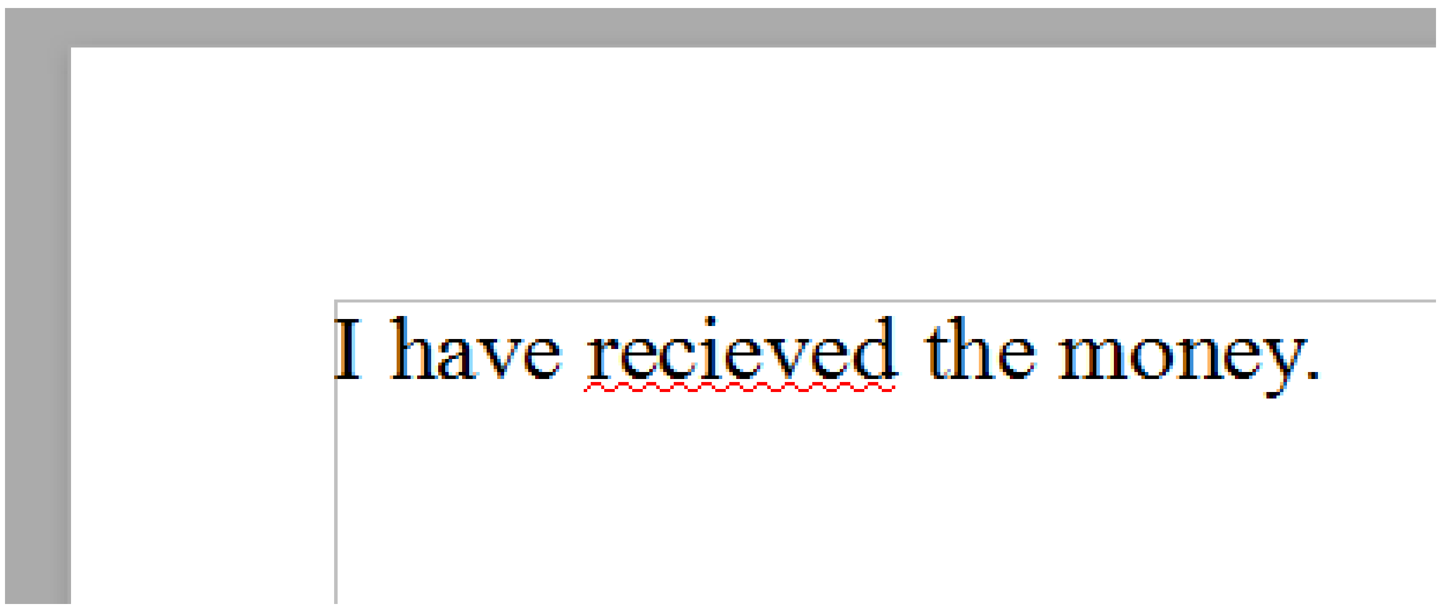
# Spelling Checking Algorithms

Doug Blank
Spring 2012
Algorithms: Design and Practice

# The Red Wavy Line



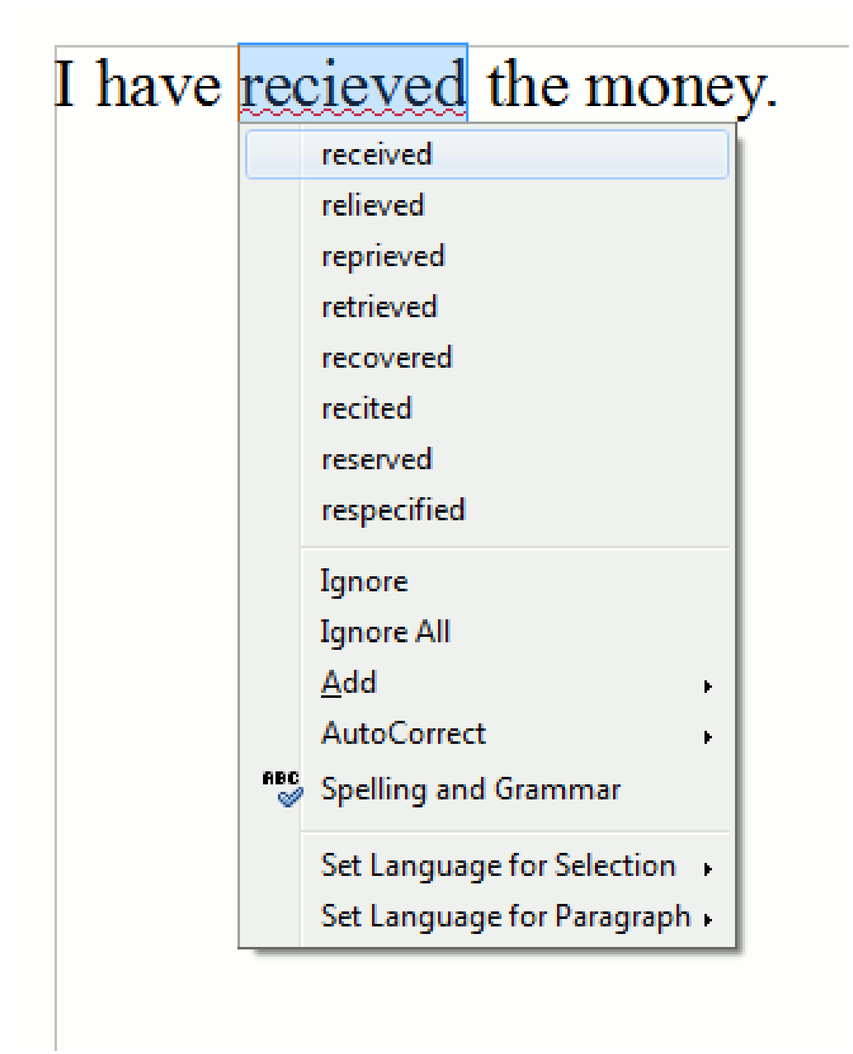
# Is it a “word”?

