Announcements

- APT-7 due Thursday, April 16
- APT-8 due Tuesday, April 21
- Assignment P6 Huffman due April 22
  - All late work turned in by April 22 (APTs and Asgns)
  - Except Huffman grace through April 23
- Exam 2 last chance to take through Friday 4/17
- Final Exam will be on April 30 – any time on this day
- APT Quiz 2 is April 12-18 – Your own work!
- Assignment P7 Optional out – Extra Credit!

P7 Assignment – Create
Optional – Extra Credit

- Make something creative about CompSci 201
- Earn 2 points to your highest exam score.
- What you create
  - Video (to share)
  - Advertisement for CompSci 201
  - Story/Song about CompSci 201
  - Green Dance
  - Comic/Poem (to share)
  - Or Just give us feedback (not to share)

Susan Rodger
April 15, 2020
Plan for the Day

- **Algorithm Paradigms via APTs**
  - Backtracking Algorithms
  - Greedy Algorithms
  - Next week: more algorithmic paradigms

- **Huffman Compression**
  - Quintessential 201 (and greedy) assignment?
  - Greedy, trees, arrays, recursion, priority queues, reading files, writing files, bits, bytes, oh my!

Backtracking Summary

- Enumerate all possible moves/choices
  - Nqueen? Each column and each row in column
  - Blob-fill? Each neighbor: fill, and unfill
  - GridGame: try a move, follow it, undo/repeat

- State/Board often two-dimensional array/grid
  - Not efficient, but thorough: try it all

GridGame Backtracking Redux

- Helper method: do and undo

```java
private int winCount(char[][] board) {
    int count = 0;
    for(int r=0; r < 4; r++) {
        for(int c=0; c < 4; c++) {
            if (isMoveLegal(board,r,c)) {
                board[r][c] = 'X';
                int opponentCount = winCount(board);
                if (opponentCount == 0) {
                    count += 1;
                }
                board[r][c] = '.';
            }
        }
    }
    return count;
}
```
Collaborative APT Solving

- How many paths to reach cheese? 3 below
  - E,E,S,S,S,S
  - S,S,S,S,E,E
  - S,S,E,E,S,S: always closer, never further to goal

BackTrackRat

- Take a step toward the cheese, try every step …
  - If that works? Add +1 to total
  - If that doesn’t work? back-track

Backtracking APTs

- Often use grid[][] to store state/moves
  - In Java this is actually an array of arrays

- int[][] a = new int[4][4] for example
  - What is a[0]? What is a[0][0]?

- Often move must be explicitly undone
  - Sometimes just try everything

Rat/Transform input to grid

Input: String[], transform to char[][]
- ["X..X.X.", "XX.C.X.", ".......", ".X.X..", ".X.X.", ".......", "R.XX..."]

- [0][0] is upper left, 0th row/column
- Start at rat and ...
  - Try each step closer to cheese

E – moves east
S – moves south
Transform, Initialize, Solve

- State and behavior: local or instance variables
- Scope?
- Rectangle?
  - loop bounds

```java
public int numRoutes(String[] enc) {
    int ratRow = 0;
    int ratCol = 0;
    myBoard = new char[enc.length][enc[0].length()];
    for(int r=0; r < myBoard.length; r++) {
        for(int c=0; c < myBoard[r].length; c++) {
            myBoard[r][c] = enc[r].charAt(c);
            if (myBoard[r][c] == 'X') {
                ratRow = r;
                ratCol = c;
            } else if (myBoard[r][c] == 'C') {
                myCheeseRow = r;
                myCheeseCol = c;
            }
        }
    }
    return moveCount(ratRow, ratCol);
}
```

Base cases for cheese-finding

- Off the grid? No paths to cheese
- On an 'X'? No paths to cheese
- On the cheese? One path to cheese

```java
private int moveCount(int r, int c) {
    if (r < 0 || c < 0 || r >= myBoard.length || c >= myBoard[r].length) {
        return 0;
    }
    if (myBoard[r][c] == 'X') return 0;
    if (r == myCheeseRow && c == myCheeseCol) return 1;
}
```

Try every possible step that ...

