CSCI-1200 Data Structures
Test 2 — Practice Problem Solutions

1 Flipping & Sorting Words [/ 18]

Finish the implementation of the function `FlipWords` that takes in an *alphabetically sorted* STL `list` of STL `string`s named `words` and modifies the list. The function should remove all palindromes (words that are the same forwards & backwards). The function should insert the flipped (reversed) version of all other words into the list, *in sorted order*. For example this input list:

```
bard civic diva flow pots racecar stop warts
```

Should be changed to contain:

```
avid bard diva drab flow pots stop straw warts wolf
```

You may not use STL `sort`. You may assume the input list does not contain any duplicates. And after calling the `FlipWords` function the list should not contain any duplicates.

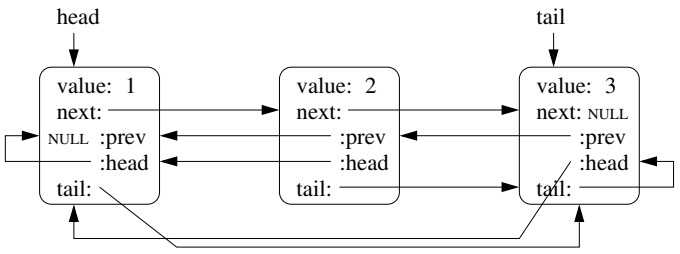
Solution:

```
std::string reverse(std::string &word) {
  std::string answer(word.size(),' ');
  for (int i = 0; i < word.size(); i++) { answer[i] = word[word.size()-1-i]; }
  return answer;
}
void FlipWords(std::list<std::string> &words) {
  std::list<std::string>::iterator current = words.begin();
  while (current!= words.end()) {
    std::string flip = reverse(*current);
    if (flip == *current) {
      current = words.erase(current);
    } else {
      std::list<std::string>::iterator tmp = words.begin();
      while (tmp != words.end() && flip > *tmp) {
        tmp++;
      }
      if (tmp == words.end() || flip != *tmp) {
        words.insert(tmp,flip);
      }
      current++;
    }
  }
}
```

2 "Smart" List Nodes [/ 18]

Ben Bitdiddle thinks he has stumbled on a brilliant idea to make each `Node` of a doubly linked list "smart" and store global information about the list. Each `Node` will have a pointer to the `head` and `tail` `Node`s of the overall list.

```
class Node {
public:
  Node* head;
  Node* tail;
  Node* next;
  Node* prev;
  int value;
};
```



Help him by finishing the implementation of `PushFront` to add a new element to the list. *Note: You should not change the* `value` *inside of any existing* `Node`*s.*

Solution:

```
void PushFront(Node* &head, Node* &tail, int v) {
  Node* tmp = new Node;
  tmp->value = v;
  if (head == NULL) {
```

```
    assert (tail == NULL);
    tmp->next = tmp->prev = NULL;
    tmp->head = tmp->tail = tmp;
    head = tail = tmp;
  } else {
    tmp->prev = NULL;
    tmp->next = head;
    tmp->tail = tail;
    head->prev = tmp;
    head = tmp;
    while (tmp != NULL) {
      tmp->head = head;
      tmp = tmp->next;
    }
  }
}
```

3 Dynamically Allocated Student Schedules [/ 30]

Alyssa P. Hacker has joined the Rensselaer Center for Open Source Software and is working on a program to help students manage their schedules over their time at RPI. She will use a two dimensional array to store courses taken each term. The declaration for two key classes is shown on the right:

Alyssa's program assumes that all undergraduate RPI degree programs require students to take 32 4-credit courses. She also assumes that each specific student takes the same number of courses per term throughout their time at RPI.

Your task is to implement the critical functions for this class with dynamically allocated memory, as they would appear in the `Student.cpp` file. Make sure to use the private helper functions as appropriate so your code is concise.

