IDC Herzliya

Efi Arazi School of Computer Science

Introduction to Computer Science

Final Examination + Solution

- The exam lasts 3 hours. There will be no time extension.
- Use your time efficiently. If you get stuck somewhere, leave the question and move on to another question.
- Use of digital devices, books, lecture notes, and anything other than the exam pages is forbidden. All the materials that you need for the exam are supplied with the exam.
- Answer all questions on the supplied exam pages. Erase clearly what you don't want us to grade.
- You can answer any question in either English or Hebrew.
- If you feel a need to make an assumption, you may do so as long as the assumption is reasonable and clearly stated.
- If you can't give a complete answer, you may give a partial answer. A partial answer will award partial points.
- If you are asked to write code and you feel that you can't write it, you may describe what you wish to do in English or Hebrew. A good explanation will award partial credit.
- If you are asked to write code that operates on some input, there is no need to validate the input unless you are asked to do so. Likewise, if you are asked to write a method that operates on some arguments, there is no need to validate the arguments unless you are asked to do so.
- There is no need to document the code that you write, unless you want to communicate something to us.
- Your code will be judged, among other things, on its conciseness (תמציתיות) and elegance. Unnecessarily long or cumbersome (מסורבל) code will cause loss of points, even if it provides the correct answer.
- No points will be taken for trivial syntax errors.
- Your hand-writing must be clear and easy to read.
  Unreadable answers will get a 0 grade.
- If you want, you can separate the help pages for easier reference.

Good Luck!

Questions 1-13 (78 out of 100 points) are about *compressed arrays*, described in help pages 1-3. You must spend ten minutes reading these help pages *now*, before you read the questions below.

Questions 1-8 focus on implementing some of the `CompArray` methods. When you implement these methods, assume that all the other methods in the class have been implemented correctly, and you can use these methods as you please. You are welcome to use these methods even if the resulting code will be less efficient. In other words, you should focus more on elegance, and less on efficiency.

Remember that in the implementations that you have to write, the elements in the lists that represent the arrays are not ordered, and don't have to be ordered.

Questions 9-13 focus on how to make some of the `CompArray` methods more efficient.

1. (5 points) The `toString` method returns a string consisting of all the (*index*, *value*) pairs representing the non-zero elements of the array, with a space after each pair, in the same order in which they appear in the list that represents the array. Each pair is enclosed in parentheses. Implement the method (there is no need to use `StringBuilder`):

```
/** Returns a textual representation of this compressed array. */
public String toString() {
   String ans = "";
   Node current = this.first.next;  // skips the first dummy node
   while (current != null) {
      ans += current.toString();
      current = current.next;
   }
   return ans;

}
```

2. (5 points) The `get` method returns the value of the array at index i. Read the method's documentation and implement it.

```
/** Returns the value at index i of the array.
 *  For example, if the list is (2,3.0) (5,2.1) (7,-3.3),
 *  get(5) returns 2.1 and get(4) returns 0.
 *  Assumes that i is between 0 and the array length (no need to check). */
 public double get(int i) {
    Node current = this.first.next;  // skips the first dummy node
    while (current != null) {
       if (current.index == i) {
          return current.val;
       }
       current = current.next;
    }
    return 0;
}
```

3. (7 points) The following constructor creates a compressed array that represents the given array. Implement the constructor.

```
/** Constructs a compressed array from the given array. */
public CompArray(double[] arr) {
   first = new Node(0,0);  // first node is dummy
   length = arr.length;
   size = 0;
   for (int i = 0; i < arr.length; i++) {
      if (arr[i] != 0) {
         Node newNode = new Node(i, arr[i]);
         newNode.next = first.next;
         first.next = newNode;
         size++;
      }
   }
}
```

Another possible and acceptable solution is to use the `set` method within the loop.

