Com S 227
Fall 2023
Miniassignment 3
60 points (out of 30 possible)

Due Date: Friday, November 17, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm November 16)

10% penalty for submitting 10 days late (by 11:59 pm November 27)

No submissions accepted after November 27, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus for details.

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

There are two problems, both of which involve a recursive search. The first one (part A) involves searching a maze, and the second one (part B) is to find all solutions to a game based on "twenty-four". Both are fun, and neither one should be terribly difficult if you have a good understanding of Lab 7. The first problem is 30 points and the second problem will be counted as another 30 points "extra credit". (Just to be clear: There will be 110 points for miniassignments, which count as 4% of your grade, so if you add 30 extra points in that category it could potentially add about 1% to your overall grade. So the main benefit of doing these problems is that you will understand recursion better, which benefits you somewhat on our final exam and benefits you more proufoundly in Com S 228 and 311.)

This is a miniassignment and the grading is automated. If you do not submit it correctly and we have to run it by hand, you will receive at most half credit.

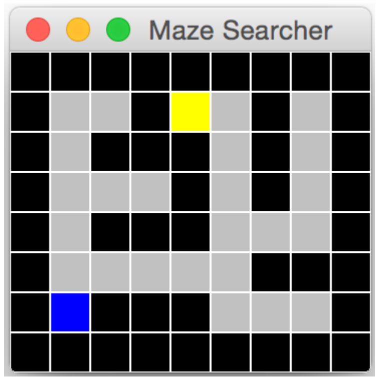
Part A - maze searcher

This is a fun problem to give you some practice with recursion. This had always been one of my favorite kinds of examples to go back to and think about when I was in my second year of college and trying to visualize how recursion works.

Your task is to implement the recursive method **`search`** in the class **`MazeExplorer`**. The sample code includes a skeleton for the **`MazeExplorer`** class that you can use. *You'll want to be sure you have done and understood lab 7 first.*

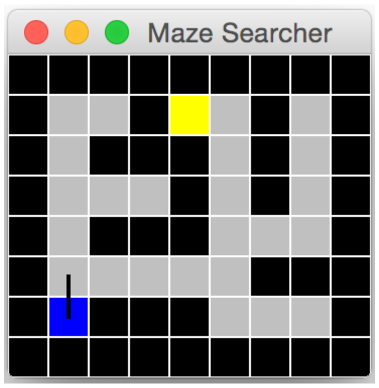
The sample code includes a lot of related stuff in the packages **`maze_api`** and **`maze_ui`**. *Do not modify any of that code*, just implement the **`search`** method.

The **`search`** method will recursively search for a path in a 2D maze. We can picture a 2D maze like this, where the blue cell is the starting point, the yellow cell is the princess we have to rescue (the "goal"), and the black cells are boundaries ("walls").

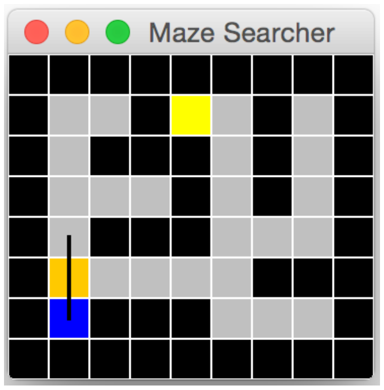


The challenge with searching a maze (or anything that is non-linear, such as a file hierarchy) is keeping track of where you've been and being able to *backtrack* and resume searching in a different direction from a previous point. It turns out that recursion is ideal for this. What does it mean to "backtrack"? Just return from the method call you're in, and resume execution where you left off!

