2024

AP® Computer Science A

Sample Student Responses and Scoring Commentary

Inside:

Free-Response Question 4

- ☑ **Scoring Guidelines**
- ☑ **Student Samples**
- ☑ **Scoring Commentary**

Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares `"int G=99, g=0;"`, then uses `"while (G < 10)"` instead of `"while (g < 10)"`, the context does **not** allow for the reader to assume the use of the lower-case variable.*

Question 4: 2D Arrays — 9 points

Canonical solution

(a) — **3 points**

```
public Location getNextLoc(int row, int col)
{
   if (row == grid.length – 1)
   {
      return new Location(row, col + 1);
   }
   else if (col == grid[0].length – 1)
   {
      return new Location(row + 1, col);
   }
   else if (grid[row + 1][col] < grid[row][col + 1])
   {
      return new Location(row + 1, col);
   }
   else
   {
      return new Location(row, col + 1);
   }
}
```

(b) — **6 points**

```
public int sumPath(int row, int col)
{
   int sum = 0;

   while (row < grid.length – 1 || col < grid[0].length - 1)
   {
      sum += grid[row][col];

      Location loc = getNextLoc(row, col);
      row = loc.getRow();
      col = loc.getCol();
   }
   return sum + grid[row][col];
}
```

(a) `getNextLoc`

	Scoring Criteria	Decision Rules	
1	Guards against out-of-bounds access of grid elements	Responses **can** still earn the point even if they • fail to access any element of `grid` in this part, as long as the guard prevents the returned `Location` from being out of bounds Responses **will not** earn the point if they • return a `Location` that would be out of bounds	**1 point**
2	Accesses both an element of `grid` to the right and an element of `grid` below `row` and `col`	Responses **can** still earn the point even if they • access elements of `grid` out of bounds Responses **will not** earn the point if they • fail to access elements of `grid` correctly	**1 point**
3	Returns `Location` of appropriate grid element (*algorithm*)	Responses **can** still earn the point even if they • incorrectly guard against out-of-bounds access of grid elements Responses **will not** earn the point if they • call the `Location` constructor incorrectly • fail to consider all four cases	**1 point**
		Total for part (a)	**3 points**

(b) `sumPath`

	Scoring Criteria	Decision Rules	
4	Initializes and increases variable to store sum of grid values	Responses **can** still earn the point even if they • fail to initialize a local variable in a recursive solution, as long as an element of the grid is added to the recursive call Responses **will not** earn the point if they • initialize the variable to something other than `0` or an element of grid • increment the sum variable using something other than an element of grid	**1 point**
5	Determines the path based on successive calls to `getNextLoc` while current position is not the bottom-right position of grid (*no bounds errors*) (*algorithm*)	Responses **can** still earn the point even if they • fail to access an element of `grid` • call `getNextLoc` incorrectly • access row/column of next location incorrectly Responses **will not** earn the point if they • fail to call `getNextLoc` • fail to use row/column derived from `getNextLoc` return value in subsequent calls • stop loop early (omit required path locations) or late (violate `getNextLoc` precondition) due to incorrect boundary condition	**1 point**
6	Calls `getNextLoc` (*in the context of a loop*)	Responses **can** still earn the point even if they • call `getNextLoc` within an incorrect loop Responses **will not** earn the point if they • call `getNextLoc` on the class or on an object other than `this` (use of `this` is optional) • fail to call `getNextLoc` with two `int` arguments	**1 point**
7	Calls `getRow` and `getCol` on a `Location` object	Responses **will not** earn the point if they • call either method incorrectly	**1 point**

8	Accesses a `grid` element at positions derived from the call to the next location method	Responses **can** still earn the point even if they • access an incorrect `grid` element • only access the grid at `row` and `col`, if the solution is recursive and the parameters of the recursive call are derived from a call to the next location method	**1 point**
9	Computes sum of values along path (*algorithm*)	Responses **can** still earn the point even if they • stop loop early or late due to incorrect boundary condition • fail to return the computed sum (*return not assessed in this part*) Responses **will not** earn the point if they • fail to include the first or last visited location in the sum	**1 point**
		Total for part (b)	**6 points**
		Total for question 4	**9 points**

Important: Completely fill in the circle that corresponds to the question you are answering on this page.

Question 1 ○ Question 2 ○ Question 3 ○ Question 4 ●

Begin your response to each question at the top of a new page.

a)

```
public location getNextLoc(int row, int col)
{

   if (row == grid.length -1);
     {
      return new location( row , col +1 );
     }

else if( col == grid[0].length -1)
     {
      return new location (row+1, col );
     }

else
     {
      if (grid [row+1][col] > grid [row][col+1])
        {
         return new location (row, col +1);
        }
        else
         {
          return new location (row +1, col);
         }

     }

}
```

Important: Completely fill in the circle that corresponds to the question you are answering on this page.