A few examples of usage are shown below.

```
class Course {
public:
  Course(const std::string &p="XXXX", int n=1000)
    : prefix(p), num(n) {}
  /* member functions omitted */
private:
  std::string prefix;
  int num;
};
class Student {
public:
  Student();
  Student(int courses_per_term_);
  Student(const Student& s);
  const Student& operator=(const Student& s);
  ~Student();
  int numTerms() const { return num_terms; }
  const Course& getCourse(int t, int c) const
      { return data[t][c]; }
  /* additional member functions omitted */
private:
  void initialize();
  void copy(const Student& s);
  void destroy();
  int num_terms;
  int courses_per_term;
  Course** data;
};
```

```
// a typical student takes 4 courses per term for 8 terms
Student regular;          assert (regular.numTerms() == 8);
// if a student takes 5 courses per term, they can finish in 3.5 years
Student overachiever(5);  assert (overachiever.numTerms() == 7);
// students who take 3 courses per term will require 5.5 years
Student supersenior(3);   assert (supersenior.numTerms() == 11);
/* details of how courses are scheduled omitted */
```

Solution:

```
Student::Student() {
  num_terms = 8;
  courses_per_term = 4;
  initialize();
}
```

```
Student::Student(int courses_per_term_) {
  courses_per_term = courses_per_term_;
  num_terms = ceil(32 / float(courses_per_term));
  initialize();
}
Student::Student(const Student& s) {
  copy(s);
}

const Student& Student::operator=(const Student& s) {
  if (this != &s) {
    destroy();
    copy(s);
  }
  return *this;
}

Student::~Student() {
  destroy();
}
void Student::initialize() {
  data = new Course* [num_terms];
  for (int i = 0; i < num_terms; i++) {
    data[i] = new Course[courses_per_term];
  }
}

void Student::copy(const Student& s) {
  courses_per_term = s.courses_per_term;
  num_terms = s.num_terms;
  initialize();
  for (int i = 0; i < num_terms; i++) {
    for (int j = 0; j < courses_per_term; j++) {
      data[i][j] = s.data[i][j];
    }
  }
}

void Student::destroy() {
  for (int i = 0; i < num_terms; i++) {
    delete [] data[i];
  }
  delete [] data;
}
```

4 Reverse Iterators [/ 10]

Complete the function below named `reverse` that takes in an STL `list` as its only argument and returns an STL `vector` that contains the same list except in reverse order. You should use a *reverse iterator* and you may not use `push_back`.

Solution:

```
template <class T>
std::vector<T> reverse(const std::list<T> &my_list) {
  std::vector<T> answer (my_list.size());
  int i = 0;
  typename std::list<T>::const_reverse_iterator itr = my_list.rbegin();
  while (itr != my_list.rend()) {
    answer[i] = *itr;
    i++;
    itr++;
  }
  return answer;
}
```

5 Order Notation [/ 5]

Rank these 6 order notation formula from fastest(1) to slowest(6).

Solution: 1 $O(8 \cdot s \cdot w \cdot h)$
Solution: 4 $O((s \cdot w \cdot h)^8)$
Solution: 6 $O((8 \cdot w \cdot h)^s)$
Solution: 5 $O(w \cdot h \cdot 8^s)$
Solution: 2 or 3 $O((s + w \cdot h)^8)$
Solution: 2 or 3 $O(w \cdot h \cdot s^8)$

NOTE: The ordering of the '2' vs. '3' depends on the relative size of the variables h, w, and s.
If $w = h = s$ **:** $(w + w \cdot w)^8 = w^{16} > w \cdot w \cdot w^8 = w^{10}$.
If $w = h$ & $s = w^2$ **:** $(w^2 + w \cdot w)^8 = w^{16} < w \cdot w \cdot (w^2)^8 = w^{18}$.

6 Dynamic Tetris Arrays [/26]

6.1 HW3 `Tetris` Implementation Order Notation [/6]

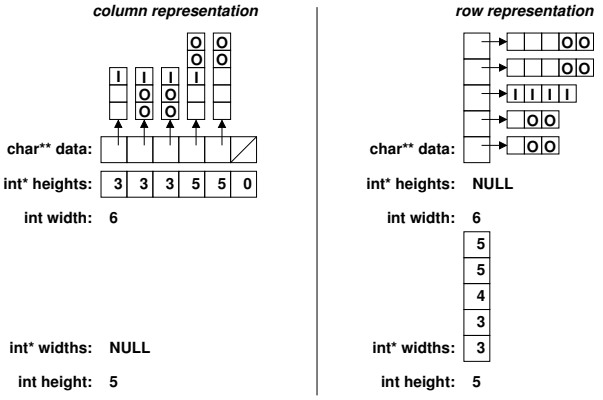
Grading Note: -1.5pts each unanswered or incorrect.

