CS106B Midterm Review

Ashley Taylor, Anton Apostolatos

Today's Session Overview

- Logistics
- Functions & Tracing
- ADTs
- Big-O

  Anton

- Recursion
  - Tips, tricks, and everything in between!

  Ashley

Logistics

Midterm Logistics

- Thursday November 3rd, 7-9pm
- Split into two locations based on last name:
  - A-R: Cemex Auditorium (GSB)
  - S-Z: Braun Auditorium (Mudd)
- Open books, open notes, open mind (but closed computer)
- Pencils *highly* recommended, pens accepted

What's on the midterm

- Topics:
  - Functions and Pass by Reference
  - ADTs (Maps, Sets, Stacks, Queues, Vectors and Grids)
  - Big-O Notation
  - Recursion
- Four/five questions total
  - One Tracing/Big-O question
  - Remainder will be code-writing questions

Recursion in real-life

What's **not** on the midterm

- Structs
- Pointers
- Defining classes
- Linked Lists
- Memoization
- Sorting

Source: XKCD

Functions

Tip #0: Understand what happens when you pass by value vs. reference

Tip #1: Make sample mystery code and test yourself and your friends

Source: International Obfuscated C Code Contest

Tracing Exercise
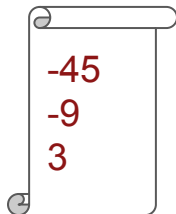
```cpp
int main() {
    int frodo = 7;
    int sam = 5;
    int merry = 4;
    int pippin = cervantes(sam, dumas(frodo, sam, merry));
    cout << sam << endl;
    cout << pippin << endl;
    cout << merry << endl;
    return 0;
}
```

```cpp
int cervantes(int &sancho, int quixote) {
    sancho *= quixote;
    return quixote--;
}



int dumas(int athos, int &aramis, int &porthos) {
    if (athos + aramis < porthos) {
        athos = aramis - porthos;
    } else {
        porthos--;
    }
    return aramis - (athos + 7);
}
```

Tracing Exercise

```cpp
int main() {
    int frodo = 7;
    int sam = 5;
    int merry = 4;
    int pippin = cervantes(sam, dumas(frodo, sam, merry));
    cout << sam << endl;
    cout << pippin << endl;
    cout << merry << endl;
    return 0;
}
```

Console

-45
-9
3

```cpp
int cervantes(int &sancho, int quixote) {
    sancho *= quixote;
    return quixote--;
}
```

```cpp
int dumas(int athos, int &aramis, int &porthos) {
    if (athos + aramis < porthos) {
        athos = aramis - porthos;
    } else {
        porthos--;
    }
    return aramis - (athos + 7);
}
```

ADTs

Vector
(1D list of elements)

43	12	0	-20	32

add(***value***)	O(1)	Adds a new value to the end of this vector.
clear()	O(1)	Removes all elements from this vector.
equals(***v***)	O(N)	Returns `true` if the two vectors contain the same elements in the same order.
get(***index***)	O(1)	Returns the element at the specified index in this vector.
insert(***index, value***)	O(N)	Inserts the element into this vector before the specified index.
isEmpty()	O(1)	Returns `true` if this vector contains no elements.
mapAll(***fn***)	O(N)	Calls the specified function on each element of the vector in ascending index order.
remove(***index***)	O(N)	Removes the element at the specified index from this vector.
set(***index, value***)	O(1)	Replaces the element at the specified index in this vector with a new value.
size()	O(1)	Returns the number of elements in this vector.
subList(***start, length***)	O(N)	Returns a new vector containing elements from a sub-range of this vector.
toString()	O(N)	Converts the vector to a printable string representation.