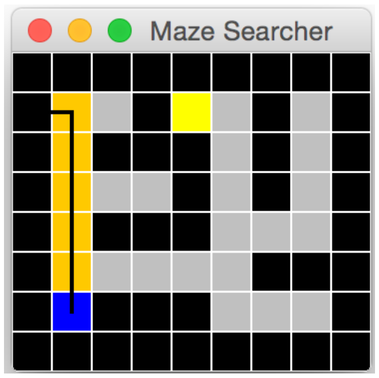
As an example, suppose you are initially standing in the blue square. You first lay down a stick to show you've started searching the cell. You plan to go to the next cell up, so you put the stick pointing up:



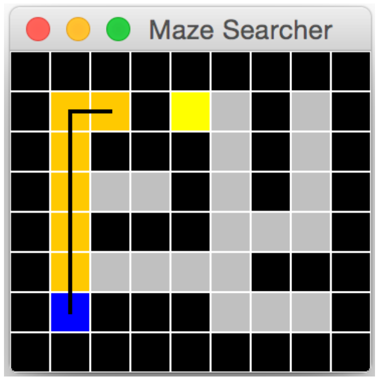
Next you step into the cell above. Again, lay down a stick, and point it upwards to show which direction you're going next:



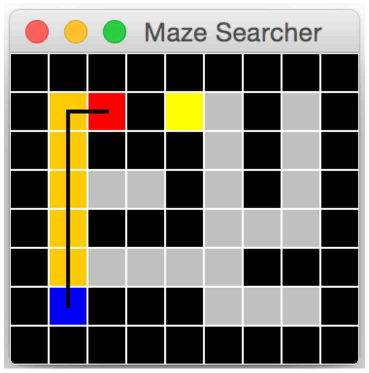
Well, keep on doing that. When you get up to the top (row 1, column 1) you can't go up, so you try other neighboring cells. If you look at the cell below, you see the stick, so you know you've already been there), so try going left:



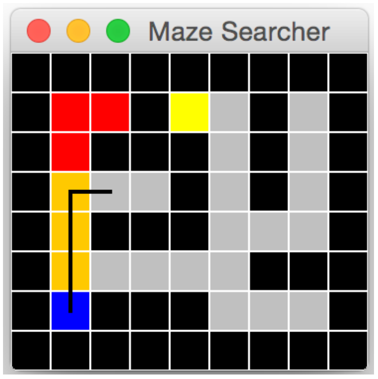
Well, that's just a wall. Try going right:



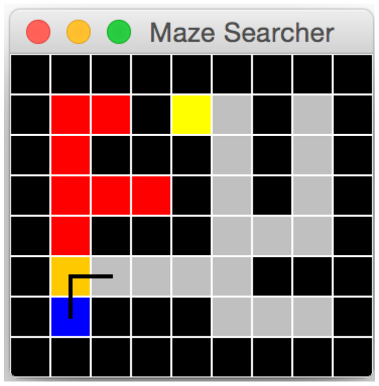
Ok, this is a cell you haven't seen yet. From here, you check up (wall), down (wall), left (already been there), and right (wall), and you're stuck. So you *backtrack* to the cell where you just were. (As you leave the cell, put a red rock on it to mark the fact that it's a dead end.):



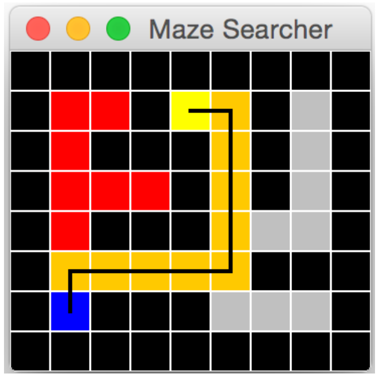
Now back at row 1, column 1, how do you know whether you've searched the other three neighbors already? You can tell since the stick is pointing right that you've already searched all its neighbors. ***That's because we're always checking neighbors in the order up, down, left, then right***. You return and keep backtracking; at row 2, column 1, you see the stick pointing up, so you check down (already been there), left (wall), and right (wall). Backtrack to row 3, column 1, again check down (already been there), to the left (wall) and then to the right, discovering an unexplored cell:



