CSCI-1200 Data Structures
Test 2 — Practice Problem Solutions

1 Linked Tube Repair [/ 33]

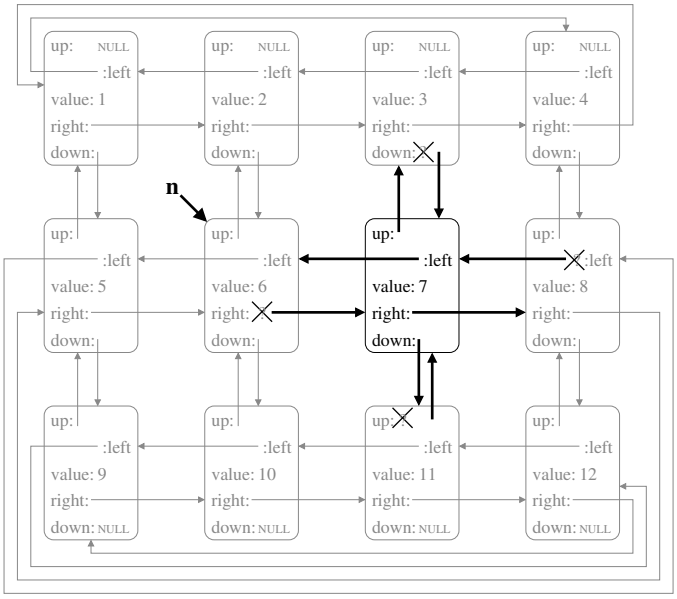
```
template <class T>
class Node {
public:
  // REPRESENTATION
  T value;
  Node<T> *up;
  Node<T> *down;
  Node<T> *left;
  Node<T> *right;
};
```

Alyssa P. Hacker is working on a modified linked list that is both two-dimensional and circular. A small sample with *height*=3 and *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.

1.1 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 `repair(n,7);`

Solution: The diagram should be neatly edited to add a new Node box. The value 7 should be stored in the box. The 4 pointers from the new box should point at the appropriate neighbors and the 4 neighbors should point back at the new Node.



1.2 Thinking about Tube `repair` Complexity [/ 3]

The `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. *You may want to complete the implementation on the next page before answering.*

Solution: The repair can be solved in constant time by walking around the hole. Special case code must be written if the hole is at the top or bottom edge of the tube – but both cases can be solved in constant time. However, if the height of the structure is 1, The only way to get to the right edge of the hole is to walk to the left, through all the nodes in the circumference. $O(c)$, where c = the circumference.

1.3 Tube `repair` Implementation [/ 13]

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.

Solution:

```
template <class T> void repair(Node<T>* n, const T &value) {
  Node<T>* tmp = new Node<T>;
  tmp->value = value;
  // left
  tmp->left = n;
  tmp->left->right = tmp;
  // up
  if (n->up != NULL) {
    tmp->up = n->up->right;
    tmp->up->down = tmp;
    tmp->right = n->up->right->right->down;
  } else {
    // special case, hole at the top edge
    tmp->up = NULL;
  }
  // down
  if (n->down != NULL) {
    tmp->down = n->down->right;
    tmp->down->up = tmp;
    tmp->right = n->down->right->right->up;
  } else {
    // special case, hole at the bottom edge
    tmp->down = NULL;
  }
  // right
  if (n->up == NULL && n->down == NULL) {
    // non-constant time special case: height == 1
    Node<T>* tmp2 = n;
    while (tmp2->left != NULL) tmp2 = tmp2->left;
    tmp->right = tmp2;
  }
  tmp->right->left = tmp;
}
```

1.4 Non-Iterative `destroy_tube` Implementation [/ 13]

Now write `destroy_tube` (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.

