“Guessing game” assignment

Izaak van Dongen

October 30, 2017

Contents

Listings

List of Figures

Introduction

This is an assignment about guessing games and appropriate strategies. This is linked to searching algorithms, although in this case it’s a sort of searching algorithm where you’re searching for the item, rather than searching for the index of a known item.

Both binary searches and linear searches can be adapted to that end, as both work by comparisons of the target and “guesses”. These are analogous to asking the user questions - in the case of linear search, the question “is your number equal to . . . ?”, and in the case of binary search, “is your number greater than . . . ?”

Therefore, essentially we need only to implement binary and linear search where the "comparison function" delegates to the user. However, interpolation search, which has the complexity $O(\log(\log(n)))$ (on uniform data sets, which we can assume a user-generated target is), requires knowledge of the *value* of the target, so it can be interpolated on the current upper and lower bounds. This means we will only achieve the $O(\log(n))$ complexity of binary search.

Sadly, we also won't be able to implement any such search algorithm for the set of reals $\mathbb{R}$, or any set with equivalent or higher cardinality (eg $\mathbb{C}=\mathbb{R}^2 \Rightarrow |\mathbb{C}|=|\mathbb{R}|$). This is because they are not a countably infinite set - there exists no bijection between them and the natural numbers. If we assume we could write a search algorithm to guess a real number, each of the algorithm's guesses could be labeled with a natural number, implying there does exist some enumeration of the reals. This contradicts lots of established knowledge (eg Cantor's diagonal argument) and so we know this won't be possible.

Code for this assignment has been written in Pascal, targeted at and tested in fpc (version 3.0.0+dfsg-2), using mode `{$MODE OBJFPC}`. This document was developed in LaTeX.

Where I've deemed it necessary, I've left comments, although often the code really is relatively simple, and the functions are sufficiently described by their names and the simplicity of the operations ("self-documented").

At some points, I use some algebraic notation. I use $\wedge$, which means logical and, to indicate that multiple statements are true.

My general model/definition for a "search algorithm" is an algorithm that can ask only boolean questions of a user, and must use ideally as few of these as possible to guess a target number that the user knows.

Generally, I have provided pseudocode for my algorithms. My pseudocode is probably pretty close to structured english, but I feel like if I were to make it much more code-like it would more or less be Pascal/Python.

Boilerplate

To create a program that performs such a search to guess a number, some utility functions will be needed, that aren't really particularly algorithmic. I'm implementing these in a Pascal unit, named `PUser.pas`. Having these in a separate unit means the code concerned with the actual algorithms can be more "purely" expressed. It also means that that code can become more modular itself, as another caller could feasibly implement their own boilerplate, and the abstracted algorithms will still work.

oracle

This is a function that can retrieve a boolean value from the user. It's named "oracle" as it acts as the comparison oracle the search functions must appeal to. The only parameter it takes is a message to display. The unix-inspired ruling it uses is that if an input starts with a "y", it is considered affirmative, and negative otherwise. The broad approach it uses is like this:

```
If the user's response is empty, the result is negative
If the user's response is not empty and begins with a lower or uppercase "y", the result is positive
Otherwise, the result is negative
```

Listing 1: oracle pseudocode

The catch for lower or uppercase is implemented by coercing the entire string to lowercase, using `LowerCase`. It's implemented as:

```
1  function oracle(msg: string): boolean;
2  var
3      usr_input: string;
4  begin
5      write(msg);
6      readln(usr_input);
7      if (length(usr_input) > 0) and (LowerCase(usr_input)[1] = 'y') then
8          oracle := true
9      else
10         oracle := false;
11 end;
```

Listing 2: oracle implementation

NB: The lack of semicolon in line 8 is on purpose, as this is how an if-else statement works in Pascal (as it's considered one statement, the whole affair needs one semicolon).

read_lohi

NB this function is currently unused in the source, but it was used for testing one of the earlier adaptations of the binary search algorithm. It also provides some interesting talking points about call signatures in Pascal.

