IDC Herzliya

Efi Arazi School of Computer Science

Introduction to Computer Science

Final Exam Sample 3 + 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, or anything other than this exam form is not allowed. All the materials that you need for answering this exam are supplied with the exam.

• Answer all questions on this exam form.

• Your handwriting must be clear, and easy to read. Unclear answers will get a 0 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 in Hebrew. A good explanation will award partial points.

• If you are asked to write code that operates on some input, there is no need to validate the input unless you are explicitly asked to do so. Likewise, if you are asked to write a function that operates on some parameters, there is no need to validate the parameters unless you are explicitly asked to do so.

• There is no need to document the code that you write, unless you want to communicate something to us.

• The code that you will write will be judged, among other things, on its conciseness, elegance, and efficiency. Unnecessarily long or cumbersome code will cause loss of points, even if it provides the correct answer.

• No points will be taken for trivial syntax errors.

Good Luck!

0

General comment: There is typically more than one correct solution to a programming question. Therefore, the actual solution need not be identical to the "school solution" given here. As a rule, programming solutions are judged for their correctness, elegance, and efficiency.

1. (15 points) Read the "Bubble Sort" help page, and implement the function below.

// Sorts the elements of the given array, in place, in increasing order.

// Must use the Bubble Sort algorithm.

public static void sort(int[] arr) {

int n = arr.length;

boolean sorted = false;

while (!sorted) {

sorted = true;

for (int j=0; j < n-1; j++) {

if (arr[j+1] < arr[j]) {

swap(arr, j, j+1);

sorted = false;

}

}

}

}

// Swaps, in place, the positions of elements i and j in the given array.

private static void swap(int[] arr, int i, int j) {

int tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

}

Comment: It is OK to write this program without a helper method. In this case, the swapping code is written directly in the main loop.

2. (5 points) Suppose that we have to sort *N* elements, using *Bubble Sort*. What is the algorithm's running time efficieny, using the big *O* notation? Explain your answer.

Answer: *O*(*N*2), or *O*(*N*\*(*N*-1)), which is essentially the same thing (for a large *N*). In the worst case we have to do *N* sorting stages, and each stage requires *N*-1 comparisons.

3. (6 points) Propose a version of the *Bubble Sort* algorithm that runs faster than the basic version of the algorithm described in the help page. Don't write any code, just describe the general idea. (If needed, take another look at the example in the *Bubble Sort* help page).

Answer: At the end of each sorting stage *i*, the element in position *N*-*i* has arrived to the right place, and will not be swapped any further. Therefore, in each sorting stage *i*, we can evaluate only the elements in positions 0 to *N*-*i* (instead of evaluating *N* elements in each stage). This cuts down half of the required comparisons.

4. (12 points) Consider the function ����. The simple but expensive way to compute this function is computing the value of the expression �� ∙ �� ∙ ⋯ ∙ ��, *n* times.

As it turns out, the ����function has a nice property that can lead to a much better solution. For example, note that ��24 = ��12 ∙ ��12. And how to compute, say, ��25? Well, think about it.

Use the insight described above for generalizing and implementing an efficient recursive algorithm for computing ����, for any non-negative *n*.

1

// Computes *x* raised to the power of *n*, recursively, and efficiently.

public static int power(int x, int n) {

if (n == 0) return 1;

if (n == 1) return x;

if ((n % 2) == 0) {

double p = power(x, n/2);

return p \* p;

} else {

return x \* power(x,n-1);

}

}

Comments:

• Solutions that are based *only* on the recursive implementation "return x \* power(x,n-1)" are just as inefficient as computing , *n* times. And, they don't use the insight described in the question.

• Solutions that call power(x,n/2) *twice* are correct, but inefficient.

Questions 5-12 deal with *queues*, and *balanced queues*.

Read help pages 2,3,4 before proceeding.

General comments about questions 5-12:

• When iterating over the elements of the qs array, either "for each" loops, or regular loops using an index, are acceptable.

