Today’s Session Overview

- Logistics
- Functions & Tracing
- ADTs
- Big-O
- Sorting
- Recursion and Memoization
Logistics
Midterm Logistics

- Thursday May 4th, 7-9pm
- Split into two locations based on last name:
  - A-N: Hewlett 200
  - O-Z: 420-040
- Only single double-sided sheet of notes allowed
- We will provide a reference sheet
- Can take the exam using computer or by hand
What’s on the midterm

- **Topics:**
  - Functions and Pass by Reference
  - ADTs (Maps, Sets, Stacks, Queues, Vectors and Grids)
  - Big-O Notation
  - Sorting
  - Recursion and Memoization

- **Four/five questions total**
  - One Tracing/Big-O question
  - Remainder will be code-writing questions
What’s **not** on the midterm

- Pointers
- Defining classes

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

```c
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;
}

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

```c
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;
}
```

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

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

```
 Console

-45
-9
3
```
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 and column 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(numRows, numCols) | 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 and column location in this grid with a new value. |
| size() | O(1) Returns the total number of elements in the grid. |
| toSingleString() | O(N) Converts the grid to a printable single-line string representation. |
| toPrintingGrid() | 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)

<table>
<thead>
<tr>
<th>push</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td></td>
</tr>
</tbody>
</table>

- **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.

### Queue
(FIFO linear structure)

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

- **back()**: O(1) Returns the last value in the queue by reference.
- **clear()**: O(1) Removes all elements from the queue.
- **deque()**: O(1) Removes and returns the first item in the queue.
- **enqueue(value)**: O(1) Adds `value` to the end of the queue.
- **equals(queue)**: 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)

- "Chris"
- "Jenny"
- "Mehran"
- "867-5309"
- "685-9138"
- "866-6586"

Set
(Collection of unique elements)

- "if"
- "the"
- "of"
- "down"
- "to"
- "from"
- "by"
- "in"
- "she"
- "you"
- "why"
- "him"

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>O(N)</td>
<td>Removes all entries from this map.</td>
</tr>
<tr>
<td>containsKey(key)</td>
<td>O(log N)</td>
<td>Returns true if there is an entry for key in this map.</td>
</tr>
<tr>
<td>equals(map)</td>
<td>O(N)</td>
<td>Returns true if the two maps contain the same elements.</td>
</tr>
<tr>
<td>get(key)</td>
<td>O(log N)</td>
<td>Returns the value associated with key in this map.</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
<td>Returns true if this map contains no entries.</td>
</tr>
<tr>
<td>keys()</td>
<td>O(N)</td>
<td>Returns a Vector copy of all keys in this map.</td>
</tr>
<tr>
<td>mapAll(fn)</td>
<td>O(N)</td>
<td>Iterates through the map entries and calls fn(key, value) for each one.</td>
</tr>
<tr>
<td>put(key, value)</td>
<td>O(log N)</td>
<td>Associates key with value in this map.</td>
</tr>
<tr>
<td>remove(key)</td>
<td>O(log N)</td>
<td>Removes any entry for key from this map.</td>
</tr>
<tr>
<td>size()</td>
<td>O(1)</td>
<td>Returns the number of entries in this map.</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
<td>Converts the map to a printable string representation.</td>
</tr>
<tr>
<td>values()</td>
<td>O(N)</td>
<td>Returns a Vector copy of all values in this map.</td>
</tr>
</tbody>
</table>

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

Write the following two methods for the Queue<int> class, allowing someone to use it as if it were a Stack<int>

```cpp
void push(Queue<int> &q, int entry) // Places element at front
int pop(Queue<int> &q) // Removes most recent element placed
```
void push(Queue<int> &q, int entry) {
    q.enqueue(entry);
    for (int i = 0; i < q.size() - 1; i++) {
        q.enqueue(q.dequeue());
    }
}

int pop(Queue<int> &q) {
    return q.dequeue();
}
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**

```java
Set<String> commonContacts(Map<String, int> myRolo, Map<String, int> theirRolo) {
    Set<String> commonContacts = new HashSet<>();
    for (String myContact : myRolo.keySet()) {
        for (String theirContact : theirRolo.keySet()) {
            if (myContact == theirContact) {
                commonContacts.add(myContact);
            }
        }
    }
    return commonContacts;
}
```

**Big-O? O(n^2 \log(n))**
ADT Exercise II - **Using sets**

```java
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?** O(n log(n))
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 - $O(N^2M)$

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(N)
                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 - \( \textbf{O(N \log(M))} \)

If \texttt{vec} has \( N \) elements and \texttt{database} has \( M \) elements:

