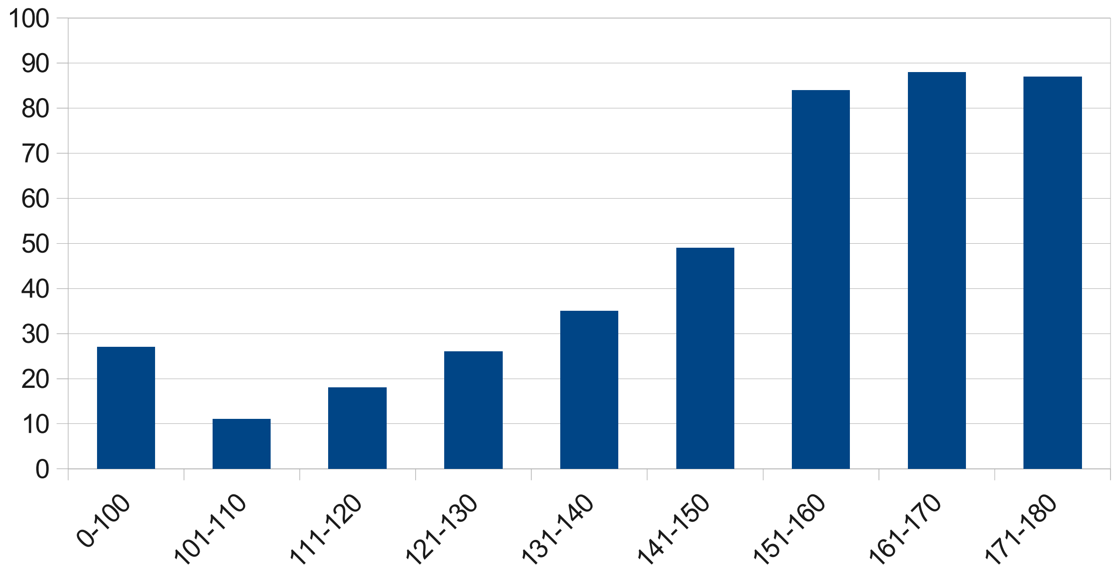
CS106B
Spring 2013

Handout #25S
June 3, 2013

CS106B Midterm Exam #2 Solutions

Below is the score distribution for the first midterm exam:



Overall, the final statistics were as follows:

Mean: 148 / 180 (82.2%)

Median: 156 / 180 (86.7%)

Standard Deviation: 29 (16%)

We are **not** grading this course using raw point totals and will instead be grading on a (fairly generous) curve. Roughly speaking, the median score corresponds to roughly a B/B+. As always, if you have any comments or questions about the midterm or your grade on the exam, please don't hesitate to drop by office hours with questions! You can also email Dawson or Keith with any questions.

If you think that we made any errors in our grading, please feel free to submit a regrade request to us. Just write a short (one-paragraph or so) description of what you think we graded incorrectly, staple it to the front of your exam, and hand your exam to either Dawson or Keith by Monday, June 10 at 2:15PM.

Regrade requests should probably be for cases where we have made an error while grading the exam (such as marking correct code as incorrect or totaling points incorrectly). We grade based on a criteria and try to apply deductions consistently, so in the interest of fairness we usually don't make local adjustments to point deductions.

Problem One: Linked Lists (30 Points)

```
Node* concatenateCircularlyLinkedLists(Node* first, Node* second) {
    lastOf(first)->next = second;
    lastOf(second)->next = first;
    return first;
}

Node* lastOf(Node* list) {
    Node* result = list;
    while (result->next != list) result = result->next;
    return result;
}
```

Common mistakes included trying to rewire the lists by changing local variables pointing to the cells rather than the contents of those cells, accidentally reusing the pointer that found the end of the first list to find the end of the second list without first modifying its location, and leaving off the return statement.

Problem Two: Hash Tables (40 Points)

```
bool OurHashMap::remove(string key) {
    int bucket = hashCode(key) % numBuckets;
    Cell* prev;
    Cell* curr;

    for (prev = NULL, curr = buckets[bucket]; curr != NULL;
         prev = curr, curr = curr->next) {
        if (curr->key == key) break;
    }

    if (curr == NULL) return false;
    if (prev == NULL) {
        buckets[bucket] = curr->next;
    } else {
        prev->next = curr->next;
    }

    delete curr;
    numElems--;
    return true;
}
```

Common mistakes included forgetting to decrement **numElems** at the end of the function, incorrectly handling the case where the element to remove was the first in its bucket, or deleting the cell without correctly rewiring the rest of the linked list around it.

