Homework 1: Writeup

University of Maine COS 554: Data Structures & Algorithms

Tristan Zippert

September 25, 2023

Contents

1 Algorithm Overview

The problem of enumerating all the possible ways a pogo sticks can be used to reach the goal is done with a recursive algorithm in Common Lisp. The main algorithm is called `Optimal-sort()` and takes in `n` as the goal, and `d` as the list of available pogo sticks in the backpack. As it goes through the recursive calls, the variable `n` represents the current distance the protagonist is from the exit door. The second parameter, `d`, represents the available pogo list with the distance values covered by each pogo stick. This list is iterated through if there are elements of the list. The final input to the algorithm is a optional input, called `jmp-lst`, which is a list containing the pogo sticks that get the player to the exit. This list is updated through the recursive call and eventually gets combined with the final pogo stick combination list before the algorithm returns

1.1 Code Overview

This section covers the code implementation of the algorithm, such as base cases, looping and the final algorithm. The code is written using Steel Bank Common Lisp, as it is a runtime for Common Lisp is open source with permissive license. Common Lisp was chosen considering its ease of use when writing recursive algorithms (and algorithms in general).

1.1.1 Base cases

Because this algorithm is recursive, proper base cases need to be defined. The first base case checks if the current value of `n`, the players distance from the exit. When `n`, the player, is at the goal then a reversal of the current jump list is returned to get added to throughout the recursive calls. The second check determines if the pogo list is blank, in which case `nil` is returned.
The final check determines if the player went beyond the exit with a pogo stick, which would result in a "splat". In this case, `nil` is also returned to prevent this jump from being added to the pogo list of moves. This code is reference (1) from the code described in the `Optimal-jump` function block.

```
(cond
  ((= n 0) (list (reverse jmp-lst)))
  ((null d) nil)
  ((< n 0) nil))
```

Figure 1: Algorithm base cases in a Lisp `cond` construct.

1.1.2 Code Loop

The second part of the `Optimal-jump` algorithm – showcased in part (2) of the overall algorithm below – generates a list variable and loops through the pogo list elements. Specifically, it binds a list called `output` as a blank list. After the blank list is bound to the `output` variable, a iteration through the pogo list is initiated. This uses Common Lisps `dolist` construct, which is functionally equal to a `for each` loop in other languages. During the iteration it uses `elem` for each value of pogo-list `d`, and if `n` is less than or equal to an element of pogo list a `when` statement is activated. Within the `when` construct, `output` list is set to itself appended to a recursive function call of `Optimal-list`.
The recursive call to `Optimal-list` subtracts the current distance to the goal from the element in the list as its first parameter. The second parameter is the pogo-list, `d`, as its used throughout the recursive calls. The third parameter in the recursive call is the resulting list containing the current `element` pushed to the jump-list. After the recursive part of the algorithm, the `output` list is returned, pushing through the stack with recursion to get the final output.

```
(let ((output nil))
 (dolist (elem d)
   (when (<= elem n)
     (setq output
   (append output (optimal-jump (- n elem)  d
        (cons elem jmp-lst))))))
 output))
```

1.1.3 Combined Algorithm

Below is the combined algorithm code programmed in Common Lisp. As described above, it has the base cases and looping part combined into one algorithm. The `declare` statements are used to specify what type the inputs to the algorithm are, allowing Lisp to perform some assembly level optimization to the function.

```
(defun optimal-jump(n d &optional (jmp-lst '()))
```

```
  "Optimal-jump algorithm returns a list of all possible iterations of the pogo-list,d, to get to the goal n"
  (declare (type integer n))
  (declare (type list d jmp-lst))
  (cond
    ((= n 0) (list (reverse jmp-lst)))
    ((null d) nil)
    ((< n 0) nil)
    (t(let ((output nil))
 (dolist (elem d)
   (when (<= elem n)
     (setq output
   (append output (optimal-jump (- n elem)  d
        (cons elem jmp-lst))))))
 output))))
```

Figure 2: Main algorithm for optimal jump in Common Lisp

Code Ouput

```
OPTIMAL-JUMP
```

Figure 3: Lisp REPL compilation output for `optimal-jump`

Considering that this is the main algorithm, and the user interaction is done through the terminal, the algorithm was tested with a printed function call. The printed function call does not format the output, as its just for debug purposes. An example of the function call and its result is displayed below.

```
(format t "~s" (optimal-jump 5 '(5 10 1 3)))
```

Code Ouput

((5) (1 1 1 1 1) (1 1 3) (1 3 1) (3 1 1))

Figure 4: Output of the function call above.

