1. Simple algorithmic tracing (5 points)
   
   STAN, SRI, UTAH, BBN, CMU, NRL, HARV, MIT, RAND, UCLA

2. Recursion (15 points)

```c
/*
 * Function: filenameMatches
 * Usage: if (filenameMatches(filename, pattern)) . . .
 * --------------------------------------------------------------
 * This function checks to see whether filename matches the pattern,
 * which consists of three types of characters:
 * *
 * 1. The character ?, which matches any single character
 * 2. The character *, which matches any string of characters
 * 3. Any other character, which matches only that character
 */

bool filenameMatches(string filename, string pattern) {
    if (pattern == "") return (filename == "");
    switch (pattern[0]) {
        case '?':
            if (filename == "") return false;
            return filenameMatches(filename.substr(1), pattern.substr(1));
        case '*':
            for (int i = 0; i <= filename.length(); i++) {
                if (filenameMatches(filename.substr(i), pattern.substr(1))) { return true; }
            }
            return false;
        default:
            if (filename == "" || pattern[0] != filename[0]) return false;
            return filenameMatches(filename.substr(1), pattern.substr(1));
    }
}
```
3. Heap-stack diagrams and memory tracing (10 points)

Using explicit addresses:

Heap:

- 3.14159
- 1010
- 2.71828
- NULL
- NULL

Stack:

- 2.71828
- FF00
- 1008
- FFE0
- 2.71828
- FFE8
- key
- NULL
- numericTree

Using pointers:

Heap:

- 3.14159
- NULL
- 2.71828
- NULL
- NULL

Stack:

- 2.71828
- FF00
- 1000
- FFF0
- FFF8
- key
- numericTree
4. Linear structures and hash tables (15 points)

```cpp
/*
 * Implementation notes: rehash
 * ------------------------
 * This code walks through every cell in the old bucket array and reinserts
 * the key/value pair into the new hash table.
 */

template<typename KeyType, typename ValueType>
void HashMap<KeyType, ValueType>::rehash(int nBuckets) {
    int oldNBuckets = this->nBuckets;
    Cell **oldBuckets = buckets;
    this->nBuckets = nBuckets;
    count = 0;
    buckets = new Cell *[nBuckets];
    for (int i = 0; i < nBuckets; i++) {
        buckets[i] = NULL;
    }
    for (int i = 0; i < oldNBuckets; i++) {
        for (Cell *cp = oldBuckets[i]; cp != NULL; cp = cp->link) {
            put(cp->key, cp->value);
        }
    }
    delete[] oldBuckets;
}
```

5. Trees (15 points)

```cpp
/*
 * Implementation notes: fillVector
 * -----------------------------
 * The strategy for filling a vector is simply a matter of executing
 * an inorder traversal of the tree, adding all the nodes before this
 * one, then the current node, and finally all nodes after this one.
 */

void fillVector(BSTNode *node, Vector<BSTNode *> & v) {
    if (node != NULL) {
        fillVector(node->left, v);
        v.add(node);
        fillVector(node->right, v);
    }
}

/*
 * Implementation notes: rebuildTree
 * -------------------------------
 * The rebuildTree method operates by selecting a new root as the
 * node at the midpoint of the sorted vector. It then recursively
 * fills in the left and right subtrees by applying the same
 * strategy one level down.
 */

BSTNode *rebuildTree(Vector<BSTNode *> & v, int start, int end) {
    if (start > end) return NULL;
    int mid = (start + end) / 2;
    BSTNode *np = v[mid];
    np->left = rebuildTree(v, start, mid - 1);
    np->right = rebuildTree(v, mid + 1, end);
    return np;
}
6. Graphs (15 points)

```c
/* Constants */
const int MAX_SUGGESTIONS = 3;  /* Maximum number of friend suggestions */

/*
 * Function: suggestFriends
 * Usage: suggestFriends(g, person);
 * --------------------------------------------------
 * Makes suggestions for new friends for the specified person in the
 * graph. This function lists up to MAX_SUGGESTIONS people, sorted
 * in descending order by the number of mutual friends.
 */

void suggestFriends(Graph<Node, Arc> & g, Node *person) {
    Set<Node *> candidates = g.getNodeSet();
    candidates.clear();
    for (Node *node : g.getNeighbors(person)) {
        candidates += g.getNeighbors(node);
    }
    candidates -= person;
    candidates -= g.getNeighbors(person);
    PriorityQueue<Node *> queue;
    for (Node *node : candidates) {
        queue.enqueue(node, -countMutualFriends(g, person, node));
    }
    cout << "Friend suggestions:" << endl;
    for (int i = 0; i < MAX_SUGGESTIONS && !queue.isEmpty(); i++) {
        Node *node = queue.dequeue();
        int count = countMutualFriends(g, person, node);
        string noun = (count == 1) ? "friend" : "friends";
        cout << " " << node->name << " (" << count << " mutual ";
        cout << ((count == 1) ? "friend" : "friends") << ")" << endl;
    }
}

