As mentioned before, a single pdf file shall be submitted for all non-code exercises. I have to deduct 2P. now for not adhering to the guidelines.

- **Avoid Vague Statements:**
  - “Too Many if Statements” should specify examples or point to exact areas in the code where this is a problem.
  - “Loops Seem Inefficient” needs a clearer explanation of the inefficiency, like time complexity or specific scenarios where performance might degrade.
- **Professional Tone:**
  - The phrase “Basic Programming 101 knowledge ;)” is dismissive and unprofessional. Keep feedback constructive and neutral.
- **Focus on Actionable Feedback:**
  - Instead of “it’s not clear why a try-catch block is used,” suggest explicitly clarifying the exceptions being handled in the code with comments or documentation.
- **Evidence-Based Feedback:**
  - “There might be a security risk” is speculative. It’s better to highlight a specific code area and why it might be vulnerable or request more context to verify concerns.
- **Provide Concrete Suggestions:**
  - Instead of saying “try to make it simpler,” suggest specific refactoring techniques, such as introducing helper methods or reducing nesting.
- **Testing Suggestions:**
  - Simply stating “there could be more tests” is too vague. Identify specific test cases that might be missing, such as edge cases or scenarios related to input validation.
- **Focus on Specificity:**
  - “Add more comments” could point to specific parts of the code that are unclear or require better explanation.
- **Be Consistent:**
  - The reviewer praises the tests in one place but then says they are insufficient. Ensure feedback is aligned or clarify the scope of improvement.
- **Avoid Generalizations:**
  - Avoid statements like “the code could be more efficient” without identifying specific inefficiencies or suggesting optimizations.
- **Mention Positive Aspects Explicitly:**

- Highlight specific strengths of the code to balance the critique and motivate the developer. For example, mention that the exception validation is a good safeguard.

Σ5/5P.

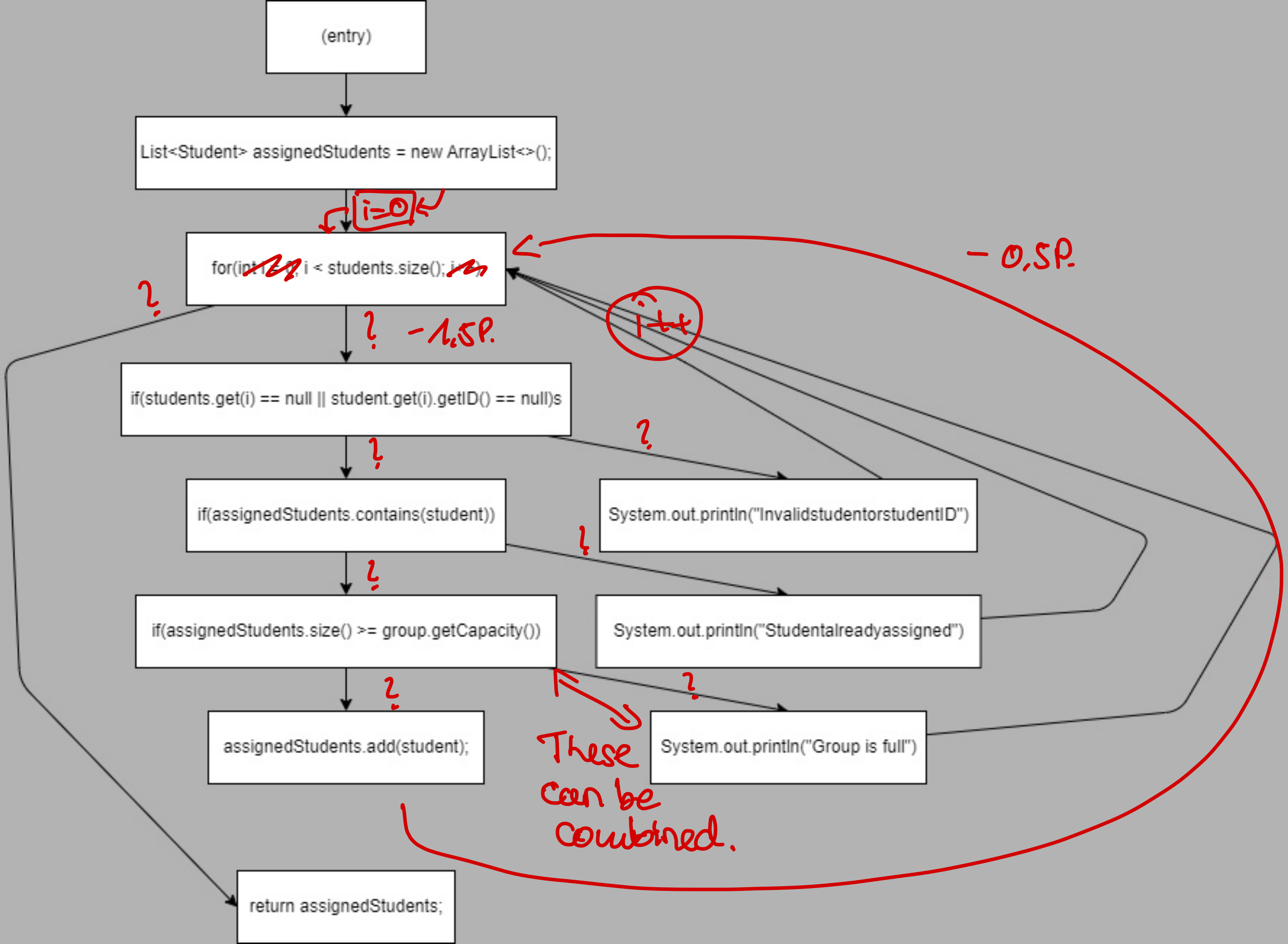
Test Cases	TC1	TC2	TC3	TC4	TC5	TC6	TC7
(a < 0)	X		X	X			
(a = 0)		X				X	
(a > 0)					X		X
(b < 0)			X				
(b = 0)						X	
(b > 0)	X	X		X	X		X
(c < 0)				X			
(c = 0)		X	X	X			
(c > 0)	X				X		X
Exception			X			X	
Input (a)	-5	0	minInt	-1	50	0	maxInt
Input (b)	10	10	-1	maxInt	10	0	1
Input (c)	5	5	0	minInt	5	5	1
Expected Output	0	0	Exception	0	0	Exception	1
Result	0	0	Exception	0	0	Exception	1

