MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Problem Set 1

Search and Hide Games

Issued: Tuesday, September 9

Due: In recitation on Wednesday, September 17

Read "How should I write up my problem set?" on the 6.001 homepage.
Reading for this Problem Set: Section 1.1 of *SICP*.
Reading for Lectures: Section 1.2.

Introduction

1 Tutorial Preparation

Tutorial Exercise 1: Write a tail-recursive procedure that will add the integers from 1 to n, where n is an argument provided to the procedure.

Tutorial Exercise 2: Modify the above procedure to add the squares of the integers from 1 to n.

Tutorial Exercise 3: Do Exercise 1.5 of *SICP*.

Tutorial Exercise 4: Do Exercise 1.6 of *SICP*.

2 Guessing Games

Even before the invention of GameBoys (back in the dark ages), parents found ways to entertain young children on long family car trips by singing and playing games like "I Spy" or "I know a number."

In this last game, a parent chooses a range of numbers, say 0 to 77, and one child, called the *hider*, is told to pick a "hidden" integer in the range (0 and 77 included), and then announces, "I am thinking of a number." Another child, called the *searcher*, now chooses a point, say 30, and asks the hider, "Is your number higher than 30?" This is the only kind of question the searcher may ask. The game is over when the searcher knows for sure what the number is. The object for the searcher is to guess the number with as few questions as possible.

In Lab, Exercise 1:

1A. Use `M-x load-problem-set` so that the code in the problem set files is evaluated. Then, evaluate the expressions

```
(define hider naive-hider)
(define searcher user-searcher)
```

This tells the computer that you want to be the searcher (i.e., you want to use the `user-searcher` procedure), and that you want to play against a hider known as the `naive-hider`.

Now, evaluate the expression

```
(play 100)
```

This means that you want to play the game you just set up, and that the number the hider chooses should be in the range from 0 to 100 (inclusive). The procedure `play` will return as its value the number of questions the you asked in your game before finding the hidden number.

Play the game. What strategy did you use? How many questions did you ask? What number did the hider choose?

1B. Of course, in any given game, you might do particularly well just out of luck. Let's run a tournament, to see how well you can do on average over several games. Evaluate

```
(tournament 4 70)
```

to play four games on the interval zero to 70.

The `tournament` procedure prints out results for each game, and at the end prints out the mean (average) and deviation of the number of questions required in the games. (*Deviation* serves as a measure of how widely spread the numbers are about their mean.)

```
(tournament 10 77)

    ;;  game no: 1  game-length: 12
    ;;  game no: 2  game-length: 20
    ;;  game no: 3  game-length: 14
    ;;  game no: 4  game-length: 13
    ;;  game no: 5  game-length: 12
    ;;  game no: 6  game-length: 13
    ;;  game no: 7  game-length: 15
    ;;  game no: 8  game-length: 15
    ;;  game no: 9  game-length: 15
    ;;  game no: 10  game-length: 11

The MEAN game length was 14, with DEVIATION 2.408
```

1C. Finally, try

```
(define hider user-hider)
(define searcher random-searcher)
```

Now you will be the hider in the game. Play a tournament of four games on a reasonable-size interval. How many questions did you average against the `random-searcher`?

2.1 Basic Searching and Hiding Strategies

To define your own hider or searcher, all you have to do is write a procedure. A searcher is defined by a procedure that takes the endpoints of an interval, say 0 and 77, and returns the *split-point* that the searcher wants to ask about. A hider is defined by a procedure that, given the endpoints and the split point from the searcher, returns the answer `higher` or `not-higher`.

For example, the random searcher you played against hasn't a clue, and simply guesses at random, using the special `guess` procedure provided in this problem set. (Given the interval endpoints, `guess` generates at random an integer in the interval.)

```
(define (random-searcher low high)
  (guess low (dec high)))
```

The implacable searcher works through the interval by searching through the interval from low to high.

```
(define (implacable-searcher low high)
  low)
```

The *binary* searcher, splits the interval as equally as possible:

```
(define (binary-searcher low high)
  (floor (+ low (/ (- high low) 2))))
```

So if the interval is 0 to 77, the binary searcher's first question is "Is your number at most 38?"

One of the simplest hiders is the naive hider you played against in Exercise 1. It always tries to hide at the number 42.

Pre-Lab, Exercise 2: Take a look at the definition of `naive-hider` in the file `ps1-hide.scm`. What does it do in those cases when it is unable to hide at 42? You are welcome to play some experimental games to discover the answer, but you should give a simple explanation in English of how the `naive-hider` code works to cause this behavior.

When older, more devious children are hiders, they *do not actually pick a number in advance*. Instead, they give answers that prolong the game for as many questions as possible; of course, they must choose answers so it looks like they really have naively picked a number in advance. In other words, the devious hider keeps choosing an interval to "hide in" while the searcher tries to narrow down the size of the hider's interval until there is no place left to hide.

