Binary Search

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
  if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid]) return binsearch(v, low, mid, x);
  else return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
   return binsearch(v, 0, v.size()-1, x);
}
```

++

Erase and Insert

1. The erase member function (for STL vector and STL list) takes in a single argument, an iterator pointing at an element in the container. It removes that item, and the function returns an iterator pointing at the element after the removed item.
2. Similarly, there is an insert function for STL vector and STL list that takes in 2 arguments, an iterator and a new element, and adds that element immediately before the item pointed to by the iterator. The function returns an iterator pointing at the newly added element.
3. Even though the erase and insert functions have the same syntax for vector and for list, the vector versions are O(n), whereas the list versions are O(1).
4. Iterators positioned on an STL vector, at or after the point of an erase operation, are invalidated. Iterators positioned anywhere on an STL vector may be invalid after an insert (or push back or resize) operation.
5. Iterators attached to an STL list are not invalidated after an insert or erase (except iterators attached to the erased element!) or push back/push front.

++

Doubly Linked Lists

```
// A simplified implementation of a generic list container
class, including the iterator, but not the const_iterators.
Three separate classes are defined: a Node class, an iterator
class, and the actual list class.  The underlying list is
doubly-linked, but there is no dummy head node and the list
is not circular.
// ——————————————————
// NODE CLASS
template <class T> class Node {
public:
  Node() : next_(NULL), prev_(NULL) {}
  Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
  // REPRESENTATION
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};
// A "forward declaration" of this class is needed
template <class T> class dslist;
// ——————————————————
// LIST ITERATOR
template <class T> class list_iterator {
public:
  // default constructor, copy constructor, assignment
operator, & destructor
  list_iterator() : ptr_(NULL) {}
  list_iterator(Node<T>* p) : ptr_(p) {}
  list_iterator(const list_iterator<T>& old) : ptr_(old.ptr_) {}
  list_iterator<T>& operator=(const list_iterator<T>& old) {
    ptr_ = old.ptr_;  return *this; }
  ~list_iterator() {}
  // dereferencing operator gives access to the value at the pointer
  T& operator*()  { return ptr_->value_;  }
  // increment & decrement operators
  list_iterator<T>& operator++() { // pre-increment, e.g., ++iter
    ptr_ = ptr_->next_;
    return *this;
  }
  list_iterator<T> operator++(int) { // post-increment, e.g., iter++
    list_iterator<T> temp(*this);
    ptr_ = ptr_->next_;
    return temp;
  }
  list_iterator<T>& operator--() { // pre-decrement, e.g., --iter
    ptr_ = ptr_->prev_;
    return *this;
  }
  list_iterator<T> operator--(int) { // post-decrement, e.g., iter--
    list_iterator<T> temp(*this);
    ptr_ = ptr_->prev_;
    return temp;
  }
  // the dslist class needs access to the private ptr_ member variable
  friend class dslist<T>;
  // Comparions operators are straightforward
  bool operator==(const list_iterator<T>& r) const { return ptr_ == r.ptr_; }
  bool operator!=(const list_iterator<T>& r) const { return ptr_ != r.ptr_; }
private:
  // REPRESENTATION
  Node<T>* ptr_;    // ptr to node in the list
};
// ——————————————————
// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
template <class T> class dslist {
public:
  // default constructor, copy constructor, assignment
operator, & destructor
  dslist() : head_(NULL), tail_(NULL), size_(0) {}
  dslist(const dslist<T>& old) { this->copy_list(old); }
  dslist& operator= (const dslist<T>& old);
  ~dslist() { this->destroy_list(); }
  // simple accessors & modifiers
  unsigned int size() const { return size_; }
  bool empty() const { return head_ == NULL; }
  void clear() { this->destroy_list(); }
  // read/write access to contents
  const T& front() const { return head_->value_;  }
  T& front() { return head_->value_; }
  const T& back() const { return tail_->value_; }
  T& back() { return tail_->value_; }