- Closer to the cheese only, see line 50
  - What do we return? Recursive help

```java
int distanceNow = cheeseDistance(r,c);
int[] deltaRow = {-1,0,0,1};
int[] deltaCol = {0,-1,1,0};
int total = 0;
for(int k=0; k < deltaRow.length; k++) {
    int nr = r + deltaRow[k];
    int nc = c + deltaCol[k];
    if (cheeseDistance(nr,nc) < distanceNow) {
        int next = moveCount(nr,nc);
        total += next;
    }
}
return total;
```

WOTO


I changed my mind
We’ve come so far, so fast, that ever so many others could begin shedding old habits too. After all, freedom isn’t just an external concept, framed by our laws. It’s a gift of the spirit that we must give ourselves, in this case by going towards brighter shades of ‘out’. Bottom line: If you want to change the future, start living as if you’re already there.

https://www.huffingtonpost.com/lynn-conway/the-many-shades-of-out_b_3591764.html

Greedy Algorithm: Huff Prelude

- **Optimization**: Best choice, maximal or minimal
  - Make a choice that looks good locally
  - But local best leads to global optimum
  - In later courses: prove greedy is optimal

- **Canonical example? Change with minimal # coins**
  - Change for $0.63, change for $0.32
  - What if we’re out of nickels, change for $0.32?

Greedy Algorithms

- In change making with US coins: minimize # coins
  - Choose highest denomination. Repeat
  - Works with infinite number of each coin
    - Example with $0.32 and no nickels?

- Shortest path algorithm: choose "closest" point, move there: overall best. Careful on "closest"

- Huffman Compression: optimal per-character
  - Can't compress better one-char-at-a-time
Greedy APTs

- [https://www2.cs.duke.edu/csed/newapt/olympic.html](https://www2.cs.duke.edu/csed/newapt/olympic.html)
- How is Olympic Candles greedy?
  - What candle should be lit on first night? Why?
- [https://www2.cs.duke.edu/csed/newapt/voterigging.html](https://www2.cs.duke.edu/csed/newapt/voterigging.html)
- How is VoteRigging greedy?
  - From whom should a vote be taken? Why?

Olympic Candles APT

- Given (different) heights of N-candles
  - Day one: light one, Day two: light two, ...
  - When lit? Burns one inch of height
- How many days until candles out?
  - [2,2,2] --- 3 nights
  - [1,2,2], [0,1,2], XXX, not greedy doesn't work
  - [1,2,2], [1,1,1], [0,0,0]
  - [5,2,2,1] --- 3 nights
    - [4,2,2,1], [3,1,2,1], [2,0,1,1]

Greedy Olympic Solution

- On night N, which candles chosen to burn?
  - The tallest N candles, decrement each, repeat
  - N = 1,2,3,… until you don’t have enough
  - How to determine tallest N candles?
  - What’s complexity here?
- Worst-case? Re-sort each time, repeat N times
  - Final result? O(N² log N), why?

Candle Pseudo-code

- Can we sort in reversed order?
  - `Comparator.reverseOrder()`? no!

```java
public int numberOfNights(int[] candles) {
    int maxN = candles.length;
    for(int k=0; k < maxN; k++) {
        Arrays.sort(candles);
        int numToBurn = k+1;
        // for each of highest numToBurn candles
        // burn an inch if there is an inch to burn
        // if not? return numToBurn-1
        return -1; // never reach here
    }
}
```
Algorithmic Processes that Scale

• What is the 'A' in APT?
  • Typically efficiency is NOT an issue here?
  • What about in a tech/job interview?
• Consider Olympic candles, what candles burn?
  • Have to burn $N^2$ candles in worst-case
  • Find "best" better than $\log(n)$ per candle? no

• What does efficiency mean for algorithms?
  • Is $O(N^2)$ ok for sorting?

Huffman is a greedy algorithm

Overview of Huffman
Lossy v Lossless Compression

• RAW format compared to JPEG format
  • Tradeoffs – another example of "it depends"

• Why do you ZIP files/folders?
  • Upload to Dropbox/Box/Google Drive

• What are advantages of MP3
  • You were 0-3 years old
Huffman is Optimal

- We create an encoding for each 8-bit character
  - Can’t do better than this on per-character basis

- Normally ‘A’ is 65 and ‘Z’ is 90 (ASCII/Unicode)
  - A is 01000001 and Z is 01011010
  - Why does this make sense? 8- or 16-bit/char
  - Why doesn’t this make sense?
- Unicode and images/sound, use all 8 bits

Leveraging Redundancy

- If there are 1,000 “A” and 10 “Z” characters …
  - Use fewer bits for “A” and more bits for “Z”

- Huffman treats all A’s equally, no context
  - We use fewer bit for 'A', but are all A's equal?
  - Could use context: more than 8-bits at a time

- Other compression techniques can do better
  - Faster and better compression, more complex

Aside: Bit Interpretation

- What can we tell from file extensions
  - Foo.class, bar.jpg, file.txt, coolness.mp3
  - How does OS know how to open these?

