Final Review 2

CS16: Introduction to Data Structures & Algorithms
Spring 2018
Outline

- Recursion
- Dynamic Programming
- HTA Refresher
  - Functional programming
  - Hardness
  - PageRank
  - Edit Distance
“Something defined in terms of itself”
Recursion

- What is a recursive problem?
  - a problem defined in terms of itself
- What is a recursive function?
  - a function defined in terms of itself
  - example: Fibonacci
- **Note**: at each level problem/function/pic gets easier/smaller
- How can we solve recursive problems?
Recursive Algorithms

- Algorithms that call themselves
  - Call themselves on smaller inputs (sub-problems)
  - Combine the results to find solution to larger input

- Recursive algorithms
  - Can be very easy to describe & implement :-)
  - Can be hard to analyze :-(

Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

\[ F(n) = F(n - 1) + F(n - 2) \]

**base cases:**

\[ F(0) = 1 \& F(1) = 1 \]
Fibonacci

- Defined by the recursive relation
  - \( F_0 = 0 \), \( F_1 = 1 \)
  - \( F_n = F_{n-1} + F_{n-2} \)
- We can implement this with a recursive function

```python
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Q: Is any work happening?
Recursion & Clones

- At each intersection
  - clone yourself twice and send one Left and one Right
  - wait for clones to report a path to exit (if it exists) and its length
  - pick direction that gets you to exit the fastest
Outline

- Recursion
- Dynamic Programming
- Functional programming
- Hardness
- PageRank
- Edit Distance
What is Dynamic Programming?

- Algorithm design paradigm/framework
  - Design efficient algorithms for optimization problems
- Optimization problems
  - “find the best solution to problem X”
  - “what is the shortest path between u and v in G”
  - “what is the minimum spanning tree in G”
- Can also be used for non-optimization problems
When is Dynamic Programming Applicable?

- Condition #1: sub-problems
  - The problem can be solved recursively
  - Can be solved by solving sub-problems
- Condition #2: overlapping sub-problems
  - Same sub-problems need to be solved many times
- Core idea
  - Solve each sub-problem once and store the solution
  - Use stored solution when you need to solve sub-problem again
Steps to Solving a Problem w/ DP

- What are the **sub-problems**?
- What is the “**magic**” step?
  - Given solution to a sub-problem…
  - …how do I get solution to the problem?
- Which **topological order** on sub-problems can I use?
  - so that solutions to sub-problems available when needed
- Design iterative **algorithm**
  - that solves sub-problems in order and stores their solution
Topological Order?

- Valid topological orderings
  - 15, 16, 22, 141
  - 22, 15, 16, 141
  - 15, 22, 16, 141
Topological Order on Sub-Problems!
The Fibonacci Problem

- Given $n$ compute
  - $Fib(n) = Fib(n-1) + Fib(n-2)$
  - with base cases $Fib(0) = 0$ and $Fib(1) = 1$

- What are the **sub-problems**?
  - $Fib(n-1), Fib(n-2), \ldots, Fib(1), Fib(0)$

- What is the **magic** step?
  - $Fib(n) = Fib(n-1) + Fib(n-2)$

Magic step is usually not provided!!
The Fibonacci Problem

- Which **topological order** should I use?
  - $\text{Fib}(0), \text{Fib}(1), \ldots, \text{Fib}(n-1), \text{Fib}(n)$
The Fibonacci Problem

- Design iterative **algorithm**

```python
function Fib(n):
    fibs = []
    fibs[0] = 0
    fibs[1] = 1

    for i from 2 to n:
        fibs[i] = fibs[i-1] + fibs[i-2]

    return fibs[n]
```
The Change Problem

- Given \( T \) ($ amount) and \( c_1, \ldots, c_d \) (coins) compute
  - \( \text{change}(T) = \text{smallest number of coins that sum to } T \)
- What are the **sub-problems**?
  - \( \text{change}(T-c_1), \ldots, \text{change}(T-c_d) \)
- What is the **magic** step?
  - \( \text{change}(T) = \min\{\text{change}(T-c_1), \ldots, \text{change}(T-c_d)\} + 1 \)

**Magic step was not provided!!**
The Change Problem

- Which **topological order** should I use?
  - change(1), change(2), ..., change(T)
- Design iterative **algorithm**

