Algorithms in Java and Python:
The Triangle Route Problem

Nicholas Bennett
Grass Roots Consulting
nickbenn@g-r-c.com

17 October 2010

Copyright and License Information

Table of Contents

This page intentionally left blank.

1. Introduction

In this lesson, we'll develop an efficient method for finding the least expensive route over a somewhat peculiarly shaped network. Then, we'll add an implementation of that method to a program already written in Java, and to one written in Python. Finally, we'll use the completed programs to solve an instance of the routing problem that – because of its magnitude – would be impossible to solve without an efficient approach.

Definition and a simple example

In a triangular arrangement of numbers, such as the one in Figure 1, a *triangle route* is a path starting at the *apex* (top) and ending at the base, moving to an adjacent number in the next row down at each step. In this case, all routes start with **3**, then move either to **6** or **5** in the second row. From **6** in the second row, the next step would be to **5** or **4** in the third row; from **5** in the second row, the valid steps are to **4** or **3** in the third row.

```
         3
      6     5
   5     4     3
5     6     8     9
```

Figure 1 - Example triangular arrangement of numbers

Obviously, moving from the apex to the base of Figure 1 requires 3 steps, and any route will touch 4 numbers along the way (including the starting and ending numbers). Since we have 2 alternatives at each step, and there are 3 steps required, the number of possible routes is $2 \times 2 \times 2 = 2^3 = 8$. If we add a row to the triangle, the number of steps increases by 1, and the number of possible routes is doubled.

To solve the triangle route problem, we need to find the triangle route with the smallest sum of the numbers along the route, including the starting and ending points. This is an example of a *combinatorial optimization* problem.[1] One way to solve the problem is to calculate the sum for every possible route, and select the route with the smallest sum. This approach – examining every possible solution to a problem, to find the best one – is called an *exhaustive search*.

1 In this problem, the *decision space* (the set of all possible steps that can be taken) is *discrete* – i.e. a solution isn't allowed to include fractional or partial decision values. This type of problem – finding the best combination of decisions in a discrete decision space – is called a *combinatorial optimization* problem. Combinatorial optimization problems appear in many real-world situations, but they're especially common in resource allocation and network problems.

Pencil-and-paper task 1: Find the minimum-cost triangle route

Using any method you like, find the triangle route with the smallest sum for the data shown in Figure 1.[2]

Solution

You probably didn't have much trouble solving the problem. Even if we do an exhaustive search of all routes, it's doesn't take long to see the lowest sum is 18, along the following route:

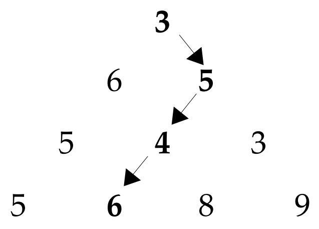


Figure 2 - Least-cost triangular route

From this point on, we'll refer to the numbers in the triangle as *costs*, and the sum of numbers found along a route as the *cost* of the route. The route with the smallest cost is the *minimum-cost* or *least-cost route*. Finally, besides using arrows, we can identify a triangle route by the sequence of directions ("L" and "R", for left or right, respectively) followed in the steps of the route; since the apex is always the start of the route, we can leave it out. Using this terminology and notation, the least-cost route shown in Figure 2 is **RLL**, with a cost of 18.

Too many routes?

As we saw above, the number of possible routes in the triangle route problem doubles with each row added to the triangle. This type of growth (multiplication by a constant term at each step) is called *exponential growth*, and it doesn't bode well for a solution approach that uses an exhaustive search.

2 Note that the problem which served as inspiration for this lesson has the opposite objective [1]: in that problem, the goal is to find the largest sum, rather than the smallest. In fact, the concepts explored here apply equally to both objectives.

How bad is the combinatorial explosion[3] for the triangle route problem? Let's look at an example: if our triangle has 100 rows, then the number of steps is 99, and the number of possible routes is 2^{99}, or approximately 6.34×10^{29}. If we had a computer that could evaluate one trillion possible routes every second, it would still need about 20 billion years to evaluate them all.

Let's try to come up with an *algorithm* (a step-by-step procedure for solving a problem, or completing a specific task) that we can use with pencil and paper, for the case of a triangle with 8 or so rows. If we're successful, maybe that algorithm can also be used by a computer for a triangle with 100 rows – or more!

Discussion: Approaches for solving larger cases

By now, it should be clear that searching through all possible triangle routes isn't a very good way to find the best route, especially if the triangle is large. Read and discuss the following questions, and see if you can come up with an alternative approach (i.e. one that doesn't use an exhaustive search) for solving the triangle route problem.

- What algorithms (or less formal methods) did you use for task 1?
- Can you describe your reasoning in detail?
- How could your methods be applied to a larger triangle?
- If we reverse the direction of travel, by starting the routes at the base of the triangle and traveling to the apex, would that change the costs of the routes? Could it result in a different route being the least-cost route?
- Do you think that knowing the least-cost route through a section of the triangle would help us find the least-cost route through the whole triangle?

3 In combinatorial optimization, when the size of the *solution space* (the set of possible solutions) grows much faster than the number of decision values, we say the solution space has a *combinatorial explosion*.

Consider the problem of assigning 4 teachers to 4 different classes. The number of possible combinations is $4! = 4 \times 3 \times 2 \times 1 = 24$. We could easily evaluate all 24 alternatives to find the best one. But what about 40 classes and 40 teachers? In that case, the number of possible combinations can be as high as $40!$, or about 8.16×10^{47}. Of course, many combinations would be excluded from the solution space as invalid (e.g. we might not want to allow combinations where English teachers are assigned to teach math classes). Nonetheless, it's not hard to formulate an assignment problem that's too large for even the fastest computer to evaluate all of the allowed solutions in a reasonable amount of time.

This page intentionally left blank.

2. *A More Efficient Method*

Solving from the bottom up

```
               11
            6       5
        8      17      13
    3       7       8       7
15     11       5      13      14
19      6      17      18       5      10
```

Figure 3 - Example triangle with six rows

The triangle in Figure 3 has 6 rows; therefore, all its triangle routes are 5 steps long, and there are $2^5 = 32$ possible triangle routes. However, inside the triangle are many smaller triangles – some of which have their bases along the bottom row of the larger triangle. We can take advantage of this by building our triangle routes from bottom to top, working from smaller to larger triangles.

Let's start with the second-to-last row – the one with the cost values **15**, **11**, **5**, **13**, and **14**. Each of these numbers is located at the apex of a triangle of 3 numbers – the apex itself, and the 2 values directly below. For example, **15** is at the apex of a triangle with **19** and **6** on its base; to find the least-cost route for just that triangle, we only need to step from **15** to the smaller of the two values below it (i.e. **6**) and we have a route with a cost of **21**. We can mark that step with an arrow from **15** to **6**; we'll also replace the number **15** by the number **21**, to show that the minimum cost from here to the base, once a route arrives to this point, is **21**. If we work across that row, finding the minimum-cost routes for each of the small triangles, we get the result in Figure 4.

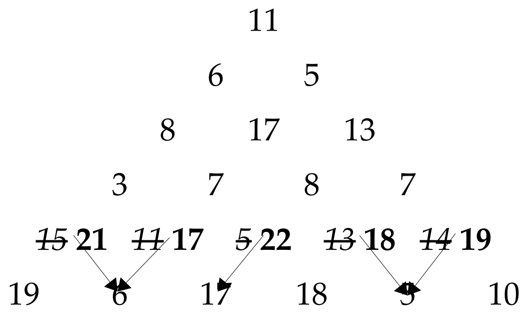


Figure 4 - Finding least-cost routes from second-to-bottom row to base

Now let's move up one row, to the cost values **3**, **7**, **2**, and **7**. Each of these numbers is the apex of a triangle of 3 numbers, and we've already identified the least-cost routes from the 2 numbers below the apex. For example, **3** has the values **21** and **17** below it; these are the costs of the best routes that start at those locations; obviously, the least-cost route starting with the **3** must step to the **17**, with a total cost of $17+3 = 20$. We complete the rest of the row in the same fashion (see Figure 5).

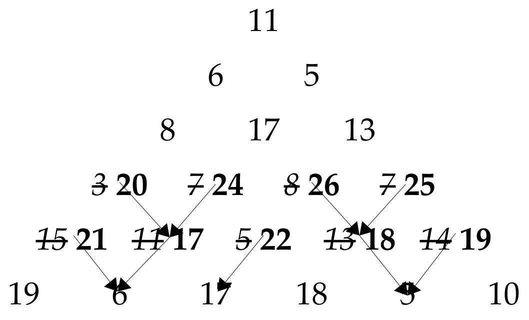


Figure 5 - Backtracking to find least-cost routes from next higher row

We can continue building the least-cost route in this backtracking fashion, moving up row by row, until we reach the apex of the largest triangle. At this point, the least-cost route for the entire triangle can be found by reversing direction and following the arrows from the apex to the base; there will be only one such route. (If desired, and if we've been drawing the arrows in pencil, we can erase those that aren't on the least-cost route.)

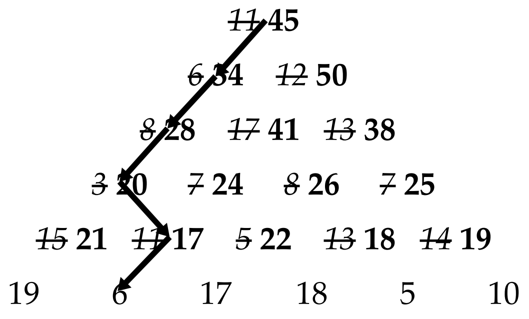


Figure 6 - Least-cost route from apex to base

The least-cost triangle route for the triangle in Figure 3 is thus **LLRL**, with a cost of 45 (see Figure 6).

To apply this approach to other triangle route problems, it would be useful to have a more formal, written description of the algorithm that we're following. In the description that follows, please note the use of indentation and order to indicate the structure and flow of the algorithm.

Bottom-up algorithm for finding the least-cost triangle route

- Starting with the second-to-bottom row, and moving up row by row to the apex:
    - Starting with the left-most item in the current row, and moving one by one across to the right-most item:
        - $c_C =$ the current item.
        - $c_L =$ the item below and to the left of the current item.
        - $c_R =$ the item below and to the right of the item.
        - If $c_L \leq c_R$:
            - Draw an arrow from c_C to c_L.
            - Replace c_C with $c_C + c_L$.

          Otherwise:
            - Draw an arrow from c_C to c_R.
            - Replace c_C with $c_C + c_R$.
- The least-route is found by following the arrows from the apex to the base.
- The value now at the apex of the triangle is the minimum triangle route cost.[4]

4 Of course, this isn't the only way to express the algorithm. For instance, we could use *pseudocode* (see Appendix B).

Pencil-and-paper task 2: Using the bottom-up algorithm

Using the bottom-up algorithm (above), solve this triangle route problem.

```
                              4
                         9         8
                    8        13        13
               5         8        12         5
          19        10        15         8        11
     7         13        12        10        15         7
14        13         6        10        12        13         9
     9         7        15         4         2         5         7        12
```

Figure 7

Dynamic programming

We can prove that our algorithm works (i.e. it finds the least-cost route) quite simply: Using the definition of a triangle route with the triangle in Figure 8, we know that any route from ***A*** to the base must pass through ***B*** or ***C***; thus, we know that the least-cost route from ***A*** to the base must include either the least-cost route from ***B*** to the base, or the least-cost route from ***C*** to the base, based on which one has the smaller cost. Similarly, the least-cost route from ***B*** to the base must include one of the least-cost routes from either ***D*** or ***E*** to the base – and so on, all the way down.

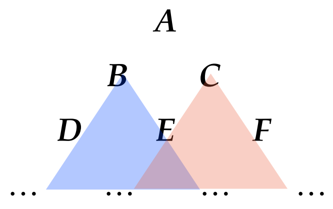


Figure 8 - Illustration of optimal substructure

Working from the bottom up, finding the least-cost route for each sub-problem, and computing its cost, involves comparing 2 numbers and adding 2 numbers. In a triangle with n rows, we do this $n(n-1)/2$ times, vs. the 2^{n-1} route evaluations and comparisons required by an exhaustive search. Clearly, our current algorithm is much more efficient than an exhaustive search [2].

When the solution to a problem can be constructed efficiently from the solutions of its sub-problems, we say that the problem has *optimal substructure*. This is one of the conditions for using *dynamic programming* – a set of problem-solving techniques in mathematics and computer science, where a problem is defined (and solved) in terms of its sub-problems [3]. (When the sub-problems don't overlap, the techniques are usually referred to as *divide and conquer*, rather than dynamic programming [4].)

Notwithstanding the name, dynamic programming isn't necessarily dynamic, and it doesn't necessarily involve computer programming. “Dynamic” was originally used because it sounded impressive, and “programming” refers to the optimization of plans of action – i.e. programs. In any event, as the term is currently used, it applies to our algorithm.

The dynamic programming approach we've been using is called a *bottom-up* method – not because the algorithm starts close to the base of the triangle, but because it starts with the smallest sub-problems, and systematically builds solutions to larger and larger sub-problems. There are also dynamic programming approaches that start with the original problem and move to smaller sub-problems; these are called *top-down* methods [3]. A top-down approach often follows directly from the definition of a problem in terms of its sub-problems (for example, the logic used in the proof above follows a top-down approach). On the other hand, a bottom-up algorithm is often a more straightforward way to arrive at a solution.

What's it good for?

So our algorithm is efficient – but what good is it, really? After all, finding a route with the lowest sum through a triangle of numbers doesn't seem like the most practical problem.

