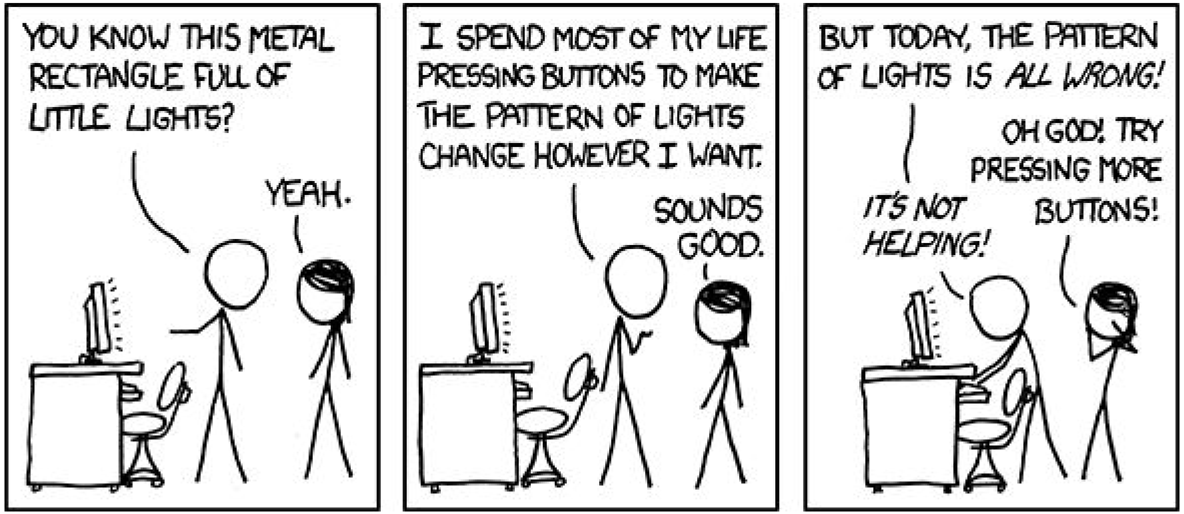
YEAH - Word Play

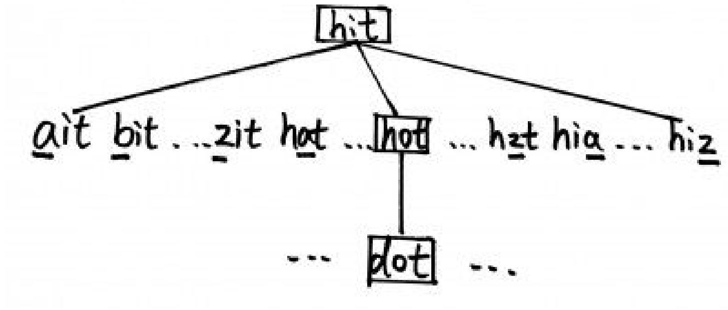
Anton Apostolatos


YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS *ALL WRONG!*
IT'S NOT HELPING!
OH GOD! TRY PRESSING MORE BUTTONS!