```python
function change(T):
    opt = []
    for t from 1 to T:
        for i from 1 to d:
            if c_i < t
                if opt[t-c_i] + 1 < opt[t]
                    opt[t] = opt[t-c_i] + 1
    return opt[T]
```
Longest Common Subsequence

- Given strings \( V \) of \( n \) characters and \( W \) of \( m \) characters compute
  - length of longest common subsequence (LCS) of \( V \) and \( W \)
  - ex: LCS(Providence, Rodent) = 5
    - “roden”
  - not necessarily contiguous
- What are the sub-problems?
  - LCS(\( V_{n-1} \), \( W_{m-1} \)), \ldots, LCS(\( V_1 \), \( W_1 \))
  - where \( X_i \) is the prefix of \( X \) until \( i \)th character
- What is the magic step?
  - LCS(\( V, W \)) = ?
Longest Common Subsequence

- Given LCS($V_{n-1}, W_{m-1}$)
- Case #1: $V_n = W_m$
  - last characters are equal
  - LCS increases by 1
- Case #2: $V_n \neq W_m$
  - last characters are not equal
  - if LCS ends with $V_n$ then it is LCS($V_n, W_{m-1}$)
  - if LCS does not end with $V_n$ then it is LCS($V_{n-1}, W_m$)
- The LCS is $\max\{\text{LCS}(V_n, W_{m-1}), \text{LCS}(V_{n-1}, W_m)\}$
Longest Common Subsequence

- What is the **magic** step?
  - LCS($V, W$) is
    - LCS($V_{n-1}, W_{m-1}$) + 1 if $V_n = W_m$
    - $\max\{\text{LCS}(V_n, W_{m-1}), \text{LCS}(V_{n-1}, W_m)\}$ if $V_n \neq W_m$

- Which **topological order** should I use?
  - “left to right and top row to bottom row”
Longest Common Subsequence

- Design iterative **algorithm**

```python
function LCS(V, W):
    common = []
    for i from 0 to n+1
        common[i,0] = 0
    for j from 0 to m+1
        common[0,j] = 0
    for i from 1 to n+1
        for j from 1 to m+1
            if V[i] = W[j]
                common[i,j] = common[i-1,j-1] + 1
            else
                common[i,j] = max(common[i,j-1], common[i-1,j])
    return common[m,n]
```
Functional Programming
Final Review

CS16: Introduction to Data Structures & Algorithms
Spring 2018
Outline

- Paradigm
- Approaches
- Using Map
- Using Reduce
- Practice Problems
Functional Programming Paradigm

- A style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions.

- Programs written in this paradigm rely on smaller methods that do one part of a larger task. The results of these methods are combined using function compositions to accomplish the overall task.
Approaches

- How do we decide to use reduce vs. map?
  - Map creates a one to one mapping - we use it for problems that involve doing the same thing to multiple elements.
  - Reduce is more for modification - problems that involve ‘choices’ or returning a subset.
Using Map

- How to choose the function?
  - What do you want to have happen to each element in the input_list?
  - Other variables needed for the function can be created outside of the map call if needed!

- Quick Tip
  - Built-ins/bound functions do not need to have their arguments written out.

```python
return map(hash, input_list)
```
Using Reduce 1/2

- How to choose the binary function?
  - Takes in the acc and each successive element in the input list.
  - Think about how to breakdown your task!
    - the max of an entire list -> the max of two integers
    - remove all successive duplicates -> check if 2 elements are equal
  - Remember ternary syntax!

