EE4483 – AI and Data Mining

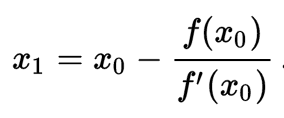
Name: Lim Jia Song John

Matriculation Number: U1321554J

# 1a) Presenting two algorithms

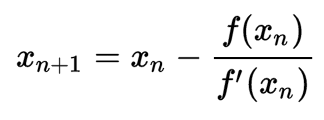
## Newton Raphson Algorithm:

The method starts with a function *f* defined over the [real numbers](https://en.wikipedia.org/wiki/Real_number) *x*, the function's [derivative](https://en.wikipedia.org/wiki/Derivative) *f'*, and an initial guess *x*0 for a root of the function *f*. If the function satisfies the assumptions made in the derivation of the formula and the initial guess is close, then a better approximation *x*1 is



Geometrically, (x1, 0) is the intersection of the x-axis and the [tangent](https://en.wikipedia.org/wiki/Tangent) of the graph f at (x0, f(x0)).

{\displaystyle x\_{1}=x\_{0}-{\frac {f(x\_{0})}{f'(x\_{0})}}\,.}



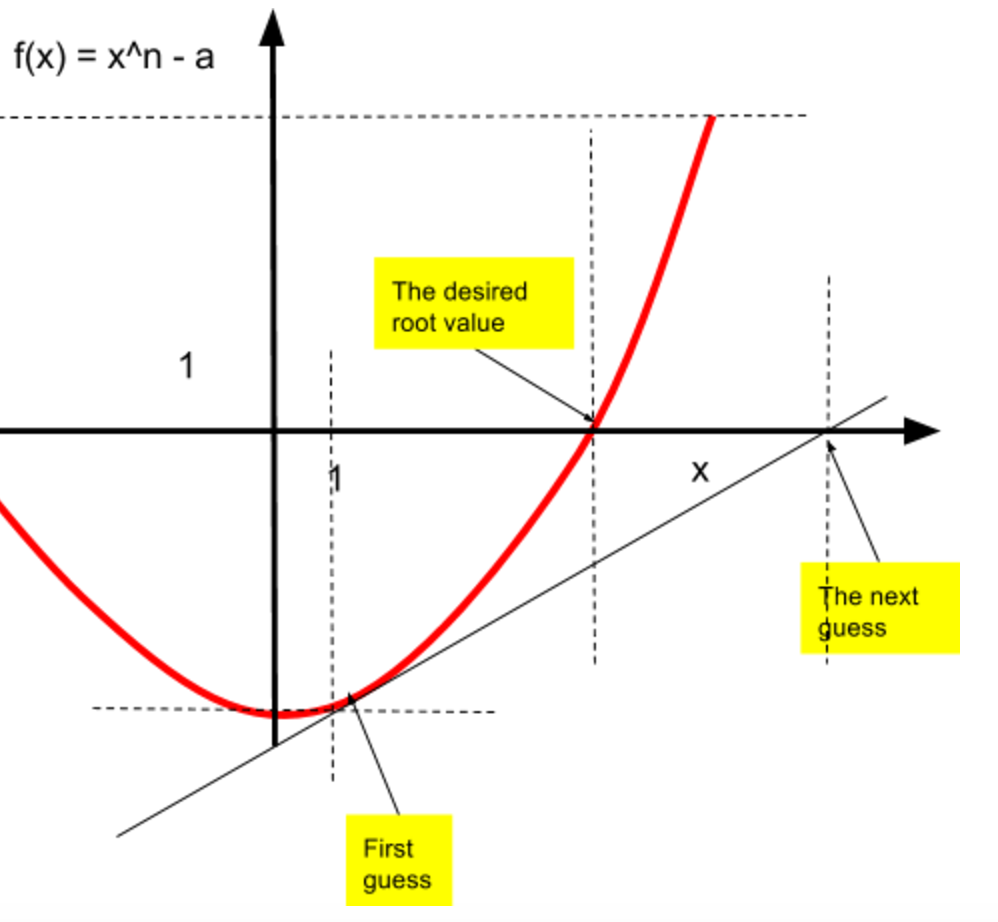
For this method to work, an initial arbitrary value is taken. This method is then recursively called to find better approximations of the root. With each iteration, the approximated root value becomes closer and closer to the actual root value. Once the difference between each successive root operation becomes a very small value, we can approximate that to be the nth root value of the target number.