/*
 * Function: countMutualFriends
 * Usage: int n = countMutualFriends(g, n1, n2);
 * --------------------------------------------------
 * Returns the number of mutual friends shared by n1 and n2 in the
 * graph g. You can write this function without passing the graph
 * as an argument, but doing so makes it impossible to take advantage
 * of the getNeighbors and isConnected methods provided by the Graph
 * class.
 */

int countMutualFriends(Graph<Node, Arc> & g, Node *n1, Node *n2) {
    int count = 0;
    for (Node *node : g.getNeighbors(n1)) {
        if (g.isConnected(node, n2)) count++;
    }
    return count;
}
```
7. Data structure design (15 points)

7a)

```c
/* Private section */

private:
/

  /* Implementation notes: BigInt data structure
  * ==============================================================
  * The BigInt data structure stores the digits in the number in
  * a linked list in which the digits appear in reverse order with
  * respect to the items in the list. Thus, the number 1729 would
  * be stored in a list like this:
  *
  *       start
  *       +-------+-------+-------+-------+-------
  *       |      o------>| 9 |     ->| 2 |     ->| 7 |     ->| 1 |
  *       +-------+-------+-------+-------+-------/
  *       |      o------| o------| o------| NULL|
  *       +-------+-------+-------+-------+-------+
  *
  * The sign of the entire number is stored in a separate instance
  * variable, which is -1 for negative numbers and +1 otherwise.
  * Leading zeros are not stored in the number, which means that
  * the representation for zero is an empty list.
  */
/

  /* Type: Cell
  * ======
  * This structure type holds a single digit in the linked list.
  */

struct Cell {
    int digit;
    Cell *link;
};
/

  /* Instance variables */

Cell *start;           /* Linked list of digits */
int sign;              /* Sign of the number (-1 or +1) */
```
7b)

```cpp
/*
 * File: bigint.cpp
 * _____________
 * This file implements the bigint.h interface.
 */

#include <ctype>
#include <string>
#include "bigint.h"
#include "error.h"
using namespace std;

/*
 * Implementation notes: BigInt constructor
 * _________________________________
 * The code for this constructor offers a minimal implementation
 * that matches what we would expect on an exam. In a more
 * sophisticated implementation, it would make sense to include
 * a test to avoid storing leading zeros in the linked list. In
 * this implementation, calling BigInt("00042") creates a
 * BigInt with a different internal representation than
 * BigInt("42"), which is probably a bad idea.
 */

BigInt::BigInt(string str) {
    if (str == "" || str == "-") error("BigInt: illegal format");
    start = NULL;
    sign = 1;
    if (str[0] == '-') {
        sign = -1;
        str = str.substr(1);
    }
    for (int i = 0; i < str.length(); i++) {
        char ch = str[i];
        if (!isdigit(ch)) error("BigInt: illegal format");
        Cell *cp = new Cell;
        cp->digit = ch - '0';
        cp->link = start;
        start = cp;
    }
}
```
/*
 * Implementation notes: BigInt destructor
 *
 * The code for the destructor is similar to that of the other
 * classes that contain a linked list. You need to store the
 * pointer to the next cell temporarily so that you still have
 * it after you delete the current cell.
 */

BigInt::~BigInt() {
    Cell *cp = start;
    while (cp != NULL) {
        Cell *next = cp->link;
        delete cp;
        cp = next;
    }
}

/*
 * Implementation notes: toString
 *
 * This method could also be written as a wrapper method that
 * calls a recursive function that creates the reversed string
 * one character at a time. I've used an iterative formulation
 * here to avoid having to declare the private method in the
 * bigintpriv.h file.
 */

string BigInt::toString() {
    string str = "";
    for (Cell *cp = start; cp != NULL; cp = cp->link) {
        str = char(cp->digit + '0') + str;
    }
    if (sign == -1) str = "-" + str;
    return str;
}