Approximating Annuity using Binary Search Algorithm and Ternary Search Algorithm

Hokki Suwanda - 13519143
Major in Informatics Engineering
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganesha Street no.10, Bandung City
13519143@std.stei.itb.ac.id

***Abstract*—Annuity is very frequent in our lives. Many has tried determining the value of annuity, even a formula is generated. However, the formula is quite hard to remember and derive. Using decrease and conquer, specifically binary search algorithm and ternary search algorithm, we will approximate annuity value.**

Keywords—annuity; decrease and conquer; binary search algorithm; ternary search algorithm

I. Introduction

As humans grow, humans will be interacting with money more frequently. Money is very important in our lives. However, keep in mind that **money is not everything nor money can buy everything**. They will save money, make money, and use money. Some are very good at making money that they have so much money. Some are very bad at saving money that they have almost no money.

Those good at making money need to save their money somewhere other than their own house. In our current era, banks exist for people to save their money. Using bank services, however, is not free, it costs money. The fee is usually paid monthly at a constant interval. Some even invest their money to make more money out of it.

Those bad at saving money will try to loan some money from another party, mainly official financial companies, especially banks. Those who loan from official financial companies will need to pay the amount they loan back to the loaner routinely. The paid amount for every payment is constant.

Year by year, medical fee is always growing. This condition is very bad, considering the frailness of us humans. Due to this, some worried not having money to pay the medical fee. This leads to people using health insurance. However, using health insurance also costs money, which we will pay regularly until the period is over.

Those forms of routine constant payment are annuity. In fact, annuity has so many benefits. Its benefits are not limited to future preparation, living benefit. Yet, keep in mind that owning an annuity may instead cost more money.

II. Base Theory

A. Annuity

Annuity is a series of payments in constant intervals. In banking theory, annuity is routine payment done by one party to another party such as banks and insurance companies. There are many cases of annuity in our world, insurance payments being a case. Other examples of the cases are pension annuity and home mortgage.

The purchaser gives series of contributions in an interval. Meanwhile, the insurer is then obligated to make periodic payments to the purchaser at the future, depending on the type of annuity being used.

In financial mathematics, annuity consists of two, installment fee and interest. Financial mathematics provides a formula to determine credit annuity, which we will derive. Let a_i be the installment fee for i-th payment and b_i be the interest for i-th payment. For n payment times, equation (1) is satisfied.

$$A = a_1 + b_1 = a_2 + b_2 = ... = a_n + b_n \quad (1)$$

where A is the credit annuity.

If we loaned M with an interest rate of m, for i-th payment, both equations (2) and (3) are satisfied.

$$a_1 + a_2 + a_3 + ... + a_n = M \quad (2)$$

$$a_i = a_1 (1 + m)^{n - 1} \quad (3)$$

Equation (2) shows that sum of all installment fee is the loaned money M. Which means, for i-th payment, a_i will be subtracted from M. Interest fee b_i is the product of interest rate and the rest of unpaid loaned money. Which means

$$b_1 = Mm \quad (4)$$

because none of M has been paid. Geometric series formula gives

$$a_1((1+m)^n - 1) / m = M. \quad (5)$$

By combining equations (4) and (5), value of A can be counted using equation (6). Equation (6) is the so called annuity formula. However, the formula will not be used as we will approximate A with binary search algorithm and ternary search algorithm.

$$A = Mm / (1 - (1+m)^{-n}) \quad (6)$$

B. Recursion

Recursion is a process where a function, for example *f*, calls another instance of *f*. The recursion "happening" is named *recurrence*, similar to the word "occurrence". Most recursions are easy to implement because they are easy to identify. Recursion is mostly used in *Dynammic Programming* and *Depth First Search Algorithm*.

Recursion consists of two parts, base and recurrece. Base is usually one or more cases where the problem is small enough to solve or where the solution is on the surface, which doesn't require any complex calculations and/or computations. The recurrence is the part where *f* calls another instance of *f*.

The advantage of recursion is that recursion can be used in a scenario where regular looping cannot solve. Another advantage of recursion is, already stated above, easy to implement because the logic is easy to understand. The drawback is that recursion takes a lot of memory space. Systems stores the process on a stack, which grows by time because of recursion.