The Newton Raphson is a algorithm general enough to find the root of any type of function. In this case, we are finding the root of x to the power of 1/n.

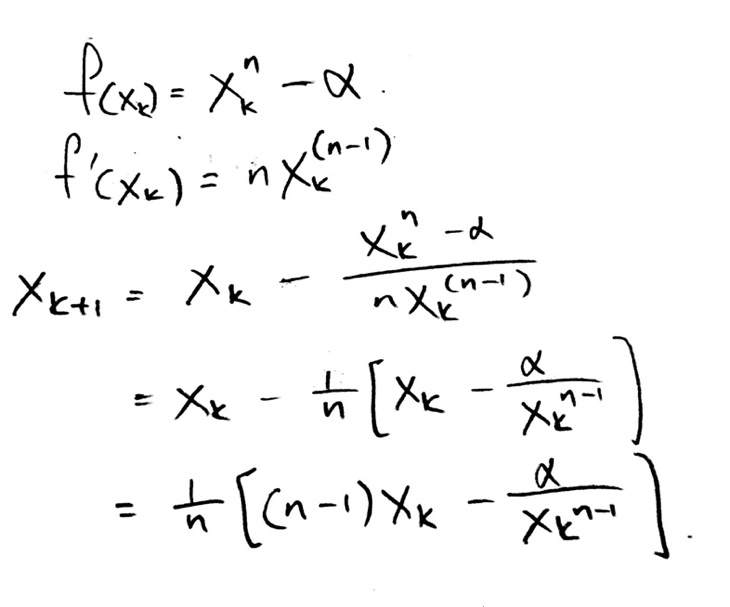
Applying the Newton Raphson to our specific problem, the function f(x) = x1/n has the following form:

F(x) = xn + a, where ‘a’ is the target number, and ‘x’ is the nth root of that number, and f(x) is 0.

For instance, in the case of 641/3, a is 64, and x is 4, and n is 3. The solution to this problem is: 0 = 43 - 64

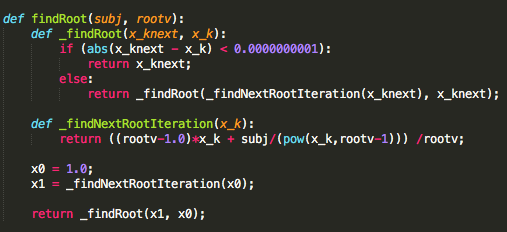


From the Newton Raphson, we can derive the following expression to calculate the nth guess of the root value.

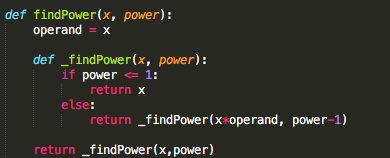


The root is found when subsequent guesses are almost the same value. A threshold number for the difference between subsequent guesses has to be set. If the difference between subsequent iterations is less than this threshold, then we can return the number and conclude that the returned value is a very close approximation of the root value of the target number.

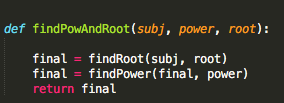
The above algorithm was then written in code:



The above algorithm can be used to find x1/n . xm/n is equivalent to xm \* x1/n . Hence another recursive function was written to find xm.



The complete solution to the problem x simply involves putting the two algorithms together as follows:



## Binary Search Algorithm:

The binary search algorithm is a half interval search that has a logarithmic complexity.

When applied to find the root of a number.

The algorithm takes in four parameters: the first number, the last number, the target number and the root value.

