COMP 225 Algorithms and Data Structures
Assignment 1 Specification

Due on Friday April 29, 2016 (Week 7)
(worth 15% of overall unit marks)

Objectives - To gain experience in programming with various data structures; to practise building larger applications from a variety of data structures and algorithms; to adapt basic algorithm design strategies to achieve the best performance possible within the constraints of the given data structures.

The Taxman Cometh - Taxman is a number game that is played against the computer. The human player chooses an upper limit, say N, and the game is played with the integers from 1 to N. During the course of the game, the human and the computer each accumulate a total. The objective of the game is for the human to accumulate a larger total than the computer, hereafter referred to as the Taxman.

The player's total accumulates by selecting one of the numbers left in the game. The Taxman then gets *all* the numbers left in the game that are divisors of the player's chosen number. Once the numbers are used (by either the player or the Taxman), they are removed from the game.

There is one major restriction on the numbers that the player may select. As in real life, the Taxman must always get something, so, the player can never select a number unless at least one proper divisor remains in the game. Once no numbers with divisors remain, the Taxman gets all the numbers left and the game is over. The winner is the one who has a larger total.

For example, suppose that the upper limit is 6 and the game is played with the numbers 1, 2, 3, 4, 5, and 6. If the player is greedy and chooses 6, the Taxman gets all the divisors of 6, namely 1, 2, and 3. But now, the only numbers left in the game are 4 and 5. Neither has a divisor left in the game, so the Taxman gets those too, and wins 15 (1 + 2 + 3 + 4 + 5) to 6. However, if the player is a bit smarter and chooses 5 first, the player gets 5 and the Taxman gets 1. Now the numbers remaining are 2, 3, 4, and 6, and the smart player chooses 4 (before 6), giving the Taxman 2. Finally, the player chooses 6 and wins 15 (5 + 4 + 6) to 6 (1 + 2 + 3). Notice that, if the player had chosen 6 before 4, the Taxman would have received both 2 and 3, leaving 4 without any divisors in the game. After the Taxman collects 4, the final score is a much closer 11-10 victory over the Taxman. When played with more than 50 numbers, the game is quite challenging.

Note that Robert K. Moniot from Fordham College has made a demo version of the game of Taxman which is available at `http://www.dsm.fordham.edu/~moniot/taxman.html`. You may want to play the game there with various numbers (say, for N = 6, 11, 15, 25) to get a better feel for it.

Your Task - There will be three stages in this assignment.

Stage I Your task in this stage is to write a program that plays the role of the Taxman. It will first ask for the upper limit N (≥ 2), and then will display the running scores together with the remaining numbers; obtain a choice from the player and update both the player's and the Taxman's scores. It will determine the winner when the player has no more legal choices.

We know that the game is over when there are no numbers with divisors left in the game. However, we don't need to check every number left in the game to see if it has any divisors. In fact, the only integers that can ever qualify as divisors are 1 to (N `div` 2) and each of these numbers will be the divisor of some other number in the game. When this collection of possible divisors is empty, the game is over.

Your program must include a Taxman class (to encapsulate the main functionality of your program). There should be only one call from the `main()` procedure to a suitable method of this class, say `run()`, to start off the Taxman game. Your program must also include a Player class which implements the required functionality for the player including the choice of the next number. Any other classes may be provided as needed/desired.

You must use sets as the primary data structure in the Taxman program. You may naturally represent the original collection of numbers as a set of integers from 1 to the upper limit N. The collection of divisors of a number, the collection of the remaining legal choices can all be represented by sets of integers.

Note that you are not required to implement your own set class (or any other collection class) for the numbers in the game as part of the assignment; instead, you can use one of the set interfaces that are available in the java.util package but ideally should pick one that provides all the set operations required for the Taxman program. Also there is **no need** to implement a GUI for the assignment. If you do that, you will not get any extra marks.

You should test your program for Stage I with various values of N (as a minimum, you should try for N = 6, 11, 15, 25).

Stage II In this stage, you are required to extend the program so that it plays the role of the Player as well as the Taxman. In Stage I of the game, after reading N, the Player selects the numbers and the Taxman responds until the game is over. The problem is: how does the Player make a sensible move when presented with a number of choices?

So, before proceeding to write the program for this stage, you must devise a greedy strategy for this purpose. There are of course many possible greedy strategies we can employ. Note that when we employ a greedy strategy, we just take whatever number it suggests to us and keep going; there is no going back to try a different number.

A naïve greedy strategy (hereafter referred to as NAIVE) is to take the largest number left that still has divisors in the game. That is, at every turn, take the *largest legal choice*.

In the game with integers from 1 to 6, the NAIVE strategy, like the greedy player described above in Stage I, takes 6 first and immediately loses 6 to 15. This doesn't mean that the NAIVE strategy always leads to losses. For example, consider the game with the upper limit of 11 which is played with the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11. According to the NAIVE strategy, we must choose 11 first, and the Taxman gets 1. Then we take 10 as it is the largest number left in the game, and the Taxman gets its divisors 2, 5. Next, we take 9, and the Taxman gets its divisor 3. Finally we take 8 and the Taxman gets its divisor 4. The Taxman then gets the remaining numbers 6, 7 as they have no divisors left. And the player wins with the score of 38 against 28.

