CS106B Handout 25S
Winter 2019 March 13, 2019

CS106B Final Exam Solutions

This handout contains solutions to the practice final exam. Please, please, please, please, *please* do not read over any of the solutions here until you have a working solution for each problem on the practice exam. If you're stuck on one of these problems and aren't sure how to proceed, ping your section leader, ask a question on Piazza, or visit the CLaIR to get some hints and suggestions. Once you've seen the solution, you can't "unsee" it. Ultimately, it's more important to have a deep understanding of how to come up these solutions than it is to just see some code.

Problem One: Recursive Problem-Solving

Rigging a Tournament

There are many ways to solve the first part of this problem. Here are three options.

```
/* To find the winner of a tournament, split the tournament in half. Find the
 * winner in the first and second halves, then have them play a game against
 * one another.
 */
string overallWinnerOf(const Vector<string>& initialOrder) {
    /* Base Case: If there’s one player left, that player wins! */
    if (initialOrder.size() == 1) return initialOrder[0];

    int half = initialOrder.size() / 2;
    return winnerOf(overallWinnerOf(initialOrder.subList(0, half)),
                    overallWinnerOf(initialOrder.subList(half, half)));
}

                                    /* … or … */

/* Pair off the players so that each plays a game against the next player in the
 * ordering, then form an elimination tournament from those players in the same
 * relative order and see who wins!
 */
string overallWinnerOf(const Vector<string>& initialOrder) {
    /* Base Case: If there’s one player left, that player wins! */
    if (initialOrder.size() == 1) return initialOrder[0];

    Vector<string> nextRound;
    for (int i = 0; i < initialOrder.size(); i += 2) {
        nextRound += winnerOf(initialOrder[i], initialOrder[i + 1]);
    }

    return overallWinnerOf(nextRound);
}

                                    /* … or … */

/* Treat the Vector like a Queue! Pull off the first two players, have them play
 * a game against one another, then put the winner on the back. This process will
 * pair off the players in the same order as required by the tournament bracket.
 */
string overallWinnerOf(const Vector<string>& initialOrder) {
    /* Base Case: If there’s one player left, that player wins! */
    if (initialOrder.size() == 1) return initialOrder[0];

    /* Drop off the first two players. */
    auto nextRound = initialOrder.sublist(2, initialOrder.size() - 2);

    /* Put the winner on the back. */
    nextRound += winnerOf(initialOrder[0], initialOrder[1]);
    return overallWinnerOf(nextRound);
}
```

The second part of this problem is essentially a permutations problem: we just list off all possible ways to order the players and see if our favorite player ever wins!

```
bool canRigFor(const string& player, const Set<string>& allPlayers,
               Vector<string>& initialOrder) {
    return canRigRec(player, allPlayers, initialOrder, {});
}

bool canRigRec(const string& player, const Set<string>& allPlayers,
               Vector<string>& initialOrder, const Vector<string>& soFar) {
    /* Base Case: If everyone is already placed, see if our player wins! */
    if (allPlayers.isEmpty()) {
        if (overallWinnerOf(soFar) == player) {
            initialOrder = soFar;
            return true;
        }
        return false;
    }
    /* Recursive case: Try all possible next players. */
    for (string nextPlayer: allPlayers) {
        auto nextOrder = soFar;
        nextOrder += nextPlayer;
        if (canRigRec(player, allPlayers - nextPlayer, initialOrder, nextOrder)) {
            return true;
        }
    }
    /* Oh well, guess it's not possible. */
    return false;
}
```

Why we asked this question: We chose this question primarily to make sure that you were comfortable with recursive backtracking and recursive problem-solving strategies. The first part of this problem was designed to assess whether you were comfortable looking at a tree structure and deducing some sort of recursive pattern from it, and we hoped that the fact that there are several different solution routes you can choose from would make that part a good warm-up. The second part of this problem is essentially just a permutations problem, and we hoped that you'd be comfortable looking over the structure of the problem and recognizing this particular detail.

Fun fact: this problem is based on some research done by one of my former CS103 TAs, Michael Kim, who is currently a Ph.D student here.

