Compare the letters of two words

Return number of letters two words have in common

- Only works with words of the same length
- Regardless of position & upper/lower case
- Repeat letters match number of times in common

Examples:

- PEA vs EAT: 2
- TREE vs TRUE: 3
- APE vs pea: 3

Improving Compare

- "Working" is not the end
- You program for other coders, not the computer
    - Including future you
- Skimmability means no need to READ all the code

Array vs Object?

There are multiple approaches

First, let's review an approach using strings

- Which is basically an array of characters

Keep the problem simple

- If a letter matches
    - DO need to prevent it from double matching
    - Do NOT need to know WHERE it matched
- The more complex the info you "save"
    - The more complex the naming

This works…technically

```
function compare(str1,str2){
  var count=0;
  str1=str1.toUpperCase();
  str2=str2.toUpperCase();
  for(var i=0;i<str2.length;i++) {
    if(str1.includes(str2[i])) {
      count++;
      str1 = str1.replace(str2[i], '');
    }
  }
  return count;
}
```

Read the README

- `var` is for old engines, not modern
- Prefer `const`, use `let` when you can't
- Notice how informative the `let` below becomes

```
function compare(str1,str2) {
  let count=0;
  str1=str1.toUpperCase();
  str2=str2.toUpperCase();
  for(let i=0;i<str2.length;i++){
    if(str1.includes(str2[i])) {
      count++;
      str1=str1.replace(str2[i],'');
    }
  return count;
}
```

Whitespace makes code better

- Better despite no logic changes
- Comments should explain WHY, not WHAT
- "Paragraphs" of content with blank lines

```
function compare( str1, str2 ) {
  let count = 0;
  str1 = str1.toUpperCase();
  str2 = str2.toUpperCase();

  for (let i = 0; i < str2.length; i++){
    if (str1.includes(str2[i])) {
      count++;
      str1 = str1.replace(str2[i], ''); // Prevent rematch
    }
  }

  return count;
}
```

Naming things is important

Good names let your code explain itself

- `count` is a count of what?
- `str1`? `str2`?
- `i`?

To read/use this you have to mentally translate

```
str1 = str1.replace(str2[i], '');
```

Why not use the translated version right away?

```
word = word.replace(guess[index], '');
```

You will always be renaming

Each *line* is more clear, but overall is messy

```
function compare( word, guess ) {
  let matched = 0;
  word = word.toUpperCase();
  guess = guess.toUpperCase(); // Case insensitive match

  for (let index = 0; index < guess.length; index++){
    if (word.includes(guess[index])) {
      matched++;
      word = word.replace(guess[index], ''); // Stop rematch
    }
  }

  return matched;
}
```

Standard Array and String Methods useful

Some tried a nested for loop

- Instead of .includes() or .indexOf()

There are solutions for common problems

- Be sure to read up on methods on MDN

For...of loops when useful

A C-style for loop, but we don't care about the index!

```
function compare( word, guess ) {
  let matched = 0;
  word = word.toUpperCase();
  guess = guess.toUpperCase(); // Case insensitive match

  for(const guessLetter of guess) {
    if (word.includes(guessLetter)) {
      matched++;
      word = word.replace(guessLetter, ''); // Stop rematch
    }
  }

  return matched;
}
```

Before

```
function compare(str1,str2){
  var count=0;
  str1=str1.toUpperCase();
  str2=str2.toUpperCase();
  for(var i=0;i<str2.length;i++) {
    if(str1.includes(str2[i])) {
      count++;
      str1 = str1.replace(str2[i], '');
    }
  }
  return count;
}
```

After

```
function compare( word, guess ) {
  let matched = 0;
  word = word.toUpperCase();
  guess = guess.toUpperCase(); // Case insensitive match

  for(const guessLetter of guess) {
    if (word.includes(guessLetter)) {
      matched++;
      word = word.replace(guessLetter, ''); // Stop rematch
    }
  }

  return matched;
}
```

What it does is more clear

- Not about length

Common Concerns

- Do we worry about O()?
    - Performance?
    - Efficiency?
    - Time and Memory complexity?

