# CS 106B, Autumn 2015
# Midterm Exam

**Your Name:** ________________________________________________

**Section Leader:** ______________________

***Honor Code:*** *I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I am submitting are my own work. I agree not to talk about the exam contents to anyone until a solution key is posted by the instructor.*

**Signature:** ________________________ ← **YOU MUST SIGN HERE!**

**Rules:** *(same as posted previously to class web site)*

- This exam is to be completed by each student **individually**, with no assistance from other students.
- You have **2 hours** (120 minutes) to complete this exam.
- This test is **open-book**, but **closed notes**. You may not use any paper resources.
- You may not use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- On code-writing problems, you do not need to write a complete program, nor `#include` statements. Write only the code (function, etc.) specified in the problem statement.
- Unless otherwise specified, you can write **helper functions** to implement the required behavior. But when asked to write a function, do not declare any **global variables**.
- Do not abbreviate code, such as writing ditto marks ("") or dot-dot-dot marks (...).
- If you wrote your answer on a back page or attached paper, please **label this** clearly to the grader.
- Follow the Stanford **Honor Code** on this exam and correct/report anyone who does not do so.

*Good luck! You can do it!*

*Special thanks to colleagues such as Stuart Reges, Keith Schwarz, Mehran Sahami, Jerry Cain for problem ideas.*

| # | Description | Earned | Max | Grader (initial) |
|---|---|---|---|---|
| 1 | Parameters and Pointers, read | | 6 | |
| 2 | Collections, read | | 6 | |
| 3 | Collections, write | | 9 | |
| 4 | Big-Oh, read | | 6 | |
| 5 | Recursion, read | | 6 | |
| 6 | Recursion, write | | 9 | |
| 7 | Backtracking, write | | 9 | |
| 8 | Pointers and Linked Nodes, write | | 6 | |
| | **TOTAL** | | **57** | |

## 1. Parameters and Pointers (read)

The following code C++ uses parameters/returns and produces **three lines of output**. What is the output?

```
int parameterMystery9(int* b, int& c, int a) {
    (*b)++;
    c += 3;
    a += 5;
    cout << c << " " << a << " " << *b << endl;
    return a;
}

int main() {
    int a =   10;
    int b =  200;
    int c = 3000;

    parameterMystery9(&a, b, c);
    int d = parameterMystery9(&b, c, a);
    cout << a << " " << b << " " << c << " " << d << endl;

    return 0;
}
```

*Write the output below, exactly as it would appear on the console.*

## 2. Collections (read)

Write the **output** that would be produced by the following function when passed each of the following `Queue`s and `int`s. Write the output exactly as it would appear on the console.

*Note : A Stack displays/prints in {bottom ... top} order, and a Queue prints in {front ... back} order.*

```
void collectionMystery9(Queue<int>& queue, int p) {
    Stack<int> stack;
    int count = 0;
    int size = queue.size();
    for (int i = 0; i < size; i++) {
        int element = queue.dequeue();
        if (element < p) {
            queue.enqueue(element);
        } else {
            count++;
            stack.push(count);
            stack.push(element);
        }
    }

    while (!stack.isEmpty()) {
        queue.enqueue(stack.pop());
    }

    cout << queue << endl;
}
```

**a)** `queue = {1, 2, 3, 4, 5},  p = 4`

**b)** `queue = {67, 29, 115, 84, 33, 71, 90},  p = 50`

## 3. Collections and File I/O (write)

Write a function named **areaCodes** that prints information about the most commonly occurring area code in a file of telephone numbers. Your function accepts a string parameter representing a filename of input.

The input file contains a collection of telephone numbers, one per line, in the format of the example shown at right. Each phone number begins with a three-digit area code. You may assume that every line of the file is in exactly the format shown, without any other characters or formatting or spacing.

```
650-723-2273
206-685-2181
800-356-9377
800-347-3288
650-725-7411
520-297-6312
206-543-1695
800-266-2278
206-543-2969
```

Your function should open and read the contents of this input file and figure out which area code occurs most frequently in the data. If multiple area codes are **tied** for being the most frequent, print the numerically smallest of the ones that tied. For example, in the data at right, the most common area codes are 800 and 206, each of which has 3 phone numbers with that area code, so your function should consider 206 to be the most commonly occurring area code.

After determining which is the most common area code, your function should print out all of the phone numbers from that area code, in **sorted numerical order**, one per line. For example, if the input above at right comes from a file named `phonenumbers.txt`, then the call of `areaCodes("phonenumbers.txt");` should print the following output:

```
206-543-1695
206-543-2969
206-685-2181
```

If the file is missing or unreadable, your function should produce no output. If the file exists, you may assume that it contains at least one phone number, that every line of input in the file is in the exact valid format described above, and that every phone number in the file will be unique.