The code was also separated into two functions to make it easier to time


```
0: (OPTIMAL-JUMP 5 (5 10 1 3))
  1: (OPTIMAL-JUMP 0 (5 10 1 3) (5))
  1: OPTIMAL-JUMP returned ((5))
  1: (OPTIMAL-JUMP 4 (5 10 1 3) (1))
    2: (OPTIMAL-JUMP 3 (5 10 1 3) (1 1))
      3: (OPTIMAL-JUMP 2 (5 10 1 3) (1 1 1))
        4: (OPTIMAL-JUMP 1 (5 10 1 3) (1 1 1 1))
          5: (OPTIMAL-JUMP 0 (5 10 1 3) (1 1 1 1 1))
          5: OPTIMAL-JUMP returned ((1 1 1 1 1))
        4: OPTIMAL-JUMP returned ((1 1 1 1 1))
      3: OPTIMAL-JUMP returned ((1 1 1 1 1))
      3: (OPTIMAL-JUMP 0 (5 10 1 3) (3 1 1))
      3: OPTIMAL-JUMP returned ((1 1 3))
    2: OPTIMAL-JUMP returned ((1 1 1 1 1) (1 1 3))
    2: (OPTIMAL-JUMP 1 (5 10 1 3) (3 1))
      3: (OPTIMAL-JUMP 0 (5 10 1 3) (1 3 1))
      3: OPTIMAL-JUMP returned ((1 3 1))
    2: OPTIMAL-JUMP returned ((1 3 1))
  1: OPTIMAL-JUMP returned ((1 1 1 1 1) (1 1 3) (1 3 1))
  1: (OPTIMAL-JUMP 2 (5 10 1 3) (3))
    2: (OPTIMAL-JUMP 1 (5 10 1 3) (1 3))
      3: (OPTIMAL-JUMP 0 (5 10 1 3) (1 1 3))
      3: OPTIMAL-JUMP returned ((3 1 1))
    2: OPTIMAL-JUMP returned ((3 1 1))
  1: OPTIMAL-JUMP returned ((3 1 1))
0: OPTIMAL-JUMP returned ((5) (1 1 1 1 1) (1 1 3) (1 3 1) (3 1 1))
```

Figure 5: Call stack of `(optimal-jump 5 '(5 10 1 3))` call

just the algorithm, as opposed to timing the user interactions and outputs to IO.

1.1.4 IO interaction code

Since the main algorithm was developed separate from the user interaction function, a function had to be developed to handle IO to the user. This function uses `read-line` to get the line input from the user, then it uses `uiop:split-string` to split the string on whitespace. It then converts the list from the `uiop:split-string` into a list of numbers. When it calls `Optimal-jump`, it uses the first input of the formed input list as `n` and the rest as the pogo list, `d`.

```
(defun input-from-user()
  (let ((inp '()) (check nil) (result nil))
    (setq check (uiop:split-string (read-
line) :separator " "))
    (if check
  (progn
    (dolist (elem check)
      (push (parse-integer elem :junk-allowed t) inp)
```

```
      )
    (setq inp (reverse inp))
    (setq result (optimal-jump (car inp) (cdr inp)))
    (if result
        (dolist (elem result)
  (format t "~s~%" elem)
  )
        (format t "~s~%" nil)
        )
    )
  )
     )
   )
```

Figure 6: User input and output code

2 Measurement

Measurements were performed using Common Lisps `time` function on the main algorithm. Specifically, the total run time metric was used out of the output from the Lisp `time` function. As for the inputs of the function, tests were conducted with `n` equal to 1 up to 25, and `d` containing values all the values up to that iteration of `n`. As shown in the following code snippet for `n` =5, with values of `d` containing all the values up to 5 using the Lisp `time` function. The tests were created with the worst case of `n` and `d` in mind.

```
(time (optimal-jump 5 '(1 2 3 4 5)))
```

Code Ouput

Evaluation took:
0.000 seconds of real time
0.000073 seconds of total run time (0.000072 user, 0.000001 system)
100.00% CPU
0 bytes consed

Figure 7: SBCL output of `optimal-jump`, with `n` =5 and `d` containing values up to `n`.