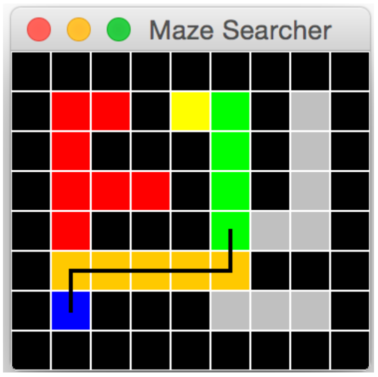
But it leads to another dead end, so you end up backtracking again. When you eventually get back to row 5, column 1 (i.e. just above the starting cell) you can again check down (already been there), left (wall) and right, to find another unexplored cell to the right:



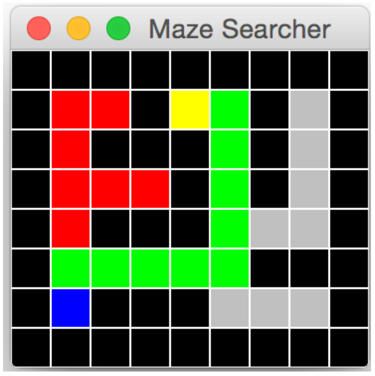
You continue searching neighboring cells, exploring up, down, left, and right (always in that order) marking with sticks as you go.



At row 1, column 4, you discover you're at the goal! Now, as you backtrack to your previous cell, put down a green rock.



This is a bit different from backtracking from a dead end: since you've already found the princess, you don't bother to search any more neighboring cells. Some parts of the maze remain unexplored when you finally backtrack all the way to the start:



This is a recursive algorithm, because basically what we're saying is:

```
to search a maze starting from cell c:
    search the maze starting at the neighboring cells of c
```

This reduces the problem to a smaller problem, provided that we avoid searching from cells we've already visited. And we have base cases: finding the goal, or a wall or dead end. The process we carried out in pictures above could be written in pseudocode like this:

```
to search a maze starting from cell c:
    if c is the goal
        return true (success)
    else if c isn't an unexplored cell (i.e. it's a wall, or been there already)
        return false
    else
        for each neighboring cell d (in the order up, down, left, right)
            mark cell c to record the direction of d
            recursively search the maze starting from d
            if the search returns true
                mark cell c as found and return true
    // if we fall through to this point after searching all four directions…
    mark cell c as a dead end and return false
```

The enum type maze_api.CellStatus

As you search, an important part of the process is to "mark" each cell with its current state, like what we did with the sticks and the red and green rocks, above. The state of a cell could be:

```
NOT_STARTED
SEARCHING_UP
SEARCHING_DOWN
SEARCHING_LEFT
SEARCHING_RIGHT
FOUND_IT
DEAD_END
```

To keep track of these possible values, we define a set of 7 constants in the **`enum`** type **`CellStatus`** having the names above. Note that the **`MazeCell`** type also has two methods **`isWall()`** and **`isGoal()`** for identifying whether it's a wall or is the goal. (The status would be ignored for wall and goal cells.)

The class maze_api.MazeCell

A **`MazeCell`** is basically just a container for a status, along with instance variables indicating whether it's a wall or goal. See the methods **`isWall()`**, **`isGoal()`**, **`getStatus()`**, and **`setStatus()`**. If you look at the code, you'll notice there is also something called an "observer". This is not something you have to worry about - the idea of the observer is that when a cell's status changes, it will call the observer's **`statusChanged`** method. This can be used by an application such as the sample UIs to respond to the status change. The type of the observer is **`GridObserver`**, which is an example of a Java **`interface`**. (You don't have to worry about any of this now, but we will have a lot to say about interfaces in a couple of weeks.)

The class maze_api.TwoDMaze

The **`TwoDMaze`** class encapsulates a 2D array of **`MazeCell`** objects. You'll really just need the method **`getCell()`** to access the current cell in the search.

