Guessing game assessment

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:

```
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:

```
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:

```
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
```

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.

```
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
```

```
            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

Full source:

```
{$MODE OBJFPC}

unit PUser;

interface

function oracle(msg: string): boolean;
procedure read_lohi(lo_default, hi_default: integer;
                    out low_val, hi_val: integer);

implementation

uses SysUtils;

type
    OracleFunction = function(query: integer): boolean;

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;

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
```

```
41              hi_val := hi_default;
42      except
43          on E: EConvertError do begin
44              writeln('Conversion error occurred, reverting to
    defaults');
45              low_val := lo_default;
46              hi_val := hi_default;
47          end;
48      end;
49  end;
50
51  initialization
52  begin
53      writeln('To answer affirmatively, you should write anything
    starting with '
54            , 'a "y". Anything else will be considered negative.
    Lower and upper'
55            , ' bounds can be provided in argv');
56  end;
57
58  end.
```

Listing 5: Full PUser.pas

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.

Source:

```
1   {$MODE OBJFPC}
2
3   unit PLinear;
4
5   interface
6
7   procedure perform_lsearch;
8
9   implementation
10
11  uses PUser, SysUtils;
12
13  function linear_search: integer;
14  var
15      i: integer = 0;
16  begin
17      while not oracle('Is your number equal to ' + IntToStr(i) + '?
    ') do
18          i := i + 1;
19      linear_search := i;
20  end;
21
22  procedure perform_lsearch;
```

```
23 var
24     guess: integer;
25 begin
26     writeln('Think of a number x >= 0');
27     guess := linear_search;
28     writeln('Your number was ', guess);
29 end;
30 
31 end.
```

Listing 6: Full PLinear.pas