Question 1 ○ Question 2 ○ Question 3 ○ Question 4 ●

Begin your response to each question at the top of a new page.

b)

```
public int sumPath(int row, int col)
{
  boolean bool = true;
  int sum = 0;

  while (bool)
  {
    location temp = getNextLoc(row, col)
    row = temp.getRow;
    col = temp.getCol;
    if (temp.getRow == grid.length-1 && temp.getCol
        == grid[0].length-1)
    {
      bool = false
    }
    sum += grid[row][col];
  }
  return sum;
}
```

Important: Completely fill in the circle that corresponds to the question you are answering on this page.

Question 1 ○ Question 2 ○ Question 3 ○ Question 4 ●

Begin your response to each question at the top of a new page.

```
Public Location getNextLoc(int row, int col)
{
  Location smaller=newLocation(row+1, col);
  if(row +1 > grid.length) grid[row+1
   {
     smaller = new Location(row,col+1);
   }
 if(col +1 > grid[0].length)
 {
   smaller = new Location[row+1, col];
 }
  if(grid[row][col+1]< grid[row+1,col])
   {
     smaller = new Location(row,col+1);
   }
  return smaller;
```

Important: Completely fill in the circle that corresponds to the question you are answering on this page.

Question 1 ○ Question 2 ○ Question 3 ○ Question 4 ●

Begin your response to each question at the top of a new page.

```
Public int sumPath (int row, int col)
{ int sum = grid[row][col];
  Location tile = new Location (row, col);
while (tile.getRow() < grid.length && tile.getCol() < grid[0]
                                                .length)
  {
    tile = GridPath.getNextLoc(row, col);
    int val = grid[tile.getRow()][tile.getCol()];
    sum += val;
  }
  return sum;
```

Important: Completely fill in the circle that corresponds to the question you are answering on this page.

Question 1 ○ Question 2 ○ Question 3 ○ Question 4 ●

Begin your response to each question at the top of a new page.

```
a.)public location getNextLoc(int row, int col){
    if(grid[row][col+1] != null && grid[row+1][col] != null){
      if( grid[row][col+1] > grid[row+1][col]){
        return grid[row][col+1];
      }else{
        return grid[row+1][col];
      }
}else if(grid[row][col+1] != null ){
   return grid[row][col+1];
 }else{
    return grid[row+1][col];
   }
}
```

Important: Completely fill in the circle that corresponds to the question you are answering on this page.

Question 1	Question 2	Question 3	Question 4
○	○	○	●

Begin your response to each question at the top of a new page.

b.)
```
public int sumPath(int row, int col){
    Location current = grid[row][col];
    int sum = 0;
    while(current != grid[grid.length][grid[0].length])
    {
        current = getNextLoc(row,col);
        sum += current;
    }
return sum;
}
```

Question 4

Note: Student samples are quoted verbatim and may contain spelling and grammatical errors.

Overview

This question tested the student's ability to:

- Write program code to create objects of a class and call methods (Skill 3.A).
- Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).
- Write program code to create, traverse, and manipulate elements in 2D array objects (Skill 3.E).

This question involved the traversal of a two-dimensional (2D) array of `int` values, accessing variables and methods within the class being written, and creating and accessing instances of a separate, provided class representing grid coordinate pairs. In addition to an unusual traversal pattern, the algorithms required careful understanding of boundaries in a 2D array as well as writing code to guard against out-of-bounds access.

In part (a) students were asked to write a non-void method, `getNextLoc`, which has two integer parameters, `row` and `col`, and returns a `Location` object that represents the smaller of two neighbors of the `grid` element at `row` and `col`. The neighbors to be considered are the grid elements below and to the right of `row` and `col`. The method must verify that a neighbor exists before accessing it to perform a comparison: if only one of the neighbors exists (because the other would be out of bounds), the `Location` of the existing neighbor is returned. The precondition of the method guarantees that `row` and `col` are not the bottom-right corner of the grid, so at least one neighbor always exists.

In part (b) students were asked to write a non-void method, `sumPath`, which computes and returns the sum of all values on a particular path in `grid`. The path begins with the element at the location indicated by parameters `row` and `col` and is determined by successive calls to the `getNextLoc` method written in part (a). The path ends when the element in the last row and the last column of `grid` is reached. Again, the precondition of the method guarantees that `row` and `col` are not the bottom-right corner of the grid, so there will always be at least two positions in the path to traverse; the method must ensure that both the starting value and the ending value are included in the sum (and must not violate any method preconditions along the way).

