CSCI-1200 Data Structures
Test 2 — Practice Problems

Note: This packet contains selected practice problems from Test 2 from three previous years. Your test will contain approximately one third to one half as many problems (totalling ~100 pts).

1 The Dynamic Tetris Slide [35 pts]

Our implementation of the Tetris game for Homework 3 only allowed pieces to drop vertically. The full game rules also allow pieces to move horizontally, which can be used by a skilled player to tuck in underneath an “overhang”. In this problem we will extend our solution with the slide function that allows the square piece, the 'O' piece, to slide one space to the right. For this problem you don’t need to worry about sliding any other piece shape, or about sliding to the left.

Below are two example Tetris games showing how this function works.

![Example Games]

The representation for the Tetris class consists of 3 private member variables: data, heights, and width. The memory layout for the 4th diagram above is shown to the right. Remember that we must maintain the arrays to be exactly as long as necessary to store the blocks on the board. The space character is used to represent empty air underneath a block.

The slide function takes in 2 integers, the row and column of the lower left corner block of the square 'O' piece that we want to slide.

We will also implement the can_slide function which first tests whether a piece is able to slide to the right. It will return false if the 'O' piece at the specified row and column is already at the right edge of the board, e.g., calling can_slide(1,4) in the third image above returns false. It will return false if the 'O' piece at the specified row and column is blocked by another piece on the board, e.g., calling can_slide(2,2) in the 5th image above will return false.

1.1 Algorithm Analysis [5 pts]

Assume that the game board has width w, the height of the tallest column is h, and the number of blocks (total number of piece characters and space characters) is b. What is the Big O Notation for the running time of your can_slide and slide functions that you have implemented on the next two pages? Write two to three concise and well-written sentences justifying your answers.

can_slide:

slide:
1.2  can_slide Implementation [ 12 pts ]

bool Tetris::can_slide(int row, int column) const {

    // First, let's do some error checking on input arguments
    // and the current board state. This will help when we need
    // to debug this new function. Write if/else statements
    // and/or assertions to verify your assumptions.

    // sample solution: 8 line(s) of code

    // Now, we can do the logic necessary to determine whether this piece
    // can slide to the right.

    // sample solution: 6 line(s) of code

}
1.3 slide Implementation [18 pts]

```cpp
void Tetris::slide(int row, int column) {
    assert (can_slide(row, column) == true);
}
```

sample solution: 26 line(s) of code
std::vector<std::string> a;
std::list<std::string> b;
// omitted: initialize both containers to hold n = a large number of words

01 a.push_front("apple");
02 b.push_front("banana");
03 a.push_back("carrot");
04 b.push_back("date");
05 std::vector<std::string>::iterator itr_a = a.begin();
06 std::list<std::string>::iterator itr_b = b.begin();
07 itr_a = a.insert(itr_a,"eggplant");
08 itr_a += 5;
09 itr_a = a.erase(itr_a);
10 itr_b += 5;
11 itr_b = b.insert(itr_b,"eggplant");
12 ++itr_b;
13 itr_b = b.erase(itr_b);
14 a.sort();
15 b.sort();
16 std::sort(a.begin(),a.end());
17 std::sort(b.begin(),b.end());

Which lines result in a compilation error?

Which lines cause a segmentation fault?

Which lines have a memory leak?

Which lines run in $O(1)$ time?

Which lines run in $O(n)$ time?

Which lines run in $O(n \log n)$ time?

Which lines run in $O(n^2)$ time?
Alyssa P. Hacker and Ben Bitdiddle are working on a team project based on the linked grid of `Node` data structure from Homework 5. Alyssa suggests they start with the `print_perimeter` function, which takes in a pointer to a `Node` named `start`, and walks around the edge of the grid in a *clockwise direction*. The function should print the value stored in every `Node` visited.

For example, `print_perimeter(start)` for the diagram shown on the right will print this sequence of values to the screen:

```
B C D H L K J I E A
```

Alyssa says it’s ok to assume that the grid is at least two rows tall and at least two columns wide and that `start` definitely points to a `Node` somewhere on the edge/perimeter of the grid.

### 3.1 Implement `print_perimeter` [ 12 pts ]

```cpp
template <class T> class Node {
public:
  T value;
  Node<T> *up,*down,*left,*right;
};
```
Meanwhile, Ben is working on a function named `rebutton`, which takes in 2 arguments: `start`, a pointer to a `Node` on the top edge of the grid and a bool `shift_up`. The function makes a vertical cut to the right of `start` and reconnects the `Nodes` on either side of the cut shifted up (below left) or shifted down (below right) one row. Ben claims that calling `rebutton(start, true)` followed by `rebutton(start, false)` will restore the original grid. And vice versa.