```python
a if condition else b
```
Using Reduce 2/2

- How to choose the accumulator?
  - Needs to be of the type that you are returning
  - If you’re comparing elements to elements, you should use the first element as a list as the starting acc, not an empty list!
    - Why?
  - If you’re comparing elements to a condition, you should use an empty list not the first element!
    - Why?
Import List Syntax

- `[x]`
  - makes a list out of element `x`

- `my_list[-1]`
  - returns the last element in `my_list`
  - `-1` is the element before `0`

- `my_list + [x]`
  - returns a new list of `my_list + [x]`, and does not modify the original list.
  - don’t use `append`! this modifies the original list and returns nothing.
Practice Problems

› Write a function that will turn a list of nouns into adverbs. (ex: loud -> loudly)

› Write a function that sums the total length of a list of strings. (ex: [“hi”, “cs16”] -> 6)

› Write a function that counts the number of times the string “dog” appears in a list of strings.

› Write a function that removes numbers less than 10 from a list of ints.
Practice Problem Answers

- $\text{map}(\lambda \text{el}: \text{el}+"\text{ly}"\, , \text{input\_list})$
- $\text{reduce}(\lambda \text{acc}, \text{el}: \text{acc}+\text{len}(\text{el})\, , \text{input\_list}, 0)$
- $\text{reduce}(\lambda \text{acc}, \text{el}: \text{acc}+[\text{el}] \text{ if } \text{el} == \text{"dog"} \text{ else acc, input\_list}, 0))$
- $\text{reduce}(\lambda \text{acc}, \text{el}: \text{acc}+[\text{el}] \text{ if } \text{el} > 10 \text{ else acc, input\_list}, [])$
PageRank Final Review

CS16: Introduction to Data Structures & Algorithms
Spring 2018
PageRank Overview

- How can we best search for a web page?
- Ranks web pages by importance
- Unlike text files, web pages have links to other web pages
- Underlying assumption is that “more important websites are likely to receive more links from other websites.”
PageRank Algorithm

- The Web is a graph $G = (V, E)$
  - Vertices are web pages
  - Edges are links
- The pagerank of a page $P$ is a function of the number of other pages that point to it and the rank of those pages
- If the pages that point to $P$ have a high pagerank, $P$'s pagerank will be very high
Basic PageRank Pseudocode

BasicPageRank(G, k):
  for v in V:
    v.rank = 1/|V|;
  for i from 1 to k
    for v in V:
      for u in v.incoming:
        v.rank = v.rank + u.rank/|u.outgoing|;

Runtime: \(O(|V| + |E|)\)
Basic PageRank Example
Basic PageRank Example

Initial state

A → B, rank = 1/5
B → D, rank = 1/5
C → B, rank = 1/5
D → E, rank = 1/5
E → D, rank = 1/5

A → C, rank = 1/5
C → A, rank = 1/5
Basic PageRank Example

A ----> B ----> D
    |    |      
    v    v      k = 1
C ----> B ----> D
    |    |      
    v    v      rank = 1/2
E

rank = 7/10
rank = 1/5
rank = 1/5
rank = 1/5
rank = 1/5

Basic PageRank Example

rank = 1/5

A

B

rank = 3/2

D

rank = 4/5

C

E

rank = 1/5

rank = 1/5

k = 2
Basic PageRank Example

- Rank of A: $\frac{1}{5}$
- Rank of B: $\frac{13}{5}$
- Rank of C: $\frac{1}{5}$
- Rank of D: $\frac{11}{10}$
- Rank of E: $\frac{1}{5}$

$k = 3$
Basic PageRank

- What is an issue with this algorithm?
  - Pagerank can be concentrated between a subset of nodes in the graph
Full PageRank Algorithm

- In the real PageRank algorithm, \( u \) does not give all its pagerank to its neighbors
  - Instead, it gives its neighbors only a \( \delta \) fraction of its pagerank
  - It then distributes the remaining \((1 - \delta) \cdot \text{rank}(u)\) of its pagerank evenly among all vertices

- Runtime: \( \mathcal{O}(|V| + |E|) \)

\[
\text{rank}(v) = \sum_{u \in \text{incoming}(v)} \left( \delta \cdot \frac{\text{rank}(u)}{|\text{outgoing}(u)|} + \frac{(1 - \delta)}{|V|} \cdot \text{rank}(u) \right)
\]
How do we know when to stop?

- Since a vertex’s pagerank is based on its incoming edges, when do we stop updating it?