4. (10 points) The `set(int i, double v)` method sets the value at index `i` to be `v`, if `v` is non-zero. If `v` is zero, and before calling this method the value at the given index was not zero, then the element is removed from the list. Read the method's documentation and implement it.

```
/** Sets the value at index i to v. Assumes that i is a valid value (no need to check).
 *  Note that the list that represents the compressed array should contain only
 *  non-zero elements. Therefore, if v is zero, the element must be removed from
 *  the list. For example, if the compressed array is (2,3.0) (5,2.1) (7,-3.3)
 *  and we do set(5,0), the list becomes (2,3.0) (7,-3.3). */
public void set(int i, double v) {
   Node prev = this.first;
   Node current = this.first.next; // skips the first dummy node
   while ((current != null) && (current.index != i)) {
      current = current.next;
      prev = prev.next;
   }

   // If v is zero, removes the element from the list.
   if ((v == 0) && (current != null)) {
      prev.next = prev.next.next;
      size--;
   }

   // If v is not zero, sets the value of the element to v.
   if ((v != 0) && (current != null)) {
      current.val = v;
   }

   // If v is not zero and we are at the end of the list,
   // Adds the non-zero element to the end of the list.
   if ((v != 0) && (current == null)) {
      Node newNode = new Node(i, v);
      prev.next = newNode;
      size++;
   }
}
```

Another possible and acceptable solution is to write an implementation that maintains the order of elements in the list.

5. (5 points) The `toArray()` method creates a regular array from this compressed array.
Read the method's documentation and implement it. Remember: when you create a regular array in Java, the compiler initializes all the array elements to zero.

```
/** Returns an array containing all the elements of this compressed array, as well as the zero elements. **/
public double[] toArray() {
   double[] ans = new double[this.length];
   Node current = this.first.next;  // skips the first dummy node
   while (current != null) {
      ans[current.index] = current.val;
      current = current.next;
   }
   return ans;
}
```

Note: when an array is created, Java initializes all the array entries to zeros. If the solution includes code that initialize the array (unnecessarily), that's fine.

6. (8 points) The following method adds the values of the given compressed array to the values of this compressed array, provided that the two arrays have the same length. Read the method's documentation and implement it.

```
/** Adds the values of the other compressed array to the values of this compressed array.
 *  If the array lengths are not equal, throws a run-time exception.
 *  Note 1: the addition operation changes the list that represents this compressed array.
 *  Note 2: the resulting list must contain only non-zero elements, as usual. **/
public void add(CompArray other) {
   if (this.length != other.length) {
      throw new RuntimeException("the array lengths are not compatible");
   }

   // Adds the corresponding values into this compressed array
   Node current = other.first.next;  // skips the dummy node
   while (current != null) {
   int index = current.index;
      set(index, current.val + this.get(index));
      current = current.next;
   }
}
```

Another possible and acceptable solution: if the implementation of the `set` method in question 4 maintains order, it is possible to write a more efficient solution that goes through the two sorted sets without using `get`.

Answer question 7, or question 8. Answer only one question.

7. (8 points) We define two arrays to be equal if they have the same length and the same values in every index. Read the method's documentation and implement it.

```
/** Checks if the other compressed array is equal to this compressed array.
 *  Assumes that the two arrays have the same length (no need to check). */
 public boolean equals(CompArray other) {
    for (int i = 0; i < this.length; i++) {
       if (this.get(i) != other.get(i)) return false;
    }
    return true;
}
```

Another possible and acceptable solution: if the implementation of the `set` method in question 4 maintains order, it is possible to write a more efficient solution that goes through the two sorted sets without using `get`.

8. (8 points) The inner product of two arrays (that have the same length) is the sum of the products of array elements at corresponding indices. For example, the inner product of 2 0 3 and 4 2 5 is 2*4+0*2+3*5 = 23. Read the method's documentation and implement it.

```
/** Returns the inner product of this compressed array and the other one.
 *  Assumes that the two arrays have the same length (no need to check). */
public double innerProduct(CompArray other){
   double ans = 0;
   // Computes the inner product
   Node current = this.first.next;  // skips the dummy node
   while (current != null) {
      ans += current.val * other.get(current.index);
      current = current.next;
   }
   return ans;
}
```

Another possible and acceptable solution: if the implementation of the `set` method in question 4 maintains order, it is possible to write a more efficient solution that goes through the two sorted sets without using `get`.