### 3.2 Implement `rebutton` [ 14 pts ]

*sample solution: 28 line(s) of code*
Write a recursive function named `max_coin_path` that searches a 2D grid of "coins", an STL vector of STL vector of non-negative integers, for a path back to the origin (0,0). In walking from the start location (lower right corner of grid) to the origin (upper left corner), the path is only allowed to move up or left one grid space at a time. The goal is to find a path that maximizes the sum of the coins along the path. The function should return the maximum sum.

For the example shown above right, the path

(3,4) (3,3) (3,2) (2,2) (2,1) (1,1) (1,0) (0,0)

collects coins with values 1 + 2 + 1 = 4, which is the maximum coin sum that can be achieved on this grid. The path achieving that sum should be stored in the second argument passed to the function, an STL list of Locations named `path`. Note: there are a few similar paths that have the same sum. Your function may return any of these optimal paths.

### 4.1 Usage [2 pts]

You will implement the `max_coin_path` on the next page. But first, complete the initial call to the `max_coin_path` function below. Assume `grid` has already been initialized; for example, with the data shown above. What additional information does your function need to get started?

```cpp
std::list<Location> path;
int max_coin_sum = max_coin_path(grid, path, );
```

### 4.2 Algorithm Analysis [5 pts]

Assume that the grid width and height are `w` and `h` respectively, the number of non-zero coins in the grid is `c`, and the value of the maximum coin is `m`. What is the Big O Notation for the running time of your answer on the next page? Write three to four concise and well-written sentences justifying your answer.
4.3 Implementation [ 16 pts ]

Now implement the `max_coin_path` function. Remember: it should be recursive.

*sample solution: 27 line(s) of code*
5 Linked Tube Repair [ / 33 ]

Alyssa P. Hacker is working on a modified linked list that is both two-dimensional and circular. A small sample with \( \text{height}=3 \) and \( \text{circumference}=4 \) is shown below. Each templated Node has pointers to its 4 neighbors. The top and bottom edges of the tube structure have NULL pointers. But the left and right edges wrap around, like a circularly linked list. This cylindrical tube structure may have any number of nodes for its height and its circumference.

5.1 Tube repair Diagram [ / 4 ]

First Alyssa wants to tackle the challenge of repairing a hole in the structure. Assume a single Node is missing from the structure, and we have a pointer \( n \) to the Node immediately to the left of the hole. Modify the diagram below to show all of the necessary edits for a call to \( \text{repair}(n,7) \);

```
template <class T>
class Node {
public:
    // REPRESENTATION
    T value;
    Node<T> *up;
    Node<T> *down;
    Node<T> *left;
    Node<T> *right;
};
```

5.2 Thinking about Tube repair Complexity [ / 3 ]

The \( \text{repair} \) function should have constant running time in most cases. Describe an example structure with a single missing Node that can be repaired, but not in constant time. Write 2-3 concise and well-written sentences. \textit{You may want to complete the implementation on the next page before answering.}
5.3 Tube repair Implementation

Now, implement repair, which takes 2 arguments: a pointer to the Node immediately to the left of the hole and the value to be stored in the hole. You may assume a single Node is missing from the structure.

sample solution: 26 line(s) of code
Now write `destroyTube` (and any necessary helper functions) to clean up the heap memory associated with this structure. The function should take a single argument, a pointer to any `Node` in the structure. You may assume the structure has no holes or other errors. You cannot use a `for` or `while` loop.

`sample solution: 17 line(s) of code`
Complete the Vec assignment operator implementation below, while minimizing wasted heap memory. Assume the allocator is most efficient when all heap allocations are powers of two (1, 2, 4, 8, 16, etc.)

```cpp
1 template <class T>
2 Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
3     if (this != &v) {
4         delete m_data;
5         m_size = v.m_size;
6         m_alloc = v.m_alloc;
7         m_data = new T[m_size];
8         for (int i = 0; i < m_size; ++i) {
9             m_data[i] = v.m_data[i];
10         }
11     }
12     return *this;
13 }
```

Add code below to perform a simple test of the assignment operator:

```cpp
Vec<double> v; v.push_back(3.14159); v.push_back(6.02); v.push_back(2.71828);
```

Is line 12 necessary? Continue your testing code above with a test that would break if line 12 was omitted.

