Problem 1 [10 points]

A [2 points]

Answer = No

B [3 points]

Answer = 0, 6, 7, 2, 3, 1
Lose 1 point for each mistake

C [2 points]

Answer = O(N)

D [3 points]

Answer = [*empty*, 1, 2, 3, 6, 7]
Array can be zero indexed (like pqueue assignment) or one indexed
Lose 1 point for each mistake.

Problem 2: Rubik's Cube [10 points]

Solution

```
bool isSolvable(Cube & cube, CubeSet & seen) {
   if(isSolution(cube)) return true;
   if(seen.contains(cube)) continue;
   seen.add(cube);

   for(int i = 0; i < 6; i++) {
      // rotate clockwise
      rotate(cube, i, true);
      if(isSolvable(cube, seen)) return true;
      rotate(cube, i, false);

      // technically don't need counter clockwise
   }
   return false;
}
```

Base Case [4 points]

Error: Doesn't check if current cube is a solution
Error: Doesn't check if current cube is in the seen set
Error: Doesn't update seen set
Error: Doesn't properly return in any of the base cases.

Each error loses one point.

Recursive Case [6 points]

Major Error: Doesn't loop over each face
Major Error: Doesn't rotate the cube
Major Error: Doesn't make a recursive call
Major Error: Doesn't return true if the recursion returns true.
Major Error: Doesn't backtrack the move

Minor Error: doesn't return false if all recursive cases fail

Note: since three clockwise moves is a counter clockwise move, technically only need one of the two directions

Perfect: 6 points	1 major: 4 points	attempt: 1 point
1 minor: 5 points	2 major: 2 points	else: 0 points

Problem 3: Sparse Grid [10 points]

Solution

```
class SparseGrid {
public:
   SparseGrid(int rows, int cols);
   void put(int row, int col, double value);
   double get(int row, int col);
private:
   void checkBounds(int row, int col);
   int rows;
   int cols;
   Map<string, double> values;
};

SparseGrid::SparseGrid(int rows, int cols) {
   this->rows = rows;
   this->cols = cols;
}

void SparseGrid::checkBounds(int row, int col) {
   if(row<0 || col <0) throw "out of bounds";
   if(row>=rows || col>=cols) throw "out of bounds";
}

string getKey(int row, int col) {
   return integerToString(row) + "," + col;
}

void SparseGrid::put(int row, int col, double value) {
   checkBounds(key, value);
   string key = getKey(row, col);
   values[key] = value;
}

double SparseGrid::get(int row, int col) {
   checkBounds(key, value);
   string key = getKey(row, col);
   if(!values.contains(key)) return 0; //not necessary
   return values[key];
}
```

Problem 3: Sparse Grid Continued [10 points]

Bounds Checking [3 points]

Major error: No instance variable to store bounds
Major error: Doesn't attempt to set bounds i-var in constructor
Major error: Doesn't checks out of bounds

Minor error: Forgets the "this" pointer in a case where it is necessary.

0 errors: 3 points
1 minor: 2 points
1 major: 1 point
Worse: 0 points

Key Translation [2 points]

Way to translate (row,col) to keys. There are two options, either use a string representation of the keys or to use the C++ stl pair. Award students either full credit for a perfect solution. 1 point for an attempt (with any mistake, eg forgetting integerToString) and 0 else.

Get and Set [5 points]

Major error: No data structure instance variable
Major error: Data structure instance variable with lookup time worse than log N
Major error: Doesn't gets and/or puts from data structure

Minor error: Returns 0 if key not found (note that this is the default for looking up a key not in the map)

Note: no student should get 3 points

0 errors: 5 points
1 minor: 4 points
1 major: 2 points
Attempt: 1 point
Worse: 0 points

Problem 4: Salty Passwords [10 points]

```
void addUser(string user, string password, Map<string,string> &db){
   string saltyPassword = integerToString(hashSum(password + SALT));
   db[user] = saltyPassword;
}

bool logIn(string user, string password, Map<string,string> &db) {
   if(!db.contains(user)) return false;
   string saltyPassword = integerToString(hashSum(password + SALT));
   return db[user] == saltyPassword;
}
```