In fact, the triangle route problem can be seen as a network routing problem – albeit one with a rather unusually shaped network. If we rotate Figure 1 by 90 degrees counterclockwise, and treat each number's logical position as a node on a network, and the number itself as the *weight* (a general term from network theory, used to represent cost, distance, time, etc.) of the edges leading into that node, the network routing problem is clear: find the minimum-weight route from **Start** to **Finish** (see Figure 9).

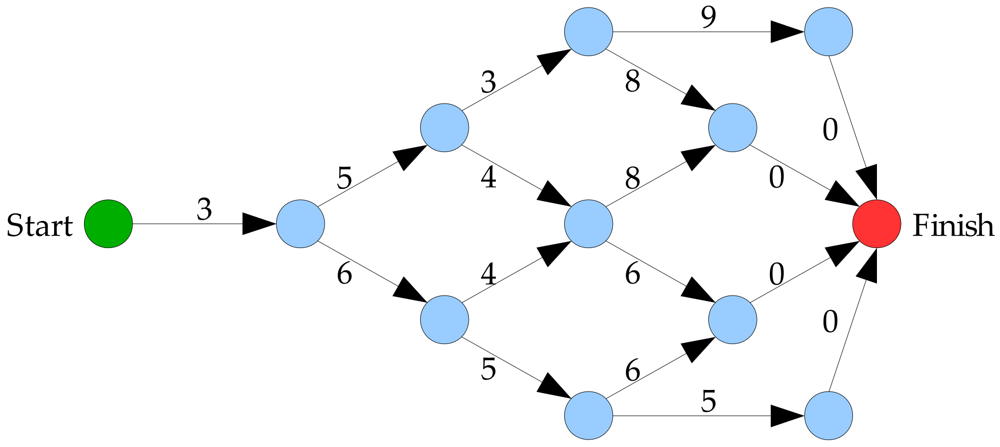


Figure 9

With some adjustments, the algorithm we're using for finding the least-cost triangle route could also be used to find the minimum-weight route over a much more general network. However, we'll stick to triangle routes for this lesson.

3. General Implementation Features

Introduction

As noted earlier, an exhaustive search for the least-cost route through a triangle with 100 rows is practically impossible. However, because our new algorithm is much more efficient, we'll try something ambitious: after implementing the algorithm in Java and Python, we'll use it to find the least-cost route for a triangle with 500 rows!

For this lesson, we won't be writing our programs from scratch: the code for reading the triangular arrangement of costs from a data file, and for keeping track of the cost and steps in a route has already been written for us. In fact, even the code for the algorithm is included here, but we'll be transcribing it from this document into our programs; by doing so, and reviewing what we've written, we'll start to see how key concepts are expressed in these two programming languages, and we'll get a taste of the details we need to pay attention to when programming – and the problems that can happen when we don't.

Data layout

The data file and in-memory data structures our program use don't look exactly like the triangles we've seen so far – but they're not all that different. If we shift our triangles to the left, the triangle in Figure 8 becomes the triangle in Figure 10; the latter's a more accurate picture for how the data will be stored and managed. So a step down and to the left needs to be handled in our code as a step straight down.

A			
B	***C***		
D	***E***	***F***	
…	…	…	…

Figure 10 - In-memory arrangement of triangle costs

Development Environment

The remainder of the lesson assumes we're writing our code with the NetBeans *integrated development environment* (IDE – a program for editing, running, and debugging code) [5]. NetBeans is used primarily for Java development, but it can also be used with Python and other languages. With minor adjustments, these materials can be used with almost any Java 1.5/5.0+ or Python 2.5.x-2.7.x (but not Python v3.x) tool set.

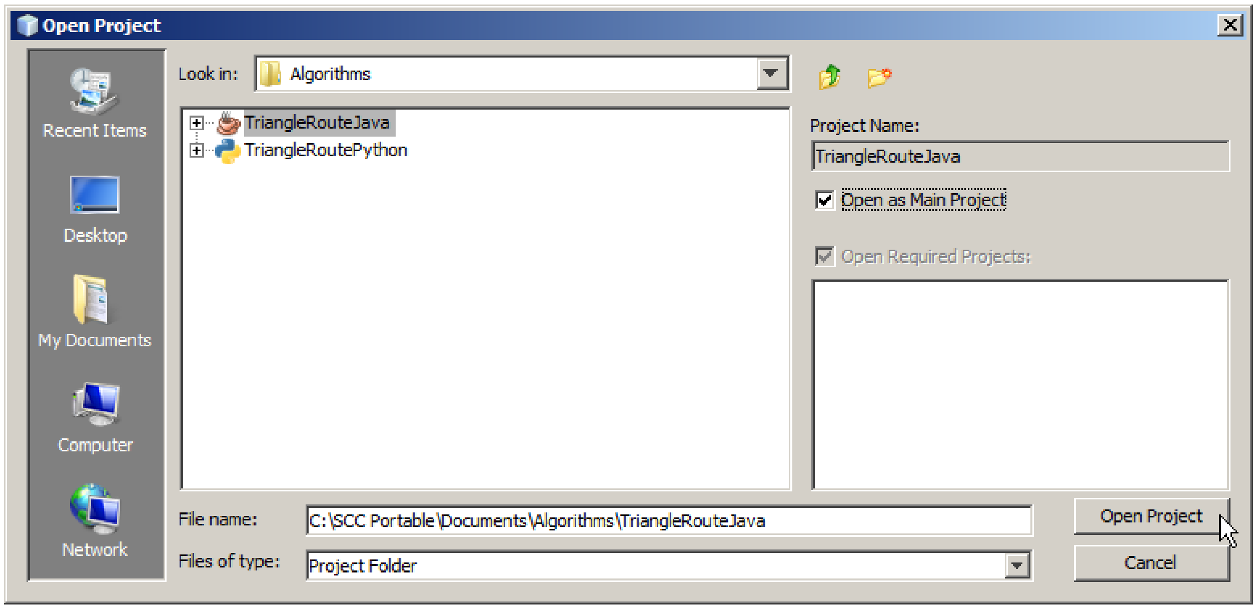
This page intentionally left blank.

4. Java Implementation

Open the Java project

Launch the NetBeans application. Depending on how it's being run (e.g. from a flash drive vs. a hard drive), and the configuration of your computer, it can take a while to start up, so be patient.

Once in NetBeans, if you haven't previously opened the `TriangleRouteJava` project, you'll need to do so now. To open a project, select the **File/Open Project…** menu command. In the **Open Project** window (see Figure 11), navigate to the directory containing the materials for this lesson, select `TriangleRouteJava` (it may have a coffee cup icon next to it), check the **Open as Main Project** checkbox, and click the **Open Project** button.



Look in: Algorithms

TriangleRouteJava
TriangleRoutePython

Project Name: TriangleRouteJava

[✓] Open as Main Project

[✓] Open Required Projects:

File name: C:\SCC Portable\Documents\Algorithms\TriangleRouteJava

Files of type: Project Folder

Figure 11 - Opening TriangleRouteJava project

After the project opens, you should see `TriangleRouteJava` listed in the NetBeans **Projects** view (Figure 12), with a number of folders below it.

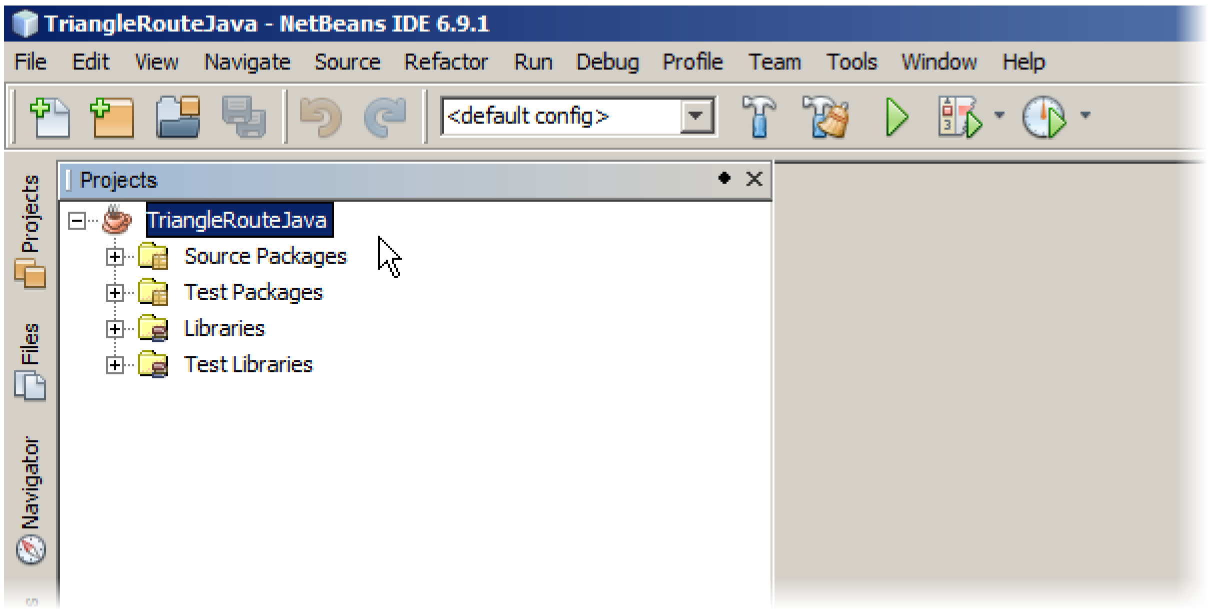

TriangleRouteJava - NetBeans IDE 6.9.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
Files
Navigator
TriangleRouteJava
Source Packages
Test Packages
Libraries
Test Libraries

Figure 12 - Structure of project contents

All of the Java code we'll write will be in one file – one that already exists in the project – so let's open that file now as well.

In the **Projects** view of NetBeans, expand the folder of the `TriangleRouteJava` project, then the `org.nm.challenge.kickoff.triangleroute` package, and locate the file `Optimizer.java` (Figure 13).

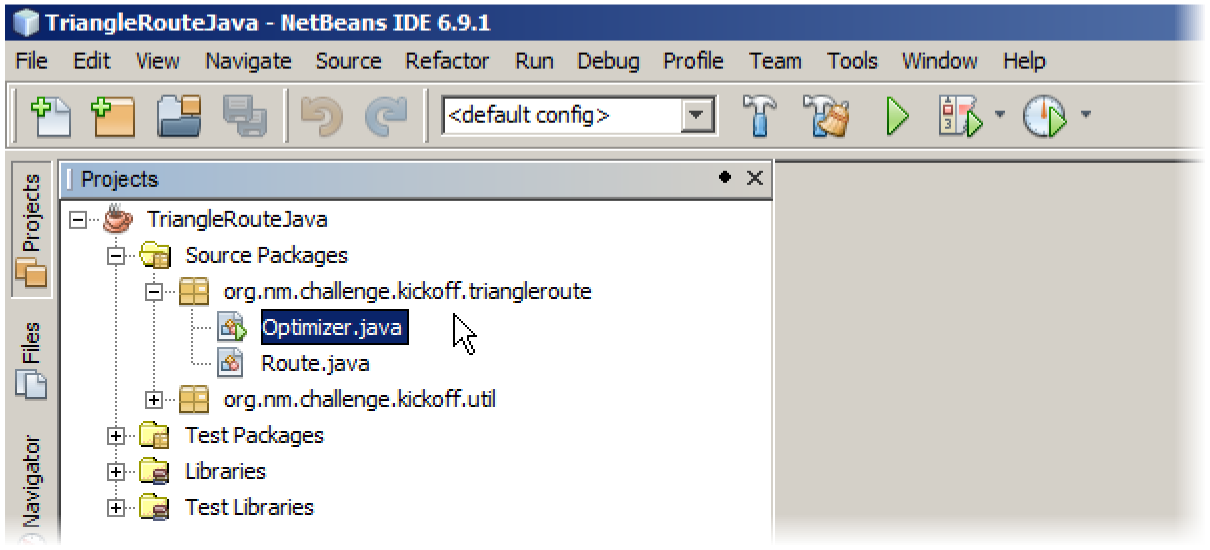
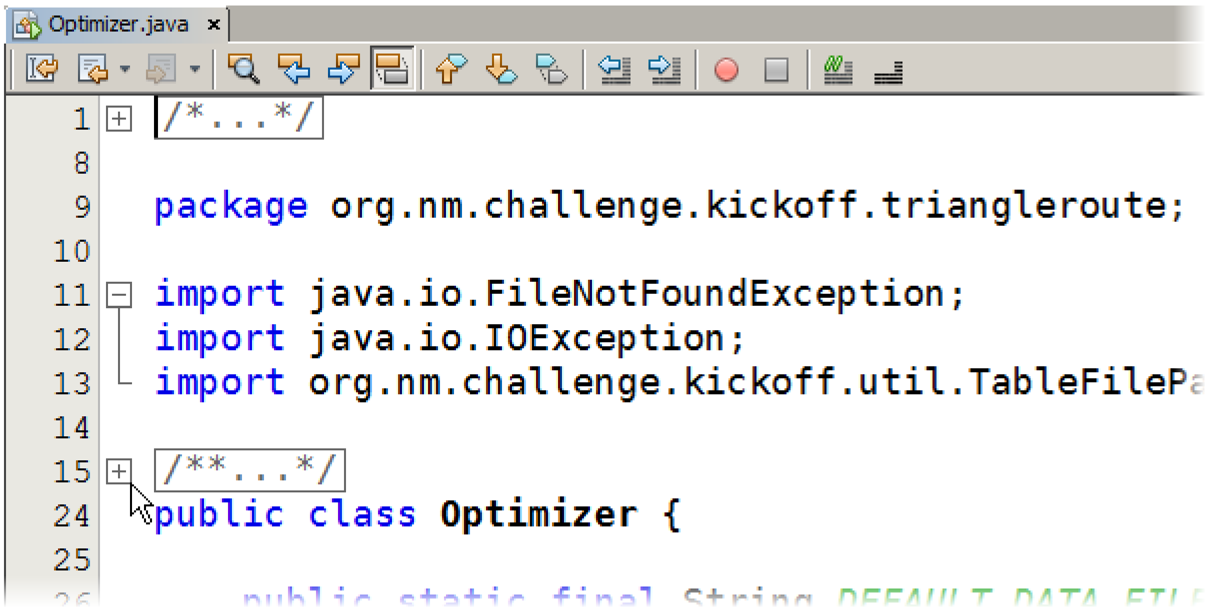

