```
(* The types of mutable queues. *)
type 'a queuenode = { v : 'a;
                      mutable next : 'a queuenode option}

type 'a queue = { mutable head : 'a queuenode option;
                  mutable tail : 'a queuenode option }

let qn1 : int queuenode = {v = 1; next = None;}
let qn2 : int queuenode = {v = 2; next = Some qn1;}
let q : int queue = {head = Some qn2; tail = Some qn1;}
```

```
public int[][] growByTwo(int[][] arr) {
   int height = arr.length * 2;
   int width = arr[0].length * 2;
   int[][] newArr = new int[height][width];
```

```
public int[][] growByTwo2(int[][] arr) {
   int height = arr.length * 2;
   int width = arr[0].length * 2;
   int[][] newArr = new int[height][width];

   for(int i=0; i<newArr.length; i++ ){
      for(int j=0; j<newArr[i].length; j++) {
         newArr[i][j] = arr[i/2][j/2];
      }
   }
   return newArr;
}
```

j. $\boxed{T}$ F In our GUI library, an `event_listener` is a first-class function stored in the hidden state of a `notifier` widget. When an event occurs in the widget, the `notifier` invokes all of the stored `event_listeners`.

k. $\boxed{T}$ F In the OCaml ASM, first-class functions are stored in the heap and may have local copies of variables that were on the stack when they were defined.

HELPER FX TO FIND COUNT NEIGHBORS

b. (18 points) Complete the definition of `step` below. (You may find the static library methods `Math.min` and `Math.max` useful, but don't worry if you don't end up using them: there are a number of different ways to write a correct solution.) Your solution should call `liveOrDie` at some point.

Answer:

```
public static int[][] step(int[][] current) {
  int width = current.length;
  int height = current[0].length;
  int[][] next = new int[width][height];
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < height; j++) {
      int count = 0;
      for (int x = Math.max(i - 1, 0); x <= Math.min(i + 1, width-1); x++) {
        for (int y = Math.max(j - 1, 0); y <= Math.min(j + 1, height-1); y++) {
          if (i != x || j != y) {
            count += current[x][y];
          }
        }
      }
      next[i][j] = liveOrDie(current[i][j], count);
    }
  }
  return next;
}
```

c. (4 points) Consider the following two possible implementations of a function `is_singleton`, which is intended to return `true` when given a valid queue that has exactly one element and `false` if given a valid queue that has zero or more than one elements. (Note: it does not matter what the functions do when given an *invalid* queue.)

```
(* A *)
let is_singleton (q:'a queue) : bool =
  q.head <> None && q.head == q.tail

(* B *)
let is_singleton (q:'a queue) : bool =
  match q.head, q.tail with
  | Some qn1, Some qn2 -> qn1 == qn2
  | _, _ -> false
```

b. (7 points) Here is a (correct) implementation of the `take` operation on streams and one example test case. This version of `take` is written using standard *recursion*:

```
let take (n:int) (s:'a stream) : 'a list =
 let rec loop (i:int) : 'a list =
  if i <= 0 then [] else (s.produce ())::(loop (i-1))
 in
 loop n

let test () : bool =
 let s = constant_stream 42 in
 take 5 s = [42; 42; 42; 42; 42]
;; run_test "take five from the constant 42 stream"
```

Rewrite `take`, by completing the code template below, to use *iteration via tail recursion* instead of plain recursion. Your implementation should use constant stack space and may make use of the library function `List.rev`, which reverses a list. *Hint: you will need to add an extra argument to* `loop`.

Answer:

```
let take (n:int) (s:'a stream) : 'a list =
  let rec loop (i:int) (acc:'a list) : 'a list =
    if i <= 0 then List.rev acc else
      loop (i-1) (s.produce () :: acc)
  in
  loop n []

    }
```

e. $\boxed{\text{T}}$ F The use of `Math.max` in the `HPair` implementation of `getHeight` is an example of a static method invocation.

f. $\boxed{\text{T}}$ F It is possible to add a second method to the `LabelController` interface *without* modifying the `Label` class and still have the label class compile without error.

g. Which of the following should we add as an additional requirement to strengthen the queue invariant so that it ensures proper encapsulation? (And thus correctly ruling out the situation depicted in Figure 1 as invalid.)