Hash + Salt [5 points]

Major error: uses hashrandom
Major error: forgets to add SALT

Minor error: forgets to use integer to string
Minor error: uses hashlength

Perfect: 5 points
1 minor: 4 points
2 minors: 3 points
1 major: 2 points
Attempt: 1 point
Else: 0 points

Maintain Database [4 points]

Major error: Has user as key, password as value
Major error: Doesn't put into database
Major error: Doesn't get from database
Major error: Uses external database (not db)

Perfect: 4 points
1 major: 2 points
Attempt: 1 point
Else: 0 points

Check User Exists [1 point]

Either they got it perfect (1 point) or they didn't (0 points)

Problem 5: Narcissistic Trees [10 points]

```
Tree * reflect(Tree * original) {
   Tree * copy = new Tree;
   copy->value = original->value;
   for(Tree * child : children) {
      copy->children.insert(0, reflect(child));
   }
   return copy;
}
```

Deep Copy [3 points]

Major error: doesn't make a new tree
Major error: doesn't copy over the value of the original.

Perfect: 3 points
1 Major: 1 point
2 Major: 0 points

Recursion [6 points]

Major error: doesn't loop over children
Major error: doesn't call recursive reflection on each child
Major error: doesn't have a mechanism to reverse the order
Major error: armslength recursion related error

Perfect: 6 points
1 Major: 4 points
2 Major: 2 points
Attempt: 1 point
Else: 0 points

The Return [1 point]

Either they return the tree * that they intended to be the reflection (1 point) or they didnt (0 points)

Problem 6: Seam Carving [12 points]

Solution 1: Dijkstra over Pixels

```
Vector<Pixel *> getBestSeam(Grid<Pixel *> & image) {
   PriorityQueue<Vector<Pixel *> > queue;
   for(int i = 0; i < image.numCols(); i++) {
      Vector<Pixel *> start;
      start.add(image[0][i]);
      queue.enqueue(start, image[0][i]->cost);
   }
   while(!queue.isEmpty()) {
      double cost = queue.peekPriority();
      Vector<Pixel *> currPath = queue.dequeue();
      Pixel * currNode = currPath[currPath.size() - 1];
      // notice: no need for a seen set
      if(currNode->row == image.numRows() -1) {
         return currPath;
      }
      for(int i = -1; i <= +1; i++) {
         if(!image.inBounds(row + 1, col + i)) continue;
            Pixel * neighbor = image[row + 1][col + i];
            Vector<Pixel *> newPath = currPath;
            newPath.add(neighbor);
            newPath.add(newPath, cost + neighbor->cost);
         }
      }
   }
}
```

Problem 6: Seam Carving Continued [12 points]

Solution 2: Make Graph and Call Dijkstra

```
string name(int r, int c) {
   return integerToString(r) + "," + c;
}

Vector<Pixel *> getBestSeam(Grid<Pixel *> & image) {
   BasicGraph graph;
   graph.addVertex("start");
   graph.addVertex("sink");
   for(int r = 0; r < image.numRows(); r++) {
      for(int c = 0; c < image.numCols(); c++) {
         graph.addVertex(getName(r, c))
      }
   }

   // add edges
   for(int r = 0; r < image.numRows() - 1; r++) { //note the -1
      for(int c = 0; c < image.numCols(); c++) {
         for(int dc = -1; dc <= +1; dc++) {
            if(!image.inBounds(r, c + dc)) continue;
            Edge * e = graph.addEdge(name(r, c), name(r + 1, c + dc));
            e->setWeight(image[r + 1][c + dc]->importance);
         }
      }
   }
   for(int c = 0; c < image.numCols(); c++) {
      Edge * sourceEdge = graph.addEdge("start", name(0, c));
      sourceEdge->setWeight(image[0][c]->importance);
      Edge * sinkEdge = graph.addEdge(name(image.numRows()-1, c),
"sink");
      sinkEdge->setWeight(0);
   }

   // run dijkstra
   Vector<Pixel*> soln = dijkstra(graph, "start", "sink");
   soln.remove(soln.size() - 1);
   soln.remove(0);
   return soln;
}
```