Sample: 4A
Score: 8

In part (a) point 1 was earned because the response properly guards against out-of-bounds access of grid elements with the expressions `row == grid.length - 1` and `col == grid[0].length - 1`. To earn this point, the response must ensure that any access of elements of `grid` occurs within the boundaries of the 2D array. (Note that two-dimensional (2D) array objects that are not rectangular are outside the scope of the AP Computer Science A Course and Exam.) Point 2 was earned because the response accesses elements at `grid[row + 1][col]` and `grid[row][col + 1]`. To earn this point, the response must access both a grid element below

Question 4 (continued)

`grid[row][col]` (i.e., by adding 1 to the first index) and a grid element to the right of `grid[row][col]` (i.e., by adding 1 to the second index). Note that this point would be earned even if the access occurred outside the boundaries of the 2D array; boundary issues are assessed in point 1. Point 3 was earned because the response returns an appropriate `Location` object in all four cases. To earn this (algorithm) point, the response must correctly identify which location in the grid is to be returned in all four cases (bottom row, rightmost column, smaller element below, smaller element to the right). The response must also call the `Location` constructor with the appropriate row and column for each case and return that `Location` object.

In part (b) point 4 was earned because the variable `sum` is initialized to `0` and is increased using values from `grid`. To earn this point, the response must store a sum of `grid` values in a variable. The variable must be initialized to either `0` or a value from `grid` and must be incremented solely using values from `grid`. If the response uses recursion, the variable storing the sum is not required, as long as the sum is performed as a result of the recursive call. Adding anything else to this variable (e.g., a `Location` object) will not earn this point. Point 5 was earned because the response properly determines the path based on successive calls to `getNextLoc` without any bounds errors. The variables `row` and `col` are used to store the current position within the grid being examined; when the bottom-right position of the grid is reached, the `boolean` variable controlling the loop is set to `false`, ending the loop. To earn this point, the response must make successive calls to `getNextLoc` (in a loop) and use the results of those calls to traverse the grid until the bottom-right position is reached. The response must also ensure that the traversal does not stop too early (by stopping at any point other than the bottom-right corner) or too late (by continuing past the bottom-right corner). Point 6 was earned because `getNextLoc` is called with two `int` arguments inside the loop. To earn this point, the response must make a syntactically valid call to `getNextLoc` with two `int` arguments. Point 7 was earned because `getRow` and `getCol` are called on `Location` objects. To earn this point, the response must call both methods on a `Location` object without arguments. The calls may occur on any `Location` object. Missing parentheses on a parameter-less method is one of the minor errors for which no penalty is assessed on this exam. (See the "No Penalty" section of the Scoring Guidelines for a complete list.) Point 8 was earned because elements of `grid` are accessed at positions derived from the call to `getNextLoc`. Point 9 was not earned because the solution does not include in the sum the first location visited on the path. To earn this point, the response must sum the values visited along its path, even if the path is incorrectly constructed. Note that the sum must include the first and last values along the path; there are multiple strategies for including those positions. This response correctly returns the computed sum; if the response had calculated the sum correctly but did not return the sum, this point would have been earned, as the return is not assessed in part (b).

Sample: 4B
Score: 6

In part (a) point 1 was not earned because the response allows out-of-bounds access of grid elements when `row == grid.length - 1` or `col == grid[0].length - 1`. Point 2 was earned because the response accesses elements at `grid[row][col + 1]` and `grid[row + 1][col]`. Point 3 was earned because the response returns an appropriate `Location` object in all four cases.

Question 4 (continued)

In part (b) point 4 was earned because the response initializes a variable (`sum`) to an element of `grid` and increments `sum` with elements of `grid`. Point 5 was not earned because the path algorithm stops too late. The response loops until the bottom-right position of `grid` is reached and then calls `getNextLoc` on that position, violating the precondition on `getNextLoc`. Point 6 was not earned because `getNextLoc` is incorrectly called using the name of the class, `GridPath`. Point 7 was earned because `getRow` and `getCol` are called on `Location` objects. Point 8 was earned because elements of `grid` are accessed at positions derived from the call to `getNextLoc`. The incorrect syntax on the call to `getNextLoc` is assessed in point 6. Point 9 was earned because the response computes the sum of values along the visited path.

Sample: 4C
Score: 2

In part (a) point 1 was not earned because the response incorrectly guards against out-of-bounds access by comparing grid elements to `null`. Point 2 was earned because the response accesses elements at `grid[row][col + 1]` and `grid[row + 1][col]`. Point 3 was not earned because no `Location` object is created.

In part (b) point 4 was not earned because the variable `sum` is incremented with `Location` objects instead of grid values. Point 5 was not earned because an out-of-bounds grid element is accessed in the loop condition. Point 6 was earned because `getNextLoc` is called with two `int` arguments inside the loop. Point 7 was not earned because `getRow` and `getCol` are never called. Point 8 was not earned because the only valid grid element accessed in the method is in the initialization of `current`; thus, no grid element is derived from a method call. Point 9 was not earned because the sum fails to include the first visited location.