TriangleRouteJava - NetBeans IDE 6.9.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
Files
Navigator
TriangleRouteJava
Source Packages
org.nm.challenge.kickoff.triangleroute
Optimizer.java
Route.java
org.nm.challenge.kickoff.util
Test Packages
Libraries
Test Libraries

Figure 13 - Locating **`Optimizer.java`** file

Double-click `Optimizer.java` to open that file in the editor. Depending on your NetBeans configuration, you should see line numbers along the left side of the editing window. You might also see that some numbers seem to be skipped; when you notice that, also note that there's a an expandable control (a plus sign) to the right of the line number just before the skipped lines. (If no lines are skipped, don't worry about it.)



```
/*...*/

package org.nm.challenge.kickoff.triangleroute;

import java.io.FileNotFoundException;
import java.io.IOException;
import org.nm.challenge.kickoff.util.TableFileP

/**...*/
public class Optimizer {

    public static final String DEFAULT_DATA_FILE
```

Figure 14 - Code folding of initial comment and documentation comments

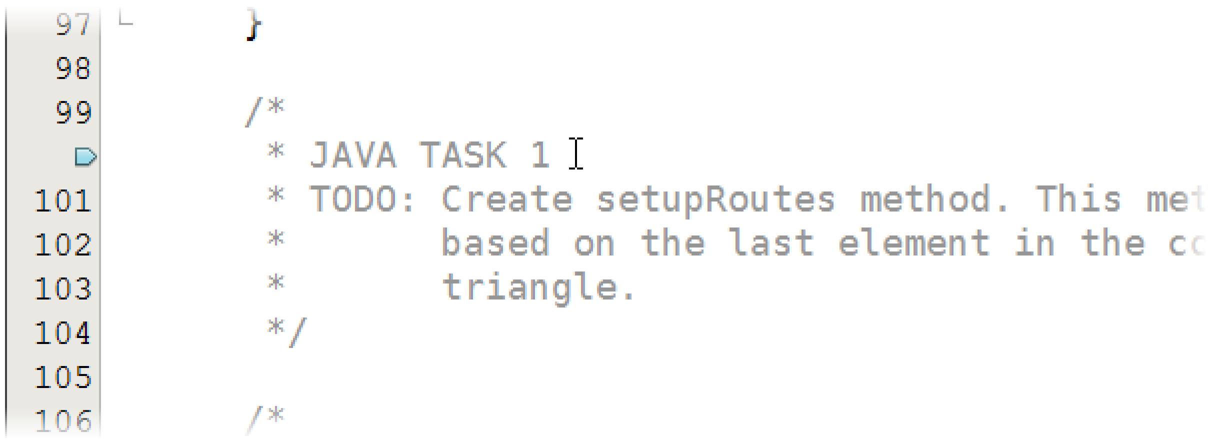
Like many programming editors, NetBeans can be configured to perform *code folding* – i.e. temporarily collapse some portions of the source code to make it easier for the programmer to navigate through a file. In this example, NetBeans is folding certain kinds of *comments* (non-executable code) by default. In any event, we probably won't need to worry about code folding for now.

The file we've opened, `Optimizer.java`, contains the definition of the **`Optimizer`** *class*. In Java – as in many other program languages – a class is a construct for packaging together data and *methods* (functionality) common to all objects of that type. In this case, objects of the **`Optimizer`** type will hold the cost values for a triangle that are read from a file, and they'll have methods that use an implementation of the bottom-up algorithm to find the least-cost route for that triangle.

Some Java classes also have a special method (**`main`**) that acts as a starting point for running a Java *application* (program). An application might use dozens of classes, but there has to be at least one class with a **`main`** method, so that the Java runtime environment (JRE) knows which code should be started when the application runs. Fortunately, **`Optimizer`** has a **`main`** method already written for us.

Java task 1: `setupRoutes`

Around line 99, you'll see a comment marking the location of the first code we'll write:



```
 97         }
 98
 99         /*
             * JAVA TASK 1
101          * TODO: Create setupRoutes method. This met
102          *       based on the last element in the co
103          *       triangle.
104          */
105
106         /*
```

Figure 15 - Placeholder comment for Java task 1

As seen in Figure 15, you might see a small blue marker instead of a line number. This is a bookmark; NetBeans lets us use bookmarks to set navigation points in a file.

Our first task, as the comments indicate, is to write a **`setupRoutes`** method that will build the first set of routes from the bottom row of the triangle. In the code provided to us, the costs in the triangle are stored in an array called **`costs`** – actually, **`costs`** is an array of arrays: each element in costs corresponds to a row of the triangle, and is itself an array of numbers, containing the cost values for a single row. **`costs`** is defined as an *instance variable* of the **`Optimizer`** class – i.e. all **`Optimizer`** objects (*instances*) have their own **`costs`** array.

Instead of modifying the values in the triangle, as we did on paper, we'll use a different kind of object, a **`Route`** (which has been provided for us), to keep track of the least-cost route from the current point in the triangle to the base. A **`Route`** instance can be created with just a starting cost, or based on another **`Route`** instance. In **`setupRoutes`**, we'll create initial routes that have no steps, with costs equal to the values in the bottom row.

Create the **`setupRoutes`** method by typing (below the placeholder comment) the code seen in Listing 1. Note that you don't need to re-write the comment; it's shown here to help you see where to add the new code.

As you type the code, please note the following:

- Parentheses, square brackets, and curly braces have different meanings in Java, and are not interchangeable.

- Java is *case-sensitive:* capitalization (or lack of same) matters; in particular, all Java keywords are lower-case.
- All simple statements (but not all lines) in Java must end with a semicolon; all that appear in this code are required.
- Like many other programming editors, NetBeans has certain code completion features that are enabled by default. For example, when you type a left curly brace and press the **Enter** key, NetBeans creates a blank line, and follows that with another line containing an automatically generated matching right curly brace. If you then type another right curly brace, you'll have mismatched braces, which will cause a syntax error. NetBeans handles parentheses, square brackets, and single and double quotes in a similar way, but you generally don't even need to press the **Enter** key for NetBeans to generate the matching character in those cases.

```
/*
 * JAVA TASK 1
 * TODO: Create setupRoutes method. This method must fill the routes array,
 *       based on the last element in the costs array (i.e. the base of the
 *       triangle.
 */
private void setupRoutes() {
    int[] finalStepCosts = costs[costs.length - 1];
    routes = new Route[finalStepCosts.length];
    for (int i = 0; i < finalStepCosts.length; i++) {
        routes[i] = new Route(finalStepCosts[i]);
    }
}
```

Listing 1 - **`setupRoutes`** method

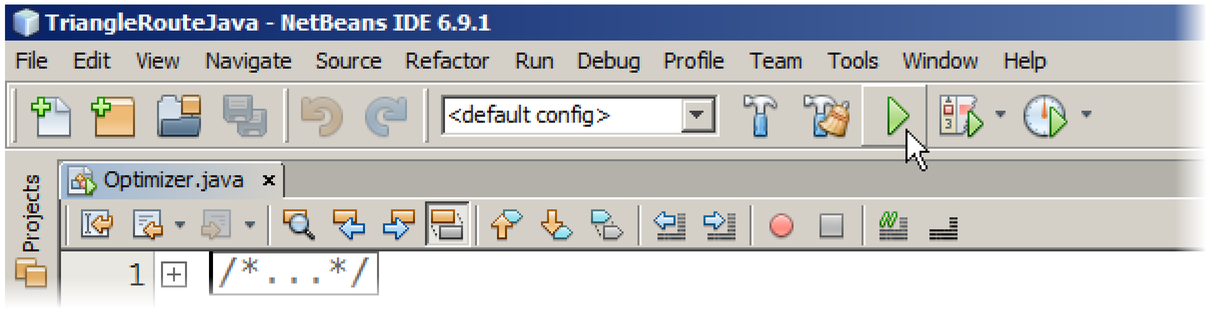
What does the **`setupRoutes`** method do? Let's look at the key elements briefly:

- The **`private`** access modifier keyword makes this method invisible to code outside **`Optimizer`**.
- All Java method declarations must state the type of data that the method will return to the *caller* (the code calling the method). The return type used here, **`void`**, indicates that no data is returned by this method.
- The name of the method is **`setupRoutes`**, and the empty parentheses that follow the method name declare that this method takes no *parameters* (additional data passed to a method when it's invoked). The name and parameter list, along with the access modifier and return type, make up the *signature* of a method.

- The left curly brace following the method signature starts the *block statement* (a groups of statements within curly braces; a method's block statement is also called the *method body*) that implements the method's logic; the right curly brace six lines below that ends the block statement.
- Methods and other block statements can include *local variables* – variables used only within the enclosing curly braces. We've declared the local variable **`finalStepCosts`** as a reference to an **`int[]`** (array of integers), and set it to refer to the array that holds the values from the last row of the triangle.
- The **`Optimizer`** class has a variable called **`routes`**, which is a reference to an array of **`Route`** objects; however, it doesn't refer to an actual array initially. In **`setupRoutes`**, we're making it refer to a new array of **`Route`** objects, which has as many elements as the number of elements in **`finalStepCosts`**.
- The **`for`** iteration statement is a workhorse of many programming languages. Among other things, it's used to perform some operation for each element of an array. Here, we iterate over the **`finalStepCosts`** array, creating a new **`Route`** object for each element and storing that object in the **`routes`** array.

Our code still isn't doing much, but we should make sure that it doesn't have any syntax errors. (We won't be able to check for logical errors until we write the code that invokes the **`setupRoutes`** method.) By default, NetBeans checks our code as we write, and it compiles the code as soon as we save it, so go ahead and save the file (the **File/Save** menu option, or the **Ctrl-S** key combination). Look for any red underlines under the code you've written, or red markers in the left margin; if you hold your mouse over a red line or marker, a pop-up will give you information about the error found. (At this point, any errors you'll see will most likely be caused by spelling or punctuation mistakes.) Starting with the error closest to the top of the file and working down, try to use the error messages to figure out and correct the problem. When all reported errors have been corrected, save the file again.

Besides compiling, we can actually run our program. Since the class we're working on (**`Optimizer`**) has a **`main`** method, we can run it as a Java application with the **Run/Run File** menu command. Alternatively, since we set the **TriangleRouteJava** project to be the main project when we opened it, and since **`Optimizer`** was already configured as the main class in the project, we can click on the green triangular button in the toolbar near the top of the NetBeans window (see Figure 16); that button runs the **`main`** method of the main class in the main project.

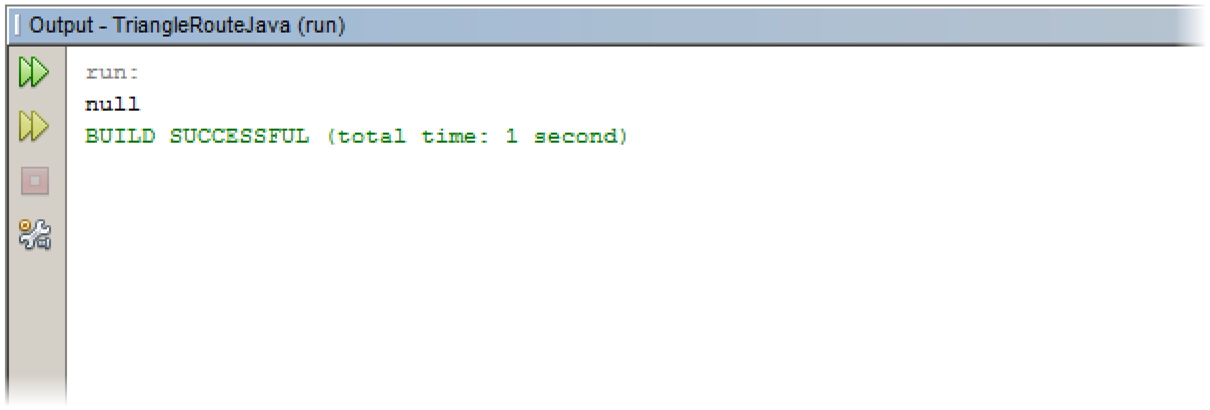

TriangleRouteJava - NetBeans IDE 6.9.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
Optimizer.java

```
1 /*...*/
```

Figure 16 - The “Run main project” button

When the program runs, the results are displayed in the **Output** panel, usually located at the bottom of the screen. The **`Optimizer.main`** method (i.e. the **`main`** method of the **`Optimizer`** class) is programmed to load the triangle data from a file, solve for the least-cost route, and print out the results. But because we haven't finished writing the code to find the least-cost route, we don't have any meaningful results yet. Instead, what **`Optimizer.main`** prints out as the **`Route`** object for the least-cost route is **`null`** – a Java keyword used to indicate a reference to nothing (see Figure 17).

Many programmers make the mistake of writing code in an all-or-nothing fashion. They write a lot of code, without organizing it so it can be tested along the way; when they finally do test, the errors are many and entangled, making it more difficult to correct them than it should be. We'll try to avoid that mistake, by writing small pieces of code, then compiling and testing after each addition, even if the new code doesn't do much.