Big O Notation

- Represents the order of growth
    - Run time or memory
    - Ignores multipliers not based on data count
- A loop over `n` items is `O(n)`
    - 2 sequential loops over `n` (2N) still `O(n)`
- Nested loops, `n` times over `n` times, is `O(n^2)`
- Inefficient algorithms run fine on small `n`
    - Fall apart with big `n`
    - Chessboard story = `2^n`

Dirty Secret: Dev Efficiency part of Performance

- Schools/interviews focus on efficiency
    - To make it a habit
    - Once it is habit, we worry less
        - Not zero worry, but less
- Outside of truly restricted resources
    - We may not worry until `O(n^2)`
    - Biz size matters: 100 million vs 1 million
    - Developer Efficiency often considered
        - Dev time more costly/rare than hardware
        - Keep it easy to understand/change/write

But nested arrays ARE `O(n^2)`

That's usually just too inefficient

- Dev time is costly/hard to scale
- But too slow is too slow

Jobs will have different levels of concern

- Algorithm efficiency is just one part
- But overall concern is very real
- Avoid situations you know are bad

Strings similar to arrays for O()

- We are looping over the letters (`n` letters)
- Looping hidden in built-in methods still looping
  - `.includes()` - `O(n)`
  - `.indexOf()` - `O(n)`
  - `.replace()` - `O(n)`

Performance Conclusion

Our Word Guess unlikely to be a performance problem

- `n` will always stay small
- But this `compare()` could be used in many ways
    - Including with large `n`
- Better to avoid `O(n^2)` whenever possible

Array or String approach here "works", but not great

Object Approach

Different algorithm concept

- Not: "compare each letter of each word"
- Instead: "count letters in both, then compare"

Counting letters of a word

```
const wordLetters = {};

for( let letter of word ) {
  if( wordLetters[letter] === undefined) {
    wordLetters[letter] = 1;
  } else {
    wordLetters[letter] += 1;
  }
}
```

Example: `GEESE`

```
{ G: 1, E: 3, S: 1 }
```

- Object lets us read count of a letter in `O(1)` time
    - Compare to `str.includes()` (`O(n)`)
- Loop, but not nested

This works...technically

```
function compare( word, guess ) {
  let count = 0;
  const obj = {};
  for(let i=0; i<word.length; i++) {
    if(obj[word[i].toLowerCase()]===undefined)) {
      obj[word[i].toLowerCase()]=1;
    } else {
      obj[word[i].toLowerCase()]++;
    }
  }
  for(let i=0; i<guess.length; i++) {
    if(obj[guess[i].toLowerCase()] > 0) {
      obj[guess[i].toLowerCase()]--;
      count++;
    }
  }
  return count;
}
```

Visual space makes it easier to skim

- Just like text, use space to make it easier to skim.
- Use "paragraphs" - blank lines between ideas
- There is no reward for tiny squished code

```
function compare( word, guess ) {
  let count = 0;
  const obj = {};

  for( let i = 0; i < word.length; i++ ) {
    if( obj[word[i].toLowerCase()] === undefined ) {
      obj[word[i].toLowerCase()] = 1;
    } else {
      obj[word[i].toLowerCase()]++;
    }
  }
//...
}
```

Good Variable Names are Important

- Variable and function names: main source of info!
- Name for what it holds/represents, not how
- No need to take out a few letters - just hurts

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( let i = 0; i < word.length; i++ ) {
    if( letterCount[word[i].toLowerCase()] === undefined ) {
      letterCount[word[i].toLowerCase()] = 1;
    } else {
      letterCount[word[i].toLowerCase()]++;
    }
  }
//...
}
```

Good Variable Names are HARD

Bad Names:

- `obj`, `ary`, `tmp`, `str`
- `map`, `dict`, `len`, `list`
- anything spleled wrong
    - VERY COMMON!
- `i`, `j`
    - what is it?

Usually Bad Names (too vague):

- `data`, `result`, `retval`, `count`

Do you actually need that index value?

- use `for..of` to get the value you care about (letter)

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( const letter of word ) {
    if( letterCount[letter.toLowerCase()] === undefined ) {
      letterCount[letter.toLowerCase()] = 1;
    } else {
      letterCount[letter.toLowerCase()]++;
    }
  }
//...
}
```

Pull out and name values

- Move repeated logic to another function
- DRY - Don't Repeat Yourself (common principle)
- DRY - Don't Repeat Yourself (common principle)

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( const letter of word ) {
    const lower = letter.toLowerCase();
    if( letterCount[lower]) === undefined ) {
      letterCount[lower] = 1;
    } else {
      letterCount[lower]++;
    }
  }
//...
}
```

Remove unneeded focus

- NOT about being **shorter**
- IS about **focus** of the eye

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( const letter of word.toLowerCase() ) {
    if( letterCount[letter] === undefined ) {
      letterCount[letter] = 1;
    } else {
      letterCount[letter]++;
    }
  }
//...
}
```

Use Truthy/Falsy

- Improve skimmability
- Draw eye to important parts
    - Not `===` or `isSomething`
- Remember: 0 is **falsy** (fine here, not always)