- ☐ Every `qnode` reachable from `q.head` has no aliases.
- ☐ Every reference value reachable from `q` has at most two aliases.
- ☒ Every `qnode` reachable from `q` is reachable only by following a series of references starting from `q.head` or `q.tail`.

g. T $\boxed{\text{F}}$ In the Java ASM, large data structures such as object values are stored in the stack, not the heap.
No: In both ASMs, large structures live in the heap.

h. $\boxed{\text{T}}$ F In the OCaml ASM, bindings of variables to values in the stack are immutable, while in Java they are mutable.

i. $\boxed{\text{T}}$ F A Java variable of type `String` behaves like an OCaml variable of type `string option ref`.

2. Java Jargon (8 points)

Briefly (two sentences max) define the phrase "dynamic dispatch" as it applies to Java.

This phrase refers to the fact that the result of invoking a method on an object depends on the object's dynamic class, not its static type.

h. $\boxed{\text{T}}$ F In Java, *simple inheritance* refers to the idea that a subclass extends its parent only by adding new fields or methods.

i. $\boxed{\text{T}}$ F In Java, a *static method* is associated with the class containing it, not an instance of the class.

TODO

3. **Java Array Programming** (18 points)

Recall that in Java a two dimensional array might be *ragged*, which means that it is not "rectangular" in shape. More precisely, a ragged 2D array `a` has an index `i` such that `a[0].length` is not equal to `a[i].length`.

Write a function `pad`, that takes a potentially ragged 2D array of integers and returns a "padded" copy `p`, which is the smallest rectangular array such that if `a[i][j]` is defined (i.e. doesn't lead to an `ArrayIndexOutOfBoundsException`) then `p[i][j] = a[i][j]` and otherwise `p[i][j] = 0`.

Pictorially, if `a` is as shown below, then `pad(a)` will be the same as `a` but with `0`'s filling out the rectangle.:

```
     a                      pad(a)
0 1 2 3 0                 0 1 2 3 0
4 5                       4 5 0 0 0
6 7 8                     6 7 8 0 0
9                         9 0 0 0 0
```

You may assume that the input array `a` is not null and that it contains no null sub-arrays.

```
// assume that a is non-null and that it contains no null elements
public static int[][] pad(int[][] a) {
  int[][] result = new int[a.length][];
  int max = 0;
  for(int i=0; i<a.length; i++) {
    if (a[i].length > max) {
      max = a[i].length;
    }
  }
  for(int i=0; i<a.length; i++) {
    result[i] = new int[max];
    for(int j=0; j<a[i].length; j++) {
      result[i][j] = a[i][j];
    }
  }
  return result;
}
```

Use the four step design methodology to implement a static method called `isGoodSquare` that takes as input a two-dimensional array of ints and returns true if and only if the array is a square matrix where the sum of the numbers in every horizontal row and every vertical column is the same. The method should return **`false`** for ill-formed inputs (null, non-square array). On an empty (0-length) input array, it should return **`true`**.

a. Step 1 is *understanding the problem.* You don't have to write anything for this part—your answers below will demonstrate whether or not you succeeded with Step 1.

b. Step 2 is *formalizing the interface.* We have done this for you:

```
public static boolean isGoodSquare(int[][] sq) { ... }
```

c. The next step is *writing test cases.* For example, one possible testcase is a valid good square:

```
@Test
public void testValidGoodSquare() {
  int sq[][] = {{8,1,6},{3,5,7},{4,9,2}};
  assertEquals(true, isGoodSquare(sq));
}
```

The interesting parts of this test are the name, which should communicate the reason for the test ("valid good square"), plus the expected result value and the array to be tested. To avoid writing too much boilerplate, we might abbreviate this test case as follows:

Test name	Input array	Expected output
Valid good square	`{{8,1,6},{3,5,7},{4,9,2}}`	`true`

On the next page, write three more test cases for this method in the same style.