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

If \texttt{vec} has \( N \) elements and \texttt{database} has \( M \) elements:
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);
    vec.remove(0);
    int itemInDB = database.contains(item);
    return itemInDB + overlap(vec, database);
}
```
Big-O Exercise III - \( O( N \times (N + \log(M)) ) \)

If \( \text{vec} \) has \( N \) elements and \( \text{database} \) has \( M \) elements:

```c++
int overlap(Vector<int> &vec, Set<int> &database) {
    if (vec.isEmpty()) {
        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.
Sorting
Sorting

- We won’t ask you to come up with your own sorting algorithm
- But… we expect you to know and understand the ones we went over in class
  - Which one is best for different situations?
  - What is the time complexity?
  - Could you trace through a few steps of the algorithm?
Recursion

Based on slides by Ashley Taylor
if (problem is sufficiently simple) {
    Directly solve the problem.
    Return the solution.
} else {
    Split the problem up into one or more smaller problems with the same structure as the original.
    Solve each of those smaller problems.
    Combine the results to get the overall solution.
    Return the overall solution.
}
Base Case

Simplest form of the problem -- try empty string, 1 or 0, or an empty vector

Recursive Step

Where the magic happens
Forms of recursion

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- Print/find all the solutions
Steps to Solving Recursive Backtracking Problems

1. Identify the type of problem
2. Determine your base case(s)
3. Determine the format of your recursive call(s)
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. “Reset” all your variables (i.e. add the element back to the vector or return the string to its original form)
Forms of recursion

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

- 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:** return true if one of potential choices is true; otherwise false
Forms of recursion

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

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

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

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- **Print/find all the solutions**
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
Recursion - 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.
  - Is “cef” is a subsequence of “abcdef”? **Yes!**
  - Is “db” is a subsequence of “abcdef”? **No: not in order**
  - Is “gh” is a subsequence of “abcdef”? **No: not in string**
Recursion - 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

\[ a < b < c \ldots \]

We can assume that the input string is lowercase
Recursion Exercise

Write a string recursive function `longestIncSubseq(string input)` that returns the longest increasing subsequence in a string

```
longestIncSubseq("abcdef") → "abcdef"
longestIncSubseq("zyxwvu") → "u" or "v" or ...
longestIncSubseq("chris gregg") → "chir"
```
What type of recursion question is this?
Forms of recursion

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

- 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
- We should work with only the first character of the string in each step!
string longestIncreasingSubsequence(string input) {
    return longestIncSubseq(input, "");
}

string longestIncSubseq(string input, string subseq) {
    // BASE CASE
    if (input == "") return subseq; // No more characters to process

    // RECURSIVE CASES
    // CASE I - skipping the first char in input
    string withoutChar = longestIncSubseq(input.substr(1), subseq);

    // CASE II - picking the first char in input -- only if valid
    string withChar = "";
    if (subseq.length() == 0 || input[0] > subseq[subseq.length() - 1]) {
        withChar = longestIncSubseq(input.substr(1), subseq + input[0]);
    }

    // RETURN: choose the “best” of the recursive calls
    return (withChar.size() > withoutChar.size()) ? withChar : withoutChar;
}
Memoization
Consider a human pyramid:
Consider a human pyramid, where every person weighs 200lbs.

What’s the weight on a certain person’s knees?
Write the recursive function

```java
double weightOnBackOf(int row, int col)
```

(row, col): row and col of person we’re interested in
Memoized Human Pyramids
Consider wanting the weight on the back of \((3, 2)\)
Consider wanting the weight on the back of $(3, 2)$
Consider wanting the weight on the back of (3,2)
Consider wanting the weight on the back of $(3, 2)$
Consider wanting the weight on the back of $(3, 2)$
Consider wanting the weight on the back of \((3,2)\)
Consider wanting the weight on the back of \((3,2)\)

We already calculated this before!
Consider wanting the weight on the back of \((3, 2)\)
Consider wanting the weight on the back of \((3, 2)\)

We already calculated this before!

*Let’s store it!*
Memoization speeds things up!

Before:
```c++
// RecursiveFunction
Ret recursiveFunction(Arg a) {
    if (base-case-holds) {
        return base-case-value;
    } else {
        do-some-work;
        return recursive-step-value;
    }
}
```

After:
```c++
// RecursiveFunction
Ret recursiveFunction(Arg a, Table& table) {
    if (base-case-holds) {
        return base-case-value;
    } else if (table contains a) {
        return table[a];
    } else {
        do-some-work;
        table[a] = recursive-step-value;
        return recursive-step-value;
    }
}
Questions?