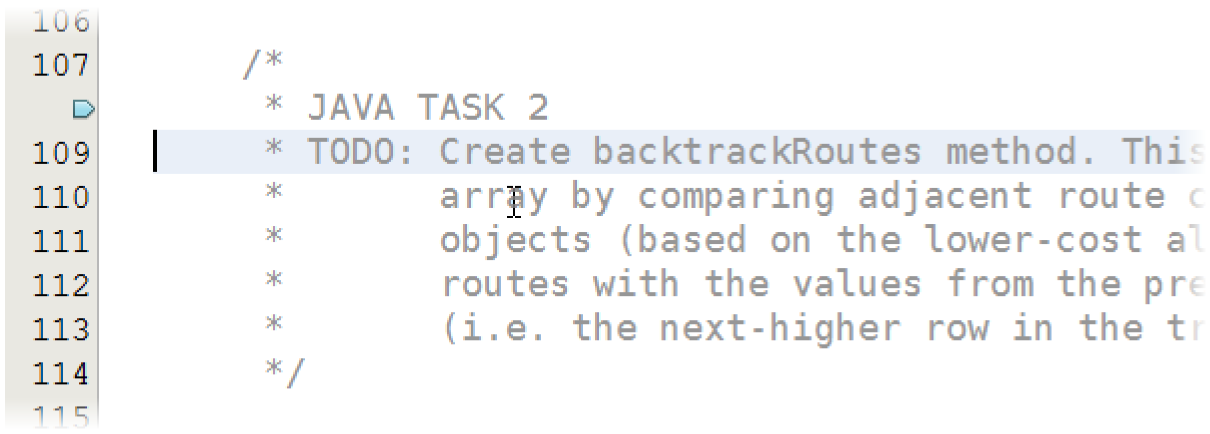
```
run:
null
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 17 - Initial Java execution output

Now that you've completed the **`setupRoutes`** method, you can delete the placeholder comments for task 1 if you want. If you do, make sure to delete the block comment start (“/*”) on line 99, the block comment end (“**`*/`**”) on line 104, and everything in between. (Feel free to do this after each task.)

Java task 2: `backtrackRoutes`

Move down a few lines to the next block comment, and you'll see the placeholder for the next task: writing the **`backtrackRoutes`** method. (Don't worry if your line numbers don't match what you see in Figure 18; they may or may not match, depending on deletion of placeholder comments, line breaks in the code, etc.)



```
/*
 * JAVA TASK 2
 * TODO: Create backtrackRoutes method. This
 *       array by comparing adjacent route c
 *       objects (based on the lower-cost al
 *       routes with the values from the pre
 *       (i.e. the next-higher row in the tr
 */
```

Figure 18 - Placeholder comment for Java task 2

In the **`backtrackRoutes`** method, we need to examine the contents of the **`routes`** array, to see which of each pair of adjacent routes is less costly. This will tell us which direction we should step, from the item located above that pair in the triangle. Then we'll create a new **`Route`** object based on the less costly route, insert the item above into the route, and add its cost (from the **`costs`** array) to the cost of the route. As we do this for each pair, we'll place the new **`Route`** object into a new array of routes; when we're done with the row, we'll assign the **`routes`** variable to refer to this new array. At the completion of one invocation of the **`backtrackRoutes`** method, there'll be one fewer item in the **`routes`** array than there were before, since we'll have moved up one row in the triangle.

Add the code from Listing 2 to the **`Optimizer`** class, to create the **`backtrackRoutes`** method. This one is longer than **`setupRoutes`** (actually, this is the largest single piece of code we'll write in either implementation), so pay close attention to the new code (remember, don't re-write the comment lines), including spelling, case, punctuation, parentheses, brackets, and braces. (One thing you don't have to worry about is indentation: the automatic indenting done by NetBeans is fine for our purposes.)

```java
/*
 * JAVA TASK 2
 * TODO: Create backtrackRoutes method. This method must update the routes
 *       array by comparing adjacent route costs, creating new Route
 *       objects (based on the lower-cost alternative), and updating the new
 *       routes with the values from the previous element of the costs array
 *       (i.e. the next-higher row in the triangle).
 */
private void backtrackRoutes() {
    Route[] result = new Route[routes.length - 1];
    int[] prependedCosts = costs[routes.length - 2];
    for (int i = 0; i < prependedCosts.length; i++) {
        Route leftRoute = routes[i];
        Route rightRoute = routes[i + 1];
        if (leftRoute.getCost() <= rightRoute.getCost()) {
            result[i] = new Route(leftRoute)
                .prepend(Step.LEFT, prependedCosts[i]);
        }
        else {
            result[i] = new Route(rightRoute)
                .prepend(Step.RIGHT, prependedCosts[i]);
        }
    }
    routes = result;
}
```

Listing 2 - `backtrackRoutes` method

Some of the code we wrote here should be familiar from the previous method. Once again, we have **`private`** access, a **`void`** return type, and an empty parameter list. Of course, the block statement making up the **`backtrackRoutes`** method implementation is very different from that of **`setupRoutes`**, since the task being performed is different.

- We declare an array of **`Route`** objects, referred to as **`result`**, as a local variable.
- We declare another local variable, **`prependedCosts`**, and assign to it the element of the **`costs`** array for the next higher row of the triangle. (Note that we use the current length of **`routes`** to determine the correct index into the **`costs`** array.)
- Next, we iterate over the **`prependedCosts`** array. For each item, we do the following:
    - Declare **`leftRoute`** and **`rightRoute`** as local variables, referring (respectively) to the **`Route`** starting below and to the left, and below and to the right, of the current cost item.
    - If the cost of **`leftRoute`** is less than or equal to the cost of **`rightRoute`**, we construct a new **`Route`** object based on **`leftRoute`**, and then *prepend* (insert at the start) a step in the left direction, and the current item's cost, to that route.

Otherwise, the new **`Route`** object is based on **`rightRoute`**, and the prepended step is in the right direction. In either case, the new Route object is placed in the **`result`** array.

In prepending a step to the route, we're using a finite set of constant values to represent the step directions. In this case, an *enum* (a special kind of class) named **`Step`** has been provided, with two possible values: **`LEFT`** and **`RIGHT`**.

(Note that we've broken some of the expressions into two lines. In general, Java doesn't care about line breaks, as long as we break between tokens, and not in the middle of an identifier or literal value.)

- After iteration is complete, and the **`result`** array is fully populated with new **`Route`** objects, the **`route`** variable is made to refer to the **`result`** array.

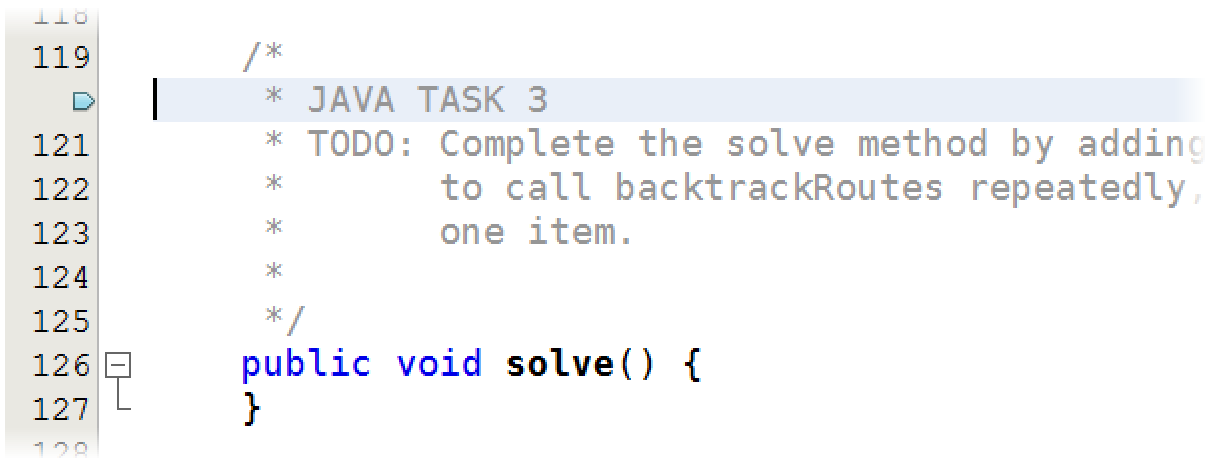
Once again, we need to correct any errors that NetBeans has flagged in our code, and save our changes.

Java task 3: `solve`

Though we've written most of the new code we'll need, neither of the two new methods is being invoked yet.

In the code provided to us, the **`main`** method creates a new **`Optimizer`** object, and the *constructor* (a block statement similar to a method, but used for the specific purpose of initializing a new object from a given class type) for the **`Optimizer`** class reads the cost data from a file. Then, the **`Optimizer.solve`** method is called, to find the least-cost route for the triangle. However, the **`solve`** method isn't calling the new **`setupRoutes`** and **`backtrackRoutes`** methods yet – actually, it's not really doing anything yet – so our final Java programming task is to add those invocations to the **`solve`** method.

Start by scrolling down to the placeholder comment for task 3 (see Figure 19). This time, you'll notice that there's already a **`solve`** method – in fact, if there weren't, NetBeans would be unable to compile or run the program, since the **`main`** method calls it. But the **`solve`** method is empty: there's no code between the curly braces that enclose the method's implementation. For the final additions to the **`Optimizer`** class, we need to be very careful to write the new code as part of the **`solve`** method's block statement – i.e. between the curly braces.



```
    /*
     * JAVA TASK 3
     * TODO: Complete the solve method by adding
     *       to call backtrackRoutes repeatedly,
     *       one item.
     *
     */
    public void solve() {
    }
```

Figure 19 - Placeholder comment for Java task 3

This time, the code we need to write is fairly simple: **`setupRoutes`** must be *invoked* (called), and then **`backtrackRoutes`** must be invoked repeatedly, until the **`routes`** variable has only a single element – namely, the **`Route`** object containing the least-cost route that starts at the apex of the triangle. (Remember: each invocation of **`backtrackRoutes`** moves one row up in the triangle.)

Add the code from Listing 3 to the **`Optimizer`** class. As before, you don't need to re-write the comments; much more important, you must not re-write the method signature, or the curly braces that enclose the method body: you should only add the four lines of code between the curly braces that enclose the method body.

```
/*
 * JAVA TASK 3
 * TODO: Complete the solve method by adding code to call setupRoutes and
 *       to call backtrackRoutes repeatedly, until the routes array has only
 *       one item.
 */
public void solve() {
    setupRoutes();
    while (routes.length > 1) {
        backtrackRoutes();
    }
}
```

Listing 3 - `solve` method

The changes we made this time were quite simple:

- The **`setupRoutes`** method is called. As you may remember, that method populates the **`routes`** array with trivial routes that only include the bottom row of the triangle.

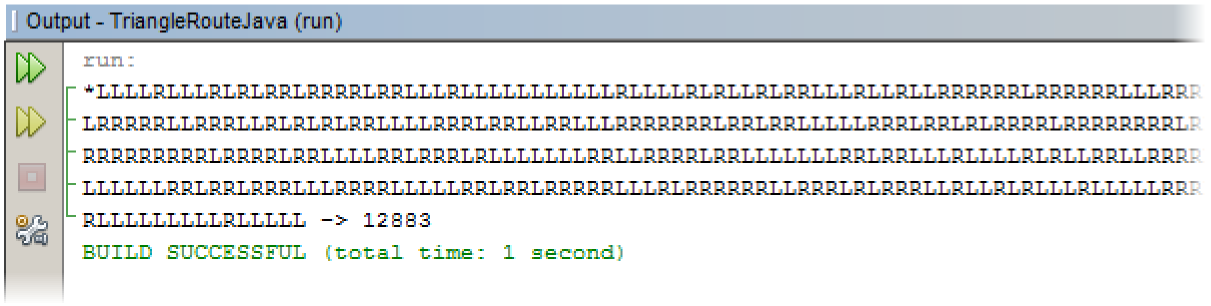
- The **backtrackRoutes** method is called repeatedly, in a **while** loop. A **while** loop repeats as long as a given condition is true – in this case, as long as the length of the **routes** array (i.e. the number of **Route** objects in the array) is greater than 1.
- Remember that right after calling **setupRoutes**, the number of elements in **routes** is equal to the number of items along the bottom of the triangle; every time **backtrackRoutes** is called, the number of routes decreases by one, since the new routes start one row higher up in the triangle. Obviously, when the number of elements in **routes** is equal to 1, we've reached the apex of the triangle, and the single **Route** object in the **route** array represents the least-cost route from the apex of the triangle to its base. At that point, the **while** condition prevents any further iteration, and the **solve** method is done.

Check, fix, and save your changes.

Java task 4: Test the program

Since the **main** method is already calling the **solve** method, and the **solve** method now invokes the **setupRoutes** and **backtrackRoutes** methods, our program should be ready to run. Of course, it's possible that we accidentally mistyped some of the code, and though it's syntactically correct, there are logical errors. But let's not get ahead of ourselves.

Run your program as described at the end of Java task 1 (p. 13). If all of your code has been written correctly, the **Output** panel will look something like Figure 20.



```
run:
*LLLLRLLLRLRLRRLRRRRLRRLLLRLLLLLLLLLLLLRLLLLRLRLLRLRRLLLRLLRLLRRRRRRLRRRRRRLLLRRR
LRRRRRLLRRRLLRLRLRLRRLLLLRRRLRRLLRRLLLRRRRRRRLRRLRRLLLLLRRRLRRLRLRRRRLRRRRRRRRLR
RRRRRRRRRLRRRRLRRLLLLRRLRRRLRLLLLLLLLRRLLRRRRLRRLLLLLLLRRLRRLLLRLLLLRLRLLRRLLRRRR
LLLLLLRRLRRLRRRLLLRRRRLLLLLRRLRRLRRRRRLLLRLRRRRRRLLRRRLRLRRRLLRLLRLRLLLRLLLLLRRR
RLLLLLLLLLRLLLLL -> 12883
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 20 - Output produced by successful execution of correct code

From this output, we see that the least-cost route for our 500-row triangle starts by stepping left 4 times (the start of the route is displayed as an asterisk), right 1 time, left 3 times, and so on (of course, every step also moves down 1 row). Just before it reaches the base, it steps right 1 time, then left 5 times. The total cost for this route is 12,883.