Possible answers: Unequal sums, Not square, Null matrix, empty matrix, unary matrix

d. The final step is to *implement the method.* Please do so below. Do not use any external libraries.

Answer:

```
public static boolean isGood(int[][] sq) {

  if (sq == null) return false;
  int len = sq.length;

  if (len == 0) return true;

  int candSum = 0;
  for(int i = 0; i < len; i++) {
    candSum += sq[0][i];
  }

  for(int i = 0; i < len; i++ ) {
    int sum1 = 0;
    int sum2 = 0;
    if (sq[i] == null || sq[i].length != len) return false;




for(int j = 0; j < len; j++) {
  sum1 += sq[i][j];
  sum2 += sq[j][i];
}
if (sum1 != candSum || sum2 != candSum) return false;
}

return true;
```

Implement a function, called `dedup`, which removes all adjacent repeated elements from a queue. In other words, this function should remove any value from the queue that is equal to the value that occurs immediately before it.

For example, if a queue `q` contains the values `1,2,2,3,3,2` (in that order) then after an execution of `dedup q`, the queue `q` should contain `1,2,3,2` (in that order), where the second `2` and the second `3` have been removed.

might see it in a .mli file.

```
val dedup : __________ 'a queue -> unit ______________
```

```
let dedup (q : 'a queue) : unit =
     let rec helper (curr : 'a qnode) (nxt : 'a qnode option) : unit =
      begin match nxt with
          | Some n -> if curr.v = n.v then
                  (curr.next <- n.next;
                       if curr.next = None then
                                 q.tail <- Some curr
                           else
                       helper curr curr.next)
                else
                       helper n n.next
          | None -> ()
          end
     in begin match q.head with
     | Some qn -> helper qn qn.next
     | None -> ()
     end
```

```
type 'a qnode = { v : 'a; mutable next : 'a qnode option; }
type 'a queue = { mutable head : 'a qnode option;
                  mutable tail : 'a qnode option }
```

Use the design process to implement a function, called `join`, that takes two queues and modifies them so that all of the `qnodes` of the second queue are moved to the end the first queue while retaining their order.

For example, suppose queue `q1` contains the values `1,2` and queue `q2` contains the values `3,4`. Then after an execution of `join q1 q2`, the queue `q1` should contain `1,2,3,4`, and `q2` should be empty.

The join function may assume that `q1` and `q2` are valid queues, generated by the standard queue operations. If `q1` and `q2` are aliases to the *same* queue, then `join` should have no effect. As the purpose of this function is to mutate its arguments, it should always return `()`.

```
let join (q1:'a queue) (q2 : 'a queue) : unit =
  if (q1 == q2 || q2.head == None) then ()
  else begin match q1.tail with
  | Some qn -> (qn.next <- q2.head;
             q1.tail <- q2.tail)
  | None -> (q1.head <- q2.head;
           q1.tail <- q2.tail)
  end;
  q2.head <- None;
  q2.tail <- None
```

a. (6 points) Below is the main function that generates the window shown in the example. Complete the body of the event listener for the slider so that as the slider changes value in response to mouseclicks, the label always displays the slider's current value.

Hint: read over the next part to see the interface of the Slider widget.

