This lab is worth 10%. Each lab stream has been divided up into 1 hour slots, during one of which you will sit the practical test.

Please make sure you come to your scheduled lab stream, unless you have arranged otherwise with us.

Overview

Most of the labs that we get you to do during this course are formative in nature; i.e. you learn new things and acquire new skills during the course of completing them. This lab is a bit different. Instead of being formative this lab is summative in nature. It allows you (as well as us) to look back and see what knowledge and skills you have acquired so far.

In the previous assessed labs you could complete the work at any time before the deadline, using whatever resources you wished to. In this lab you must complete the task during the allocated time, with minimal resources at your disposal. Using only a terminal window, a basic text-editor, javac, and java, you must write some programs which meet the given specifications.

Remember we are *not* asking you to create your programs for the first time during your allocated slot. You should have already written all of the code required for this test during previous labs. All you need to do during this lab is to be able to reproduce some of your previous work.

We encourage you to write your code initially using whatever resources you need. Once you have done this, and are sure that your programs are working correctly, try to write them again without using any resources (you might need to peek occasionally). Keep doing this until you can write the code completely unaided. The best and recommended way to prepare for the test is to clearly understand what your code is doing. That way you don't have to write it exactly the same way each time. If you prepare well for this lab then you will find it pretty straightforward. Writing code like this, without any outside assistance, will build your confidence to tackle more demanding programming tasks.

Resources

You will be provided with semi-complete solutions to the lab work from week 4, 7, 10, and 11. You must implement the methods with missing bodies in order to pass the tests. Most of the code that you will have to write is code that should have previously been written by you. The associated lab handouts are available to view and are located

on the desktop of the test machines. The Java API can also be viewed by entering *api* in a terminal.

Note: You are not permitted to access your home directory, or any other files or computers, nor may you use the Internet during this lab.

The code that you will need to complete is as follows:

Part A - TableauApp from week 4 (2%)

Two of the following four methods (randomly selected by us) need to be implemented:

`rowLengthsDecrease(int[][] t)` A method that returns `true` if no row is longer than a preceding row, otherwise `false`.

`rowValuesIncrease(int[][] t)` A method that returns `true` if from left to right in any row, the integers are increasing, otherwise `false`.

`columnValuesIncrease(int[][] t)` A method that returns `true` if from top to bottom in any column, the integers are increasing, otherwise `false`.

`isSetOf1toN(int[][] t)` A method that returns `true` if the set of integers used is $\{1, 2, \ldots, n\}$ where n is the number of cells, otherwise `false`.

Part B - Tableau from week 7 (2%)

One of the following two methods (randomly selected by us) needs to be implemented:

`addToRow(Cell curr, int value)` This method takes a **`cell`** as a starting point and follows *right* pointing links until it finds a value which is greater than the given value or until the right pointing link is **`null`**. If it finds a bigger value then it replaces it with the given value and returns the previous value. If it comes to the end of the row it adds a new cell with the given value and returns **`null`**.

`addValue(Integer value)` You will need to take care of the case where the tableau is empty, as well as implementing the other cases. You can call the **`addToRow`** method to add the value to the first row. If **`addToRow`** returns **`null`** there is nothing more to do. If it returns a value then that value must be inserted into the

row below. If the row below is empty then a new cell should be added as the only item in that row, otherwise just call **`addToRow`** again to add the returned value to the row below.

Note that if you have to implement the addValue() method in Part B the empty tableau case has NOT been done for you.

Part C - Selection Sort or Insertion Sort from week 10 (2%)

The `sortNums()` method of either `SelectionSort.java` or `InsertionSort.java` (chosen at random by us) must be completed.

Selection sort assumes that you know how to pick out the *smallest* item in an array of items. We conceptually break the array into two pieces, left and right. We have n items altogether.

