

Make Up Set

Total Points: 30

Three problems must be implemented for full credit. Each section will state how many problems to implement within it. The required (starred) problem marked in the set must be implemented by everyone. See 2436 Grading Guide Sheet for additional grading/submission information. Partial credit will be given. (10 points each)

Section One- Comprehensive Review. Choose two problems.

1. A rectangular two-dimensional grid shows the style of houses at each location. A neighborhood is defined to be a contiguous set of houses of the same type that exceed some minimum number of houses. The program will use 4 different ways of defining neighbors:

- 4 neighbors normal- Cells have at most 4 neighbors, the cells above, below, left, and right of a cell. Cells on the edge only have 3 neighbors. The 4 corner cells only have 2 neighbors.
- 4 neighbors wrap- All cells have 4 neighbors. The edge and corner cells neighbors wrap around to the other edges.
- 8 neighbors normal- Cells have at most 8 neighbors, the cells above, below, left, right, and the 4 diagonals of a cell. Cells on the edge only have 5 neighbors. The 4 corner cells only have 3 neighbors.
- 8 neighbors wrap- All cells have 8 neighbors. The edge and corner cells neighbors wrap around to the other edges.

Consider this example of neighbors for the 8-neighbor wrap model with cells numbered for clarity.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Consider cell number 1 in the upper left corner. With wrapping its 8 neighboring cells are:

16	13	14
4		2
8	5	6

Write a program that prints the number of neighborhoods in a given data set given the minimum size of a neighborhood and the 4 ways of defining neighboring cells. A given house can belong to at most one neighborhood per data set.

Input:

- The first line will contain a single integer N that indicates the number of data sets.
- The first line in a data set will be a single integer M that indicates the minimum number of connected houses (size) required that need to be connected to form a neighborhood. $M \geq 1$.

- The first line of a data set will be two integers R and C separated by a single space. R indicates the number of rows in the data set's matrix and C indicates the number of columns per row in the data set. R and C will both be ≥ 3 .
- The next R lines will be the rows of the matrix in order. Each line will contain C upper case letters separated by single spaces. All letters will be between A and Z inclusive. Each letter represents the style of the house at that location in the matrix.

Output:

For each data set print out to the screen a labeled line with 4 integers separated by single spaces. The integers are the number of neighborhoods in the data set given the minimum required size based on the 4 rules for neighbors: 4 normal, 4 wrap, 8 normal, and 8 wrap in that order. For C++ use the string class. Let the user input the file name from the keyboard. Refer to the sample output below.

Sample File:

```
3
3
4 3
A B C
D E F
G H I
J K L
3
5 7
A B A B A B A
A B A B A B A
A B C B A C A
A B C B A C A
A B A B A B A
5
3 7
A B B C B B A
B C C B C C B
A B B C B A A
```

Sample Run:

```
Enter file name: neighbor.txt

Set 1: 0 0 0 0
Set 2: 5 6 5 6
Set 3: 0 1 2 3
```

Name the program: FindNeighborsXX.java or FindNeighborsXX.cpp, where XX are your initials.

2. Write a parser for the Data Query Language specification for a company's database software. The program will also test your parser by reading in a sample database, running some queries, and printing the results of those queries to the screen. A database is just a table, where each column has a name, and each row is a database entry consisting of values for each of the columns. The language itself is straight forward, where queries are of the form:

```
SELECT <ITEM> WHERE <NAME><OP><VALUE> [SORT <STYPE> [INT] <COL> [SORT ...]]
```

and:

- SELECT always begins a query

- **<ITEM>** denotes which columns' data entries to return in the query, and can be:
 - an asterisk (*) which means to report the data entries from all columns, or
 - a comma separated list of column names (with a space after each comma) from which you are to report the data entries.
 - Note: If there is no comma, you will report only the data entries from the **<ITEM>** column meeting the query criteria.
- **WHERE** starts an optional segment of the query consisting of **<NAME>**, **<OP>** and **<VALUE>** which are used to run a logical check against the data entry to see if it should be included in the results:
 - **<NAME>** is the name of one of the columns.
 - **<OP>** is a logical operator, either =, <, >, <= or >=.
 - **<VALUE>** is the value to compare against. A column's data should be treated as a string if the = operator is used, and an integer otherwise.
- **SORT** is an optional segment of the query (denoted by [...]) consisting of **<STYPE>**, **<INT>** which is optional, **<COL>**, and another optional embedded SORT segment, which denotes how to re-sort the results:
 - Note: if SORT is missing, leave the data entries in the order found in the data file, otherwise:
 - **<STYPE>** is either DESC for a descending sort, or ASC for an ascending sort.
 - **<INT>** is optional, and if it is present the comparison should be done as integers or done as strings if INT is not specified.
 - **<COL>** is the name of the column to sort on.
 - If there is a second optional SORT clause:
 - take the results of the first sort and, while preserving the groupings of **<COL>**, re-sort those groups by a new **<COL>** using a new **<STYPE>** and possibly by INT, if so denoted in the second SORT clause.
 - if the second SORT is missing, leave the data entries in the order resulting from the first sort.