Grid
(2D list of elements)

4	90	20
12	0	33

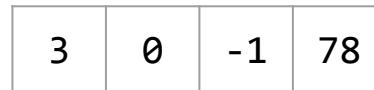|		
equals(***grid***)	O(N)	Returns `true` if the two grids contain the same elements.
fill(***value***)	O(N)	Sets every grid element to the given value.
get(***row, col***)	O(1)	Returns the element at the specified row/col position in this grid.
height()	O(1)	Returns the grid's height, that is, the number of rows in the grid.
inBounds(***row, col***)	O(1)	Returns `true` if the specified row and column position is inside the bounds of the grid.
isEmpty()	O(1)	Returns true if the grid has 0 rows and/or 0 columns.
mapAll(***fn***)	O(N)	Calls the specified function on each element of the grid.
numCols()	O(1)	Returns the number of columns in the grid.
numRows()	O(1)	Returns the number of rows in the grid.
resize(***nRows, nCols***)	O(N)	Reinitializes the grid to have the specified number of rows and columns.
set(***row, col, value***)	O(1)	Replaces the element at the specified row/col location in this grid with a new value.
size()	O(1)	Returns the total number of elements in the grid.
toString()	O(N)	Converts the grid to a printable single-line string representation.
toString2D()	O(N)	Converts the grid to a printable 2-D string representation.
width()	O(1)	Returns the grid's width, that is, the number of columns in the grid.

Stack
(LIFO linear structure)

push → → pop

43
21
-1
89

Queue
(FIFO linear structure)

enqueue → | 3 | 0 | -1 | 78 | → dequeue

clear()	O(1)	Removes all elements from this stack.
equals(*stack*)	O(N)	Returns **true** if the two stacks contain the same elements in the same order.
isEmpty()	O(1)	Returns **true** if this stack contains no elements.
peek()	O(1)	Returns the value of top element from this stack, without removing it.
pop()	O(1)	Removes the top element from this stack and returns it.
push(*value*)	O(1)	Pushes the specified value onto this stack.
size()	O(1)	Returns the number of values in this stack.
toString()	O(N)	Converts the stack to a printable string representation.

back()	O(1)	Returns the last value in the queue by reference.
clear()	O(1)	Removes all elements from the queue.
dequeue()	O(1)	Removes and returns the first item in the queue.
enqueue(*value*)	O(1)	Adds **value** to the end of the queue.
equals(*q*)	O(N)	Returns **true** if the two queues contain the same elements in the same order.
front()	O(1)	Returns the first value in the queue by reference.
isEmpty()	O(1)	Returns **true** if the queue contains no elements.
peek()	O(1)	Returns the first value in the queue, without removing it.
size()	O(1)	Returns the number of values in the queue.
toString()	O(N)	Converts the queue to a printable string representation.

Map
(Collection of key/value pairs)

Set
(Collection of unique elements)
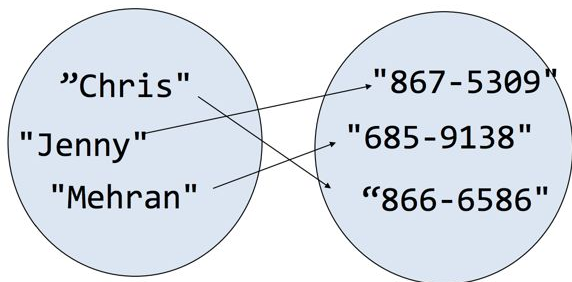
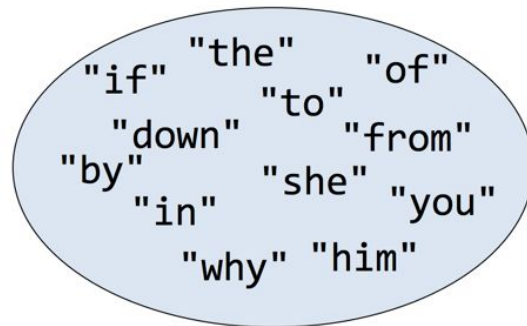|		
clear()	O(N)	Removes all entries from this map.
containsKey(**key**)	O(log N)	Returns **true** if there is an entry for **key** in this map.
equals(**map**)	O(N)	Returns **true** if the two maps contain the same elements.
get(**key**)	O(log N)	Returns the value associated with **key** in this map.
isEmpty()	O(1)	Returns **true** if this map contains no entries.
keys()	O(N)	Returns a **Vector** copy of all keys in this map.
mapAll(**fn**)	O(N)	Iterates through the map entries and calls **fn(key, value)** for each one.
put(**key, value**)	O(log N)	Associates **key** with **value** in this map.
remove(**key**)	O(log N)	Removes any entry for **key** from this map.
size()	O(1)	Returns the number of entries in this map.
toString()	O(N)	Converts the map to a printable string representation.
values()	O(N)	Returns a **Vector** copy of all values in this map.

