Practice Questions for Final Examination

CSCI-141 Computer Science 1

Fall 2013

1. **(Functions)** Turtles are capable of drawing equilateral polygons.

   - Draw a straight line 4 times, turning 90 degrees between each line to create a square.
   - Draw a straight line 6 times, turning 60 degrees between each line to create a hexagon.
   - Turtles draw circles by drawing very short lines and turning a very small amount each time.

   Ignoring the built-in `circle` function, write the code for your own `drawCircle` function to have a turtle draw a circle for a given radius by using line segments.

2. **(Recursion)** A number a is a power of b if it is divisible by b and a/b is a power of b. Write a function, `isPower` that takes parameters a and b and returns `True` if a is a power of b and `False` otherwise. Assume it is a precondition of this function that a and b are both greater than 0.

   (a) *Write* the function using *recursion*.

   (b) Show a *substitution trace* for `isPower(27, 3)` and `isPower(40, 4)`.

   (c) *Write* the function again *iteratively* using a loop.

3. **(Strings)** An *anagram* is a word formed by rearranging the letters of another word. For example, `angel` and `glean` are anagrams of each other. Design a function `anagram` that takes two `strings` and returns a `boolean` indicating whether or not the strings are anagrams of each other.

   (a) **Strategy:** Write an outline of your approach.

   (b) **Code:** Write your Python code.

   (c) **Testing:** Provide four test cases, using specific values of input parameters, and for each test case state the expected output.

4. **(Sorting)** What is the big-O time complexity for *insertion sort*? Provide two sample data sets of eight (8) values each that illustrate insertion sort running in *best* and *worst* case times.

5. **(Searching)** What are the logic errors in the following `binarySearch` function?

```
def binary_search(data, target, start, end):
    if start >= end:
        return -1

    mid_index = (start + end) // 2
    mid_value = data[mid_index]

    if target == mid_value:
        return mid_index
    elif target < mid_value:
        return binary_search(data, target, start, mid_index-1)
    else:
        return binary_search(data, target, start+1, end)
```

6. **(Greedy Algorithms, Lists, Files, Classes, Loops)** You are a student who wants to take as many courses as possible without taking any course that overlaps with another. Assume the courses are in a file that contains one course per line. The information for each course is the **course name**, the **start time** and the **stop time** (using 24-hour time).

   There are many ways we could approach solving this problem in a greedy manner, but only one is always optimal (chooses the most number of non-overlapping courses):

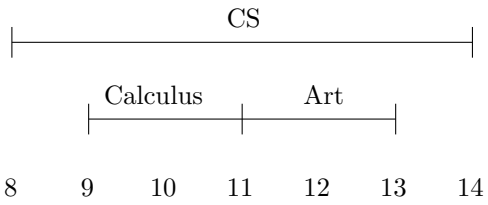
   (a) **Earliest Start Time** - Consider courses in ascending order based on *start time*. Take the first course and add it to your schedule, then "cross out" any other course that conflicts with it. Repeat this until there are no courses left.

COURSES:

Art 11 13

CS 8 14

Calculus 9 11



RESULT:

CS (8-14)

OPTIMAL:

Calculus (9-11)

Art (11-13)

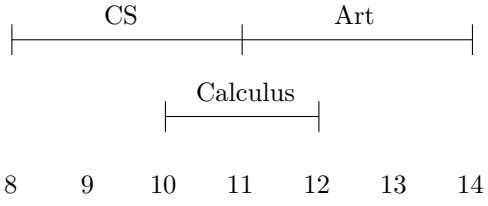
   (b) **Shortest Interval** - Consider courses in ascending order based on *length*. Take the first course and add it to your schedule, then "cross out" any other course that conflicts with it. Repeat this until there are no courses left.

COURSES:

Art 11 14

CS 8 11

Calculus 10 12



RESULT:

Calculus (10-12)

OPTIMAL:

CS (8-11)

Art (11-14)

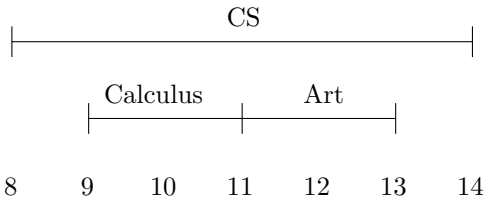
(c) **Earliest Finish Time** - Consider courses in ascending order based on *finish time*. Take the first course and add it to your schedule, then "cross out" any other course that conflicts with it. Repeat this until there are no courses left.

COURSES:

Art 11 13

CS 8 14

Calculus 9 11



RESULT:

Calculus (9-11)

Art (11-13)

OPTIMAL:

Calculus (9-11)

Art (11-13)

It turns out that choosing courses based on **earliest finish time** is always optimal. **For this problem we guarantee the file will be ordered based on finish time.**

(a) Design a class, `Course`, that can be used to represent a course.