I wanted to provide some facility for setting the upper bound and lower bound of a binary search. As this function wants to kind of "return" two things, it has a nontrivial call signature, which is worth thinking about. One possible solution might be to define some kind of `Bounds` container class that can hold both values.

However, Pascal provides a way to output multiple values, and this is through an `out` parameter. It could also be done using a `var` parameter, although there's a slight difference. Both of these work, and in both cases the variable is passed by reference, but `out` is slightly more specific - it's used when the input value of the variable is unneeded, which in this case is true, as I'm only concerned with using it as a channel for output. Once I've declared something as an `out` parameter, I can assign to it and the caller can then use the new value (like the `readln` function).

I will be reading these from the command line arguments, as I'm developing it as a console application. As this is quite a simple application, I will just manually parse arguments with some if statements. The function will roughly do the following:

```
Gather default values for the upper bound and lower bound
If a first command line argument is present
    Set the lower bound to this number
Otherwise, set it to the default
If a second command line argument is present
    Set the upper bound to this number
Otherwise, set it to the default
If either argument given was not an integer
    Set both to the defaults
```

Listing 3: read_lohi pseudocode

This is then implemented in Pascal using `ParamStr` and `ParamCount`. Interestingly, as it doesn't return anything in the conventional manner, this isn't a function but a procedure.

```pascal
procedure read_lohi(lo_default, hi_default: integer;
                    out low_val, hi_val: integer);
begin
    try
        if ParamCount >= 1 then
            low_val := StrToInt(ParamStr(1))
        else
            low_val := lo_default;
        if ParamCount >= 2 then
            hi_val := StrToInt(ParamStr(2))
        else
            hi_val := hi_default;
    except
        on E: EConvertError do begin
            writeln('Conversion error occurred, reverting to defaults');
            low_val := lo_default;
            hi_val := hi_default;
        end;
    end;
end;
```

Listing 4: read_lohi implementation

Linear Search

One approach to this is by a "linear search". This involved, basically, a kind of "brute force" approach - sequentially making guesses until one is correct. The single advantage of this algorithm is that it has no upper bound, and can even feasibly be made to work without a lower bound.

Positive integers

The most simple approach is to assume the number is some x such that

$$x \in \{0\} \cup \mathbb{N} \tag{1}$$

ie the set of natural numbers including 0. Guesses can then be made sequentially like so:

```
Set the current ``guess'' to 0
While the guess is wrong
    Increment the guess by one
```

Listing 5: Linear search on $\mathbb{N}$ pseudocode

In Pascal, this could be implemented like so:

```
function linear_search: integer;
var
    i: integer = 0;
begin
    while not oracle('Is your number equal to ' + IntToStr(i) + '? ') do
        i := i + 1;
    linear_search := i;
end;
```

Listing 6: Linear search on $\mathbb{N}$ implementation

All integers

The flaw in the previous program is that if the user sneakily decides to think of a negative number, this program won't ever terminate. This can be solved by using some enumeration of the set of all integers $\mathbb{Z}$. The simplest of these is the sequence 0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6 ... This sequence doesn't duplicate 0, and has the property of "spreading out" from 0. A sequence kind of like this with a lot of overhead could be represented by simply generating the set of natural numbers $\mathbb{N}$ and then taking each of these and its negation. However, if we consider it as more of a mathematical sequence, it *can* be represented by a closed formula

$$U_n = (-1)^n \left\lfloor \frac{n}{2} \right\rfloor \tag{2}$$

However, it's much better represented inductively with a condition:

$$U_1 = 0$$
$$U_{n+1} = \begin{cases} -U_n + 1 & \text{if } U_n \leq 0 \\ -U_n & \text{if } U_n > 0 \end{cases}$$

This can also quite easily be implemented in code:

```
function linear_search: integer;
var
    i: integer = 0;
begin
    while not oracle('Is your number equal to ' + IntToStr(i) + '? ') do
        if i <= 0 then
            i := -i + 1
        else
            i := -i;
    linear_search := i;
end;
```

Listing 7: Linear search on $\mathbb{Z}$ implementation

This does not need any double loops, duplicate code to explicitly negate each number, or extra boilerplate logic to prevent duplicating a 0.