Match up the `Tetris` class member functions from HW3 with the appropriate order notation, where w is the width of the board and h is the maximum height of any column. Assume the solution is efficient, but uses only the 3 member variables specified in the original assignment (`data`, `heights`, and `width`).
Note: Some letters may be used more than once or not at all.

Solution:	**c**	`void add_piece(char piece,int rotation,int position);`	a) $O(1)$
	a	`int get_width();`	b) $O(w)$
	e	`int remove_full_rows();`	c) $O(h)$
	b	`int get_max_height();`	d) $O(w+h)$
	e	`void destroy();`	e) $O(w*h)$

6.2 `Tetris` Representation Conversion [/20]

Now let's revisit the details of the dynamic memory representation for the game of Tetris. Your task is to convert a Tetris board from the *column representation* we used for HW3 to a *row representation*. In addition to the three member variables in our HW3 Tetris class: `data`, `heights`, and `width`, we add 2 additional member variables: `widths` and `height`. In the column representation we don't need the `widths` variable, so it is set to `NULL`. Each time the board is modified to add Tetris pieces or score full rows the `height` variable is updated as necessary to store the maximum height of any column.



The diagram on the left shows an example `Tetris` board first in *column representation* and then in *row representation* — the "before" and "after" diagrams for a call to the new `Tetris` class member function `convert_to_row_representation`. Note that once in row representation the `heights` variable isn't needed and we set it to `NULL`.
The `convert_to_row_representation` function takes no arguments.

Now write the `Tetris` class member function `convert_to_row_representation` as it would appear in the `tetris.cpp` implementation file. You may assume that before the call the board is in the column representation and the member variables are all set correctly. Make sure your code properly allocates new memory as needed and does not have memory leaks.

Solution:

```
void Tetris::convert_to_row_representation() {
  // allocate the top level arrays
```

```
  widths = new int[height];
  char** tmp = new char*[height];
  // for each row...
  for (int h = 0; h < height; h++ ) {
    // calculate the width of each row
    widths[h] = 0;
    for (int w = 0; w < width; w++ ) {
      if (heights[w] > h && data[w][h] != ' ') widths[h] = w+1;
    }
    // allocate a row of the correct width in the tmp structure
    assert (widths[h] > 0);
    tmp[h] = new char[widths[h]];
    // fill in the row character data
    for (int w = 0; w < widths[h]; w++) {
      if (heights[w] > h)
        tmp[h][w] = data[w][h];
      else
        tmp[h][w] = ' ';
    }
  }
  // cleanup the old structure
  delete [] heights;
  heights = NULL;
  for (int i = 0; i < width; i++) {
    delete [] data[i];
  }
  delete [] data;
  // point to the new data
  data = tmp;
}
```

7 Mystery Recursion [/9]

For each function or pair of functions below, choose the letter that best describes the program purpose or behavior.

A) infinite loop
B) factorial
C) integer power
D) the answer is 42
E) function is not recursive
F) sum of the digits
G) syntax error
H) modulo 2
I) reverse the digits
J) multiplication
K) greatest common divisor
L) other

Solution: K

```
int mysteryONE(int x, int y) {
  if(y == 0)
    return x;
  else
    return mysteryONE(y, x % y);
}
```

Solution: F

```
int mysteryTWO(int x) {
  if (x == 0)
    return 0;
  else
    return mysteryTWO(x/10)
           + x%10;
}
```

Solution: H

```
int mysteryTHREEa(int x);

int mysteryTHREEb(int x) {
  if (x == 0)
    return 1;
  else
    return mysteryTHREEa(x-1);
}

int mysteryTHREEa(int x) {
  if (x == 0)
    return 0;
  else
    return mysteryTHREEb(x-1);
}
```

Solution: J

```
int mysteryFOUR(int x, int y) {
  if (x == 0)
    return 0;
  else
    return y +
      mysteryFOUR(x-1,y);
}
```

Solution: I

```
int mysteryFIVEa(int x, int y) {
  if (x == 0)
    return y;
  else
    return mysteryFIVEa
       (x/10, y*10 + x%10);
}

int mysteryFIVEb(int x) {
  return mysteryFIVEa(x,0);
}
```