- Does it ever stabilize?
- Yes! This is what $k$ represents - the number of iterations at which the rank stabilizes.
- We can use spectral analysis to prove that this stabilizes.
Online Algorithms and Competitive Analysis

Final Review

CS16: Introduction to Data Structures & Algorithms
Spring 2018
Offline Algorithms

- An offline algorithm has access to all of its data at the start – it “knows” all of its data in advance.
- Most of what you have done in this class has been offline (or at least given offline).
Online Algorithms

- An *online algorithm* does not have access to all of the data at the start.
- Data is received *serially*, with no knowledge of what comes next.
- How do you make a good algorithm when you don’t know the future?
Ski-Rental Problem (Offline)

Offline solution is somewhere on the blue line

Total $  

$500

Time

10 Days

49
Ski-Rental Problem (Online)

Total $

$1000

$500

Days

Online solution is somewhere on the purple line
Comparing Solutions

Total $

$500

Days

Online solution

Offline solution

10
Competitive Analysis

- Analyzing an online algorithm by comparing it to an offline counterpart
- **Competitive ratio**: Ratio of performance of an online algorithm to performance of an optimal offline algorithm

\[
\text{perf online} \leq c \cdot \text{perf offline} + \alpha
\]

- Our ski-rental solution has a competitive ratio of 2, since we are never more than 2 times as bad as the offline solution
- Our online algorithm is “2-competitive” with the offline solution
Defining Problem Hardness

- Hardness of problem is defined by the runtime of the best solution
  - A bad sorting algorithm could be $O(n!)$, but sorting in general isn’t considered hard, because we have fast algorithms to solve it

- Polynomial Runtimes
  - $O(n), O(n^2), O(n^{500})$
  - Problems with these solutions are **tractable**

- Super-Polynomial Runtimes
  - $O(n!), O(2^n), O(n^n)$
  - Problems with these solutions are **intractable**
Categories of Hardness

- **NP**
  - The set of problems for which we can verify the correctness of a solution in polynomial time

- **P**
  - A subset of NP, where the problem is solvable in polynomial time

- **NP-Complete**
  - “The hardest problems in NP”
  - Solution is checkable in polynomial time
  - Not known whether there exist any polynomial time algorithms to solve them

- **NP-Hard**
  - Problems that are “at least as hard as the hardest problems in NP”
  - Don’t necessarily have solutions that are checkable in polynomial time
Constraint Relaxation

- See if you can solve an ‘easier’ version of the problem, by removing some of the properties that make the problem hard.

- In real life
  - “what would you do if you could not fail?”
  - “which job would you take if they all paid equally?”
AUTOCORRECT AND AUTOCOMPLETE

CS16: Introduction to Data Structures & Algorithms
Outline

1. Autocorrect
2. Levenshtein Edit Distance
3. Autocomplete
4. Tries
5. Autocorrect with a Trie
Edit Distance

• The “edit distance” between two words is the number of insertions, deletions, and/or substitutions needed to transform one word into the other. Gives rise to **alignment**:

  \[
  \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
  & C & A & N & A & D & A & - & (\text{start word}) \\
  \text{B A N A N A S} & B & A & N & A & N & A & S & (\text{end word}) \\
  \end{array}
  \]

• Edit distance: 3 (1 insertion, 2 substitutions)
• Also called “Levenshtein edit distance” (LED)
Step 1: Make a Table

- colWord = “HELLO”
- rowWord = “SMELLY”

- Leave an empty row and column at the front of each word – they represent the base cases of the recurrence
Step 2: Describe each Cell in Words

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $T[i,j]$ holds the edit distance between the sub words `rowWord[0:i]` and `colWord[0:j]`

- Ex. – $T[4,2]$ is the edit distance between “SMEL” and “HE”
Step 3: Recurrence Relation

\[
