“Guessing game” assignment

Izaak van Dongen

October 21, 2017

Contents

Listings

1 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.

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.

2 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.

2.1 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:

```
1 If the user's response is empty, the result is negative
2 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:

```
function oracle(msg: string): boolean;
var
    usr_input: string;
begin
    write(msg);
    readln(usr_input);
    if (length(usr_input) > 0) and (LowerCase(usr_input)[1] = 'y')
    then
        oracle := true
    else
        oracle := false;
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).

2.2 read_lohi

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.

```
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

3 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.

3.1 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

3.2 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:

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

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.

3.3 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:

$$\begin{array}{ccccc} (-2,2) & (-1,2) & (0,2) & (1,2) & (2,2) \\ (-2,1) & (-1,1) & (0,1) & (1,1) & (2,1) \\ (-2,0) & (-1,0) & (0,0) & (1,0) & (2,0) \\ (-2,-1) & (-1,-1) & (0,-1) & (1,-1) & (2,-1) \\ (-2,-2) & (-1,-2) & (0,-2) & (1,-2) & (2,-2) \end{array} \tag{4}$$

Where the first couple of terms of the enumeration are:

$$
\begin{array}{l}
(0,0), \\
(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1),(1,0), \\
(2,1),(2,2),(1,2),(0,2),(-1,2),(-2,2),(-2,1),(-2,0),\ldots \\
(3,1),(3,2),\ldots
\end{array}
\tag{5}
$$

Note that the first term is $\frac{0}{0}$. This technically is not a rational numbers 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

$$
\frac{X_i}{Y_i} \tag{6}
$$

is the ith element of our enumeration.

We know the first term is $frac00$. Hence, we have $X_1 = 0 \land Y_1 = 0$

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 $X_i \geq 0 \land Y_i = 0 \Rightarrow X_{i+1} = X_i + 1 \land Y_{i+1} = 1$

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

$$
X_i > 0 \land -X_i \leq Y_i < X_i \land Y_i \neq 0 \Rightarrow X_{i+1} = X_i \land Y_{i+1} = Y_i + 1 \tag{7}
$$

We can produce similar definitions for the other three sides:

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

4 Source

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