Solution: B

```
int mysterySIX(int x) {
  if (x == 0)
    return 1;
  else
    return x *
        mysterySIX(x-1);
}
```

8 Collecting Words [/ 18]

Write a function named `Collect` that takes in two *alphabetically sorted* STL `lists` of STL `strings` named `threes` and `candidates`. The function searches through the second list and removes all three letter words and places them in the first list in alphabetical order. For example, given these lists as input:

```
threes:      cup dog fox map
candidates:  ant banana egg goat horse ice jar key lion net
```

After the call to `Collect(threes, candidates)` the lists will contain:

```
threes:      ant cup dog egg fox ice jar key map net
candidates:  banana goat horse lion
```

If there are n and m words in the input lists, the order notation of your solution should be $O(n + m)$.

Solution:

```
void collect(std::list<std::string> &threes, std::list<std::string> &candidates) {
  // start an iterator at the front of each list
  std::list<std::string>::iterator itr = threes.begin();
  std::list<std::string>::iterator itr2 = candidates.begin();
  // loop over all of candidate words
```

```
  while (itr2 != candidates.end()) {
    // if the candidate is length 3
    if ((*itr2).size() == 3) {
      // find the right spot for this word
      while (itr != threes.end() && *itr < *itr2) {
        itr++;
      }
      // modify the two lists
      threes.insert(itr,*itr2);
      itr2 = candidates.erase(itr2);
    } else {
      // only advance the pointer if the length is != 3
      itr2++;
    }
  }
}
```

9 Constantly Referencing `DSStudent` [/ 12]

The expected output of the program below is:

```
chris is a sophomore, his/her favorite color is blue, and he/she has used 1 late day(s).
```

However, there are a number of small but problematic errors in the `DSStudent` class code. Hint: This problem's title is relevant! Only one completely new line may be added (line 6), and the 7 other lines require one or more small changes. These lines are tagged with an asterisk, `*`. Your task is to rewrite each incorrect or missing line in the appropriately numbered box. *Please write the entire new line in the box.*

```
  1 class DSStudent {
  2  public:
* 3   DSStudent(std::string n, int y)
  4     : name(n) {
* 5     int entryYear = y;
* 6
  7   }
* 8   std::string& getName() const {
  9     return name;
 10   }
*11   const std::string& getYear() {
 12     if (entryYear == 2014) {
 13       return "freshman"; }
 14     } else if (entryYear == 2013) {
 15       return "sophomore";
 16     } else if (entryYear == 2012) {
 17       return "junior";
 18     } else {
 19       return "senior";
 20     }
 21   }
*22   void incrLateDaysUsed() const {
 23     days++;
 24   }
*25   int& getLateDaysUsed() const {
 26     return days;
 27   }
*28   std::string FavoriteColor() {
 29     return color;
 30   }
 31  private:
 32   std::string name;
 33   std::string color;
 34   int entryYear;
 35   int days;
 36 };
```

```
37
38 int main() {
39   DSStudent s("chris",2013);
40   s.FavoriteColor() = "blue";
41   s.incrLateDaysUsed();
42   std::cout << s.getName()
43             << " is a " << s.getYear()
44             << ", his/her favorite color is " << s.FavoriteColor()
45             << ", and he/she has used " << s.getLateDaysUsed()
46             << " late day(s)." << std::endl;
47 }
```

3 **Solution:** `DSStudent(const std::string &n, int y)`

5 **Solution:** `entryYear = y;`

6 **Solution:** `days = 0;`

8 **Solution:** `const std::string& getName() const {`

11 **Solution:** `std::string getYear() const {`

22 **Solution:** `void incrLateDaysUsed() {`

25 **Solution:** `int getLateDaysUsed() const {`

28 **Solution:** `std::string& FavoriteColor() {`

10 Efficient Occurrences [/ 22]

Write a *recursive* function named `Occurrences` that takes in a *sorted* STL `vector` of STL `string`s named `data`, and an STL `string` named `element`. The function returns an integer, the number of times that `element` appears in `data`. Your function should have order notation $O(log\ n)$, where n is the size of `data`.