add(**value**)	O(log N)	Adds an element to this set, if it was not already there.
clear()	O(N)	Removes all elements from this set.
contains(**value**)	O(log N)	Returns **true** if the specified value is in this set.
equals(**set**)	O(N)	Returns **true** if the two sets contain the same elements.
first()	O(log N)	Returns the first value in the set in the order established by a for-each loop.
isEmpty()	O(1)	Returns **true** if this set contains no elements.
isSubsetOf(**set2**)	O(N)	Implements the subset relation on sets.
mapAll(**fn**)	O(N)	Iterates through the elements of the set and calls **fn(value)** for each one.
remove(**value**)	O(log N)	Removes an element from this set.
size()	O(1)	Returns the number of elements in this set.
toString()	O(N)	Converts the set to a printable string representation.

ADT Exercise I

Write a Boolean non-recursive function `isBalanced(string input)` that checks for balanced parentheses or brackets.

```
isBalanced("((1)[])") → true

isBalanced("(sda()[((hello))](chris))") → true

isBalanced("[(]C)1(22)([tw]))") → false
```

ADT Exercise I - **Use a stack!**

```cpp
bool isBalanced(string input) {
    Stack<char> charOrder;
    for (char ch : input) {
        if (ch == '(' || ch == '[') {
            charOrder.push(ch);
        } else if (ch == ')') {
            if (charOrder.isEmpty()
                    || charOrder.pop() != '(') return false;
        } else if (ch == ']') {
            if (charOrder.isEmpty()
                    || charOrder.pop() != '[') return false;
        }
    }
    return charOrder.isEmpty();
}
```

Program logic:

- Skip non-parens characters
- Push to stack when opening parens encountered
- When closing parens encountered, check to see if stack is not empty and top item in stack is corresponding opening parens
- When we finish, need to make sure we have no parens we haven't opened, so stack must be empty
- **Challenge:** Do this recursively!

ADT Exercise II

Write a function

`commonContacts(Map<string, int> myRolo, Map<string, int> theirRolo)`

that returns a set of all the common contacts between two rolodexes

ADT Exercise II - **Using vectors**

```
Set<string> commonContacts(Map<string, int> myRolo, Map<string, int> theirRolo) {
  Set<string> commonContacts;
  for (string myContact : myRolo.keys()) {
      for (string theirContact : theirRolo.keys()) {
          if (myContact == theirContact) {
              commonContacts.add(myContact);
          }
      }
  }
  return commonContacts;
}
```

ADT Exercise II - **Using sets**

```
Set<string> commonContacts(Map<string, int> myRolo, Map<string, int> theirRolo) {
    // Make a set for each rolodex
    Set<string> myContacts;
    Set<string> theirContacts;
    for (string contact : myRolo.keys()) {
        myContacts.add(contact);
    }
    for (string contact : theirRolo.keys()) {
        theirContacts.add(contact);
    }
    // Finds the intersection between two sets
    return myContacts * theirContacts;
}
```

Big-O Notation

Big-O Exercise I

If `vec` has `N` elements and `database` has `M` elements:

```cpp
int overlap(Vector<int> &vec, Set<int> &database) {
    int total = 0;
    for (int itemOne : vec) {
        for (int itemTwo : database) {
            if (itemOne == itemTwo) {
                total++;
            } else {
                for (int itemThree : vec) {
                    cout << "Wut" << total;
                }
            }
        }
    }
    return total;
}
```

Big-O Exercise I - **O(N²M)**

If `vec` has `N` elements and `database` has `M` elements:

```cpp
int overlap(Vector<int> &vec, Set<int> &database) {
    int total = 0;                          // O(1)
    for (int itemOne : vec) {               // O(N * M * N)
        for (int itemTwo : database) {          // O(M * N)
            if (itemOne == itemTwo) {               // O(1)
                total++;                                // O(1)
            } else {
                for (int itemThree : vec) {         // O(N)
                    cout << "Wut" << total;             // O(1)
                }
            }
        }
    }
    return total;                           // O(1)
}
```

