Programming Assignment #7: Run Like Hell

COP 3503, Spring 2019

Due: Sunday, April 21, *before* 11:59 PM

Abstract

In this assignment, you will practice the art of dynamic programming. I think you will find this problem to have approximately the same level of difficulty as Fibonacci and the 0-1 Knapsack problem. It lends itself to an elegant recursive solution – albeit an inefficient one – that can be translated directly into a fast DP solution that only requires (at most) a 1D array.

This will be excellent practice for the final exam, where you are almost certain to encounter a DP problem of similar difficulty.

Deliverables

RunLikeHell.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

One peculiarly warm April day, while on a solitary hike through the Mountains of Obscure Knowledge, with nothing in his knapsack but a bottle of water, a package of pinwheel cookies, and his trusty copy of *The Algorithm Design Manual* by S.S. Skiena, a young algorithmist came upon a precipice. He approached with caution, and when he reached the ledge, he caught his breath, for there below him sprawled the most resplendent valley he had ever seen. Lush green foliage and glistening blue rivulets carpeted the valley floor, and towering cliffs encircled the vale (though none was quite so high as the algorithmist's precipice). From those cliffs flowed countless waterfalls that fed the crystalline streams below. Majestic, brightly colored birds swooped and whirled over the valley, and their clarion birdsong was mystery, and it was knowledge, and it was comforting and sweetly sad; it seemed to the algorithmist that theirs was the song of all things known and all knowledge yet to come. And at the very heart of the valley stood a tree near as tall as the mountain that the algorithmist had climbed, and from that tree dangled giant golden pears.

Enchanted, the algorithmist stopped to rest awhile.

As he gazed upon the lofty and unlikely fruits of the golden pear tree, thoughts of dynamic programming came unbidden to his mind. With surprising clarity, he recalled his studies of the topic, and his understanding began to crystallize in ways it never had before. The mysteries of optimal substructure unfurled before him like the wings of the birds below. He saw, in that moment, that the branches of an exponential recursive function were as the branches of the golden pear tree, tangled and unruly; that memoized results were as overripe, falling pears splatting juicily against lower limbs; and that the bottom-up technique of dynamic programming was as the seeds of a fallen pear coming to sprout anew. In his mind, all these approaches became one: many paths from the same source, all leading to the same goal, like the roaring waterfalls whose waters all wended toward the golden pear tree at the heart of the vale. As his insight grew, the rivulets in the valley below began to swell, until they had become a single body of water that gently drowned the valley floor. And so the young algorithmist was awakened to the beauty of DP.

It was then that the algorithmist felt the ground rumble beneath him. At first, he took the tremor for the earth-moving awesomeness of dynamic programming (and perhaps it was), but the dangerous reality of the matter quickly set in: the cliff he was sitting on was starting to crumble!

The algorithmist sprang to action. His only chance for survival was to run as fast as possible to more stable ground before the rocky ledge disintegrated beneath his feet. As he ran toward safety, new crevices formed everywhere he stepped. The cracks were following him, spreading through the rocky ledge at an alarming rate. He needed to get off that cliff altogether. He needed to...

RUN LIKE HELL!

While the algorithmist is running for his life, he passes beneath a row of evenly-spaced "obscure knowledge blocks." Each block is clearly labeled with the amount of obscure knowledge the algorithmist will gain for jumping up and hitting it. However, given the speed at which he is running, the algorithmist is unable to hit them all. If he jumps to hit one block, he is unable to jump back up in time to hit the very next block in the sequence (see Figure 1).

