# Test 1: Compsci 100

Owen Astrachan

September 30, 2009

Name: _____ (1 point)

NetID/Login: _____

Honor code acknowledgment (signature) _____

| | value | grade |
|---|---|---|
| Problem 1 | 26 pts. | |
| Problem 2 | 20 pts. | |
| Problem 3 | 15 pts. | |
| Problem 4 | 16 pts. | |
| Problem 5 | 12 pts. | |
| TOTAL: | 90 pts. | |

This test has 17 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes. The last page is blank, if you use it make a note for the problem.

In writing code you do not need to worry about specifying the proper `import statements`. Don't worry about getting method names exactly right. Assume that all libraries and packages we've discussed are imported in any code you write.

**PROBLEM 1 : (*Complex or Imaginary (26 points)*)**

Consider the method `calc` below in the class `Test1`. When the `main` method is executed the output is

```
127
511
511
1023
```

You'll be asked several questions about this method and other methods below.

```
public class Test1 {
   public int calc(int n){
       int sum = 0;
       int val = 1;
       while (val <= n){
           sum += val;
           val *= 2;
       }
       return sum;
   }
   public static void main(String[] args){
       Test1 t = new Test1();
       System.out.printf("%d\n",t.calc(100));
       System.out.printf("%d\n",t.calc(256));
       System.out.printf("%d\n",t.calc(511));
       System.out.printf("%d\n",t.calc(512));
   }
}
```

**Part A (6 points)**

What is the value returned by each of the calls `calc(31)`, `calc(32)`, and `calc(33)`.

**Part B (3 points)**

What is the smallest value of `x` such that `calc(x)` returns 2047? Justify your answer briefly.

(continued)

**Part C (3 points)**

In terms of $n$, what is the exact value returned by the call `calc(2^n)`, justify your answer briefly.

**Part D (4 points)**

What is the runtime complexity of the call `calc(n)`, use big-Oh and justify your answer.

**Part E (4 points)**

What is the value of the expression `calc(calc(256))`? In terms of $n$, what value is the expression `calc(calc(2^n))` – base your answer to the latter question on what you wrote for Part C?

(continued, sort of)

**Part F (6 points)**

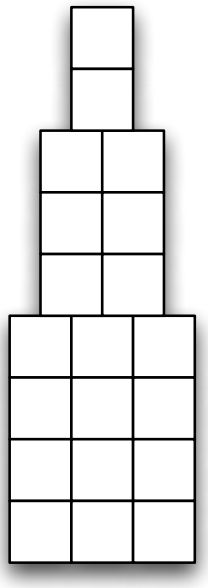
Complete method `bigKey` that returns the key in parameter `map` whose associated value is the largest.

```
public String bigKey(Map<String,Integer> map){

}
```

**PROBLEM 2 :** (***Tower of Power (20 points)***)

The picture below shows a *4-tower*. The bottom rectangle is $4 \times 3$ and this is topped by a $3 \times 2$ rectangle which is topped by a $2 \times 1$ rectangle. In general, for an *n-tower* there will be $n - 1$ rectangles with the bottom rectangle being $n \times (n - 1)$ and the top rectangle being $2 \times 1$.



**Part A (3 points)**
The *4-tower* is 9 units high. How high is a *6-tower*? Justify your answer briefly.

**Part B (3 points)**
How high high is a *100-tower*? Justify briefly.

**Part C (3 points)**
A *4-tower* is made of $12 + 6 + 2 = 20$ blocks. How many blocks are there in a *6-tower*?

**Part D (3 points)**
By definition we define `B(n)` to be the number of blocks in an *n-tower* so that `B(4) = 20` and `B(3) = 8`. Explain why the following relationship/equation is true

$$B(n) = B(n-1) + (n \times (n-1))$$

**Part E (5 points)**

Write a method `blocks` so that `blocks(n)` returns the number of blocks in an *n-tower*, e.g., so that `blocks(4)` returns 20 and `blocks(3)` returns 8. To earn 5/5 points you must write the method recursively. You can earn 4/5 points for writing a correct iterative method.

```
public int blocks(int n){









}
```

**Part F (3 points)**

Using big-Oh, how many blocks are there in an *n-tower*? Justify your answer briefly. The correct answer is either $n$, $n^2$, $n^3$, or $n^4$.