Here is a hider that take advantage of this idea. The *elbow-room* hider always sneakily chooses the larger part to hide in:

```
(define (elbow-room-hider low split-point high)
   (if (< (- high split-point) (inc (- split-point low)))
       not-higher
       higher))
```

Pre-Lab, Exercise 3:

3A. The implacable searcher and the elbow-room hider play the same way every time, so when they play against each other, the number of questions on the interval 0 to n is determined solely by the interval size.

How many questions does the implacable searcher use against the elbow-room hider? What does the hidden number turn out to be?

3B. The binary searcher and the elbow-room hider also play the same way every time. Here is a procedure that computes how many questions they are going to ask:

```
(define (binary-vs-elbow-questions n)
  (if (= n 0)
      0
      (+ 1 (binary-vs-elbow-questions (floor (/ n 2))))))
```

Write a tail-recursive procedure `binary-vs-elbow-tr` that computes the same result.

In Lab, Exercise 4: Run twenty game tournaments of the random searcher against the elbow-room hider on several intervals whose lengths differ by at least a factor of ten. For each interval, compare the average number of questions used by the random searcher with the number of questions which the binary searcher would need (you may use `binary-vs-elbow-questions` to compute these numbers).

Formulate a hypothesis of the expected value of the ratio of the questions needed by the random searcher and the binary searcher; you are not required to supply a rationale for your hypothesis, just make a plausible guess.

Pre-Lab, Exercise 5: Define a searcher that behaves like the binary searcher on intervals of even length and like the random searcher on intervals of odd length. You should be able to write your code without copying the text of the definitions of either the `binary-searcher` or the `random-searcher`.

There's also a *random-answer* hider:

```
(define (random-answer-hider low split-point high)
  (if (= (guess 0 1) 0)
      not-higher
      higher))
```

In Lab, Exercise 6:

6A. Run some tournaments – with different size games – of the random-answer hider against the implacable searcher. Summarize your results.

6B. You should have noticed that the implacable searcher does well against the random-answer hider. A more plausible `random-location-hider`, picks a location at random in the current interval and answers as though it were at that location. While the random-answer hider answers "higher" with probability one half, the random location hider is more likely to answer "higher" when the interval above the split point is larger than the interval below. Define such a `random-location-hider`. Run some tournaments of the random-location hider against the implacable searcher. Summarize your results.

In Lab, Exercise 7: Procedures `play` in the file `play.scm`, and `tournament` in the file `tour.scm` are simple, tail-recursive procedures worth studying.

7A. Define a procedure called `play-reporting-high` that plays a searcher/hider game but returns the number of times the hider said "higher" (instead of the number of questions asked). (Remember, that whenever you submit a new procedure, you are supposed to select and run some test cases indicating that your procedure works properly.) *Hint*: `play-reporting-high` is definable by using a small modification of the "helper" procedure `finish-round`.

7B. Notice that the "helper" procedures, `tournament-loop` and `display-outcome`, of `tournament` have two arguments which never change while they are running, namely, `tournament-length` and `n`. Take advantage of Scheme's block structure (see *SICP*, Section 1.1.8) to eliminate these arguments. *Hint*: Reorganize the code so the helper procedures are defined within the `tournament` procedure.

In Lab and After Lab, Optional Exercise 8: Suppose the hider is allowed to *lie*, but only in response to *one* question. It decides which question, and naturally the searcher is not told which answer, if any, is the lie.

In this kind of one-lie game the players need only keep track of two more points, `maybe-low` and `maybe-high`, in addition to `low` and `high`. The points in non-decreasing order are `low`, `maybe-low`, `maybe-high`, `high`. If the hider hasn't lied yet, then the hider really is in the subinterval between `maybe-low` and `maybe-high`. The players also keep track of a boolean-valued variable `caught-in-lie` whose value becomes true when the searcher can detect the lie.

For example, the *risk-averse* searcher can't tolerate any uncertainty and, if necessary, it will repeat a question to make sure the answer is true. After it detects a lie, it acts like the binary searcher.

```scheme
(define (risk-averse-searcher
          low maybe-low maybe-high high caught-in-lie)
  (cond
    (caught-in-lie (binary-searcher low high))
    ((< low maybe-low) maybe-low)    ;ask MAYBE-LOW again
    ((< maybe-high high) maybe-high) ;ask MAYBE-HIGH
    (else (binary-searcher low high))))
```

The risk-averse searcher will find any one-lie hider using twice as many questions as the binary-searcher would have used against a truthful hider. There are other searchers who can do better against one-lie hiders.

Define the best searcher you can – the one who asks the fewest questions in the worst case – against a one-lie hider whom you may assume is as sneaky a hider as is possible. Turn in your definition along with an explanation of your search strategy. The 6.001 staff will run your searcher against its own sneaky hider and will award a prize to the searcher who does best over a selection of game sizes.

See Next Page

Turn in this sheet along with your answers to the questions in the problem set:

How much time did you spend on this homework assignment?

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).

If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

Otherwise, write "I worked alone using only the course reference materials." and sign your statement.