Big-O Exercise II

If `vec` has `N` elements and `database` has `M` elements:

```cpp
int overlap(Vector<int> &vec, Set<int> &database) {
    int total = 0;
    for (int item : vec) {
        total += database.contains(item);
    }
    return total;
}
```

Big-O Exercise II - **O(N log(M))**
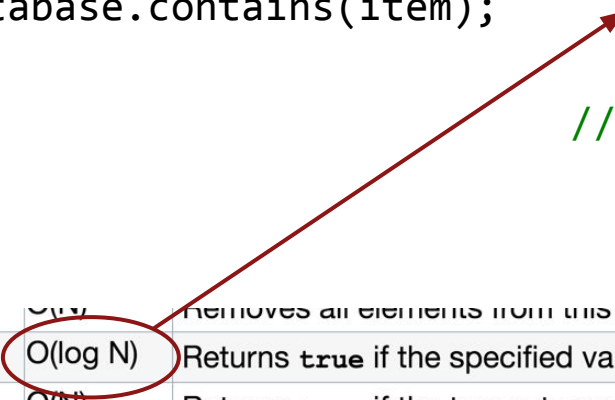
If `vec` has `N` elements and `database` has `M` elements:

```
int overlap(Vector<int> &vec, Set<int> &database) {
    int total = 0;                              // O(1)
    for (int item : vec) {                      // O(N * log(M))
        total += database.contains(item);        // O(log(M))
    }
    return total;                               // O(1)
}
```

~~clear()~~	~~O(N)~~	~~Removes all elements from this set.~~
contains(**value**)	O(log N)	Returns **true** if the specified value is in this set.
~~equals(**set**)~~	~~O(N)~~	~~Returns **true** if the two sets contain the same elements.~~

Big-O Exercise III

If `vec` has `N` elements and `database` has `M` elements:

```cpp
int overlap(Vector<int> &vec, Set<int> &database) {
    if (vec.isEmpty()) {
        return 0;
    }
    int item = vec[0];
    vec.remove(0);
    int itemInDB = database.contains(item);
    return itemInDB + overlap(vec, database);
}
```

Big-O Exercise III - $O(N * (N + \log(M)))$

If `vec` has `N` elements and `database` has `M` elements:

```cpp
int overlap(Vector<int> &vec, Set<int> &database) {
    if (vec.isEmpty()) {                          // O(1 * 1)
        return 0;                                      // O(1)
    }
    int item = vec[0];                            // O(1)
    vec.remove(0);                                // O(N)
    int itemInDB = database.contains(item);       // O(log(M))
    return itemInDB + overlap(vec, database);     // O(1)
}
```

Insight: each recursive step is O(N + log(M)) and there are N recursive calls total

Recursion

Parts to recursion

- Base case (simplest form of the problem):
  - HINT: try the empty string, the numbers 1 or 0, or an empty vector
- Recursive step

Steps to Solving Recursive Backtracking Problems

1. Identify the type of problem
2. Determine your base cases (helped by 1)
3. Determine the format of your recursive call(s) (helped by 1)
4. Identify your choices
   a. Usually, we apply the choice to the first character of the string or first element in a vector
5. Identify how to undo your choices
   a. i.e. add the element back to the vector or return the string to its original form
   b. "Reset" all your variables

Backtracking templates

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- Print/find all the solutions

Determine whether a solution exists

- Should return a boolean
- Base case: validate the solution so far; return true if valid, false otherwise
- Recursive step: Or of all potential choices

Determine Existence: Monopoly Pieces

- You're playing Monopoly with your friends, and each one is very picky and will only play with certain pieces.
- Given a set of Monopoly pieces (represented as strings), a vector of friends, and a Map from friends to acceptable pieces, can you assign each of your friends a unique piece?