Considering all the upper limits from 2 to 11, the NAIVE strategy will in fact lead to losses of the games with N = 4, 6, 7, 8, 9, 10 but wins of the games with N = 2, 5, 11. So, that is not really very smart especially when the (human) player can win all of those games easily (except for N = 3 which is a tie), maybe by working out all possible sequences of choices on a piece of paper.

Now the question is whether you can devise a better strategy than the NAIVE strategy or even one that always wins.

Your programming task for this stage is to extend the Taxman program with the implementation of a greedy strategy of your choice. As a first attempt, you may implement the NAIVE strategy, however, for *better grades* (see mark allocation below), you must devise a different and more intelligent strategy (hereafter referred to as GREEDY to distinguish it from NAIVE).

You will modify the part of the program (in the Player class) that obtains a choice from the player, and implement a greedy strategy (either NAIVE or GREEDY) to choose the next number for the player. Now the only input to the extended program is the upper limit N. Note that there will be no further interaction with the user after reading in the upper limit N.

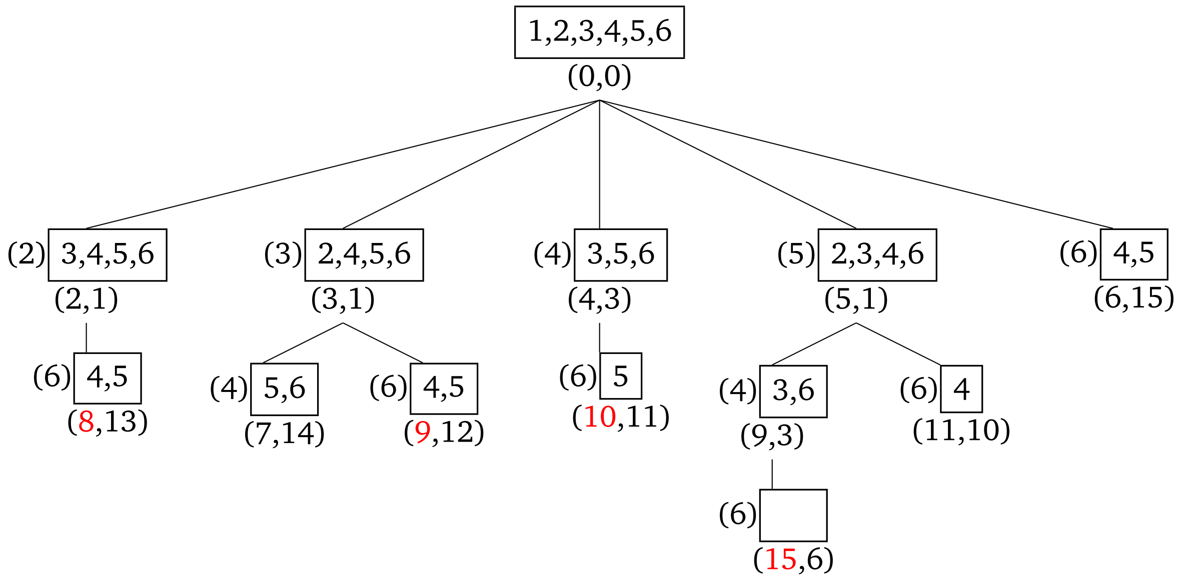
You may test your program with games for all values of N up to a certain limit, say 25. Does you program find a *winning* or a *non-losing* sequence of choices for every given game? Note that for N = 3 , we know that the best outcome for the player is a tie, so, it is not possible to find a winning strategy for every game. In theory, it is not known whether there is a winning strategy for every integer value N, but we are not really concerned with that in this assignment.

Stage III Now that you have implemented a greedy strategy in Stage II, you may want to investigate the game a little deeper to see if the adopted strategy really helps the player find a sequence of legal choices which yields an *optimal* total for any given game. After all, the player would really like to maximize his/her winning margin. An optimal total for a game is no worse than any other total obtained by following any sequence of legal choices.(Note that there may or may not be the *optimum* total for a given game.)

For the game with the upper limit of 6, we know that there are two winning sequences of legal choices. The optimal strategy here is to take 5, 4, 6, in that order. (This strategy also happens to be the optimum one.)

So, in Stage III, you will extend the Taxman program from Stage II with the implementation of a brute-force algorithm for searching a sequence of legal choices which yields an optimal total (hereafter referred to as OPTIMAL). This will require to check all the sequences of legal choices possible while keeping track of an optimal one. You may employ *recursive backtracking* as a design strategy for this brute-force algorithm.