  // modify the linked list structure
  void push_front(const T& v);
  void pop_front();
  void push_back(const T& v);
  void pop_back();
  typedef list_iterator<T> iterator;
  iterator erase(iterator itr);
  iterator insert(iterator itr, const T& v);
  iterator begin() { return iterator(head_); }
  iterator end() { return iterator(NULL); }
private:
  // private helper functions
  void copy_list(const dslist<T>& old);
  void destroy_list();
  //REPRESENTATION
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
// ——————————————————
// LIST CLASS IMPLEMENTATION
template <class T> dslist<T>& dslist<T>::operator= (const
dslist<T>& old) {
  // check for self-assignment
  if (&old != this) {
    this->destroy_list();
    this->copy_list(old);
  }
  return *this;
}
template <class T> void dslist<T>::push_front(const T& v) {
  Node<T>* newp = new Node<T>(v);
  // initially empty list as a special case
  if (!head_) head_ = tail_ = newp;
  else {
    // normal case: at least one node
    newp->next_ = head_;
    head_->prev_ = newp;
    head_ = newp;
  }
  ++size_;
}
template <class T> void dslist<T>::pop_front() {
  Node<T>* oldp = head_;  // save the current head pointer
  if (size_ == 0) return;
  if (head_ == tail_) { // special case: deleting the last node
    head_ = NULL;
    tail_ = NULL;
  }
  else head_ = head_->next_;
  --size_;
  // remove node
  delete oldp;
}
template <class T> void dslist<T>::push_back(const T& v) {
  Node<T>* newp = new Node<T>(v);
  // special case: initially empty list
  if (!tail_) head_ = tail_ = newp;
  else {
    // normal case: at least one node already
    newp->prev_ = tail_;
    tail_->next_ = newp;
    tail_ = newp;
  }
  ++size_;
}
template <class T> void dslist<T>::pop_back() {
  // You Can Do It!
}
// do these lists look the same (length & contents)?
template <class T> bool operator== (dslist<T>& left, dslist<T>& right) {
  if (left.size() != right.size()) return false;
  typename dslist<T>::iterator left_itr = left.begin();
  typename dslist<T>::iterator right_itr = right.begin();
  // walk over both lists, looking for a mismatched value
  while (left_itr != left.end()) {
    if (*left_itr != *right_itr) return false;
    left_itr++; right_itr++;
  }
  return true;
}
template <class T> bool operator!= (dslist<T>& left, dslist<T>&
right) { return !(left==right); }
template <class T> typename dslist<T>::iterator
dslist<T>::erase(iterator itr) {
  assert (size_ > 0);
  --size_;
  iterator result(itr.ptr_->next_);
  // One node left in the list.
  if (itr.ptr_ == head_ && head_ == tail_) head_ = tail_ = 0;
  // Removing the head in a list with at least two nodes
  else if (itr.ptr_ == head_) {
    head_ = head_->next_;
    head_->prev_ = 0;
  }
  // Removing the tail in a list with at least two nodes
  else if (itr.ptr_ == tail_) {
    tail_ = tail_->prev_;
    tail_->next_ = 0;
  }
  // Normal remove
  else {
    itr.ptr_->prev_->next_ = itr.ptr_->next_;
    itr.ptr_->next_->prev_ = itr.ptr_->prev_;
  }
  delete itr.ptr_;
  return result;
}
template <class T> typename dslist<T>::iterator
dslist<T>::insert(iterator itr, const T& v) {
  ++size_ ;
  Node<T>* p = new Node<T>(v);
  p->prev_ = itr.ptr_->prev_;
  p->next_ = itr.ptr_;
  itr.ptr_->prev_ = p;
  if (itr.ptr_ == head_) head_ = p;
  else p->prev_->next_ = p;
  return iterator(p);
}
template <class T> void dslist<T>::copy_list(const dslist<T>& old) {
  size_ = old.size_;
  // Handle the special case of an empty list.
  if (size_ == 0) {
    head_ = tail_ = 0;
    return;
  }
  // Create a new head node.
  head_ = new Node<T>(old.head_->value_);
  // tail_ will point to the last node created and therefore will move
  // down the new list as it is built
  tail_ = head_;
  // old_p will point to the next node to be copied in the old list
  Node<T>* old_p = old.head_->next_;
  // copy the remainder of the old list, one node at a time
  while (old_p) {
    tail_->next_ = new Node<T>(old_p->value_);
    tail_->next_->prev_ = tail_;
    tail_ = tail_->next_;
    old_p = old_p->next_;
  }
}
template <class T> void dslist<T>::destroy_list() {
  if (head_ == NULL) return;
  while (head_ != NULL) {
    Node<T>* tmp = head_;
    head_ = head_->next_;
    delete tmp;
  }
}
```

++

Merge Sort