It also has a constructor that takes a string array that can be used to initialize a maze. For example, the maze illustrated at the beginning of this document was initialized from the string array below. This is good to know, in case you want to create your own mazes for testing or for fun. (The 'S' character is the start and the '$' character is the goal. The methods **`getStartRow()`**

and **getStartColumn()** can be used to find the position of the start. See the class **ui.RunSearcher** for more examples.)

```
public static final String[] MAZE2 = {
    "#########",
    "#  #$ # #",
    "# ### # #",
    "#   # # #",
    "# ###   #",
    "#     ###",
    "#S###   #",
    "#########",
  };
```

The MazeExplorer UIs

There is a simple console UI that you might find useful for debugging. To run it, run **maze_ui.ConsoleUI**. In between cell updates, it prints the grid and pauses for you to press ENTER before continuing. In printing the grid, the cell states are represented as follows:

NOT_STARTED	(blank)
SEARCHING_UP	^
SEARCHING_DOWN	v
SEARCHING_LEFT	<
SEARCHING_RIGHT	>
FOUND_IT	*
DEAD_END	x

To get an idea of what to expect, the assignment posting includes a log of the console output from running the searcher with the console UI on the maze **RunSearcher.MAZE1**.

There is also a GUI (that was used to generate the color illustrations earlier in this pdf). The main method is in **maze_ui.RunSearcher**. You can edit the **main** method to initialize with one of several predefined mazes to try. You can also adjust the speed of the animation by adjusting the value of the **sleepTime** variable.

To get an idea what to expect, there is a brief animation in the assignment posting on Canvas.

Part B - Twenty-four game solver

This is another backtracking problem. It is not really much longer than the maze searcher from part A, but there are some additional issues to think about. Your task is simply to implement the method **findSolutions** in the class **TwentyFour**.

This is loosely based on a well-known game called the "twenty-four game". The idea is that you are given a list of numbers and a target value (in the traditional form of the game, the target value is always 24). You must combine your numbers using addition, subtraction, multiplication, and division to obtain the target (implicitly you are also using parentheses since you can group the operations). For example, suppose your list of numbers is 2, 3, 4, 5, with a target value of 21. Some possible solutions would be ((5 – 2) * (3 + 4)), ((3 – 2) + (4 * 5)), or (3 * (2 + 5)).
(Note that in our version of the rules, you are *not* required to use all the numbers, and division is only allowed when there is no remainder. This is different from the traditional version.)
The goal of the **`findSolutions`** method is to create a list containing all possible solutions, represented as strings. It is ok for the list to contain duplicates and the results do not need to be in any order. Please note, for example, that we regard "(3 * (2 + 5))" and "(3 * (5 + 2))" as *different* solutions (this simplifies things in the long run).

Warm up

Before you start, be sure you understand the file lister example from Lab 7:

```
public static void listAllFiles(File f)
{
  if (!f.isDirectory())
  {
    // Base case: f is a file, so just print its name
    System.out.println(f.getName());
  }
  else
  {
    //  Recursive case: f is a directory, so print its name,
    //  and then recursively list the files and subdirectories it contains
    File[] files = f.listFiles();
    for (int i = 0; i < files.length; i += 1)
    {
     listAllFiles(files[i]);
    }
  }
}
```

This example illustrates a common use case for recursion: we have a tree-like structure (the hierarchy of directories and files on your hard drive) and we need to explore every branch of the tree to find all the files. The key to using recursion is to solve a problem by reducing it to smaller instances. In the case of the file hierarchy, a "smaller instance" corresponds to descending into a lower level of the tree.
As an aside: a minor change to the example above allows us to collect a *list* of all the files instead of printing them:

```
private static void createListOfFiles(File file, ArrayList<String> list)
{
  if (!file.isDirectory())
  {
    // base case
    list.add(file.getName());
  }
  else
  {
    // recursively search the subdirectory
    for (File f : file.listFiles())
    {
      createListOfFiles (f, list);
    }
  }
}
```

Note that the `results` list is created once by the *caller* of the method, and the same list is passed into each recursive call, so that in the end it contains the name of every file that was ever found.

The search for solutions to the twenty-four game is similar, in the sense that recursion is used to explore a tree-like structure representing all possible combinations of the numbers with arithmetic operations. In the case of searching a file hierarchy, the "tree" actually exists on your hard drive, but in the case of the twenty-four game, the "tree" is conceptual. The figures on the following pages illustrate this.