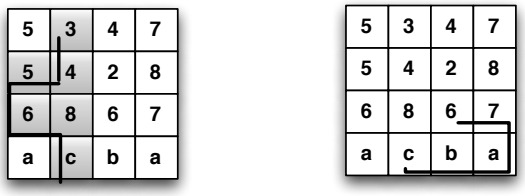
**PROBLEM 3 : (*Hillclimb (15 points)*)**

In this problem you'll finish code designed to find the longest hill-climbing grid-path in which each step in a path goes "higher" to an adjacent location in the grid.

For example, in the grids below two longest hill-climbing grid-paths are shown from different starting locations: on the left starting at (0,1) and on the right starting at (2,2) in the grid. Each grid point is labeled with an elevation: either 0-9 or `'a'-'z'`, with `'a'` corresponding to 10, `'b'`=11, `'c'`=12, ... `'z'`=35. Each step in a valid hill-climbing grid-path must increase in elevation and must be either up, down, left, or right from the current grid location.

Note that going right from (0,1) yields a grid-path of length 4 and going left from (0,1) yields a grid-path of length 2, but going down ultimately yields a grid-path of length 6. There is more than one grid-path of length six starting at (0,1) since the path could travel through the values (3,4,5,6,a,c) instead of (3,4,5,6,8,c).

Starting at (2,2) and going left yields a grid-path of length 3 as would going down from (2,2). But going right from (2,2) yields a hill-climbing grid-path of length 5.




**Part A (3 points)**

In a $4 \times 4$ grid, it is possible to have a grid-path of length 16 starting from (0,0) the upper left. Why?

**Part B (3 points)**

In a $7 \times 7$ grid it's not possible to have a grid-path of length 49. What's the longest hill-climbing grid-path in a $7 \times 7$ grid and why?

**Part C (3 points)**

In a $n \times n$ grid explain how to put values into the grid so that the longest path from any grid point has length 1, i.e. there is no path of length 2 starting from any grid point. (Every path starting at a point includes at least the point so has minimal length 1).

**Part C (6 points)**

Complete the code below by adding a few lines in the place indicated at the end of `find`. The goal is to make `longestPath` return the length of the longest hill-climbing grid-path in its parameter. The first part of `longestPat` simply sets up the grid with int values corresponding to the parameter `data`.

In the code you write in `find`, you'll need to determine which of the recursively-defined values is the largest, and use this to return the proper value. You may find `Math.max(x,y)` useful, the method returns the larger of its two parameters.

```
public class HillClimb {
    int[][] grid;
    public int longestPath(String[] data, int r, int c){
        grid = new int[data.length][data[0].length()];
        for(int j=0; j < grid.length; j++){
            for(int k=0; k < grid[0].length; k++){
                char ch = data[j].charAt(k);
                if (Character.isDigit(ch)) grid[j][k] = ch - '0';
                else                       grid[j][k] = ch - 'a' + 10;
            }
        }
        return find(r,c,-1);
    }
    /**
      * Returns length of longest hill-climbing path starting from (r,c)
      * in grid where the step onto (r,c) extends a path from a grid point whose
      * elevation is given by parameter lastValue
      */
    private int find(int r, int c, int lastValue){

        // if we step off the grid there is no path
        if (r < 0 || c < 0 || r >= grid.length || c >= grid[0].length) return 0;

        // if this point is lower than last there's no path that includes this point
        if (grid[r][c] <= lastValue) return 0;

        int current = grid[r][c];
        int left  = find(r,c-1,current);
        int right = find(r,c+1,current);
        int up    = find(r-1,c,current);
        int down  = find(r+1,c,current);
        // return value with calculations here









    }
}
```