min/max cases missing.

−1,5P.

Parameters	Description
a	Refers to the totalStudents parameter (number of students to assign)
b	Refers to the groupSize parameter (number of students per group)
c	Refers to the availableGroups parameter (number of groups available)

Σ5,5/7P.



```
(entry)
List<Student> assignedStudents = new ArrayList<>();
for(i~~nt i = 1;~~ i < students.size(); ~~i++~~)
if(students.get(i) == null || student.get(i).getID() == null)s
System.out.println("Invalidstudentorstudent ID")
if(assignedStudents.contains(student))
System.out.println("Studentalreadyassigned")
if(assignedStudents.size() >= group.getCapacity())
System.out.println("Group is full")
assignedStudents.add(student);
return assignedStudents;
```

- **Coverage Analysis:** What are the numbers?
  - **Test 1 (testInvalidStudentId):** Covers Nodes 1 → 2 → 3 → 4 → 10 → 11. ✓
  - **Test 2 (testSuccessfulAssignment):** Covers Nodes 1 → 2 → 3 → 5 → 7 → 9 → 10 → 11. f
- **Uncovered:** Nodes 6 and 8 (conditions for already assigned and group full are not tested).
- **Coverage:** ~~7/9~~ statements covered: 77.78%

You have 10 statements in your CFG. -1P.

2. **Branch Coverage**

- **Decisions:**
  - Node 3: Null student/ID check (True/False).
  - Node 5: Already assigned check (True/False).
  - Node 7: Group full check (True/False).
- **Coverage Analysis:**
  - **Test 1 (testInvalidStudentId):** Covers Node 3 (True). Does not test False branches of Nodes 5 and 7.
  - **Test 2 (testSuccessfulAssignment):** Covers Node 3 (False), Node 5 (False), and Node 7 (False). Does not test True branches of Nodes 5 and 7.
- **Coverage:** 4/6 branches covered: 66.67% Ff

3. **Condition Coverage**

- **Conditions:**
  - Node 3: (students.get(i) == null || students.get(i).getID() == null)
  - Node 5: (assignedStudents.contains(student))
  - Node 7: (assignedStudents.size() >= group.getCapacity())
- **Coverage Analysis:**
  - **Test 1:** Node 3: Tests students.get(i) == null (True). Does not test students.get(i).getID() == null (True/False).
  - **Test 2:** Node 3: Tests students.get(i) == null (False) and students.get(i).getID() == null (False). Node 5: Tests (assignedStudents.contains(student)) (False). Node 7: Tests (assignedStudents.size() >= group.getCapacity()) (False).
- **Uncovered:** Node 3: students.get(i).getID() == null (True). Node 5 and 7: True branches.
- **Coverage:** 5/8 conditions covered: 62.5% Ff

4. **Path Coverage**

- **Paths:**
  - 1 → 2 → 3 (True) → 4 → 10 → 11.
  - 1 → 2 → 3 (False) → 5 (False) → 7 (False) → 9 → 10 → 11.
  - 1 → 2 → 3 (False) → 5 (True) → 6 → 10 → 11.
  - 1 → 2 → 3 (False) → 5 (False) → 7 (True) → 8 → 10 → 11.

5 paths.

- **Coverage Analysis:**
  - **Test 1:** Covers Path 1.
  - **Test 2:** Covers Path 2.
- **Uncovered:** Paths 3 and 4.
- **Coverage:** 2/4 paths covered: 50%

(✓) −0,5P.

Observations and Suggestions - **Uncovered Cases:** - Already Assigned Students: Add a test case where a student is already assigned. - Full Group: Add a test case where the group is full. - Null IDs: Add a test case where students.get(i).getID() is null. - **Improved Tests:** - Add more test cases to achieve 100% statement, branch, condition, and path coverage. - Explicitly test each edge case (e.g., empty group, multiple students, duplicate students).

Explanation missing on how loops are handled for path coverage. −1P.

Σ 3,5/8P.