CS 106X, Autumn 2016
Final Exam, Monday, December 12, 2016

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 submit are my own work.*

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

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

- This is an **individual exam**; you are to complete it yourself without assistance from others.
- You have 3 hours (180 minutes) to complete this exam.
- This test is **open-book**, but **closed notes**. You may not use any printed 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. Some particular problems have style constraints or other code constraints, so read each problem carefully. We reserve the right to deduct points for solutions that are grossly inefficient or wasteful of resources.
- 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.
- Please do not **abbreviate** code, such as writing ditto marks ("") or dot-dot-dot marks (...).
- Unless otherwise specified, you may define **helper functions** but you may not declare **global variables**.
- 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!

Problem	Description	Earned	Possible
1	Linked Lists (read)		6
2	Linked Lists (write)		10
3	Binary Search Trees (read)		6
4	Binary Trees (write)		10
5	Graphs (read)		6
6	Graphs (write)		10
7	Hashing (read)		6
8	Inheritance (read)		6
TOTAL	**Total Points**		**60**

1. Linked Lists (read)

Consider the following linked list of `ListNode` objects, with a pointer named `front` that points to the first node:

```
front -> [55] -> [10] -> [2] -> [3] -> [4] -> [20] -> [7] -> [6] -> [8] -> [9] -> [12] -> [15] /
```

Draw the final state of the linked list after the following code runs on it. If a given node is removed from the list, you don't need to draw that node, only the ones that remain reachable in the original list.

```
void linkedListMystery(ListNode*& front) {
    ListNode* curr = front;
    ListNode* next = curr->next;
    while (next != nullptr) {
        if (curr->data % 5 == 0) {
            front = front->next;
        } else if (curr->data % 2 == 0 && next->data % 2 == 0) {
            curr->next = next->next;
        } else if (curr->data % 3 == 0) {
            next->data++;
            curr->data--;
            curr = next;
        }
        curr = next;
        next = next->next;
    }
}
```

(student initials)

2. Linked Lists (write)

Write a function **`clump`** that groups together nodes in a linked list that store the same value. Your code should rearrange the linked list so that all occurrences of duplicate values will occur in consecutive order at the site of the first occurrence of that value in the list. For example, you should "clump" all the 4s in the list at the site of the first 4 in the list. The node clumps should remain in the same relative order as in the original list.

Your function accepts two parameters: a reference to a `ListNode` pointer representing the front of the linked list *(see reference sheet)*, and an integer *max* for the maximum number of values to clump together; any additional occurrences of that same value must be removed (and their memory must be freed). For example, if *max* is 3 but there are 5 occurrences of the value 10, you should keep only 3 of those 5 occurrences and remove the other 2 occurrences of 10 from the list.

Suppose a `ListNode` pointer variable named `front` points to the front of a list storing the following values:

```
{1, 6, 5, 2, 6, 4, 5, 3, 5, 8, 5, 2, 8, 4, 5, 6, 8, 6}          // original list
```

After the call of `clump(front, 99);` , the list should store the following elements (clumps are underlined):

```
{1, 6, 6, 6, 6, 5, 5, 5, 5, 5, 2, 2, 4, 4, 3, 8, 8, 8}          // after clump(front, 99);
```

In the preceding call, the *max* value passed was very large, so no elements needed to be removed from the list. If the call had instead been `clump(front, 2);` , the list would instead store the following elements afterward. Notice that the third and fourth occurrence of 6, the third through fifth occurrences of 5, and the third occurrence of 8 were removed.

```
{1, 6, 6, 5, 5, 2, 2, 4, 4, 3, 8, 8}                            // after clump(front, 2);
```

Your function should work properly for a list of any size.

If the value of *max* passed is 0 or negative, you should throw an integer **exception**.

Note that the goal of this problem is to modify the list by modifying **pointers**. It might be easier to solve it in other ways, such as by changing nodes' `data` values or by rebuilding an entirely new list, but such tactics are forbidden.

Constraints: For full credit, obey the following restrictions in your solution. A violating solution can get partial credit.

- **Do not modify the `data` field** of any existing nodes.
- **Do not create any new nodes** by calling `new ListNode(...)`.
  You may create as many `ListNode*` pointers as you like, though.
- Do not use any auxiliary **data structures** such as arrays, vectors, queues, maps, sets, strings, etc.
- Do not **leak memory**. If you remove a node from the list, free its memory.
- Your code must run in no worse than **$O(N^2)$ time**, where N is the length of the list.

Write your answer on the next page.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

---

2. Linked Lists (write)
Writing Space

(student initials)

2. Linked Lists (write)
Writing Space

(student initials)

3. Binary Search Trees (read)

(a) Write the binary search tree that would result if these elements were **added** to an empty **binary search tree** of integers (a simple BST, not a re-balancing AVL tree) in this order.