Pick the smallest item in the section from 0 to (n - 1) and swap it with whatever is in position 0. Pick the smallest item in the section 1 to (n - 1) and swap it with whatever is in position 1. Now pick the smallest item in section 2 to (n - 1) and swap it with whatever is in position 2...you get the idea. Stop when you get to position n - 2, since the item now in the last position (n - 1) must be the largest item.

```
for each position p in the array a except the last one {
   find the smallest item from position p to position (n – 1)
   swap the smallest item with whatever is at position p now
}
```

Insertion sort works the same way many people sort a hand of cards. We imagine that everything to the left of a certain point is already sorted. We take the first item to the right of that and "pull it out" (leaving a "gap"). We then move everything in the sorted part one place over to the right until a gap opens up at just the right place for us to "insert" what we pulled out. Our left-hand-side is still sorted, but now it is one item longer, and the right-hand-side is one item shorter.

```
for each position p in array a except the first {
   pull out the item at p and store it in variable 'key'
   move each item that is to the left of position p, and is
      greater than key, one place to the right
   put key in the leftmost vacated position
}
```

Part D - Merge Sort or Quick Sort from week 11 (2%)

The `sortNums()` method (and any dependent methods) of either `MergeSort.java` or `QuickSort.java` (chosen at random by us) must be completed.

```
mergeSort(left, right)
   if left < right
      mid = (left + right) / 2
      mergeSort(left, mid)
      mergeSort(mid + 1, right)
      merge(left, mid + 1, right)

merge(left, mid, right)
   copy nums[left...right] to temp[left...right]
   i = left, j = left, k = mid
   while i < mid and k <= right
      if temp[i] < temp[k] then nums[j++] = temp[i++]
      else nums[j++] = temp[k++]
   while i < mid   nums[j++] = temp[i++]
   while j <= right   nums[j++] = temp[k++]

quickSort(left, right)
    if left < right
        p = partition(left, right)
        quickSort(left, p)
        quickSort(p+1, right)

partition(left, right)
    pivot = nums[left]
    hole = left, i = left+1, j = right
    loop forever
        while j > hole && nums[j] >= pivot
           j--
        if j == hole then exit loop
        nums[hole] = nums[j]
        hole = j
        while i < hole && nums[i] < pivot
           i++
        if i == hole then exit loop
        nums[hole] = nums[i]
        hole = i
    nums[hole] = pivot
    return hole
```

Part E - Heap sort from week 11 (2%)

The `sortNums()` method (and any dependent methods) must be completed.

Remember that when using an array to represent a heap the children of item i have indexes $2 * i + 1$ and $2 * i + 2$. In order to sort the array using heap sort you first *heapify* the array by calling `siftDown()` on each index from $array.length/2 - 1$ back to the root index (0). After the array has become a heap you then sort it by swapping the root with the last item (shrinking the heap by 1) and *sift down* the new root to the correct place. The `siftDown()` method just swaps an item with the largest child that is bigger than it (if such a child exists) and then calls `siftDown()` on the new index to continue sifting downwards until it finds its correct place.

The skeletion `HeapSort.java` that we provide you with contains a `swap()` method that will swap two values in the `nums[]` array referencing i and j and updating the GUI. To avoid problems you shouldn't use i and j which are declared in `Sorter.java` anywhere else in your heap sort class.

Marking

You must complete your programs and get them marked by a demonstrator before the end of your time slot. All of your programs should be in the **`week12`** package. You can check that your programs are working using *241-check* as usual. You can run *241-check* (but not *241-submit*) in the lab beforehand, as well as during the test. **Note that comments are not required**. Also, no style checks are performed (although it's still a good idea to keep your code tidy). Each part of the test is worth 2%. No marks are given for partially completed programs. If you are unable to successfully complete all of the tasks within the allotted time you will be given the opportunity to do the test again on the final Tuesday of the semester.

If you just want to check one part of the test using the *241-check* script you can do so like this

```
PT=A 241-check
```

to check part A, B, C, D, or E (make sure there are no spaces around the = sign).

If you have any questions about this practical test, or the way it will be assessed, please see Iain or send an email to ihewson@cs.otago.ac.nz.