• When iterating using an index, a common error is using this[i] instead of qs[i]. If you made this error, and, most likely, repeated it throughout answers 5-12, we made a point deduction *only once*.

5. (5 points) Implement the following BalancedQ method:

// Returns the total number of elements in all the internal queues in this balanced queue. public int getSize() {

int size = 0;

for (LinkedQ q : qs) {

size += q.size();

}

return size;

}

Answer question 6, or question 7.

6. (8 points) Implement the following BalancedQ method:

// Adds the given element to the first shortest internal queue.

public void add(int e) {

LinkedQ qMin = qs[0];

for (LinkedQ q : qs) {

if (qMin.size() > q.size()) {

qMin = q;

}

}

qMin.add(e);

}

2

7. (8 points) Implement the following BalancedQ method:

// Removes the element at the head of the first longest internal queue,

// and returns the removed element.

public int remove() {

LinkedQ qMax = qs[0];

for (LinkedQ q : qs) {

if (qMax.size() < q.size()) {

qMax = q;

}

}

return qMax.remove();

}

Answer all the remaining questions.

8. (10 points) Implement the following BalancedQ method. Make sure that the implementation will not change the internal queues. In other words, it is not allowed to use the remove operation. Hint: Use an iterator.

// Returns a linked queue containing all the elements in this balanced queue.

// The returned queue contains all the elements in the first internal queue,

// then all the elements in the second internal queue, and so on.

public LinkedQ bigQ() {

LinkedQ lq = new LinkedQ();

for (LinkedQ q : qs) {

if (!q.isEmpty()) {

Iterator itr = q.iterator();

while (itr.hasNext()) {

lq.add(itr.next());

}

}

}

return lq;

}

9. (6 points) Suppose that the balanced queue uses *N* internal queues, and manages a total of *M* elements, in all the internal queues. What is the running-time efficiency of the add and remove operations of the BalancedQ class, in terms of big *O*?

Answer: Both have *O*(*N*) efficiency, because of the maximum / minimum computation which is done each time we add or remove an element. Each of these max/min finding requires (worst case) *N* iterations. *M* is irrelevant to this question.

Comment: It is OK to assume that the add, remove, and size operations of the LinkedQ class are each *O*(1). Students that assumed otherwise may have given answers that are not *O*(*N*). In such a case, there should be an explanation to the answer.

10. (6 points) Suppose that instead of using the add method specified in the BalancedQ class, we wish to use an addRandom method. Read the documentation given below, and implement the method.

// Selects at random one of the internal queues, and adds the given element to it. // Each queue is selected with equal probability.

public void addRandom(int e) {

int index = (int)(Math.random() \* qs.length);

qs[index].add(e);

}

3

11. (6 points) Suppose we want to make our balanced queue application more general. In particular, we want to manage not only queues of integer numbers, but queues of *any given object*. For example, queues of jobs in a factory, queues of queries in a search engine, queue of airline reservations, and so on. Describe how you will go about making this extension. In particular, explain which changes must be made in each class, but don't write any code.

Answer: The current solution manages queues of int values. If we wish to manage queues of any given object, we should go through the Node, Iterator, LinkedQ, and BalancedQ classes, and replace each type reference "int *data*" with "T *data*" (or some other capital letter other than T), in each place in the code that uses *data* to refer to the elements of the queues. This will make the code *generic*.

12. (6 points) This question is about the LinkedQ class (not about BalancedQ). One way to implement this class is to start from a "blank page", and write all the required code. Is there another, simpler way to go about developing this class? Describe the simpler approach, and two reasons why it may be a better approach.

Answer: We can base the implementation of LinkedQ on the implementation of Java's LinkedList class, or some other, similar LinkedList implementation, using *inheritance*. This will require overriding the insert and remove methods. The answer should also mention two advantages. For example: (i) writing less code, (ii) making less errors, (iii) using code which is familiar to other programmers.