Find a Solution: Monopoly Pieces

```cpp
bool monopoly(Vector<string> & friends,
              Set<string> & unusedPieces,
              Map<string, Set<string>> & constraints) {
  if (friends.isEmpty()) {
    return true;
  }
  string currFriend = friends[0];
  friends.remove(0);
  //continued on the next slide
```

Find a Solution: Monopoly Pieces

```java
for (string piece : constraints[currFriend]) {
  if (unusedPieces.contains(piece)) {
    unusedPieces.remove(piece);
    if (monopoly(friends, pieces, constraints)) {
      unusedPieces.add(piece);
      friends.add(0, currFriend);
      result[currFriend] = piece;
      return true;
    }
    unusedPieces.add(piece);
  }
}
return false;
}
```

Find a Solution

- Base case: validate the running solution; return the solution if it's valid, otherwise an empty solution
- Recursive step: Iterate through all the choices until one of them returns a valid solution, then return that solution

Find a Solution: N Queens

- Courtesy of CS106X
- A Queen in chess can move diagonally, horizontally, and vertically (any straight line)
- Place N queens in an N*N grid such that none can attack another, or return an empty grid if we can't
- A spot in the grid is true if a queen is placed there

Find a Solution: N Queens (helper function)

```cpp
bool isSpotSafe(Grid<bool> board, int row, int col) {
 for (int i = 0; i < board.numRows(); i++) {
   for (int dRow = -1; dRow <= 1; dRow++) {
     for (int dCol = -1; dCol <= 1; dCol++) {
       //check if queen is present on diagonal, row, or column
       if (board.inBounds(row + dRow * i, col + dCol * i) &&
           board[row + dRow * i][col + dCol * i]) {
         return false;
       }
     }
   }
 }
 return true;
}
```

Find a Solution: N Queens

```cpp
Grid<bool> placeQueens(int n) {
  Grid<bool> board(n, n, false);
  placeQueens(board, 0);
  return board;
}
```

Key insight: sometimes it's easier to pass the solution in by reference and populate it, and have the return type be a boolean indicating whether a solution was actually found. While the boolean might not matter in this outer function, it can make the recursion significantly cleaner

Find a Solution: N Queens

```cpp
bool placeQueens(Grid<bool> & board, int col) {
    if (col == board.numCols()) {
        return true;
    }
    for (int row = 0; row < board.numRows(); row++) {
        if (isSpotSafe(board, row, col)) {
            board[row][col] = true;
            if (placeQueens(board, col + 1)) {
                return true;
            }
            board[row][col] = false;
        }
    }
    return false;
}
```

Find the Best Solution

- Base Case: is it a valid solution? If so, return it; otherwise, return a default/empty solution
- Recursive step: make all the recursive calls, then output the "best" (longest, least cost, etc.) of all of them

Find the Best Solution: Longest Increasing Subsequence

Given a string, a **subsequence** is another string where all the characters of the subsequence appear in the string in the same relative order, but the not every character from the string needs to appear.

"cef" is a subsequence of "abcdef", but "db" is not because the characters are in the wrong order and "gh" is not because neither "g" nor "h" are in "abcdef"

Find the Best Solution: Longest Increasing Subsequence

We want to find the longest subsequence from the given string such that every letter is strictly "greater" than the one before it

A < B < C…

We can assume that the input string is lowercase

Find the Best Solution: Longest Increasing Subsequence

Each character in the original string is either in the subsequence or not in the subsequence (that's our choice)

We should process the string in order because the letters in the subsequence should be in the same order as they were in the original string

Find the Best Solution: Longest Increasing Subsequence

```
string longestIncreasingSubsequence(string input) {
    return longestIncSubseq(input, "");
}
```

Find the Best Solution: Longest Increasing Subsequence

```
string longestIncSubseq(string input, string subseq) {
 //base cases:
 int length = subseq.size();
 if (length > 1 &&
     subseq[length - 1] <= subseq[length - 2]) {
   return ""; //not increasing subsequence
 }
 if (input == "") {
   return subseq; //no more characters to process
 }
//continued on next slide
```