Solution:

```
template <class T> void destroy_row(Node<T>* n) {
  if (n->right != NULL)
    destroy_row(n->right);
  delete n;
}

template <class T> void destroy_rows(Node<T>* n) {
  if (n->down != NULL)
    destroy_rows(n->down);
  n->left->right = NULL;
  destroy_row(n);
}

template <class T> void destroy_tube(Node<T>* n) {
  if (n->up != NULL)
    destroy_tube(n->up);
  else
    destroy_rows(n);
}
```

Note: The following attempt at a flood fill solution does not work. It will attempt to free the same Nodes multiple times.

```
template <class T>
void destroy_tube_buggy(Node<T>* n) {
  Node<T>* u = n->up;
  Node<T>* d = n->down;
  Node<T> *l = n->left;
  Node<T> *r = n->right;
  if (u != NULL) u->down = NULL;
  if (d != NULL) d->up = NULL;
  if (l != NULL) l->right = NULL;
  if (r != NULL) r->left = NULL;
  delete n;
  if (u != NULL) destroy_tube_buggy(u);
  if (d != NULL) destroy_tube_buggy(d);
  if (l != NULL) destroy_tube_buggy(l);
  if (r != NULL) destroy_tube_buggy(r);
}
```

2 Rehashing the `Vec` Assignment Operator [/ 15]

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.)

Solution:

```
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 = pow(2,ceil(log2(m_size)));
7       m_data = new T[m_alloc];
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:

Solution:

```
Vec<double> v; v.push_back(3.14159); v.push_back(6.02); v.push_back(2.71828);
Vec<double> v2; v2.push_back(1.0); v2.push_back(2.0);
assert (v2.size() != v.size () && v2[1] != v[1]);
v2 = v;
assert (v2.size() == v.size () && v2[1] == v[1]);

// NOTE: the following line calls the copy constructor, not the assignment operator:
//   Vec<double> another = v;
// It does exactly the same thing as:
//   Vec<double> another(v);
```

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

Solution: We need the return value if the result of the assignment is stored or used as part of a larger expression.

```
Vec<double> v3; v3.push_back(100.0);
v[1] = 1.414;
v3 = v2 = v;
assert (v3.size() == v2.size() && v3.size() == v.size() && v3[1] == v2[1] && v3[1] == v[1]);

// NOTE: The rules for precedence of the assignment operator go right to left.
// The statement above is the same as:
//   v3 = (v2 = v);
```

What is the purpose of line 3? Write code for a test that would break if lines 3 and 10 were omitted.

Solution: It prevents errors for self-assignment. If we delete the object, we have nothing to use as our example to copy from!

```
v = v;
assert (v.size() == 3 && v[2] == 2.71828);
```

3 Essay Revision: Embellishment [/ 14]

Write a function `embellish` that modifies its single argument, `sentence` (an STL `list` of STL `string`s), 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
```

Solution:

```
void embellish(std::list<std::string> &sentence) {
  std::list<std::string>::iterator itr = sentence.begin();
  std::string prev = "";
  while (itr != sentence.end()) {
    std::string word = *itr;
    if (word == "pretty") {
      sentence.insert(itr,"very");
      itr++;
    } else if (prev == "grey" && word == "puppy") {
      itr++;
      sentence.insert(itr,"with");
      sentence.insert(itr,"a");
      sentence.insert(itr,"wet");
      sentence.insert(itr,"nose");
    } else {
      itr++;
    }
    prev = word;
  }
}
```

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?

Solution: We do an outer loop over all w words. If every word is "`puppy`" we will do 4 inserts for every word. STL `list insert` is an O(1) function. So for `list` the overall runtime is $O(w + w * 4 * 1) = O(w)$. However, STL `vector insert` is an $O(w)$ operation, so this function (as written) will be $O(w + w * 4 * w) = O(w^2)$.

4 Essay Revision: Redundant Phrases [/ 15]

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`

Solution:

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

void redundant(words &sentence, const words &phrase, const words &replace) {
  assert (phrase.size() > replace.size());
  words::iterator s = sentence.begin();
  while (s != sentence.end()) {
    // see if this word is the start of the phrase
    bool match = true;
    words::iterator s2 = s;
    for (words::const_iterator p = phrase.begin(); p != phrase.end(); p++) {
      if (s2 == sentence.end() || *s2 != *p) {
        match = false;
        break;
      }
      s2++;
    }
    // move to the next word if this isn't a match
    if (!match) { s++; continue; }
    // otherwise, remove an appropriate number of words
    for (int i = 0; i < phrase.size() - replace.size(); i++) {
      s = sentence.erase(s);
    }
    // and overwrite the words with the replacement phrase
    for (words::const_iterator r = replace.begin(); r != replace.end(); r++, s++) {
      *s = *r;
    }
  }
}
```

5 Don't Ignore Compilation Warnings! [/ 15]

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'
```