There is an alternative to recursion, which is by using a data structure called "stack". The reason is simple, the behavior of stack is analogous to simulating recurrence. Calling another instance of *f* means that we push the arguments to the stack. Stack is a data structure that follows the behavior of stacks (of things) in real world. Only the topmost element of a stack is accessible. Think of a stack as deck of cards where we can only see and draw the top card of the deck.

Recursion is not always implemented in the form of functions or procedures. Some recursions are implemented different. For example, using stack as stated above. Some recursions are also implemented in a standard loop with changing parameters. The implementation, however, depends on the problem to be solved. If the problem can be solved by using recursion with standard loop, implementing the recursion with standard loop is very recommended.

C. Decrease and Conquer

Decrease and Conquer is an algorithm design that reduces the size of the problem into more than one subproblems (usually two subproblems). One of the subproblems is then chosen to be processed, usually recursively. It means that most of the time, the same algorithm used to process the problem is also used to process the subproblem. The solution for the chosen subproblem **is** then extended to obtain the solution of the problem. In some old literatures, Decrease and Conquer is categorized in Divide and Conquer.

Divide and Conquer is an algorithm design that splits the size of the problem into more than one subproblems (usually two subproblems). All subproblems is then processed, usually recursively by using the same algorithm on every subproblem. The solution of every subproblems generated is then combined as a solution of the problem. Some examples of divide and conquer problems and algorithms are merge sort algorithm, quick sort algorithm, convex hull, closest pair problem, and multiplication.

There is a major difference between Decrease and Conquer and Divide and Conquer. Decrease and Conquer **reduces** the problem whereas Divide and Conquer **splits** the problem.

There are three variants of Decrease and Conquer based on the problem reduction, which are :

1) Constant Decremental Decrease and Conquer – Reducing the problem size by a constant, usually 1. Some algorithms that belong in this category are insertion sort, depth-first search, breadth-first search, topological sort, and permutation generator algorithm.

2) Factored Decremental Decrease and Conquer – Reducing the size of the problem by a constant factor, usually 2. Some algorithms and problems from this category are binary search, ternary search, fake-coin problems, and russian peasant multiplication problem.

3) Variabled Decremental Decrease and Conquer – Reducing the size of the problem by undetermined size. Some problems and algorithms belonging to this category are interpolation search, euclid's algorithm, and selection problem.

D. Binary Search

Binary Search is Decrease and Conquer with factored decrement of 2. Binary Search has the same characteristics as Decrease and Conquer, which is reducing the problem into two subproblems and choosing one subproblems satisfying the constraints to solve. The word "constraint" is not literal, it means that it is logically satisfied.

Binary Search is a searching technique by repeatedly reducing the problem into half until the problem is small enough to be solved without using any complex computations. If the answer of the problem is found immediately, the algorithm stops. If the answer of the problem is not found, the algorithm continues. This algorithm is usually used to search one out of so many datas inside a program.

The requirement is that the data has to be sorted so the subproblems to be chosen can be determined. If the data has not been sorted, then there is no way to determine the right subproblems to choose because both the left and right half of the problem can contain the solution of the problem. Thus, if the data is sorted, the solution will be on only one side, which can be chosen logically depending on the problem.

Consider a number guessing problem. The program will try to guess the number a player is holding, which is an integer in the interval of 1 to *N*, in *K* tries. If *X* is small, the program can query the number sequentially. However, if *N* is big and *K* is small, binary search algorithm can be used to solve it in a complexity of $O(log_2N)$, with an assumption where value of *K* has to be at least log_2N. The complexity will be proven afterwards. The pseudocode for number guessing problem using binary search is as follows.

```
procedure BinSearch(input l: integer, input r: integer)
{ Guess the number the player is holding by using binary search. Player gives input ">" or "<=" depending whether the number program generated is "greater than" or "less or equal than" the number player is holding }
{ K defined outside }
Declaration:
mid: integer
Algorithm:
if (l < r) and (K > 0) then
   mid ← (l + r) div 2
   { guess mid }
   K ← K - 1
   { player gives input }
   if (input = ">") then
       { the guess is too big }
       BinSearch(l, m - 1)
   else { input = "<=" }
       BinSearch(m + 1, r)
```

