L19: Greedy Algorithms, Huffman Coding

Alex Steiger

CompSci 201: Spring 2024

3/25/2024

1

Person in CS: Kathleen Booth

- 1922 – 2022
- British Mathematician, PhD in 1950
- Worked to design the first ***assembly language*** for early computer designs in the 1950s
- May have been the first woman to write a book on programming
- Early interest in ***neural networks***

2

Logistics, Coming up

- Today: Monday 3/25
  - Project P4: Autocomplete due
- This Wednesday 3/27
  - APT 7 (tree recursion problems) due
- Monday 4/1
  - Nothing due—P5 the week after

3

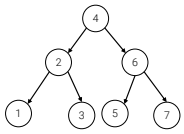
Today's agenda

- Solve HeightLabel APT
- Introduce Greedy Algorithms
- Huffman Coding (P5: Huffman)
  - Uses trees and greedy algorithms
  - Back to more tree-based data structures on Wed

4

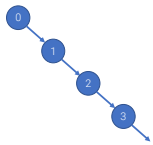
Balance and Trees

Balanced



For each node, left and right subtrees have roughly equal number of nodes.

Unbalanced



One subtree has many more nodes than the other.

5

Recurrence relation and runtime for traversing a *balanced* tree

- *T(n)* time for **`count(tree)`** with *n* nodes (balanced)

```
public int count(TreeNode tree) {
    if (tree == null) {
        return 0;
    }
    return 1 + count(tree.left) + count(tree.right);
}
```

n/2 nodes in this subtree

n/2 nodes in this subtree

- T(n) = 2T(n/2) + O(1)
- = O(n)

6

Recurrence relation and runtime for traversing *unbalanced* tree

- ***T(n)*** time for **`count(tree)`** with ***n*** nodes (unbalanced)

```
public int count(TreeNode tree) {
    if (tree == null) {
        return 0;
    }
    return 1 + count(tree.left) + count(tree.right);
}
```

1 node in this subtree

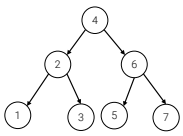
n-1 nodes in this subtree

- T(n) = T(1) + T(n-1) + O(1)
- = O(1) + T(n-1) + O(1)
- = O(n)

7

Balance Binary Search Tree Runtime (add, contains)

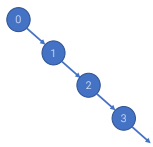
Balanced



T(n) = T(n/2) + O(1)
= O(log(n))

We will return to this problem later!

Unbalanced



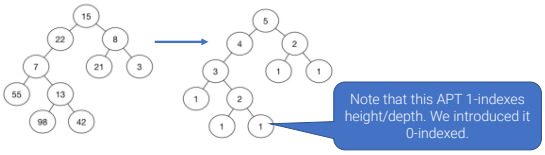
T(n) = T(n-1) + O(1)
= O(n)

8

HeightLabel APT

https://www2.cs.duke.edu/csed/newapt/heightlabel.html

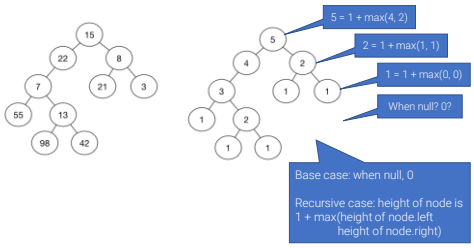
- Create a new tree from a tree parameter
  - Same shape, nodes labeled with height
  - Use **`new TreeNode`**. With what values …



Note that this APT 1-indexes height/depth. We introduced it 0-indexed.

9

Solving HeightLabel in Pictures



10

Live Coding HeightLabel

11

Solving HeightLabel in Code

```
private int height(TreeNode t) {
    if  (t == null) return 0;
    return 1 + Math.max(height(t.left),
                        height(t.right));
}
```

Base case: when null, 0

Recursive case: height of node is 1 + height of node.left + height of node.right

```
public class HeightLabel {
    public TreeNode rewire(TreeNode t) {
        // replace with working code
        return null;
    }
}
```

Method doesn't just calculate height, is supposed to create and return new tree with new nodes...

```
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
     return new TreeNode(height(t),
                         rewire(t.left),
                         rewire(t.right));
}
```

Using height helper method, get height, create new node, return.

12

Rewire runtime?

- Recurrence of this correct code? **T(n) =**

- **2T(n/2)+O(n)**
  - Balanced tree
- **T(n-1)+O(n)**
  - Unbalanced

```
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
     return new TreeNode(height(t),
                         rewire(t.left),
                         rewire(t.right));
}
private int height(TreeNode t) {
    if  (t == null) return 0;
    return 1 + Math.max(height(t.left),
                        height(t.right));
}
```

T(n)

T(n/2) if balanced

O(n)

T(n/2) if balanced

14

HeightLabel Complexity

- Balanced?
  - **T(N) = 2T(n/2) + O(n) → O(N log N)**
- Unbalanced,
  - **T(N) = T(N-1) + O(N) → O(N^2)**
- Doable in **O(N)** time? Yes, if we don't call height
  - Balanced: **T(N) = 2T(N/2) + O(1)**
  - Unbalanced: **T(N) = T(N-1) + O(1)**

15

HeightLabel in O(N) time

- If recursion works, subtrees store heights!