- Break string up into “words”
- Look up each word in a dictionary
- If found, then a word
- Else, not a word

# Not So Simple

- Would the dictionary store all variations of a word?
  - stump, stumped, stumping, stumps
- Some words are only correct if they have proper capitalization
  - Washington vs. washington
- Some “words” have spaces in them
  - au pair, et cetera, etc.

# Spelling Suggestions: How?



# Hot Topic!

- Levenshtein, self-correcting codes, 1966
- Wagner and Fischer, string-to-string correction problem, 1974
- Boyer-Moore, fast string matching, 1977
- Knuth, fast pattern matching, 1977
- Sellers, evolutionary distances, 1980
- Ukkonnen, approximate string matching, 1985
- Zobel and Dart, approximate string matches in a large lexicon, 1995

# Distance

- A distance between two “strings” can be computed that gives a measurement of the number of steps needed to turn one string into the other
  - distance(“apple”, “appl”) => 1 (deletion)
  - distance(“apple”, “bapple”) => 1 (insertion)
  - distance(“apple”, “bpple”) => 1 (substitution)
  - distance(“receive”, “recieve”) => 2
- Commutative, Transitive

# Levenshtein Distance #1

```
def distance(s1, s2):
    if len(s1) == 0: return len(s2)
    if len(s2) == 0: return len(s1)
    if s1[0] == s2[0]:
        return distance(s1[1:], s2[1:]) + 0
    else:
        return min( distance(s1[1:], s2[1:]) + 1,
                    distance(s1, s2[1:]) + 1,
                    distance(s1[1:], s2) + 1)
```

# Levenshtein Distance #2

```
def distance(s1, s2):
    if len(s1) == 0: return len(s2)
    if len(s2) == 0: return len(s1)
    cost = 0 if (s1[0] == s2[0]) else 1
    return min( distance(s1[1:], s2[1:]) + cost,
                distance(s1, s2[1:]) + 1,
                distance(s1[1:], s2) + 1)
```

# Problem!

- Recursive?
- Doesn't save previously computed answers

# “Dynamic Programming”

- Saving previously computed “subproblems”
- Typically using iteration, array

# Levenshtein Distance

```
int LevenshteinDistance(char s[1..m], char t[1..n])
{
   for i from 0 to m
    d[i, 0] := i // the distance of any first string to an empty second string
  for j from 0 to n
    d[0, j] := j // the distance of any second string to an empty first string
  for j from 1 to n
  {
    for i from 1 to m
    {
      if s[i] = t[j] then
        d[i, j] := d[i-1, j-1]       // no operation required
      else
        d[i, j] := minimum
                   (
                     d[i-1, j] + 1,  // a deletion
                     d[i, j-1] + 1,  // an insertion
                     d[i-1, j-1] + 1 // a substitution
                   )
    }
  }
  return d[m,n]
}
```

# Memoize

- Save the result of a computation based on the arguments given
- Results are “cached” and used later

# Memoize

```
def func(param1, param2):
    # have I computed this before?
    # if so, recall results, and return them
    # else, compute, save, and return them
```

# Python Function Decorators

- Uses the syntax "@fname" on line before function
- fname is a function which takes a function as an argument, and returns a function

# Function Decorators

```
def dec(f):
    print(“Here!”)
    return f

@dec
def func(a, b):
    return a + b
---------
Here!
>>> func(1, 2)
3
```

# Function Decorators

```
def dec(f):
    def m(*args):
        print(“Here!”)
        return f(*args)
    return m

@dec
def func(a, b):
    return a + b
--------
>>> func(1, 2)
Here!
3
```

# Memoized Levenshtein Distance

```
def memoize(f):
    cache = {}
    def m(*args):
        if args not in cache:
            cache[args] = f(*args)
        return cache[args]
    return m


@memoize
def distance(s1, s2):
    if len(s1) == 0: return len(s2)
    if len(s2) == 0: return len(s1)
    cost = 0 if (s1[0] == s2[0]) else 1
    return min( distance(s1[1:], s2[1:]) + cost,
                distance(s1, s2[1:]) + 1,
                distance(s1[1:], s2) + 1)
```

# Problem?

- The iterative, array-based method is basically equivalent to the recursive, memoized version
- However!
  - Many languages have a limited recursive call stack

# Lesson

- Try to separate the big idea from any implementational details
- The big idea is the algorithm