For comparison, the values in the triangle are random integers, ranging from 1 to 99, with an average of about 50. If we start at the apex and flip a coin 499 times, stepping down and to the left on heads, and down and to the right on tails, we'll end up at the base, with an expected route cost of about 500×50, or $25{,}000$. Even if we didn't already know that our algorithm is guaranteed to find the least-cost route, we have initial evidence that it does a good job of finding a much better-than-average route.

What if your program fails with an error message, or doesn't get the same answer? The different kinds of logical errors that can creep into code are too numerous to list, but here are a few that might show up in our program:

- *Null pointer errors*

  The code tries to perform an operation on an array or object, but the variable that is supposed to refer to that array or object actually refers to nothing (**`null`**). This is usually caused by accidental omission of a statement that assigns a non-null value to an array or object reference variable.

  When this error happens, an exception (an error condition detected by the JRE) occurs, and a message to that effect is usually printed or written to an error log. In NetBeans, these error messages are displayed in the **Output** panel. If this happens, we can use the line number indicated by the error message, and check for a missing assignment statement somewhere above that line in the code.

- *Array index out of bounds*

  When we read or write to an array, Java checks to make sure that we're not accessing an element at a position outside the limits of the array. If we are, an exception occurs.

  The culprit could be the statement we're using to create the array; we might be specifying an incorrect size. Or, it could be that we're iterating over the elements of an array in a **`for`** loop, but we're incorrectly accessing a different array – with a different size – in the loop. Or we could be using the wrong index variable to access the desired array. In any event, as before, we can use the line number indicated in the error message to help track down the problem.

- *Bad stopping condition or bad update logic in a **`while`** or **`for`** loop*

  This is often as simple as testing for the opposite inequality to what is intended. For example, we might intend for a **`while`** loop to continue iterating as long as **`x`** is greater than or equal to 1 – but we accidentally write it as "**`while (x <= 1) {…}`**". Or, we might not be updating **`x`** correctly in the loop. Neither problem seems serious, but depending on the specific error, and the initial value of **`x`**, this mistake could end up causing our loop never to iterate, to iterate fewer or more times than expected – or even to iterate forever (an *infinite loop*). In either case, the JRE generally can't detect this problem for us, because it can't read our minds and figure out that what we wrote wasn't what we meant.

Quite often, we don't suspect an infinite loop until we realize that our program seems to get stuck on some operation, and never completes it. If this seems to be the case, we can use the NetBeans debugger to help find the error; but before doing that, we should review our **`for`** and **`while`** loops, to make sure that they're written correctly.

- *Valid-but-unintended syntax*

  This is a broad category, but one of the most common sources of this kind of problem is leaving out curly braces for what is intended to be a block statement in a **`for`** or **`while`** loop. The usual result is that instead of repeating a whole block of statements multiple times, only the first of those statements is repeated.

  This could show up as the program displaying an incorrect answer, or an exception causing the program to halt; like some other error conditions, it could even ripple through the program, showing up as a different kind of error at another location in the code. This type of error can be one of the more difficult to track down, unfortunately – which is one of the reasons that many programmers get into the habit of using block statements even when they aren't strictly needed; for them, typing the start and end curly braces because almost a reflex action, triggered whenever they write **`for`** and **`while`** loops, **`if`** statements, etc. (This is also one of the main reasons for the automatic curly brace generation feature in NetBeans and other programming editors.)

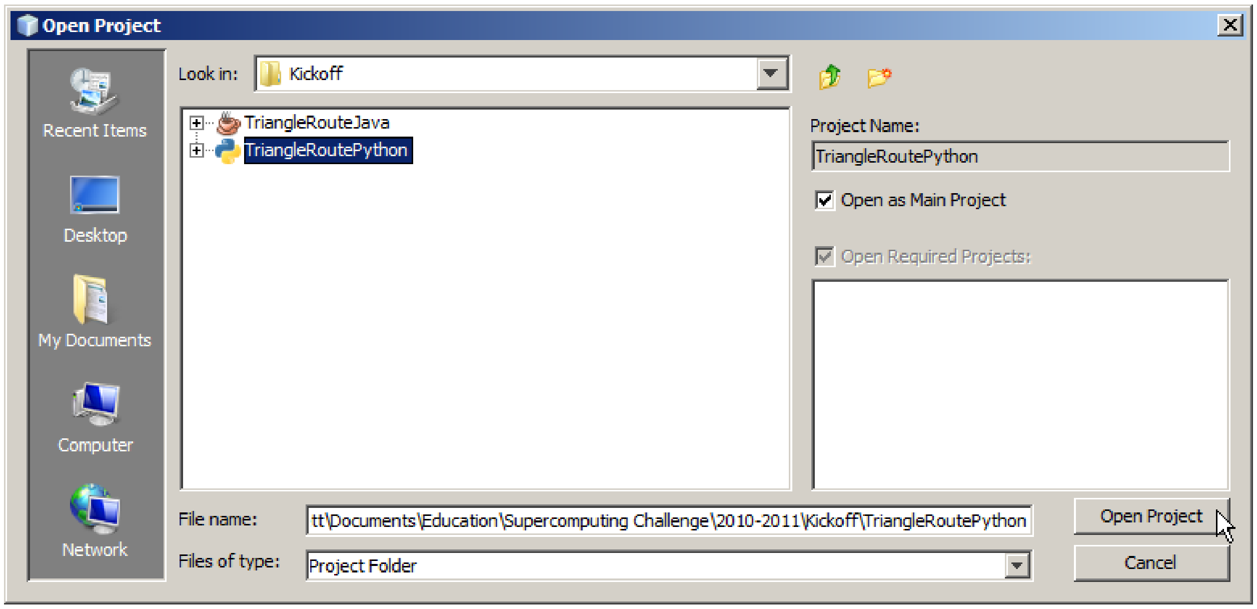
If you do get different results than those in Figure 20, or if your program halts without completing its calculations, use the supplied code listings, the information above, and any error messages to try and resolve the problems.

Once you have your program running correctly, close the **TriangleRouteJava** project by right-clicking on the project name in the **Projects** view, and selecting **Close** from the context menu. (This step isn't required, but it will help avoid confusion as you begin working on the Python implementation.)

5. *Python Implementation*

Open the Python project

To open the Python language version of the project, follow the instructions given in "Open the Java project" (see page 13) for the Java implementation; however, rather than opening `TriangleRouteJava`, select and open `TrianglеRoutePython` instead.



Look in: Kickoff

TriangleRouteJava
TriangleRoutePython

Project Name: TriangleRoutePython

[✓] Open as Main Project

[✓] Open Required Projects:

File name: tt\Documents\Education\Supercomputing Challenge\2010-2011\Kickoff\TriangleRoutePython

Files of type: Project Folder

Figure 21 - Opening TriangleRoutePython project

Once again, all of our new code will be written in a single file that's already in the project. The file is `optimizer.py`, and it's located in the `Sources` folder of the `TriangleRoutePython` project (see Figure 22). Double-click on `optimizer.py` to open it in the editor (see Figure 23). Again, you may notice that some code folding has been done; as before, this can be ignored.

Like the `Optimizer.java` file, the `optimizer.py` file defines a class called **`Optimizer`** (see Figure 24); the parallel naming between the two versions was deliberate for this lesson. However, while a one-to-one correspondence between file names and class names is a strict rule in Java, Python has no such requirement, and it's quite common to find Python *modules* (files) that contain multiple classes, none of them with the same name as the module. (Java files can contain multiple classes, but only one non-nested class in a file may be declared public, and it must have the same name as the file.)

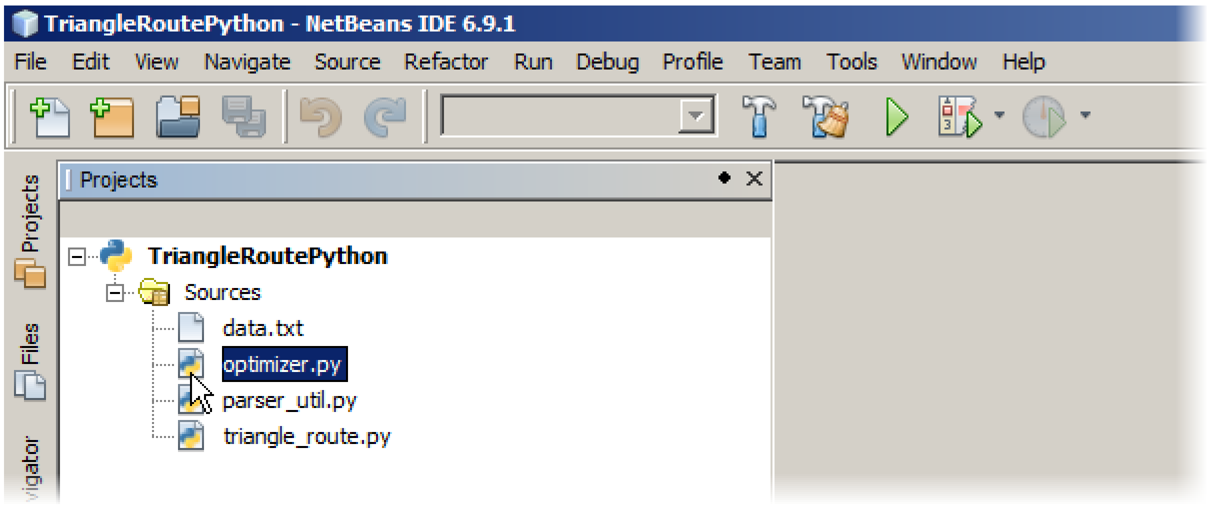
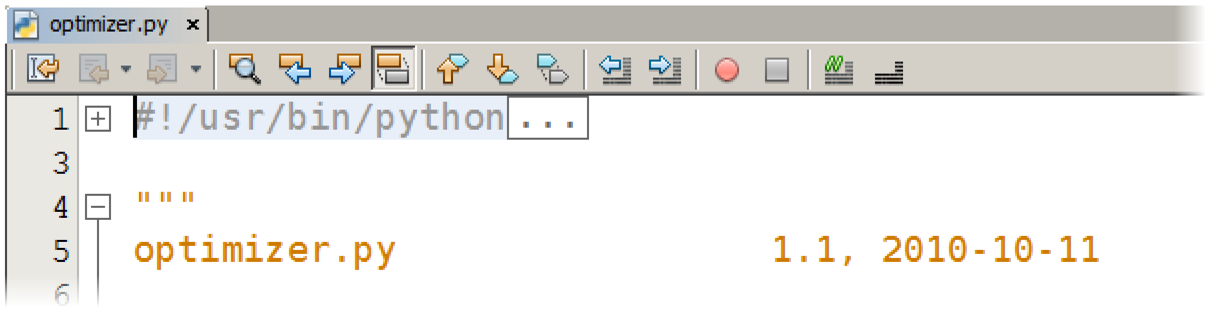

TriangleRoutePython - NetBeans IDE 6.9.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Projects
Files
vigator
Projects
TriangleRoutePython
Sources
data.txt
optimizer.py
parser_util.py
triangle_route.py

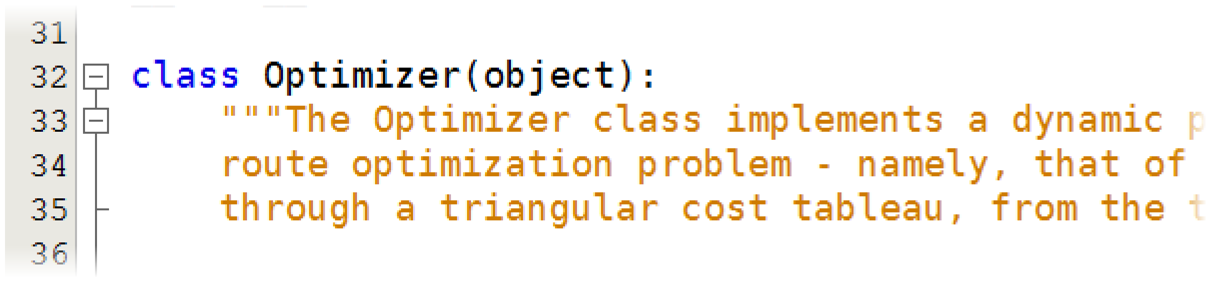
Figure 22 - Locating `optimizer.py` file



```
#!/usr/bin/python...

"""
optimizer.py                       1.1, 2010-10-11
```

Figure 23 - Code folding of initial comment



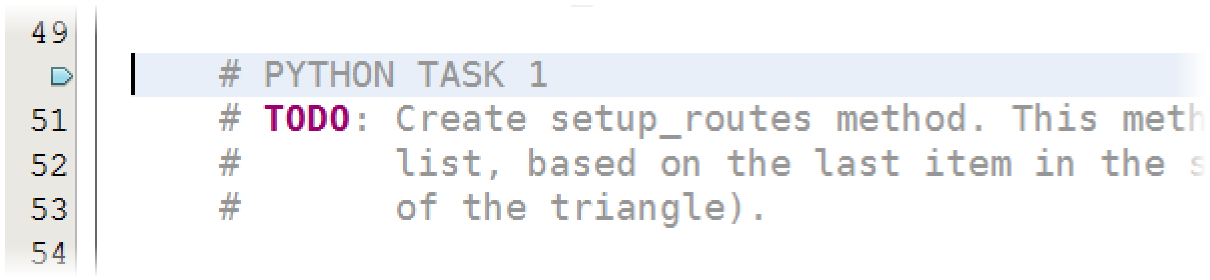
```

class Optimizer(object):
    """The Optimizer class implements a dynamic p
    route optimization problem - namely, that of
    through a triangular cost tableau, from the t
```

Figure 24 - `Optimizer` class declaration

Python task 1: `setup_routes`

At line 50 or so, you'll find the placeholder comment for the first method we'll be adding to the Python **`Optimizer`** class.



```
    # PYTHON TASK 1
    # TODO: Create setup_routes method. This meth
    #       list, based on the last item in the s
    #       of the triangle).
```

Figure 25 - Placehold comment for Python task 1

Functionally, Python task 1 is the same as the Java task 1. However, some of the surface details are different (and as we'll see, the implementation code is quite different). For one thing, we'll follow the convention used by many Python programmers, and use underscores in multi-word identifiers. Also, the structures holding the costs and routes for the triangle are called **`_costs`** and **`_routes`**, with initial underscores; Python doesn't support access modifiers (e.g. **`private`**), so the initial underscore is often used as a hint that the variable should be treated as if it were private.