```
using namespace std;
// The driver function for mergesort.  It defines a scratch
vector for temporary copies.
template <class T> void mergesort(vector<T>& values) {
  vector<T> scratch(values.size());
  mergesort(0, int(values.size()-1), values, scratch);
}
// Here's the actual merge sort function.  It splits the vector
in half, recursively sorts each half, and then merges the two
sorted halves into a single sorted interval.
template <class T> void mergesort(int low, int high, vector<T>&
values, vector<T>& scratch) {
  cout << "mergesort:  low = " << low << ", high = " << high << endl;
  if (low >= high)  //  intervals of size 0 or 1 are already sorted!
    return;
  int  mid = (low + high) / 2;
  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);
  merge(low, mid, high, values, scratch); // O(n)
}
// merge: O(n), where n = high-low. Non-recursive function to
merge two sorted intervals (low..mid & mid+1..high) of a
vector, using "scratch" as temporary copying space.
template <class T> void merge(int low, int mid, int high,
vector<T>& values, vector<T>& scratch) {
  cout << "merge:  low = " << low << ", mid = " << mid << ", high
= " << high << endl;
  int i=low;        // "top" of pile a  [low -> mid]
  int j = mid+1; // "top" of pile b  [mid+1 -> high]
  int k=low;      // the next slot in the sorted
                  // result currently in scratch
  for ( ; k <= high ; k++) {
    // check to see if one of the piles is empty
    // if (i > mid || j > high) break;
    if (i <= mid && (j > high || values[i] < values [j])) {
      scratch[k] = values[i];
      i++;
    } else {
      scratch[k] = values[j];
      j++;
    }
  }
  // copy scratch back to values
  for (k=low ; k <= high ; k++) values[k] = scratch[k];
}
```

++

Nonlinear Word Search

```
// helper function to check if a positioin has already been
used for this word
bool on_path(loc pos, std::vector<loc> const& path) {
  for (unsigned int i=0; i<path.size(); ++i)
    if (pos == path[i]) return true;
  return false;
}
bool search_from_loc(loc pos, const std::vector<std::string>& bd,
const std::string& word, std::vector<loc>& path ) {
  path.push_back(pos);
  if (path.size() == word.size()) return true;
  for (int i = std::max(pos.row-1, 0); i < std::min(int(bd.size()),
pos.row+2); ++i) {
    for (int j = std::max(pos.col-1, 0); j < std::min(int(bd[i].size()),
pos.col+2); ++j) {
      if (on_path(loc(i,j), path)) continue;
      if (bd[i][j] == word[path.size()]) {
        if (search_from_loc(loc(i,j), bd, word, path)) return true;
      }
  }
  path.pop_back();
  return false;
}
```

++

8 Queens

