CS 116 – Winter 2015

Assignment 07

Due at noon on Wednesday, March 18

Assignment Guidelines:

- **This assignment consists mostly of material as covered in Module 07. All solutions must be in Python.**
- **You may use structural and generative recursion in your solutions. Do not use loops (from Module 08).**
- Your solutions should be placed in files **a07qY.py**, where Y is a value from 1 to 4.
- Download the testing module from the course web page. Include `import check` in each solution file.
  - When a function produces a floating point value, you must use `check.within` for your testing. Unless told otherwise, you may use a tolerance of 0.0001 in your tests. Note the following new restrictions:
- Do not import any other modules other than `math` and `check`.
- Do not define helper functions locally.
- Examples and tests are not required for any helper functions.
- Do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post (#19) on Piazza. You are always allowed to use define your own helper functions, as long as they meet the assignment restrictions.
- You may use global constants in your solutions.
- Do not use global variables for anything other than testing.
- Download the interface file from the course Web page to ensure that all function and structure names are spelled correctly and each function has the correct number and order of parameters. Use the function headers and full structure definitions in your submitted files for each question.
- For full style marks, your program must follow the **Python section** of the CS116 Style Guide. In particular,
  - Be sure to include all the steps of the design recipe for all required functions: including purpose statements, effects, contracts (including requirements), examples (note the new style), and tests.
  - You are not required to submit any templates with your solutions.
  - The purpose should be written in your own words and must include meaningful use of the function's parameter names.
  - There will be marks assigned for the appropriate use of constants, helper functions, choice of meaningful names, and appropriate use of whitespace.
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
- Do not send any code files by email to your instructors or ISAs. It will not be accepted by course staff as an assignment submission, even if you are having issues with MarkUs. Course staff will not debug code emailed to them.
- Test data for all questions will always meet the stated assumptions for consumed values.
- Read each question carefully for specific restrictions.
- No late assignments will be accepted.
- Check MarkUs and your basic test results to ensure that your files were properly submitted. In particular:
- A misnamed file or function will receive no marks.
- Do not copy any text from the interactions window in WingIDE or from this pdf into your programs (even as comments). It will make your submitted file unreadable in MarkUs and you will receive no marks (correctness or style) for that question.
- Be sure to review the Academic Integrity policy on the Assignments page.

Language: Python 3
Coverage: Module 07

Assignment 07

Due at noon on Wednesday, March 18

1. In class (and in Module 07 slides) we covered a quadratic (O(n^2)) implementation of the function `singles`. Recall that this function consumes a list `lst` and produces a list containing only the unique elements of `lst`.

   For example: `singles([ 4, 1, 4, 17, 1]) => [4, 1, 17]`

   Write an implementation of `singles` with the following modification: the function should consume *a list of numbers* `lst`, and the produced list should contain the unique elements of `lst` **in increasing order**. Your implementation should have a running time of O($n \log n$).

   For example: `singles([4, 1, 4, 17, 1])` => `[1, 4, 17]`

   For this question you can use built-in functions `sort` and `sorted` that have a guaranteed O($n \log n$) runtime.

2. **Divide and conquer** refers to the technique that was used in our implementation of `Mergesort`: breaking down the problem into two subproblems, and then combining solutions to these subproblems to solve the original problem.

   Use the **divide-and-conquer** approach to write a Python function `smallest_diff`. This function consumes a sorted list of integers (in non-decreasing order), `numbers` and produces the smallest difference between any two adjacent elements in the list. You can assume that `len(numbers) >= 2`. Your function should divide the list as evenly as possible.

   For example: `smallest_diff([5, 500, 505, 600, 650, 10000]) => 5`

3. We will call a string `s` *repetitively alternating* if there is such a string `w` and its reverse `w`^{rev}, such that `s` can be constructed by alternately concatenating `w` and `w`^{rev}, i.e.:

   `s = w + w`^{rev} `+ w + ... + w`^{rev}, or
   `s = w + w`^{rev} `+ w + ... + w`^{rev} `+ w,`
   and `len(s) > len(w) >= 2.`

   For example: strings "`abbaabba`" and "`bobbob`" are repetitively alternating, and strings "`I love recursion!`" and "`ababba`" are not.

   Write a Python function `alternating` that consumes a string `s` and produces `True` if `s` is repetitively alternating, and `False` otherwise. One possible approach to solving this problem is to check if any of `s`'s prefixes fit the definition of `w` as described above.

   Note: you can easily compute `w`^{rev} by using the extended slicing syntax. You can read about it here: https://docs.python.org/2.3/whatsnew/section-slices.html

4. Write a Python function `ith_in_sorted` that consumes a non-empty list of distinct integers `lst` and an index `i` `(0 <= i < len(lst))`, and produces the integer at position `i` in the `lst` sorted in increasing order. The implementation of this function would be trivial if you sorted the list. However sorting can be fairly expensive, and we will attempt to solve this problem without sorting by using the following algorithm:

   a. The first number in `lst` will become a special number, *the separator*.

b. Put all the numbers in `lst` that are smaller than the separator into a list called `smaller`
c. Put all the numbers in `lst` that are greater than the separator into a list called `larger`.
d. If `len(smaller) == i`, then the separator would be at position `i` in the sorted list, and it is the answer to the problem.
e. If `len(smaller) > i`, then the number we are looking for must be in `smaller`, and it would have to be at position `i` if we were to sort `smaller` (this can be solved recursively!).
f. If `len(smaller) < i`, then the number we are looking for must be in `larger`, but what would its position be in `larger` if `larger` was sorted? Recall that the separator and the numbers from `smaller` come before all the numbers in `larger`, and there are `len(smaller) + 1` of them. This means that the number we are looking for will be at position `(i – len(smaller) – 1)` in `larger` (this can be solved recursively!).

Examples:

```
    ith_in_sorted([17, -5, 3, 0, 2, 100], 2) => 2
    ith_in_sorted([300, 200, 100], 0) => 100
```