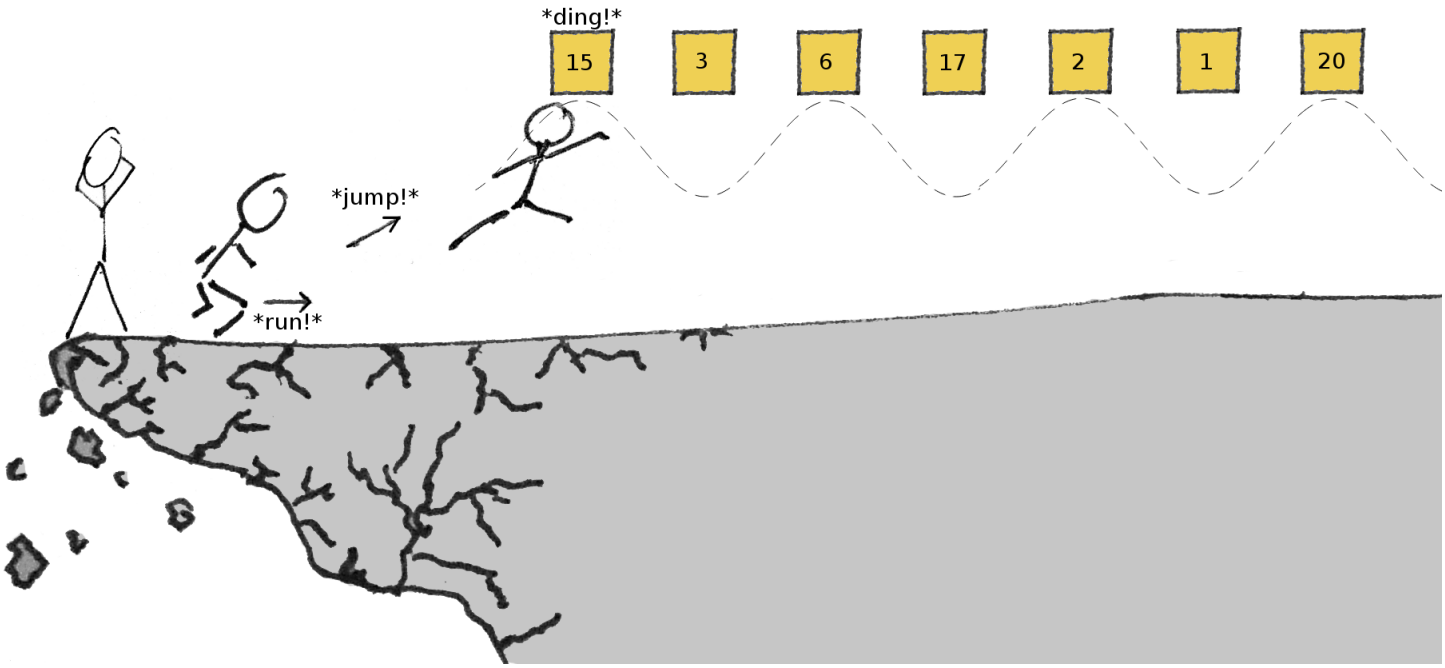

Figure 1: *Speed and trajectory prevent the algorithmist from hitting two consecutive blocks in the sequence.*

The goal of the algorithmist is to maximize the amount of knowledge he gains as he flees the mountain. Notice that in some cases, it is advantageous for the algorithmist to skip more than one block in a row. For example, Figure 2 shows the sequence of jumps that will optimize the algorithmist's knowledge gain for this particular sequence of blocks.