Common mistakes: Most people got the first part of this problem right, or otherwise had a solution that was almost entirely correct. Nice job! The most common mistakes we saw were choosing the wrong base case (for example, assuming there were always at least two players) or indexing errors in the `Vector`.

For the second part of the problem, by far the most common mistake we saw was not recognizing that this problem is fundamentally a permutations problem. Solutions that didn't approach the problem this way typically were much more complicated and didn't correctly try all the necessary options.

Problem Two: Linear Structures

Self-Organizing Lists

Here's one possible solution:

```
class MoveToFrontSet {
public:
    MoveToFrontSet();
    ~MoveToFrontSet();
    bool contains(const string& str);
    void add(const string& str);
    void delete(const string& str);
private:
    struct Cell {
        string value;
        Cell* next;
    };
    Cell* head;
}

/* Constructor makes the head null to signify that no elements are present. */
MoveToFrontSet::MoveToFrontSet() {
    head = nullptr;
}

/* Destructor is our typical “deallocate a linked list” destructor. */
MoveToFrontSet::~MoveToFrontSet() {
    while (head != nullptr) {
        Cell* next = head->next;
        delete head;
        head = next;
    }
}

bool MoveToFrontSet::contains(const string& str) {
    Cell* prev = nullptr;
    Cell* curr = head;

    /* Scan the list, keeping track of the current pointer and previous pointer,
     * until we find what we want or fall off the list.
     */
    while (curr != nullptr && curr->value != str) {
        prev = curr;
        curr = curr->next;
    }

    /* If we didn’t find it, curr will be null since we walked off the list. */
    if (curr == nullptr) return false;

    /* If we found it and it’s not at the head of the list, move that element to
     * the front of the list.
     */
    if (curr != head) {
        prev->next = curr->next;
        curr->next = head;
        head = curr;
    }
    return true;
}
```

```
void MoveToFrontSet::add(const string& str) {
    /* If this element already exists, we're supposed to move it to the front.
     * That's automagically handled for us by the contains call!
     */
    if (contains(str)) return;

    /* Put a new cell at the front of the list. */
    Cell* cell = new Cell;
    cell->value = str;
    cell->next = head;
    head = cell;
}

void MoveToFrontSet::remove(const string& str) {
    /* See if the element is here. If not, there's nothing to do. */
    if (!contains(str)) return;

    /* Nifty fact: the element to remove is now at the front of the list, since
     * looking for it put it there! So just take it off the front.
     */
    Cell* toRemove = head;
    head = head->next;
    delete toRemove;
}
```

Why we asked this question: We included this question for a number of reasons. First, we wanted to give you a chance to demonstrate what you'd learned about class design and working with linked lists. We figured this particular problem worked well because it involved linked list manipulations (along the lines of what you did in the Priority Queue assignment) and the idea of having different member functions call one another. Second, we thought this particular linked list exercise of splicing out a node from a singly-linked list and moving it to another location would allow you to demonstrate whether you were comfortable with the idea of maintaining two pointers into a linked list (something you likely needed in the course of implementing the singly-linked list priority queue) and of rewiring cell pointers. Finally, we thought this problem was interesting in of itself. This is an example of a *self-adjusting data structure*, and this particular structure is often used in data compression (look up *move-to-front encoding*). It's also related to the more popular *splay tree*, an extremely fast and simple binary search tree data structure.

Common mistakes: We saw a number of solutions that contained memory errors, such as allocating cells unnecessarily (often, pointers were initialized to `new Cell` rather than `nullptr`) or reading from a cell after deleting it.

Many solutions attempted to implement contains in terms of insertion and deletion rather than the other way around. While in principle this works, it's not at all efficient (it's much faster to reorder existing linked list cells than it is to produce new cells from scratch) and makes the logic a lot trickier and therefore more error-prone.