T[i,j] = \min(T[i,j-1] + 1, T[i-1,j] + 1, T[i-1][j-1] + \{1,0\})
\]

depending on whether \(\text{rowWord}[i] == \text{colWord}[j]\)
Step 4: Decide the Order of Filling in the Table

- Can think of the table as a DAG
  - Directed edges represent dependencies
- Cells must be filled in a topologically sorted order
- You can fill this one out left to right, top to bottom, i.e. increasing $i$ and increasing $j$ order
Step 5: Fill in the Table

<table>
<thead>
<tr>
<th>Final word</th>
<th>“”</th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>M</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Final answer!

- First row and first col: base cases
  - Edit distance between empty string and sub-word is always just the length of the sub-word
  - The extra space adds the base case
- Everything else is the regular case
function editDistance(word1, word2):
    word1 = " " + word1  // pad out words with leading space
    word2 = " " + word2

    numCols = word1.size
    numRows = word2.size
    T = 2D array of size [numRows x numCols]

    for r from 0 to numRows-1:  // base cases!
        T[r][0] = r
    for c from 0 to numCols-1:
        T[0][c] = c

    for r from 1 to numRows-1:
        // go through by rows, going down
        for c from 1 to numCols-1:
            T[r][c] = min(T[r-1][c] + 1,  // insertion
                          T[r][c-1] + 1,  // deletion
                          T[r-1][c-1] + (0 if word1[c] == word2[r] else 1))  // substitution

    return T[numRows-1][numCols-1]  // get the final answer
Recreating the Alignment

- We can trace back to determine what edits transform “HELLO” to “SMELLY”
- Diagonal -> match/substitution
- Vertical -> insertion
- Horizontal -> deletion

<table>
<thead>
<tr>
<th>Final word</th>
<th>Start word</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>H E L L O</td>
</tr>
<tr>
<td>“”</td>
<td>0 1 2 3 4 5</td>
</tr>
<tr>
<td>S</td>
<td>1 1 2 3 4 5</td>
</tr>
<tr>
<td>M</td>
<td>2 2 2 3 4 5</td>
</tr>
<tr>
<td>E</td>
<td>3 3 2 3 4 5</td>
</tr>
<tr>
<td>L</td>
<td>4 4 3 2 3 4</td>
</tr>
<tr>
<td>L</td>
<td>5 5 4 3 2 3</td>
</tr>
<tr>
<td>Y</td>
<td>6 6 5 4 3 3</td>
</tr>
</tbody>
</table>
Autocomplete
Trie

• A trie stores an entire corpus of words in lesser space than it would take to store the words in a set

• A trie is a prefix tree
  • Every nod represents a prefix of a word

Yellow marks the end of a word
Trie Nodes

- Each node has a dictionary of children so it can easily map a letter to the correct child node.
- We also need to keep track of which nodes are the ends of words.

```python
class Node:
    children = dict()
    isWordEnd = False
```

Yellow marks the end of a word.
function insert(node, word):
    if word.length == 0:
        node.isWordEnd = true
        return

    firstLetter = word[0]
    if firstLetter not in node.children:
        node.children[firstLetter] = new Node()
    insert(node.children[firstLetter], word[1:end])
Lookup in a Trie

```
function find(node, prefix):
    // Output: the last node of the prefix, if it exists
    if prefix.length == 0:
        return node

    firstLetter = prefix[0]
    if firstLetter not in node.children:
        return "prefix not found"
    return find(node.children[firstLetter], prefix[1:end])
```
Autocomplete

- Call **find()** to find the last node in the prefix chain
- Run **DFS** using that node as a root to compile a list of matches
- Example: Autocomplete “AN”
Autocomplete

function autocomplete(prefix):
    start = find(Trie.root, prefix)  //find node representing prefix
    return dfs(start, prefix)  //find all children words

function dfs(node, wordSoFar):
    //wordSoFar is the current prefix path from the root
    matches = []
    if node.isWordEnd:
        matches.append(wordSoFar)
    for child in node.children:
        childMatches = dfs(child, wordSoFar + child’s key)
        matches += childMatches
    return matches