Solution: This code is attempting to print a vector in reverse order. But it will go into an infinite loop because the condition is always true.

```
    int zero = 0;
*   for (unsigned int i = vec.size()-1; i >= zero; i--) {
      std::cout << vec[i] << std::endl;
    }
```

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

Solution: The function below does not handle the case when a == b. Essentially the return value is uninitialized in this case and could be anything.

```
    int larger_value (int a, int b) {
     if (a > b) return a;
     if (b > a) return b;
*   }
```

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

Solution: We've forgotten to initialize the sum variable, so unfortunately all of the work is wasted. The final value of sum could be anything.

```
    int sum;
    for (int i = 0; i < vec.size(); i++) {
*     sum += vec[i];
    }
```

```
warning: returning reference to local temporary object / reference to stack memory
associated with a local variable returned
```

Solution: The code below reads a file into a vector of strings, and wants to save memory & copying by returning this data by reference, but since this local variable is on the stack it disappears as soon as we go out of scope (leave the function). We cannot safely return this vector by reference.

```
    std::vector<std::string>& load_data(std::ifstream &istr) {
      std::vector<std::string> answer;
      std::string s;
      while (istr >> s) { answer.push_back(s); }
*     return answer;
    }
```

```
warning: expression result unused / expression has no effect
```

Solution: The code below is attempting to add one to every element in the vector... but nothing changes! Instead of addition, the code should either use the += operator or the ++ operator.

```
    for (unsigned int i = 0; i < vec.size(); i++) {
*     vec[i] + 1;
    }
```

```
warning: unused variable / unused parameter
```

Solution: We intended to use the count variable to count the number of values we read from the file and divide the sum by this value. But the code is incomplete! Whoops!

```
    int compute_average(std::ifstream &istr) {
      int sum = 0;
*     int count = 0;
      int x;
      while (istr >> x) {
        sum += x;
      }
      return sum;
    }
```

6 Cyber Insecurity [/ 5]

Ben Bitdiddle wrote the following code fragment to manage his personal information.

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

my_information.txt

```
name: Ben Bitdiddle
password: pa$$word
SSN: 123-45-6789
```

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

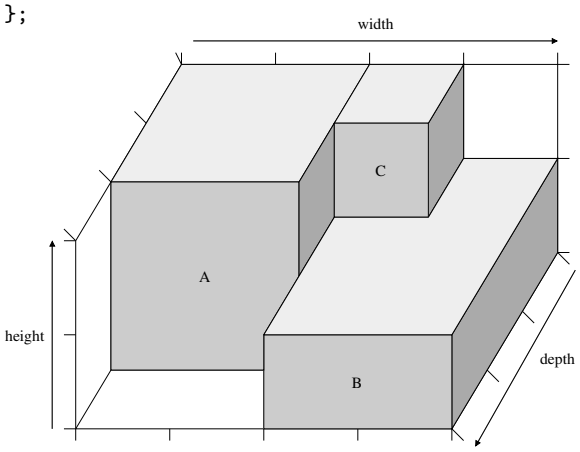
(false)	Lines 2 & 3 will produce an "uninitialized read" error when run under `gdb` or `lldb`.
(false)	Line 5 is not a valid way to initialize an iterator.
TRUE	Ben's credit card information is not saved back to the file.
TRUE	This program might behave differently if re-run on this computer or another computer.
TRUE	A memory debugger might detect an "unaddressable access of freed memory" error on Line 9.
TRUE	If we move lines 6 & 7 after line 9, this code fragment will run without memory errors.
(false)	This code contains memory leaks that can be detected by Dr. Memory or Valgrind.
TRUE	These password choices disqualify Ben from any job in computer security.

Solution: The core problem with the code above is that an iterator is attached to a vector (line 5), then the vector is edited via calls to `push_back` (lines 6 & 7), and afterwards the iterator is used to access an element in the vector (line 9). Vector iterators may be invalidated (and you should assume they are invalidated) after any edit that might reallocate/relocate the data vector (e.g., `push_back`, `insert`, and `resize`). Note that calls to `erase` are also dangerous because the data has been shifted and the iterator may point to different data (depending whether it was before or after the erase point).

7 Boxy Storage Solutions [/ 25]

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.

```
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);
```

```
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;
}
```

7.1 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`.