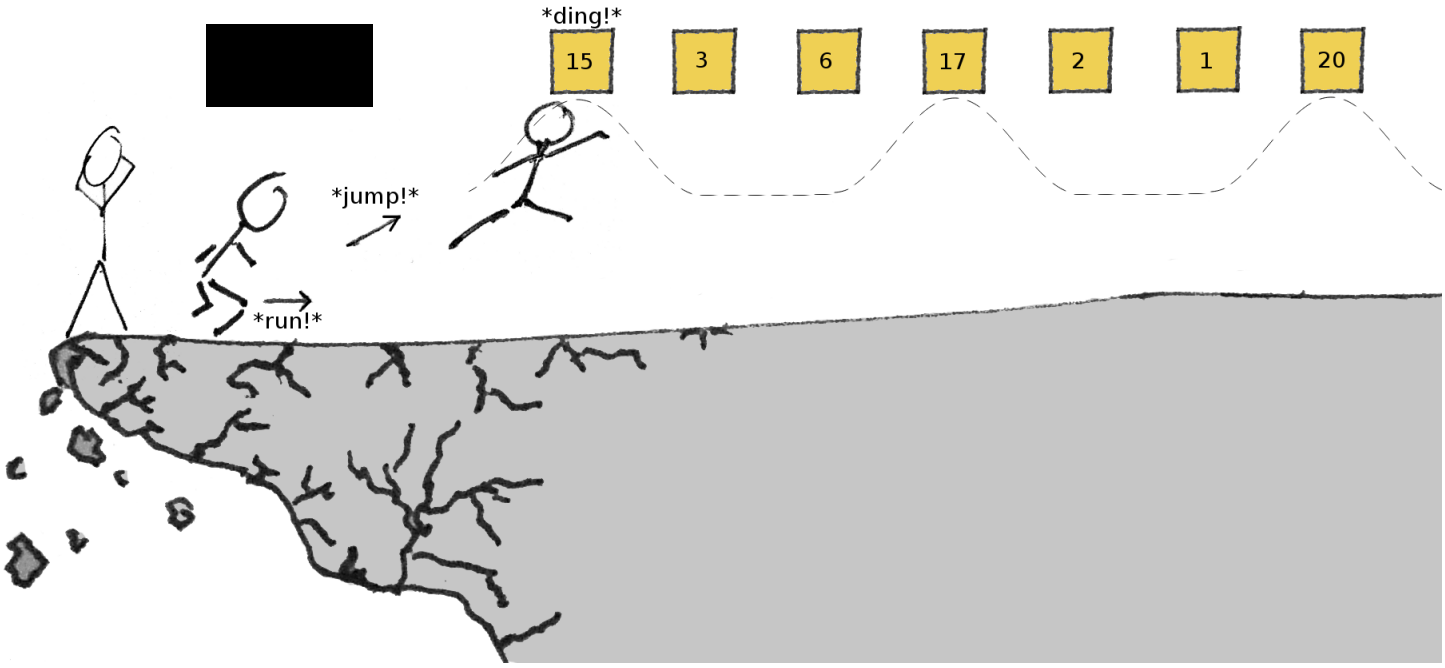

Figure 2: *For the path shown in this figure, the algorithmist's total knowledge gain is 15 + 17 + 20 = 52, which is the maximum knowledge gain possible for this particular sequence of blocks.*

In this assignment, the values of these blocks, ordered from left to right, will be given to you as an array of positive integers. For example, the array corresponding to the blocks in Figures 1 and 2 is:

```
int [] blocks = { 15, 3, 6, 17, 2, 1, 20 }
```

Your goal is to write a DP algorithm that will determine the algorithmist's maximum knowledge gain for any arbitrary sequence of blocks. (In the example above, the answer is 52.) For more examples, see the attached test cases, but please realize that those test cases are not comprehensive. You should also create some of your own.

Method and Class Requirements

Implement the following methods in a class named *RunLikeHell*. Please note that they are all ***public*** and ***static***.

`public static int` `maxGain(int [] blocks)`

> Given an array of positive integers containing the block values that the algorithmist will encounter (in order, from left to right), return the maximum knowledge the algorithmist can gain before making his way safely off the mountain. Recall that the algorithmist's speed and trajectory prevent him from ever hitting two blocks that are right next to each other.
>
> You *must* use an O(n) dynamic programming solution to receive credit. The space complexity must also be no worse than linear. I suggest starting with a top-down recursive algorithm and then transforming it into a bottom-up DP approach using the technique taught in class.

`public static double` `difficultyRating()`

> Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

`public static double` `hoursSpent()`

> Return a realistic estimate (greater than zero) of the number of hours you spent on this assignment.

Compiling and Testing on Eustis (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. To compile your program with one of my test cases:

```
javac RunLikeHell.java TestCase01.java
```

2. To run this test case and redirect your output to a text file:

```
java TestCase01 > myoutput01.txt
```

3. To compare your program's output against the sample output file I've provided for this test case:

```
diff myoutput01.txt sample_output/TestCase01-output.txt
```

If the contents of *myoutput01.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

```
seansz@eustis:~$ diff myoutput01.txt sample_output/TestCase01-output.txt
seansz@eustis:~$ _
```

Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

4. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *RunLikeHell.java* and all the test case files and typing:

```
bash test-all.sh
```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

1. Style Restrictions (Same as in Program #1) (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two

operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do <u>not</u> have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

2. Grading Criteria and Miscellaneous Requirements

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

100%	Passes test cases using an O(*n*) dynamic programming solution.

Important Note! Additional point deductions may be imposed for poor commenting and whitespace. Significant point deductions may be imposed for violating the style restrictions listed above. You should also still include your name and NID in your source code.

All the policies stated in the previous programming assignment specifications still apply:

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! Programs that do not compile on Eustis will receive zero credit. When testing your code, you should ensure that you place *RunLikeHell.java* alone in a directory with the test case files (source files, the *sample_output* directory, and the *test-all.sh* script), and no other files. That will help ensure that your *RunLikeHell.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

Important! You might want to remove *main()* and then double check that your program compiles without it before submitting. Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! Your program should not print anything to the screen. Extraneous output is disruptive to the grading process and will result in severe point deductions. Please do not print to the screen.

Important! **No file writing.** Please do not write to any files from *RunLikeHell.java*.

Important! **Please do not create a java package.** Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! **Name your source file, class(es), and method(s) correctly.** Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to implement a required method, or failing to make certain methods public, private, static, and/or non-static (as required), may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statement when creating the test cases that we'll use for grading. Please reflect carefully on the kinds of edge cases that might cause unusual behaviors for any of the methods you're implementing.

Start early! Work hard! Ask questions! Good luck!