Assignment 2

This assignment is worth 15% of the total course marks.

Due date: The assignment is due at **4pm on Friday 24 October.** Late assignments will attract a penalty of 10% of the total assignment mark per day, and submissions more than 5 days late will not be marked. Only extensions on Medical Grounds or Exceptional Circumstances will be considered, and in those cases students need to submit an application for extension of progressive assessment form (http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf) to the lecturer (email is acceptable) or the ITEE Enquiries desk on Level 4 of GPSouth Building) prior to the assignment deadline.

Resources:

The Assignment 2 zip file, available from Blackboard (under Learning Resources/Course Materials), contains the Java classes that are mentioned in this handout. You will need those source files to complete this assignment.

Your Java Build Path will need to include the JUnit 4 Library to run the JUnit tests.

Problem overview:

The University of Woolloomooloo receives donations from which it is able to fund a number of projects, such as building a new library, repairing the façade of a heritage building, paying for scholarships etc. Each donation is conditional, in that it may only be spent on projects approved by the donor. Having so many donations, but also so many projects to undertake, the University has recently got itself into some strife: having committed to fund projects that it cannot afford, given their existing donations and the conditions upon them. Not wanting to make the same mistake twice, they want some software that can help them determine whether or not they can completely fund a set of projects, given the conditional donations that are available to them. If they are able to completely fund the set of projects, they want to know how to allocate funds to those projects from their pool of donations.

More specifically, the University of Woolloomooloo wants a program that takes a list of donations and a set of projects as input where:

- each *donation* has a unique name, a total donation amount (in whole dollars) and a set of projects that the donation may be spent on. Initially, no dollars from any of the donations have been spent.
- each *project* has a unique name, a total cost, and initially has no allocated funds. The total cost of a project is in whole dollars.

and either

- returns *false* if there no way to completely fund all of the given projects using the donations, leaving both the input list of donations and set of projects unmodified;
- or otherwise returns *true* and allocates to each project funding from the donations. The allocation to each project must be complete (accounting for the entire cost of that project exactly), and it may not violate the conditions of the donations (e.g. you can only allocate money from a donation to the projects that it may be spent on). The unspent funds in each donation should reflect the allocation that was made.

Example 1: Given the donations and projects

donations			
name	total	unspent	possible projects
D0	10	10	P0, P1
D1	10	10	P1, P2
D2	5	5	P0
D3	5	5	P2

projects		
name	cost	allocation
P0	10	-
P1	10	-
P2	10	-

the program should return the value *true*, updating the donations and projects by making a suitable allocation of donations to projects:

donations			
name	total	unspent	possible Projects
D0	10	0	P0, P1
D1	10	0	P1, P2
D2	5	0	P0
D3	5	0	P2

projects		
name	cost	allocation
P0	10	(D0, 5), (D2, 5)
P1	10	(D0, 5), (D1, 5)
P2	10	(D1, 5), (D3, 5)

Example 2: Given the donations and projects

donations			
name	total	unspent	possible projects
D0	10	10	P0, P1, P2
D1	20	20	P0

projects		
name	cost	allocation
P0	10	-
P1	10	-
P2	10	-

the program should return *false*, and not modify the donations or projects in any way. That is because there is no possible way to completely fund all of the projects using the conditional donations available.

Note that not all projects may have the same cost, as in the above examples. Money may be left over in the donations after a complete allocation is made. In general, there may be many different compete allocations of funds that are valid.

Task 1: Naïve recursive algorithm implementation [3 marks]:

One way to implement the program that the University of Woolloomooloo wants is to implement a naïve recursive algorithm that basically tries to allocate a dollar at a time to each possible project it could be allocated to:

```
Algorithm canAllocate(donations, projects)
Input:   A list of donations that have not been allocated to any projects yet, and a set of
         projects that currently have no allocated funds.
Output: Returns false if there is no way to completely fund the projects using the donations,
         leaving both the inputs unmodified; otherwise returns true and allocates funding
         from the donations to the to projects.

return canAllocateHelper(donations, projects, 0)

Algorithm canAllocateHelper(donations, projects, i)
Input:   A list of donations (that may be partially spent), a set of projects that may have
         partial (or complete) allocations of funds, and an index 0 <=i <= donatons.size()
Output: Returns false if there is no way to complete the funding of the projects using the
         (unspent portion of the) donations from index i onwards, and does not modify the
         inputs; or returns true otherwise, completing the allocation of funds to each project.

if all of the projects have been completely allocated then
         return true
if index i is at the end of the donations list (i.e. no more donations to allocate)
         return false

donation ← donations.get(i)

if there are no funds left in the donation to spend or there are no projects that donation
could be spent on that still need funding then
         return canAllocateHelper(donations, projects, i+1)

for each of the projects p that donation could be spent on, and still needs funding
         allocate one dollar of donation to p
         if canAllocateHelper(donations, projects, i) then
                  return true
         else
                  deallocate one dollar from donation to p

return false
```

Implement the naïve recursive algorithm above in the method `canAllocate` that belongs to the class `a2.NaiveAllocator` that is available in in the assignment zip file. (Within the confines of this recursive approach) do so as efficiently as you can - you are allowed to choose exactly what parameters you want your helper method to take and so forth as long as you implement the recursive solution correctly.

You must not change the name or signature of the method `canAllocate`, or the name or package of the class to which it belongs. Do not change class `a2.Donation` or `a2.Project` in any way. You should also not write any code that is operating-system

specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine. Your source files should be written using ASCII characters only.

You may (and should) use the Java 7 SE API in your implementation, but no other libraries should be used. (It is not necessary and makes marking hard.)

Any extra classes that you write should be included in the file `a2.NaiveAllocator` as private nested classes. Any additional class methods should also be private.

To help you get your solution right, you should write you own JUnit tests in `a2.test.NaiveAllocatorTest` (some basic tests are included to get you started). Tests that you write will not be marked. We will test your implementation with our own tests.

Task 2: Naïve recursive algorithm analysis and justification [4 marks]:

Let n be the number of donations, m the number of projects, x the total number of dollars worth of donations available, y the total number of dollars required to fund all of the projects.

What is a lower bound on the worst-case time complexity of your recursive implementation from Task 1 in terms of parameters n, m, x and y?

Express your answers in big-Omega notation, making the bound as tight as possible, and simplifying your answer as much as possible. Justify your answer. Describe when the worst-case behaviour (w.r.t. time complexity) would occur – give an example.

Task 3: Iterative algorithm implementation [8 marks]:

There is another way to implement the program using an iterative approach.

We say that x dollars of funding could be *transferred* from one project p to another q when x dollars of donations that have been currently allocated to p may also be spent on project q. For example for the following donations

donations			
name	total	unspent	possible projects
D0	10	5	P0, P1, P3
D1	10	0	P0, P1
D2	5	0	P0, P2

if the donations (D0, 5), (D1, 10) and (D2, 5) were currently allocated to project P0 then up to 15 dollars could be transferred from P0 to P1.

Given a set of donations and an incomplete allocation of those donations to projects, we know that you cannot completely fund the projects using the donations if there does not exist an integer x>0 and a list of n>0 distinct projects

path = $[P_0,\ldots,P_{n-1}]$

from the set of projects such that:

1. Project path[n-1] is currently underfunded by at least x dollars
2. x dollars of funding could be allocated from the available donations to project path[0].
3. for each i=n-2 to 0, x dollars from project path[i] could be transferred to project path[i+1]

That means that we can find a complete allocation of donations to projects, if one exists, by repeatedly:

- finding a non-negative integer x and a path of n>0 distinct projects path satisfying properties (1)-(3) above
- for i=n-2 to 0 transferring x dollars of allocated funds from project path[i] to path[i+1]
- allocating x dollars from the available donations to project path[0]

until a complete allocation is found. Such a path of projects may be efficiently found using a graph traversal algorithm like depth-first search or breadth-first search (treating the projects like vertices and the ability to transfer funds from one project to another like an edge). If at any time during the process we discover that no such path exists, then we can conclude that no allocation is possible, de-allocate any allocations that we have tried to make and return false. The algorithm will terminate because at least one extra dollar is allocated on each iteration, and the total dollars required to fund all of the projects completely is finite.