Solution:

```
  Storage(const Storage &s);
  Storage& operator=(const Storage &s);
  ~Storage();
```

Note: The *convention* for C/C++ is that the assignment operator returns an object. It should return `this` rather than the right-hand operand, `s`. It should return a reference (to avoid unneccessary copying). Whether it returns a const or a non-const is debateable. Whoever called this function must have had write access to the `this` object, so returning a non-const reference is safe.

7.2 `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:

Solution:

```
void Storage::remove(Box *b, int w, int d, int h) {
  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++) {
        assert (data[i][j][k] == b);
        data[i][j][k] = NULL;
      }
    }
  }
  delete b;
}
```

Now write the `Storage` class destructor:

Solution:

```
Storage::~Storage() {
  for (int w = 0; w < width; w++) {
    for (int d = 0; d < depth; d++) {
      for (int h = 0; h < height; h++) {
        if (data[w][d][h] != NULL) {
          remove(data[w][d][h],w,d,h);
        }
      }
      delete [] data[w][d];
    }
    delete [] data[w];
  }
  delete [] data;
}
```

8 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 `print`s before and after a call to `transpose`:

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

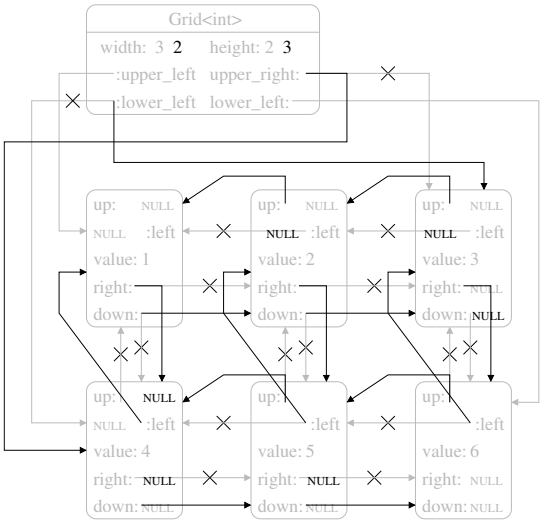
```
1   2   3   4
8   7   6   5
9  10  11  12

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

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

8.1 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()`.



Solution:

8.2 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.*

Solution: We need to update a few variables in the Grid manager class, and we need to visit every node in the structure and modify the links. If we do things in an organized manner we can do so with a small (constant) number of helper variables, and it does not require expensive logic. Number of nodes $\rightarrow$ Overall: $O(w \times h)$.

8.3 Implementation [/ 15]

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

Solution:

```
template <class T> void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

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

Solution:

```
template <class T> void Grid<T>::transpose() {
  Node<T> *row = upper_left;
  while (row != NULL) {
    Node<T> *next_row = row->down;
    Node<T> *element = row;
    while (element != NULL) {
      Node<T> *next_element = element->right;
      swap(element->up,element->left);
      swap(element->right,element->down);
      element = next_element;
    }
    row = next_row;
  }
  swap(width,height);
  swap(upper_right,lower_left);
}
```

9 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 `list`s of words (STL `string`s). 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.

9.1 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.

Solution: We need to walk through all of the slots and all of the elements in every slot/list to find all of the misplaced words. The walk is $O(w) = O(v \times m)$. Deciding if a word is misplaced is constant (calculating the length of the word, and deciding if it greater† than the element before and less than the element after). Removing the word is constant. All misplaced words will trigger a walk of the vector $O(v)$, plus a walk of the single list to find the appropriate insertion point, $O(m)$. Inserting the word is constant. Overall: $O(v \times m + x \times (v + m))$. If x is large, and v is small (thus x and m are close to w), then the running time is $O(w^2)$ – that is, an inefficient insertion sort algorithm. But if x is quite small (as described in the problem instructions), then the running time is closer to $O(m \times v) = O(w)$.

†Note: comparing two very long words to determine which comes first alphabetically is actually linear $O(v)$. So the `organize` walk is actually $O(w \times v) = O(v^2 \times m)$. And the `place` walk is actually $O(m \times v)$. Overall $O(v^2 \times m + x \times (v + m \times v))$. Simplfied: $O(w^2)$ for large x, small v. And $O(w \times v)$ for small x.

9.2 Helper Function Implementation [/ 12]

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.