For running the code, the command `sbcl -dynamic-space-size 8192` was used to increase the amount of dynamic space available to SBCL to 8192

MB. Lisp standard optimization was used, meaning there was no `(declaim (optimize))` used during compilation.

2.1 Results

The resulting times were recorded and graphed using a line plot. Around the `n` = 20 mark, each following run would double in run-time from the previous. This compares to the response in question 5 as the predicted run time was determined to be exponential with the amount of values with `d`.

N	Time (seconds)
1	0.000001
2	0.000002
3	0.000002
4	0.000002
5	0.000004
6	0.000006
7	0.000013
8	0.000027
9	0.000057
10	0.000130
11	0.000359
12	0.000743
13	0.001520
14	0.003598
15	0.007462
16	0.016481
17	0.033916
18	0.063142
19	0.127026
20	0.367729
21	0.839049
22	2.260689
23	5.083805
24	14.534721
25	34.100754

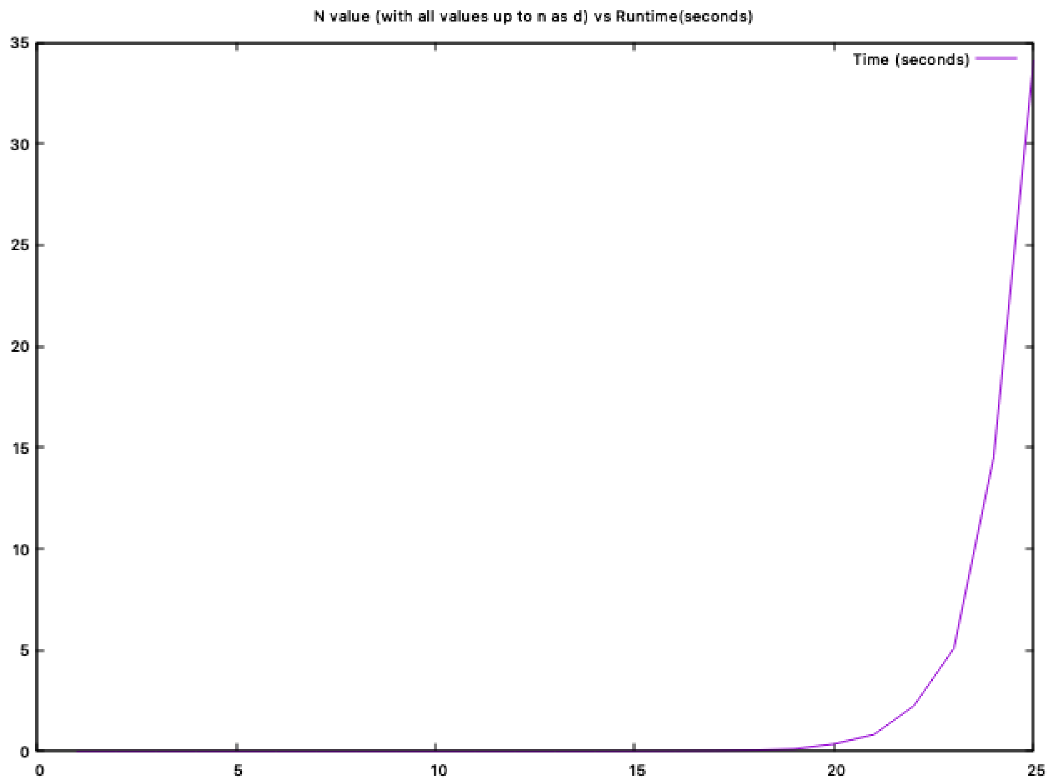

Figure 8: Graph generated from results in table 1

3 Known Bugs and limitations

A known limitation occurs when the `d` list input contains a '0' value, which will cause the `optimal-jump` to crash. Another limitation occurs with `n` values greater than '25', and a `d` list containing values up to or greater than '25', where it takes a substantial amount of memory and time to compute - as in it would take longer than 5 minutes and uses more than 8 gigabytes of RAM.

4 Credit

4.1 Help Received

- Matthew Brown: Code review and question discussion. We agreed to share test cases and benchmarks

4.2 Website resources

- HyperSpec
- Common Lisp Cookbook