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Hex Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo.class</td>
<td>0000000: cafe babe 0000 0034 001d 0a00 0600 0f09</td>
</tr>
<tr>
<td></td>
<td>0000010: 0010 0011 0800 120a 0013 0014 0700 1507</td>
</tr>
<tr>
<td></td>
<td>0000020: 0016 0100 063c 696e 6974 3e01 0003 2829</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PicassoGuernica.jpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewed using &quot;open ..&quot; and via &quot;xxd ..&quot;</td>
</tr>
<tr>
<td>Wikimedia &quot;knows&quot; how to display?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000: ffd8 ffe0 0010 4a46 4946 0001 0100 0001</td>
</tr>
<tr>
<td>0000010: 0001 0000 ffdb 0043 0008 0606 0706 0508</td>
</tr>
<tr>
<td>0000020: 0707 0709 0908 0a0c 140d 0c0b 0b0c 1912</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000: 4944 3303 0000 0000 0048 5458 5858 0000</td>
</tr>
<tr>
<td>0000010: 001a 0000 0045 6e63 6f64 6564 2062 7900</td>
</tr>
<tr>
<td>0000020: 4d79 7374 6572 7920 4d65 7468 6f64 5452</td>
</tr>
</tbody>
</table>
Huffman: Better Encoding

- Rather than 8 or 16 bits for every character
  - Fewer bits for frequently occurring characters
  - "go go gophers": from $13 \times 3 = 39$ to 37. Wow!

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>ASCII</th>
<th>3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>g</td>
<td>103</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>o</td>
<td>111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>p</td>
<td>112</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>h</td>
<td>104</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>e</td>
<td>101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>r</td>
<td>114</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>s</td>
<td>115</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>sp.</td>
<td>32</td>
</tr>
</tbody>
</table>

ASCII 3 bits

- g 1100111
- o 1101111
- p 1110000
- h 1101000
- e 1100101
- r 1110010
- s 1110011
- sp. 1000000

Create and Use Huffman Tree

- Create Tree, Create Encodings, Compress file
  - Read file to create tree, read file to compress
  - In tree: left is 0 and right is 1: example of Trie
    - Frequently occurring characters close to root

Huffman Compress Steps

- Read file and count every occurrence
  - Map of "char" to frequency, or a[c] += 1
- Create Tree using greedy algorithm
  - Use priority queue until one root left
- Create encodings based on tree
  - Every root-to-leaf path is encoding: character in leaf, path is encoding: store in map
- Read file and write new encoding for each char
  - Careful attention to indicate end of file

Greedy: create tree from counts

- All weighted nodes in PQ
- Remove two smallest
- Put together, add back
  - Heavy chosen late
Finishing go go gophers

- Tree -> Encodings
  - 0 left/1 right
  - How many bits? 37!!
  - More chars, more saving

From Trie to Encodings

- Compress: create encodings for each char/leaf
  - Similar to LeafTrails APT
  - Each 8-bit chunk/char mapped to encoding, e.g., in an array with codings[‘p’] == “1010”

  - We have 256 different 8-bit chunks, but have encodings for as many as 257 "characters"!
    - "character" is any 8-bit chunk, pixel to ASCII
    - PSEUDO_EOF is sentinel value, not real, but …

Benchmarking Huff with kjv10.txt

<table>
<thead>
<tr>
<th>Encoding Length</th>
<th># values with this length</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1,159,124</td>
</tr>
<tr>
<td>4</td>
<td>1,487,471</td>
</tr>
<tr>
<td>5</td>
<td>712,325</td>
</tr>
<tr>
<td>6</td>
<td>485,333</td>
</tr>
<tr>
<td>7</td>
<td>261,611</td>
</tr>
<tr>
<td>8</td>
<td>84,107</td>
</tr>
<tr>
<td>9</td>
<td>81,467</td>
</tr>
<tr>
<td>10</td>
<td>48,019</td>
</tr>
<tr>
<td>11</td>
<td>21,065</td>
</tr>
<tr>
<td>12</td>
<td>1,863</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoding Length</th>
<th># values with this length</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1,108</td>
</tr>
<tr>
<td>4</td>
<td>664</td>
</tr>
<tr>
<td>5</td>
<td>476</td>
</tr>
<tr>
<td>6</td>
<td>225</td>
</tr>
<tr>
<td>7</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>1,108</td>
</tr>
<tr>
<td>14</td>
<td>664</td>
</tr>
<tr>
<td>15</td>
<td>476</td>
</tr>
<tr>
<td>16</td>
<td>225</td>
</tr>
<tr>
<td>17</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>44</td>
</tr>
<tr>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>6</td>
</tr>
</tbody>
</table>