*mid = (first + last)/2*

The algorithm performs half interval searches between the first number and the last number to try to find the root. The target number is ‘separated’ into two halves.

The lower half has the range: first ≤ number ≤ mid.

The upper half has the range: mid ≤ number ≤ last

*if (x < c^n) :*

*return BinSearch(a,c,x,n)*

*else:*

*return BinSearch(c,b,x,n)*

Once the middle number has been found, a test is performed to see if it is indeed the desired root value. This is done by taking the value of the nth power of the half interval.

The function calls itself recursively until the first number and the last number are equivalent. However, it recursively calls itself with the first and last number being constantly updated. If the nth power of the half interval is more than the target number, the lower half of the target number is searched. Or else, the upper half of the target number is searched.

Once the first and the last number are equal, that number returned is the desired root value.

The initial ‘first’ and ‘last’ number can be set as 1 and the target number respectively. However, in order to optimize the search algorithm so that fewer recursive searches are called, a function to get the range is utilized.

*def GetRange(x,n):*

*y=1*

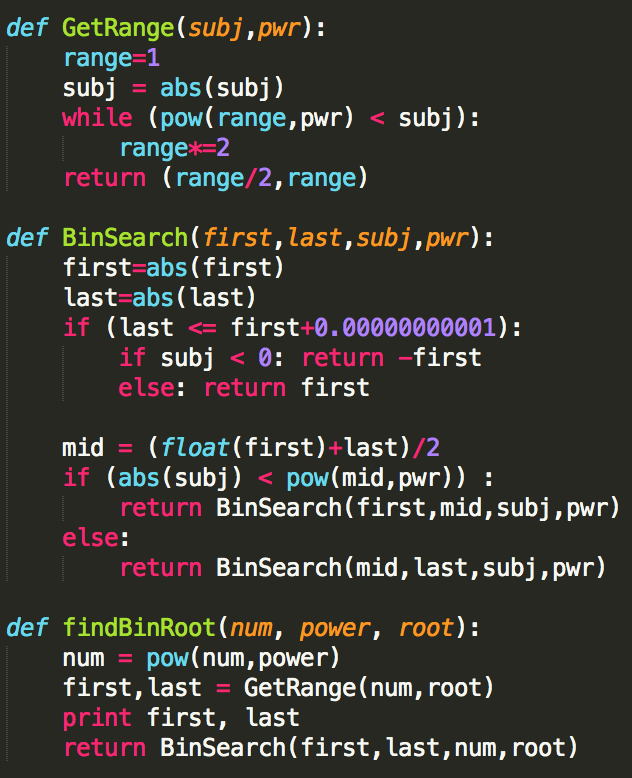
*while (y^2 < x):*

*y\*=2*

*return (y/2, y)*

How this algorithm works is that it narrows down the search range for possible root values. It does this by returning the integer value whose nth power is the smallest possible number above the target number.

The complete code is as follows:



# 1b) Proving the validity of the two algorithms

To show the validity of the algorithms, all we need to show that for a given data set, the algorithm converges to a certain limit and then returns that value. Both algorithms are recursive, and with each recursion, the size of the dataset processed is reduced.

