Compsci 201 Midterm 2 Review

James Wei
Professor Peck
4/7/2014
List of Topics

- Linked lists
- Recursion
- Recurrence Relations
- Trees
- Heaps
- Priority Queues
- Recursive Backtracking
- Sorting
Linked Lists

• How do we build a LinkedList?
  public class Node {
    public int value;
    public Node next;
    public Node ( int v, Node n ) {
      value = v;
      next = n;
    }
  }

• Traverse by storing pointer to HEAD and referencing its NEXT pointer
Linked Lists

• Comparison with ArrayList:
  – ArrayList works by storing an array and resizing it as necessary, hence why it is so fast—we can access different indices and make changes just as we would with an array
  – Removal in ArrayList is a bit slower since we need to recopy the entire array to eliminate an element
Linked Lists

- What are the runtime differences in these two implementations?

<table>
<thead>
<tr>
<th>Big O</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Removal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked Lists

- What are the runtime differences in these two implementations?

<table>
<thead>
<tr>
<th>Big O</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insertion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Removal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked Lists

- What are the runtime differences in these two implementations?

<table>
<thead>
<tr>
<th>Big O</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insertion</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Removal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked Lists

<table>
<thead>
<tr>
<th>Big O</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insertion</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Removal</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

• What are the runtime differences in these two implementations?
• How can ArrayList insert be O(1)?
  – Insert is O(1) amortized
• When would you want to use Linked Lists?
Linked Lists

• Linked List practice!
• Write the following method:
  
  ```java
  public void removeDuplicates(Node head) {
    // your code here
  }
  ```

• `removeDuplicates` takes in the head of a Linked List and removes all duplicate values from that list
Recursion

• What is recursion?
  – https://piazza.com/class/hp0ej0ja6vo2ec?cid=589
  – Strategy for solving harder problems by breaking them down into smaller subproblems
  – Three easy steps:
    1) Recursion step: how do we break a problem into easier subproblems?
    2) Resolution step: how do we combine the answers to our subproblems into an answer for our harder problem?
    3) Base step: when do we stop recursing?
Recursion

• Let’s start with an easy example: finding the nth fibonacci number.

• We will write the following method:
  
  ```java
  public int fibonacci(int n) {
    // returns the nth fibonacci number
  }
  ```
Recursion

public int fibonacci(int n) {

}

• What is the “hard” problem here?
  – Finding the nth fibonacci number.

• How can we break this down to easier subproblems?
  – Finding the n-1th and n-2th fibonacci number.
Recursion

public int fibonacci(int n) {
    int a = fibonacci(n-1);
    int b = fibonacci(n-2);
}

• Let’s start by adding our recursion step. How do we break the hard problem down in code?
public int fibonacci(int n) {
    int a = fibonacci(n-1);
    int b = fibonacci(n-2);
    return a + b;
}

• Next comes the resolution step. How do we combine these subproblem answers into our hard problem answer?
Recursion

public int fibonacci(int n) {
    if (n == 1 || n == 2) { return 1; }
    int a = fibonacci(n-1);
    int b = fibonacci(n-2);
    return a + b;
}

• Finally, the base step. When do we stop recursing?
Recursion

• Recursion practice!
• Write the following method:
  public String countDown(int n) {
    // your code here
  }
• countDown takes in an int and prints out a String counting down from that int to 0 and then back up to that int.
  – For example, countDown(3) should return “3...2...1...0...1...2...3”
Recursion

- Recursion practice!
- Need a harder problem?
- Write the following method:
  ```java
  public ArrayList<Integer> factorize(int n) {
    // your code here
  }
  ```
- Factorize takes in an int and returns an ArrayList containing all of that number’s prime factors. Assume that the number’s prime factors all belong in the set [2, 3, 5, 7, 11].
Recurrence Relations

• We want to be able to determine the runtime performance of our recursive algorithms.
• We solve recursive problems by breaking them into subproblems, so let’s express our algorithm’s runtime in terms of the runtimes of its recursive calls!
Recurrence Relations

• What do recurrence relations look like?
• Here’s a random example:

\[ T(n) = 2^T(n/2) + O(1) \]

– \(T(n)\): time to solve algorithm with input size \(n\)
– \(2^T(n/2)\): time to solve recursive calls made by algorithm (with input size \(n/2\) each)
– \(O(1)\): time to perform remaining logic
Recurrence Relations