What is the purpose of line 3? Write code for a test that would break if lines 3 and 10 were omitted.
Write a function `embellish` that modifies its single argument, `sentence` (an STL list of STL strings), adding the word “very” in front of “pretty” and adding “with a wet nose” after “grey puppy”. For example:

```
the pretty kitty sat next to a grey puppy in a pretty garden
```

Should become:

```
the very pretty kitty sat next to a grey puppy with a wet nose in a very pretty garden
```

**sample solution:** 20 line(s) of code

If there are \( w \) words in the input sentence, what is the worst case Big O Notation for this function? If we switched each STL list to STL vector in the above function, what is the Big O Notation?

| STL list: | STL vector: |
Complete **redundant**, which takes a sentence and 2 phrases and replaces all occurrences of the first phrase with the second, shorter phrase. For example “pouring down rain” is replaced with “pouring rain”:

it is pouring down rain so take an umbrella → it is pouring rain so take an umbrella

Or we can just eliminate the word “that” (the replacement phrase is empty):

I knew that there would be late nights when I decided that CS was the career for me → I knew there would be late nights when I decided CS was the career for me

typedef std::list<std::string> words;

```cpp
void redundant(sentence, phrase, replace) {
    // sample solution: 19 line(s) of code
}
```
Write a useful but buggy segment of code (or function) that will compile with no errors but will produce the indicated compilation warning. Put a star ⭐ next to the line of code that will trigger the warning. Write a concise and well-written sentence describing the intended vs. actual (buggy) behavior of the code.

warning: comparison of integers of different signs: 'int' and 'unsigned int'

warning: control reaches / may reach end of non-void function

warning: variable is uninitialized when used here / in this function

warning: returning reference to local temporary object / reference to stack memory associated with a local variable returned
Ben Bitdiddle wrote the following code fragment to manage his personal information.

```cpp
std::ifstream istr("my_information.txt");
std::string s;
std::vector<std::string> data;
while (istr >> s) { data.push_back(s); }
std::vector<std::string>::iterator password = data.begin()+4;
data.push_back("credit_card:");
data.push_back("1234-5678-8765-4321");
data[4] = "qwerty";
std::cout << "my password is: " << *password << std::endl;
```

Write “True” in the box next to each true statement. Leave the boxes next to the false statements empty.

- Lines 2 & 3 will produce an “uninitialized read” error when run under gdb or lldb.
- Line 5 is not a valid way to initialize an iterator.
- Ben’s credit card information is not saved back to the file.
- This program might behave differently if re-run on this computer or another computer.
- A memory debugger might detect an “unaddressable access of freed memory” error on Line 9.
- If we move lines 6 & 7 after line 9, this code fragment will run without memory errors.
- This code contains memory leaks that can be detected by Dr. Memory or Valgrind.
- These password choices disqualify Ben from any job in computer security.
Eva Lu Ator is working on her capstone project to manage physical storage facilities. She’s mapped out the overall design and started implementation of the two classes.

```cpp
class Box {
public:
    Box(int w, int d, int h) :
        width(w), depth(d), height(h) {}
    int width;
    int depth;
    int height;
};

Storage storage(4,3,2);
assert (storage.available_space() == 24);
Box *a = new Box(2,2,2);
assert (storage.add(a,0,0,0));
Box *b = new Box(3,2,1);
assert (!storage.add(b,2,0,0));
delete b;
Box *b_rotated = new Box(2,3,1);
assert (storage.add(b_rotated,2,0,0));
Box *c = new Box(1,1,1);
assert (storage.add(c,2,0,1));
assert (storage.available_space() == 9);
```

```cpp
class Storage {
public:
    Storage(int w, int d, int h);
    // FILL IN FOR PART 1
    bool add(Box *b, int w, int d, int h);
    int available_space();
private:
    void remove(Box *b, int w, int d, int h);
    Box ****data;
    int width;
    int depth;
    int height;
};

bool Storage::add (Box *b, int w, int d, int h) {
    for (int i = w; i < w+b->width; i++) {
        if (i >= width) return false;
        for (int j = d; j < d+b->depth; j++) {
            if (j >= depth) return false;
            for (int k = h; k < h+b->height; k++) {
                if (k >= height) return false;
                if (data[i][j][k] != NULL) return false;
            }
        }
    }
    for (int i = w; i < w+b->width; i++) {
        for (int j = d; j < d+b->depth; j++) {
            for (int k = h; k < h+b->height; k++) {
                data[i][j][k] = b;
            }
        }
    }
    return true;
}
```
11.1 Missing functions from Storage Class Declaration [ / 5 ]

Her friend Ben Bitdiddle doesn’t remember much from Data Structures, but he reminds her that classes with dynamically-allocated memory need a few key functions. Fill in the missing prototypes for PART 1.