By assuming *K* is at least log_2N, the time to process binary search algorithm is

$$T(N) = T(N/2) + 1, \quad (2)$$

where

$$T(1) = 0. \quad (3)$$

Which results in

$$T(N) = log_2N = O(logN). \quad (4)$$

The complexity of binary search algorithm is, in fact, relatively small. Many people prefers using binary search over other searching algorithm. However, there is another searching algorithm with concepts similar to binary search algorithm. It is ternary search algorithm.

E. Ternary Search

Ternary Search is Decrease and Conquer with factored decrement of 3. Ternary Search has the same characteristics as Decrease and Conquer, which is reducing the problem into three subproblems and choosing one subproblems satisfying the constraints to solve. The word "constraint" is not literal, it means that it is logically satisfied.

Ternary Search is a searching technique by repeatedly reducing the problem into one third of original size until the problem is small enough to be solved without using any complex computations. If the answer of the problem is found immediately, the algorithm stops. If the answer of the problem is not found, the algorithm continues. This algorithm is usually used to search one out of so many datas inside a program.

The requirement is that the data has to be sorted so the subproblems to be chosen can be determined. If the data has not been sorted, then there is no way to determine the right subproblems to choose because all three of the left, middle, and right one third of the problem can contain the solution of the problem. Thus, if the data is sorted, the solution will be on only one side, which can be chosen logically depending on the problem.

Consider a reverse number guessing problem. The player will try to guess the number the program is holding, which is an integer in the interval of 1 to *N*, in *K* tries. If *X* is small, the player can query the number sequentially. However, if *N* is big and *K* is small, ternary search algorithm can be used to solve it in a complexity of $O(logN)$, with the assumption where value of *K* has to be at least $2log_3N$. The complexity will be proven afterwards. The pseudocode for reverse number guessing problem from player's side using ternary search algorithm is as follows.

```
procedure ReverseGuess(input l: integer, input r: integer)
{ Player guesses the number N program is holding by using ternary search. Program outputs ">" or "<=" depending whether the number player guessed is "greater than" or "less or equal than" the number program is holding }
Declaration:
mid1, mid2: integer
Algorithm:
if (l < r) then
   mid1 ← l + ((r - l) div 3)
   mid2 ← r - ((r - l) div 3)
   {player guess mid1, program outputs}
   {player guess mid2, program outputs}
   if (output1 = ">") then
       ReverseGuess(l, mid1 - 1)
   else if (output2 = "<=") then
```

```
        ReverseGuess(mid2, r)
    else
        ReverseGuess(mid1, mid2)
```

By assuming K is at least $2log_3N$, the time to process binary search algorithm is

$$T(N) = T(N/3) + 2, \quad (5)$$

where

$$T(1) = 0. \quad (6)$$

Which results in

$$T(N) = 2log_3N = O(logN). \quad (7)$$

The complexity of ternary search algorithm is, in fact, relatively small. However, if compared to complexity of binary search algorithm, ternary search is a bit worse. Time complexity for binary search algorithm is $T(N) = log_2N$ whereas time complexity for ternary search algorithm is $T(N) = 2log_3N$. The little difference in time complexity makes most people prefer using binary search algorithm instead of ternary search algorithm. As the time complexity for binary search algorithm and ternary search algorithm does not differ much, both algorithm are still usable.

III. Determining Annuity

A. Requirements

One of the requirements of binary search and ternary search algorithm being usable is that the data have to be sorted. Annuity satisfies the requirements. Let A be the true amount of monthly annuity to be paid, counted by using the annuity formula. If we try to pay with the value B where $B > A$, after the fee period is over, the credit will be overpaid. The other way around, if we try to pay with the value B where $B < A$, the credit is underpaid. However, if we pay the credit with the value B where $B = A$, the credit is paid the exact amount. Which means, annuity has a sorted areas, which are underpaid, exact, and overpaid. This is shown in the illustration below.