Find the Best Solution: Longest Increasing Subsequence

```
string longestIncSubseq(string input, string subseq) {
//continued from previous slide
 string withChar = longestIncSubseq(input.substr(1),
                                    subseq + input[0]);
 string withoutChar = longestIncSubseq(input.substr(1),
                                       subseq);
 //choose the "best" of the recursive calls
 if (withChar.size() > withoutChar.size()) {
   return withChar;
 }
 return withoutChar;
}
```

Count the Number of Solutions

- Base Case: validate the running solution. Return 0 if invalid, 1 if valid
- Recursive step: return the sum of all the recursive calls

Count the Number of Solutions: Valid Maze

- Sometimes it's important to make sure that there is exactly one solution, so we'll want to count all the possible solutions. A good example is with a maze.
- Given a starting point and an ending point and a maze represented as a Grid, count the number of unique paths from the start to the end.
- The maze will have true at a certain row/col we can pass through that spot, otherwise false (if there is a wall)

Count the Number of Solutions: Valid Maze

```cpp
int mazeSolutions(Grid<bool> & maze, int startRow,
                  int startCol, int endRow, int endCol) {
 if (!maze[startRow][startCol]) {
  // can't travel through walls
   return 0;
 }
 if (startRow == endRow && startCol == endCol) {
   return 1; //reached our goal
 }
 //recursive step is on the next slide
}
```

Count the Number of Solutions: Valid Maze

```cpp
int mazeSolutions(Grid<bool> & maze, int startRow,
                  int startCol, int endRow, int endCol) {
 maze[startRow][startCol] = false; // can't visit same spot twice
 int numSolutions = mazeSolutions(maze, startRow + 1,
                   startCol, endRow, endCol);
 numSolutions += mazeSolutions(maze, startRow - 1,
                startCol, endRow, endCol);
 numSolutions += mazeSolutions(maze, startRow,
                startCol + 1, endRow, endCol);
 numSolutions += mazeSolutions(maze, startRow,
                startCol - 1, endRow, endCol);
 maze[startRow][startCol] = true; // "undo" our choice
 return numSolutions;
}
```

Find all Solutions

- Base Case: Validate the solution; if valid, print it (or add it to the set of found solutions); if not valid, don't print/return the empty set
- Recursive Step: Make all recursive calls. If you are returning a set, add each recursive result to the set

Find All Solutions: Clumsy Thumbsy

- You want to write a program that will autocorrect words. Given a string that represents a single (potentially misspelled) word, a lexicon of English words, a map that maps from a character to a string of the characters near it, and an admissible number of errors, find the Set of all potential intended words.
- Citation: Jerry Cain

Find All Solutions: Clumsy Thumbsy

```
Set<string> autocorrect(string word,
                        Map<char, string> & nearLetters,
                        Lexicon & dictionary, int maxTypos) {
  return autoCorrectHelper(word, nearLetters, dictionary,
                           maxTypos, "");
}
```

Find All Solutions: Clumsy Thumbsy

```
Set<string> autocorrect(string remaining, Map<char, string> &
nearLetters, Lexicon & dictionary, int allowableTypos, string
builtUp) {
  Set<string> result;
  if (allowableTypos < 0 || !dictionary.containsPrefix(builtUp)) {
    // too many typos, or no potential to build word
    return result; //empty set
  } else if (remaining == "") {
    if (dictionary.contains(builtUp)) {
      // if word, add it to set
      result.add(builtUp);
    }
    return result;
  } //continued on next slide
```

Find All Solutions: Clumsy Thumbsy

```cpp
Set<string> autocorrect(string remaining, Map<char, string> &
nearLetters, Lexicon & dictionary, int allowableTypos, string
builtUp) {
 Set<string> result; //same result from previous slide
 char curr = remaining[0];
 string rest = remaining.substr(1);
 for (int i = 0; i < nearLetters[curr].length; i++) {
    result += autocorrect(rest, nearLetters,
         allowableTypos - 1, builtUp + nearLetters[curr]);
 }
 //can also choose not to change character
 result += autocorrect(rest, nearLetters, dictionary,
                       allowableTypos, builtUp + curr);
}
```

More practice

For more practice, check out section handouts 3 and 4 as well as the lectures and section handouts from cs106x, located at cs106x.stanford.edu