Rational numbers

Interestingly, a linear search can also be generalised to work on the set of rational numbers $\mathbb{Q}$. This is closely related to the fact that the cardinality of the set of rationals

$$|\mathbb{Q}| = |\mathbb{N}| \tag{3}$$

This means there exists a bijection from $\mathbb{N}$ to $\mathbb{Q}$, and hence there must exist some enumeration of the elements of $\mathbb{Q}$, which can be obtained by applying the bijection to all the naturals in order.

We can easily come up with such an enumeration by considering the set of rationals as a grid with two integer axes representing numerator and denominator (in other words, the set $\mathbb{Z}^2$). We can then "spiral outwards" from the origin. This will be easiest to do in squares aligned with the axes, through which we move in the anticlockwise direction, although it could also be done in the diagonal direction. The first part of the grid, in terms of co-ordinate pairs, will be like this:

$$\begin{array}{l}
(-2,\ \ 2)(-1,\ \ 2)(0,\ \ 2)(1,\ \ 2)(2,\ \ 2)\\
(-2,\ \ 1)(-1,\ \ 1)(0,\ \ 1)(1,\ \ 1)(2,\ \ 1)\\
(-2,\ \ 0)(-1,\ \ 0)(0,\ \ 0)(1,\ \ 0)(2,\ \ 0)\\
(-2,-1)(-1,-1)(0,-1)(1,-1)(2,-1)\\
(-2,-2)(-1,-2)(0,-2)(1,-2)(2,-2)
\end{array}$$

Where the first couple of terms of the enumeration are:

$$\begin{array}{l}
(0,0),\\
(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1),(1,0),\\
(2,1),(2,2),(1,2),(0,2),(-1,2),(-2,2),(-2,1),(-2,0),...\\
(3,1),(3,2),...
\end{array} \tag{4}$$

Note that the first term is $\frac{0}{0}$. This technically is not a rational number as its value is undefined, but we can deal with that later - first we can establish the enumeration, and then "filter" it for unwanted pairings. Note also that this enumeration will not preserve any conception of size of fractions - $\frac{1}{2}$ will occur long before $\frac{1}{100}$.

We know that this path will reach all of the points on this grid, hence this enumeration will eventually reach the target - and this is where a linear search is useful. First, we will need to consider how we can implement this enumeration. We can consider the desired behaviour on each side of a square, and the conditions describing each such side.

Let us consider two sequences X and Y, where we say that

$$\frac{X_i}{Y_i} \tag{5}$$

is the ith element of our enumeration.

We know the first term is $\frac{0}{0}$. Hence, we have $X_1=0 \wedge Y_1=0$

We also say that we jump to the next layer of square when we "hit" the positive x-axis. To start the next square, we move to the right one and up one, as if we just move to the right, we stay on the axis and move right again. This means $X_i \geq 0 \wedge Y_i=0 \Rightarrow X_{i+1}=X_i+1 \wedge Y_{i+1}=1$

If we are on the right side of the square, we know that X_i is positive. We also know that $-X_i \leq Y_i < X_i$. When this is true, we want to move "up", so increment Y. We also know that $Y_i=0$ is considered a special case. Hence, we say that

$$X_i>0 \wedge -X_i \leq Y_i < X_i \wedge Y_i \neq 0 \Rightarrow X_{i+1}=X_i \wedge Y_{i+1}=Y_i+1 \tag{6}$$

We can produce similar definitions for the other three sides:

$$\begin{array}{l}
Y_i>0 \wedge -Y_i < X_i \leq Y_i \Rightarrow X_{i+1}=X_i-1 \wedge Y_{i+1}=Y_i\\
X_i<0 \wedge X_i < Y_i \leq -X_i \Rightarrow X_{i+1}=X_i \wedge Y_{i+1}=Y_i-1\\
Y_i<0 \wedge Y_i \leq X_i < -Y_i \Rightarrow X_{i+1}=X_i+1 \wedge Y_{i+1}=Y_i
\end{array}$$