Example 3: Starting from the following donations and projects,

donations			
name	total	unspent	possible projects
D0	10	10	P0, P1
D1	10	10	P1, P2
D2	5	5	P0
D3	5	5	P2

projects		
name	cost	allocation
P0	10	-
P1	10	-
P2	10	-

we could find the solution provided in Example 1, using the iterative approach by:

1. First observing that \$10 could be transferred to P0 along path=[P0], yielding temporary allocation:

donations			
name	total	unspent	possible projects
D0	10	0	P0, P1
D1	10	10	P1, P2
D2	5	5	P0
D3	5	5	P2

projects		
name	cost	allocation
P0	10	(D0, 10)
P1	10	-
P2	10	-

2. Then observing that \$10 could be transferred to P2 along path=[P2], yielding:

donations			
name	total	unspent	possible projects
D0	10	0	P0, P1
D1	10	0	P1, P2
D2	5	5	P0
D3	5	5	P2

projects		
name	cost	allocation
P0	10	(D0, 10)
P1	10	-
P2	10	(D1, 10)

3. $5 could be transferred to P1 along path=[P2, P1], giving:

donations			
name	total	unspent	possible projects
D0	10	0	P0, P1
D1	10	0	P1, P2
D2	5	5	P0
D3	5	0	P2

projects		
name	cost	allocation
P0	10	(D0, 10)
P1	10	(D1, 5)
P2	10	(D1, 5), (D3, 5)

4. $5 could be transferred to P1 along path=[P0, P1], giving us a complete allocation:

donations			
name	total	unspent	possible projects
D0	10	0	P0, P1
D1	10	0	P1, P2
D2	5	0	P0
D3	5	0	P2

projects		
name	cost	allocation
P0	10	(D0, 5), (D2, 5)
P1	10	(D1, 5), (D0, 5)
P2	10	(D1, 5), (D3, 5)

Example 4: Starting from the following donations and projects,

donations			
name	total	unspent	possible projects
D0	10	10	P0, P1, P2
D1	20	20	P0

projects		
name	cost	allocation
P0	10	-
P1	10	-
P2	10	-

we can show that there is no possible allocation for Example 2 by observing that:

1. First observing that $10 could be transferred to P0 along path=[P0], yielding temporary allocation:

donations			
name	total	unspent	possible projects
D0	10	0	P0, P1, P2
D1	20	20	P0

projects		
name	cost	allocation
P0	10	(D0, 10)
P1	10	-
P2	10	-

2. $10 could be transferred to P1 along path=[P0, P1], yielding temporary allocation:

donations			
name	total	unspent	possible projects
D0	10	0	P0, P1, P2
D1	20	10	P0

projects		
name	cost	allocation
P0	10	(D1, 10)
P1	10	(D0, 10)
P2	10	-

3. Then we can observe that although P2 is underfunded, there is no path of money from the unallocated donations to project P2 and so we can deallocate our existing allocations and return false.

Your task is to implement this iterative algorithm for solving the problem as efficiently as you can in the method `canAllocate` that belongs to the class `a2.IterativeAllocator` that is available in the assignment zip file.

You must not change the name or signature of the method `canAllocate`, or the name or package of the class to which it belongs. Do not change class `a2.Donation` or `a2.Project` in any way. You should also not write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine. Your source files should be written using ASCII characters only.

You may (and should) use the Java 7 SE API in your implementation, but no other libraries should be used. (It is not necessary and makes marking hard.)

Any extra classes that you write should be included in the file `a2.IterativeAllocator` as private nested classes. Any additional class methods should also be private.

To help you get your solution right, you should write you own JUnit tests in `a2.test.IterativeAllocatorTest` (some basic tests are included to get you started). Tests that you write will not be marked. We will test your implementation with our own tests. Our tests will include large tests that will check that your implementation will work on larger sized problems and will be equipped with timeouts.

Task 4: COMP7505 ONLY: Iterative algorithm analysis and justification [3 marks]

What is the worst-case time complexity of your iterative implementation in Task 3?