**PROBLEM 4 : (*Big Courses (16 points)*)**

In this problem you'll write code to find information in data about enrollment in courses at Duke. The format of the data is as follows where each String represents one student (shown by first name) and the courses that student is taking — separated from the name by a colon `':'` and with each course separated by a comma `','` from other courses.

```
String[] classes = {
       "owen:math104,compsci100,econ55,soc17",
       "fred:compsci100,econ55,evanth120,eos11",
       "mary:eos11,compsci82,mus74,evanth120",
       "nancy:math104,phy22,chem155,compsci82",
       "fran:chem155,evanth120,soc17,psych11",
       "george:psych11,evanth120,chem155,pubpol21"
};
```

As an example of how to get information from this data, the method `enrollment` below returns the number of students enrolled in a specific course specified by parameter `course`. This output is generated by the code that follows using array `classes` as above.

```
2     compsci100
4     evanth120
2     eos11
3     chem155
0     art57
```

Here's the code generating this output. You'll be asked questions about this code and you'll be asked to write more code manipulating this data.

```
import java.util.*;
public class ACES {

    public int enrollment(String[] list, String course){
        int count = 0;
        for(String s : list) {
            String[] first = s.split(":");
            String[] courses = first[1].split(",");
            if (Arrays.asList(courses).indexOf(course) >= 0){
                count++;
            }
        }
        return count;
    }

    public static void main(String[] args){
        String[] classes = /* not shown, see above */
        ACES m = new ACES();
        for(String s : new String[]{"compsci100","evanth120", "eos11", "chem155","art57"}){
            System.out.printf("%d\t%s\n",m.enrollment(classes, s),s);
        }
    }
```

**Part A (4 points)**

Briefly explain in words the purpose of both calls to `split` and the purpose of the call to `indexOf` in the implementation of `enrollment` that's shown.

**Part B (4 points)**

Describe in words how to write code to print a list of courses sorted by enrollment, with the course with the most students printed first and with ties (equal enrollment) broken by printing course names alphabetically. For example, for the data above the output could be partially shown as:

```
evanth120, chem155, compsci100, ..., psych11, soc17, mus74, phy22, pubpol21
```

Don't write code here (see Part C), but write about how your code would work. Be precise enough to make it clear what your method is, but don't go into too much detail.

**Part C (8 points)**

Write the method `enrollSort` whose parameter is a list of student data and which returns an array of Strings. The array is course names, sorted by enrollment with the highest enrollment first and ties broken alphabetically as described above. You can write inner-classes and other methods, be sure to show them.

```
import java.util.*;
public class ACES {

   public String[] enrollSort(String[] list){

   }
}
```

**PROBLEM 5 : (*Over hill, Over dale, Overlap (12 points)*)**

In this problem there is a lengthy explanation of the problem followed by three questions. In this problem you're modeling requests for one classroom; the requests are specified by the start-time and end-time for each class meeting in the room, e.g., `(1,3)` means a two-hour class starting at one o'clock and `(2,5)` means a three-hour class starting at two o'clock (ending at five o'clock). In this problem all times are after noon and before midnight. Class requests are considered *intervals* with a start time and an end time.

We want to keep rooms occupied and busy, but busy is the first priority. This means we want to schedule as many classes/requests as possible in the room. Consider the requests below.

```
(1,2) (1,3) (3,4), (1,4), (4,5), (3,5), (2,5), (6,7)
```

We could schedule the classes as follows. In both scenarios three classes are scheduled and the room is idle for one hour.

```
(1,2)  (2,5) (6,7)
```

or we could schedule these classes:

```
(1,3)  (3,5) (6,7)
```

However, if we schedule the following classes we get four classes using the room although there are two idle hours.

```
(1,2) (3,4) (4,5) (6,7)
```

It's not possible to schedule more than four classes given the requests shown (though other schedules with four classes are possible too.)

You'll write code to manipulate the interval requests. You'll also write code to produce the list of class requests that keeps the room most busy, i.e., that schedules the most classes in the room. To do this you'll implement the following algorithm in Java.