To descend down branches of this conceptual tree of possible combinations, we need to decompose the problem into smaller sub-problems so that recursion can be used. There are two possible ways to reduce the problem to a smaller sub-problem:

1. a) Choose two numbers *a* and *b* in the list, remove them from the list, and replace them with a single number *a + b, b + a, a * b, b * a, a – b, b – a, a / b,*or *b / a.* This reduces the list size by1.
2. b) Choose a single number in the list, and remove it. This allows us to find solutions in which not all the numbers are used; again the list size is reduced by 1.

The base case is easy: if you have a list of numbers of size 1, you can just check whether that value is equal to the target.

The idea is illustrated in the following figures using the list of numbers 2, 3, 3, 5 and the target value 11. In Figure 1, we start by choosing a pair of values from the list. In this example, we are examining the path in which we chose 2 and 5. We can replace the 2 and the 5 with 2 + 5, 2 * 5, 2 – 5, or 5 – 2. The possible division expressions 2 / 5 and 5 / 2 are disallowed because both have a nonzero remainder. This leads to four lists with only three numbers in each.

The same strategy can now be recursively applied to each of the three-element lists, and here we examine the branch in which the pair 3, 3 is chosen. These values can be replaced with 3 + 3, 3 * 3, 3 – 3, or 3 / 3, leading to four lists with only two numbers in each. Then we recursively apply the same strategy. In this case there is only one possible pair to choose, so we replace the 10 and 1 with the possible values 10 + 1, 10 * 1, 10 – 1, and 1 – 10. Now each subproblem is a list of length 1, so we can directly compare each of the values with the target, 11, and there is one solution.

