Drew Hurdle
July 6, 2014
Algorithms and Data Structures I
Professor Cummings

Lab 2
Morse Code

Section 1: How it works

The algorithm works by exhaustive backtracking.
The algorithm works in the following manner:

void findPossibilities(String code, PossibleMessage message)

- If the code's length is 0
  - Success case
  - The algorithm has found a possible message and adds it to the list of possible messages.
  - The algorithm returns.
- An array of words that could be the next word in the code is generated.
  - This is done by taking every possible length of the next word (1 to code length) and compiling the sets returned by getWordsFromCode.
  - Any words that are initials (i.e.
- If the array is empty
  - Failure case.
  - Return.
- For every word in the list
  - Calculate the frequency of this word following the previous word.
  - If this value is greater than the threshold passed into the object on construction
    - Recursive case
      - findPossibilities(code.subString (lastWord.length), message + thisWord);

Section 2 : Help Received

I discussed the scoring system with Jon Pearl, but I don't believe I changed anything based on his advice.

I had been weeding out possible messages that contained initials after the list of all possible messages had been generated, which meant I was branching when I didn't need to be. Devon Truman suggested I do it before branching on the word, which is why I put it in the generation of the list of possible next words.

Section 3 : Analysis

Because of the unpredictable nature of the getWordsFromCodes function in the provided DecodingDictionary class, we first must find an upper bound for this result. I did this by finding the longest list the method returned during my testing (using the full string of the hidden message in morse provided). I used a 0 threshold so I wouldn't be getting biased results from this test. I factored into this my subtraction of words which were initials (the character at index 1 was '.').

This list was 32 words long.

The shortest morse length is, of course, 1. Therefore, the recursion for the method is T(n - 1).

$T(0) = 1;$

$T(n) = 32n * T(n - 1) + c, T(0) = 1$

$T(n - 1) = 32(n-1) * T(n - 2) + c, T(0) = 1$

$T(n - 2) = 32(n-2) * T(n - 3) + c, T(0) = 1$

$T(n) = 32n * (32(n-1) * (32(n-2) * T(n - 3) + c) + c) + c, T(0) = 1$

$T(n) = 32(n! - (n-k)!) * T(n - k) + kc, T(0) = 1$

$k = n$ to get $T(0)$

$T(n) = 32n! * T(0) + nc, T(0) = 1$

$T(n) = 32n! + nc, T(0) = 1$

$T(n) \varepsilon \Theta (2^n)$ Factorial growth.

However, we know that the performance is significantly better than this.

Imagine an upper bound that was the maximum number of words a morse code of length n would return of all possible lengths of n.

That is to say, u = $\sum_{i=1}^{i \leq n} F(i)$ Where F(i) returns the maximum number of words the getWordsFromCode will return for a code of length i.

As we can clearly see that this will be a finite constant number, we can call it u and plug it into the original equation to reduce the algorithm complexity to exponential.

T(0) = 1
T(n) = u * T(n - 1) + c, T(0) = 1
T(n - 1) = u * T(n - 2) + c, T(0) = 1
T(n - 2) = u * T(n - 3) + c, T(0) = 1
T(n) = u (u (u T(n-3) + c) + c) + c

T(n) = u^k T(n-k) + c * $\sum_{i=0}^{i<n} u^i$

n = k

T(n) = u^n + c * $\frac{(u^{n+1}-1)}{(u-1)}$

Therefore, we can clearly see that T(n) ε Θ (2^n) (Exponential)

As such, we can easily do the math on a limit to prove T(n) is better than n!.

$$\frac{lim}{n\text{->}\infty}\left(\frac{2^n}{n!}\right) = \frac{2}{1}\cdot\frac{2}{2}\cdot\frac{2}{3}\cdot\ldots\ \frac{2}{n}$$

As we can clearly see that the numerator remains multiplicative of 2 while the denominator is multiplicaticative of an increasing number (n) every step, as n->∞, the limit goes to 0. Therefore, Θ(2^n) < Θ(n!)

Section 4 - Empirical Testing

Testing was performed on a 2013 Macbook Air i7 with 8 GB of RAM running OS X. I forced the JIT to compile to machine code before testing. Testing was performed by comparing System.currentTimeMillis() before and after running the code a number of times. Recorded times include only the algorithm runtime and not loading the decoding dictionary.

I decided on these techniques for testing for the folioing reasons:

- Averaging over many runs gives more accurate results
- Without forcing JIT to compile, averaged results are inaccurate
- System.currentTimeMillis was used because it’s very cheap and reliable

Averaged Algorithm Runtime

Test number	Message	Message length (in morse)	Threshold	Rank of real result	Time (ms)	Number of runs averaged over
1	TO BE OR NOT TO BE	30	200	1	0.4695	4000
2	I AM YOUR FATHER	34	200	4	0.71775	4000
3	SHOW ME THE MONEY	34	200	4	1.6655	4000
4	YOU CAN’T HANDLE THE TRUTH	53	200	5	5.3035	4000
5	LIFE IS LIKE A BOX OF CHOCOLATES	76	100	1	25.235	200