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( const letter of word.toLowerCase() ) {
    if( !letterCount[letter] ) {
      letterCount[letter] = 1;
    } else {
      letterCount[letter]++;
    }
  }
//...
}
```

Cautious use of Conditional Operator

- When assigning a value, can reduce "visual noise"
- ...or INCREASE visual noise
- Remember: Shorter is NOT the exact goal
- ...I'll pass this time

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( const letter of word.toLowerCase() ) {
    letterCount[letter] =
      letterCount[letter] ? letterCount[letter] + 1 : 1;
  }
//...
}
```

Pull out logic into more functions

- creates list of instructions instead of math
- Good to make the code DRYer
- ...I'll pass this time
    - Avoid Hasty Abstractions (AHA) principle
        - AHA opposes excessive DRYness

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  const increment = count => count ? count + 1 : 1;

  for( const letter of word.toLowerCase() ) {
    letterCount[letter] = increment(letterCount[letter]);
  }
//...
}
```

Not always post-inc/decrement

- ++ and -- aren't the only way to increase/decrease
- += 1 and -= 1 work, and allow for other numbers
- draw focus to what you're actually doing

```
function compare( word, guess ) {
  //.. some code above

  for( const letter of guess.toLowerCase() ) {
    if( letterCount[letter] ) {
      letterCount[letter] -= 1;
      matches += 1;
    }
  }

  return matches;
}
```

Defaulting and Short-Circuiting

- `&&` and `||` short circuit
- `&&` and `||` return a value
    - Not just boolean: `foo = foo || 'default';`
- Often used when `if` checks for truthyness
    - And assigns a value either way

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( const letter of word.toLowerCase() ) {
    letterCount[letter] = letterCount[letter] + 1 || 1;
  }

//...
}
```

Before...

```
function compare( word, guess ) {
  let count = 0;
  const obj = {};
  for(let i=0; i<word.length; i++) {
    if(obj[word[i].toLowerCase()]===undefined)) {
      obj[word[i].toLowerCase()]=1;
    } else {
      obj[word[i].toLowerCase()]++;
    }
  }
  for(let i=0; i<guess.length; i++) {
    if(obj[guess[i].toLowerCase()] > 0) {
      obj[guess[i].toLowerCase()]--;
      count++;
    }
  }
  return count;
}
```

...and After

```
function compare( word, guess ) {
  let matches = 0;
  const letterCount = {};

  for( let letter of word.toLowerCase() ) {
    letterCount[letter] = letterCount[letter] + 1 || 1;
  }

  for( let letter of guess.toLowerCase() ) {
    if( letterCount[letter] ) {
      letterCount[letter] -= 1;
      matches += 1;
    }
  }

  return matches;
}
```

The right answer?

"What is the right answer?"

It depends :)

- Which is easier to
- ...Understand?
- ...Maintain/change/adjust?
- ...Test and confirm?
- What is `O()`?

My Current Favorite - NOT Most Efficient!

```javascript
function compare( word, guess ) {
  function letterCountsOf( someWord ) { // Can move outside
    const letterCounts = {};

    someWord.toUpperCase().split('').forEach( letter => {
      letterCounts[letter] = letterCounts[letter] + 1 || 1;
    });

    return letterCounts;
  }

  const wordCounts = letterCountsOf(word);
  const guessCounts = letterCountsOf(guess);
  let matched = 0;

  for( const letter in guessCounts ){
    const wordCount = wordCounts[letter] || 0;
    const guessCount = guessCounts[letter] || 0;
    matched += Math.min( wordCount, guessCount );
  }
  return matched;
}
```

Why My Fav If Not Most Efficient?

```
function compare( word, guess ) {
  const wordCounts = letterCountsOf(word);
  const guessCounts = letterCountsOf(guess);
  let matched = 0;

  for( const letter in guessCounts ){
    const wordCount = wordCounts[letter] || 0;
    const guessCount = guessCounts[letter] || 0;
    matched += Math.min( wordCount, guessCount );
  }
  return matched;
```

- Clear logic
  - `letterCountsOf` separate from using counts
- Easy to change
  - Arrangement of Steps
  - Return value

Summary

- Functions should try to be 1-15 lines
- Names should be meaningful even by themselves
- Skimmability is about managing **focus**
    - Avoid visual noise
    - Avoid "squishing"
- People will argue about how best to do this
    - ...just like with human languages

Summary - Part 2

Impacts your grade:

- Meaningful Names (**useful** meaning!)
    - Not `i`, `obj`, `tmp`
- Aim for skimmability
- Never use `var`; Use `const` if can
- Never use loose comparison
    - Use truthy/falsy when appropriate