Problem Three: Tree Structures

Agglomerative Clustering

There are many ways to implement the `leavesOf` function. The first solution works by returning values upward through the recursion, and the second solution works by filling an outparameter:

```
Set<double> leavesOf(Node* root) {
    /* Base Case: The set of values in a leaf node is just the one value here. */
    if (root->left == nullptr) return { root->value };

    /* Recursive Step: Combine the sets from the left and right subtrees. */
    return leavesOf(root->left) + leavesOf(root->right);
}
```

```
Set<double> leavesOf(Node* root) {
    Set<double> result;
    leavesOfRec(root, result);
    return result;
}

void leavesOfRec(Node* root, Set<double>& result) {
    if (root->left == nullptr) {
        result += root->value;
    } else {
        leavesOfRec(root->left,  result);
        leavesOfRec(root->right, result);
    }
}
```

Here's one possible implementation of the clustering code:

```
/* Helper struct representing a pair of nodes. */
struct NodePair {
    Node* first;
    Node* second;
};

Set<Node*> cluster(const Set<double>& values, int numClusters) {
    Set<Node*> trees = makeSingletons(values);

    /* Keep merging trees until we have the specified number required. */
    while (trees.size() != numClusters) {
        /* Find the closest pair of trees. */
        NodePair closest = closestTreesIn(trees);

        /* Merge them together. */
        Node* root  = new Node;
        root->left  = closest.first;
        root->right = closest.second;
        root->value = averageOf(leavesOf(root));

        /* Remove the old trees from the set. */
        trees -= closest.first;
        trees -= closest.second;

        /* Add in this new tree. */
        trees += root;
    }

    return trees;
}
```

```
/* Computes the average of a set of doubles. */
double averageOf(const Set<double>& values) {
    double total = 0.0;
    for (string value: values) {
        total += value;
    }
    return total / values.size();
}

/* Given a set of values, forms a set of singleton trees from those values. */
Set<Node*> makeSingletons(const Set<double>& elems) {
    Set<Node*> result;
    for (double val: elems) {
        Node* singleton = new Node;
        singleton->value = val;
        singleton->left = singleton->right = nullptr;
        result += singleton;
    }
    return result;
}

/* Given a pair of doubles, returns the distance between them. */
double distance(double one, double two) {
    /* Distance is the absolute value of their difference. */
    return fabs(one - two);
}

/* Given a set of trees, returns the two trees with the closest roots. */
NodePair closestTreesIn(const Set<Node*>& trees) {
    NodePair result;
    double bestDistance = INFINITY;

    for (Node* one: trees) {
        for (Node* two: trees) {
            if (one != two) { // Don't merge a tree with itself!
                if (distance(one->value, two->value) < bestDistance) {
                    result.first = one;
                    result.second = two;
                    bestDistance = distance(one->value, two->value);
                }
            }
        }
    }

    return result;
}
```

Why we asked this question: As the title suggests, this question was designed to see how comfortable you've become working with trees and tree structures. The first part of this problem was a tree recursion problem to make sure you were comfortable with the idea of exploring all the nodes in a tree. The second part of this problem (for which I have to give credit to Chris Piech for the concept) was designed to let you demonstrate what you'd learned about bottom-up tree assembly. We hoped that the core idea – find two close trees, merge them, repeat – would let you show what you'd learned about collections of tree nodes, assembling trees bottom-up, and reporting multiple values across different function calls.

It turns out that with the right data structures you can implement this algorithm so that it runs in time O(n log n). Take CS166 if you're curious how to do this!

Common mistakes: For the first part of this problem, the most common mistake we saw was messing up the base case, often by not stopping the recursion when the base case is triggered. We also saw several solutions that treated a node with two children as the base case, which will cause errors when giving a single leaf node as input. We also saw a number of solutions that included a lot of unnecessary if statements to check the left and right subtrees independently after establishing the node isn't a leaf, which wasn't wrong per se but isn't something that really should be included.

For the second part of the problem, we saw a lot of solutions that forgot to initialize the child pointers of newly-allocated singleton nodes, leading to memory issues with the returned trees. We also saw a lot of minor off-by-one or scoping errors in the logic to find the two trees with the closest roots, most of which are pretty easily correctable. From an algorithmic standpoint, many solutions worked by computing the average of the *roots* of the two merged trees rather than the *leaves*, which produces answers that aren't quite in line with what the algorithm should produce. Finally, we saw a number of solutions that had minor issues when removing the two trees that were merged together from the overall collection, either due to simply forgetting to check this or due to removing by index from a `Vector`, forgetting that one of the two indices will be incorrect after one element is removed.