As shown in *Fig. 1.* there consists three areas. The leftmost (the lightest grey) area represents the values of B that causes the credit to be underpaid. The middle (the moderate grey) area represents the area where value of B is "exactly paid". Do note that the "exactly paid" amount becomes an area because there are mathematical limitations for a floating point, thus very small errors exists, and the area is the consequence of neglecting the very small errors. The rightmost (the darkest grey) area represents values of B that causes the credit to be overpaid. By using the area illustrated in *Fig. 1.*, the writer will use Decrease and Conquer Algorithm, especially Binary Search and Ternary Search Algorithm to approximate the value of A using B.

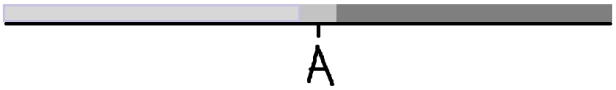


Fig. 1. Sorted area of annuity.

We need to represent these three areas as a number, because computation using number is easier. Let a negative number (–1) represents underpaid. Let 0 be a number that represents "exactly paid". Let 1 be a positive number representing overpaid. With this, we can now fully declare that annuity is sorted. Thus we can then use binary search algorithm and ternary search algorithm.

There are two alternatives of approximating credit annuity using binary search algorithm or ternary search algorithm. The first alternative is analogous to number guessing game and reverse number guessing game. However, it is not simple to be implemented because the number that we should guess is unknown to even the program.

The better alternative to approximate credit annuity is by "testing" the guessed number, as in number guessing game. The test is required to identify which area the guessed number belongs to, either underpaid, exactly paid, or overpaid. The test requires little to no knowledge about annuity. The knowledge we need is in these three equations below.

$$a + a(m + 1) + a(m + 1)^2 + ... + a(m + 1)^{n-1} = M, \quad (8)$$

$$b = Mm, \quad (9)$$

and

$$A = a + b \quad (10)$$

where m is the interest rate, A is the credit annuity, M is the price to be paid, n is the number of times to be paid, a is installment price, and b is interest price.

With three equations above, we can make the "testing" algorithm for the guessed number. Let the guessed number which we use to approximate A be B. The steps of the testing algorithm is as follows:

1) Initiate M_t as M, M_t represents the remaining price to pay.

2) If n is not zero, which means we still have to pay the annuity, subtract $B - b_t$ from M_t. In other words, replace M_t by $M_t(m + 1) - B$. This formula comes from equation (10) and equation (9). Don't forget to subtract 1 from n.

3) Repeat step *2)* until n is 0.

4) Check the value of M_t. If M_t is in the range of error, then B is exactly paid. If M_t is negative, then B is underpaid. Else, B is overpaid.

5) The testing judgement, *exactly paid*, *underpaid*, *overpaid* is then changed into 0, –1, and 1 respectively.

With five steps stated above, we can then propose a testing algorithm to test the value of *B* which we use to approximate the value of *A*. Below is the pseudocode of the testing algorithm.

```
function test(B: real) → integer
{ Returns -1 if B is underpaid, 0 if B is exactly paid, and 1 if B is overpaid. }
{ n, m, M, and maximum error of EPSILON is defined }
Declaration:
i : integer
Mt : real
Algorithm:
Mt ← M
for i ← 1 to n do
   Mt ← ((1 + m) * Mt) – B
if (Mt > EPSILON) then
   { overpaid }
   return 1
else if (Mt < -EPSILON) then
   { underpaid }
   return -1
else
   { exactly paid }
   return 0
```

With the testing algorithm finished, we can now approximate the value of *A* using *B* without knowing its true value.