*Constraints:* For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- You may open and **read the file only once**. Do not re-open it or rewind the stream.
- You should choose an **efficient** solution. Choose data structures intelligently and use them properly.
- You may create **one collection** (stack, queue, set, map, etc.) or nested/compound structure as auxiliary storage. A nested structure, such as a set of vectors, counts as one collection. A temporary collection variable that is merely a replica or reference to some other collection (such as, `Vector<int> v = myMap.get(k);`) is fine and does not count as a second structure. (You can have as many simple variables as you like, such as `int`s or `string`s.)

*Write your answer on the next page.*

## 3. Collections and File I/O (write)
Writing Space

## 4. Big-Oh (read)

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of variable *N*. (In other words, the algorithm's runtime growth rate as *N* grows.)

This problem is **multiple-choice**. For each part below, **circle your answer** on the right side from the choices provided. Only one answer is correct for each problem. Make sure to make it clear which choice you are circling; if we cannot tell with certainty which answer you have circled, you will be marked incorrect.

**Question** | **Answer**

**i)**

```
int sum = 0;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        sum++;
    }
    for (int j = 1; j < N + 5; j++) {
        for (int k = 1; k < 99999; k++) {
            sum++;
        }
    }
}
cout << sum << endl;
```

| | |
|---|---|
| a) O(1) | b) O(log N) |
| c) O(N) | d) O(N log N) |
| e) O($N^2$) | f) O($N^2$ log N) |
| g) O($N^3$) | h) other |

**ii)**

```
Stack<int> stack;
Set<int> set;
for (int i = 0; i < N; i++) {
    stack.push(N);
    set.add(N);
}
while (!stack.isEmpty()) {
    set.remove(stack.pop());
}
cout << "done!" << endl;
```

| | |
|---|---|
| a) O(1) | b) O(log N) |
| c) O(N) | d) O(N log N) |
| e) O($N^2$) | f) O($N^2$ log N) |
| g) O($N^3$) | h) other |

**iii)**

```
Map<int, int> map;
for (int i = 0; i < N; i++) {
    for (int j = 4; j <= 2*N + 1; j++) {
        map.put(i, j);
    }
}
Queue<int> queue;
for (int k : map) {
    queue.enqueue(k);
}
cout << "done!" << endl;
```

| | |
|---|---|
| a) O(1) | b) O(log N) |
| c) O(N) | d) O(N log N) |
| e) O($N^2$) | f) O($N^2$ log N) |
| g) O($N^3$) | h) other |

**iv)**

```
Vector<int> vector;
HashSet<int> hashset;
for (int i = 4; i <= N + 7; i++) {
    hashset.add(i);
}
for (int num : hashset) {
    vector.add(num);
}
while (!vector.isEmpty()) {
    vector.remove(0);
}
cout << "done!" << endl;
```

| | |
|---|---|
| a) O(1) | b) O(log N) |
| c) O(N) | d) O(N log N) |
| e) O($N^2$) | f) O($N^2$ log N) |
| g) O($N^3$) | h) other |

## 5. Recursion (read)

For each call to the following recursive function, write the value that would be returned.

```
int recursionMystery9(int x, int y) {
    if (x < 0) {
        return -recursionMystery9(-x, y);
    } else if (y < 0) {
        return -recursionMystery9(x, -y);
    } else if (x == 0 && y == 0) {
        return 0;
    } else {
        return 100 * recursionMystery9(x / 10, y / 10)
             + 10 * (x % 10) + y % 10;
    }
}
```

| Call | Result |
|---|---|
| **a)** `recursionMystery9(12, 49);` | |
| **b)** `recursionMystery9(73, -8);` | |
| **c)** `recursionMystery9(-248, -3795);` | |

## 6. Recursion (write)

Write a recursive function named **`replaceAll`** that accepts three parameters: a string `s`, a char `from`, and a char `to` as parameters, and returns a new string that is the same as `s` but with any occurrences of `from` changed to `to`. For example, the call of `replaceAll("crazy raccoons", 'c', 'k')` should return `"krazy rakkoons"` and the call of `replaceAll("BANANA", 'A', 'O')` should return `"BONONO"`.

Your function is case-sensitive; if the character `from` is, for example, a lowercase `'f'`, your function should not replace uppercase `'F'` characters. In other words, you should not need to write code to handle case issues in this problem.