9. (5 points) What is the running-time of your `add` method implementation? Your answer must include the terminology "$O(...)$" and "N", and a definition of what N stands for. Explain *why* the running time is what you think it is.

Answer: In the `add` implementation shown here, we go through every element of this list, and, for each element, we call the `get` method on the other list. Since both operations are $O(N)$, the overall running-time is $O(N^2)$, where N is the length of the arrays.

Another possible and acceptable solution: if the implementation of the `set` method in question 4 maintains order, the answer is $O(N)$.

10. (5 points) In the current specification and implementation of `add`, the method adds the other array into the current array, changing the current array. Suppose that we also want an `addNew` method that does a similar array addition operation, *without* changing the current array. Here is an example of how this method can be used by some client code: `CompArray cArr3 = cArr1.addNew(cArr2).`

a. Write the API documentation of `addNew` (2-3 lines of text), and the method signature (one line).

Answer:

```
/** Returns a compressed array which is the sum of this compressed array
 *  and the other compressed array. **/
 public CompArray addNew(CompArray other) {
```

b. Describe how you will implement the `addNew` method. There is no need to write code. Instead, describe in writing the main steps of the implementation.

Answer: the method should start by creating a new empty `CompArray` object, using the constructor. Let's call this object `sum`. Next, the method will use a loop to add all the elements of the `this object` into `sum`. Next, the method will use a loop to add all the elements of the `other` object into `sum`. Finally, the method will return `sum`.

<u>We now assume that we have to build another implementation of the `CompArray` class. In this implementation, the elements in the lists that represent arrays will always be sorted, in increasing order, according to the value of `index`.</u>

11. (8 points) Write a new implementation of the `set` method, so that the list elements will always be ordered as described above.

/** Sets the value at index i to v. Assumes that i is a valid value (no need to check).
 * Note that the list that represents the compressed array <u>should be ordered</u>, and should
 * contain only non-zero elements. Therefore, if v is zero, the element must be removed from
 * the list. For example, if the compressed array is (2,3.0) (5,2.1) (7,-3.3)
 * and we do set(5,0), the list becomes (2,3.0) (7,-3.3). */

```
public void set(int i, double v) {
      Node prev = this.first;
      Node current = this.first.next; // skips the first dummy node
      while ((current != null) && (current.index <= i)) {
            current = current.next;
            prev = prev.next;
      }

      // If v is not zero and we are at the end of the list,
      // Adds the non-zero element to the end of the list.
      if ((v != 0) && ((current == null) || current.index >  i)) {
            Node newNode = new Node(i, v);
            prev.next = newNode;
            newNode.next = current;
            size++;
            return;
      }

      // If v is zero, removes the element from the list.
      if (v == 0 && current.index == i) {
            prev.next = prev.next.next;
            size--;
      }

      // If v is not zero, sets the value of the element to v.
      if ((v != 0) && (current.index == i)) {
            current.val = v;
      }
}
```

Another possible and acceptable answer: if the student already wrote a sorted implementation of the `set` method in question 4, then s/he should just say so and get full credit for this question. The student should get full credit here even if the answer to question 4 is incorrect, since we already deducted points for this in question 4.

12. (6 points) This question is about the `CompArray(double[] arr)` constructor.

(a) Given that the list elements must be ordered as described above, describe in words how the implementation of the constructor should change.

Answer: there is no need to change the implementation. Array elements are always sorted according to the index values. Therefore, this constructor always creates a sorted list.

(b) what will be the running-time of the constructor implementation? Your answer must include the terminology "*O*(...)" and "*N*", and a definition of what *N* stands for. Explain *why* the running time will be what you think it will be.

Answer: the running-time will be *O*(*N*), where *N* is the length of the array. Explanation: the constructor uses a loop that has *N* iterations, one for each array element.

13. (3 points) Suppose that we make all the necessary changes in the `CompArray` class, so that the lists that represent arrays will always be ordered as described above. Will we have to make any changes in the API and signatures of the class methods? If so, which changes?