- 49, 36, 42, 72, 77, 86, 75, 40, 24, 55, 27, 21, 82, 60, 39

(b) Now draw below what would happen to your tree from the end of (a) if all of the following values were **removed**, in this order *(using the BST remove algorithm shown in lecture)*:

- 42, 77, 49, 60

(c) Examine your tree from the end of part (b) (<u>after</u> all removes are done) and answer the following questions about it.

- Is the overall tree balanced? Circle one. **Yes** **No**
- If the tree is **balanced**, briefly explain how you know this by writing your written justification next to the tree. If the tree is ***not*** **balanced**, circle and/or clearly mark **<u>all</u>** node(s) that are unbalanced.

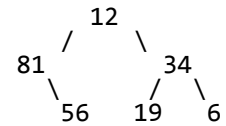
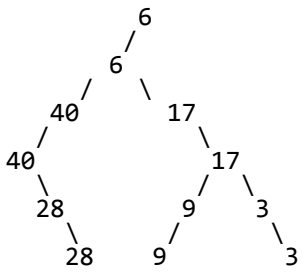
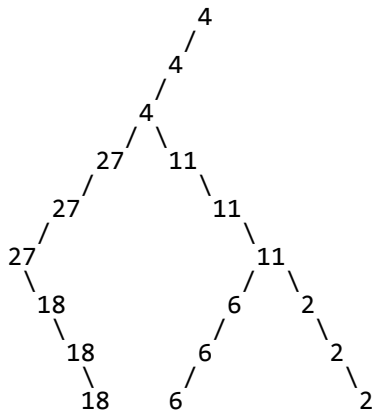
(student initials)

4. Binary Trees (write)

Write a recursive function named **stretch** that replaces each single binary tree node with multiple nodes with smaller values. Your function accepts two parameters: a reference to a `TreeNode` pointer representing the root of a binary tree *(see reference sheet)*, and an integer "stretching factor" *K*. Your function should replace each node *N* with *K* nodes, each of which stores a `data` value that is 1/*K* of *N*'s original value, using integer division.

The new clones of node *N* should extend from their parent in the same direction that *N* extends from its parent. For example, if *N* is its parent's left child, the stretched clones of *N* should also be their parent's left child, and vice versa if *N* was a right child. The root node is a special case because it has no parent; we will handle this by saying that its stretched clones should extend to the left.

For example, suppose a variable named `root` refers to the root of the tree below at left. If we then make the function call of `stretch(root, 2);` , notice that the root node of value 12 has become two nodes of value 6. Its left child 81 has stretched into two leftward branch nodes storing 40 (which is 81 / 2 rounded down). Its right child 34 has stretched into two rightward branch nodes storing 17. And so on. We also demonstrate the result of calling your function on the same original tree with a stretch factor of 3, in which each node stretches itself into three smaller valued nodes. The root of 12 stretches into 3 nodes of 4; the children of 81 and 34 stretch into 3 nodes of 27 and 11 respectively; and so on.

tree	after `stretch(root, 2);`	after `stretch(root, 3);`
		

If the stretch factor *K* passed is 0 or negative, throw an integer **exception**.

Constraints: For full credit, obey the following constraints in your solution. A violating solution can get partial credit.

- Do not create any **data structures** (arrays, vectors, sets, maps, etc.).
- Do not create any **unnecessary TreeNode objects**. If your stretch factor is *K*, you should create *K*-1 new nodes for each existing node in the tree, but no more. In particular, do not throw away the existing nodes that were present in the tree; modify and reuse them as much as possible.
- Do not **leak memory**.
- For full credit, your solution should be at worst O(*N* * *K*) time, where *N* is the number of elements in the tree. You must also solve the problem using a **single pass** over the tree, not multiple passes.
- You may define **private helper** functions if you like.
- Your solution must be **recursive**. Loops are allowed, but your overall traversal of the tree should use recursion.

Write your answer on the next page.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

---

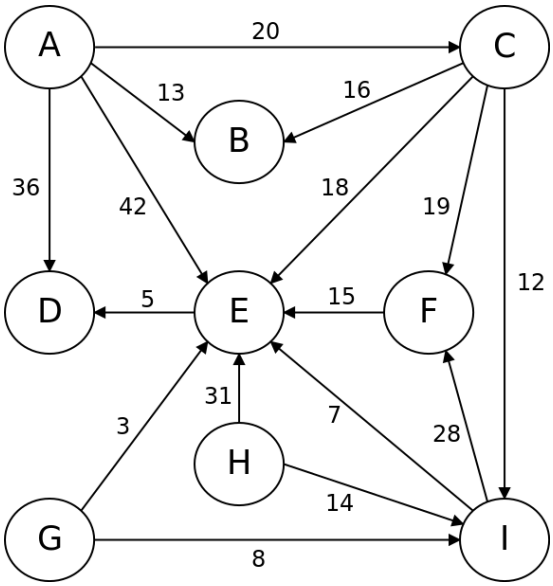
4. Binary Trees (write)
Writing Space

(student initials)