(One more page)

4

13. (15 points) Read the *Binary Numbers* help page. Then write a Vic program that computes the decimal value of a binary number. The program's input starts with -1 (minus 1), then come the digits of the binary number, and then comes another -1. For example, the binary number 110 appears in the input as the sequence of numbers -1, 1, 1, 0, -1. The binary number may have leading 0's. For example, 00101 will appear in the input as -1, 0, 0, 1, 0, 1, -1. The program reads the input, and prints the decimal value of the binary number represented by the input. Note: Your program must be written in the symbolic Vic language documented in the Vic help page. The program must use reasonable variable names, and indentation, as shown in the program example in the Vic help page.

read

store minusone

load zero

store result

LOOP:

read

store x

sub minusone

gotoz END

load result

add result

add x

store result

goto LOOP

END:

load result

write

stop

Comments:

• Using other names for variables and labels is fine, as long as the names are descriptive.

• At the beginning of the program, instead of executing "read" and then "store minusone", we can execute "load zero", "sub one", "store minusone". In this case we also have to execute a "read" command that gets rid of the first "-1" in the input.

• Another possibility is not using a "minusone" variable at all. In this case, the code can compare each read input to the value -1 directly. This solution will be less efficient, since each iteration will have to use more commands than just "sub minusone".

5

Bubble Sort (questions 1-3)

This section describes a sorting algorithm called *Bubble Sort* (the reason behind the algorithm's name is not important).

The input to the algorithm is typically an array.

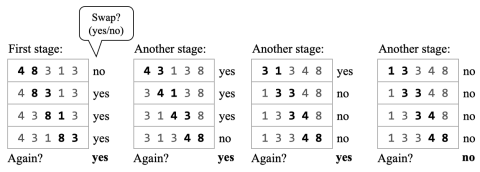
The algorithm proceeds in stages, as follows.

In each stage, we go through all pairs of elements, from left to right.

If a pair is out of order, we swap the two elements (change their positions in the input). If any swaps were made during the stage, we make another stage.

We stop when no swaps were made during the stage.

For example, if we have to sort the input **4, 8, 3, 1, 3**, in increasing order, the algorithm proceeds as following:

6

Queues(questions 5-12)

Queue

A *queue* (״תור״ (is a simple data structure: A sequence of elements with entry and exit points. When asked to add a given element, we add it to the tail of the queue. When asked to remove an element, we remove, and return, the element at the head of the queue.

(These operations are sometimes called enque and dequeu, but we prefer the more intuitive names add and remove).

A queue can be implemented using a linked list. Such an implementation is called "linked queue".

Balanced Queue

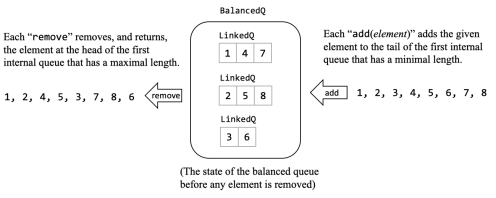
A *balanced queue* (מאוזן״ ״תור (is a data structure that uses several internal queues.

When asked to add an element to the balanced queue (add(*element*)), we add it to the *shortest* internal queue. When asked to remove an element from the internal queue, we remove, and return the element, from the *longest* internal queue.

If there is more than one internal queue that has a minimal length, we select the first queue that has a minimal length. If there is more than one internal queue that has a maximal length, we select the first queue that has a maximal length.

These rules ensure that all the internal queues will always have more or less the same number of elements.

Here is an example of a balanced queue that uses three internal queues (*N*=3). The picture shows what happens when we add several elements, one by one, and then remove elements, one by one.



Such data structures are used to support the servicing of the millions of calls (פניות (that arrive every second to search engines, call centers, reservation systems, and so on. The system uses several (in some cases, *many*) servers for handling the calls, and a balanced queue for "balancing the load" among the servers.

A balanced queue can be implemented using an array of linked queues. The next two help pages document and illustrate such an application.