1. Sort the intervals by end time, breaking ties by the length of the interval (longer intervals are larger). The lowest end time comes first in the sorted array — see the sorted list above that starts with (1,2) and ends with (6,7) for an example.
2. Schedule the first (least) interval by adding it to an `ArrayList` being built that will contain the answer.
3. Now consider each of the remaining intervals in sorted order, call the interval being considered the *current* interval.
   (a) If the current interval overlaps with the last interval added to the `ArrayList`, skip it.
   (b) Otherwise, if the current interval doesn't overlap, add the current interval to the `ArrayList` (it's now the last-added interval).

The definition for the class `Interval` you'll use in this problem is provided on the next page. You'll implement several methods using this class.

```
private static class Interval implements Comparable<Interval>{

    private int myStart;
    private int myEnd;

    public Interval(int start, int end){
        myStart = start;
        myEnd = end;
    }
    /**
     * Returns value < 0 if this interval less than o,
     * returns value > 0 if this interval greater than o,
     * and returns 0 if this interval is equal to o.
     */
    public int compareTo(Interval o) {
        int diff = myEnd - o.myEnd;
        // complete code here in Part C
    }

    /**
     * Return time from the end of this interval to the beginning
     * of Interval i. If this interval doesn't overlap with i
     * then return the time between when this ends and when i starts.
     * @param i is the other interval
     * @return the time until interval i
     */
    public int timeUntil(Interval i){
        return i.myStart - myEnd;
    }

    /**
     * Returns true if and only if this interval overlaps Interval i.
     * @param i is the interval considered for overlap with this one
     * @return true if and only if the intervals overlap
     */
    public boolean overLaps(Interval i){
        if (i.myEnd <= myStart || myEnd <= i.myStart) return false;
        return true;
    }

    public String toString(){
        return "["+myStart+","+myEnd+"]";
    }

}
```

**Part A (4 points)**

Write the method `idleTime` which returns the total time between all intervals in a sorted, non-overlapping array of intervals. For example, consider these arrays:

```
{  (1,3)  (3,5) (6,7)  }
{  (1,2) (3,4) (4,5) (6,7) }
```

The first array has one hour of idle time between (3,5) and (6,7). The second array has two hours of idle time.

```
/**
  * Assuming list is sorted, and intervals don't overlap
  * returns the number of hours not in any interval in list,
  * only hours between the start of list[0] and the end of
  * list[list.length-1]end are considered.
  * @param list is the sorted list of non-overlapping intervals
  * @return the number of idle time, time not in any interval
  */
public int idleTime(Interval[] list){









}
```

**Part B (4 points)**

Write method `mostBusy` which returns an array of intervals which contains the maximal number of non-overlapping intervals from the parameter `list`. Assume the `Interval` class `compareTo` method is implemented correctly (so the sort call below works). Use the algorithm at the beginning of this problem to create a maximal array of non-overlapping intervals. The `ArrayList` that is part of the algorithmic description is initialized in the starter-code provided, and the `ArrayList` is converted to an array in the return statement. Add code as necessary to implement the algorithm.

```
/**
 * Returns array containing maximal number of non-overlapping from list
 * @param list is an array of intervals
 * @return array containing maximal number of non-overlapping intervals
 */
public Interval[] mostBusy(Interval[] list){
    ArrayList<Interval> maximal = new ArrayList<Interval>();
    Arrays.sort(list);









    return maximal.toArray(new Interval[0]);
}
```

**Part C (4 points)**

Implement the `compareTo` method in the class `Interval` so that intervals are compared as specified earlier in this problem: an Interval $a$ is less than an interval $b$ if and only if $a$ ends before $b$ ends or they end at the same time and the length of interval $a$ is less than the length of interval $b$.

```
private static class Interval implements Comparable<Interval>{

    private int myStart;
    private int myEnd;

    /**
     * Returns value < 0 if this interval less than o,
     * returns value > 0 if this interval greater than o,
     * and returns 0 if this interval is equal to o.
     */
    public int compareTo(Interval o) {
        int diff = myEnd - o.myEnd;









}
```

(nothing on this page)