A query will return:

- if **<ITEM>** is an asterisk (*):
 - it will return all the data entries in the rows that match the WHERE segment, or
 - all the data entries in all rows if the WHERE segment is missing.
- otherwise:
 - it will return the data entries of the columns from the **<ITEM>** part of the query if there is matching data from the WHERE segment, or
 - **<EMPTY>** if there is no matching data.

Input

- The input file contains first the database data and then the queries.
- Database data:
 - The first line of the input file will contain two integers c r separated by a space, where the c is the number of columns and r is the number of database entries.

- The second line will contain a list of column names for each column *c* of the database, where the names are separated by the pipe character "|" as shown below.
- The next *r* lines will each contain a pipe delimited list of values for each column *c* of the database. A value for the last column is optional, as shown below.
- Queries:
 - Following the database data, there will be a line containing a single integer *n* that indicates the number of queries to follow.
 - Each of the following *n* lines will contain a single query in the format described above.

Output

For each query you will print the string Query #*n* on a line by itself, where *n* is the number of the query, starting at 1. Then you will print the data from the query, one data entry per row. Each item from a data entry should be separated by a space. If the query returns no data, then you should print the string <EMPTY> on a line by itself. There should be a blank line after each set of query output. Use any appropriate data structure. For C++ use the `string` class. Let the user input the file name from the keyboard. Refer to the sample output below.

Sample File

```
8 13
Name|ID#|Grade|ELA|Math|Science|SS|Notes
ROBERTS, MATIANA|212090|11|P|N|N|P|jr grad
BARROWS, GEORGE|102559|11|P|N|N|P|jr grad
BROOKS, ROGER|102661|12|N|N|N|P|2011
BROOKS, IAN|104132|11|P|N|P|P|jr grad
BUCHANAN, ALEXANDER|102453|12|N|N|N|P|jr grad
DELGADO, ABIGAIL|540250|10|P|P|N|N|2011
DODSON, NICOLE|240289|11|N|X|X|N|jr grad
ROGERS, ROBERT|742034|10|P|N|N|P|
JONES, RICHARD|770945|11|P|P|N|P|
ALLEN, ALAN|770814|12|P|N|N|P|2012
ROBERTS, ROBERT|770460|11|X|X|N|X|2012
JONES, ROGER|770152|11|P|N|N|P|
ROGERS, MARTIN|587716|10|P|N|P|P|jr grad
2
SELECT Name, ID# WHERE SS=P SORT DESC INT Grade
SELECT * SORT ASC SS SORT DESC Name
```

Sample Run:

Enter the file name: query.txt

```
Query #1
BROOKS, ROGER 102661
BUCHANAN, ALEXANDER 102453
ALLEN, ALAN 770814
ROBERTS, MATIANA 212090
BARROWS, GEORGE 102559
BROOKS, IAN 104132
JONES, RICHARD 770945
```

JONES, ROGER 770152
 ROGERS, ROBERT 742034
 ROGERS, MARTIN 587716

Query #2

DODSON, NICOLE 240289 11 N X X N jr grad
 DELGADO, ABIGAIL 540250 10 P P N N 2011
 ROGERS, ROBERT 742034 10 P N N P
 ROGERS, MARTIN 587716 10 P N P P jr grad
 ROBERTS, MATIANA 212090 11 P N N P jr grad
 JONES, ROGER 770152 11 P N N P
 JONES, RICHARD 770945 11 P P N P
 BUCHANAN, ALEXANDER 102453 12 N N N P jr grad
 BROOKS, ROGER 102661 12 N N N P 2011
 BROOKS, IAN 104132 11 P N P P jr grad
 BARROWS, GEORGE 102559 11 P N N P jr grad
 ALLEN, ALAN 770814 12 P N N P 2012
 ROBERTS, ROBERT 770460 11 X X N X 2012

Name the program: DataQueryXX.java or DataQueryXX.cpp, where XX are your initials.

