CS106B Autumn 2021

Practice Midterm 1

(Print name **legibly**)

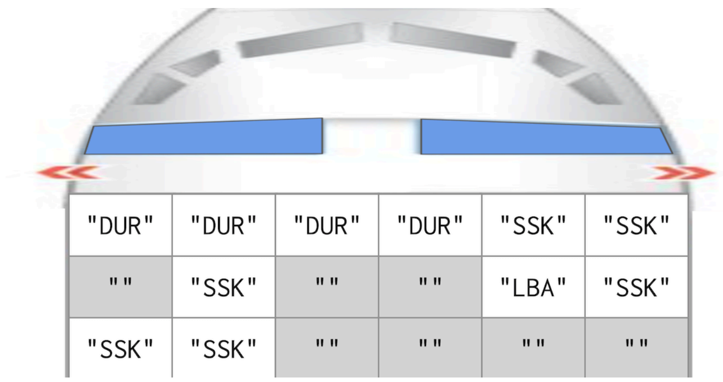
(SUID number)

Exam instructions: There are 5 questions worth a total of 70 points. Write all answers directly on the exam. This printed exam is **closed-book** and **closed-device**; you may refer only to your one letter-sized page of prepared notes and the provided reference sheet. You are required to write your SUID number in the blank at the top of each odd numbered page.

C++ coding guidelines: Unless otherwise restricted in the instructions for a specific problem, you are free to use any of the CS106 libraries and classes. You don't need `#include` statements in your solutions, just assume the required `vector`, `strlib`, etc. header files are visible. You do not need to declare prototypes. You are free to create helper functions unless the problem states otherwise. Comments are not required, but when your code is incorrect, comments could clarify your intentions and help earn partial credit.

(signature)

Problem 1: C++ fundamentals (12 points)
The seating chart for an airplane is stored in a `Grid<string>`. Each grid element is the group reservation code if that seat is assigned or an empty string if the seat is unoccupied. The diagram below is a seat grid of 3 rows and 6 columns. Of the 18 total seats, 11 are reserved and 7 are empty.



"DUR"	"DUR"	"DUR"	"DUR"	"SSK"	"SSK"
""	"SSK"	""	""	"LBA"	"SSK"
"SSK"	"SSK"	""	""	""	""

A *free block* is a sequence of adjacent empty seats within a row. In the diagram above:

- row 0 has no free block
- row 1 has two free blocks, a block of size 1 starting at location `{1, 0}` and a block of size 2 starting at location `{1, 2}`
- row 2 has a free block of size 4 starting at location `{2, 2}`

Write the function `findFreeBlock` that searches `seatGrid` for a free block of at least size `k`.

```
bool findFreeBlock(Grid<string>& seatGrid, int k, GridLocation& loc)
```

The function has three parameters:

- ○ `seatGrid`, a `Grid<string>` as described above
- ○ `k`, the size of the free block to find. You may assume that `k > 0`.
- ○ `loc`, a `GridLocation` passed by reference
    - ▪ `loc` is assigned to the start location of the free block if one is found. If there is more than one free block of the requested size, it may choose any of them.
    - ▪ If there is no such free block, `loc` is unchanged.

The function returns `true` if a free block was found and `false` otherwise.

If `seatGrid` is the diagram above:

- `findFreeBlock(seatGrid, 5, loc)` returns `false`, `loc` is unchanged.
- `findFreeBlock(seatGrid, 3, loc)` returns `true`, `loc` is either `{2, 2}` or `{2, 3}`.

```
bool findFreeBlock(Grid<string>& seatGrid, int k, GridLocation& loc)
```

Problem 2: ADTs (15 points)
*Note: This question builds on the airplane **seatGrid** introduced in the previous question.*

You are given the helper function **getSeatsForBlock** that returns a set of locations for the block of **k** seats starting at location **loc**:

```
Set<GridLocation> getSeatsForBlock(GridLocation loc, int k) {
    Set<GridLocation> seats;
    for (int i = 0; i < k; i++) {
        GridLocation next = {loc.row, loc.col + i};
        seats.add(next);
    }
    return seats;
}
```

The call **getSeatsForBlock({1, 2}, 2)** returns the set of locations **{ {1, 2}, {1, 3} }**. You will use this helper when implementing the **reseatGroup** operation.

In addition to the **seatGrid** previously described, the airline also maintains a **reservationDB** of type **Map<string, Set<GridLocation>>** . In the map, each key is a group reservation code and associated value is the set of seat locations assigned to the group. Here is the **reservationDB** that corresponds to the **seatGrid** diagram shown on page 2.

```
 Key          Value
---------------------------------
"DUR"   ->   { {0,0}, {0,1}, {0,2}, {0,3} }
"LBA"   ->   { {1,4} }
"SSK"   ->   { {0,4}, {0,5}, {1,1}, {1,5}, {2,0}, {2,1} }
```

The group "SSK" has been assigned non-adjacent seats. Write the function **reseatGroup** that attempts to reassign the entire group to adjacent seats in a single row.

```
int reseatGroup(Grid<string>& seatGrid,
                Map<string, Set<GridLocation>>& reservationDB,
                string groupCode)
```

The function has three parameters:

- **seatGrid**, a **Grid<string>** representing the airplane seat grid
- **reservationDB**, a **Map<string, Set<GridLocation>>** of reservations
- **groupCode**, the group reservation code of the group to reseat. You may assume the group has an entry in **reservationDB** and seat assignments in **seatGrid**.