Answer: since all the changes are strictly in the implementation, the API of the new version should remain the same as the original one. Another possible and acceptable addition to this answer: the API can be extended to mention the efficiency of the implementation.

<u>Questions 14-15 are about the Vic program described in help pages 4-5.</u>

14. (8 points) What will be the values of the following memory locations <u>after</u> the `Mystery` program ends its execution on the input shown in help page 5?

Answer:
```
84:  12
85:   0
86:   0
87:   0
88: 333
89:  24
90: 899
91:  29
92:   4
93:
94: 433
95:  55
96:  12
97:
```

Another possible and acceptable answer: the program puts zeros in addresses 85, 86, and 87, and leaves all the other values of the shown addresses unchanged.

Partial correct answer: if the student ignored the exam instructions and based his/her answer on the code shown in the screen shot, where only commands 00-15 are shown, then the answer should be "all the shown addresses remain the same, except for address 85, which becomes zero". This answer awards 6 points (out of 8).

15. (5 points) What is the *general operation* that the `Mystery` program is designed to perform?

Answer: the program reads three inputs. The first input is required in order to implement the program's operation. If we call the second input *base* and the third input *n*, the program sets the *n* memory addresses starting from address *base* to 0. Any other answer that says the same thing in a different language is acceptable.

Partial correct answer: if the student gave a partial correct answer to question 14, and if s/he says "the program sets the address whose value is the second input to zero", the answer awards 3 points (out of 5).

Question 16 is about the `Palindrome` program described in help page 6.

16. (12 points) Write a recursive implementation of the palindrome method. If you want, you can write a non-recursive implementation for a maximum of 6 points instead of 12. You must write one implementation only.

Note: the method `Math.random()` returns a `double` value greater than or equal to 0, and less than 1.

Answer:

```
public class Palindrome {

    public static void main(String[] args) {
        palindrome(Integer.parseInt(args[0]));
    }

    /** Prints a random, odd numeric palindrome of the given degree (n). */
   public static void palindrome(int n) {
      int digit = (int) (10*Math.random());
      System.out.print(digit);
      if (n == 1) return;
      else palindrome(n-1);
      System.out.print(digit);
   }
}
```

Compressed arrays (help page 1)

In many applications we have to use arrays that contain many zero values. In order to save memory, we'll develop a `CompressedArray` class that represents arrays using linked lists that contain only the non-zero elements of the arrays. The list elements are (*index*, *value*) pairs, where *index* is the index in the array and *value* is the non-zero value at this index.

For example, the array `0.7`, `0.3`, `0`, `0`, `0`, `2.5`, `0`, `0`, `0`, `0` is represented using the list `(0,0.7) (1,0.3) (5,2.5)`. Each such list object also has a `length` and a `size` that represent the array's length and the list's size, respectively. In this example we have `length` = 10 and `size` = 3.

Note that in the implementation that you have to write, the lists are not necessarily ordered.
For example, a list like `(4,2.5) (0,0.7) (1,0.3)` can also represent the array shown above. The list elements can appear in any order, since the list contains all the information that we need in order to reconstruct the original array from the compressed representation. Here are a few more examples:

- `0`, `0`, `5.1`, `0`, `0`, `0`, `0.8` can be represented by the list `(2,5.1) (6,0.8)`; `length` = 7 and `size` = 2.
- `3.1`, `7.12` can be represented by the list `(0,3.1) (1,7.12)`; `length` = 2 and `size` = 2.
- An array that has 500,000 elements of which location 17 contains 9, location 354,722 contains 7, and all the other locations contain 0 is represented by the list `(354722,7.0) (17,9.0)`; `length` = 500000 and `size` = 2. Once again, note that the elements' order is insignificant.