Another important detail about both of these variables is that they're *lists*, rather than arrays. (**`_costs`** is actually a list of lists, just as the Java variable **`costs`** was an array of arrays.) Lists are a native data structure in Python (arrays are not), that can hold any kind of data (a single list can even hold multiple kinds of data at once), and can grow or shrink as needed. Further, many of the operations that require iteration over an array in Java (and many other languages) can be expressed very compactly in Python, without explicit iteration statements. For the **`setup_routes`** method, we'll use one of these list processing techniques, *called list comprehension*.

Make your first addition to `optimizer.py` by writing the code shown in Listing 4 below the placeholder comment. Please note that Python, like Java, is case-sensitive. Also, unlike Java, indentation is important in Python (more on that below).

```
# PYTHON TASK 1
# TODO: Create setup_routes method. This method must fill the self._routes
#       list, based on the last item in the self._costs list (i.e. the base
#       of the triangle).
def setup_routes(self):
      self._routes = []
      final_step_costs = self._costs[-1]
      self._routes = [Route(cost) for cost in final_step_costs]
```

Listing 4 - **`setupRoutes`** method

Let's review the new code:

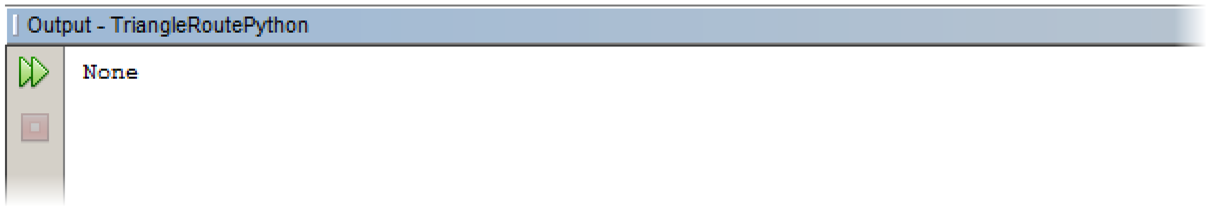
- The **`def`** keyword is used to define a new method or function. Notice that in Python, we don't declare the type of data the method returns – or indeed, if it returns anything or not.
- The **`setup_routes`** method has a parameter list containing a single item, **`self`**. Within the body of a method, **`self`** refers to the current instance of the class (in this case, **`Optimizer`**), and this parameter is required when defining a method that will be invoked on objects of a class, However, we can also define *functions* that aren't associated with a class, and these don't have **`self`** as a parameter.
- Python doesn't use curly braces to enclose block statements (called *suites* in Python); instead, indentation is used for this purpose. This means that careful attention to consistent indentation is a must, when programming in Python.
- In Python, when accessing instance variables, we must refer to them using an instance identifier. Inside a method definition, **`self`** refers to the current instance, so we use **`self._costs`** to access **`_costs`**, and **`self._routes`** to access **`_routes`**.
- Just as we don't declare return types for Python methods, we don't declare types for variables; the type of a variable is inferred from the type of data assigned to it. Here, we declare the local variable **`final_step_costs`**, and assign to it the last item in the **`_costs`** list. Since **`_costs`** is a list of lists, each of its items is itself a list, and **`final_step_costs`** is a list. (As with array indices in Java, we use square brackets in Python to access items in a list. However, this operation is more flexible in Python; in this case, we use an index value of **`-1`** to get the last item in the list. We'll see more ways to refer to list items in the next task.)
- Like Java, Python uses a **`for`** statement for iteration. It can be used very much like the Java **`for`**; in this case, however, it's being used in *list comprehension*. In list comprehension, a list is constructed by evaluating a given expression for every item in a list (we can optionally filter the source list items, but we're not doing that here). So rather than saying “for every item in a list, do the following suite of

operations," we're saying "construct a new list by evaluating this expression for every item in an existing list." This might seem like a trivial distinction, but it's the key to some of Python's elegance of expression.

We're constructing a new **`Route`** object for each item in the **`final_step_costs`** list; using list comprehension, these Route objects are themselves being gathered into a list. Finally, we assign that resulting list to **`_routes`**.

Once again, save your changes. Because of the nature of Python, some errors caught when writing Java code aren't detected until runtime with Python. However, NetBeans will do its best to find structural errors, and flag them with the usual red marks.

When all flagged errors are corrected, run the program, as described at the end of Java task 1 in the Java Implementation section (p. 13). This time, instead of the result **`null`**, you'll get its Python equivalent, **`None`** (see Figure 26).



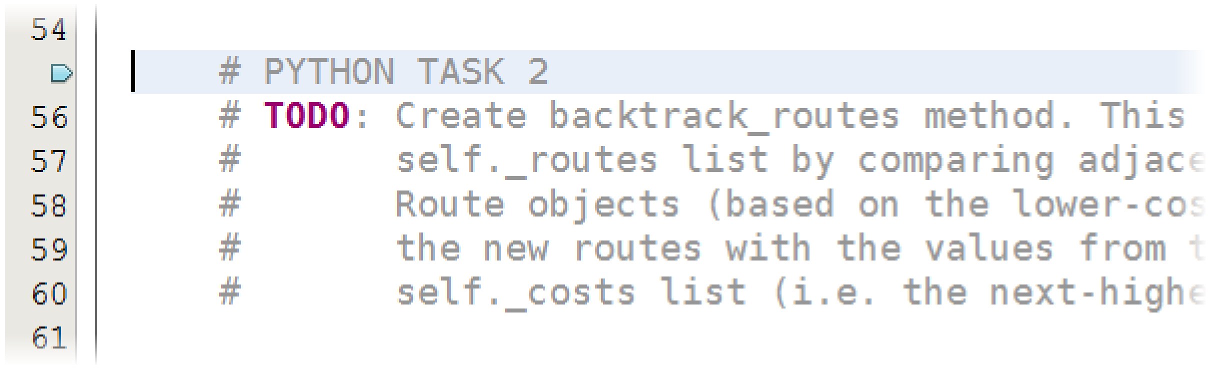
```
None
```

Figure 26 - Initial Python execution output

(If you want to delete the placeholder comments, note that Python doesn't have block comments; instead, the **`#`** character is used to start a comment that continues only until the end of that line. On the plus side, that means we don't have to be quite as careful when deleting Python comments as we do when deleting block comments in Java.)

Python task 2: `backtrack_routes`

Scroll down a bit in the `optimizer.py` file, and locate the placeholder comment for task 2 (see Figure 27).



```
    # PYTHON TASK 2
    # TODO: Create backtrack_routes method. This
    #       self._routes list by comparing adjace
    #       Route objects (based on the lower-cos
    #       the new routes with the values from t
    #       self._costs list (i.e. the next-highe
```

Figure 27 - Placeholder comment for Python task 2

In the **backtrack_routes** method, we'll use another one of the powerful list processing tools offered by Python, the **map** function. This function lets us transform (i.e. map) one list to another, by specifying a function that should be applied to each item in the first list; the results of applying that function are gathered into the resulting list. (If this sounds similar to list comprehension, it is: there's a lot of overlap between these two techniques, but each also has unique capabilities.)

In fact, the **map** function can use more than one list at a time as an input value. We can process multiple lists at once, taking corresponding values from all of them to produce the resulting list items. That's what we'll do in **backtrack_routes**.

In our algorithm, we create a new set of least-cost routes for each row by looking at the pairs of least-cost routes that start in the row below, and comparing their costs to find the step directions for the routes starting in the current row. Then, those step directions are inserted into the selected routes, and the cost values in the current row are added to the route costs. We can show part of this with Figure 28, where c_0, c_1, c_2, etc. are the costs in the current row, and r_0, r_1, r_2, etc. are the previously computed least-cost routes that start in the next row down.

$$\begin{array}{ccccccccccccccc} & c_0 & & c_1 & & c_2 & & c_3 & & c_4 & & \ldots & & c_{n-1} & \\ r_0 & & r_1 & & r_2 & & r_3 & & r_4 & & r_5 & & \ldots & & r_n \end{array}$$

Figure 28 - Row of costs with row of least-cost routes below

Note that the row of costs has n items, and the row of routes has $n+1$ items. After processing these rows, we'll have a row of n routes.

We can rearrange the contents of these two rows into three lists.[5] Note that all of the r values, apart from r_0 and r_n, appear in both of the route lists. This makes sense, since they're used as the left alternative for one c value, and the right alternative for another.

$$\begin{array}{rcl} \text{costs} & = & [c_{0,}\, c_{1,}\, c_{2,}\, \ldots,\, c_{n-1}] \\ \text{left routes} & = & [r_{0,}\, r_{1,}\, r_{2,}\, \ldots,\, r_{n-1}] \\ \text{right routes} & = & [r_{1,}\, r_{2,}\, r_{3,}\, \ldots,\, r_{n}] \end{array}$$

Each of these lists contains n items. Given any triplet (c, r, r') of corresponding items taken from the three lists, we can completely determine the direction of the step to take

5 If we were using a language like MATLAB, we might instead rearrange the values into a $3 \times n$ matrix. For every programming language, there's a

from *c* for a least-cost route. The **map** function lets us express this in a very compact fashion.

Add the **backtrack_routes** method to the **Optimizer** class by writing the code from Listing 5 below the placeholder comment.

```
# PYTHON TASK 2
# TODO: Create backtrack_routes method. This method must update the
#       self._routes list by comparing adjacent route costs, creating new
#       Route objects (based on the lower-cost alternative), and updating
#       the new routes with the values from the previous item in the
#       self._costs list (i.e. the next-higher row in the triangle).
def backtrack_routes(self):
    left_routes = self._routes[0:-1]
    right_routes = self._routes[1:]
    prepended_costs = self._costs[len(self._routes) - 2]
    self._routes = map(lambda left, right, cost:
        Route(left).prepend(Step.left, cost) if left.cost <= right.cost
            else Route(right).prepend(Step.right, cost),
        left_routes, right_routes, prepended_costs)
```

Listing 5 - **setupRoutes** method

The first thing you might have noticed is that the Python version of this method is much shorter than the Java version; this is partly due to the fact that we don't have any extra for curly braces, partly due to the use of the **map** function, and partly due to the use of a *conditional expression* (also known as a *ternary operator*, a conditional expression takes three input expressions, evaluates one of them, and depending on whether it's true or false, returns the value of one of the other two). Let's look at the method in more detail:

- We declare the local variables **left_routes** and **right_routes**, each of which contains a *slice* (a portion of a sequence) of the **_routes** list. In the case of **left_routes**, we're assigning to it the slice of **_routes** that starts with the first item, and ends just before the last item. For **right_routes**, we're taking the slice of **_routes** that starts with the second item and includes all remaining items in the list. (These slices are some examples of the advanced list indexing features in Python.)
- The item of the **_costs** list corresponding to the current row is assigned to the local variable **prepended_costs**. (Note that we index **_costs** with the same calculation we used in Java.)
- These three lists (**left_routes**, **right_routes**, **prepended_costs**) are used together to create a new list, via the **map** function. The **map** function uses at least two parameters:

  - First, a function that will be used to create the new list's items, using the items from the existing lists. In this case, we're using a *lambda* (an anonymous function, declared inline) that takes three parameters: **`left`**, **`right`**, and **`cost`**. The second part of **`lambda`** is the expression (in terms of the lambda's parameters) that will be evaluated for each set of values passed to the lambda. In this case, the expression is a conditional expression that returns a new **`Route`** instance based on **`left`** or **`right`** (with **`cost`** added) depending on which one has the lower cost.
  - The remaining parameters to **`map`** are the lists that it will use for input values. Here, the lists are **`left_routes`**, **`right_routes`**, and **`prepended_costs`**.

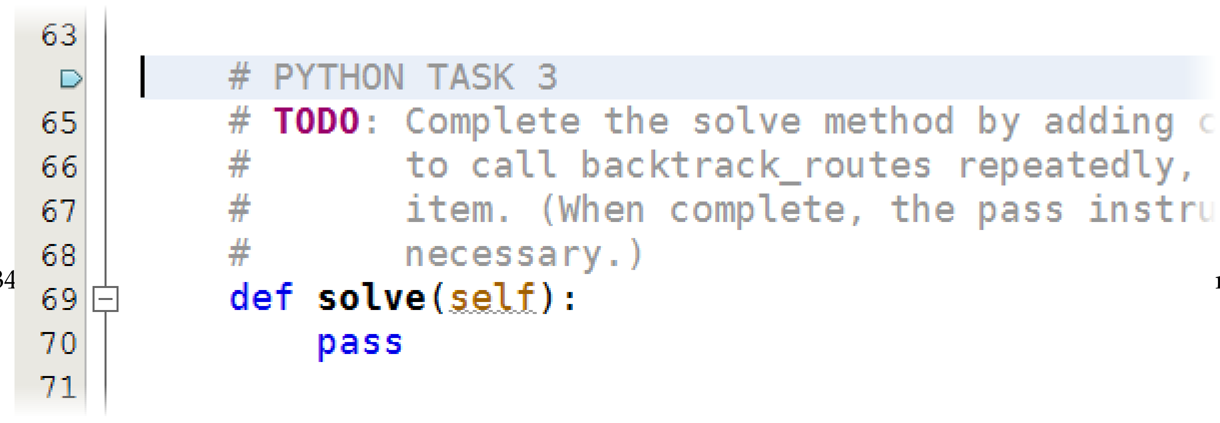
  The map function iterates implicitly over all of its input lists simultaneously. Each set of list items (in this case, the *i*th item of **`left_routes`**, the *i*th item of **`right_routes`**, and the *i*th item of **`prepended_costs`**) are passed as parameters to the specified function (e.g. our lambda). The results are then gathered into a new list.