Problem Four: Graphs and Graph Algorithms

Finding All Shortest Paths

There are many possible solutions to this problem. Here are two:

```
Map<string, int> distancesFrom(const string& start, const Map<string, Set<string>>& graph) {
    Queue<Vector<string>> worklist;
    worklist.enqueue({start});

    Set<string> visited = { start };
    Map<string, int> result;
    result[start] = 0;

    while (!worklist.isEmpty()) {
        Vector<string> path = worklist.dequeue();
        string last = path[path.size() - 1];
        for (string next: graph[last]) {
            if (!visited.contains(next)) {
                visited.add(next);
                result[next] = path.length();

                Vector<string> nextPath = path;
                nextPath += next;
                worklist.enqueue(nextPath);
            }
        }
    }
    return result;
}
```

```
Map<string, int> distancesFrom(const string& start, const Map<string, Set<string>>& graph) {
    Queue<Vector<string>> worklist;
    worklist.enqueue({start});

    Set<string> visited = { start };
    Map<string, int> result;

    while (!worklist.isEmpty()) {
        Vector<string> path = worklist.dequeue();
        string last = path[path.size() - 1];

        result[last] = path.size() - 1;
        for (string next: graph[last]) {
            if (!visited.contains(next)) {
                visited.add(next);

                Vector<string> nextPath = path;
                nextPath += next;
                worklist.enqueue(nextPath);
            }
        }
    }
    return result;
}
```

For the final part, we just do a standard exhaustive recursive search.

```
Vector<Vector<string>>
allShortestPathsBetween(const string& start, const string& end,
                        const Map<string, Set<string>>& graph) {
    Map<string, int> distances = distancesFrom(start, graph);

    Vector<Vector<string>> result;
    recFindShortestPaths(end, start, graph, distances, result, { end });
    return result;
}

void recFindShortestPaths(const string& end, const string& start,
                          const Map<string, Set<string>>& graph,
                          const Map<string, int>& distances,
                          Vector<Vector<string>>& result,
                          const Vector<string>& soFar) {
    /* Base Case: If we made it to the end, reverse the path. We're done! */
    if (soFar[soFar.size() - 1] == start) {
        Vector<string> reversed;
        for (int i = soFar.size() - 1; i >= 0; i--) {
            reversed += soFar[i];
        }
        result += reversed;
        return;
    }

    /* Recursive Step: Take all possible steps that are in the right direction. */
    for (string next: graph[end]) {
        if (distances[next] < distances[end]) {
            auto nextPath = soFar;
            nextPath += next;
            recFindShortestPaths(next, start, graph, distances, result, nextPath);
        }
    }
}
```

Why we asked this question: We chose this question to see whether you had built up a good intuition for the different graph algorithms we've talked about this quarter. The first part of this question was designed to see whether you were comfortable taking a working implementation of a common graph algorithm (here, breadth-first search) and modifying it to get out extra information. In practice, it's common to take core graph algorithms like BFS and DFS and to tweak or modify them to get them to produce extra information. The second part of this question was designed as a hybrid problem that explored recursive enumeration and graph searches. Our intent was that you could demonstrate that you'd built up a good intuition for how graph search algorithms generate paths incrementally.

Common mistakes: On part (i) of this problem, one of the most common mistakes we saw was maintaining a global "hops" counter that was incremented on each iteration and used to store the distance to each node. This approach breaks down if you dequeue multiple elements from the queue that are at the same distance to the root (say, one hop away), since the distances assigned to the nodes adjacent to *those* nodes will then be different based on the order in which they're dequeued. We also saw a number of solutions that forgot to assign a distance to the start node.

On part (ii) of this problem, we saw many solutions that formed paths that didn't consist of adjacent nodes, either by iterating over every node in the graph (not just the ones adjacent to the current node) or by maintaining a collection of possible paths and extending each of those paths with a node adjacent to *any* path. We also saw a number of smaller type errors like mixing and matching nodes, paths, and lists of paths, and a number of efficiency issues like recomputing the distances from the start node at each step. Finally, we saw many solutions that had simple indexing issues reversing paths at the final step.