Here is an example of client code that uses `CompressedArray` objects. Read it carefully:

```
package comparrays;

public class CompArrayTest {
   public static void main(String args[]) {
      // Constructs a compressed array of length 10 containing zeros.
      CompArray cArr1 = new CompArray(10);
      cArr1.set(1,1.8); cArr1.set(4,1.3); cArr1.set(6,2.0); // sets some elements to values
      System.out.println(cArr1); // prints (1,1.8) (4,1.3) (6,2.0)

      // Gets the first 6 values of the array:
      System.out.println(cArr1.get(0));    // prints 0
      System.out.println(cArr1.get(1));    // prints 1.8
      System.out.println(cArr1.get(2));    // prints 0
      System.out.println(cArr1.get(3));    // prints 0
      System.out.println(cArr1.get(4));    // prints 1.3
      System.out.println(cArr1.get(5));    // prints 0

      // Constructs a compressed array and sets its elements from the values of the given array.
      CompArray cArr2 = new CompArray(new double[] {0.7, -1.8, 0, 0, 0, 1.0, 3.1, 0, 0, 0});
      System.out.println(cArr2); // prints (0,0.7) (1,-1.8) (5,1.0) (6,3.1)

      // Checks if the compressed arrays cArr1 and cArr2 are equal:
      System.out.println(cArr1.equals(cArr2)); // prints false

      // Creates an array from the compressed array cArr1, and prints it.
      double a[] = cArr1.toArray();
      for (double e : a) System.out.print(e + " "); // prints 0 1.8 0 0 1.3 0 2.0 0 0 0 0

      // Prints the arrays (again), and then performs the array operation cArr1 = cArr1 + cArr2.
      // The addition is possible since cArr1 and cArr2 have the same length (10).
      System.out.println(cArr1); // prints (1,1.8) (4,1.3) (6,2.0)
      System.out.println(cArr2); // prints (0,0.7) (1,-1.8) (5,1.0) (6,3.1)
      cArr1.add(cArr2);
      System.out.println(cArr1); // prints (0,0.7) (4,1.3) (5,1.0) (6,5.1)
   }
}
```

Implementation: each `CompArray` object is implemented as a linked list of `Node` objects.
The `Node` class implementation is described in the next page.

The Node Class (help page 2)

```
package comparrays;

/** Represents a node in a linked list.
 *  A node has an index (int), a value (double),
 *  and a pointer to another Node object (which may be null). **/
public class Node {
   int index;   // index in the array        (package-private)
   double val;  // value at this index       (package-private)
   Node next;   // pointer to the next Node  (package-private)

   /** Constructs a node with the given data.
    *  The new node will point to the given node (next). */
   Node(int index, double val, Node next) {
      this.index = index;
      this.val = val;
      this.next = next;
   }

   /** Constructs a node with the given data.
    *  The new node will point to null. */
   Node(int index, double val) {
      this(index, val, null);
   }

   /** Returns a textual representation of this node. */
   public String toString() {
      return "(" + index + "," + val + ")";
   }
}
```

