

## PRACTICE MIDTERM EXAM #1

NAME (LAST, FIRST): \_\_\_\_\_

SUNET ID: \_\_\_\_\_@stanford.edu

Problem	1	2	3	4	5	TOTAL
Topic	Strings and ADTs	Pointers, Memory	Recursion	Classes	Big-O	
Score						
	20	10	13	20	9	72

## Instructions:

- The time for this exam is **2 hours**, or 120 minutes. There are 72 points possible, so about 1.5-2 minutes per point. This should give you a general sense of pacing, though each person will have different problems that challenge them more or less than others.
- Use of anything other than a pencil, eraser, pen, one 8.5x11 page (two sides) of notes, and the official textbook is prohibited. In particular, **no** computers or digital devices of any kind are permitted. Blank scratch paper may be provided by proctors and does not need to be turned in.
- PLEASE rip off the pages of library reference in the back of the exam and do not turn them in.
- Please do NOT staple or otherwise insert new pages into the exam. Changing the number of pages in the exam confuses our automatic scanning system. Thanks.
- SCPD and OAE: Please call or text Cynthia @ 760-845-7489 if you have a question.

Please sign *before* you begin:

I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

---

 \_\_\_\_\_  
 (Signature) (Date)

- 
1. **Strings and ADTs (20pts).** In the English language, some combinations of adjacent letters are more common than others. For example, h often follows t (“th”), but less frequently would you see x following t (“tx”). Knowing how often a given letter follows other letters in the English language is useful in many contexts.

*Example context: in cryptography, we use this data to crack substitution ciphers (codes where each letter has been replaced by a different letter, for example: A→M, B→T, etc.) by identifying which possible decoding substitutions produce plausible letter combinations and which produce nonsense.*

For this problem, use our English lexicon and at least one other well-chosen ADT to compile data on how often letters follow each other.


- Write a function `static void printCharPairFrequencies(const Lexicon& english)` that compiles this data and then prints 2-character strings of letters and a count of how often that pairing occurred in our list of English words (as given in the lexicon).
- Format the string and count with a space between them, and each string and count on their own line. Example:

```
aa 194
ab 8121
ac 10801
...
zw 17
zy 305
zz 651
```

- Print the pairings in alphabetical order, skipping any character pairs that do not occur in any word the lexicon.
- Pairings that occur more than once in the same word should be counted as separate occurrences. (Example: “re” occurs twice in “reread”)
- To do this you will need to loop over the entire lexicon. A `foreach` loop would work for this purpose.

```
static void printCharPairFrequencies(const Lexicon& english) {
```

---

- 
2. **Pointers and Memory (10pts).** Draw the state of memory at the end of the execution of this code. Be careful in showing where your pointers originate and terminate (outer box vs. inner box). Leave uninitialized or unspecified areas blank, and clearly mark NULL (write NULL or draw a slash through the box). Draw the components in the appropriate stack and heap areas marked for you. Mark memory that has been deleted by enclosing it in a circle with a slash through it, like this:  but leave it where it is, and do not change/remove any pointer arrows or other values unless they are actually changed.

```
Slayer* vamp = new Slayer[2];
vamp[1].angel = 2;
vamp[1].buffy = new Slayer;
vamp[0].buffy = vamp;
vamp[0].angel = 3;
delete vamp[1].buffy;
vamp[1].buffy = NULL;
//Draw the state of memory now
```

```
struct Slayer {
    int angel;
    Slayer * buffy;
};
```

DRAWING:

Stack:

Heap:

---

- 
3. **Recursion (13pts). [by Marty Stepp]** Write a recursive function **moveToEnd** that accepts a string *s* and a character *c* as parameters, and returns a new string similar to *s* but with all instances of *c* moved to the end of the string. The relative order of the other characters should be unchanged from their order in the original string *s*. If the character is a letter of the alphabet, all occurrences of that letter in either upper or lowercase should be moved to the end and converted to uppercase. If *s* does not contain *c*, it should be returned unmodified. The following table shows calls to your function and their return values. Occurrences of *c* are underlined for clarity.

Call	Returns
moveToEnd("he <u>l</u> lo", 'l')	"heo <u>LL</u> "
moveToEnd("he <u>l</u> lo", 'e')	"hll <u>oE</u> "
moveToEnd("he <u>l</u> lo <u>T</u> HERE", 'e')	"hll <u>o</u> <u>T</u> HRE <u>EE</u> "
moveToEnd("hello there", 'q')	"hello there"
moveToEnd("ba <u>n</u> A <u>n</u> A ra <u>m</u> A", 'A')	"bnn rm <u>AAAAA</u> "
moveToEnd(" <u>x</u> ", 'x')	" <u>X</u> "
moveToEnd("", 'x')	""

You may not construct any data structures (no array, vector, stack, etc.), and you may not use any loops to solve this problem; you must use recursion. Also, you may not use any global variables in your solution.

```
string moveToEnd(string s, char c) {
```

```
}
```

- 
4. **Classes (20pts)**. This is a problem about using classes to organize your classes (so meta!). Given that we are halfway through the quarter, you will most likely have started to think about which classes to take next quarter. We can think about each class as storing information such as: day of the week of its lecture, lecture start time, and lecture duration. Your goal in this problem is to implement a `Lecture` class that will represent this information.

For simplicity, we assume that all Stanford classes meet either Monday-Wednesday-Friday (MWF) or Tuesday-Thursday (TTH), and no other combinations. Your `Lecture` should represent the lecture times for exactly one class. For example, for CS106B, your `Lecture` should remember that we meet MWF at 1430 (i.e., 2:30 PM) for 50 minutes (which you of course already knew).