## Newton Raphson

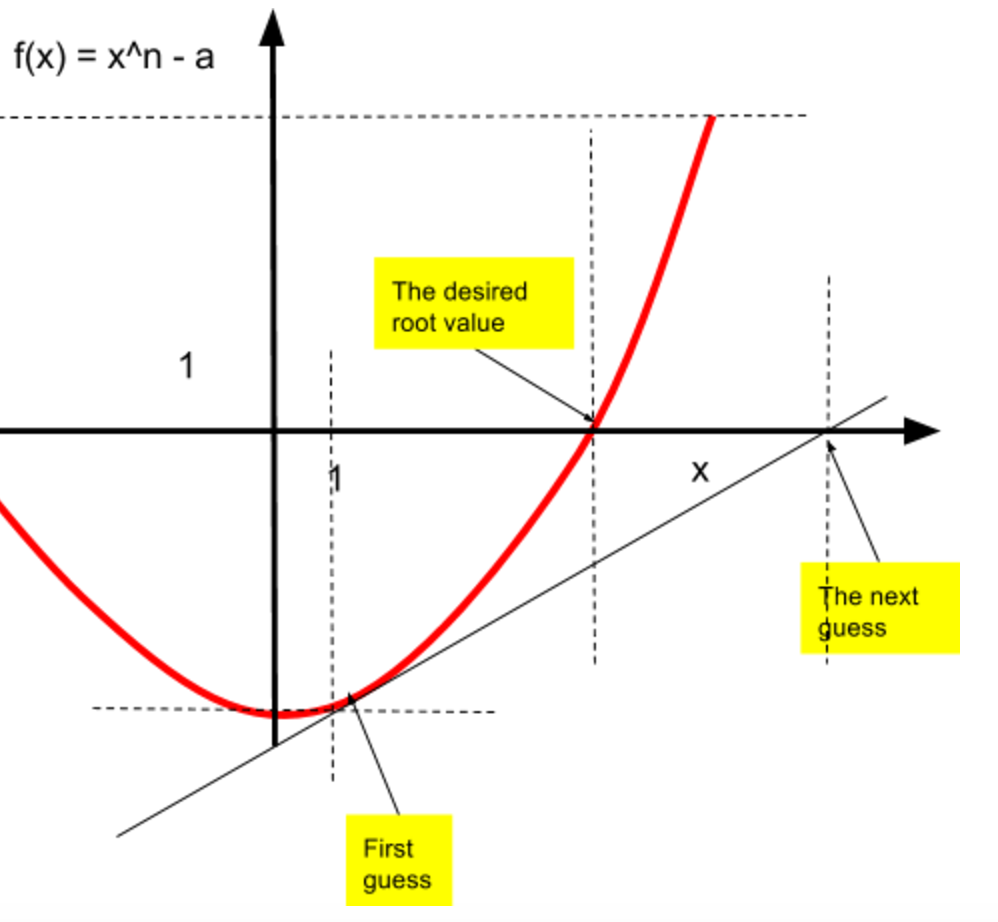
The newton Raphson method usually converges to a root for most cases, but not all.

Consider the following function:

In such a function, the red line represents a point on the function whose gradient is exactly 0, in which it would never intersect the x-axis for a solution to be found. In such a case, there would not be a solution to the problem.

The Newton Raphson is an algorithm that can be applied to find the root of a wide set of problems. In this case, we use it to find the solution of a number to the power of root ‘n’.

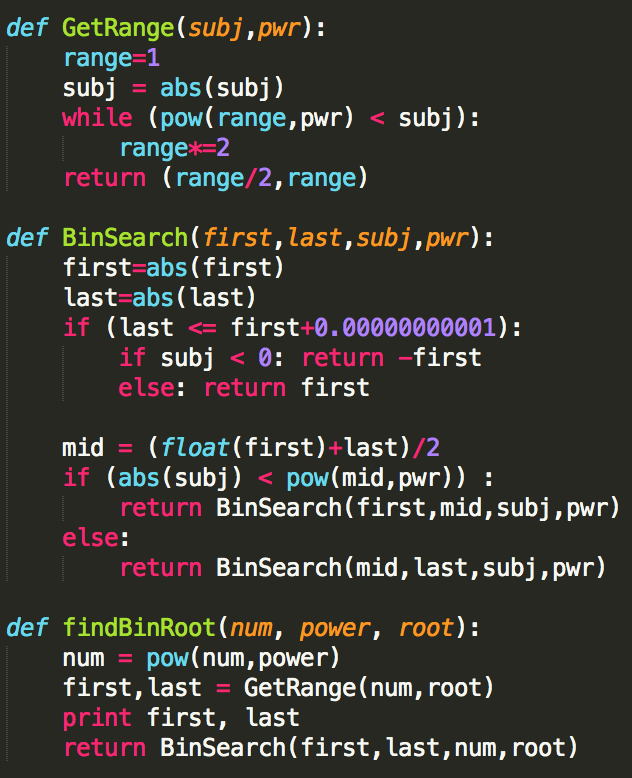
In this case, the Newton Raphson function looks something like the following:



Regardless of the value of ‘n’, f(x) always intersects the y-axis at the value ‘a’, where ‘a’ is the number which the root operation is being performed on. Hence, so long as we choose a non-zero initial value of ‘x’, the gradient at that point will actually become steeper as we approach the solution. Hence, with each iteration, f’(xk) intersects the x-axis closer and closer to the solution to the function. When applied to this case, the Newton Raphson actually ‘accelerates’ to the solution to the function. As such, the Newton Raphson is guaranteed to find a solution to this problem.

## Binary Search Algorithm

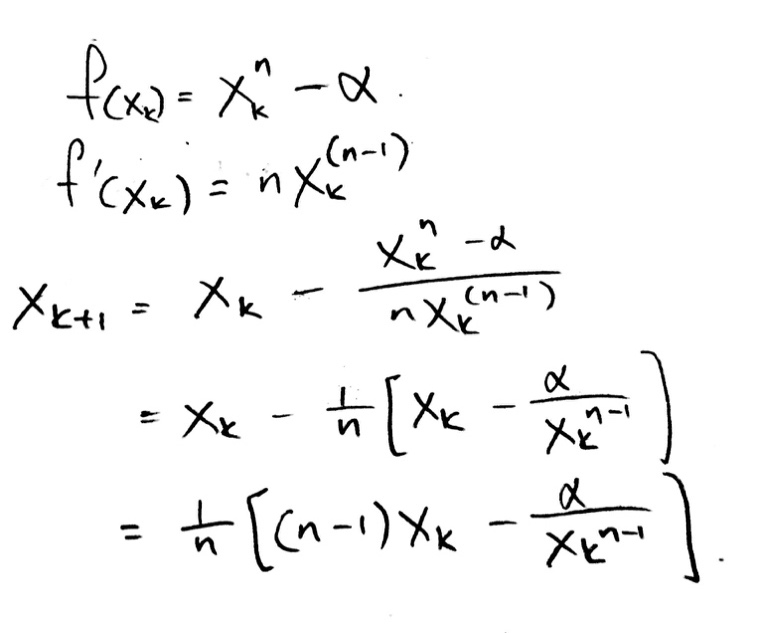
The Binary Search Algorithm is always valid because with each recursive call, the problem set is reduced by exactly half. The algorithm starts out with defining a suitable range of possible solutions to the problem. It is important to select a suitable range in order to get to the right answer. As such, my solution has the method: GetRange(subj, pwr) in order to get that suitable range.



Assuming that the range chosen is suitable, with each successive iteration, the possible solutions to the solution to the problem than before.

# 1c) Comparing the two search algorithms and their complexity

## Newton Raphson



Time complexity of the Newton Raphson is:

O(n) = log(n) \* F(n)

F(n) is the time taken to calculate f(x)/f’(x) with n-digit precision. However, the Newton Raphson has a quadratic rate of convergence, which is to say that it approaches the solution at a quadratic rate.

Therefore, the time complexity is O(log(n))

## Binary Search

Time complexity for the binary search is calculated as follows:

T(n) = worst case time complexity

T(n) = b + T(n/2)

= b + b +T(n/4)

= b + b + b + T(n/8)

= k\*b + T(n/2k)

n/2k = 1 🡪 2k = n 🡪 k = log(n)

= b\*log(n) + T(1)

= b\*(log(n) + 1)

b is the time taken for key comparisons.

Therefore, time complexity is O(log(n))

The problem set is halved with each iteration of the binary search function. Hence the rate of convergence is of a factor 2.

# Conclusion: Comparison and evaluation of both algorithms

Both algorithms have time complexity of O(log(n))

The binary search however, has a smaller speed of convergence as compared to the newton raphson method. As such, the Newton Raphson tends to be the better solution than the Binary Search.