TP OOP N°5

Master 1 MISC

École d'ingénieurs du Littoral-Côte-d'Opale

Université du Littoral Côte d’Opale

Document à rendre :

un rapport de TP qui contient

les commandes utilisées,

avec description et références à l'énoncé,

et commentaires éventuels.

Attention

Lisez ATTENTIVEMENT les explications et les consignes de travail.

Attention

Remarque : vous conserverez une trace numérique de toutes les actions réalisées dans le fichier de scripts. Cela vous permettra de relancer des ensembles de commandes en une seule fois. Vous remettrez le fichier à la fin de chaque séance.

Exercise 1:

Let's play a little game to give you an idea of how different algorithms for the same problem can have wildly different efficiencies. User is going to randomly select an integer in some range (0 and 100). The computer shall keep guessing numbers until it find the user's number, and the user will tell each time if the number is found or if the guess was too high or too low.

We can further personalize the application for the guessing game, in order to keep track of the set of reasonable guesses using a few variables. Let the variable *min* be the current minimum reasonable guess for this round, and let the variable *max* be the current maximum reasonable guess. The *input* to the problem is the number *n* the highest possible number that your opponent is thinking of. We assume that the lowest possible number is zero, but it would be easy to modify the algorithm to take the lowest possible number as a second input.

Here's a step-by-step description of using binary search to play the guessing game:

1. Let *min*=1 and *max*=*n*.
2. Guess the average of *max* and *min* rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set *min* to be one larger than the guess.
5. If the guess was too high, set *max* to be one smaller than the guess.
6. Go back to step two.

Write a python class to implement the above functionality, along with the test program in form of a GUI.

Exercise 2**:**

You have an array containing the prime numbers between 0 and 100 in sorted order: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]. There are 25 items in the array. About how many items of the array would binary search have to examine before concluding that a number is not in the array, and therefore not prime?

Here's the pseudocode for binary search, modified for searching in an array. The inputs are the array, which we call `array`; the number `n` of elements in `array`; and `target`, the number being searched for. The output is the index in `array` of `target`:

Here is pseudocode for binary search that also handles the case in which the target number is not present:

```
1. Let min = 0 and max = n-1.
2. If max < min, then stop: target is not present in array. Return -1.
3. Compute guess as the average of max and min, rounded down (so that it is an integer).
4. If array[guess] equals target, then stop. You found it! Return guess.
5. If the guess was too low, that is, array[guess] < target, then set min = guess+1.
6. Otherwise, the guess was too high. Set max = guess - 1.
7. Go back to step 2.
```

Write a python class to implement the above functionality, along with the test program.

x-x-x-x-x