- The list returned by the **`map`** function is assigned to the **`_routes`** list.
- Note that the parameters to **`map`**, and the two parts of **`lambda`**, are spread across multiple lines. Python is more restrictive than Java in how lines may be broken: line breaks between expressions inside parentheses or brackets are always allowed; otherwise, line breaks require the explicit use of the backslash ("\") as a line-continuation character.

Fix any errors that NetBeans has marked in your code, then save your changes.

Python task 3: `solve`

Scroll down further in the `optimizer.py` file to locate the placeholder comment for task 3 (see Figure 29), our final programming task. Note that the **`solve`** method signature is already written for us. Also, there's something we haven't seen before: the **`pass`** statement. This placeholder statement serves a useful purpose: when a Python statement must include a suite (as is the case for a method declaration), but none is supplied, an error occurs; we can avoid that error by using **`pass`**.



```
    # PYTHON TASK 3
    # TODO: Complete the solve method by adding c
    #       to call backtrack_routes repeatedly,
    #       item. (When complete, the pass instru
    #       necessary.)
    def solve(self):
        pass
```

Figure 29

For this task, the code we need to write is almost identical to the code we wrote for the Java version: we need to invoke the **`setup_routes`** method, and that then invoke the **`backtrack_routes`** method repeatedly, until **`_routes`** contains only a single item.

Add the code from Listing 6 to `optimizer.py`. Be very careful not to re-write the method signature. (On the other hand, the **`pass`** statement can be deleted, if desired.)

```
# PYTHON TASK 3
# TODO: Complete the solve method by adding code to call setup_routes and
#       to call backtrack_routes repeatedly, until self._routes has only one
#       item. (When complete, the pass instruction will no longer be
#       necessary.)
def solve(self):
    self.setup_routes()
    while len(self._routes) > 1:
        self.backtrack_routes()
```

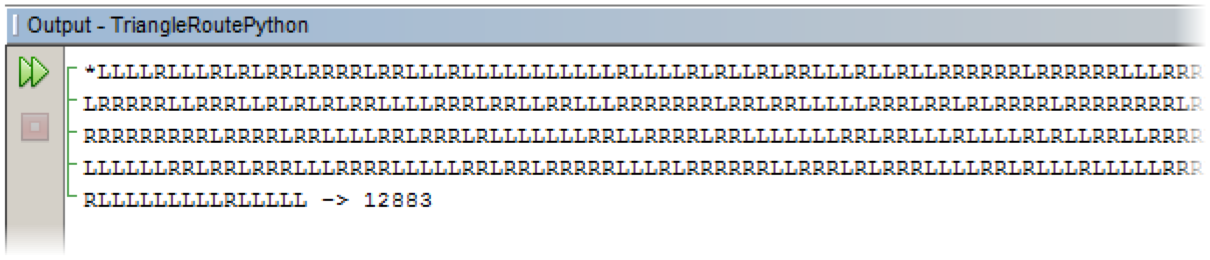
Listing 6 - **`solve`** method

This method is pretty self-explanatory, especially after writing the Java version. However, note that the method calls include **`self`**; this is required, just as it is when accessing an instance variable such as **`_routes`**. In fact, making the method calls in this fashion causes the reference to the current **`Optimizer`** instance to be passed to the methods as the value of the **`self`** parameter.

Fix any marked errors and save your changes.

Python task 4: Test the program

Run your program as before. If the code is correct, you'll see the same route and cost as those produced by the Java implementation (see Figure 30). (Depending on the Python version NetBeans is configured to use, the display details might be different than those shown; nonetheless, the route steps and cost should match.)



```
*LLLLRLLLRLRLRRLRRRRLRRLLLRLLLLLLLLLLLRLLLLRLRLLRLRRLLLRLLRLLRRRRRRLRRRRRRLLLRRR
LRRRRRLLRRRLLRLRLRLRRLLLLRRRLRRLLRRLLLRRRRRRRLRRLRRLLLLLRRRLRRLRLRRRRLRRRRRRRRLR
RRRRRRRRRLRRRRLRRLLLLRRLRRRLRLLLLLLLLRRLLRRRRLRRLLLLLLLRRLRRLLLRLLLLLRLRLLRRLLRRRR
LLLLLLRRLRRLRRRLLLRRRRLLLLLRRLRRLRRRRRLLLRLRRRRRRLLRRRLRLRRRLLLLLRRLRLLLLRLLLLLLRRR
RLLLLLLLLLRLLLLL -> 12883
```

Figure 30

Most of the errors we could potentially make while programming in Java we can also make with Python. Of course, each language has its own wrinkles in that regard. Here are a few to watch out for:

- The fact that we don't declare variable or method types in Python makes for less verbose code than we usually write in Java, but it also opens up the possibility of accidentally assigning one type of value to a variable intended to store another type. When we do that in Java, the compiler catches the erroneous assignment early; in Python, it will show up unexpected results (or a runtime error, if we're lucky) when we run the program.
- In Java, variables must be declared before they are used in any way. If we declare the variable **`finalRowCosts`**, but then try to assign a value to **`finalrowcosts`** (note the difference in capitalization), the Java compiler catches this, and makes us correct it before compiling the program. In Python, however, a variable is declared implicitly the first time we assign a value to it. This can result in very unexpected results arising from simple typos, if we're not careful in our spelling and capitalization.
- Inconsistent indentation, and indentation using a mixture of spaces and tabs, will usually be caught by NetBeans (and some other IDEs) before we run a program; however, it can sometimes elude early detection and produce a parsing error at runtime. And if we've forgotten to indent something that's supposed to be part of a suite, we'll probably get unexpected and incorrect results from the program execution.

  In this case, if our program returns unexpected results, one thing we can do is try to step through the code manually, comparing it to the bottom-up algorithm (p. 7), or to the pseudocode version (see Appendix B) – to make sure that we understand which operations are part of the iterated steps, and which should be performed conditionally. If we're still confused, stepping through the code with a debugger can help a lot.
- List slicing, list comprehension, and the **`map`** function are very powerful features of Python, but they can be tricky to master. It's very easy to end up with a list of a different length (or of a different dimension – e.g. a list of lists, instead of a list) than we intended. Sometimes, this results in an error when accessing elements of the list, just as an out-of-bounds array access in Java produces an error. In some cases, however, the flexibility of Python list manipulation makes catching and fixing these errors fairly difficult, since Python supports many different types of list transformations with slight variations in syntax.

  When we suspect something like this is happening, we can use the debugging facilities in NetBeans to observe the values stored in the lists as the program executes.

- The ternary conditional expression in Python is ordered differently than analogous constructs in C/C++, Java, Logo, and many other languages. In most programming languages, the expression to be evaluated and tested is the first of the three expressions that make up the ternary expression; in Python, it's the second. Also, unlike Java (but like C and many other languages), Python can treat the result of virtually any expression as a *boolean* (true or false) value. Given these two factors, it's very easy to write the Python ternary expression incorrectly, without the error being caught in parsing, compilation, or even at runtime. (Ironically, this can be more of a problem for those with some programming experience in other languages than for complete beginners.)

If your results don't match those shown in Figure 30, or if your program halts without completing its calculations, use the supplied code listings, the information above, and any error messages to try and resolve the problems.

This page intentionally left blank.

6. Conclusion

Summary discussion

In this lesson, you've seen a simple example of a combinatorial optimization problem. You've also seen that the increase in the number of potential solutions quickly makes an exhaustive search through all of them impossible (or practically so). On the other hand, you've seen an algorithm that solves the problem quite efficiently – but here we run into some questions that we haven't addressed yet:

- What does "efficient" mean, in the context of an algorithm for solving some problem?
- How can we measure the efficiency of an algorithm? How can we do so without implementing the algorithm on a computer first?
- Are there some problems for which having an efficient method for getting a "good enough" solution is preferable to having a method that gets the best solution, but does so much less efficiently? Can you think of any examples?

In completing the Java and Python tasks, you've gotten a small taste of what's involved in implementing an algorithm in a computer program; you've also seen how some processing concepts are expressed in very different ways in these two programming languages. You probably haven't yet spent enough time programming in either language to feel completely comfortable with it, but at least you've seen the tip of the programming iceberg in each. Based on that experience, discuss the following questions:

- What did you like about each of the two languages?
- What did you dislike about each?
- What aspects, of either language, did you find especially difficult to understand?
- What types of programs or projects do you think Java would be best suited for?
- What types of programs or projects do you think Python would be best suited for?
- Of the features you saw in the two languages, were there any that you think could be especially useful to you, in some project you're doing (or considering doing)? What are they?

Online resources for additional study and practice

There are many resources available to help you build your skills as a programmer, or as a programming mathematician or scientist. In addition to the countless books and traditional classes on programming, there are many free resources available online.

The standard installation of Python includes a tutorial that can help you learn its syntax and explore the core libraries [6]. The standard Java installation doesn't include a tutorial, but a very extensive one can be downloaded free of charge from the Oracle Technology Network [7].

Beyond the tutorials, there are several puzzle and practice sites – some dedicated to specific programming languages, others open to multiple languages. Of particular note is CodingBat, which has several sets of practice problems on a variety of programming concepts and techniques; many of the problems are posted in both Java and Python versions [8]. Possibly the most powerful feature of CodingBat is the online execution environment, allowing users to compile and test their code solutions from almost any computer connected to the Internet.

For more experienced programmers – as well as programming students eager for a challenge – Project Euler hosts an online collection of hundreds of puzzle problems (including the one that inspired this lesson), ranging in difficulty from fairly easy to very hard [9]. An important feature of this site is the discussion forum: when a person submits a correct answer to a problem, access is granted to a discussion thread devoted to that problem; members of the user community use these discussions to share their solutions and learn from each other. Solving a problem and then seeing how others apply different techniques and languages to the same problem is a great way to learn new concepts – and to expand your programming skills from one language to many.

There are online databases of algorithms that can also be valuable resources – to practitioners needing algorithms for specific tasks, and also to students who want to learn more about some of the most important tools in computer science and applied mathematics. Two such sites are the National Institute of Standards and Technology's Dictionary of Algorithms and Data Structures, and Algorithmist [10],[11]. The NIST DADS site is the more complete of the two, and it's more extensively cross-linked to other algorithm-oriented sites. Algorithmist hasn't been in existence as long, but it's structured as a wiki, with community members building the knowledge base in a collaborative fashion, and it has a lot of potential beyond it's already solid collection.

Ultimately, whatever programming languages you want to learn, and however you learn best, there are resources available to you. Take advantage of them.

References

[1] Colin Hughes, "Problem 67: Using an efficient algorithm find the maximal sum in the triangle?", ProjectEuler, April 2004. [Online]. Available: http://projecteuler.net/index.php?section=problems&id=67. [Accessed: Oct. 6, 2010].

[2] Wayne L Winston, *Operations Research: Applications and Algorithms*, Belmont, CA: Wadsworth, 1994, pp. 1005-1011.

[3] Parag H. Dave, Himanshu B. Dave, *Design and Analysis of Algorithms*, India: Pearson, 2008, pp. 269-308.

[4] "Dynamic programming", Wikipedia, Oct. 4, 2010. [Online]. Available: http://en.wikipedia.org/wiki/Dynamic_programming. [Accessed: Oct. 9, 2010].

[5] NetBeans v6.9.1. [Download]. Sun Microsystems, Inc., Aug. 4, 2010. Available: http://netbeans.org/downloads/. [Accessed: Oct. 10, 2010].

[6] Python Programming Language. [Download]. Python Software Foundation, Aug. 24, 2010. Available: http://www.python.org/download/. [Accessed: Oct. 10, 2010].

[7] The Java Tutorials. [Online and download]. Oracle Corp., Jul. 7, 2010. Available: http://download.oracle.com/javase/tutorial/. [Accessed: Oct. 10, 2010].

[8] Nick Parlante, CodingBat, 2010. [Online]. Available: http://www.codingbat.com. [Accessed: Oct. 11, 2010].

[9] Colin Hughes, Project Euler, Oct. 10, 2010. [Online]. Available: http://www.projecteuler.net/. [Accessed: Oct. 11, 2010].

[10] Dictionary of Algorithms and Data Structures. [Online]. National Institute of Standards and Technology, Sep. 27, 2010. Available: http://xw2k.nist.gov/dads/. {Accessed: Oct. 12, 2010].

[11] Algorithmist. [Online]. Algorithmist.com, Oct. 11, 2010. Available: http://www.algorithmist.com. [Accessed: Oct. 12, 2010].

Acknowledgements

Development of this material and the accompanying program code was funded in part by the New Mexico Supercomputing Challenge.

Janet Penevolpe, LD Landis, Roger Critchlow, and Drew Einhorn aided in the development of this material and the accompanying program code, through their questions, comments, and suggestions.

The underlying problem addressed in this material was inspired by Problem 67 at Project Euler [1].