3. Write a virtual computer that runs programs written in a simple programming language. Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be no blank lines separating data sets. A single data set has up to 1000 lines, each of the format "<Line Number> <Instruction>" where:

- Line Number: ($1 \leq \text{Line Number} \leq 1000$) is an integer indicating the line number for the following Instruction. Within a given data set, Line Numbers are unique, and Line Numbers always increase from one line to the next.
- Instruction: One of the following:
 - "LOAD X value" – Set variable X's value to value
 - "ADD X value" – Increase variable X's value by value
 - "PRINT X" – Display the value of variable X to standard output followed by a newline.
 - "IF X == value GOTO line" – If the value of variable X equals value continue execution at the instruction with Line Number equal to line, otherwise execution moves to the next instruction as usual.
 - "END" – The last instruction in every data set (which will only appear once).

Note:

- X is a non-empty character string of up to 10 characters representing a variable name
- value is an integer ($0 \leq \text{value} \leq 1000$)
- line is a valid Line Number from the current data set
- Assume that every variable has an initial value of 0 that is reset at the beginning of each data set.

For each data set, there will be at least one line of output to the screen. The first line of output for each data set will read, "START N" where N is an integer identifying which data set is being processed. N will be 1 for the first data set and increment by one for each additional data set. Also, output will be the

results from all the PRINT X statements executed by the virtual computer. Use any appropriate data structure. For C++ use the string class. Let the user input the file name from the keyboard. Refer to the sample output below.

Sample File:

```

1 LOAD var 1
10 PRINT var
11 END
5 END
19 PRINT noinit
20 LOAD noinit 10
21 PRINT noinit
30 LOAD x 5
35 LOAD y 1
40 ADD x 5
45 ADD y 1
50 IF y == 5 GOTO 70
60 IF z == 0 GOTO 40
70 PRINT x
80 END

```

Sample Run:

```

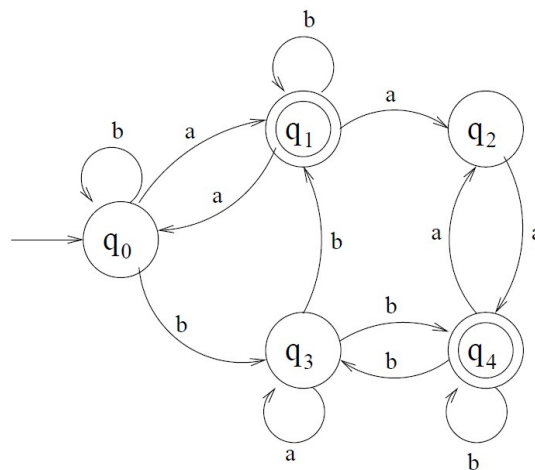
Enter file name: virtual.txt

START 1
1
START 2
START 3
0
10
25

```

Name the program: VirtualComputerXX.java or VirtualComputerXX.cpp, where XX are your initials.

4. Depicted in the figure below is a nondeterministic finite automaton (NFA) that we will refer to as M. Each circle represents a state and each arrow labeled by a symbol represents a transition from one state to another (or itself) associated with that symbol. The unlabeled arrow points to the initial state. Double circles correspond to final states. By convention, we name the states in an NFA $q_0, q_1, \dots, q_{m-1}$, where m is the number of states and q_0 is the initial state. The set of symbols appearing as labels on transitions is called the alphabet of the NFA.



Consider the string baab. In M there are three paths beginning at the initial state that "spell out" that string:

$$\begin{array}{l}
 q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \\
 q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0 \xrightarrow{b} q_3 \\
 q_0 \xrightarrow{b} q_3 \xrightarrow{a} q_3 \xrightarrow{a} q_3 \xrightarrow{b} q_4
 \end{array}$$

Since at least one of these paths (namely, the last one) ends in a final state, M is said to accept baab. If, on the other hand, all paths spelling out a string end in non-final states, that string is said to be rejected by M. An example is the string baa, as the three paths that spell it out end in the non-final states q_0 , q_2 , and q_3 , respectively. Notice that it is possible for there to be no paths that spell out a string, in which case it is certainly rejected. An example of this is aab (or any string of which aab is a prefix). (The only path spelling out aa ends in q_2 , but there is no outgoing transition from there labeled b; hence, even if q_2 were a final state, any string having aab as a prefix is rejected.)

