Programming Assignment #8: Run Like Hell

COP 3503, Fall 2013

**Due:** Saturday, December 7, 11:59 PM via Webcourses@UCF

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.

Deliverables
RunLikeHell.java

run!
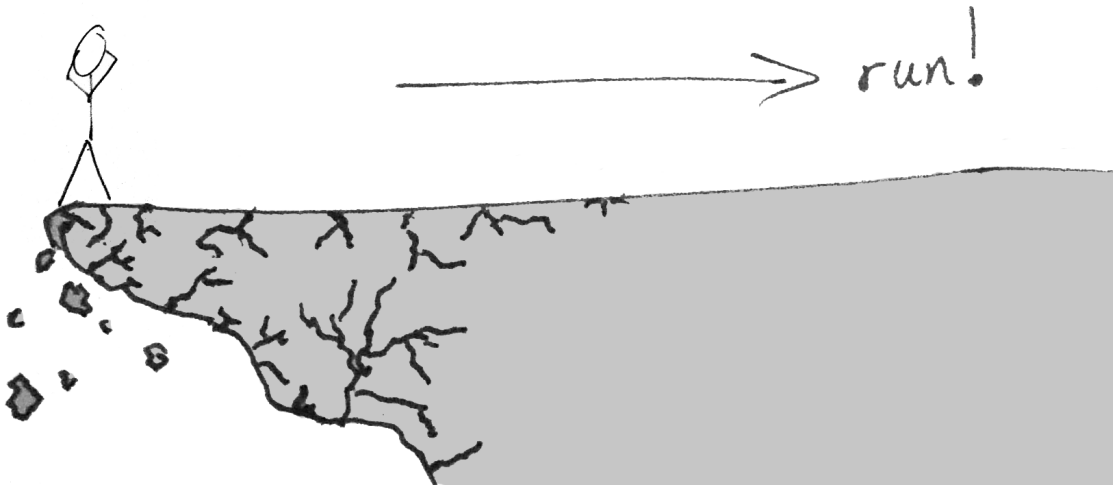
One crisp December day, while on a solitary hike through the Mountains of Obscure Knowledge, with nothing in his knapsack but a bottle of water, a package of pinwheel cookies, and his trusty copy of *The Algorithm Design Manual* by S.S. Skiena, a young algorithmist came upon a precipice. He approached 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).
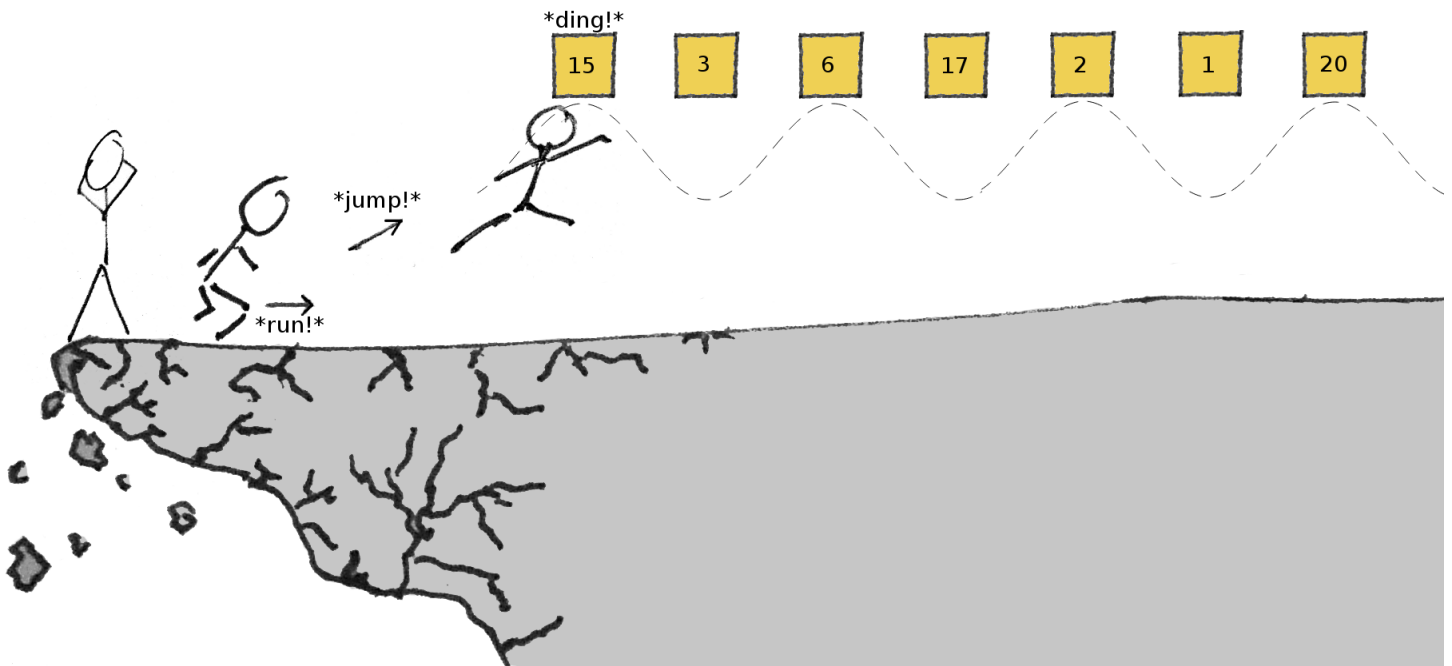
Figure 1: *Speed and trajectory prevent the algorithmist from hitting two consecutive blocks in the sequence.*