- Balanced? **O(N)**,
  - **2T(n/2)+O(1)**
- Unbalanced, **O(N)**
  - **T(N-1)+O(1)**

```
public TreeNode rewire(TreeNode t) {
    if (t == null) { return null; }
    TreeNode leftOfMe = rewire(t.left);
    TreeNode rightOfMe = rewire(t.right);
    int lHeight = 0;
    int rHeight = 0;
    if (leftOfMe != null) { lHeight = leftOfMe.info; }
    if (rightOfMe != null) { rHeight = rightOfMe.info; }
    return new TreeNode(
            x: 1+Math.max(lHeight, rHeight),
            leftOfMe,
            rightOfMe);
}
```


16

L19-WOTO1-HeightLabel-Sp24

Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *

solutions

2

Here is the recursive height helper method we wrote (1 indexes height). Which recurrence relation best describes the runtime complexity of height as a function of N = the number of

nodes in the tree t **assuming the tree is balanced**? *

```
1   private int height(TreeNode t) {
2       if  (t == null) return 0;
3       return 1 + Math.max(height(t.left),
4                           height(t.right));
5   }
```

- (●) T(N) = 2T(N/2) + O(1)
- ( ) T(N) = 2T(N/2) + O(N)
- ( ) T(N) = 2T(N-1) + O(1)
- ( ) T(N) = T(N-1) + O(1)

3

Here is another version of the height helper method. Will it work correctly? *

```
1   private int height(TreeNode t) {
2       int value = 1 + Math.max(height(t.left), height(t.right));
3       if  (t == null) return 0;
4       return value;
5   }
```

- ◯ Yes, this is equivalent to the previous approach
- ◯ No, this will not compile
- ◯ No, this computes different height values
- ◉ No, this generates a runtime exception

4

Here is an equivalent version of the rewire method that uses the height helper method. Which recurrence relation best describes the runtime complexity of rewire as a function of N = the number of nodes in the tree t **assuming the tree is balanced**? You may assume that the height method is O(N). *

```
 8   public TreeNode rewire(TreeNode t) {
 9       if (t == null) return null;
10       int myHeight = height(t);
11       TreeNode lchild = rewire(t.left);
12       TreeNode rchild = rewire(t.right);
13       return new TreeNode(myHeight, lchild, rchild);
14   }
```

- ◯ T(N) = 2T(N/2) + O(1)
- ◉ T(N) = 2T(N/2) + O(N)

- ◯ T(N) = 2T(N-1) + O(1)
- ◯ T(N) = T(N-1) + O(1)

5

Suppose you run this recursive method on some tree t. During the execution of the program, what is myHeight for the **first** (in terms of execution of the code) TreeNode that gets returned by line 13 on any of the recursive invocations of the method? *

```
 8   public TreeNode rewire(TreeNode t) {
 9       if (t == null) return null;
10       int myHeight = height(t);
11       TreeNode lchild = rewire(t.left);
12       TreeNode rchild = rewire(t.right);
13       return new TreeNode(myHeight, lchild, rchild);
14   }
```

- ◯ The height of the tree, because we start at the root
- ◉ 1, because a leaf node will be the first created (and this is 1-indexed)
- ◯ Cannot determine without seeing the particular tree

Greedy Algorithms for Discrete Optimization

17

Optimization

- Find the solution that maximizes or minimizes some objective
- Example: **Knapsack**
  - Find the bundle of items with maximum value without exceeding a budget.
  - What should you buy if you have $10?
  - (Only one of each item)

Items	Value	Cost
	2	$1
	1	$1
	12	$10

18

Greedily Searching for Optima

- Start with a partial solution. In each iteration make a step toward a complete solution.
- ***Greedy principle***: In each iteration, make the step that "best improves" the solution (e.g., the lowest cost or highest value step).
- Knapsack example:
  - Partial solution is a set of items you can afford
  - Greedy step: Add the item with best value per cost ratio that you can afford with remaining money

19

Local Optima vs Global Optima?

Greedy algorithms do **not** always guarantee to find the best overall solution, called ***global optima***.

0. Start with $10
1. Buy apple, best value/cost. $8 remaining
2. Buy banana (can't afford pizza). $7 remaining
3. Done: Can't afford any more items. **Total value of items = 3**

Items	Value	Cost	Value/Cost
	2	$1	2
	1	$1	1
	12	$10	1.2

But just buying the pizza has value 12, which is the (only) global optimum

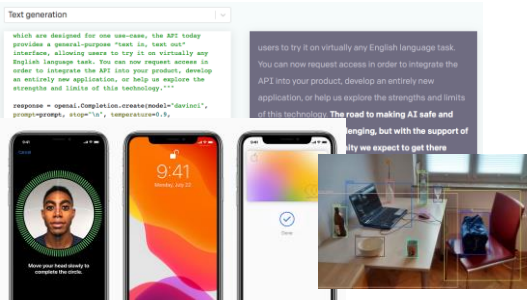
20

Why Learn Greedy Algorithms?