Dijkstra Over Pixels [12 points]

Major Bugs

- Doesn’t have a mechanism to search for paths starting at each top pixel.
- Doesn't use a PQ
- Priority is seriously wrong.
- In update: doesn’t check pixels that are below + adjacent to the current end of path
- No attempt to track path/previous in some form, so can't reconstruct path
- Returned path is empty, corrupted, or very different from what is found by the algorithm
- Doesn’t recognize reaching the bottom row as an end condition

Minor Bugs

- Attempts but updates node's cost incorrectly (should be costSoFar + importance of the neighbor)
- uses integers rather than doubles to track the cost
- returned path is backwards
- returned path is malformed in a minor way; e.g. contains a duplicate element, skip an element, etc.
- doesn’t check if the path goes out of bounds of the image.
- In update: also makes paths with pixels above or to the side

Minor Bugs on non-path algorithm

- Does not set cost of nodes to infinity at beginning
- Changes cost of node even if not better path
- Doesn't update previous pointer when changing node's priority
- Lowers priority of node not in the priority queue

Score (2 minors = 1 major)

Perfect:	12 points
1 Minor	11 points
1 Major:	9 points
2 Major:	7 points
3 Major:	5 points
3 Major + 1 Minor:	3 points
4 Major:	1 point

Problem 6: Graph + Implemented Dijkstra [12 points]

Major Bugs

- Doesn't have a mechanism to search for paths starting at each top pixel.
- Doesn't have a mechanism to search for paths that end at any bottom pixel
- Makes paths that can have more than one pixel per row (eg connects a node to all neighbors, not just the ones below)
- Doesn't add edge weights

Minor Bugs

- Doesn't add vertices before edges
- Edge weights are somehow inconsistent with pixel weights
- Doesn't fix path so that it doesn't include a "sink" or "source" node

Score (2 minors = 1 major)

Perfect:	12 points
1 Minor	10 points
1 Major:	9 points
2 Major:	6 points
3 Major:	3 points
Attempt:	1 point

Problem 7: Dutch Social Network

```
class Network {
private:
   struct Node {
      string name;
      Node * parent;
      int size;
   };
   Map<string, Node *> nodeMap;
};

void Network::nieuw(string user) {
   nodeMap[user] = new Node;
   nodeMap[user]->name = user;
   nodeMap[user]->size = 1;
}

void Network::betonen(string userA, string userB) {
   Node * rootA = getRoot(nodeMap[userA]);
   Node * rootB = getRoot(nodeMap[userB]);
   if(rootA->size > rootB->size) {
      rootA->parent = rootB;
      rootA->size += rootB->size;
   } else {
      rootB->parent = rootA;
      rootB->size += rootA->size;
   }
}

void Network::aangesloten(string userA, string userB) {
   Node * rootA = getRoot(nodeMap[userA]);
   Node * rootB = getRoot(nodeMap[userB]);
   return rootA == rootB
}

void Network::getRoot(Node * node) {
   if(node->parent == NULL) {
      return node;
   }
   return getRoot(node->parent);
}
```

Problem 7: Minimal Social Network Continued [8 points]

Major Bugs

- Doesn't keep track of tree size to make sure that tree remains balanced
- In add friend: doesn't connect root of the connected component
- In isConnected: doesn't check if root is the same
- Doesn't have O(log N) mechanism to look up tree node from name

Minor Bugs

- Doesn't update size when connecting trees.

Score (2 minors = 1 major)

Perfect:	8 points
1 minor	6 points
1 major	5 points
1 major + 1 minor	3 points
2 majors:	2 points
Worse:	0 points