Solution:

```
// the recursive helper function
int occurrences(const std::vector<std::string> &data, const std::string &element,
                int s1, int s2, int e1, int e2) {
  // s1 & s2 are the current range for the start / first occurence
  // e1 & e2 are the current range for the end / last occurence (+1)
  assert (s1 <= s2 && e1 <= e2);
  if (s1 < s2) {
    // first use binary search to find the first occurrence of element
    int mid = (s1 + s2) / 2;
    if (data[mid] >= element)
      return occurrences(data,element,s1,mid,e1,e2);
    return occurrences(data,element,mid+1,s2,e1,e2);
  } else if (e1 < e2) {
    // then use binary search to find the last occurrence of element (+1)
    int mid = (e1 + e2) / 2;
    if (data[mid] > element)
      return occurrences(data,element,s1,s2,e1,mid);
    return occurrences(data,element,s1,s2,mid+1,e2);
  } else {
    // the simply subtract these indices
    assert (s1 == s2 && e1 == e2 && e1 >= s1);
    return e1 - s1;
  }
}

// "driver" function
```

```
int occurrences(const std::vector<std::string> &data, const std::string &element) {
  // use binary seach twice to find the first & last occurrence of element
  return occurrences(data,element,0,data.size(),0,data.size());
}
```

11 Short Answer [/ 8]

11.1 What's Wrong? [/ 4]

Write 1-2 complete and concise sentences describing the problem with this code fragment:

```
std::vector<std::string> people;
people.push_back("sally");
people.push_back("brian");
people.push_back("monica");
people.push_back("fred");
std::vector<std::string>::iterator mom = people.begin() + 2;
std::vector<std::string>::iterator dad = people.begin() + 1;
people.push_back("paula");
std::cout << "My parents are " << *mom << " and " << *dad << std::endl;
```

Solution: Any iterators attached to an STL vector should be assumed to be invalid after a call to `push_back` (or `erase` or `resize`) because the internal dynamically allocated array may have been relocated in memory (or the data shifted). Dereferencing the pre-`push_back` iterators to print the data is dangerous since that memory may have been deleted/freed.

11.2 Fear of Recursion [/ 4]

Rewrite this function without recursion:

```
class Node {
public:
  std::string value;
  Node* next;
};
```

```
void printer (Node* n) {
  if (n->next == NULL) {
    std::cout << n->value;
  } else {
    std::cout << "(" << n->value << "+";
    printer (n->next);
    std::cout << ")";
  }
}
```

Solution:

```
void printer (Node* n) {
  int count = 0;
  while (n != NULL) {
    if (n->next != NULL) {
      std::cout << "(" << n->value << "+";
      count++;
    } else {
      std::cout << n->value;
    }
    n = n->next;
  }
  std::cout << std::string(count,')');
}
```

12 Converting Between `Vec` and `dslist` [/ 26]

Ben Bitdiddle is working on a project that stores data with two different data structures: our `Vec` and `dslist` classes. Occasionally he needs to convert data from one format to the other format. Alyssa P. Hacker suggests that he write a copy-constructor-like function for each class that takes in a single argument, the original format of the data. For example, here's how to convert data in `Vec` format to `dslist` format:

```
// create a Vec object with 4 numbers
Vec<int> v;  v.push_back(1);  v.push_back(2);  v.push_back(3);  v.push_back(4);
// create a dslist object that initially stores the same data as the Vec object
dslist<int> my_lst(v);
```

Here are the relevant portions of the two class declarations (and the `Node` helper class):

```
template <class T> class Vec {
public:
  // conversion constructor
  Vec(const dslist<T>& lst);
  /* other functions omitted */
  // representation
  T* m_data;
  unsigned int m_size;
  unsigned int m_alloc;
};
```

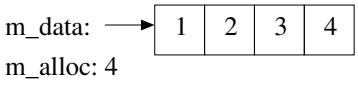
```
template <class T> class Node {
public:
  Node(const T& v):
    value_(v),next_(NULL),prev_(NULL){}
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};

template <class T> class dslist {
public:
  // conversion constructor
  dslist(const Vec<T>& vec);
  /* other functions omitted */
  // representation
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
```

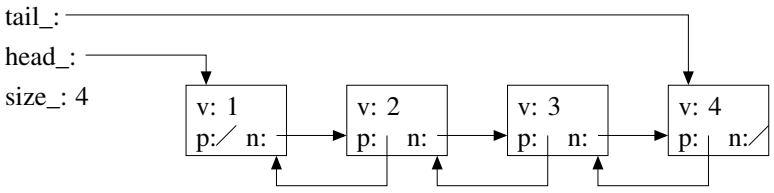
Ben asks about access to the private member variables of one class from a member function of the other. Alyssa says he can write the functions assuming he has full access to the private member variables. (She promises to teach him how to use the `friend` keyword to make that work after Test 2.)