Problem Three: Binary Search Trees (30 Points)

```
/* Version 1: Recursive traversal. */
Node* lowestCommonAncestorOf(Node* root, Node* v1, Node* v2) {
      if (v1->value < root->value && v2->value < root->value)
            return lowestCommonAncestorOf(root->left, v1, v2);
      if (v2->value > root->value && v2->value > root->value)
            return lowestCommonAncestorOf(root->right, v1, v2);
      return root;
}

// ---------------------------------------------------------------------------- //

/* Version 2: Storing access paths. */
Node* lowestCommonAncestorOf(Node* root, Node* v1, Node* v2) {
      Vector<Node*> ap1 = accessPathFor(root, v1);
      Vector<Node*> ap2 = accessPathFor(root, v2);

      for (int i = min(ap1.length(), ap2.length()) - 1; i >= 0; i--) {
            if (ap1[i] == ap2[i]) {
                  return ap1[i];
            }
      }
}
Vector<Node*> accessPathFor(Node* root, Node* v) {
      Vector<Node*> result;
      while (true) {
            result += root;
            if (v == root) return result;
            if (v->value < root->value) root = root->left;
            else root = root->right;
      }
}

// ---------------------------------------------------------------------------- //

/* Version Three: Using node depths. */
Node* lowestCommonAncestorOf(Node* root, Node* v1, Node* v2) {
    Map<Node*, int> depths1 = nodeDepthsFor(root, v1);
    Map<Node*, int> depths2 = nodeDepthsFor(root, v2);

    Node* deepest;
    int maxDepth = -1;

    foreach (Node* v in depths1) {
       if (depths2.containsKey(v) && depths1[v] > maxDepth) {
           deepest = v;
           maxDepth = depths[1];
       }
    }
    return deepest;
}
```

Common mistakes included accidentally reading **NULL** pointers, treating the lowest common ancestor as the common ancestor with the smallest/greatest value, not handling the case where one node was an ancestor of the other, or not handling the case where the nodes were at different depths.

Problem Four: Grid (50 Points)

```
private:
   int** elems;
   int numRows, numCols;
};

LazyGrid::LazyGrid(int nRows, int nCols) {
    if (nRows < 0 || nCols < 0) error("Can't make a negative-sized grid.");
    numRows = nRows;
    numCols = nCols;

    elems = new int*[numRows];
    for (int i = 0; i < numRows; i++) {
        elems[i] = NULL;
    }
}

LazyGrid::~LazyGrid() {
    for (int i = 0; i < numRows; i++) {
        delete[] elems[i];
    }
    delete[] elems;
}

int LazyGrid::getAt(int row, int col) {
    if (row < 0 || col < 0 || row >= numRows || col >= numRows) {
        error("Out of bounds!");
    }
    return elems[row] != NULL? elems[row][col] : 0;
}

void LazyGrid::setAt(int row, int col, int value) {
    if (row < 0 || col < 0 || row >= numRows || col >= numRows) {
        error("Out of bounds!");
    }
    if (elems[row] == NULL) {
        elems[row] = new int[numCols];
        for (int col = 0; col < numCols; col++) {
            elems[row][col] = 0;
        }
    }
    elems[row][col] = value;
}
```

Common mistakes included forgetting to initialize the rows to zero when creating them, forgetting to perform bounds-checking, deallocating the top-level array without deallocating the individual arrays, or deallocating the individual arrays without deallocating the top-level array.

Problem Five: Relatively Short Answer (30 Points)

(i) Dynamic Arrays and Linked Lists (10 Points)

Reasons to prefer dynamic array over a linked list:

- Faster overall performance: dynamic arrays are amortized O(1) and the hidden constant is lower than that of a linked list.
- If the stack reaches a maximum size and is continuously pushed and popped, only a finite number of allocations will ever be performed.
- When full, there is minimal space overhead (just the size/capacity fields and a pointer to the elements.)

Reasons to prefer a linked list over a dynamic array:

- Worst-case efficient, not just amortized efficient.
- Dynamic arrays will leave behind a huge amount of empty space if filled then emptied.
- Lower space overhead if elements in the linked list are large (one pointer overhead per element, versus one extra element overhead per element in a dynamic array).

(ii) Hash Tables and BSTs (10 Points)

Reasons to prefer a hash table over a balanced binary search tree:

- Faster average-case performance than a balanced binary search tree.
- Does not require a comparison function.
- Slightly more space efficient: if you have *n* elements and *n*/2 buckets, there will be roughly 3/2 pointers per element stored.

Reasons to prefer a balanced binary search tree over a hash table:

- Better worse-case performance (if a bad hash function is picked, hash table operations degrade to O(*n*)).
- Stores elements in sorted order.
- Does not require a hash function.
- Worst-case efficient, not just amortized efficient.

(iii) Treesort (7 Points)

Treesort is closest to **quicksort**. When the first element is inserted into the BST, it acts as a pivot element – all smaller elements go to the left and all larger elements go to the right – where they are then recursively partitioned and subdivided further. The runtimes of quicksort and treesort share similar characteristics: quicksort is best-case O(*n* log *n*), as is treesort (*n* insertions taking O(log *n*) time each), and has worst-case O(n^2) runtime, as does treesort (*n* insertions taking O(*n*) time each). The worst cases occur in the same situations, which occur when there is a bad pivot (one in which the pivot / inserted element is close to the maximum / minimum values).

Insertion sort is similar to treesort in that at each step in the algorithm the tree stores a sorted sequence formed from the starting elements of the input, but beyond that they are quite different – their runtimes, best cases, and worst cases are not the same. (Try running treesort on a sorted sequence; insertion sort would run in time O(*n*) here, while treesort runs in time O(n^2).

(iv) CS106B: The Sequel **(3 Points)**

If you could give a one-sentence suggestion to future CS106B students, what would it be? (We'll give you full credit regardless of what your answer is, as long as it's one sentence long!)

The Most Common Responses:

Start assignments early.

Go to the LaIR and office hours with questions.

Attend lectures.

Draw pointer diagrams when you get stuck!

Recursion shows up <u>everywhere</u>!