Solution:

```
void place(std::vector<std::list<std::string> > &words, const std::string& word) {
  int count = 0;
  std::vector<std::list<std::string> >::iterator itr = words.begin();
  while (itr != words.end()) {
    if (word.size() == count) {
      std::list<std::string>::iterator itr2 = (*itr).begin();
      while (itr2 != (*itr).end()) {
        if (word < *itr2) {
          (*itr).insert(itr2,word);
          return;
        }
        itr2++;
      }
      (*itr).insert(itr2,word);
      return;
    }
    itr++;
    count++;
  }
}
```

9.3 Organize Implementation [/ 12]

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.

Solution:

```
void organize_words(std::vector<std::list<std::string> > &words) {
  int count = 0;
  std::vector<std::list<std::string> >::iterator itr = words.begin();
  while (itr != words.end()) {
    std::list<std::string>::iterator itr2 = (*itr).begin();
    std::string last = "";
    while (itr2 != (*itr).end()) {
      std::string word = *itr2;
      if (word.size() != count || (last != "" && word < last)) {
        itr2 = (*itr).erase(itr2);
        place(words,word);
      } else {
        last = *itr2;
        itr2++;
      }
    }
    itr++;
    count++;
  }
}
```

10 Merge-Spiration: Recursive Interval Computation [/ 15]

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]`

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

Solution:

```
Interval compute_interval(const std::vector<float> &data, int i, int j) {
  // cannot compute an interval for no values
  assert (i <= j);
  if (i == j) return Interval(data[i],data[i]);
  int mid = (i+j)/2;
  Interval low = compute_interval(data,i,mid);
  Interval high = compute_interval(data,mid+1,j);
  if (low.min > high.min) low.min = high.min;
  if (low.max < high.max) low.max = high.max;
  return low;
}

Interval compute_interval(const std::vector<float> &data) {
  return compute_interval(data,0,data.size()-1);
}
```

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.

Solution: Calculating the max and min of an (unsorted) sequence of numbers requires only a linear scan/visit of the elements, comparing each element to the current min & max. n elements and $2n$ comparisons, so overall $= O(n)$. Ben's algorithm will also visit each element once, and at each recursive call it will do 2 comparisons. If we draw out the tree we see that we have n recursive calls. So the algorithms are basically equivalent in Big O Notation for performance / running time. However, function calls are expensive (more expensive than a simple loop), so in practice the running time of Ben's recursive algorithm will probably be slower (but it's not terrible).

11 How many DS students to change a lightbulb? [/ 38]

In this problem you will complete the implementation of two new classes named `Bulb` and `Lamp`. We begin with an example of how these classes are used.

First, we create a new lamp that will hold 3 bulbs and make a note of the manufacturer's recommended bulb: a 60 watt bulb with an estimated lifetime of 300 hours from Phillips. Note that initially this lamp has no bulbs installed. We install one of manufacturer's recommended bulbs and use the lamp (turn it "on") for a total of 50 hours.

```
Lamp floorlamp(Bulb(60,300,"Phillips"),3);
bool success;
success = floorlamp.install(); assert(success);
floorlamp.On(50);
assert (floorlamp.getTotalWattage() == 60);
```

Next, we attempt to install 3 bulbs, another of the manufacturer's recommended bulbs, and then two other brands of bulbs. The installation of the 3rd bulb made by Sylvania fails because there are no available sockets slots in the lamp and no bulbs are burnt out and need replacement.

```
success = floorlamp.install(); assert(success);
success = floorlamp.install(Bulb(40,120,"GE")); assert(success);
success = floorlamp.install(Bulb(120,500,"Sylvania")); assert(!success);
```

We then use the lamp for another 100 hours. Once the wattage drops (due to a burnt out bulb), we again try to install the Sylvania bulb and it is successful.

```
floorlamp.On(100);
assert (floorlamp.getTotalWattage() == 160);
floorlamp.On(50);
```

```
assert (floorlamp.getTotalWattage() == 120);
success = floorlamp.install(Bulb(120,500,"Sylvania")); assert(success);
assert (floorlamp.getTotalWattage() == 240);
```

Finally, we create a duplicate lamp. Note that when we do this, we match the bulbs currently installed in the original lamp, but the bulbs installed in the new lamp are brand new (and unused).

```
Lamp another(floorlamp);
assert (floorlamp.getTotalWattage() == another.getTotalWattage());
for (int i = 0; i < 10; i++) {
  floorlamp.On(50);
  another.On(50);
  std::cout << "compare " << floorlamp.getTotalWattage() << " "
                          << another.getTotalWattage() << std::endl;
}
```