Now that we have a way to produce such an enumeration, we can consider a filtering. The first condition is that we skip any number with a denominator of 0, as this is undefined. The second condition is that we should only consider fractions in their simplest form - ie the numerator and denominator are coprime. This means we don't duplicate any rationals and so obtain a strict bijection between the naturals and the rationals. Knowing that

$$\text{A and B are coprime} \Leftrightarrow gcd(A,B)=1 \tag{7}$$

We can simply calculate the gcd of X and Y using Euclid's algorithm, implemented like so in Pascal:

```pascal
function gcd(a, b: integer): integer;
var
  temp: integer;
begin
  while b <> 0 do begin
    temp := b;
    b := a mod b;
    a := temp;
  end;
  gcd := a;
end;
```

Listing 8: Euclid's algorithm in Pascal

This also works how we want for negative numbers - printing a couple of examples:

```
gcd 1 2 is 1
gcd 1 -2 is 1
gcd -1 2 is -1
gcd -1 -2 is -1
```

Listing 9: GCD behaviour for negative numbers

We can see that only one of the pairs 1,−2 and −1,2 are considered coprime, which is correct as $\frac{1}{-2} = \frac{-1}{2} = -\frac{1}{2}$. We can also see that -1 and -2 are not considered coprime, which is again correct as it is equivalent to $\frac{1}{2}$.

We can now put all of this together to obtain a fully featured linear search algorithm on the set of rational numbers:

```
procedure linear_search(out o_x: integer;
                        out o_y: integer);
var
    x: integer = 0;
    y: integer = 0;
begin
    while (y = 0) or (gcd(x, y) <> 1) or
      (not oracle('Is your number equal to ' + IntToStr(x) + '/' + IntToStr(y) + '? ')) do
        if (x >= 0) and (y = 0) then begin
            x := x + 1;
            y := y + 1;
        end else if (x > 0) and (-x <= y) and (y < x) then
            y := y + 1
        else if (y > 0) and (-y < x) and (x <= y) then
            x := x - 1
        else if (x < 0) and (x < y) and (y <= -x) then
            y := y - 1
        else
            x := x + 1;
    o_x := x;
    o_y := y;
end;
```

Listing 10: Linear search on $\mathbb{Q}$ implementation in Pascal

This again uses the `out` parameter trick to return multiple values. It is also a little terser than the fully qualified mathematical conditions given earlier, as some conditions are mutually exclusive/eliminated. Interestingly, the body of the while loop is technically a single statement, so no begin ...end block is needed. The condition of the loop is that it keeps going if the number is one to be skipped, or the user does not indicate that the guess is correct.

Another thing this code makes use of is short-circuit boolean evaluation. All of the "filter" checks have been compounded into one boolean expression, using the or operator. Even though there is a call to oracle in this statement, the call will only actually be evaluated if neither of the filter checks flag up - if it doesn't pass the filter, it will short-circuit so the call to oracle will be skipped.

A parting consideration is that the linear search is very "secure", in a sense - a malicious user will not be able to confuse the linear search in any way, they will only be able to jeopardise their own experience - any sequence of answers for a linear search will seem sensible to it, even if they are intended maliciously. Really, a user's only power is to lie about what their number was, which a simple command line program won't realistically be able to defend against.

Binary search

At last! The binary search algorithm is significantly faster than the linear search, in the average case (being $O(\log(n))$).