1. Sometimes a greedy algorithm is optimal (always returns global optima). Examples:
   - Huffman Compression (today, Project 5)
   - Computing shortest paths in networks/graphs
2. Sometimes the greedy algorithm is not optimal, but still works well in practice
3. A greedy algorithm is typically easy to start with for optimization problems.

21

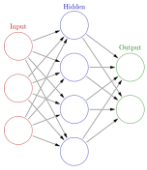
Aside: What is Machine Learning?



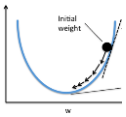
22

Aside continued – How do you "learn a model" greedily?

- Often (in deep learning) represent a model with a ***neural network***.
- Learn model: optimize parameters of network on data.



- How to optimize the parameters?
  - Greedy algorithm called ***gradient descent***
  - At each step, make a small change that best improves model performance



23

Huffman Coding

Topic of Project 5: Huffman

24

Huffman Compression

Representing data with bits: Preferably fewer bits

- Zip
- Unicode
- JPEG
- MP3

Huffman compression used in all of these and more!

25

Encoding

- Eventually, everything stored as bit sequence: 011001011…
- *Fixed length encoding*
  - Each value has a unique bit sequence of the *same length* stored in a table.
  - With N unique values to encode, need $\lceil \log_2(N) \rceil$ bits per value.
  - E.g., with 8 characters, need 3 bits per character.

ASCII coding

char	ASCII	binary
g	103	1100111
o	111	1101111
p	112	1110000
h	104	1101000
e	101	1100101
r	114	1110010
s	115	1110011
space	32	1000000

3-bit coding

char	code	binary
g	0	000
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101
s	6	110
space	7	111

26

Optimizing Encoding?

- Suppose we have three characters {a, b, c}:
  - a appears 1,000,000 times
  - b and c appear 50,000 times each
- Fixed length encoding uses 2,200,000 bits:
  - $\lceil \log_2(3) \rceil = 2$ bits per character
  - 2 bit/char * 1,100,000 chars = 2,200,000 bits
- *Variable length encoding*: Use *fewer* bits to encode *more common* values, *more* bits to encode *less common* values.
  - What if we encode: a = 1, b = 10, c = 11?
  - Only uses 1,200,000 bits.

27

Decoding Fixed Length

- Fixed Length with length k
  - Every k bits, look up in table
  - 001 001 010 110
    - 001 → o
    - 001 → o
    - 010 → p
    - 110 → s

3-bit coding

char	code	binary
g	0	000
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101
s	6	110
space	7	111

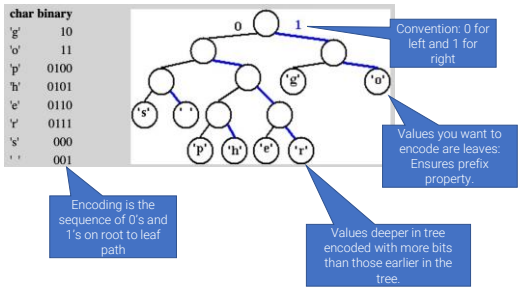
28

Decoding Variable Length

- What if we use
  - a = 1
  - b = 10
  - c = 11
- How would we decode 1011?
  - "baa" or "bc?"
- Problem: Encoding of a (1) is a ***prefix*** of the encoding for c (11). Ambiguous!

29

Prefix Property: Encoding as a Tree



30

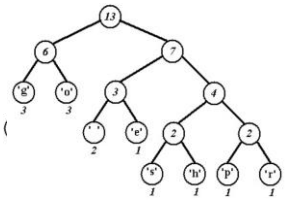
Huffman Coding

- ***Greedy*** algorithm for building an optimal variable-length encoding tree.
- High level idea:
  - Start with the leaves/values you want to encode with weights = frequency. Then repeat until all leaves are in single tree:
  - ***Greedy*** step: Choose the ***lowest-weight nodes*** to connect as children to a new node with weight = sum of children.
- Implementation? Priority queue!

31

Visualizing the Algorithm

Encoding the text "go go gophers"



char	binary
'g'	00
'o'	01
'p'	1110
'h'	1101
'e'	101
'r'	1111
's'	1100
' '	100

32

P5 Outline

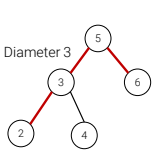
1. Write Decompress first
   - Takes a compressed file (we give you some)
   - Reads Huffman tree from bits
   - Uses tree to decode bits to text
2. Write Compress second
   - Count frequencies of values/characters
   - Greedy algorithm to build Huffman tree
   - Save tree and file encoded as bits

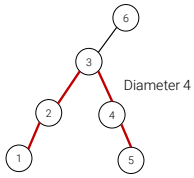
34

Diameter Problem

leetcode.com/problems/diameter-of-binary-tree

Calculate the *diameter* of a binary tree, the length of the longest path (maybe through root, maybe not, can't visit any node twice).





Live Coding

35