• More generally:
  
  Runtime = Recursive calls + Additional logic

• Runtime of the recursive algorithm in question is expressed as $T(n)$, where $n$ represents the size of the input
Recurrence Relations

• Consider our fibonacci code:
  
  ```java
  public int fibonacci(int n) {
      if (n == 1 || n == 2) { return 1; }
      int a = fibonacci(n-1);
      int b = fibonacci(n-2);
      int result = a + b;
  }
  ```

• Let’s write a recurrence relation for this:
  – Represent the call to fibonacci(n) as T(n)
  – How to represent time to run recursive calls?
  – How to represent time to perform remaining logic?

• Recurrence relation is T(n) = T(n-1) + T(n-2) + O(1) = O(2^n)
Recurrence Relations

• Here’s the good news—we’ve solved some common recurrence relations for you!

<table>
<thead>
<tr>
<th>Recurrence Relation</th>
<th>Example</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>Binary Search</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>Linear Search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>Tree traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>QuickSort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>BubbleSort</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

• For any given algorithm, write the recurrence relation and use the chart to find it’s Big Oh
Recurrence Relations

• What is the recurrence relation for this code?
  ```java
  public void sortIntoList(List<Integer> list, TreeNode root) {
    if (root == null) { return; }
    int val = root.value;
    int i;
    for (i=0; i<list.size(); i++) {
      if (list.get(i) > val) { break; }
    }
    list.add(val, i);
    sortIntoList(list, root.left);
    sortIntoList(list, root.right);
  }
  ```
• Balanced: \( T(n) = 2*T(n/2) + O(n) = O(n \log n) \)
• Unbalanced: \( T(n) = T(n-1) + O(n) = O(n^2) \)
• Bonus: how can we rewrite this method more efficiently if we assume our tree is a BST?
Recurrence Relations

• Few final notes on this:
  – You will not need to solve recurrence relations except by matching them with one in the chart
  – Don’t be fooled! If an algorithm makes a call to another, separate recursive algorithm, that recursive algorithm is NOT represented with $T!$
    E.g., in a BST class our diameter() calls height()
  – Anyone who writes $T(n)$ on the right side of the recurrence relation has made better decisions in life
Trees

- Linked Lists on steroids
- Now instead of one node having one next pointer, it has multiple children; we will focus on binary trees (two children)
- What do we change from the LinkedList node?

```java
public class Node {
    public int value;
    public Node left;
    public Node right;
    public Node ( int v, Node l, Node r ) {
        value = v;
        left = l;
        right = r;
    }
}
```
Trees

• How do we navigate across a tree?
• Three types of traversals:
  1) Preorder (my favorite) : Root Left Right
  2) Inorder : Left Root Right
  3) Postorder : Left Right Root
Trees

• How do we express a tree in terms of its traversal?
• Think of every node as the root of a subtree, and things become much easier.
• Let's find the preorder traversal of this tree on the right.
Trees

• Preorder traversal is [Root] [Left] [Right]
Trees

• Preorder traversal is [Root] [Left] [Right]
• Root is easy; it’s 10

Preorder traversal: 10 [ Left ] [ Right ]
Trees

- Preorder traversal is [Root] [Left] [Right]
- Root is easy; it’s 10
- Replace [Left] with the preorder traversal of the left subtree.

Preorder traversal: 10 [[Root][Left][Right]] [ Right ]
Trees

- Preorder traversal is [Root] [Left] [Right]
- Root is easy; it’s 10
- Replace [Left] with the preorder traversal of the left subtree.
- Now we can view this subtree as a subset of the main traversal and find its preorder

Preorder traversal: 10 5 7 12 [ Right ]
Trees

- Preorder traversal is [Root] [Left] [Right]
- Root is easy; it’s 10
- Replace [Left] with the preorder traversal of the left subtree.
- Now we can view this subtree as a subset of the main traversal and find its preorder

Preorder traversal: 10 5 7 12 13 6 1
Trees

• What makes a binary search tree special?
  – The tree is sorted: All left children are less than root and all right children are greater than root, repeated values are not allowed
  – Allows us to search the tree easily
  – Inserting also not any harder than before, we just move left or right by comparing our new value to the old value at each node
What are the runtime complexities?
- Depends on whether or not the tree is balanced.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Balanced</th>
<th>Unbalanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Remove</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Ultimately we need to know how long it takes to traverse—give me some recurrence relations for this!
- Balanced: $T(n) = T(n/2) + O(1) = O(\log n)$
- Unbalanced: $T(n) = T(n-1) + O(1) = O(n)$
Trees