Appendix A — Solution to Pencil-and-paper Task 2

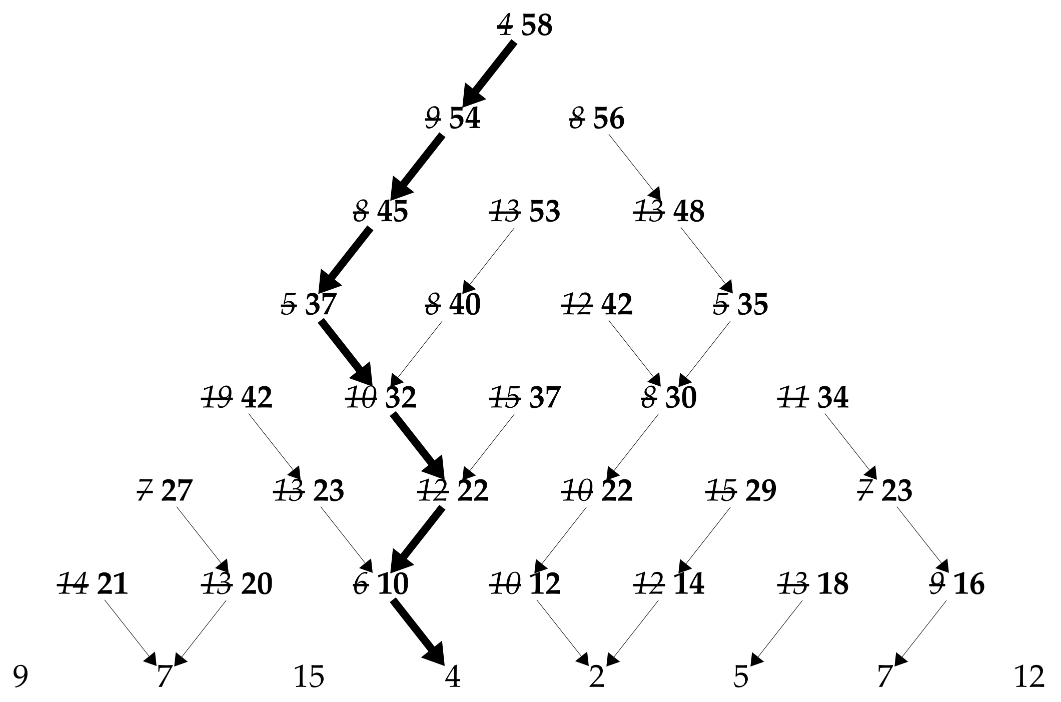


Figure 31 - Least-cost route for triangle in Figure 7 (p. 8); cost = 58

This page intentionally left blank.

Appendix B — Pseudocode for Least-cost Triangle Route Algorithm

Pseudocode is a description of an algorithm – more generally, a way of describing algorithms – that uses grammar and notation common to many programming languages, but avoids conventions specific to any single programming language. Besides borrowing conventions from programming, pseudocode often uses notation normally found in mathematical expressions as well. The main benefits of this are that potential ambiguity is greatly reduced, and translation of pseudocode to actual programming code is usually fairly straightforward.

Here's one way (among many) to use pseudocode to express the bottom-up algorithm for the least-cost triangle route (p. 7):

```
• Given:
    ◦ $n$ = number of rows in triangle
    ◦ $c_{ij} =$ value of item $j$ in row $i$ of triangle, where $i \in \{1, 2, \ldots, n\}$, $j \in \{1, 2, \ldots, i\}$
    ◦ $r_{ij}$ = route from item $j$ in row $i$ to base of triangle (initially empty)
    ◦ concat$(a, b)$ = concatenation of text values $a$ and $b$
• For $i = n - 1$ down to 1:
    ◦ For $j = 1$ to $i$:
        ▪ If $c_{(i+1)j} \le c_{(i+1)(j+1)}$:
            • $c_{ij} = c_{ij} + c_{(i+1)j}$
            • $r_{ij} =$ concat$(\text{'L'}, r_{(i+1)j})$
          Otherwise:
            • $c_{ij} = c_{ij} + c_{(i+1)(j+1)}$
            • $r_{ij} =$ concat$(\text{'R'}, r_{(i+1)(j+1)})$
• $r_{1,1}$ is least-cost triangle route
• $c_{1,1}$ is minimum triangle route cost
```

This page intentionally left blank.

Appendix C — Java Implementation

Notes

- The Java implementation is organized into the following packages and classes:
    - **`org.nm.challenge.kickoff.triangleroute`**
        - **`Optimizer`** (`Optimizer.java`)
        - **`Route`** (`Route.java`)
        - **`Step`** (`Step.java`)
    - **`org.nm.challenge.kickoff.util`**
        - **`TableFileParser`** (`TableFileParser.java`)
- In the interests of compactness, copyright notices have been stripped from these source code listings. All copyrights asserted (and licenses granted) in the “Copyright and License Information” section of this document are in force for all source code listings.
- In the interests of compactness, comments and non-essential annotations have been stripped from these source code listings. Refer to the accompanying source code files for documentation comments, code comments, and annotations.
- Please note that because of the changes noted above, and some reformatting for page width, line numbers in source code listings don't correspond to line numbers in accompanying source code files.

Optimizer.java

```java
package org.nm.challenge.kickoff.triangleroute;

import java.io.FileNotFoundException;
import java.io.IOException;
import org.nm.challenge.kickoff.util.TableFileParser;

public class Optimizer {

    public static final String DEFAULT_DATA_FILE = "data.txt";
    public static final String DEFAULT_DELIMITER = "\\s+";

    private int[][] costs;
    private Route[] routes;

    public static void main(String[] args) {
        try {
            String dataFile = DEFAULT_DATA_FILE;
            String delimiter = DEFAULT_DELIMITER;
            if (args.length > 1) {
                dataFile = args[0];
                if (args.length > 2) {
                    delimiter = args[1];
                }
            }
            Optimizer solver = new Optimizer(dataFile, delimiter);
            solver.solve();
            System.out.println(solver.getSolution());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public Optimizer(String dataFile, String delimiter)
            throws FileNotFoundException, IOException {
        costs = new TableFileParser(dataFile, delimiter).toIntArray();
        routes = null;
    }

    public Route getSolution() {
        return ((null != routes && 1 == routes.length) ?
            routes[0] : null);
    }

    private void setupRoutes() {
        int[] finalStepCosts = costs[costs.length - 1];
        routes = new Route[finalStepCosts.length];
        for (int i = 0; i < finalStepCosts.length; i++) {
            routes[i] = new Route(finalStepCosts[i]);
        }
    }

    private void backtrackRoutes() {
        Route[] result = new Route[routes.length - 1];
        int[] prependedCosts = costs[routes.length - 2];
        for (int i = 0; i < prependedCosts.length; i++) {
            Route leftRoute = routes[i];
            Route rightRoute = routes[i + 1];
            if (leftRoute.getCost() <= rightRoute.getCost()) {
                result[i] = new Route(leftRoute)
                    .prepend(Step.LEFT, prependedCosts[i]);
            }
            else {
                result[i] = new Route(rightRoute)
                    .prepend(Step.RIGHT, prependedCosts[i]);
            }
        }
        routes = result;
    }

    public void solve() {
        setupRoutes();
        while (routes.length > 1) {
            backtrackRoutes();
        }
    }

}
```

Route.java

```
package org.nm.challenge.kickoff.triangleroute;

import java.util.LinkedList;

public class Route {

    private static final char STARTING_POINT = '*';
    private static final char LEFT_PATH = 'L';
    private static final char RIGHT_PATH = 'R';
    private static final String STRING_PATTERN = "%s -> %d";

    private LinkedList<Step> steps;
    private int cost;

    public Route(int cost) {
        steps = new LinkedList<Step>();
        this.cost = cost;
    }

    public Route(Route source) {
        steps = new LinkedList<Step>(source.steps);
        cost = source.cost;
    }

    public Route prepend(Step step, int cost) {
        steps.add(0, step);
        this.cost += cost;
        return this;
    }

    public Route append(Step step, int cost) {
        steps.add(step);
        this.cost += cost;
        return this;
    }
    public int getCost() {
        return cost;
    }

```

```java
    public String toString() {
        StringBuilder buffer = new StringBuilder();
        boolean firstRow = true;
        for (Step step : steps) {
            if (!firstRow) {
                buffer.append((step == Step.LEFT) ?
                    LEFT_PATH : RIGHT_PATH);
            }
            else {
                buffer.append(STARTING_POINT);
                firstRow = false;
            }
        }
        return String.format(STRING_PATTERN, buffer.toString(), cost);
    }

}
```

Step.java

```java
package org.nm.challenge.kickoff.triangleroute;

public enum Step {
    LEFT,
    RIGHT;
}
```

TableFileParser.java

```java
package org.nm.challenge.kickoff.util;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import java.util.LinkedList;

public class TableFileParser {

    private String[][] table;

    public TableFileParser(String fileName, String delimiterRegex)
            throws FileNotFoundException, IOException {
        this(new File(fileName), delimiterRegex);
    }

    public TableFileParser(File file, String delimiterRegex)
            throws FileNotFoundException, IOException {
        FileReader reader = null;
        BufferedReader buffer = null;
        LinkedList<String[]> work = new LinkedList<String[]>();
        String line;
        try {
            reader = new FileReader(file);
            buffer = new BufferedReader(reader);
            while (null != (line = buffer.readLine())) {
                if (line.trim().length() > 0) {
                    String[] values = line.trim().split(delimiterRegex);
                    work.add(values);
                }
            }
        }
        finally {
            if (null != buffer) {
                buffer.close();
            }
            if (null != reader) {
                reader.close();
            }
            table = work.toArray(new String[0][]);
        }
    }

    public String[][] toStringArray() {
        String[][] result = new String[table.length][];
        for (int i = 0; i < table.length; i++) {
            result[i] = Arrays.copyOf(table[i], table[i].length);
        }
        return result;
    }

    public int[][] toIntArray() {
        int[][] result = new int[table.length][];
        for (int i = 0; i < table.length; i++) {
            result[i] = new int[table[i].length];
            for (int j = 0; j < table[i].length; j++) {
                result[i][j] = Integer.parseInt(table[i][j]);
            }
        }
        return result;
    }

    public double[][] toDoubleArray() {
        double[][] result = new double[table.length][];
        for (int i = 0; i < table.length; i++) {
            result[i] = new double[table[i].length];
            for (int j = 0; j < table[i].length; j++) {
                result[i][j] = Double.parseDouble(table[i][j]);
            }
        }
        return result;
    }

}
```

Appendix D — Python Implementation

Notes

- The Python implementation is organized into the following modules and classes:
    - **`optimizer`** (`optimizer.py`)
        - **`Optimizer`**
    - **`triangle_route`** (`triangle_route.py`)
        - **`Route`**
        - **`Step`**
    - **`parser_util`** (`parser_util.py`)
        - **`TableFileParser`**
- In the interests of compactness, copyright notices have been stripped from the source code listings included here. All copyrights asserted (and licenses granted) on the copyright page (p. 2) of this document are in force for all source code listings.
- In the interests of compactness, comments and non-essential annotations have been stripped from the source code listings included here. Refer to the accompanying source code files for documentation comments, code comments, and annotations.
- Please note that because of the changes noted above, and some reformatting for page width, line numbers in source code listings don't correspond to line numbers in accompanying source code files.

optimizer.py

```python
import sys

from parser_util import TableFileParser
from triangle_route import Route

class Optimizer(object):

    def __init__(self, data_file, delimiter):
        self._costs = TableFileParser(data_file, delimiter).ints()
        self._routes = None

    @property
    def solution(self):
        return (self._routes[0]
            if self._routes is not None and len(self._routes) == 1
            else None)

    def setup_routes(self):
        self._routes = []
        final_step_costs = self._costs[-1]
        self._routes = [Route(cost) for cost in final_step_costs]

    def backtrack_routes(self):
        left_routes = self._routes[0:-1]
        right_routes = self._routes[1:]
        prepended_costs = self._costs[len(self._routes) - 2]
        self._routes = map(lambda left, right, cost:
            Route(left).prepend(Step.left, cost) if left.cost <= right.cost
                else Route(right).prepend(Step.right, cost),
            left_routes, right_routes, prepended_costs)

    def solve(self):
        self.setup_routes()
        while len(self._routes) > 1:
            self.backtrack_routes()

if __name__ == "__main__":
    default_params = ['data.txt', None]
    data_file, delimiter = map(lambda default, actual:
        actual if actual is not None else default,
        default_params, sys.argv[1:])[:2]
    opt = Optimizer(data_file, delimiter)
    opt.solve()
    print(opt.solution)
```

triangle_route.py

```
import string

class Route(object):

    def __init__(self, source=None):
        if isinstance(source, Route):
            self._steps = source._steps[:]
            self._cost = source._cost
        elif source is not None:
            self._steps = []
            self._cost = source
        else:
            self._steps = []
            self._cost = 0

    def prepend(self, step, cost):
        self._steps.insert(0, step)
        self._cost += cost
        return self

    def append(self, step, cost):
        self._steps.append(step)
        self._cost += cost
        return self

    def __str__(self):
        result = ['*'] + [('R' if (step == Step.right) else 'L')
            for step in self._steps]
        return '%s -> %d' % (string.join(result, ''), self._cost)

    @property
    def cost(self):
        return self._cost

class Step(object):

    left = 0
    right = 1
```

parser_util.py

```python
from __future__ import with_statement

class TableFileParser(object):

    def __init__(self, file_name, delimiter):
        with open(file_name) as file:
            self._table = [line.strip().split(delimiter)
                for line in file
                if 0 != len(line.strip())]

    def strings(self):
        return [row[:] for row in self._table]

    def ints(self):
        return [[int(col) for col in row] for row in self._table]

    def floats(self):
        return [[float(col) for col in row] for row in self._table]
```