(b) Write a function, `mkCourse`, that can build a course object.

(c) Write a function, `readCourses`, which takes the course file as an argument and returns a list of `Course` objects.

(d) Write a function, `greedySchedule` , which take a list of `Course` objects and returns the schedule as a list of `Course` objects using the optimal greedy algorithm.

7. **(Lists)** A *permutation* of a string `s` is a string `s2` such that `s2` is a rearrangement of the letters of `s`. Note that the trivial rearrangement that leaves `s` alone counts as a permutation. Given this notion, we can talk about the collection of all distinct permutations of a string `s`. For example, for `'123'` we have `'123'`, `'132'`, `'213'`, `'231'`, `'312'`, `'321'`.

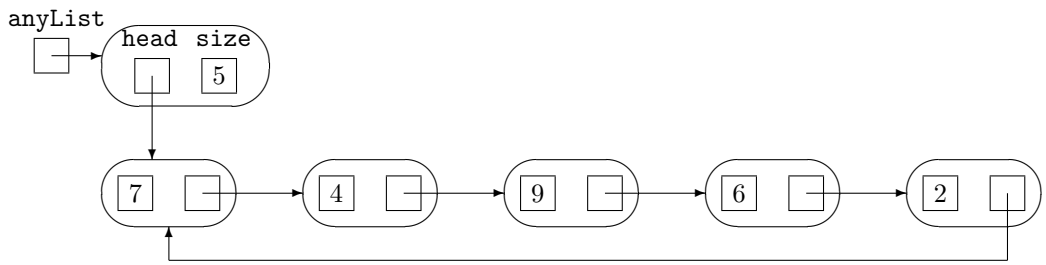
   In part (c), you will write a function to compute the list of all permutations of a string. But first in parts (a) and (b), you will write helper functions.

   (a) Write the function `insertEverywhere`, that takes a character `c` and a string `s`, and returns a list of strings with `c` inserted into `s` in every possible position. For example, `insertEverywhere('a', '123')` should return `['a123', '1a23', '12a3', '123a']`.

   (b) Write the function `insertEverywhereAll`, that takes a character `c` and a list of strings `L`, and returns a list of strings that is constructed by concatenating the results of calls to `insertEverywhere` on each of the elements of `L`. For example, `insertEverywhereAll('a', ['12', '34'])` should return `['a12', '1a2', '12a', 'a34', '3a4', '34a']`.

   (c) Using `insertEverywhereAll`, write the function `perms` that takes a string `s` and returns the list of all distinct permutations of `s`.

8. **(Linked Lists)** This problem deals with implementations of linked lists.



(a) Provide code for a function called `maxNode(anyList)` that returns the largest value in a circularly linked list whose first node is referenced by `anyList.head`. For example, in the above drawing, `maxNode(anyList)` returns 9.

(b) What is the time complexity of the `maxNode` operation in terms of list length, N?

(c) Suggest three tests for `maxNode` in terms of the values of the list.

(d) Write a function called `append(anyList, value)` that inserts a new value at the end of a circularly linked list whose first node is referenced by `anyList.head`. As shown above, a circularly linked list has its last node linked back to the head node of the list. This is a singly linked list, and the Node class has two slots, `data` and `next`.

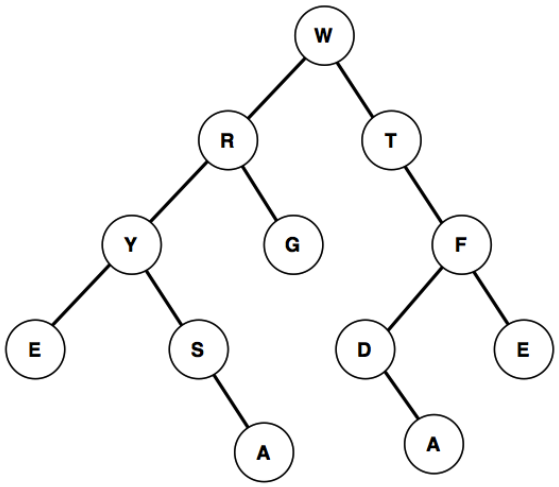
(e) What is the time complexity of `append` in terms of the list's length, N?

(f) Identify three tests for `append` in terms of the size of the list (before the append operation). The value appended is unimportant since it does not affect the behavior of the algorithm.

9. **(Trees)** Show the *binary search tree* that results from inserting these elements in the following order to an initially empty tree:

   `6, 8, 4, 5, 2, 1, 3, 7, 9, 10`.

10. **(Trees)** Given the following binary tree, indicate whether each sequence of letters is `INORDER` (left, parent, right), `PREORDER` (parent, left, right), `POSTORDER` (left, right, parent), or `NONE`.