Which results in this output:

```
compare 240 240
compare 240 240
compare 180 240
compare 120 240
compare 120 240
compare 120 240
compare 120 120
compare 120 120
compare 120 120
compare 120 120
```

11.1 `Bulb` Class Declaration [/ 14]

The `Bulb` class is missing only one function. *You will need to read the rest of the problem to determine what's missing.* Fill in the missing function – implement the function right here, within the class declaration.

```
class Bulb {
public:
  // constructors
  Bulb(int w, int l, const std::string &b) :
    wattage(w),lifetime(l),hours_used(0),brand(b) {}
```

```
Bulb(const Bulb& b) :
   wattage(b.wattage),lifetime(b.lifetime),hours_used(0),brand(b.brand) {}
```

```
  // accessors
  int getWattage() const { return wattage; }
  bool burntOut() const { return hours_used > lifetime; }
  const std::string& getBrand() const { return brand; }
  // modifier
  void On(int h) { hours_used += h; }
private:
  // representation
  int wattage;
  int lifetime;
  int hours_used;
  std::string brand;
};
```

11.2 `Lamp` Class Declaration [/ 14]

The `Lamp` class has a few more missing pieces. *Read through the rest of the problem before attempting to fill this in.* Write the *prototypes* (not the implementation!) for the four missing functions. You will implement some of these missing functions later. Also, fill in the member variables for the `Lamp` representation. Important: You may not use STL `vector` on this problem.

```
class Lamp {
public:
  // constructors, assignment operator, destructor
```

```
  Lamp(const Bulb& b, int num);
  Lamp(const Lamp &l);
  const Lamp& operator=(const Lamp &l);
  ~Lamp();
```

```
  // accessor
  int getTotalWattage() const;
  // modifiers
  bool install(const Bulb &b = Bulb(0,0,""));
  void On(int h);
private:
  // representation
```

```
  Bulb recommended;
  Bulb** installed;
  int max_bulbs;
```

```
};
```

`Lamp` Class Implementation

Here's the implementation of one of the key member functions of the `Lamp` class.

```
bool Lamp::install(const Bulb &b) {
  // first, let's figure out where to install the bulb
  int which = -1;
  for (int i = 0; i < max_bulbs; i++) {
    // check for an empty socket
    if (installed[i] == NULL) {
      which = i;
      break;
    }
    // or a socket that contains a burnt out bulb
    if (installed[i]->burntOut()) {
      which = i;
      delete installed[i];
      break;
    }
  }
  // return false if we cannot install this bulb
  if (which == -1) return false;
  if (b.getWattage() == 0) {
    // install the manufacturer's recommended bulb type
    installed[which] = new Bulb(recommended);
  } else {
    // install the specified bulb
    installed[which] = new Bulb(b);
  }
  return true;
}
```

On the last two pages of this problem you will implement three important functions for the `Lamp` class, as they would appear outside of the class declaration (in the `lamp.cpp` file) because their implementations are > 1 line of code.

11.3 `Lamp` Constructor [/ 9]

Solution:

```
Lamp::Lamp(const Bulb& b, int num) : recommended(b) {
  installed = new Bulb*[num];
  for (int i = 0; i < num; i++) {
    installed[i] = NULL;
  }
  max_bulbs = num;
}
```

11.4 `Lamp` Destructor [/ 5]

Solution:

```
Lamp::~Lamp() {
  for (int i = 0; i < max_bulbs; i++) {
    // note: this check not necessary, ok to call delete on a NULL ptr
    if (installed[i] != NULL) {
      delete installed[i];
    }
  }
  delete [] installed;
}
```

11.5 `Lamp` Assignment Operator [/ 9]

Solution:

```
const Lamp& Lamp::operator=(const Lamp &l) {
  if (this != &l) {
    for (int i = 0; i < max_bulbs; i++) {
      if (installed[i] != NULL) {
        delete installed[i];
      }
```

```
    }
    delete [] installed;
    max_bulbs = l.max_bulbs;
    recommended = l.recommended;
    installed = new Bulb*[max_bulbs];
    for (int i = 0; i < max_bulbs; i++) {
      if (l.installed[i] == NULL) {
        installed[i] = NULL;
      } else {
        installed[i] = new Bulb(*l.installed[i]);
      }
    }
  }
  return *this;
}
```