Hint: you can use the static method `Integer.toString` *to convert an int to a String.*

Hint: we have given you several lines for your answer. You may or may not need to use all of them.

```
class Main {
  public static void main(String[] args) {
     final Label lab = new Label("50");
     final Slider slider = new Slider(new Dimension(200,20));

     slider.setEventListener(new EventListener() {
       public void listen(Event e) {

              ____________int v = slider.getValue();____________

              __________String s = Integer.toString(v);__________

              ________________lab.setLabel(s);________________

       }
     });

    Widget spacer = new Space(new Dimension(10,10));
    Widget group = new Vpair(new Vpair(slider,spacer), lab);
    Widget toplevel = new Centered(group);
    GUI.createAndShowGUI(toplevel);
  }

}
```

b. (10 points) Now fill in the blanks in the `repaint` and `handle` methods to complete the implementation of the `Slider` class. Note: the `handle` method is shown on the next page.

```
public class Slider implements Widget, NotifierController {

  /* Private state of the Slider */
  private final Dimension min;
  private EventListener listener = null;
  private int value = 50;


  /* Construct a slider with an initial percentage value of 50,
     no attached event listener , and a minimum size of d */
  public Slider(Dimension d) {
     min = d;
  }

  /* Return the current percentage value of the slider */
  public int getValue() {
     return value;
  }

  /* Set an object to serve as an EventListener for the slider . The listen
     method of this object will be invoked whenever the slider changes value */
  public void setEventListener(EventListener el) {
    listener = el;
  }


  /* Draw the Slider as a rectangle taking up the entire space specified
     by the graphics context . The filled portion of the rectangle should
     match the percentage value of the slider . */
  public void repaint(Gctx gc) {
    int w = gc.getWidth();
    int h = gc.getHeight();

    int filledWidth = ___________value * w / 100;___________

    Position origin = new Position(0,0);
    gc.drawRect(origin, w, h);
    gc.fillRect(origin, filledWidth, h);
  }
```

```
/* Handle an event that occurs in this widget. You may or may not
   need to use all of the lines below. */
public void handle(Gctx gc, Event e) {
  int w = gc.getWidth();
  Position p = gc.eventPosition(e.getPosition());

          if (!e.isMouseClick() || p == null) return;
  ______________________________________________________________

                      value = p.x * 100 / w;
  ______________________________________________________________

                      if (listener != null)
  ______________________________________________________________

                        listener.listen(e);
  ______________________________________________________________


  ______________________________________________________________

}

/* Access the stored initial size */
public Dimension minSize() {
  return min;
}
```

Complete the implementation below. Don't forget to fill in a type annotation for `prev`!

```
let rec insert_before (q : 'a queue) (x : 'a) (y : 'a) =
  let rec loop (prev : 'a qnode) (qno : 'a qnode option) : unit =
    begin match qno with
    | Some qn -> if qn.v = y then
        prev.next <- Some {v = x; next = qno}
      else
        loop qn qn.next
    | None -> ()
    end in
  begin match q.head with
  | Some qn ->
      if qn.v = y then
        q.head <- Some { v = x; next = q.head }
      else loop qn qn.next
  | None -> ()
  end
```

or

```
let insert_before (q : 'a queue) (x : 'a) (y : 'a) : unit =
  let rec loop (prev : 'a qnode option) (qno : 'a qnode option) : unit =
    begin match qno with
      | Some qn ->
          if qn.v = y then
            let new_q = Some { v = x; next = qno } in
            begin match prev with
              | None -> q.head <- new_q
              | Some n -> n.next <- new_q
            end
          else loop qno qn.next
      | None -> ()
    end
  in
  loop None q.head
```

Grading Scheme: 2 points each

- *type for* `prev`*, consistent with implementation*
- *Check qn.v for equality with y (both at head and each qno)*
- *Create new qnode containing x (both at head and each qno)*
- *new qnode's next reference is qno (or Some qn) for each qno*
- *new qnode's next reference is q.head (or Some qn)*
- *update prev.next when value at qno*
- *update q.head when value at first node*
- *recursive call to loop with correct arguments*
- *initial call to loop with correct arguments*