Write a program that, given an NFA with alphabet $\{a, b\}$ and some input strings composed of a's and b's, reports, for each input string, whether it is accepted by the NFA. Input will be from a text file. The first line contains the number $m > 0$ of states in the NFA to be tested, followed by the number $k > 0$ of final states, followed by the number $t > 0$ of transitions. The second line contains k integers in the range $0..m-1$ identifying the NFA's final states. (The initial state is not explicitly identified because it is to be understood that it is state 0.) Each of the following t lines describes a single transition, which is given by two state identifiers (integers in the range $0..m-1$) p and q separated by the label (either a or b) on a transition that goes from state p to state q . The transitions need not be listed in any particular order. (The sample input below describes M.) On the following line is a positive integer r indicating how many input strings are to be processed. Each of the following r lines contains one such string. For each string given as input, generate one line of output that classifies that string as being either accepted or rejected by the NFA. Let the user input the file name from the keyboard. Must use the Java/C++ string classes for string input. A graph data structure must be used. Refer to the sample output below.

Sample File:

```

5 2 13
1 4
0 b 0
0 b 3
4 b 4
3 a 3
0 a 1
3 b 1
1 a 0
1 b 1
3 b 4
1 a 2
2 a 4

```

Sample Run:

```

Enter file name: nfa.txt

baab is accepted
aabab is accepted
bbbb is accepted
bbbababaa is accepted
aaaaaa is rejected
abbbaabaa is rejected

```

```

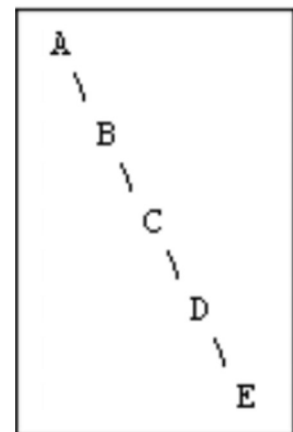
4 a 2
4 b 3
6
baab
aabab
bbbb
bbbababaa
aaaaaa
abbbaabaa

```

Name the program: NFAXX.java or NFAXX.cpp, where XX are your initials.

Required Problem- Comprehensive.

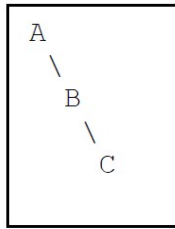
5. The binary search tree is great for storing information that allows efficient searching and/or retrieval of the data. It is designed as a collection of nodes, each with a single left node/child and a single right node/child, where at each node, the left child is always less than or equal to the parent value and the right node is always greater than the parent value. To find any item in a well-balanced tree has a best case of $O(\log_2 N)$ retrieval. However, binary search trees have a worst-case retrieval scenario of $O(N)$. This can happen if you have a binary search tree of the form shown here. To find the E in the search tree, we must visit all 5 nodes, not very efficient. Fortunately, it is possible to guarantee that binary search trees never end up in this unfortunate form, so that there is a better chance of having this quick $O(\log_2 N)$ retrieval from all binary search trees. This process is called search tree balancing. One of these techniques is to use rotations. Rotations take a specific child node, turn it into a parent, and rotate all the dependencies necessary around that node to keep the binary search tree property (as described above). There are 4 different kind of rotations that can work.



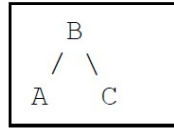
The SINGLE LEFT ROTATION

A tree is right heavy if its right subtree is larger than the left subtree in height by two or more AND the height of its right subtree's right subtree is greater than its right subtree's left subtree. In the example below, the height of A's right subtree is 2, and is zero for the left subtree. Furthermore, the height of B's right subtree is 1, and its left subtree zero. This fits the criteria, and she can perform a single left rotation about A to make B the new root, which looks like this:

From this:

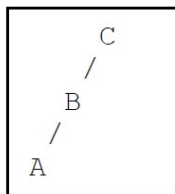


To This:

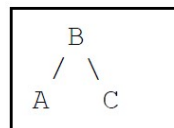
**The SINGLE RIGHT ROTATION**

If the tree is left heavy - its left subtree is larger than the right subtree in height by two or more, AND the height of its left subtree's left subtree is greater than its left subtree's right subtree - then she can perform a single right rotation about C to make B the new root, which looks like this:

From this:

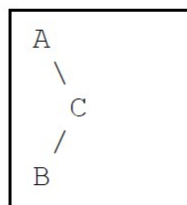


To This:

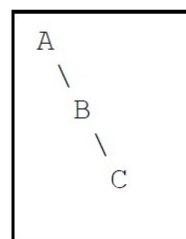
**The DOUBLE LEFT ROTATION**

If Luna has a situation where a tree is right heavy - its right subtree is larger than the left subtree in height by two or more -- AND the height of its right subtree's left subtree is greater than its right subtree's right subtree, then we must perform a double left rotation, which would look like this:

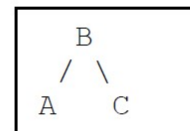
From this:



To This:



To This:

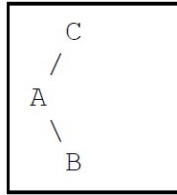


This is a single right-handed rotation about C to move B to the right child of A and then a standard single rotation about A to make B the new root.

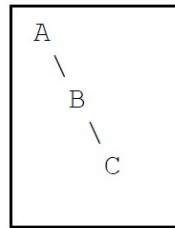
The DOUBLE RIGHT ROTATION

If we are in the situation that a node is right heavy -- its right subtree is larger than the left subtree in height by two or more -- AND the height of its right subtree's left subtree is greater than its right subtree's right subtree, then we must perform a double right rotation, which would look like this:

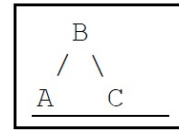
From this:



To This:



To This:



This is a single left-handed rotation about A to move B to the left child of C, and then a single left rotation about C to make B the new root.

Armed with these rebalancing techniques, write a program to detect when these rotations are necessary about the root node of a binary search tree. Given a string of words to put in binary search tree (using alphabetical order to determine where they should go in the tree), display when the input of a node makes the tree unbalanced at the root node. Input the nodes into the binary search tree in the order they are given to determine when the tree becomes unbalanced. When the tree becomes unbalanced, print which word it was that caused the imbalance, and which rotation should be used to correct it. If, in the order given, the words can all be placed into the binary search tree and remain completely balanced, then print that the tree is balanced.

The first integer from a text file will represent the number of data sets to follow. Each line will be a list of words to be placed into the binary search tree. On output, each labeled data set should return to the screen the line "UNBALANCED AFTER <unbalancing input>, A<rotation type> AT ROOT TO REBALANCE." If all the input is consumed and the root of the tree remains balanced, then output the phrase "TREE IS BALANCED." Let the user input the file name from the keyboard. Use any appropriate data structure of your choice. Refer to the sample output below.

Sample File

```

6
A B C
C B A
B A C
C A B
ALLEGORY BREAD CHARMANDER DANDELION ELEPHANT
MAYFLOWER HAIR RETINA ECLIPSE IGUANA QUACK RODEO OCTOBER SOUP
  
```

Sample Run:

Enter the file name: rotate.txt

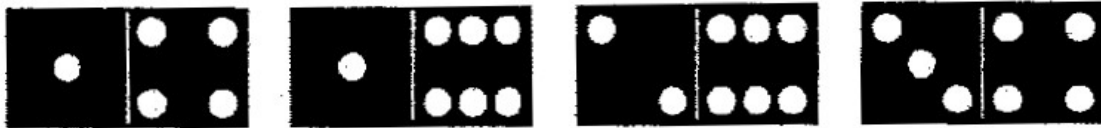
```

Tree 1: UNBALANCED AFTER C, NEED SINGLE LEFT ROTATION AT ROOT TO REBALANCE.
Tree 2: UNBALANCED AFTER A, NEED SINGLE RIGHT ROTATION AT ROOT TO REBALANCE.
Tree 3: TREE IS BALANCED.
Tree 4: UNBALANCED AFTER B, NEED DOUBLE RIGHT ROTATION AT ROOT TO REBALANCE.
Tree 5: UNBALANCED AFTER CHARMANDER, NEED SINGLE LEFT ROTATION AT ROOT TO REBALANCE.
Tree 6: TREE IS BALANCED.
  