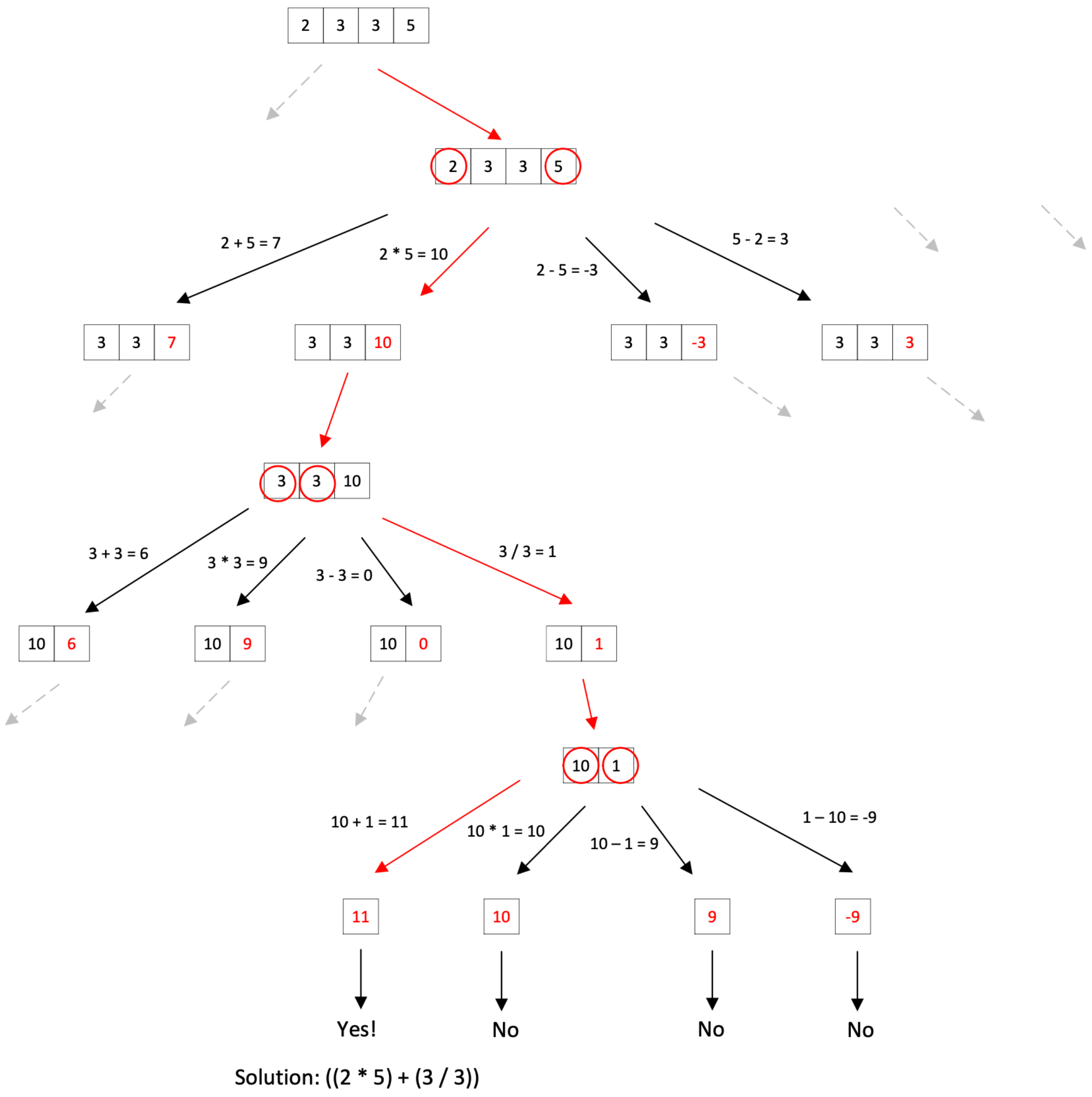


*Figure 1 - recursive decomposition leading to solution ((2 * 5) + (3 / 3)) = 11*

Figure 2 is similar, but also illustrates the fact that we need to allow subproblems that are formed just by removing an element and obtaining a shorter list, in order to obtain a solution that doesn't use all the numbers, such as (5 + (2 * 3)).