4. **Java Subtyping: Inheritance, Interfaces and Dynamic Classes (20 points)**

Hint: Draw the subtype hierarchy. Consider these Java class and interface definitions:

```java
interface I { public A method1(); }

interface J extends I { public B method2(); }

interface K { public B method3(B b); }

class A implements I, K {
   public A method1() { return new A(); }
   public B method3(B b) { return b; }
}

class B implements J {
   public A method1() { return new A(); }
   public B method2() { return new C(); }
   public B method3(B b) { return b; }
}

class C extends B implements K {
   public B method4(B b) { return new B(); }
}
```

For each code fragment below, fill in the blank with the name of the *dynamic class* of the value stored in the variable indicated on the line, or write "ill typed" if the code fragment contains a compile-time type error (i.e. Eclipse would put a red line under some part of the program).

```
(a)  K k1 = new C();                                      k1:________C____________

(b)  I i1 = new C();                                      i1:________C____________

(c)  I i2 = new A();                                      i2:________A____________

(d)  K k3;
     if (true) {k3 = new A();} else {k3 = new B();}       k3:____ill typed________

(e)  K k4;
     if (true) {k4 = new A();} else {k4 = new C();}       k4:________A____________

(f)  J j1 = (new A()).method3(new C());                   j1:________C____________

(g)  B b1 = new C();
     B b2 = b1.method4(new B());                          b2:___ill typed_________

(h)  J j2 = new B();
     J j3 = j2.method2();                                 j3:________C____________

(i)  C c1 = new C();
     Object o1 = c1.method4(new C());                     o1:________B____________

(j)  C c2 = new Object();                                 c2:___ill typed_________
```

Complete the implementation below. Don't forget to fill in a type annotation for `prev`!

```
let rec insert_before (q : 'a queue) (x : 'a) (y : 'a) =
  let rec loop (prev : 'a qnode) (qno : 'a qnode option) : unit =
    begin match qno with
    | Some qn -> if qn.v = y then
        prev.next <- Some {v = x; next = qno}
      else
        loop qn qn.next
    | None -> ()
    end in
  begin match q.head with
  | Some qn ->
      if qn.v = y then
        q.head <- Some { v = x; next = q.head }
      else loop qn qn.next
  | None -> ()
  end
```

or

```
let insert_before (q : 'a queue) (x : 'a) (y : 'a) : unit =
  let rec loop (prev : 'a qnode option) (qno : 'a qnode option) : unit =
    begin match qno with
      | Some qn ->
          if qn.v = y then
            let new_q = Some { v = x; next = qno } in
            begin match prev with
              | None -> q.head <- new_q
              | Some n -> n.next <- new_q
            end
          else loop qno qn.next
      | None -> ()
    end
  in
  loop None q.head
```

Grading Scheme: 2 points each

- *type for* `prev`*, consistent with implementation*
- *Check qn.v for equality with y (both at head and each qno)*
- *Create new qnode containing x (both at head and each qno)*
- *new qnode's next reference is qno (or Some qn) for each qno*
- *new qnode's next reference is q.head (or Some qn)*
- *update prev.next when value at qno*
- *update q.head when value at first node*
- *recursive call to loop with correct arguments*
- *initial call to loop with correct arguments*

4. **Mutable Queues Implementation** (23 points)

Implement a function, called `intersperse`, that inserts a value *between* every value in a queue.

For example, if `q` contains the values `1,2,3` (in that order) then, after an execution of `intersperse 0`
the queue `q` should contain the values `1,0,2,0,3` (in that order). On the other hand, if `q` is empty or contains a single element, then a call to `intersperse` should not modify the queue. All calls to `intersperse` should leave `q` in a valid state.

