

Answers to Practice Midterm Exam #2

Midterm #1: Tuesday, February 10, 2:15–4:15 P.M., Annenberg Auditorium

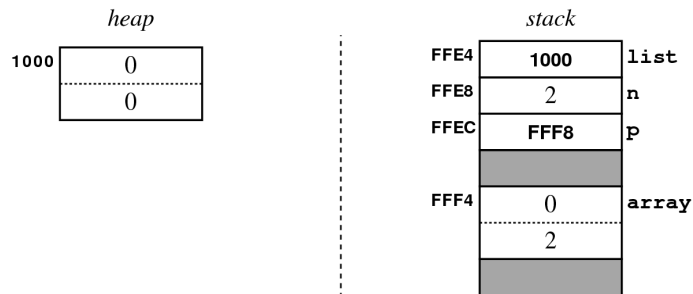
Midterm #2: Tuesday, February 10, 7:00–9:00 P.M., CEMEX Auditorium

Problem 1: Function tracing and/or memory diagramming (10 points)

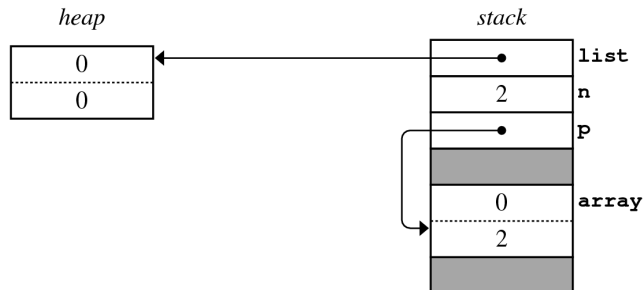
There was a small error in question 1 of the second practice exam (Handout #30) as it was distributed in class on Friday. The first line in the function `puzzle` should contain an asterisk before the variable name `list`, as follows:

```
int *list = new int[n];
```

explicit addresses



arrow representation



Problem 2: Vectors, grids, stacks, and queues (10 points)

```

/*
 * Function: extract3x3Subsquare
 * Usage: Vector<int> subsquare = extract3x3Subsquare(grid, bigRow, bigCol);
 * -----
 * Returns a vector containing the nine elements in the 3x3 subsquare
 * indicated by bigRow and bigCol. Subsquares are numbered in row-major
 * order starting from the upper left corner.
 */

Vector<int> extract3x3Subsquare(Grid<int> & grid, int bigRow, int bigCol) {
    Vector<int> result;
    int r0 = 3 * bigRow;
    int c0 = 3 * bigCol;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result.add(grid[r0 + i][c0 + j]);
        }
    }
    return result;
}

```

Problem 3: Lexicons, maps, and iterators (15 points)

There are several strategies that you could use to implement this problem. I believe that the simplest strategy is to calculate the result recursively, as follows:

```

/*
 * Function: countHailstoneSteps
 * Usage: int nSteps = countHailstoneSteps(n, cache);
 * -----
 * Returns the number of steps in the hailstone sequence beginning
 * at n. The cache parameter is a map that stores all previously
 * calculated chain lengths and is used to speed up the computation.
 * If the computation ever encounters a number it has seen before,
 * it simply returns the value from the cache.
 */

int countHailstoneSteps(int n, Map<int,int> & cache) {
    if (cache.containsKey(n)) return cache[n];
    if (n == 1) {
        return 0;
    } else {
        int count;
        if (n % 2 == 0) {
            count = 1 + countHailstoneSteps(n / 2, cache);
        } else {
            count = 1 + countHailstoneSteps(3 * n + 1, cache);
        }
        cache[n] = count;
        return count;
    }
}

```

You can, however, also code the solution iteratively using a stack (or a vector) to keep track of the values you need to insert into the cache:

```
int countHailstoneSteps(int n, Map<int,int> & cache) {
    Stack<int> path;
    while (n != 1 && !cache.containsKey(n)) {
        path.push(n);
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }
    int count = (n == 1) ? 0 : cache[n];
    while (!path.isEmpty()) {
        count++;
        cache[path.pop()] = count;
    }
    return count;
}
```

Problem 4: Recursive functions (10 points)

```
/*
 * Function: removeDoubledLetters
 * Usage: string shorter = removeDoubledLetters(str);
 * -----
 * Removes all but the first of a sequence of identical letters from str.
 */

string removeDoubledLetters(string str) {
    if (str.length() <= 1) {
        return str;
    } else if (str[0] == str[1]) {
        return removeDoubledLetters(str.substr(1));
    } else {
        return str[0] + removeDoubledLetters(str.substr(1));
    }
}
```

Problem 5: Recursive procedures (15 points)

```
/*
 * Function: tryAllOperators
 * Usage: tryAllOperators(exp, target);
 *       tryAllOperators(prefix, rest, target);
 * -----
 * Recursively replaces every ? in the expression by each of the
 * primary arithmetic operators (+, -, *, /). If the resulting
 * expression evaluates to the target integer, the function
 * prints out the expression string that generated it. The first
 * version of the function is a simple wrapper for the second,
 * which divides up the string one character at a time, keeping
 * track of the previously considered characters in prefix.
 */

void tryAllOperators(string exp, int target) {
    tryAllOperators("", exp, target);
}

void tryAllOperators(string prefix, string rest, int target) {
    if (rest == "") {
        if (evaluateExpression(prefix) == target) {
            cout << prefix << endl;
        }
    } else if (rest[0] == '?') {
        rest = rest.substr(1);
        tryAllOperators(prefix + "+", rest, target);
        tryAllOperators(prefix + "-", rest, target);
        tryAllOperators(prefix + "*", rest, target);
        tryAllOperators(prefix + "/", rest, target);
    } else {
        tryAllOperators(prefix + rest[0], rest.substr(1), target);
    }
}
```