1	2	3	4	5	6	7	Σ
20	10	15	10	15	10	20	100

Student number	2	0						
Name								

Midterm Exam — Data Structures (CS206A)

October 21, 2015, 9:00–11:45

Instructions:

- This booklet has **three pages** with **7 problems** in total. Check that you have all!
- The space provided should be sufficient for your answer. If you need scratch space, use the back side. If you need more space for your answer, you can also use the back side (but *indicate clearly on the front side* that your answer continues on the back).
- This is a **closed book** exam. You are not allowed to consult any book or notes.
- The questions have to be answered in **English**. Write clearly!
- To ensure a quiet exam environment, we will **not answer questions** during the exam. If you think there is a mistake in the question, explain so, and use common sense to answer the question.
- Before you start: Write your name and student number (one digit per square!) on **all pages** of this exam booklet (−5 points for missing names or unreadable numbers).
- Relax. Breathe. This is just an easy, silly, and stupid midterm exam.

Problem 1: (20 pts) For each of the following statements, say whether they are *true* or *false* (1 point for correct answer, −1 point for wrong answer, 0 points for no answer).

- Local variables of a method are stored in the stack frame of the method. True — False
- Stack frames are stored on the heap. True — False
- The garbage collector destroys objects that cannot be reached anymore. True — False
- If an array is created as `val A = Array(1, 2, 3)`, then the array's contents can never be changed. True — False
- All objects are stored on the heap. True — False
- When an exception happens inside a method, the method returns `null`. True — False
- The abstract data type *queue* is a LIFO. True — False
- In a statically-typed language, every variable has a type known when the program is compiled. True — False
- Divide-and-conquer is a form of recursion. True — False
- The class `class Point(val x: Int, val y: Int)` is mutable. True — False
- `enqueue` and `dequeue` are methods of the ADT Stack. True — False
- A *sentinel* is a list node that stores no list element but simplifies program code. True — False
- A circular buffer is often used to implement the queue ADT. True — False
- After setting `val s = List("CS206A", "CS206B", "CS206C")`, there are *seven* objects on the heap that can be reached from `s`. True — False
- The object `1 :: 7 :: 29 :: Nil` is an immutable object. True — False
- If `Client` is a *trait*, then `val b = new Client` creates an object of type `Client`. True — False
- The value of `(42 :: List(13, 7)).tail.head` is 13. True — False
- Recursive functions are elegant, but if the recursion is deep then stack overflow is a problem. True — False
- `def quak(f: Frog) = f.green` is a correct function definition if `Frog` is defined as `trait Frog { def green: String }`. True — False
- An exception is an object. It is stored on the heap. True — False

Problem 2: (10 pts) Fill in the following method `insertBefore` to insert element `x` into a *doubly-linked* list *in front* of node `n`, using the given `Node` class and assuming that there are sentinels at the front and rear of the list.

```
class Node(val el: String, var prev: Node, var next: Node)

def insertBefore(n: Node, x: String) {




}
```

Student number	2	0						
Name								

Problem 3: (15 pts) Draw the contents of the heap and of the active stack frames of all functions in the following code. Assume the code is started by the call `garden()`, and draw the situation at the point marked in the code.

```
def fruit(s: List[String]) {
  val k = s.tail
  if (k == Nil) {
    // draw situation here
  } else
    fruit(k)
  println(s.head)
}

def garden() {
  val L = "apple" ::
          "orange" ::
          "cherry" :: Nil
  fruit(L)
}
```

What is the output of calling `garden()`?

Problem 4: (10 pts) Express each of the following functions in Big-Oh notation (only write your answer without explanation).

(a) $n(n-1)(n-2)(n-3)/24 =$

(b) $(n-2)(\log n - 2)(\log n + 2) =$

(c) $n(n+1) - n^2 =$

(d) $n(n+1)/2 + n \log n =$

(e) $\ln((n-1)(n-2)(n-3))^2 =$

Problem 5: (15 pts) The following function `inStack` is given a stack `s` containing some integers, an integer `x`, and an *empty* queue `q`. The function returns `True` if the number `x` is contained in `s`, otherwise `False`.

Implement `inStack` without using any array, list, or any other container. You can only use a small number of integer variables. When the function returns, both the stack and the queue must be *unchanged*, that is, have exactly the *same contents* as at the beginning (so the queue `q` is empty again).

```
def inStack(s: Stack[Int], q: Queue[Int], x: Int): Boolean = {









}
```

Student number	2	0						
Name								

Problem 6: (10 pts) We have implemented a list class `LinkedList`. It contains a single field `front` for the front of the list. No sentinel is used. The node class is like this:

```
class Node(val el: String, var next: Node)
```

We are using a heuristic called *move-to-front*: every time we access an element, we move it to the head of the list. The idea is that we often search for the same element again, so having it at the front of the list speeds up the search.

Fill in the following method `moveToFront`, which makes the node *after* the given node `n` the head of the list. For instance, if the list contains A, B, C, D, E, and you call `moveToFront` with the node containing C, then the list becomes D, A, B, C, E. You can assume that there is a node after `n` (no error checking is necessary).

```
  def moveToFront(n: Node) {









  }
```

Problem 7: (20 pts) Given a sequence `s` of integers like `1, 2, 3, 4, 5, 6, 7, 8`, a *subsequence* is a sequence consisting of some elements of `s` in the original order. The elements do not need to be contiguous in `s`, so `1, 3, 4, 7` is a subsequence of the sequence above. The *empty sequence* is a subsequence of *every* sequence.

Implement a *recursive* function `subseqs` that computes all subsequences of a sequence of integers given as a list. The output is a list of the possible subsequences, in other words a list of lists. The subsequences can be arranged in any order. Do not forget the empty list as part of your output. Here are some examples of the output:

```
scala> subseqs(List())
res0: List[List[Int]] = List(List())
scala> subseqs(List(1))
res1: List[List[Int]] = List(List(1), List())
scala> subseqs(List(1, 2, 3))
res2: List[List[Int]] = List(List(1, 3), List(3), List(1, 2, 3), List(2, 3),
                             List(1), List(), List(1, 2), List(2))
```

Fill in the implementation of the function below. It must *recursively* call `subseqs` itself—you cannot define other functions.

```
def subseqs(s: List[Int]): List[List[Int]] = {
  if (s == Nil)
    return List(List())
















}
```