11.2 Storage Destructor [ / 20 ]

Eva explains to Ben that the private remove member function will be useful in implementing the destructor. First write the remove member function:

*sample solution: 10 line(s) of code*

Now write the Storage class destructor:

*sample solution: 14 line(s) of code*
12 Transpose Linked Grid [ / 27 ]

Louis B. Reasoner is working on a new member function for our Homework 5 Linked Grid named `transpose`. This function should mirror or flip the elements along the diagonal. Here’s a sample grid with integer data and how it prints before and after a call to `transpose`:

```cpp
grid.print();
std::cout << std::endl;
grid.transpose();
grid.print();
```

<table>
<thead>
<tr>
<th>1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5</td>
</tr>
<tr>
<td>9 10 11 12</td>
</tr>
</tbody>
</table>

| 1 8 9 |
| 2 7 10 |
| 3 6 11 |
| 4 5 12 |

12.1 Diagram [ / 7 ]

First neatly modify the diagram of this smaller grid below to show all of the necessary edits that must be performed by a call to `transpose()`.

```
template <class T>
class Node {
public:
    // REPRESENTATION
    T value;
    Node<T> *up;
    Node<T> *down;
    Node<T> *left;
    Node<T> *right;
};
```

12.2 Complexity Analysis [ / 5 ]

What is the Big 'O' Notation for the running time of the `transpose()` member function? Assume the grid width is \( w \) and the height is \( h \). Write 1-2 concise and well-written sentences justifying your answer. You probably want to complete the implementation on the next page before answering.
12.3 Implementation

Louis has suggested that we first implement a helper non-member function named `swap`, which will make the implementation of `transpose` more concise.

```
sample solution: 5 line(s) of code
```

Now implement `transpose`, as it would appear outside of the `Grid` class declaration.

```
sample solution: 16 line(s) of code
```
Organizing Words [ / 30 ]

Alyssa P. Hacker is working on a program to clean up a dataset of words. The task is to write a function named `organize_words` that takes in an STL vector of STL lists of words (STL strings). The function should organize the words into groups by word length, and ensure that the words are sorted within each group. Many or most of the words will already be in the right place. That is, they will already be in the slot of the vector that matches the length of the word. And the neighboring words in each slot/list will already be mostly alphabetized.

For example, given the data shown on the left, your implementation should move the four misplaced words to produce the data shown on the right.

```
0
1 diamond
2
3 gem malachite
4 jade opal rock ruby
5 geode pearl talc stone topaz
6 garnet quartz gypsum
7 amethyst azurite emerald
8 fluorite sapphire
9
```

```
0
1
2
3 gem
4 jade opal rock ruby talc
5 geode pearl stone topaz
6 garnet gypsum quartz
7 azurite diamond emerald
8 amethyst fluorite sapphire
9 malachite
```

To make the problem a little more “fun”, you are NOT ALLOWED to use:

- the STL vector subscript/indexing operator, [ ], or .at(),
- the STL sort function, or
- any of the push or pop functions on vector or list.

You may assume that the initial vector has at least as many slots as the longest word in the structure.

13.1 Complexity Analysis - Big 'O' Notation [ / 6 ]

Once you’ve finished your implementation on the next pages, analyze the running time of your solution. Assume there are \( w \) total words in the whole structure, \( v \) slots in the vector, a maximum of \( m \) words per list, and \( x \) words are misplaced and need to be moved. Write 2-3 concise and well-written sentences justifying your answer.
Alyssa suggests writing a helper function named `place` that will place a word in the correct location in the structure. Work within the provided framework below. Do not add any additional `for` or `while` loops.

```c
void place(
    // your code here
) {

    // sample solution: 2 line(s) of code
}

while ( ) {
    // sample solution: 3 line(s) of code

    while ( ) {
        // sample solution: 5 line(s) of code
    }
}
}
```

```c
// sample solution: 5 line(s) of code
```
13.3 Organize Implementation

And now write the `organize` function, which calls the `place` function. Again, work within the provided framework below and do not add any additional `for` or `while` loops.

```c
void organize_words() {
    sample solution: 2 line(s) of code

    while ( ) {
        sample solution: 2 line(s) of code

        while ( ) {
            sample solution: 8 line(s) of code
        }
    }
}
```

```c
}
```

```c
}
```

```c
}
```

```c
}
```
Ben Bitdiddle was inspired by the recursive merge sort example from Data Structures lecture and proposes it as a guide to compute the smallest interval that contains a collection of floating point numbers (e.g., the minimum and maximum). Implement Ben's idea, a recursive function named `compute_interval` that takes in an STL `vector` of `floats` and returns an `Interval` object.

For example: 6.2 4.3 10.4 2.5 8.4 1.5 3.7 → [1.5, 10.4]

```cpp
class Interval {
public:
    Interval(float i, float j) : min(i), max(j) {}
    float min;
    float max;
};

sample solution: 12 line(s) of code
```

Without resorting to personal insults, explain in two or three concise and well-written sentences why Ben’s idea isn’t going to result in significant performance improvements. Be technical.