Basic binary

In its most basic form, the binary search might work something like this in pseudocode:

```
Establish a lower bound and a strict upper bound
While the difference between the bounds is greater than one
    Guess a number in the middle of these bounds
    If the guess was too high, move the upper bound down to the guess
    If the guess was too low, move the lower bound up to the guess
```

Listing 11: Basic binary search pseudocode

Note that this algorithm will rely on some ”magic” upper and lower bound, for now. I've used some slightly careful consideration of range arithmetic and definitions - by having a strict upper bound, when the length of the range is one, the only remaining candidate in the range is the lower bound. This is one easy way to prevent off-by-one errors and too much case-checking here. It can be implemented in Pascal quite trivially, like so:

```
function binary_search(lower, upper: integer): integer;
var
    mid: integer;
    is_ge: boolean;
begin
    while upper - lower > 1 do begin
        mid := (upper + lower) div 2;
        is_ge := oracle('Is your number greater than or equal to ' + IntToStr(mid) + '? ');
        if is_ge then
            lower := mid
        else
            upper := mid;
    end;
    binary_search := lower;
end;
```

Listing 12: Basic binary search in Pascal

This function takes two parameters, the bounds, and returns the guessed number. It is also relatively immune to malicious users - again, every sequence of answers leads to a correct guess, assuming all the answers were correct.

Bound-finding binary search

Our binary search could be made significantly less naïve by not putting a bounds restriction on the user. We can actually determine a bound quite quickly, assuming nothing other than that the the user's number will be an integer.

Continuing the theme of binary and powers of two, we can simply check the “bound” between each of the adjacent powers of 2, ie first 1-2, then 2-4, then 4-8 (note that these are the good old fashioned strict upper bound ranges, so there is no overlap). This will converge to some number n in logarithmic time, again. However, this does not account for negative numbers - what we can easily do is simply consider the *absolute* value of the user's number $|n|$. We need then only ask if it is negative once, at the end. If it is negative, we will need to “flip” the range, bearing in mind to increase the new lower bound and decrease the new upper bound, to make them non-strict and strict, respectively. This leaves the single edge case where the user's number is 0, as this is not an integer power of 2. We can explicitly check for this to get around it (not very elegant, but simple enough).

In pseudocode, this might be implemented as follows:

```
If the user's number is 0, return the range 0-1
For each pair a and b of powers of 2
    If a <= |n| < b
        Remember this range, and stop guessing ranges
If the user's number is positive, return this range
Otherwise, return the ``flipped'' range
```

Listing 13: Bound-finding pseudocode

This then comes to the following implementation in Pascal:

```
procedure binary_find_bounds(out o_lower: integer;
                             out o_upper: integer);
var
    lower: integer = 1;
    upper: integer = 2;
begin
    writeln('This is the stage where I determine some bounds on your number. I'
           , ' will be asking questions about the *magnitude* of your number,'
           , ' so watch out.');
    if oracle('Is your number 0? ') then begin
        o_lower := 0;
        o_upper := 1;
    end else begin
        while not oracle('Is your number some x such that ' +
          IntToStr(lower) + ' <= |x| < ' + IntToStr(upper) + '? ') do begin
            lower := lower * 2;
            upper := upper * 2;
        end;
        if oracle('Is your number greater than or equal to 0? ') then begin
            o_lower := lower;
            o_upper := upper;
        end else begin
```

```
            o_lower := - upper + 1;
            o_upper := - lower - 1;
        end;
    end;
end;
```

Listing 14: Bound-finding subroutine in Pascal

Using this, we can successfully perform a binary search on the set of all integers $\mathbb{Z}$. In fact, we only need to first call this subroutine to determine the bounds, and then let the previous function from listing 12 do the rest.

Rational numbers

Interestingly, theoretically we already have enough "material" to put together a binary search on the set of all rationals. We can use the previously defined enumeration, and then simply ask the user to think of what the index of their number is in that enumeration, and then perform a binary search as normal to find that integer. Here, the concept of "greater than" has been entirely abstracted away from the comparison of actual rationals.