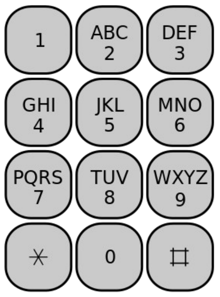
*Constraints:* For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- **Do not use any loops**; you must use recursion.
- Do not declare any **global variables**.
- **Do not call** any of the following string functions: `find`, `rfind`, `indexOf`, `contains`, `replace`, `split`.
  *(The point of this problem is to solve it recursively; do not use a library function to get around recursion.)*
- Do not use any auxiliary **data structures** like `Vector`, `Map`, `Set`, array, etc.
- You *can* declare as many primitive variables like `int`s as you like, as well as `string`s.
- You *are* allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

*Write your answer below.*

## 7. Backtracking (write)

Write a recursive function named **`phoneWords`** that uses backtracking to print all seven-letter words that correspond to the digits of a given phone number.



On a standard telephone keypad, the letters A-Z are mapped onto the phone number digits 0-9 as shown at right. A,B,C are mapped to 2, and D,E,F are mapped to 3, and so on. Many businesses choose their phone numbers so that they can be read and written as words to make them easier to remember. For example, 1-800-FLOWERS is really 1-800-356-9377. The idea of this problem is that you will be given a seven-digit telephone number as a string, such as "3569377", along with a dictionary of seven-letter words, and you are to find and print all words that could be made using the seven digits of that phone number in their original order.

Your function will accept three **parameters**: a seven-letter string representing the 7-digit phone number; a reference to a `Lexicon` representing a dictionary; and a reference to a `Map` from integers to strings, representing the phone number keypad mapping shown above at right. The map's contents are always `{2:"ABC", 3:"DEF", ..., 9:"WXYZ"}`. (This map is passed for convenience so that you don't need to write all of the code to do the number to character mapping.)

For example, if a standard English dictionary of 7-letter words is loaded into a Lexicon named `dict`, and the phone number mapping is called `numberMap`, then the call of `phoneWords("3569377", dict, numberMap);` should print:

```
FLOWERS
```

It turns out that `"FLOWERS"` is the only 7-letter word that corresponds to `"3569377"`, but some phone numbers correspond to many words. If this is the case, print all such words, one per line, in alphabetical order. For example, the call of `phoneWords("7874464", dict, numberMap);` should print:

```
PURGING
PUSHING
RUSHING
SURGING
```

If no dictionary words correspond to the given phone number, you should not produce any output.

*Assumptions:* You may assume that the phone number passed will be exactly 7 characters in length, with no hyphens, area code, or other formatting in it. All of the characters will be digits from `'0'` through `'9'`. You may also assume that every word found in the dictionary is exactly 7 letters long, and that its words consist entirely of letters A-Z in uppercase.

You may find yourself needing to convert between numeric characters like `'3'` and their integer equivalents like `3`. Recall that you can do this by casting between `int` and `char` and then adding or subtracting the integer ASCII value of `'0'`.

```
char c = '3';
int x = (int) c - '0';   // 3
```

*Efficiency:* For full credit, your code should avoid large areas of exploration that cannot lead to any solution. Specifically, if you are exploring the phone number `"3569377"` from our first example, and you have chosen to map the first three numbers `"356"` to the letters `"FLM"`, there is no dictionary word that begins with those letters, so any further exploration cannot yield a working solution. In such a case, your code should stop and backtrack immediately. You should also avoid making copies of data structures extremely high numbers of times by always passing them by reference.

*Constraints:* For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- Do not modify the `Lexicon` or `Map` passed in to your function.
- Do not declare any **global variables**.
- Your code can contain **loops** if appropriate to solve the problem, but your overall algorithm must be recursive.
- You are allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

*Write your answer on the next page.*

## 7. Backtracking (write)
Writing Space

## 8. Pointers and Linked Nodes (write)

Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown. You may construct a new `ListNode` object, but only if this is needed to create a node in the "after" picture that does not exist in the "before" picture. You are *not* allowed to change any existing node's `data` field value.
You may declare **a single `ListNode*` pointer** variable (aside from `list1` and `list2`) to point to any existing node.

You do not need to **free/delete any memory** on this problem.

To help maximize partial credit in case of mistakes, we suggest that you include optional **comments** with your code that describe the links you are trying to change, as shown in the solutions in our practice exams and section handouts.

**Before**

```
         +---+---+
list1 --> | 1 | / |
         +---+---+

         +---+---+    +---+---+
list2 --> | 2 |   |--> | 3 |   |
         +---+---+    +---+---+
```

**After**

```
         +---+---+    +---+---+
list1 --> | 3 |   |--> | 2 | / |
         +---+---+    +---+---+

         +---+---+    +---+---+
list2 --> | 1 |   |--> | 4 | / |
         +---+---+    +---+---+
```

Assume that you are using the `ListNode` structure as defined in lecture and section:

```
struct ListNode {
    int data;         // data stored in this node
    ListNode* next;   // a link to the next node in the list
    ...
};
```