To reseat the group, the function follows these steps:

- edit **seatGrid** to release the original seat assignments for the group.
- find a free block of adjacent empty seats. You should call **findFreeBlock** and **getSeatsForBlock** and may assume they work correctly.
- if a free block is found, assign that block of seats to the group and update both **seatGrid** and **reservationsDB** to reflect the new seat assignments
- if no reseating was possible, restore the group to their original seat assignments
- the group reservation code stays the same whether or not the group is reseated

The function returns the count of seat changes. If reseated, this is the number of different seats between the original and new seat assignments, or zero if no seat assignments were changed.

If `seatGrid` is the diagram on page 2, `reseatGroup(seatGrid, reservationDB, "SSK")` re-assigns the "SSK" group to the six seats in row 2 and return 4.

```
int reseatGroup(Grid<string>& seatGrid,
                Map<string, Set<GridLocation>>& reservationDB,
                string groupCode)
```

Problem 3: Code study: ADTs and Big-O (14 points)
The *expunge* operation modifies a collection by removing its first element and any duplicates of that value. You will study four versions of expunge, one each for Vector, Queue, Stack, and Set.

The `expungeVector` function removes a vector's first (index 0) element and all duplicates. This code has been tested and shown to work correctly on all inputs.

```
void expungeVector(Vector<int>& vec) {
   if (!vec.isEmpty()) {
      int first = vec[0];
      for (int i = vec.size()-1; i >= 0; i--) { // line 13
         int cur = vec[i];
         if (cur == first) vec.remove(i);
      }
   }
}
```

a) What is the Big-O runtime of `expungeVector` when running on a Vector of size N where every third element is a duplicate of the first element?

Here are some of the test cases that were used to confirm the correctness of `expungeVector`.

```
Input vector   Expected result    Actual result
-----------------------------------------------
 {}                  {}                {}
 {4, 1, 7}           {1, 7}            {1, 7}
 {4, 1, 4, 8}        {1, 8}            {1, 8}
```

The original code had no comment to provide a rationale for why the loop runs backwards. You change line 13 to iterate over the same indexes in forward order instead. Below is the edited code:

```
void expungeVector(Vector<int>& vec) {
   if (!vec.isEmpty()) {
      int first = vec[0];
      for (int i = 0; i < vec.size(); i++) { // changed line 13
         int cur = vec[i];
         if (cur == first) vec.remove(i);
      }
   }
}
```

b) Give an example test case (input, expected, actual) that would have worked correctly for the original code that will now fail after your edits.

The **expungeQueue** function intends to remove the queue's first (frontmost) element and all duplicates. It may or may not work as intended on all inputs.

```
void expungeQueue(Queue<int>& queue) {
   if (!queue.isEmpty()) {
     int first = queue.peek();
     for (int i = queue.size()-1; i >= 0; i--) {
         int cur = queue.dequeue();
         if (cur != first) queue.enqueue(cur);
     }
   }
}
```

c) What is the Big-O runtime of **expungeQueue** when running on a Queue of size N that contains no duplicate elements?

You create the test cases below to test **expungeQueue** on inputs where the queue's frontmost element is not duplicated elsewhere in the queue. **expungeQueue** works correctly on both test cases. (Note: when listing queue elements, the order is *frontmost* to *lastmost*.)

```
Input queue     Expected result    Actual result
-----------------------------------------------
 {}                  {}             {}
 {4, 1, 7}           {1, 7}         {1, 7}
```

You now move on to writing test cases when the frontmost element is duplicated.

d) Give an example test case in which the frontmost element is duplicated for which **expungeQueue** works correctly (or write None if there is no such input).

e) Give an example test case in which the frontmost element is duplicated for which **expungeQueue** does not work correctly (or write None if there is no such input).

The **expungeStack** function intends to remove the stack's first (topmost) element and all duplicates. It may or may not work as intended on all inputs.

```
void expungeStack(Stack<int>& stack) {
   if (!stack.isEmpty()) {
      int first = stack.peek();
      for (int i = stack.size()-1; i >= 0; i--) {
         int cur = stack.pop();
         if (cur != first) stack.push(cur);
      }
   }
}
```

You create the test cases below to test **expungeStack** on inputs where the stack's topmost element is not duplicated elsewhere in the stack. **expungeStack** works correctly on both test cases. (Note: when listing stack elements, the order is *bottommost* to *topmost*.

```
Input stack   Expected result    Actual result
------------------------------------------------
 {}                 {}              {}
 {7, 1, 4}          {7, 1}          {7, 1}
```

You now move on to test cases when the stack's topmost element is duplicated.

f) Give an example test case in which the topmost element is duplicated for which **expungeStack** works correctly (or write None if there is no such input).

