# Guessing game assignment

### Izaak van Dongen

### October 19, 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.

# 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`.

## 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
3  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 ;
```

<div align="center">Listing 2: oracle implementation</div>

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:

```
1 Gather default values for the upper bound and lower bound
2 If a first command line argument is present
3     Set the lower bound to this number
4 Otherwise , set it to the default
5 If a second command line argument is present
6     Set the upper bound to this number
7 Otherwise , set it to the default
8 If either argument given was not an integer
9     Set both to the defaults
```

<div align="center">Listing 3: read_lohi pseudocode</div>

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.

```
1  procedure read_lohi(lo_default, hi_default: integer;
2                       out low_val, hi_val: integer);
3  begin
4      try
5          if ParamCount >= 1 then
6              low_val := StrToInt(ParamStr(1))
7          else
8              low_val := lo_default;
9          if ParamCount >= 2 then
10             hi_val := StrToInt(ParamStr(2))
11         else
12             hi_val := hi_default;
13     except
14         on E: EConvertError do begin
15             writeln('Conversion error occurred, reverting to
   defaults');
16             low_val := lo_default;
17             hi_val := hi_default;
18         end;
19     end;
20 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:

```
1  Set the current ''guess'' to 0
2  While the guess is wrong
3      Increment the guess by one
```
Listing 5: Linear search on $\mathbb{N}$ pseudocode

In Pascal, this could be implemented like so:

```
1  function linear_search: integer;
2  var
3      i: integer = 0;
```

```
4  begin
5      while not oracle('Is your number equal to ' + IntToStr(i) + '?
       ') do
6      i := i + 1;
7      linear_search := i;
8  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:

```
1  function linear_search: integer;
2  var
3      i: integer = 0;
4  begin
5      while not oracle('Is your number equal to ' + IntToStr(i) + '?
       ') do
6          if i <= 0 then
7              i := -i + 1
8          else
9              i := -i;
10     linear_search := i;
11 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.

# 4    Source

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