Your implementation may define a single recursive helper function to traverse the queue. However it may not call any other functions, such as `from_list`, `to_list`, `deq` or `enq`.

```
let intersperse (x : 'a) (q : 'a queue) : unit =
  let rec loop (n : 'a qnode) : unit =
    begin match n.next with
     | None -> ()
     | Some nn ->
      n.next <- Some { v = x; next = Some nn };
      loop nn
  end in
  begin match q.head with
   | Some h -> loop h
   | None -> ()
  end
```

or

```
let intersperse (x :'a) (q : 'a queue) : unit =
  let rec loop (no : 'a qnode option) : unit =
    begin match no with
     | None -> ()
     | Some n ->
      begin match n.next with
       | Some nn ->
        let newnode = { v = x; next = Some nn } in
        n.next <- Some newnode;
        loop (Some nn)
       | None -> ()
      end
  end in
  loop q.head
```

Note: this next one is not quite correct as it compares two `Some` nodes with `==`. It should read `not (n.next == None)` instead. Or use pattern matching like above.

```
let intersperse (x : 'a) (q : 'a queue) : unit =
  let rec loop (qn: 'a qnode option) : unit =
    begin match qn with
      | None -> ()
      | Some n -> if not (qn == q.tail) then
                  let nxt = n.next in
                  n.next <- Some {v=x; next = nxt}; loop nxt
                else ()
   end in
   loop q.head
```

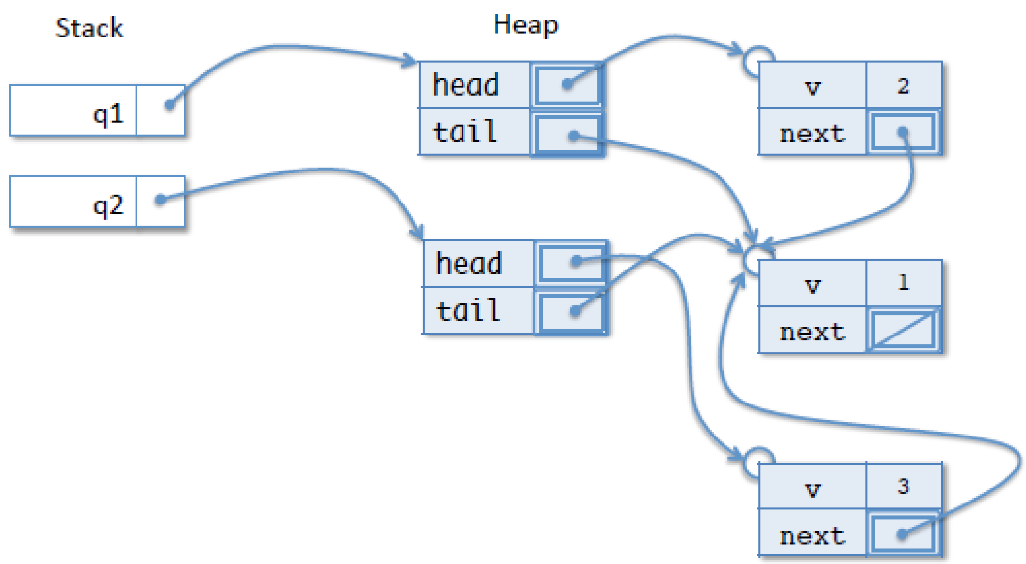

Figure 1: OCaml ASM state.

a. (8 points) Mark *all* of the following code snippets that, when run as the workspace c ASM, construct the state shown above. *There may be more than one program that work*

☐ (A)

```
let q1 = create ()
let q2 = create ()
;; enq 2 q1
;; enq 1 q1
;; enq 3 q2
;; enq 1 q2
```

☐ (B)

```
let q1 = create ()
let q2 = create ()
;; enq 2 q1
;; enq 1 q1
;; enq 3 q2
;; q2.head <- q1.head
```

☒ (C)

```
let q1 = create ()
let q2 = create ()
;; enq 2 q1
;; enq 1 q1
;; enq 3 q2
;; q2.tail <- q1.tail
;; begin match q2.head with
  | None -> failwith "impossible"
  | Some qn -> (qn.next <- q2.tail)
  end
```

☐ (D)

```
let q1 = create ()
let q2 = create ()
;; enq 2 q1
;; enq 1 q1
;; enq 3 q2
;; enq 1 q2
;; q1.tail <- q2.tail
```