CSC 321: Data Structures

Fall 2016

See online syllabus (also available through BlueLine):
http://dave-reed.com/csc321

Course goals:

- To understand fundamental data structures (lists, stacks, queues, sets, maps, and linked structures) and be able to implement software solutions to problems using these data structures.
- To achieve a working knowledge of various mathematical structures essential for the field of computer science, including graphs, trees, and networks.
- To develop analytical techniques for evaluating the efficiency of data structures and programs, including counting, asymptotics, and recurrence relations.
- To be able to design and implement a program to model a real-world system, selecting and implementing appropriate data structures.

221 vs. 222 vs. 321

221: intro to programming via scripting

- focused on the design & analysis of small scripts (in Python)
- introduced fundamental programming concepts
  - ✓ variables, assignments, expressions, I/O
  - ✓ control structures (if, if-else, while, for), lists
  - ✓ functions, parameters, intro to OO

222: object-oriented programming

- focused on the design & analysis of more complex programs (in Java)
- utilized OO approach & techniques for code reuse
  - ✓ classes, fields, methods, objects
  - ✓ interfaces, inheritance, polymorphism, object composition
  - ✓ searching & sorting, Big-Oh efficiency, recursion, GUIs

you should be familiar with these concepts (we will do some review next week, but you should review your own notes & text)

321: data-driven programming & analysis

- focus on problems that involve storing & manipulating large amounts of data
- focus on understanding/analyzing/selecting appropriate structures for problems
  - ✓ standard collections (lists, stacks, queues, trees, sets, maps)
  - ✓ mathematical structures (trees, graphs, networks)
  - ✓ analysis techniques (counting, asymptotics, recurrence relations)

When problems start to get complex…

…choosing the right algorithm and data structures are important

- e.g., phone book lookup, Sudoku solver, path finder

- must develop problem-solving approaches (e.g., brute force, backtracking)
- be able to identify appropriate data structures (e.g., lists, trees, sets, maps)

example: anagram finder

- you are given a large dictionary of 117,663 words
- repeatedly given a word, must find all anagrams of that word

  pale → leap pale peal plea

  steal→ least setal slate stale steal stela taels tales teals tesla

  banana → banana

- there are many choices to be made & many "reasonable" decisions
  - ✓ how do you determine if two words are anagrams?
  - ✓ should you store the dictionary words internally? if so, how?
  - ✓ should you preprocess the words? if so, how?
  - ✓ is a simplistic approach going to be efficient enough to handle 117K words?
  - ✓ how do you test your solution?

Possible implementations

1. generate every permutation of the letters, check to see if a word
   - how many permutations are there?
   - will this scale?

2. for each word, compare against every other word to see if an anagram
   - how costly to determine if two words are anagrams?
   - how many comparisons will be needed?
   - will this scale?

3. preprocess all words in the dictionary and index by their sorted form
   - e.g., store "least" and "steal" together, indexed by "aelst"
   - how much work is required to preprocess the entire dictionary?
   - how much easier is the task now?

HW1: Keypad verification

HW1 is posted

- part1 is to be completed in 2-person teams, due in 1 week
  we will meet to go over the code, go over holes in your knowledge/skills
- part2 is to be completed individually, builds on part1 code

both parts involve verifying codes (e.g., id numbers, security codes)

- assume that a database of valid codes has been stored
- we want to read in codes entered on a keypad, verify they are valid

HW1 part 1: verify codes

```
13579#
123*45*6789
1357#9
1024
0001010
```

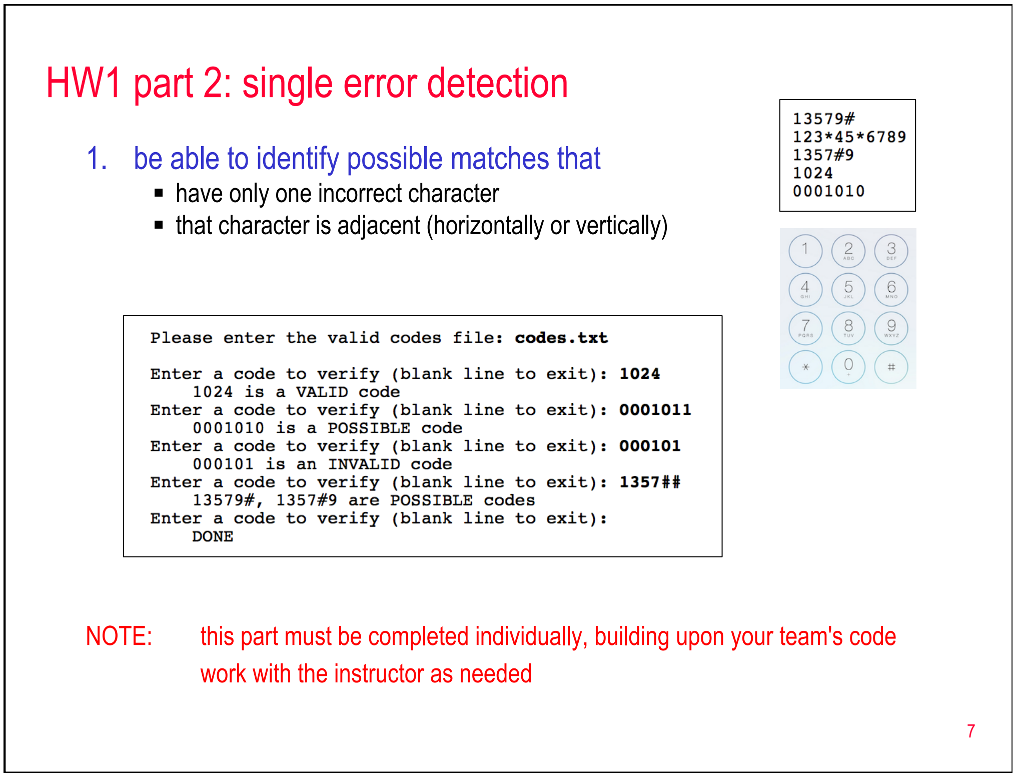
1. need to read (and store) the collection of valid codes

2. repeatedly read user-inputted codes and verify if valid
   - you may assume that codes are 1-30 characters, using only 0..9*#

```
Please enter the valid codes file: codes.txt

Enter a code to verify (blank line to exit): 1024
    1024 is a VALID code
Enter a code to verify (blank line to exit): 0001011
    0001011 is an INVALID code
Enter a code to verify (blank line to exit):
    DONE
```

ADVICE: work with your partner – you both should understand everything in your program
be introspective – identify holes, work with your partner, come see me A LOT



HW1 part 2: single error detection

1. be able to identify possible matches that
   - have only one incorrect character
   - that character is adjacent (horizontally or vertically)

```
13579#
123*45*6789
1357#9
1024
0001010
```

1
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
*
0 +
#

```
Please enter the valid codes file: codes.txt

Enter a code to verify (blank line to exit): 1024
    1024 is a VALID code
Enter a code to verify (blank line to exit): 0001011
    0001010 is a POSSIBLE code
Enter a code to verify (blank line to exit): 000101
    000101 is an INVALID code
Enter a code to verify (blank line to exit): 1357##
    13579#, 1357#9 are POSSIBLE codes
Enter a code to verify (blank line to exit):
    DONE
```

NOTE: this part must be completed individually, building upon your team's code
work with the instructor as needed