4. Binary Trees (write)
Writing Space

(student initials)

5. Graphs (read)

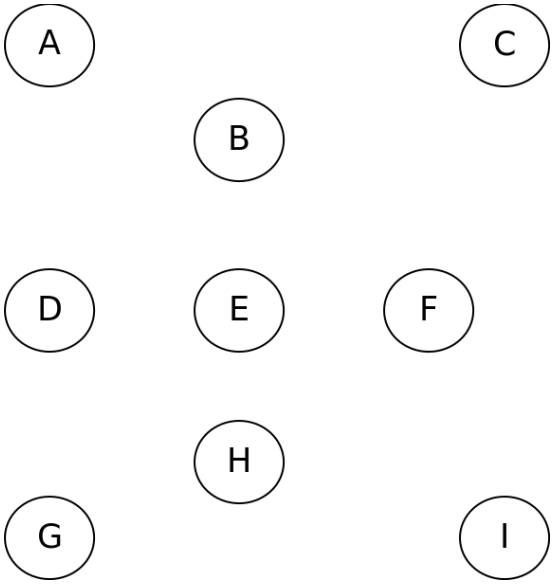


For the weighted, directed graph shown above, answer the following two questions:

a) Write a valid **topological sort** of the vertexes in the graph. If there are multiple valid sort orders, any will be fine.

- **Sort Order:** _____

b) Use Kruskal's algorithm to generate a **minimum spanning tree** of the graph. Draw your MST below.
(Kruskal's algorithm is used on undirected graphs, so for this part only, pretend that all edges above are bidirectional.)

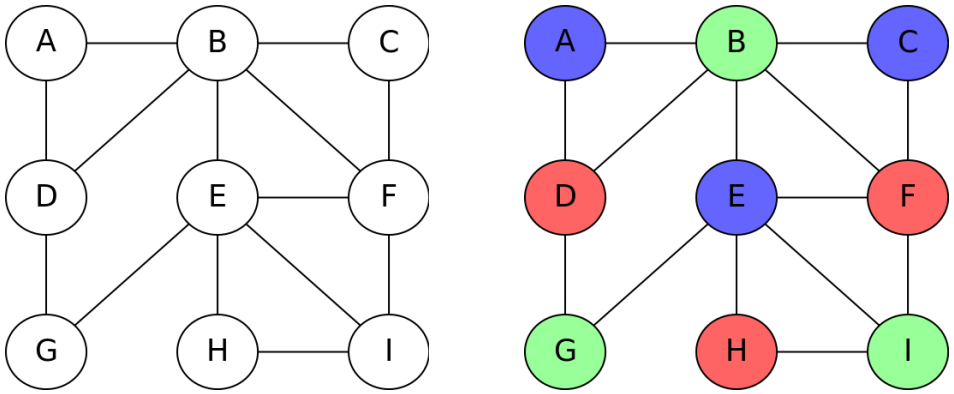


(student initials)

6. Graphs (write)

Write a function named **colorGraph** that attempts to assign colors to vertexes of an undirected connected graph from a given collection of available colors such that no neighboring vertexes have the same color as each other. Your function accepts two parameters: a reference to a `BasicGraph`, and a reference to a `Vector` of strings representing available colors. You should return a `Map` from `Vertex` pointers to strings, representing colors to assign to each vertex.

The diagram below shows an example graph that might be passed to your algorithm. Our `BasicGraph` collection is generally used to represent directed graphs, but in this case you may assume that the graph represents an undirected graph and that for every edge A → B there will also be an edge B → A. At right is an example coloring of the graph using 3 colors: blue (vertexes A, C, E), green (vertexes B, G, I), and red (vertexes D, F, H). *(We realize that colors don't show up on the printed exam; we're sorry!)* This particular graph cannot be colored successfully with fewer than 3 colors, but any value of 3 or greater must work successfully.



If the graph above is represented by a `BasicGraph` object named `graph`, and a vector named `colors` contains the strings `{"blue", "green", "red"}`, then the call of `colorGraph(graph, colors)` should return a map like the following, where the letters like A, B, C represent pointers to those corresponding `Vertex` structures in the graph:If the graph above is represented by a `BasicGraph` object named `graph`, and a vector named `colors` contains the strings `{"blue", "green", "red"}`, then the call of `colorGraph(graph, colors)` should return a map like the following, where the letters like A, B, C represent pointers to those corresponding `Vertex` structures in the graph:

```
{A:"blue", B:"green", C:"blue", D:"red", E:"blue", F:"red", G:"green", H:"red", I:"green"}
```

If there are multiple valid ways to color the graph, your function can return any one of them. If the coloring cannot be performed with the given number of colors, or if the colors vector is empty, return an empty map.