7

Linked Queue (questions 5-12)

The Node, Iterator, and LinkedQ classes, whose API is given below, are very similar to the Node, Iterator, and LinkedList classes discussed in the course lectures and homework assignments. There is no need to know how these classes are implemented, since in the exam we will only use their API's.

| // Represents a node for a linked list.  // The node holds an int value.  public class Node {  // Creates a node holding the given int value. public Node (int data)  // The int value of this node.  public int getData()  // The int value of this node, as a string. public String toString()  } | // Represents a linked list that behaves like a queue: // Elements are added to the tail of the list,  // and are removed from the head of the list.  public class LinkedQ {  // Initializes a new empty queue.  public LinkedQ()  // Returns the number of elements in this queue. public int size()  // Adds the given element to this queue.  public void add(int e)  // Removes and returns an element from this queue. public int remove()  // Returns true if this queue is empty.  public boolean isEmpty()  // Returns an iterator for this queue.  public Iterator iterator()    // A textual representation of this queue,  // listing all the queue elements.  public String toString()  } |
| --- | --- |
| // Represents an iterator for iterating through the // elements of a linked list.  public class Iterator {  // Constructs an iterator, starting at the given node. public Iterator(Node node)  // Returns true if there are more elements // in this iteration, false otherwise.  public boolean hasNext()  // Returns the next element in this iteration. // Should be called only if hasNext() is true. public int next()  } |

Below is a code segment (and output) that tests the LinkedQ class:

| Test code:  LinkedQ q = new LinkedQ();  q.add(1); q.add(2); q.add(3);  q.add(4); q.add(5);  System.*out*.println(q);  System.*out*.println(q.remove()); System.*out*.println(q.remove()); System.*out*.println(q);  q.add(6); q.add(7);  System.*out*.println(q); | Output: (some white space added)  (1 2 3 4 5) // The queue after adding 1, 2, 3, 4, 5.  1 // Remove  2 // Remove  (3 4 5) // The queue after removing two elements. (3 4 5 6 7) // The queue after adding 6, 7. |
| --- | --- |

8

Balanced Queue (questions 5-12)

| // Represents a balanced queue.  public class BalancedQ {  LinkedQ[] qs; // Array of queues  // Initializes a balanced queue with  // N empty internal queues.  public BalancedQ(int N) {  qs = new LinkedQ[N];  for (int i=0; i < N; i++) { qs[i] = new LinkedQ();  }  }  // Checks if all the internal queues are empty. // If so, returns true. Otherwise, returns false. public boolean isEmpty() {  for (LinkedQ q : qs) {  if (!q.isEmpty() {  return false;  }  }  return true;  }    // The class code continues on the right. | // Returns the total number of elements in all  // the internal queues.  public int size() {// code = question 5 }  // Adds the given element to the first shortest  // internal queue.  public void add(int e) {// code = question 6 }  // Removes the element at the head of the first longest // internal queue, and returns the removed element. public int remove() { // code = question 7 }  // Returns a linked queue containing all the elements // in this balanced queue.  // The returned queue contains all the elements  // in the first internal queue, then all the elements // in the second internal queue, and so on.  public LinkedQ bigQ() {  // code = question 8  }  // A textual representation of this balanced queue. // Lists all the internal queues, one after the other. public String toString() { // code omitted }  } |
| --- | --- |

Implementation note: The BalancedQ class code shown above illustrates the use of two styles of "for" loops: The "regular" style, and the "for each" style. In your answers, you can use either style, as you please.