```
class Queen {
public:
  Queen(int row, int col) : row_(row), col_(col) {}
  int getRow() const {return row_;}
  int getCol() const {return col_;}
  void Print() const {std::cout << "(" << row_ << ", " << col_ << ")"
<< std::endl;}
  void setPosition(int row, int col) {row_ = row; col_ = col;}
private:
  int row_;
  int col_;
};
void PrintBoard(const std::vector<Queen> queens, int
num_rows, int num_cols) {
  std::vector<std::vector<char> > grid(num_rows,
std::vector<char>(num_cols, '_'));
  for (std::vector<Queen>::const_iterator it = queens.begin(); it !=
queens.end(); ++it) {
    (*it).Print();
    grid[(*it).getRow()][(*it).getCol()] = 'Q';
  }
  for (int r = 0; r < num_rows; ++r) {
    for (int c = 0; c < num_cols; ++c) std::cout << grid[r][c];
    std::cout << std::endl;
  }
}
// Is this position safe? search fro an attck by other queens
bool SafeSquare(const std::vector<Queen> queens, int row, int col) {
  for (std::vector<Queen>::const_iterator it = queens.begin();
       it != queens.end(); ++it) {
    int qrow = (*it).getRow();
    int qcol = (*it).getCol();
    if (qrow == row) return false;
    else if (qcol == col) return false;
    else if (qcol - qrow == col - row || qcol + qrow == col + row)
      return false;
  }
  return true;
}
// Place a new queen
bool PlaceQueens(std::vector<Queen>& queens, int num_rows,
int num_cols) {
  // done if we have a queen on each row and column
  if (int(queens.size()) == num_rows)
    return true;
  // seach for a new spot
  for (int r = 0; r < num_rows; ++r) {
    for (int c = 0; c < num_cols; ++c) {
      if (SafeSquare(queens, r, c)) {
        Queen q(r, c);   // add a new queen
        queens.push_back(q);
        if (PlaceQueens(queens, num_rows, num_cols)) return true;
        queens.pop_back();  // seach failed, try the next spot
      }
    }
  }
  return false;
}
```

++

Map

1. Map search, insert and erase are O(log n).
2. Maps are ordered by increasing value of the key. Therefore, there must be an operator< defined for the key.
3. The function std::make pair **creates a pair object from the given values..**
4. The result of using [] is that the key is always in the map afterwards.
5. m.find(key) **where m is the map object and key is the search key. It returns a map iterator: If the key is in one of the pairs stored in the map, find returns an iterator referring to this pair. If the key is not in one of the pairs stored in the map, find returns m.end().**
6. Insert: m.insert(std::make_pair(key, value)); **returns a pair of a map iterator and a bool:** std::pair<map<key_type, value_type>::iterator, bool> **The insert function checks to see if the key being inserted is already in the map. If so, it does not change the value, and returns a (new) pair containing an iterator referring to the existing pair in the map and the bool value false. If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the newly added pair in the map and the bool value true.**
7. void erase(iterator p) **erase the pair referred to by iterator p.** void erase(iterator first, iterator last) **erase all pairs from the map starting at first and going up to, but not including, last.** size_type erase(const key_type& k) **erase the pair containing key k, returning either 0 or 1, depending on whether or not the key was in a pair in the map**

++

Dynamic Tetris Arrays

>>>Tetris Representation Conversion<<<

```
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;
```

++

Collecting Words

```
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);
    // only advance the pointer if the length is != 3
   }
   else itr2++;
  }
}
```
++

Efficient Occurrences

```
// 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());
}
```
++

Fear of Recursion

```
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,')');
}
```
++

Converting Between Vec and dslist

```
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_;
 }
}
```
++

Valet Parking Maps

>>>The Car class<<<

We must define operator< for Car objects so that we can sort the keys of the map.