However, this makes some rather theoretical demands of the user, and is better left as a perverse though experiment. There are far better ways to approach this. Namely, there exists a binary search tree of the set of rationals - the "Stern-Brocot" tree, which uses fractional mediants:

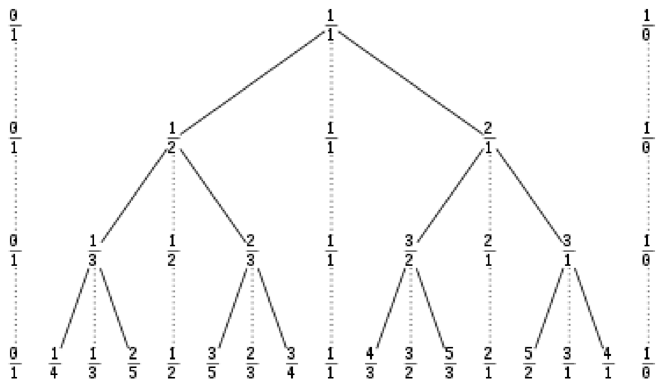

Figure 1: A Stern-Brocot tree

This also addresses the concern about comparison - this tree does preserve a proper ordering of the rationals, due to the fact that the mediant of two fractions lies strictly in between those fractions.

Interestingly, this tree has kind of similar behaviour to what we encountered in the bound-finding function - it only terminates on positive nonzero numbers. We can get around this in the same way - first check for 0, ask questions about the absolute value of the user's number and then ask if it is negative:

```
Ask if the user's number is 0
Set the lower and upper bound to 0/1 and 1/0
While you haven't guessed the user's number
    Generate the mediant of the bounds
    If this is the number, stop
    If this is too high, move the upper bound down to the mediant
    Otherwise, move the lower bound up to the mediant
```

Listing 15: Binary search on $\mathbb{Q}$ in pseudocode

This can be implemented like so in Pascal:

```
procedure binary_search(out o_x: integer;
                        out o_y: integer);
var
    mid_x, mid_y: integer;
    lo_x: integer = 0;
    lo_y: integer = 1;
    hi_x: integer = 1;
    hi_y: integer = 0;
    is_ge: boolean;
begin
    if oracle('Is your number equal to 0? ') then begin
```

```
        o_x := 0;
        o_y := 1;
    end else begin
        repeat
            mid_x := lo_x + hi_x;
            mid_y := lo_y + hi_y;
            is_ge := oracle('is your number q such that '
                         + IntToStr(mid_x) + '/' + IntToStr(mid_y)
                         + ' <= |q|? ');
            if is_ge then begin
                lo_x := mid_x;
                lo_y := mid_y;
            end else begin
                hi_x := mid_x;
                hi_y := mid_y;
            end;
        until is_ge and oracle('is your number q such that |q| = '
                         + IntToStr(mid_x) + '/' + IntToStr(mid_y) + '? ');
        o_x := mid_x;
        o_y := mid_y;
        if oracle('Is your number less than 0? ') then
            o_x := -mid_x;
    end;
end;
```

Listing 16: Binary search on $\mathbb{Q}$ in Pascal

Interface

Having implemented all of the algorithms, it was now time to write an interface. First, I wrote a number of procedures to go with each algorithm, prefixed with "perform_". They mainly just printed some information and formatted the results to the user. For example, to go with the binary search on $\mathbb{Z}$:

```
procedure perform_bsearch;
var
    guess: integer;
    lower, upper: integer;
begin
    writeln('welcome to the binary search game! Think of an integer.');
    binary_find_bounds(lower, upper);
    guess := binary_search(lower, upper);
    writeln('your number was ', guess);
end;
```

Listing 17: Binary search on $\mathbb{Z}$ wrapper procedure

Using such procedures, I then wrote the following short program:

```
while true do begin
    if oracle('Would you like to perform an integer binary search? ') then
        perform_bsearch
    else if oracle('Would you like to perform a rational binary search? ') then
        perform_fracbsearch
    else if oracle('Would you like to perform an integer linear search? ') then
        perform_lsearch
    else if oracle('Would you like to perform a rational linear search? ') then
        perform_fraclsearch;
end;
```

Listing 18: Main interface loop

Source

The full project in its directory structure, including this document, can be found at `https://github.com/elterminad0r/assignment_guessing`.