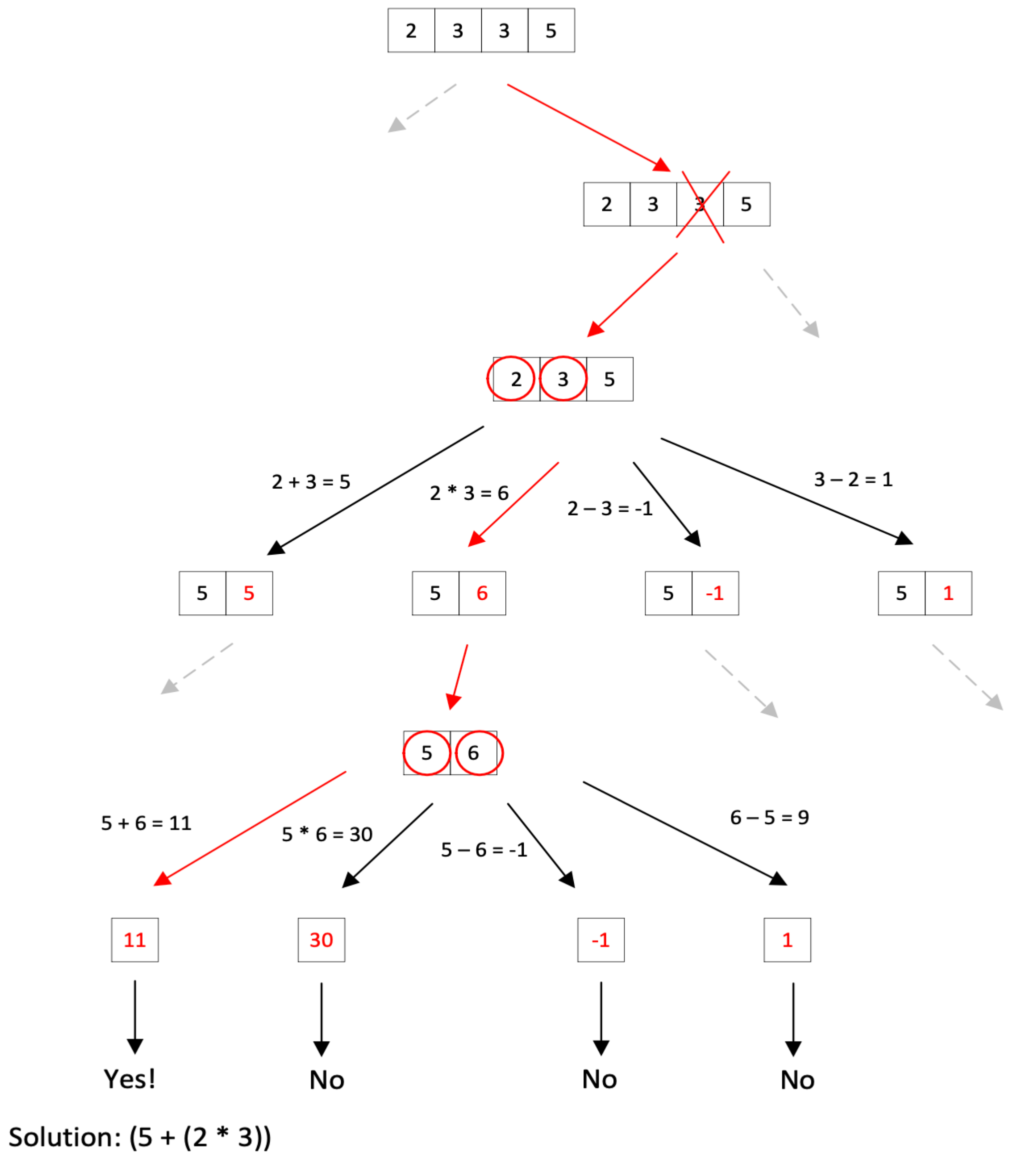


*Figure 2 - recursive decomposition leading to solution recursive decomposition leading to solution recursive decomposition leading to solution (5 + (2 * 3)) = 11*

Pseudocode

```
if the list has length 1
    if the value matches the target
        add it to the list of results
otherwise
    for each number x in the list
        create a copy of the list that does not include x
        find solutions using that list
    for each pair of numbers x, y in the list
        for each allowable arithmetic combination z of x and y
            create a copy of the list without x and y, but with z added
            find solutions using that list
```

When we say "each allowable arithmetic combination *z* of *x* and *y*", we always include *x + y, y + x, x * y, y * x, x – y,* and *y – x*. For division, x / y or x / y is only included when there is no remainder and no division by zero. (Having x + y and y + x is redundant, but it simplifies the code and the testing to always include them both.)

The `IntExpression` class

Note: the **`IntExpression`** *class is provided for you in the package* **`twenty_four_api`** *and you should not modify it.*

The examples also illustrate a further question: when we reach a base case and check that the target value has been found, how do we know what the operations were that led to it? Somehow, we must store not only the numbers in the list, but also the arithmetic expressions that were used to obtain them. For example, in the first step when we compute 2 * 5 = 10, we need to save not only the number 10, but also something like the *string* "(2 * 5)".

The class **`IntExpression`** is provided for you to make this easy. A **`IntExpression`** object just contains a number and a string. You can construct a **`IntExpression`** with a single integer or by combining two existing **`IntExpression`** objects with one of the operators '+', '*', '-', or '/'. Doing so automatically computes the result and stores a new string representation of the given values. Parentheses are always added. You obtain the integer value using the method **`getIntValue()`** and the string form using the method **`toString()`**. Here are some examples:

```
IntExpression v1 = new IntExpression(2);
IntExpression v2 = new IntExpression(3);
```

```
System.out.println(v1.toString()); // prints "2"
System.out.println(v2.toString()); // prints "3"
IntExpression v3 = new IntExpression(v1, v2, '+');
System.out.println(v3.toString()); // prints "(2 + 3)"
System.out.println(v3.getIntValue()); // 5
IntExpression v4 = new IntExpression(v2, v3, '*');
System.out.println(v4.toString()); // prints "(3 * (2 + 3))"
```