12.1 Diagrams [/ 8]

First, draw the detailed internal memory representations for a `Vec` object and a `dslist` object, each storing the numbers: `1 2 3 4`.



Solution: m_size: 4



Solution:

12.2 Implementing the Conversion Constructors [/ 18]

Now write the two conversion constructors. You may not use `push_back`, `push_front`, `insert` or iterators in your answer. Instead, demonstrate that you know how to construct and manipulate the low level memory representation.

Solution:

```
template <class T> Vec<T>::Vec(const dslist<T>& lst) {
  m_alloc = m_size = lst.size();
  if (m_alloc > 0)
    m_data = new T[m_alloc];
  else
    m_data = NULL;
  int i = 0;
  Node<T> *tmp = lst.head_;
  while (tmp != NULL) {
    m_data[i] = tmp->value_;
    tmp = tmp->next_;
    i++;
  }
}
```

```
template <class T> dslist<T>::dslist(const Vec<T>& v) {
  head_ = tail_ = NULL;
  size_ = v.size();
```

```
  Node<T> *tmp = NULL;
  for (int i = 0; i < size_; ++i) {
    tail_ = new Node<T>(v.m_data[i]);
    if (tmp != NULL) {
      tail_->prev_ = tmp;
      tmp->next_ = tail_;
    }
    if (i == 0) head_ = tail_;
    tmp = tail_;
  }
}
```

13 Matrix Transpose [/ 20]

First, study the partial implementation of the templated `Matrix` class on the right. Your task is to implement the `transpose` member function for this class (as it would appear outside of the class declaration). Remember from math class that the transpose flips the matrix data along the diagonal from the upper left corner to the lower right corner. For example:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \xrightarrow{transpose} \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

```
template <class T> class Matrix {
public:
  Matrix(int rows, int cols, const T &v);
  ~Matrix();
  int getRows() const { return rows_; }
  int getCols() const { return cols_; }
  const T& get(int r, int c) const
    { return values[r][c]; }
  void set(int r, int c, const T &v)
    { values[r][c] = v; }
  void transpose();
private:
  int rows_;
  int cols_;
  T **values;
};
```

Solution:

```
template <class T>
void Matrix<T>::transpose() {
  // move the current matrix out of the way
  T **old = values;
  // create a new top level array to store the rows
  values = new T*[cols_];
  for (int i = 0; i < cols_; i++) {
    // create each row
    values[i] = new T[rows_];
    // populate the values
    for (int j = 0; j < rows_; j++) {
      values[i][j] = old[j][i];
    }
  }
  // clean up the old data
  for (int i = 0; i < rows_; i++) {
    delete [] old[i];
  }
  delete [] old;
  // swap the counters for rows & columns
  int tmp = rows_;
  rows_ = cols_;
  cols_ = tmp;
}
```

14 Book, Page, Sentence, & Word Iteration [/ 18]

Write a function `PageWithMostSentencesWithWord` that takes in two arguments. The first argument is an STL `list` of STL `lists` of STL `lists` of STL `strings` that represents a book with pages. Each page has multiple sentences. Each sentence has multiple words. The second argument is an STL `string` with the search word. The function should return the page number that has the most sentences that contain the search word. The first page in the book

is numbered 1 (not zero). You may assume that any punctuation has already been removed and everything has been converted to lowercase.