(a) E,Y,S,A,R,G,D,A,E,F,T,W:

(b) W,R,Y,E,S,A,G,T,F,D,A,E:

(c) W,R,T,Y,G,F,E,S,D,E,A,A:

(d) E,Y,S,A,R,G,W,T,F,D,A,E:

(e) A,A,E,D,S,E,F,G,Y,R,T,W:

(f) W,Y,A,S,G,R,T,A,D,F,E,E:

(g) E,A,A,S,Y,R,G,E,D,W,F,T:

(h) E,A,S,Y,G,R,A,D,E,F,T,W:

11. **(Stacks/Queues)** Mark each operation in the set below as acceptable list operations to implement a queue, a stack, or neither.

(a) insert value at front and remove value from front
(b) insert value at rear and remove value from rear
(c) insert value at rear and remove value from front
(d) insert value at front and remove value from rear

12. **(Optimal Sorting)** For each of the sorting algorithms below, state the best-case and worst-case running time complexity (e.g., $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2n)$):

(a) Merge Sort
(b) Quick Sort
(c) Heap Sort

Of all the sorting algorithms you know, which works best if the data is already sorted? What is its big-O running time for that case?

13. **(Backtracking)** There is a problem called "Subset Sum" that asks the question, Given a "capacity" and a set of items, each with a "weight", is there a subset of the items whose weights add up exactly to the capacity? Consider solving this through backtracking. See lecture notes for the generic algorithm.

    (a) What is a valid (i.e., legal for this problem) configuration and what constitutes a single step to get from one configuration to the next?

    (b) What does a goal configuration look like?

    (c) What does a failure configuration look like?

14. **(Hashing)**

    (a) What is the expected time complexity of determining whether or not a key is in a hash-table?

    (b) What is the worst-case time complexity?

15. **(Heapsort)** Max-heapify, then sort, in the style of the heap sort algorithm, the following sequence of numbers. Show the contents of the array at each step.

$$[19, 8, 6, 20, 40]$$

16. **(Hashing)** Assume we have a hash function for strings that returns the (ordinal value of the first character in the string) % (size of the table), where ord(a) = 0 and ord(z) = 25. Draw a hash table (using chaining) of size 10 where the keys are strings and a key's value is the number of occurrences of the string. Process the following strings and insert them into the table in the order given.

    `'we','eat','ham','and','jam','and','spam','a','lot','in','camelot'`

17. **(Optimal Sorting)** Sort the following data using mergesort and quicksort. For quicksort, always choose the first value as the pivot. Show how the contents of the array changes at each step.

$$[6, 7, 8, 9, 5, 4, 3, 2, 1]$$

18. **(Backtracking)**

    (a) The following code attempts to employ backtracking to seek a solution to a problem. Part of the code is missing (the part indicated with `###`). Fill in this part of the code to complete the backtracking algorithm.

```
def solve(configuration):
    """solve: Config -> Config or None"""
    if isGoal(configuration):
        return configuration
    else:
        for child in successors(configuration):
            if isValid(child):
                solution = solve(child)
                ###
        return None
```

    (b) Underline the parts of the code that must be defined for the specific problem at hand (hint: there are 4 parts). All but one of these parts is usually defined as a separate function — for each such part, assuming that it is replaced with a function call, what data type (in general) would be sent to the function, and what data type should be returned?

19. **(Open Problem)** A grocery store is divided into n aisles labeled $0 \ldots n-1$. Each aisle contains a distinct set of grocery items. This information is given in a grocery store tuple named `inventory`, where each element is an `Item` object containing the string name of a grocery item and its aisle number. (An Item contains slots named "grocery" and "aisle".) For example:

```
(Item('bread',0), Item('muffins',0), Item('coldcuts',5), Item('toothpaste',1),
Item('eyeliner',1), Item('soda',2), Item('bottledwater',2), Item('cereal',3),
Item('milk',4), Item('eggs',4), Item('butter',4), Item('meat',5), Item('fish',5))
```

A list called `groceryList` provides a list of the names of all items someone wants to purchase. For example, `('muffins','butter','soda')`.

Design a function called `organize` that takes `inventory` and `groceryList` and creates a list of sublists, where the sublist at position i in the list contains the names of all the groceries in the grocery list that can be found in aisle i.

For the examples above, `organize(inventory,groceryList)` would return
`(('muffins'),(),('soda'),(),('butter'),())`

(a) Describe the data structures you will use internally to organize the information.

(b) Design the algorithm the function will run and describe it in pseudocode.

(c) Provide a time complexity analysis of your algorithm given an inventory tuple of length n and a grocery list of length g. If you feel you need to, state any assumptions you are making about the performance of any underlying data structures you're using.

(d) Describe test data you would use to ensure the program runs were you to write the code.