So instead of using an **`ArrayList<Integer>`**, we will use **`ArrayList<IntExpression>`** to represent a list of values. If you wanted to run the example in Figure 1, you could use a main method with statements such as the following:

```
int[] values = { 2, 3, 3, 5 };
ArrayList<IntExpression> expressions = new ArrayList<IntExpression>();
for (int x : values)
{
  expressions.add(new IntExpression(x));
}
ArrayList<String> results = new ArrayList<String>(); // empty list
TwentyFour.findSolutions(expressions, 11, results);
```

It is worth noting the difference in roles between the parameters. The result list is always passed into each recursive call unchanged, so that at the end, it will contain all solutions that have ever been found. Likewise the target value never changes. But for each recursive call, you'll be constructing a new list of **`IntExpression`**s just for that call.

To view the solutions, it is probably best to remove duplicates and sort first:

```
ArrayList<String> uniqueResults = new ArrayList<String>();
for (String r : results)
{
  if (!uniqueResults.contains(r))
  {
    uniqueResults.add(r);
  }
}
Collections.sort(uniqueResults);
for (String r : uniqueResults)
{
  System.out.println(r);
}
```

For example, the complete list of solutions for the list 2, 3, 3, 5 with target 11 (after removing duplicates and sorting as strings) is:

```
((2 * 3) + 5)
((2 * 5) + (3 / 3))
((3 * 2) + 5)
((3 * 3) + 2)
((3 + 3) + 5)
((3 + 5) + 3)
((3 / 3) + (2 * 5))
((3 / 3) + (5 * 2))
((5 * 2) + (3 / 3))
((5 + 3) + 3)
(2 + (3 * 3))
(3 + (3 + 5))
(3 + (5 + 3))
(5 + (2 * 3))
(5 + (3 * 2))
(5 + (3 + 3))
```

The expression “(2 + 3)” is considered to be *different* from the expression “(3 + 2)”. Your implementation should find all solutions and therefore needs to try both. Note, your solution can contain duplicates and the order does not matter. (The unit tests will remove duplicates and sort before checking against a correct solution.)

The sample code

The sample code is an archive of a complete Eclipse project you can import. The instructions for importing are the same as for homework 3, so refer to the pdf for homework 3 if you don't remember how to do it.

The SpecChecker

Import and run the SpecChecker as for miniassignment 2. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the console output. *Always start reading the errors at the top and make incremental corrections in the code to fix them.*

When you are happy with your results, click "Yes" at the dialog to create the zip file.

See the document “SpecChecker HOWTO”, which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links” if you are not sure what to do.

Documentation and style

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`miniassignment3`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`miniassignment3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Advice

Be sure you have done lab 7, especially the exercise described on page 2, where you use the debugger to visualize what's happening on the call stack as recursive calls are made. You might try the same thing for the file lister problem later in the lab. Notice that as you return from a recursive call on a subdirectory, you can pick up right where you left off in iterating over the list.

The maze animation is really just another way of visualizing the call stack. The orange cells are the frames that are still *active* (method call has not yet returned) and the red and green cells are calls that have *returned* a value, and are no longer on the stack.

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini3.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini3`**, which in turn contains two files, **`MazeExplorer.java`** and **`TwentyFour.java`**. Always LOOK in the zip file the file to check.

Submit the zip file to Canvas using the Miniassignment3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which is link #9 on our Canvas front page.
We strongly recommend that you just submit the zip file created by the specchecker, AFTER CHECKING THAT IT CONTAINS THE CORRECT CODE. ***If you mess something up and we have to run your code manually, you will receive at most half the points.***

> We strongly recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **mini3**, which in turn should contain the two required files. You can accomplish this by zipping up the **src** directory of your project (NOT the entire project). The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.