrono::duration
<double, nano> (diff).count() << "
ns" << endl;
    return 0;
}
```

The Main Program, ask for the base root operation and the main number, and then call the powerRoot function and then output the value and the time execution.

C. *Program Experiments*

1. $\sqrt[697]{345987}$

```
================ READY ================
Makalah
Makalah
Process started >>>
The Base Root:697

The Main Number:345987

The Result Is 1.01847
Time Execution in NanoSeconds: 6.066e+006 ns
<<< Process finished. (Exit code 0)
```

Picture VII: Program Test

Picture VIII: Online Calculator Scientific

2. $\sqrt[20]{96847312}$

```
================ READY ================
Makalah
Makalah
Process started >>>
The Base Root:20

The Main Number:96847312

The Result Is 2.50787
Time Execution in NanoSeconds: 0 ns
<<< Process finished. (Exit code 0)
```

Picture IX: Program Test

Picture X: Online Calculator Scientific

3. $\sqrt[5]{992436543}$

```
================ READY ================
Makalah
Makalah
Process started >>>
The Base Root:5

The Main Number:992436543

The Result Is 63
Time Execution in NanoSeconds: 0 ns
<<< Process finished. (Exit code 0)
```

Picture XI: Program Test

Picture XII: Online Calculator Scientific

D. *Conclusion*

In the picture above, we could see that the value given by the program is exactly the same as the calculations of the online scientific calculator in three experiments with various base and very big main numbers. The execution time too is very small since it is printed in nanoseconds.

From this we can conclude that to make n^{th} root solver such in many scientific calculator we could use one of the most powerful and fastest searching algorithm which is known by ***Binary Search Algorithm***, since not only accurate for the decimal form it is also very fast since it solved it in nanoseconds.

ACKNOWLEDGMENT

The author wants to say thank you as big to God Almighty, thanks to his mercy, the author could finish this paper. Also to the lecturer of Design Algorithm Subjects in ITB that has been guiding for one semester so I could get a lot of knowledge from this lecture, that is Mrs. Masayu Leylia Khodra. Also everyone which has assited to the process of completion of this paper such that the author could finish this paper on time.

REFERENCES

[1] R. Munir, "Algoritma Divide and Conquer" retrieved 10 May 2018 from http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf

[2] R. Munir, "Algoritma Divide and Conquer" retrieved 10 May 2018 from http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf

[3] https://www.mathopenref.com/root.html accessed in 10 May 2018

[4] https://www.britannica.com/science/root-mathematics accessed in 10 May 2018

[5] https://www.geeksforgeeks.org/divide-and-conquer-introduction/ accessed in 10 May 2018

[6] https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search accessed in 11 May 2018

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Mei 2018

Ttd

Harry Setiawan Hamjaya / 13516079