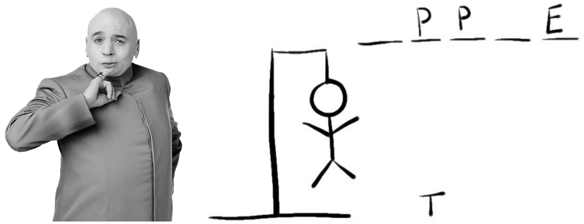
Source: XKCD

A2: Word Play

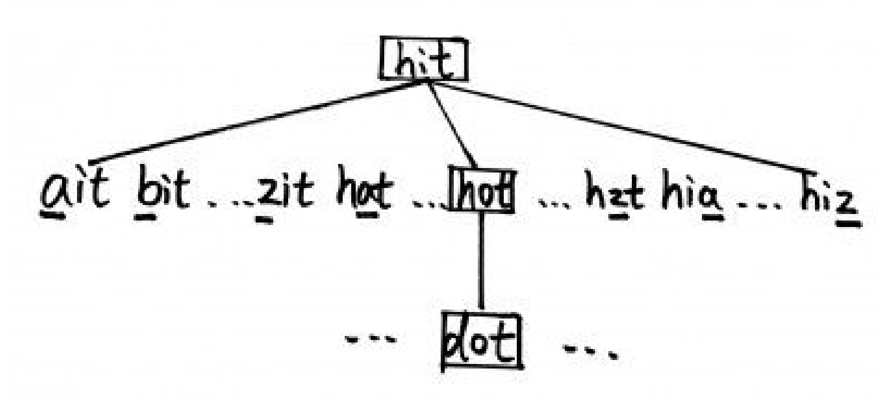
Word Ladders



Evil Hangman



Word Ladders



A **word ladder** is a connection from one word to another, where:

1) Each word is one character different than the previous

   `map` → `mat` ✓ `map` → `sit` ✗

2) Every word in the ladder is valid

   `blame` → `bhame` → `shame` ✗

3) Shortest possible!

   `bit` → `fit` ✓ `bit` → `sit` → `fit` ✗

Demo!

Pseudocode

```
create an empty queue
add the start word to the end of the queue
while (the queue is not empty):
    dequeue the first ladder from the queue

    if (the final word in this ladder is the destination word):
        return this ladder as the solution


    for (each word in the lexicon of English words that differs by one):
        if (that word has not been already used in a ladder):
            create a copy of the current ladder
            add the new word to the end of the copy
            add the new ladder to the end of the queue


return that no word ladder exists
```

How do we know it's the shortest path?

Design Decision

How to store ladder? Seen words?

Starter code

```
#include <iostream>
#include <string>
#include "console.h"
#include "RecordLadders.h"
using namespace std;

int main() {
   // [TODO: Fill this in!]
   return 0;
}
```

Steps

1. **Load the dictionary.** The file `EnglishWords.dat`, which is bundled with the starter files, contains just about every legal English word.

2. **Prompt the user for two words to try to connect with a ladder.** For each of those words, make sure to reprompt the user until they enter valid English words. They don’t necessarily have to be the same length, though – if they aren’t, it just means that your search won’t find a word ladder between them.

3. **Find the shortest word ladder.** Use breadth-first search, as described before, to search for a word ladder from the first word to the second.

Steps II

4. **Report what you've found.** Once your breadth-first search terminates:
    a. If you found a word ladder, print it out to the console, then call the function to report that you've found a word ladder:

    ```
    recordLadderBetween(start-word, end-word, ladder)
    ```

    b. If you don't find a word ladder, print out some message to that effect, then call the function

    ```
    recordNoLadderBetween(start-word, end-word)
    ```

5. **Ask to continue.** Prompt for whether to look for another ladder between a pair of words.

Tips and Tricks

- **Pick data structures wisely:** not all ADTs are made equal

- **Watch out for case sensitivity**

  `Work` ⟷ `wOrK`

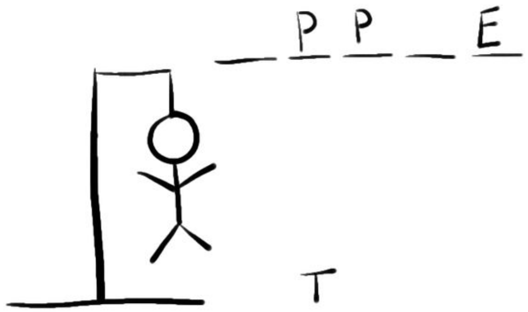
- **Ties don't matter:** don't worry about multiple ladders of the same length

  `bit` → `fit` → `fat` ✓ `bit` → `bat` → `fat` ✓

- **Make sure you call our functions!**

Questions?

Evil Hangman



Your Grandma’s Hangman

1. **Secret word chosen:** One player chooses a secret word, then writes out a number of dashes equal to the word length.