Solution:

```
int PageWithMostSentencesWithWord(const std::list<std::list<std::list<std::string> > > &book,
                                  const std::string &search) {
  int current = 0;
  int answer = -1;
  int most;
  std::list<std::list<std::list<std::string> > >::const_iterator page;
  std::list<std::list<std::string> >::const_iterator sentence;
  std::list<std::string>::const_iterator word;
  for (page = book.begin(); page != book.end(); page++) {
    current++;
    int count = 0;
    for (sentence = (*page).begin(); sentence != (*page).end(); sentence++) {
      bool found = false;
      for (word = (*sentence).begin(); word != (*sentence).end(); word++) {
        if (*word == search) found = true;
      }
      if (found) count++;
    }
    if (answer == -1 || most < count) {
      answer = current;
      most = count;
    }
  }
  return answer;
}
```

15 Linear 2048 [/ 18]

Write a *recursive* function named `Linear2048` that takes in an STL list of integers and plays a single line based version of the 2048 game. If two adjacent numbers are equal to each other in value, those two elements merge and are replaced with their sum. The function returns the maximum value created by any of the merges during play. The example shown on the right reduces the original input list with 17 values to a list with 4 values and returns the value 2048.

```
8 2 2 1024 256 32 16 8 4 1 1 2 32 32 128 512 32
8 4 1024 256 32 16 8 4 1 1 2 32 32 128 512 32
8 4 1024 256 32 16 8 4 2 2 32 32 128 512 32
8 4 1024 256 32 16 8 4 4 32 32 128 512 32
8 4 1024 256 32 16 8 8 32 32 128 512 32
8 4 1024 256 32 16 16 32 32 128 512 32
8 4 1024 256 32 32 32 32 128 512 32
8 4 1024 256 64 32 32 128 512 32
8 4 1024 256 64 64 128 512 32
8 4 1024 256 128 128 512 32
8 4 1024 256 256 512 32
8 4 1024 512 512 32
8 4 1024 1024 32
8 4 2048 32
```

Solution:

```
int linear_2048(std::list<int> &input) {
  // nothing to do if there aren't at least 2 elements
  if (input.size() <= 1) return -1;
  // start up 2 side-by-side iterators
  std::list<int>::iterator itr = input.begin();
  std::list<int>::iterator itr2 = itr;
  itr2++;
  // walk down the list, looking for 2 neighboring elements with the same value
  while (itr2 != input.end() && *itr != *itr2) {
    itr++;
    itr2++;
  }
  // if we're at the end of the list, nothing to do
  if (itr2 == input.end()) return -1;
  // double the current value
  *itr = (*itr)*2;
  // erase the element under the other iterator
  input.erase(itr2);
  // write down the current value (itr may be changed by recursive call)
  int a = *itr;
```

```
  int b = linear_2048(input);
  // return the larger value
  return std::max(a,b);
}
```

16 Mystery Function Memory Usage Order Notation [/ 6]

What does this function compute? What is the order notation of the size of the memory necessary to store the return value of this function? Give your answer in terms of n, the number of elements in the input vector, and k, the average or worst case length of each string in the input vector. Write 3-4 concise and well-written sentences to justify your answer.

```
std::vector<std::string> mystery(const std::vector<std::string> &input) {
  if (input.size() == 1) { return input; }
  std::vector<std::string> output;
  for (int i = 0; i < input.size(); i++) {
    std::vector<std::string> helper_input;
    for (int j = 0; j < input.size(); j++) {
      if (i == j) continue;
      helper_input.push_back(input[j]);
    }
    std::vector<std::string> helper_output = mystery(helper_input);
    for (int k = 0; k < helper_output.size(); k++) {
      output.push_back(input[i]+", "+helper_output[k]);
    }
  }
  return output;
}
```

Solution: This function reserves one element at a time from the input vector, recurses on the remaining vector, and then concatenates the reserved element to the front of each item in the recursion output. Thus, this function generates all *permutations* of the input vector.

By definition, the number of permutations is $n!$. The length of each permutation is $n * k$ (technically $n * k + (n-1) * 2$ with the commas and spaces). Therefore, the storage space/memory needed for the output vector is $O(n * k * n!)$.

17 LeapFrogSplit on a Doubly-Linked List [/ 26]

In this problem, we will implement the `LeapFrogSplit` function which manipulates a doubly-linked list of `Node`s. This function takes in 3 arguments: pointers to the *head* & *tail* `Node`s of a doubly-linked list, and an integer *value*. The function locates the `Node` containing that value, removes the node, splits the value in half, and re-inserts the half values into the list jumping over both of the original neighbors before and after it in the list.

