Discussion 7(11/16)

ECE 17

Statistical Analysis/Timing of code

- Always think before you implement code.
- Why?
    - Blindly implemented code and its subsequent debugging might take longer than thinking before coding
    - You might dig yourself into a hole, trying to debug dysfunctional code
    - Bad/inefficient usage of algorithms

Statistical Analysis/Timing of code

- This is for HW5 Part 1 tips, for word count and dictionary populating
    - Words with capitalized characters are the same word
        - “Hello” and “hello” are the same word
        - Sample solution => set all the words lowercase or read/parse everything as lowercase
    - Punctuation
        - “Isn’t” is a word
        - So only checking for letters here is not sufficient:
            - isalpha() or if(char >= ‘a’ && char <= ‘z’) by themselves will fail here

Timing

- Remember every nested loop you have equals to exponential growth in time
- For example:
    - `for(int i = 0; i < 5; i++)`
        - `cout("Hello")` //This will run 5 times

    - `for(int i = 0; i < 5; i++)`
        - `for(int i = 0; i < 5; i++)`
            - `cout("Hello")` //This will run 25 times

    - `for(int i = 0; i < 5; i++)`
        - `for(int i = 0; i < 5; i++)`
            - `for(int i = 0; i < 5; i++)`
                - `cout("Hello")` //This will run 125 times!!!

Timing

Example: Given a **color** and **shape** and arrays possible versions of each(**shape[]** and **color[]**). Find out of the given **color** and **shape** combination is possible. Let's say each array has 20 elements

- for(int i = 0; i < shape.size; i++) //Bad version
    - for(int j = 0; j < color.size; j++
        - if(color == color[j] && shape == shape[i])
            - Cout << "Yes"
- The bad version will run 20^2 times = 4000 iterations!!!
- If we have 100 possibilities of each, that will be 10,000 iterations!!!!

Lets try a better version

Timing

- Boolean shapePossible = false;
- Boolean colorPossible = false;
- for(int i = 0; i < shape.size; i++)
    - shapePossible= (shape==shape[i])?true:false;
- for(int i = 0; i < color.size; i++)
    - colorPossible= (color==color[i])?true:false;
- if(colorPossible && shapePossible)
    - Cout << “Yes”;

This will run 20+20 times = 40 iterations, a huge improvement.

If we have 100 possibilities of each, it will be 200 iterations, instead of 10,000

Data Compression

- Basic idea: what if we can represent longer repetitive/common with a shorter place holder
- Used in zip files
- We would want lossless compression, aka compression that has no data loss
    - Non lossless or lossy compression will be compression with data loss.
        - Ex: turning a 4k photo into a 2k photo. You can never get the lost data back

Data Compression

Fun example:

Spider-man was **happy** to see **happy** and now they’re both **happy**

If we set the word “happy” to “1”, we now get:

Spider-man was **1** to see **1** and now they’re both **1**

In replacing one word, we shaved down 12 characters

Dictionary Compression

We can expand our compression example, what if we swap every word out. We know almost every english sentence/paragraph/written work as repetitive words.

Ex:

Jingle bells, jingle bells, jingle all the way

We use the dictionary:

1 2 1 2 1 3 4 5

The encoded version uses so much less data

Dictionary	
word	**encoding**
jingle	1
bells,	2
all	3
the	4
way	5

Huffman Encoding

NOTE, this is a very very watered down version, please do your own research for this

- We want a even better version of compression, so we will use Huffman Encoding.
  - used for JPEG and MPEG-2
- Huffman Encoding takes in account for the frequency of the input and creates encoding where the more common a entry is, the shorter its encoding is.
  - Ex: if ‘a’ shows up 100 times and ‘b’ shows up 15 times, the length of encoding of ‘a’ will be shorter than length of encoding ‘b’
- Goal: set the most common entry(word or character) with the shortest encoded value to yield lowest overall data use

Huffman encoding

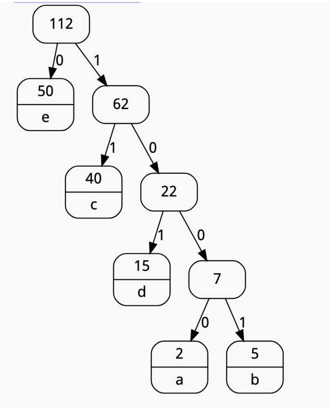
- General idea:
  - We first collect the frequency of each entry(word or character) from the input
  - Then we generate a tree with the most common entry as root
  - To generate the encoding, we set each path in the tree as ‘0’ or ‘1’, and we “walk” from the root to that node

Huffman encoding

Frequency Table	
Entry	**Freq**
a	2
b	5
c	40
d	15
e	50

- Example: Build a huffman encoding for this freq table:
  - The table means: for entry ‘a’, it showed up 2 times. ‘b’ showed up 5 times, ‘c’ showed up 40 times, and so on.

1) We build a frequency tree
   a) Each node on the main path denotes the sum of characters below it.
   b) Each node representing a character has its frequency above it



Huffman Encoding

3. We build the Encoding Table based on our tree

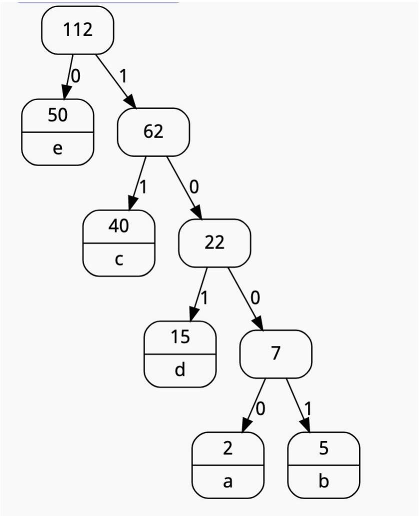
   This is achieved by “walking” from the root to that node

Notice how the highest frequency = shortest encoding
And opposite for lowest frequency

4. Now, used this encoding to replace your entries(chars, words) in your data input

<u>Again, this is a VERY VERY watered down Version, so please do you own research for Huffman encoding</u>**

Frequency Table		
Entry	**Freq**	Encoding
e	50	0
c	40	11
d	15	101
b	5	1001
a	2	1000



Any questions?