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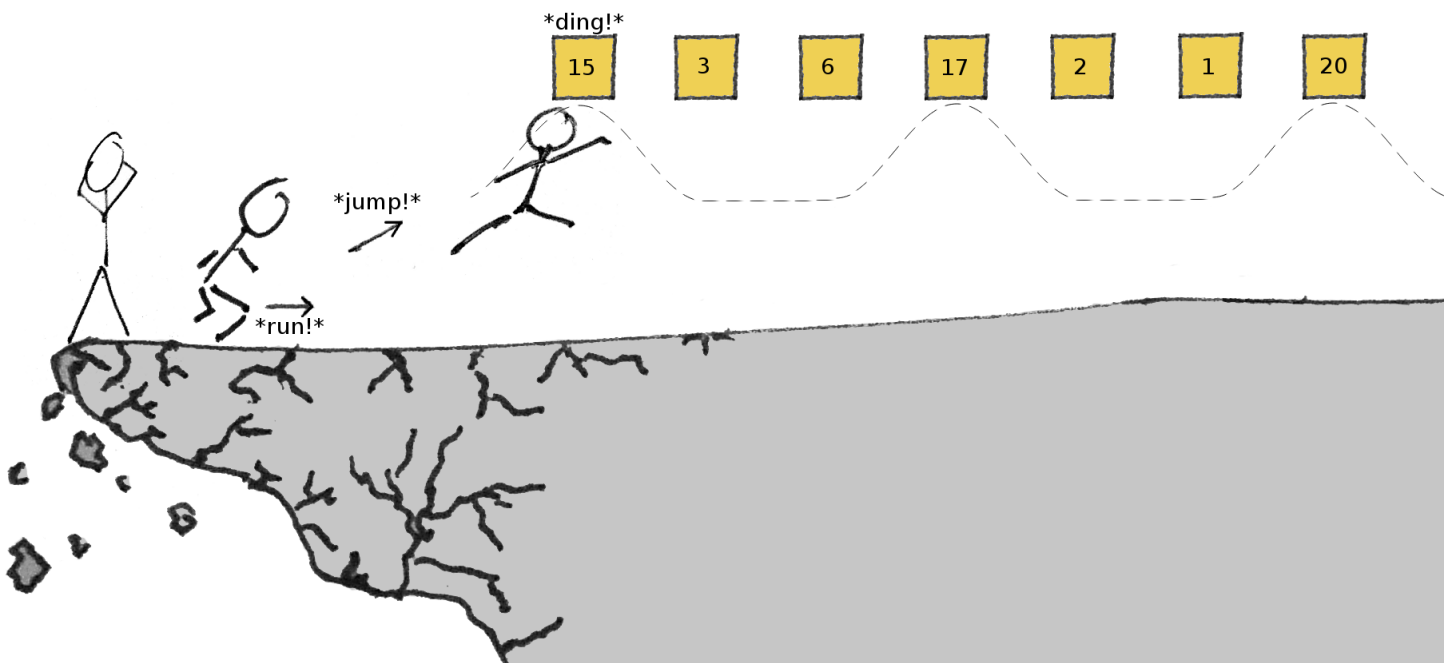
Figure 2: *For the path shown in this figure, the algorithmist's total knowledge gain is 15 + 17 + 20 = 52, which is the maximum knowledge gain possible for this particular sequence of blocks.*

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.) See `TestCases.java` for more 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 **static**. Before submitting, please be sure to test your code with the `TestCases.java`, which will show you exactly how I intend to call your methods when I test your program.

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. 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 on the range 1.0 (ridiculously easy) to 5.0 (insanely difficult).

public static double hoursSpent()

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

Grading Criteria and Miscellaneous Requirements

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

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

You still must include adequate comments and whitespace. Also, all policies stated in the previous programming assignment specifications still apply:

- Programs that do not compile will receive zero credit.
- Please remove `main()` before submitting.
- Please do not print anything to the screen during program execution.
- Please do not create a java `package`.
- Please name your methods, class, and source file correctly.
- Please test your code thoroughly with `TestCases.java`, as well as with your own test cases.

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