The `CompArray` class skeleton and API (help page 3)

```
package comparrays;

/** A compressed array is an array implementation that records only the non-zero elements of the array,
 *  using a linked list. The compressed array also has a length and a size, which are, respectively,
 *  the number  of elements in the array, and the number of the non-zero elements (which is also the number
 *  of elements in the list, not including the dummy node).  */
public class CompArray {
   private Node first;  // a pointer to the list representing this compressed array
   private int length;  // number of elements in the array
   private int size;    // number of non-zero elements in this compressed array
                        // (which is also the number of nodes in the list).

   /** Constructs a compressed array of the given length. */
   public CompArray(int length) {
      this.first = new Node(0, 0);  // each list starts with a dummy node
      this.length = length;
      this.size = 0;
   }

   /** Constructs a compressed array from the given array. */
   public CompArray(double[] arr) {}

   /** Sets the value at index i of this compressed array to v. Assumes that i is a valid value (no need to check).
    *  Note that the list that represents the compressed array should contain only non-zero elements. Therefore,
    *  if v is zero, the element must be removed from the list. For example, if the compressed array is
    *  (2,3.0) (5,2.1) (7,-3.3) and we do set(5,0), the list becomes (2,3.0) (7,-3.3).  */
   public void set(int i, double v) {}

   /** Returns the length of this array. */
   public int length() { return length; }

   /** Returns the number of non-zero elements in this compressed array. */
   public int size() { return size; }

   /** Returns the value at index i. For example, if the list is (2,3.0) (5,2.1) (7,-3.3),  get(5) returns 2.1
    *  and get(4) returns 0. Assumes that i is a valid value (no need to check). */
   public double get(int i) {}

   /** Adds the values of the other compressed array to the values of this compressed array. If the array
    *  lengths are not equal, throws a run-time exception. Notes: (1) the addition operation can change the list
    *  that represents this compressed array. (2) the resulting list must contain only non-zero elements, as usual. **/
   public void add(CompArray other) {}

   /** Returns the inner product of this compressed array and the other one (explained in question 8).
    * Assumes that the two arrays have the same length (no need to check). */
   public double innerProduct(CompArray other) {}

   /** Returns an array containing all the elements of this compressed array, including the zeros. **/
   public double[] toArray() {}

   /** Checks if the other compressed array is equal to this compressed array.
    * Assumes that the two arrays have the same length (no need to check). */
   public boolean equals(CompArray other) {}

   /** Returns a textual representation of this compressed array. */
   public String toString() {}
}
```

The Mystery Vic program (help page 4)

The program

```
00: 800
01: 413
02: 800
03: 113
04: 413
05: 800
06: 430
07: 398
08: 431
09: 331
10: 230
11: 621
12: 398
13: 0
14: 331
15: 199
16: 431
17: 313
18: 199
19: 413
20: 509
21: 0
```

Vic language documentation:

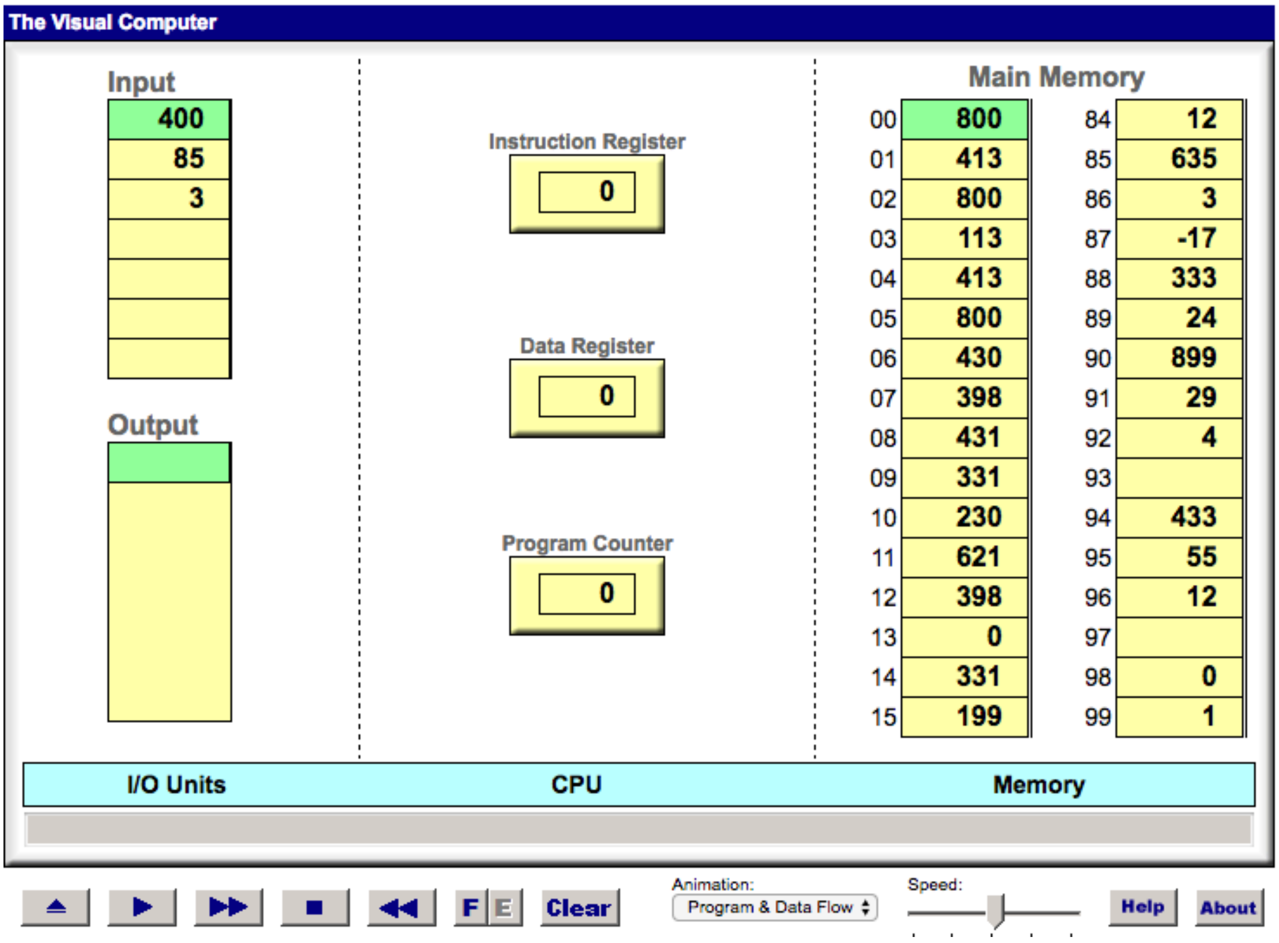
```
800:  read      (D = input)
900:  write     (output = D)
3xx:  load xx   (D = M[xx])
4xx:  store xx  (M[xx] = D)
1xx:  add xx    (D = D + M[xx])
2xx:  sub xx    (D = D - M[xx])
5xx:  goto xx   (goto xx)
6xx:  gotoz xx  (if D==0 goto xx)
7xx:  gotop xx  (if D>0 goto xx)
0  :  stop      (stop the execution)
```