```

Name the program: TreeRotateXX.java or TreeRotateXX.cpp, where XX are your initials.

Extra Credit: Choose one of the two problems below for extra credit. See 2436 Grading Guide Sheet for additional grading/submission information. Partial credit will be given. No additional extra credit for implementing both programs.

1. A box of dominoes contains 21 dominoes, each of which has two "ends". There are six "ones" which means one end is a one and the other end is a number one through six. There are five "twos" (after eliminating the duplicate 2-1 vs. 1-2). Similarly, there are four "threes", three "fours", two "fives" and one "six" for a total of 21 dominoes. Write a program given seven randomly selected dominoes from a box, tries to form a line by putting matching ends together. For example, one has the dominoes 1-4, 1-6, 2-6 and 3-4:



One could form the line: 2-6 6-1 1-4 4-3



Remember dominoes can be rotated by 180 degrees so the numbers are reversed. In the chain above the 1-6 and 3-4 dominoes have been "turned around" to fit into the chain. Input from a data file where the first line of input will contain a single integer n that indicates the number of games played. Each of the following n lines will contain seven dominoes, each in the form $x-y$ where x and y are the values for each end of one domino. Each of dominoes will be separated by a space. For each game, labeled, output to the screen: YES if all seven dominoes can be placed end-to-end as described above and the actual line, or NO if it is not possible. Use any appropriate data structure. Let the user input the file name from the keyboard. Refer to the sample output below.

Sample File:

```
2
1-2 4-3 4-4 2-5 4-2 1-4 3-5
4-5 6-6 1-6 5-6 3-4 2-1 4-1
```

Sample Run:

```
Enter file name: tiles.txt
Game 1: YES- 2-1 1-4 4-4 4-3 3-5 5-2 2-4
Game 2: NO
```

Name the program: CheckDominoesXX.java or CheckDominoesXX.cpp, where XX are your initials.

2. The classic Knapsack problem-scenario: A robber breaks into a house carrying a knapsack of a certain weight (or volume) capacity in which to place stolen items. He can only carry a certain weight capacity but wants to maximize the value he steals. There are many items to choose from, but he wants to optimize his haul. For example, his capacity might be 5 pounds. Let's also say he encounters only three items: an object that weighs 3 pounds which has a value of \$5, a second object that weighs 2 pounds at \$3 value, and a third object of weight 1 and value \$4.

Using brute force is straight forward, but with large data sets might take a while. Here is a simple description of the brute force technique. Clearly the knapsack could take each individual item by itself with room to spare but could not take all three too much weight. So, you try two at a time. Items 1 and 2 combine for a weight of 5, with a combined value of \$8. Items 2 and 3 weigh 3 pounds total with a value of \$7. Items 1 and 3 weigh only 4 pounds, but have the best value of \$9, so there is your best combination. Write a program that solves the above Knapsack problem. The first line of the input file will contain a single integer *n* that indicates the number of data sets that follow. Each data set will start with a single integer *k* denoting the total weight capacity of the knapsack, and then another integer *i* that shows how many items the robber needs to consider. Below that are listed the value and weight of each of the *i* items. There will be no more than 100 items to choose from, and the knapsack capacity will be no more than 100 pounds.

For each data set, output a set label, the total capacity of the knapsack, the total weight of the items taken, the total value taken, and an ordered list of the items taken, as found in the data file, with the first item being #1, the next #2, etc., in the exact form seen below, with single space separation as shown and one blank line after each complete output set. Let the user enter the file name from the keyboard. Refer to the sample output below.

Sample Input File:

```
2
5
3
5 3
3 2
4 1
6
5
1 5
6 4
4 3
7 2
3 1
```

Sample Run:

```
Enter the file name: knap.txt

Set 1:

Knapsack capacity = 5
Total weight = 4
Total value = 9
Item #1 - v = 5: w = 3
Item #3 - v = 4: w = 1

Set 2:

Knapsack capacity = 6
Total weight = 6
Total value = 14
Item #3 - v = 4: w = 3
Item #4 - v = 7: w = 2
Item #5 - v = 3: w = 1
```

Name the program: KnapsackXX.java or KnapsackXX.cpp, where XX are your initials.