B. Binary Search

Approximating *A* using *B* by binary search has almost the same step as number guessing game. A very obvious fact is that *A* lies in the interval [0, *M*]. We will then use binary search on the interval. Let's declare every interval in [*l*, *r*]. Let the middle floating point in the interval of [*l*, *r*] be *B*, because we approximate *A* using middle points.

```
function BinSearchApprox(l: real, r: real) → double
{ approximating the credit annuity A by using B, center of interval }
{ test function is defined }
Declaration:
B: real
test_value: integer
Algorithm:
B ← (l + r) / 2
test_value ← test(B)
if (test_value = 1) then
   { overpaid, shift left }
   return BinSearchApprox(l, B)
else if (test_value = -1) then
   { underpaid, shift right }
   return BinSearchApprox(B, r)
else {test_value = 0}
   { the solution is found }
   return B
```

The middle point *B* will then tested using testing algorithm mentioned previously. Depending on the result of the test, *l* or *r* can change. If the result of the test is underpaid, then every value smaller than *B* (left half) is also underpaid, thus *A* lies on the right half of the interval, which is [*B*, *r*]. If the result of the test is overpaid, then every value bigger than *B* (right half) is also overpaid, thus *A* lies on the left half of the interval, which is [*l*, *B*]. If the result is exactly paid, then *B* is the solution. The same algorithm is then applied to the chosen interval. Pseudocode for approximation using binary search algorithm is shown above.

C. Ternary Search

Approximating *A* using *B* by ternary search has almost the same step as reverse number guessing game. A very obvious fact is that *A* lies in the interval [0, *M*]. We will then use ternary search on the interval. Let's declare every interval in [*l*, *r*]. Let the one-third of the interval be *B1* and the two-third of the interval be *B2*.

Both *B1* and *B2* will then be tested. Depending on the result of the test, *l* or *r* can change. If *B2* is underpaid, then every value smaller than *B2* (also *B1*) is also underpaid, thus *A* lies on the right-most subproblem, which is [*B2*, *r*]. If *B1* is overpaid, then every value bigger than *B1* (including *B2*) is also overpaid, thus *A* lies on the left-most subproblem, which is [*l*, *B1*]. If *B1* is underpaid and *B2* is overpaid, then *A* lies between *B1* and *B2*, which means *A* is in [*B1*, *B2*]. If either of *B1* or *B2* is exactly paid, then it is the solution. The same algorithm is then applied to the chosen subproblem. Pseudocode for approximation using ternary search algorithm is shown below.

```
function TerSearchApprox(l: real, r: real) → double
{ approximating the credit annuity A by using B1 and B2, one-third and two-third of interval respectively }
{ test function is defined }
Declaration:
B1, B2: real
test_one, test_two: integer
Algorithm:
B1 ← l + ((r - l) / 3)
B2 ← r - ((r - l) / 3)
test_one ← test(B1)
test_two ← test(B2)
if (test_one = 0) then
   {paying with B1 will be exactly paid}
   return B1
else if (test_two = 0) then
   {paying with B2 will be exactly paid}
   return B2
else if (test_one = 1) then
   {B1 is overpaid, process left-most}
   return TerSearchApprox(l, B1)
else if (test_two = -1) then
   {B2 is underpaid, process right-most}
   return TerSearchApprox(B2, r)
else {test_one = -1 and test_two = 1}
   return TerSearchApprox(B1, B2)
```

VIDEO LINK AT YOUTUBE.

https://youtu.be/Mpz-1KNcpis

ACKNOWLEDGMENT

The writer thanks lecturers of IF2211 Algorithm Strategies who taught and guided the writer on weekly lectures. The writer is also grateful to friends that helped the writer on solving some problems regarding this paper. The writer is also grateful to the writer's tutor in LOPI who inspire the title of this paper. The writer is also very grateful to close friends who keep on supporting the writer everyday on this pandemic.

REFERENCES

[1] https://socs.binus.ac.id/2019/12/26/binary-search/

[2] https://www.geeksforgeeks.org/decrease-and-conquer/

[3] https://www.konsep-matematika.com/2016/08/anuitas-dan-angsuran-matematika-keuangan.html

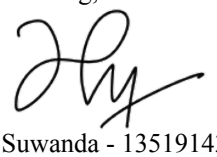
[4] https://www.akseleran.co.id/blog/anuitas-adalah/

[5] https://www.merrilledge.com/article/use-annuities-to-prepare-for-your-future#:~:text=Annuities%20are%20used%20mainly%20to,withdrawals%20or%20receiving%20periodic%20payments.

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, 5 Mei 2021



Hokki Suwanda - 13519143