Let us have a look at the complete game tree given below for the upper limit of 6. In the tree, the nodes framed in rectangles show all the remaining numbers at each stage of the game. The initial stage of the game is shown in the top-most node (the root) of the tree. Every node has as many successors as there are legal choices and a successor of the node shows all the remaining numbers after the player's choice and all of its divisors are removed from the game. The numbers in brackets to the left of each node shows which number is chosen from the legal choices available in the predecessor of that node. As there are 5 legal choices (2,3,4,5,6) for the root node, it has 5 successors (branches), one for each legal choice. When there are no legal choices left at a particular node, the node obviously has no successors and the winner can be determined. Such a node is called a leaf node.



In the game tree, the running totals are given in the bracket notation (x, y) under the nodes where x represents the player's total and y the Taxman's. In the beginning of the game (the root node), the running totals are (0,0). The totals in the leaf nodes, then, represent the *final outcome* of the game. There are two winning leaf nodes, with totals of (15,6) and (11,10). Here (15,6) means that the player gets 15 while the Taxman 6 with the optimum margin. At the node with 3,6, we can only choose 6 as 3 is its divisor, and then the Taxman gets 3 and we are left with an empty set in the only successor of this node and the game is over.

In effect, your algorithm will *navigate* this game tree using recursive backtracking, that is, there is no need to store the tree as a separate data structure. However, your algorithm will need to keep track of the sequence of choices that gives the current largest total for the game; the sequence of choices and the largest total will need to be updated when it reaches a leaf node with a total which is better than the current largest total. In the tree, the player's totals in red colour represent the updates with the final total being the last one when going from left to right through the leaf nodes in the tree.

You may also observe that in a game with a very large upper limit, there may be a lot of sequences of legal choices with no hope of a win (this is not so obvious in the small game tree for N = 6). This could happen for instance when the Taxman's total is already larger than the total the player can hope to get for the whole game. This observation presents an opportunity for optimization of the recursive backtracking algorithm which you might also want to implement for better grades. (There may also be other optimization strategies, so you may want to think carefully about which one to implement.) We will refer to the optimized version of the algorithm as OPTIMIZED.

At the end of this stage, you will have a pretty good idea about your greedy strategy (with or without any optimization). In case it cannot always find a sequence of legal choices which yields an optimal total for a given game (for N $\leq$ 25) , you may think about another greedy strategy that might do that and, if you wish, implement it instead as part of this assignment.

Your submission You must submit a single zip file named assignment1.zip on ilearn for your Eclipse Java project with all the classes in it.

You must also submit a report called assignment1.pdf which has the listing of all the program code including brief descriptions of your classes, outputs from several game plays for Stage I (with N = 6, 11, 15, 25), outputs from game plays for Stages II & III (with N = 2, ..., 25), and a discussion (of a few pages maximum) of the design decisions you have made, and the approaches you have considered, especially for Stages II (GREEDY) & III (OPTIMIZED). There is no particular structure for your discussion, so, you can highlight the most important aspects of your approach & the program but be succinct.

To contrast the difference (if any) between the greedy and brute-force strategies, the outputs of Stages II & III can be presented together as follows (-2 represents a loss with a margin of 2):

N	Strategy (Greedy)	Winning margin	Strategy (Optimal)	Winning margin
2	2	1	2	1
3	3	0	3	0
4	4	-2	3,4	4
⋮	⋮	⋮	⋮	⋮
25	...	...	...	...

You must submit your assignment by Friday of Week 7 (the exact time will be announced later). Note that **no late submissions** will be accepted, even if your computer or the submission system crashes a few hours before the deadline. To avoid this sort of a problem, it is recommended that you submit an early version of your assignment solution several days before the deadline and update it as required. In fact, you can make a submission as soon as a stage of the assignment is completed.

Note that there is **no auto-marking** for this assignment. However, your programs may be run by the markers, so, make sure that they compile and produce output.

Mark allocation (15%) This assignment is structured to allow you to decide how much effort you want to expend for the return in marks that you might hope for. Here is what is required to obtain marks in one of the performance bands for this assignment:

- **Pass** (P, 50% - 64%) A successful implementation of Stages I & II (NAIVE).
- **Credit** (CR, 65% - 74%) A successful implementation of Stages I, II (GREEDY); however, to achieve full marks, the chosen greedy strategy has to be reasonably intelligent with much better performance than NAIVE.
- **Distinction** (D, 75% - 84%) As a minimum, a successful implementation of Stage III (OPTIMAL).
- **High Distinction** (HD, 85% -100%) As a minimum, a successful implementation of Stage III (OPTIMAL & OPTIMIZED) plus a comparison of the runtime performance of OPTIMAL and OPTIMIZED. The runtime performance can be presented in a table as shown below.

N	Strategy (Optimal)	Winning margin	Runtime	Strategy (Optimized)	Winning margin	Runtime
2	2	1		2	1	
3	3	0		3	0	
4	3,4	4		3,4	4	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
25	…	…	…	…	…	…

Changelog

19/04/2016: Clarified the requirements for grade allocation.

19/04/2016: Restricted the upper limit of N to 25 in the GREEDY & OPTIMAL strategies.

20/04/2016: Revised the requirements for grade allocation.