12 Singly Linked List Subsequence Sum [/ 18]

```
template <class T>
class Node {
public:
  Node(const T& v)
    : value(v),
      next(NULL) {}
  T value;
  Node* next;
};
```

Write a recursive function named `FindSumStart` that takes the head Node of a singly-linked list storing positive numbers. The function should return a pointer to the Node that begins a subsequence of numbers that ends in the sum of that subsequence. For example, given this sequence: `5 1 4 2 3 9 6 7` the function should return a pointer to the `Node` storing 4, because $4 + 2 + 3 = 9$.

Solution:

```
template <class T> Node<T>* FindSumStart(Node<T>* n) {
  if (n == NULL) {
    return NULL;
  }
  int total = 0;
  Node<T>* tmp = n;
  while (tmp != NULL) {
    if (total == tmp->value) {
      return n;
    }
    total += tmp->value;
    tmp = tmp->next;
  }
  return FindSumStart(n->next);
}
```

Assuming the sequence has n numbers, what is the order notation for the running time of your function?

Solution: $\mathbf{O}(n^2)$

13 Reverse Splice [/ 20]

Write a function named `reverse_splice` that takes in 3 arguments: an STL list named `data` and two iterators `i` and `j`. The function should reverse the order of the data between those iterators. For example, if `data` initially stores this sequence: `1 2 3 4 5 6 7 8 9` and `i` refers to 3 and `j` refers to 7, then after the call `reverse_splice(data,i,j)`, `data` will contain: `1 2 7 6 5 4 3 8 9` , `i` will refer to element 7, and `j` will refer to element 3. Your function should return true if the operation was successful, and false if the request is invalid. Note: Your function may only use a constant amount of additional memory.

Solution:

```
template <class T>
bool reverse_splice(std::list<T> &data,
                    typename std::list<T>::iterator &i,
                    typename std::list<T>::iterator &j) {
  // checking that i comes before j within data
```

```
  if (j == data.end()) {
    return false;
  }
  // slide the splice end iterator forward (off of the last element)
  j++;
  typename std::list<T>::iterator k;
  for (k = i; k != j; k++) {
    if (k == data.end()) return false;
  }
  // use a helper iterator to keep track as we walk between the
  // endpoints of the splice
  k = j;
  while (k != i) {
    // move one element
    data.insert(k,*i);
    k--;
    i = data.erase(i);
  }
  // back the splice end iterator back onto the last element
  j--;
  return true;
}
```

14 Doubly Linked Factorization [/ 17]

```
class Node {
public:
  Node(int v) :
    value(v),
    next(NULL),
    prev(NULL) {}
  int value;
  Node* next;
  Node* prev;
};
```

Write a recursive function named `Factor` that takes in two arguments, pointers to the `head` and `tail` `Node`s of a doubly linked list. This function should look for a non-prime number in the linked list structure, break the `Node` into two `Node`s storing two of its factors, and then return true. If all elements are prime the function returns false. For example, if we start with a 3 element list containing `35 30 28` and repeatedly call Factor:

```
PrintNodes(head);
while (Factor(head,tail)) { PrintNodes(head); }
```

This is the output:

```
35 30 28
5 7 30 28
5 7 2 15 28
5 7 2 3 5 28
5 7 2 3 5 2 14
5 7 2 3 5 2 2 7
5 7 2 3 5 2 2 7
```

You may write a helper function. You do not need to write the `PrintNodes` function.

Solution:

```
bool Factor(Node* &head, Node* &tail, Node* n) {
  // base case
  if (n == NULL) return false;
  // see if this element has any factors
  for (int i = 2; i < n->value; i++) {
    if (n->value % i == 0) {
      // create a new node in front of this one
      Node* tmp = new Node(i);
      // change all of the links
      tmp->prev = n->prev;
      if (n->prev != NULL) {
        tmp->prev->next = tmp;
      }
      tmp->next = n;
      n->prev = tmp;
      n->value = n->value / i;
      // handle the special case of the first node
      if (n == head) head = tmp;
      return true;
} }
  // recurse if we couldn't split this element
  return Factor(head,tail,n->next);
}
// driver function
bool Factor(Node* &head, Node* &tail) {
  return Factor(head,tail,head);
}
```