2. **Word guessing:** The other player begins guessing letters. Whenever she guesses a letter in the hidden word, the first player reveals each instance of that letter in the word. Otherwise, the guess is wrong.

3. **End condition:** The game ends when all letters in the word have been revealed or when no guesses remain.

Demo!

Not Your Grandma's Hangman

What if the secret word isn't a specific word, but every possible word?

Player:

_ _ _ _

⇩

_ _ _ _

Computer:

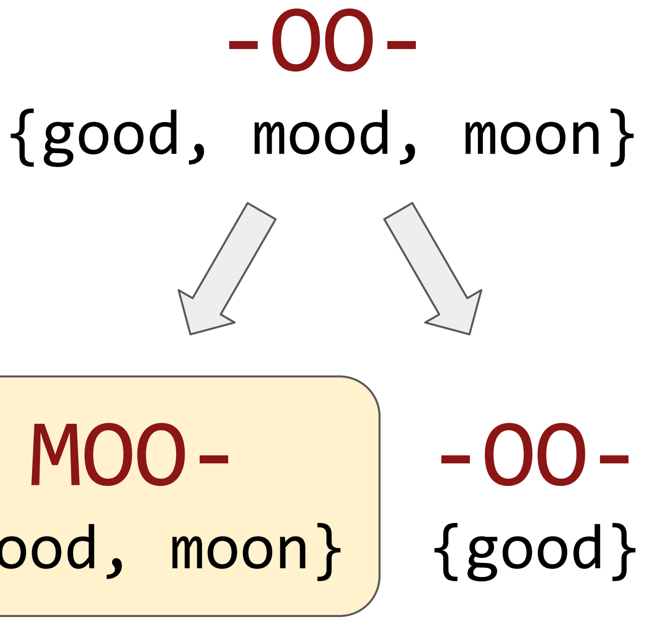
```
Word: {next}
```

⇩

```
Word(s):{able, area, … , zone}
```

Demo!

Guess: “M”



Guess: “E”

{ally, beta, cool, deal, else, flew, good, hope, ibex}

{ally, cool, good}

-E--

{beta, deal}

--E-

{flew, ibex}

E--E

{else}

---E

{hope}

Starter code

```
#include <iostream>
#include <string>
#include "console.h"
#include "Recorder.h"
using namespace std;

int main() {
   // [TODO: Fill this in!]
   return 0;
}
```

Steps

1. **Set up the game:** prompt, *in order*, for
    a. word length (*at least one word of that size must exist*)
    b. number of guesses (*integer greater than zero*)
    c. ask for running total or not (*yes/no*)

2. **Play the game**
    a. Print guesses remaining, letters guessed, blanked-out word
    b. Call our function:

```
recordTurnInfo(turn-number, current-blanked-word,
    letters-guessed-so-far, num-remaining-words, num-guesses-left)
```

Steps II

3. **Play the game** (cont...)
   a. Prompt for a single-letter guess
   b. Partition words into word families
   c. Find the most common “word family” in the remaining words, remove all words from the word list that aren't in that family, and report the positions of the letter guessed (if any) to the user. If the word family doesn't contain any copies of the letter, subtract a guess from the user.
   d. Repeat until game ends

Steps III

4. **Play the game** (cont…)
   a. If no guesses left, pick any word from remaining word list
   b. If word was guessed, print out resulting word!
   c. Call our function:

```
recordGameEnd(final-word, player-won)
```

5. **Ask to play again**

Design Decision

How to store word families?

Tips and Tricks

- **Decompose the problem into smaller pieces**

- **Letter position matters just as much as letter frequency**

  `-E-E-` vs. `EE---`

- **Don't worry about ties:** if two word families are the same size, split arbitrarily.

- **Don't enumerate word families:** if you are working with a word of length *n*, then there are *2^n* possible word families for each letter.

  "`supercalifragilisticexpialidocious`" → 2^34 ≈ # of galaxies in universe

- **Make sure you call our functions!**

Questions?