Below is test code and results:

| Test code:  // Constructs a balanced queue with 3 internal queues: BalancedQ bq = new BalancedQ(3);  // Adds some elements:  bq.add(1); bq.add(2); bq.add(3); bq.add(4); bq.add(5); bq.add(6); bq.add(7);  System.*out*.println(bq);  // Prints a "big queue" that contains all the elements // in all the internal queues, one after the other: System.*out*.println(bq.bigQ());  // Removes some elements (without printing them): bq.remove(); bq.remove(); bq.remove(); System.*out*.println(bq);  /  / Adds some elements:  bq.add(8); bq.add(9);  System.*out*.println(bq); | Output: (some white space added)  // After adding 1, 2, 3, 4, 5, 6, 7:  Queue0: (1 4 7)  The toString of BalancedQ  Queue1: (2 5)  creates this output (a string with  Queue2: (3 6)  line breaks)  // All the elements in the balanced queue,  // in the order of one internal queue after another. (1 4 7 2 5 3 6)  // After removing three elements:  Queue0: (7)  Queue1: (5)  Queue2: (3 6)  // After adding 8, 9:  Queue0: (7 8)  Queue1: (5 9)  Queue2: (3 6) |
| --- | --- |

9

Binary Numbers (question 13)

A binary number ���� ����−1 ����−2 ,⋯ , ��0 is a code in which each ����is either the digit 0, or 1. The decimal value of a given binary number can be computed efficiently, using the following formula:

��������������(���� ����−1 ����−2 ,⋯ , ��0) = (⋯ ((���� ∙ 2 + ����−1) ∙ 2 + ����−2) ∙ 2 + ⋯ + ��1) ∙ 2 + ��0 This is a bit confusing, so let's consider binary numbers with four digits:

��������������(��3 ��2 ��1 ��0) = ((��3 ∙ 2 + ��2) ∙ 2 + ��1) ∙ 2 + ��0

Examples:

��������������(��������) = ((�� ∙ 2 + ��) ∙ 2 + ��) ∙ 2 + �� = 5

��������������(��������) = ((�� ∙ 2 + ��) ∙ 2 + ��) ∙ 2 + �� = 12

��������������(��������) = ((�� ∙ 2 + ��) ∙ 2 + ��) ∙ 2 + �� = 3

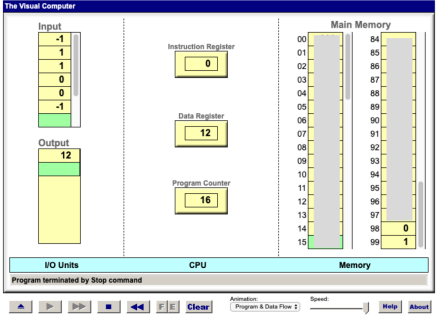
Another example (binary number with six digits):

��������������(������������) = (((((�� ∙ 2 + ��) ∙ 2) + ��) ∙ 2 + ��) ∙ 2 + ��) ∙ 2 + �� = 43

As the examples illustrates, the same formula can be used for computing the decimal value of a binary number of any length.

10

Vic (question 13)



(An example of a Vic program that computes the decimal value of the binary number 1100, which happens to be 12. The contents of the memory is hidden.)

**The Vic symbolic machine language**

In the documentation given below, D is the data register, *x* is a variable that represens an address in memory, and *LABEL* is a label that represents an address in memory. Following convention, variables are written in lower-case, and labels in upper-case.

The built-in variables zero and one are predefined to refer to addresses 98 and 99, respectively. These addresses always contain the values 0 and 1, respectively.

Programmers are welcome to make up variable names and labels as they please. The assembler translates these symbols into memory addresses, and the programmers need not worry about it.

Command: read

write

load *x*

store *x*

add *x*

sub *x*

goto *LABEL* gotoz *LABEL* gotop *LABEL* stop

Meaning:

D = input

output = D

D = *x*

*x* = D

D = D + *x*

D = D - *x*

goto *LABEL*

if (D==0) goto *LABEL* if (D>0) goto *LABEL* Stops execution

Example of a symbolic Vic program:

// Reads a number from the input. // if the number is greater than 0, sets the // variable v to 1. Otherwise, sets v to 0. read

gotop POSITIVE

load zero

store v

stop

POSITIVE:

load one

store v

stop

11