In addition to exposing the day, time, and duration for each lecture, you should also provide a couple of methods that will help students scheduling classes. Here is the interface:

```
enum DayPattern { MWF, TTH };

class Lecture {
public:
    // You'll implement these in Part (b)
    Lecture(DayPattern daysOfWeek, int startTime, int duration);
    int startTime() const;
    int durationInMinutes() const;
    DayPattern daysOfWeek() const;

    // You'll implement this in Part (c)
    int endTime() const;

private:
    // You'll fill this in in Part (a)
};
```

You may assume that no lecture will span midnight (e.g. will not start at 23:00 and last for more than one hour). You may also assume that all times will be military time in PST (e.g. 1700 not 5:00pm), so **you can handle hours and minutes as a single number** and not worry about time zones.

- a) **(5 Points)** Fill in the private section of the Lecture class. This includes any instance variables as well method prototypes for helper methods you implement.

```
class Lecture {  
public:  
    Lecture(DayPattern daysOfWeek, int startTime, int duration);  
    int startTime() const;  
    int durationInMinutes() const;  
    DayPattern daysOfWeek() const;  
  
    int endTime() const;
```

**private:**

```
    // Fill this in
```

```
};
```

b) **(10 Points)** Fill in the constructor and getters for the start time, duration, and weekday for the Lecture.

```
Lecture::Lecture(DayPattern daysOfWeek, int startTime, int duration) {
```

```
}
```

```
int Lecture::startTime() {
```

```
}
```

```
int Lecture::durationInMinutes() const {
```

```
}
```

```
DayPattern Lecture::daysOfWeek() const {
```

```
}
```



- c) **(5 Points)** Implement the method **endTime** that returns the time of day that a given lecture ends. For example, if your Lecture represents a CS106X lecture this quarter where the **startTime** is 1100, you should return 1150. You are welcome to use additional helper methods, but you are not required to.

```
int Lecture::endTime() const {
```

```
}
```

---

5. **Big-O (9pts).** Give a tight bound of the nearest runtime complexity class (worst-case) for each of the following code fragments in Big-O notation, in terms of variable N. As a reminder, when doing Big-O analysis, we write a simple expression that gives only a power of N, such as  $O(N^2)$  or  $O(\log N)$ , *not* an exact calculation. Write your answer in the blanks on the right side.

Question (3pts each)	Answer
<pre>// Let N = vec.size() int myfunction(Vector&lt;int&gt; vec){     int N = vec.size();     int sum = 0;     for(int i = 0; i &lt; vec.size(); i += (vec.size() / 3)) {         cout &lt;&lt; vec[i] &lt;&lt; endl;         sum += vec[i];     }     return sum; }</pre>	$O(\quad)$
<pre>// Let N be vec.size() of the vec of the *original* // call to recursiveFind. bool recursiveFind(Vector&lt;int&gt; vec, int key) {     if (vec.size() == 0) return false;     if (vec[vec.size()/2] == key) return true;     Vector&lt;int&gt; left;     for (int i = 0; i &lt; vec.size()/2; i++) {         left.add(vec[i]);     }     if (recursiveFind(left, key)) return true;     Vector&lt;int&gt; right;     for (int i = vec.size()/2 + 1; i &lt; vec.size(); i++) {         right.add(vec[i]);     }     if (recursiveFind(right, key)) return true;     return false; }</pre>	$O(\quad)$
<pre>// Let N be the value of the input named N int myfunction(int N) {     Grid&lt;int&gt; matrix(N / 2, N / 2);     for (int i = 0; i &lt; N/2; i++) {         for (int j = 0; j &lt; N/2; j++) {             matrix[i][j] = 3;         }     }     cout &lt;&lt; "done!" &lt;&lt; endl; }</pre>	$O(\quad)$

---

## Summary of Relevant Data Types

We tried to include the most relevant member functions and operators for the exam, but not all are listed. You are free to use ones not listed here that you know exist. *You do not need to do #include for these.*

```
class string {
    bool empty() const;
    int size() const;
    int find(char ch) const;
    int find(char ch, int start) const;
    string substr(int start) const;
    string substr(int start, int length) const;
    char& operator[](int index);
    const char& operator[](int index) const;
};
```

```
class Vector {
    bool isEmpty() const;
    int size() const;
    void add(const Type& elem); // operator+= used similarly
    void insert(int pos, const Type& elem);
    void remove(int pos);
    Type& operator[](int pos);
};
```

```
class Grid {
    int numRows() const;
    int numCols() const;
    bool inBounds(int row, int col) const;
    Type get(int row, int col) const; // cascade of operator[] also works
    void set(int row, int col, const Type& elem);
};
```

```
class Stack {
    bool isEmpty() const;
    void push(const Type& elem);
    Type pop();
};
```

```
class Queue {
    bool isEmpty() const;
    void enqueue(const Type& elem);
    Type dequeue();
};
```

```
class Map {
    bool isEmpty() const;
    int size() const;
    void put(const Key& key, const Value& value);
    bool containsKey(const Key& key) const;
    Value get(const Key& key) const;
    Value& operator[](const Key& key);
};
```

*Example for Loop:* for (Key key : mymap){...}

```
class Set {
    bool isEmpty() const;
    int size() const;
    void add(const Type& elem); // operator+= also adds elements
    bool contains(const Type& elem) const;
};
```

*Example for Loop:* for (Type elem : mymap){...}

```
class Lexicon {
    int size() const;
    bool isEmpty() const;
    void clear();
    void add(std::string word);
    bool contains(std::string word) const;
    bool containsPrefix(std::string prefix) const;
};
```

*Example for Loop:* for (string str : english){...}