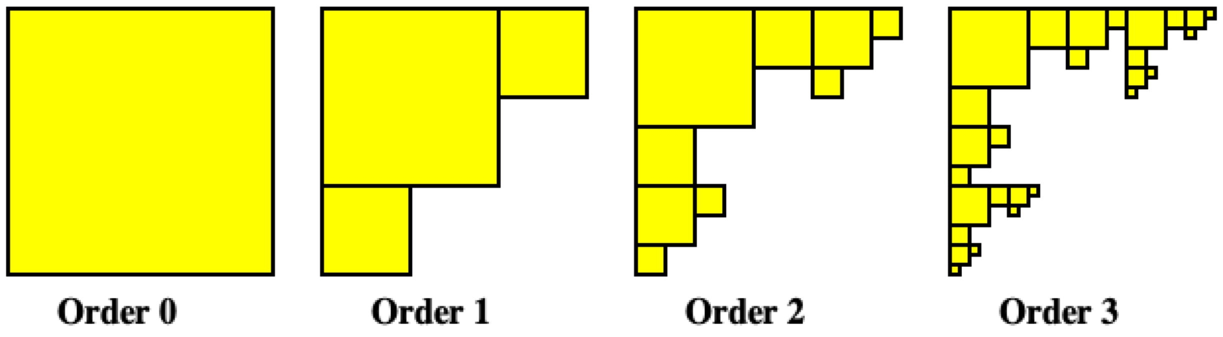
g) Give an example test case in which the topmost element is duplicated for which **expungeStack** does not work correctly (or write None if there is no such input).

Problem 4: Recursive fractal (14 points)

The Box Trio fractal is a self-similar pattern that is defined recursively:

- An order-0 Box Trio is a single filled yellow square.
- An order-n Box Trio consists of three Box Trios of order n – 1 arranged in an inverted L shape. The larger middle Box Trio has a side length equal to 2/3 of the original side length. It is flanked by two smaller Box Trios with a side length equal to 1/3 of the original side length.

Here are the first few orders of the Box Trio fractal:



We provide the function `double drawYellowBox(GPoint upperLeft, double length)` that draws a single filled yellow square of size `length` at position `upperLeft`. As a courtesy, `drawYellowBox` returns the area of the box that was drawn.

Write the function `drawBoxTrio`

```
double drawBoxTrio(GPoint upperLeft, double length, int order)
```

Here are the specifications for `drawBoxTrio`:

- The three parameters are the upper left corner, the side length, and the order.
- The function draws a Box Trio fractal of `order` with upper left corner positioned at `upperLeft` and side length equal to `length`. You may assume that `order >= 0`.
- The return value from `drawBoxTrio` is the area of the largest yellow box drawn in the fractal. You should not calculate this via a formula but instead use a recursive approach.
- The provided test below confirms the function result for some simple cases:

```
PROVIDED_TEST("Confirm result from drawBoxTrio") {
    EXPECT_EQUAL(drawBoxTrio(pt, 9.0, 0), 81.0);
    EXPECT_EQUAL(drawBoxTrio(pt, 9.0, 1), 36.0);
    EXPECT_EQUAL(drawBoxTrio(pt, 9.0, 2), drawBoxTrio(pt, 6.0, 1));
}
```

```
double drawBoxTrio(GPoint upperLeft, double length, int order)
```

Problem 5: Recursive backtracking (15 points)
A letter sequence is a string consisting of one or more lowercase letters. We define an alphabetic letter sequence as one in which the letter characters are in alphabetical order. The letter sequence `abettz` is alphabetic, `hay` is not. You are curious how many alphabetic letter sequences exist that are valid English words.

You are given the function `printCombos` that uses recursion to print all possible letter sequences of a given length. Read over this code as a starting point. You can borrow code and structure from `printCombos` when writing `countAlphabeticWords`.

```
void printCombos(int length, string soFar) {
    if (soFar.length() == length) {
        cout << soFar << endl;
    } else {
        for (char ch = 'a'; ch <= 'z'; ch++) {
            printCombos(length, soFar + ch);
        }
    }
}

// wrapper function
void printCombos(int length) {
    printCombos(length, "");
}
```

You are to write the function `countAlphabeticWords`. The one argument to the function is a Lexicon of English words. The function recursively explores all alphabetic letter sequences, counts those sequences that are valid words, and returns the total count.

Your solution is subject to the following restrictions:

- You must not create any auxiliary data structures.
- Your overall algorithm must be recursive and must use backtracking techniques to generate the result.
- You should only explore letter sequences that are alphabetic and should use the lexicon's `containsPrefix` functionality to avoid exploring dead-end paths.
- You will need to write a helper/wrapper function.
- The function returns only the count of alphabetic words. Do not print or return the words.

```
int countAlphabeticWords(Lexicon& lex)
```