```
bool operator<(const Car &a, const Car &b) {
 return (a.getMaker() < b.getMaker() ||
  (a.getMaker() == b.getMaker() && a.getColor() < b.getColor()));
}
```
++

>>>print_cars<<<

```
void print_cars(const map<Car,vector<string> > &cars) {
 map<Car,vector<string> >::const_iterator itr = cars.begin();
 while (itr != cars.end()) {
   Car c = itr->first;
   cout << "People who drive a " << c.getColor() << " " <<
                                   c.getMaker() << ":" << endl;
   vector<string>::const_iterator itr2 = itr->second.begin();
   while (itr2 != itr->second.end()) {
     cout << "  " << *itr2 << endl;
     itr2++;
   }
   itr++;
 }
}
```
++

>>>remove_cars<<<

```
bool remove_car(map<Car,vector<string> > &cars,
   const string &name, const string &color, const string &maker) {
 map<Car,vector<string> >::iterator itr = cars.find(Car(maker,color));
 if (itr == cars.end()) return false;
 if (itr->second.size() == 1 && itr->second[0] == name) {
   cars.erase(Car(maker,color));
   return true;
 }
 for (int i = 0; i < itr->second.size(); i++) {
   if (itr->second[i] == name) {
     itr->second.erase(itr->second.begin() + i);
     return true;
   }
 }
 return false;
}
```
++

Movies w/ Pair/Map/String/List/Vector/Set

>>>Defining the Map Type<<<

```
typedef std::map < std::pair< std::string, std::string >,
std::vector<std::string> > MOVIE_MAP;
```
++

>>>Counting Combos<<<

```
// determine which movie comes first alphabetically
if (movie_a < movie_b) {
 count = my_map[make_pair(movie_a,movie_b)].size();
} else {
 count = my_map[make_pair(movie_b,movie_a)].size();
}
```

Solution: There are at most m^2 rows/entries in the map. Accessing a specific row takes log in the number of rows. Querying the number of entries in a vector is constant-time. Overall: O(log m^2), which can be simplified to O(log m).

++

>>>Adding Data<<<

```
void AddPerson (const std::string &name,
 const std::list<std::string> &movies, MOVIE_MAP &my_map) {
 // two nested for loops to find all pairs in the input movies list
 for (std::list<std::string>::const_iterator itr = movies.begin();
      itr != movies.end(); itr++) {
   std::list<std::string>::const_iterator itr2 = itr;
   itr2++;
   for (; itr2 != movies.end(); itr2++) {
     // determine which movie comes first alphabetically
     if (*itr < *itr2) my_map[make_pair(*itr,*itr2)].push_back(name);
     else my_map[make_pair(*itr2,*itr)].push_back(name);
     }
   }
 }
}
```

Solution: We must add/edit k^2 rows of the map. Querying for/adding a row takes log m^2 time. push_back is constant-time (amortized). Thus overall: O(k^2 ∗ log m^2), which can be simplified to O(k^2 ∗ log m).

++

>>>You Haven’t Seen “Star Wars” Yet???<<<

```
std::list<std::string> DidNotSee(const std::string &movie, const
 MOVIE_MAP &my_map) {
 // two helper data structures to collect the people
 std::set<std::string> all_people;
 std::set<std::string> did_see;
 for (MOVIE_MAP::const_iterator itr = my_map.begin(); itr !=
my_map.end(); itr++) {
   bool flag = itr->first.first == movie || itr->first.second == movie;
   for (std::vector<std::string>::const_iterator itr2 = itr->second.begin();
       itr2 != itr->second.end(); itr2++) {
     all_people.insert(*itr2);
     if (flag) did_see.insert(*itr2);
   }
 }
 // loop through the people in the helper sets to construct
the final answer
 std::list<std::string> answer;
 for (std::set<std::string>::iterator itr = all_people.begin(); itr !=
all_people.end(); itr++) {
   if (did_see.find(*itr) == did_see.end()) answer.push_back(*itr);
 }
 return answer;
}
```

Solution: To build the set of all people (and the set of people who have seen the movie), we must visit every row in the map, and every person in each row, that’s m^2 ∗ j items and then we must add them to a set of all people, which has maximum size p (each set insert = log p). Once we have the two sets, for each person in the all_people set we search the did_see set. That’s p log p. Thus, overall: O((m^2 ∗j +p)∗ log p)

++

Matrix Transpose

```
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;
}
```
++

Book, Page, Sentence, & Word Iteration

```
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;
}
```
++

Linear 2048

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

Mystery Function Memory Usage Order Notation

```
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!).