How does this make sense?

- Length 3? 4 characters
- Length 1? 3 characters
- Four different characters!
Reading bits: BitInputStream

- Classes that interface with java.io classes

```
12+import java.io.*;
15
16 public class BitInputStream extends InputStream {  
    • Read 1-32 bits at-a-time, return int
      int bit = in.readBits(1);  
      • What can the value of bit be here?
      • No more bits to be read and you try to read?
        • Return -1, no exception thrown
```

Huff Challenges

- Your code will read and write bits-at-a-time
  - You'll benefit from "shadow-printing" so you can "see" what your code does

- Write decompress first: we give you test files
  - You can compress and decompress to test your final program
  - One program depends on the other to work

Huff Constants – HuffProcessor.java

- These cannot be changed (final) hence constants

```
16 public static final int BITS_PER_WORD = 8;
17 public static final int BITS_PER_INT = 32;
18 public static final int ALPH_SIZE = (1 << BITS_PER_WORD);
19 public static final int PSEUDO_EOF = ALPH_SIZE;
20 public static final int HUFF_NUMBER = 0xfaceb200;
21 public static final int HUFF_TREE = HUFF_NUMBER | 1;
```

- How many bits to read for counting? 8 or ...
- How big is the "alphabet"? $2^8$ or ...
- What is magic number? HUFF_TREE or ...
Writing bits: BitOutputStream

• Classes that interface with java.io classes

```java
public class BitOutputStream extends OutputStream {
    public static final int BYTE_SIZE = 8;
    private static final int INT_SIZE = 32;

    Write 9-bits representing 'A'
    out.writeBits(9, 65);
```

Decompression with Huffman

• We need the trie to decompress
  - 0001001001001100110111

• As we read a bit, what do we do?
  - Go left on 0, go right on 1
  - When do we stop? What to do?

• How do we get the tree/trie to decompress?
  - Could store 256 counts/frequencies, use same code
  - Could store trie: read and write: saves space!

Huffman Decompression Steps

• (must write header/tree/trie when compressing)
• First read tree from compressed file
• Then read compressed data one-bit-at-a-time
  • Go left or right, zero or one
  • If reach a leaf? Write out character, **reset to root**

• Careful with knowing when to stop
  • Not when out of bits, reaching **PSEUDO_EOF**

Decompress

```java
public void decompress(BitInputStream in, BitOutputStream out) {
    int magic = in.readBits(BITS_PER_INT);
    if (magic != HUFF_TREE) {
        throw new HuffmanException("invalid magic number "+magic);
    }
    out.writeBits(BITS_PER_INT, magic);
    while (true) {
        int val = in.readBits(BITS_PER_WORD);
        if (val == -1) break;
        out.writeBits(BITS_PER_WORD, val);
    }
    out.close();
}
```
You can't write just 31 bits

- Generally files are written in chunks or blocks
  - Don't write one bit at a time or even 16
  - Efficiency concerns accessing slower memory
- Generally read/write 8 or 16 or 32 .. bits-at-a-time
- In compressed file: could store # bits at beginning
  - Then read that many bits, stop when done
- In compressed file: could store sentinel at end
  - Then read until sentinel value read, stop

PSEUDO_EOF

- Not actually the end-of-file
  - A bit-sequence that does not occur in actual file being compressed
  - How do we encode this?
- Create HuffNode(PSEUDO_EOF, 1)
  - Add to PQ, create encoding, 01010111
  - Last bits written: might write 01010111000
  - Read until PSEUDO_EOF found, stop

Huff WOTO


Out Takes from GoGoGophers

- FIU, UNSW, UCB, UIUC, StackOverflow, …
  - Why is this Google-able? Why CourseHero
  - Many papers build on this example