- Tree practice!
- Write the following method:
  ```java
  public boolean validate(Node root) {
     // your code here
  }
  ```
- Validate takes in the root of a binary search tree and returns true if that tree is a valid BST, and false otherwise
• Tree practice!
• Need a harder question?
• Write the following method:
  ```java
  public boolean remove(int v) {
      // your code here
  }
  ```
• Remove takes in an int value and removes the node with that value from the tree, if it exists.
• This is tricky! Just finding the node isn’t enough, we must also resort the tree to still be a valid BST afterwards!
Heaps

• Tree with two extra properties:
  – Shape property: the tree is always balanced, with new levels being added from left to right
  – Heap property: every node is sorted after its parent and before its children

• Note that we will focus on binary heaps
Heaps

- Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
- In each step we will need to keep track of the first open index of the heap
  - Index progresses level by level from left -> right
  - For node i, children at 2i+1 and 2i+2, parent at floor((i-1)/2)
- First we add our first element as the root
Heaps

• Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
• Next we add the second element (7) to the first open index (1)
Heaps

• Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
  • Next we add the third element (2) to the first open index (2)
    – But wait! Heap property is now violated!
    – We must make a swap!
Heaps

• Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:

• Next we add the third element (2) to the first open index (2)
  – But wait! Heap property is now violated!
  – We must make a swap!
Heaps

• Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
• Next we add the fourth element (6) to the first open index (3)
  – Heap property is again violated!
  – Must make a swap
• Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
• Next we add the fourth element (6) to the first open index (3)
  – Heap property is again violated!
  – Must make a swap
Heaps

- Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
- We continue adding the rest of the elements, making swaps anytime the heap property is violated.
- Since we are adding in level-order, the shape property never fails.
Heaps

• Let’s construct a min-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
• What is the Big Oh of the add operation?
  – O(log n): insert at first open index then perform up to log n swaps…
Heaps

• If instead we had constructed a max-heap with the elements [4, 7, 2, 6, 3, 1, 5]:
  – Every node is now greater than its children and less than its parent
Heaps

- In order to remove from a heap, we always remove the root.
  - In this case we would pop off the 1 node
  - Then we would replace the root with the highest-index node, which is 5 in this case
  - Finally we would perform a series of swaps to re-heapify the heap
Priority Queues

• A priority queue is a queue which maintains priorities for each element and pops them out by order of priority
• For instance, if we create a priority queue that orders integers in ascending order and then push in the elements [3, 5, 1, 4, 2], then popping 5 times gives the order [1, 2, 3, 4, 5]
• How can we implement this?
Priority Queues

• Implement it as a heap!!!
  – We create a heap and call it a priority queue, then when we push elements in we add them to the heap as before
  – Popping off of the queue will actually pop the root node of the heap
Recursive Backtracking

- Time to add in backtracking (shit just got real).
- Let’s face it, recursion can be super inefficient.
- Backtracking refers to the idea that while we are recursing, we might not always want to recurse to the end, so that we can save time.
- Requires us to add another step to our three easy steps of recursion.
Recursive Backtracking

• Let’s amend our steps to recursion as such:
  
  1) Recursion step : how do we break a problem into easier subproblems?
  
  2) Resolution step : how do we combine the answers to our subproblems into an answer for our harder problem?
  
  3) Backtrack step : how do we go back to our old state?
  
  4) Base step : when do we stop recursing?
Recursive Backtracking

• Let’s solve a backtracking problem!
• Simplified sudoku: one nxn board
• We will write the following method:
  public boolean solve(int[][] board) {
    // your code here
  }
• Solve takes in a 2D int array representing an incomplete sudoku board, and returns true if it can be solved
  – Empty spaces are represented by -1
  – This method is DESTRUCTIVE!!
Recursive Backtracking

```java
public boolean solve(int[][][] board) {
}
```

• What is the “hard” problem here?
  – Solving the sudoku board.

• How can we break this down to easier subproblems?
  – Solving the sudoku board with one more space filled in.