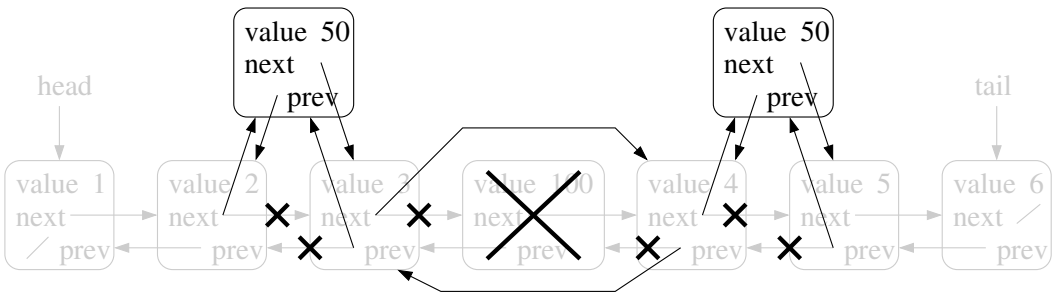
For example, if the linked list initially contains 7 nodes with the data:
`1 2 3 100 4 5 6`, then after executing `LeapFrogSplit(head,tail,100)`
it will contain 8 nodes: `1 2 50 3 4 50 5 6`.

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

17.1 Diagram [/ 5]

First, modify the diagram below to illustrate the result of `LeapFrogSplit(head,tail,100)`.

Solution:



17.2 Corner Cases & Testing [/ 7]

What "corner cases" do you need to consider for this implementation? Give 4 interesting examples of input and what you define as the correct result for each case. Write 2-3 explanatory sentences as needed.

Solution: We need to handle the case where the node in front of the target node is the head (and/or similarly where the node after the target node is the tail). In this case we must reassign the head (and/or tail) to the newly inserted node:

```
1 100 2 3 4 5 6   =>   50 1 2 50 3 4 5 6
1 2 3 4 5 100 6   =>   1 2 3 4 50 5 6 50
1 100 2           =>   50 1 2 50
```

We need to handle the case where the target Node is the first or last node in the linked list chain. If this is the case, we cannot insert one of the new nodes: ***(Note: Alternate definitions for results in these cases are possible.)***

```
100 1 2 3 4   =>   1 50 2 3 4
1 2 3 4 100   =>   1 2 3 50 4
100           =>   <empty list>
```

We should make sure that our solution works when the element is not present in the input list.

```
1 2 3 4 5 6   =>   1 2 3 4 5 6
<empty list>  =>   <empty list>
```

And we could also worry about splitting a node with an odd value...

```
1 2 3 101 4 5 6   =>   1 2 50 3 4 51 5 6
```

17.3 Implementing `LeapFrogSplit` [/ 14]

Finally, write `LeapFrogSplit`. Focus primarily on correctly performing the general case that we diagrammed on the previous page. Corner cases are worth only a small number of points.

Solution:

```
void LeapFrogSplit2(Node* &head, Node* &tail, int value) {
  // locate the element
  Node *tmp = head;
  while (tmp != NULL && tmp->value != value) {
    tmp = tmp->next;
  }
  // do nothing if the element was not found
  if (tmp == NULL) return;

  // if there is a previous element to leap backwards over...
  if (tmp->prev != NULL) {
    Node *a = new Node(value/2);
    a->next = tmp->prev;
    a->prev = tmp->prev->prev;
    if (tmp->prev != head) {
      tmp->prev->prev->next = a;
    } else {
      head = a;
    }
    tmp->prev->prev = a;
    tmp->prev->next = tmp->next;
  }

  // if there is a next element to leap forwards over...
  if (tmp->next != NULL) {
    Node *b = new Node(value - value/2);
    b->next = tmp->next->next;
    b->prev = tmp->next;
    if (tmp->next != tail) {
```

```
      tmp->next->next->prev = b;
    } else {
      tail = b;
    }
    tmp->next->next = b;
    tmp->next->prev = tmp->prev;
  }


  // reset head & tail if either or both point to the deleted element
  if (head == tmp)
    head = tmp->next;
  if (tail == tmp)
    tail = tmp->prev;
  // clean up the memory
  delete tmp;
}
```