Where:

`D`	is the data register
`M`	is the memory
`xx`	is a two digit number ranging from 00 to 99

Each input, output, register, and memory location can contain a number between -999 and 999.

The Vic computer state just before the Mystery program starts executing: (help page 5)



The Visual Computer

Input

400
85
3

Output

Instruction Register: 0

Data Register: 0

Program Counter: 0

Main Memory

00	800	84	12
01	413	85	635
02	800	86	3
03	113	87	-17
04	413	88	333
05	800	89	24
06	430	90	899
07	398	91	29
08	431	92	4
09	331	93	
10	230	94	433
11	621	95	55
12	398	96	12
13	0	97	
14	331	98	0
15	199	99	1

I/O Units CPU Memory

Animation: Program & Data Flow

Comments:

- The input unit contains a typical input of this program
- Memory locations 00...21 contain the program's code
  (of which we see only 00...15, but if we'll scroll down we'll see the rest of the program)
- The rest of the memory locations (of which we see 84...97) contain arbitrary data.
- Memory locations 98 and 99 contain 0 and 1, as usual in Vic.

The Palindrome program (help page 6)

An *odd palindrome* is a sequence of characters (in this question, digits 0-9) which is symmetric. In other words, it is the same when read left-to-right as it is right-to-left. We define the degree of the palindrome to be the positive integer *n* such that the length of the palindrome is 2*n* - 1. As you can see, the number of digits in the palindrome is always odd.

```
public class Palindrome {

    public static void main(String[] args) {
        palindrome(Integer.parseInt(args[0]));
    }

    /** Prints a random, odd numeric palindrome of the given degree (n). */
    public static void palindrome(int n) {
       // Code missing
    }

}
```

Executing the `Palindrome` program several times (examples)

```
% java Palindrome 2
353

% java Palindrome 3
79797

% java Palindrome 3
66666

% java Palindrome 3
88188

% java Palindrome 3
29392

% java Palindrome 4
7531357

% java Palindrome 4
4449444

% java Palindrome 4
2121212
```