Efficiency: Note that in order to be certain that you have exhaustively found a suitable coloring (or verified with certainty that there is no coloring), you must try every possible combination of vertex/color mappings. This can be a slow process, so for full credit, your code must avoid exploring options that are certain to be unable to find a valid result. For example, if your code assigned vertex A the color blue, it should not explore giving vertex B (A's neighbor) the same color of blue.

Assumptions: You may assume that the graph is **connected**, that the graph's state is valid, and that it contains no self-edges (e.g. from *V1* to *V1*). You may assume that there is at most one edge from any vertex *V1* to any other vertex *V2*.

Constraints: Do not modify the contents of the graph such as by adding or removing vertexes or edges from the graph, though you may modify the state variables inside individual vertexes/edges if you like. You may define **private helper** functions if so desired, and you may construct auxiliary **collections** as needed to solve this problem.

Write your answer on the next page.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

6. Graphs (write)
Writing Space

(student initials)

6. Graphs (write)
Writing Space

(student initials)

7. Hashing (read)

Simulate the behavior of a **hash map** of integers as described and implemented in lecture. Assume the following:

```
HashMap map;
map.put(3, 12);
map.put(4, 11);
map.put(77, 1);
map.put(7, 8);
map.put(5, 22);
map.put(17, 3);
map.put(19, 108);
map.put(47, 100);
map.put(44, 46);
map.put(11, 22);
map.put(2, 17);
map.put(1, 19);
map.put(11, 100);
map.put(9, 181);
map.remove(17);
map.remove(181);
if (map.containsKey(122)) {
    map.put(1, 1);
} else {
    map.put(28, 3);
}
int x = 1;
while (map.containsKey(3)) {
    if (map.containsKey(x)) {
        map.remove(x);
    }
    x++;
}
map.put(47, 100);
```

- the hash table array has an initial capacity of **10**
- the hash table uses **separate chaining** to resolve collisions
- the **hash function** returns the absolute value of the integer key, mod the capacity of the hash table
- **rehashing** occurs at the *end* of an add where the load factor is ≥ **0.75** and doubles the capacity of the hash table

Draw an array diagram to show the final state of the hash table after the following operations are performed. Leave a box empty if an array element is unused. All values must be in the proper buckets and in the proper order to receive full credit. Also write the final **size**, **capacity**, and **load factor** of the hash table.

You do not have to redraw an entirely new hash table after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to redraw the table at various important stages to help earn partial credit in case of an error. If you draw various partial or in-progress diagrams or work, please **circle your final answer**.

(student initials)

8. Inheritance and Polymorphism (read)

Consider the following classes; assume that each is defined in its own file.

```
class Grinch : public Frosty {
public:
    virtual void m2() {
        cout << "Grinch m2   ";
        m1();
    }

    virtual void m3() {
        cout << "Grinch m3   ";
        Frosty::m3();
    }

    virtual void m4() {
        cout << "Grinch m4   ";
        m3();
    }
};

class Frosty : public Santa {
public:
    virtual void m1() {
        cout << "Frosty m1   ";
    }

    virtual void m3() {
        cout << "Frosty m3   ";
    }
};

class Santa {
public:
    virtual void m1() {
        m3();
        cout << "Santa m1   ";
    }

    virtual void m3() {
        cout << "Santa m3   ";
    }
};

class Rudolph : public Santa {
public:
    virtual void m2() {
        cout << "Rudolph m2   ";
        m1();
    }

    virtual void m3() {
        cout << "Rudolph m3   ";
        Santa::m3();
    }
};
```

Now assume that the following variables are defined:

```
Santa*  var1 = new Frosty();
Frosty* var2 = new Grinch();
Santa*  var3 = new Grinch();
Santa*  var4 = new Rudolph();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the **line breaks with slashes** as in "x / y / z" to indicate three lines of output with "x" followed by "y" followed by "z".

If the statement does not compile, write "**compiler error**".
If a statement would crash at runtime or cause other unpredictable behavior, write "**crash**".

Statement	**Output**
`var1->m1();`	_____
`var1->m2();`	_____
`var1->m3();`	_____
`var2->m1();`	_____
`var2->m2();`	_____
`var2->m3();`	_____
`var3->m2();`	_____
`var4->m1();`	_____
`((Frosty*) var1)->m1();`	_____
`((Frosty*) var1)->m4();`	_____
`((Grinch*) var1)->m4();`	_____
`((Grinch*) var2)->m4();`	_____
`((Rudolph*) var3)->m2();`	_____

(student initials)