Express your answers in big-Oh notation, making the bound as tight as possible, and simplifying your answer as much as possible. Justify your answer. You should express your answer in terms of suitably chosen measures of the input size of the program, and you must explain what those parameters are.

Practical considerations:

If necessary, there may be some small changes to the files that are provided, up to 1 week before the deadline, in order to make the requirements clearer, or to tweak test cases. These updates will be clearly announced on the Announcements page of the course Blackboard site, and during the lectures.

Submission: Submit you source code files `NaiveAllocator.java` and `IterativeAllocator.java` as well as your written answers in `report.pdf` electronically using Blackboard according to the exact instructions on the Blackboard website:

https://learn.uq.edu.au/

Answers to Tasks 2 (and 4 for COMP7505 students) should be clearly labelled and included in file `report.pdf`.

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the files listed above. You are responsible for ensuring that you have submitted the files that you intended to submit in the way that we have requested them. You will be marked on the files that you submitted and not on those that you intended to submit. Only files that are submitted according to the instructions on Blackboard will be marked

Evaluation: Your assignment will be given a mark according to the following marking criteria. For COMP3506 the total marks possible is 15, and for COMP7505 the total marks possible is 18. The assignment is worth 15% for both courses.

Code must be clearly written, consistently indented, and commented. Java naming conventions should be used, and lines should not be excessively long (> 80 chars). Marks will be deducted in code analysis and efficiency questions if we cannot read and understand your solution to check your analysis or evaluate the efficiency of your solution.

Task 1: [3 marks]

Given that the solution does implement the recursive algorithm, and does not violate any restrictions (using forbidden libraries etc), marks will be allocated based on the outcome of running test cases:

- All of our tests pass 3 marks
- At least 66% of our tests pass 2 marks
- At least 33% of our tests pass 1 mark
- Work with little or no academic merit 0 marks

Note: code submitted with compilation errors will result in zero marks in this section. Code that violates any stated restrictions or doesn't implement the recursive algorithm given will receive zero marks.

Task 2: [4 marks]

- The algorithm analysis and justifications (including example) are correct and complete. 4 marks
- The algorithm analysis and justifications (including example) contain only very minor errors or omissions. 3 marks
- The algorithm analysis and justifications (including example) are reasonable but contain some errors or omissions. 2 marks
- The algorithm analysis and justifications (including example) contain many errors or omissions or a major error or omission. 1 mark
- Work with little or no academic merit 0 marks

Note: A mark of 0 will be given for this part if no reasonable attempt was made to complete Task 1 or the solution to Task 1 is difficult to read and comprehend, or incorrect.

Task 3: [8 marks]

Given that the solution does efficiently implement the iterative algorithm, and does not violate any restrictions (using forbidden libraries etc), marks will be allocated based on the outcome of running test cases. Our tests will include large tests that will check that your implementation is efficient: they will be equipped with timeouts.

- All of our tests pass — 8 marks
- At least 85% of our tests pass — 7 marks
- At least 75% of our tests pass — 6 marks
- At least 65% of our tests pass — 5 mark
- At least 50% of our tests pass — 4 marks
- At least 35% of our tests pass — 3 mark
- At least 25% of our tests pass — 2 mark
- At least 15% of our tests pass — 1 mark
- Work with little or no academic merit — 0 marks

Note: code submitted with compilation errors will result in zero marks in this section. Code that violates any stated restrictions or doesn't efficiently implement iterative algorithm given will receive zero marks.

Task 4: COMP7505 ONLY [3 marks]

- The algorithm analysis and justifications are correct and complete. — 3 marks
- The algorithm analysis and justifications are reasonable but contain some errors or omissions. — 2 marks
- The algorithm analysis and justifications contain many errors or omissions or a major error or omission. — 1 mark
- Work with little or no academic merit — 0 marks

Note: A mark of 0 will be given for this part if no reasonable attempt was made to complete Task 3 or the solution to Task 3 is difficult to read and comprehend, or incorrect.

School Policy on Student Misconduct: You are required to read and understand the School Statement on Misconduct, available on the School's website at:

http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct

This is an individual assignment. If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied.

If you are under pressure to meet the assignment deadline, contact the course coordinator **as soon as possible**.