• What makes this problem harder than regular recursion?
  – We must be able to revert our board to a previous state if we find ourselves filling in the wrong answers!
Recursive Backtracking

```java
public int[] findNextOpenSpace(int[][][] board) {
    // returns int array representing coordinates
    // of next open space, or [-1, -1] if no open spaces
}

public boolean checkValid(int[][][] board) {
    // returns true if board is valid, false otherwise
}
```

- To help you solve this problem, here’s a present—two black-box helper methods!
Recursive Backtracking

public boolean solve(int[][] board) {
    int[] next = findNextOpenSpace(board);
    int i = next[0]; int j = next[1];
    for (int v=1; v<=board.length; v++) {
        board[i][j] = v;
        boolean newBoardValid = solve(board);
    }
}

• Let’s start by adding our recursion step. How do we break the hard problem down in code?
Recursive Backtracking

public boolean solve(int[][] board) {
    int[] next = findNextOpenSpace(board);
    int i = next[0]; int j = next[1];
    for (int v=1; v<=board.length; v++) {
        board[i][j] = v;
        boolean newBoardValid = solve(board);
        if (newBoardValid) {
            return true;
        }
    }
    return false;
}

• Now the resolution step. How do we take our subproblem answer and turn it into a hard problem answer?
• Almost done! Next comes the backtrack step. How do we revert to our old state if we find our answer is not correct so far?
Recursive Backtracking

```java
public boolean solve(int[][] board) {
    if (!checkValid(board)) {
        return false;
    }
    int[] next = findNextOpenSpace(board);
    int i = next[0]; int j = next[1];
    if (i == -1 && j == -1) {
        return true;
    }
    for (int v=1; v<=board.length; v++) {
        board[i][j] = v;
        boolean newBoardValid = solve(board);
        if (newBoardValid) {
            return true;
        }
        board[i][j] = -1;
    }
    return false;
}
```

- Last step! The base step. How do we know when to stop recursing?
Recursive Backtracking

• Recursive backtracking practice!
• So SimpleSudoku was too easy, huh?
• Extend the logic such that we only return true if there exists a *unique* solution:
• The method may remain destructive, i.e. it will change the underlying 2D array when it is run.
Recursive Backtracking

• Recursive backtracking practice!
• Need an even harder problem?
• Let’s try a 2-player backtracking game.
• Write the following method:
  ```java
  public int whoWinsPilesList(ArrayList<Integer> piles) {
    // your code here
  }
  ```
• whoWinsPilesList takes in an ArrayList of Integers that represents the piles list and returns who wins, assuming both play perfectly
  – During their turn, each player removes either 1 or 2 tokens from 1 pile in the piles list
  – The first player who cannot move loses
Sorting

• Five sorts to cover:
  – Bubble sort
  – Insertion sort
  – Quick sort
  – Merge sort
  – Heap sort
• Bubble Sort:
  – Idea is that we pass through our list, comparing every pair of elements, swapping them if out of order
  – Requires up to n pass throughs, where n is the number of elements in the list

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Worst case</th>
<th>Expected case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Complexity</td>
<td>O(n)</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
</tbody>
</table>
Insertion sort:

- Create an initially empty sorted list
- Iterate over unsorted list, and for every element place it in sorted order in the sorted list
- We are taking each element and inserting it into the sorted list

<table>
<thead>
<tr>
<th>Best case</th>
<th>Worst case</th>
<th>Expected case</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(n)</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
</tbody>
</table>
Sorting

• Quick sort:
  – Recursive sorting algorithm
  – In each recursive call we randomly pick an element to be the pivot and arrange the rest of the list into two unsorted arrays, one consisting of elements below the pivot and one above
  – Recursively sort two unsorted lists and then combine them together

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Worst case</th>
<th>Expected case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
• Merge sort:
  - Recursive sorting algorithm
  - In each recursive call we first split our list into two unsorted halves and recursively sort them
  - Then we merge them back together by starting at the beginning of each sublist and adding back the element which comes first in sorted order

<table>
<thead>
<tr>
<th>Best case</th>
<th>Worst case</th>
<th>Expected case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
The Heap Sort algorithm starts by building a heap ordered according to the order by which we want to sort our list. Then, we continuously pop off the root of the heap and add those elements into the list. The heap is reconstructed as we go to maintain order.

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Worst case</th>
<th>Expected case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
One last thing—storytime!
Good luck!