++

LeapFrogSplit on a Doubly-Linked List

```
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;
}
```
++

Circular Play List

>>>Circle constructor<<<

```
Circle::Circle(const std::vector<std::string>& data) {
 // empty input -- create empty play list
 if (data.size() == 0) current = NULL;
 else {
   // create the first node
   current = new Node(data[0]);
   // step through all of the other elements, storing a pointer
to the last one.
   Node* tmp = current;
   for (int i = 1; i < data.size(); i++) {
     tmp->next = new Node(data[i]);
     // connect the bidirectional links with the previous node
     tmp->next->prev = tmp;
     tmp = tmp->next;
   }
   // connect the bidirectional links between the first & last nodes
   tmp->next = current;
   current->prev = tmp;
 }
}
```
++

>>>Implementing remove<<<

```
bool Circle::remove(const std::string& to_remove) {
 if (current == NULL) return false;
 Node *tmp = current;
 do {
   if (to_remove == tmp->value) {
     if (tmp->next == tmp) {
       // if only one element
       delete tmp;
       current = NULL;
       return true;
     }
     // if the head is pointing at the element to be removed
     if (current == tmp) current = tmp->next;
    // bypass this element in both directions
     tmp->prev->next = tmp->next;
     tmp->next->prev = tmp->prev;
     // cleanup the memory
     delete tmp;
     return true;
   }
   tmp = tmp->next;
 } while (tmp != current);
 return false;
}
```
++

Common Data

```
template <class T>
std::vector<T> common_data(const std::vector<T> &a, const
std::vector<T> &b) {
 // a local vector variable to store the common elements
 std::vector<T> answer;
 // loop over the first vector
 for (unsigned int i = 0; i < a.size(); i++) {
   bool duplicate = false;
   // check to see if this element is a duplicate of an already
   // processed element in the first vector
   for (unsigned int j = 0; j < i; j++) {
     if (a[i] == a[j]) {
       duplicate = true;
       break;
     }
   }
   if (!duplicate) {
     // loop over the elements in the second vector
     for (unsigned int k = 0; k < b.size(); k++) {
       if (a[i] == b[k]) {
         answer.push_back(a[i]);
         // make sure to break out of this loop so we don't get
         // tricked by a duplicate in the second vector
         break;
       }
     }
   }
 }
 return answer;
}
```

Solution: O(n∗(n+m)) or O(n^2 +nm) – cannot be further simplified without information on the relative sizes of n and m.

++

Possessive Grammar

```
void convert_to_possessive (std::list<std::string> &sentence) {
 std::list<std::string>::iterator word = sentence.begin();
 // for each word in the sentence
 while (word != sentence.end()) {
   std::list<std::string>::iterator item = sentence.end();
   std::list<std::string>::iterator owner = sentence.end();
   // check for match to the pattern "the XXXX of XXXX"
   if (*word != "the") word++; continue;
   item = word;
   item++;
   if (item == sentence.end()) word++; continue;
   owner = item;
   owner++;
   if (owner == sentence.end() || *owner != "of") word++; continue;
   owner++;
   if (owner == sentence.end()) continue;
   // now make the edits
   word = sentence.erase(word); // erase "the"
   sentence.insert(word,(*owner)+"'s");
   word++;
   word = sentence.erase(word); // erase "of"
   word = sentence.erase(word); // erase owner
 }
}
```
++

Recursive Order Notation Challenge

>>>O(n)<<<

```
int fooA (int n) {
 if (n <= 0) return 0;
 else return 1 + fooA(n-1);
}
```
++

>>>O(logn)<<<

```
int fooB (int n) {
 if (n <= 1) return 0;
 else return 1 + fooB(n/2);
}
```
++

>>>O(2^n)<<<

```
int fooC (int n) {
 if (n <= 0) return 1;
 else return fooC(n-1) + fooC(n-1);
}
```