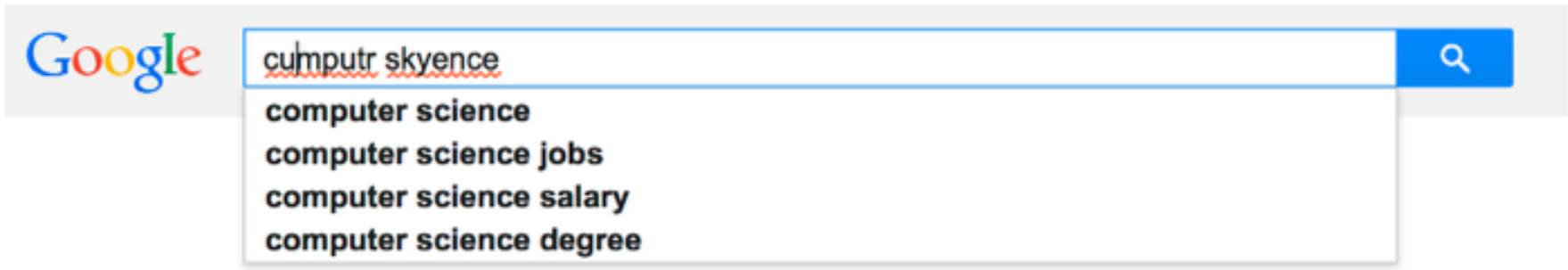
Spelling:

Edit Distance & Memoization

Spelling



- Microsoft Word, Google Search, etc., can perform *spelling correction* — have you ever wondered how they do that?!
- In order to determine if a word is misspelled, we need to be able to…
  - Compare words (misspelled with properly spelled words)
  - Pick relevant words to compare misspelled words to
- We will start by looking at how to compare words, then talk about how this can be used in a simple spelling corrector.

Edit Distance

- **Problem:** We want to determine similarity between strings.
- **Question:** What does it mean for two strings to be "close" to one another?
- **Similarity Metric:** "Edit Distance".
  - For two strings (*s1* & *s2*), the *edit distance* is defined as the minimum # of *editing operations* that transform *s1* into *s2*.
  - Possible operations
    - **Insert:** *add a single character. If string s = uv, then inserting the char. x produces uxv. This can also be denoted ε→x, using ε to denote the empty string.*
    - **Delete:** *delete a single character. e.g., uxv to uv (x→ε).*
    - **Replace:** *replace a single character x for a symbol y ≠ x. e.g., uxv to uyv (x→y).*
    - **Swap:** *swaps two characters. e.g., uxyv to uyxv.*
      - Note: this is useful for typos like "thier" vs. "their". (one op. to correct).
- Assume "cost" of each operation is equal.

Edit Distance

- Ex. editDistance(“Virginia”, “Vermont”)

Virginia	
Virginia	(replace)
Verginia	(replace)
Verminia	(replace)
Vermonia	(replace)
Vermonta	(delete)
Vermont	

- Thus, editDistance(“Virginia”, “Vermont”) = 5.
- How did we know which characters to swap, delete, etc.?

EditDistance.java: Overview

- Contains two methods: **naiveEditDistance()** and **editDistance()** that both solve the problem recursively.
- Each function looks at the first character of *s1* and *s2*.
- **If they match**, find the edit distance between the remaining strings.
- **If they don't match**, try *ALL* the possible operations to make the first characters match, then solve the remaining subproblems. The one that solves the problem in the fewest edits is the one we choose.
- Note: "match" and "replace" have the same subproblem(s) to solve, but their contributions to the edit distance differ by 1 (i.e., +0 for match, +1 for replace).

EditDistance.java (main)

- [Demo].
  - *s1* = “cats”, *s2* = “cotz”
  - *s1* = “Kate Blanchet”, *s2* = “Cate Blanchett”

```
public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     String s1, s2; // The strings to find the edit distance between

     System.out.println("Enter two strings, one per line");
     s1 = input.nextLine();
     s2 = input.nextLine();

     while (s1.length() > 0) {
          System.out.println("The edit distance between \"" + s1 + "\" and \"" + s2 +
                                                                "\" is " + naiveEditDistance(s1, s2) + "\n");
        // Get more input.
          System.out.println("Enter two strings, one per line");
          s1 = input.nextLine();
          s2 = input.nextLine();
     }
}
```

EditDistance.java (naive)

```
public static int naiveEditDistance(String s1, String s2) {
    int matchDist;    // Edit distance if first char. match or do a replace
    int insertDist;   // Edit distance if insert first char of s1 in front of s2.
    int deleteDist;   // Edit distance if delete first char of s2.
    int swapDist;     // Edit distance for twiddle (first 2 char. must swap).

    if (s1.length() == 0)
        return s2.length(); // Insert the remainder of s2
    else if (s2.length() == 0)
        return s1.length(); // Delete the remainder of s1
    else {
        matchDist = naiveEditDistance(s1.substring(1), s2.substring(1));
        if (s1.charAt(0) != s2.charAt(0))
            matchDist++; // If first 2 char. don't match must replace

        insertDist = naiveEditDistance(s1.substring(1), s2) + 1;
        deleteDist = naiveEditDistance(s1, s2.substring(1)) + 1;

        if (s1.length() > 1 && s2.length() > 1 && s1.charAt(0) == s2.charAt(1) && s1.charAt(1) == s2.charAt(0))
            swapDist = naiveEditDistance(s1.substring(2), s2.substring(2)) + 1;
        else
            swapDist = Integer.MAX_VALUE; // Can't swap if first 2 char. don't match

        return Math.min(matchDist, Math.min(insertDist, Math.min(deleteDist, swapDist)));
    }
}
```

cats

cots

cats

cots

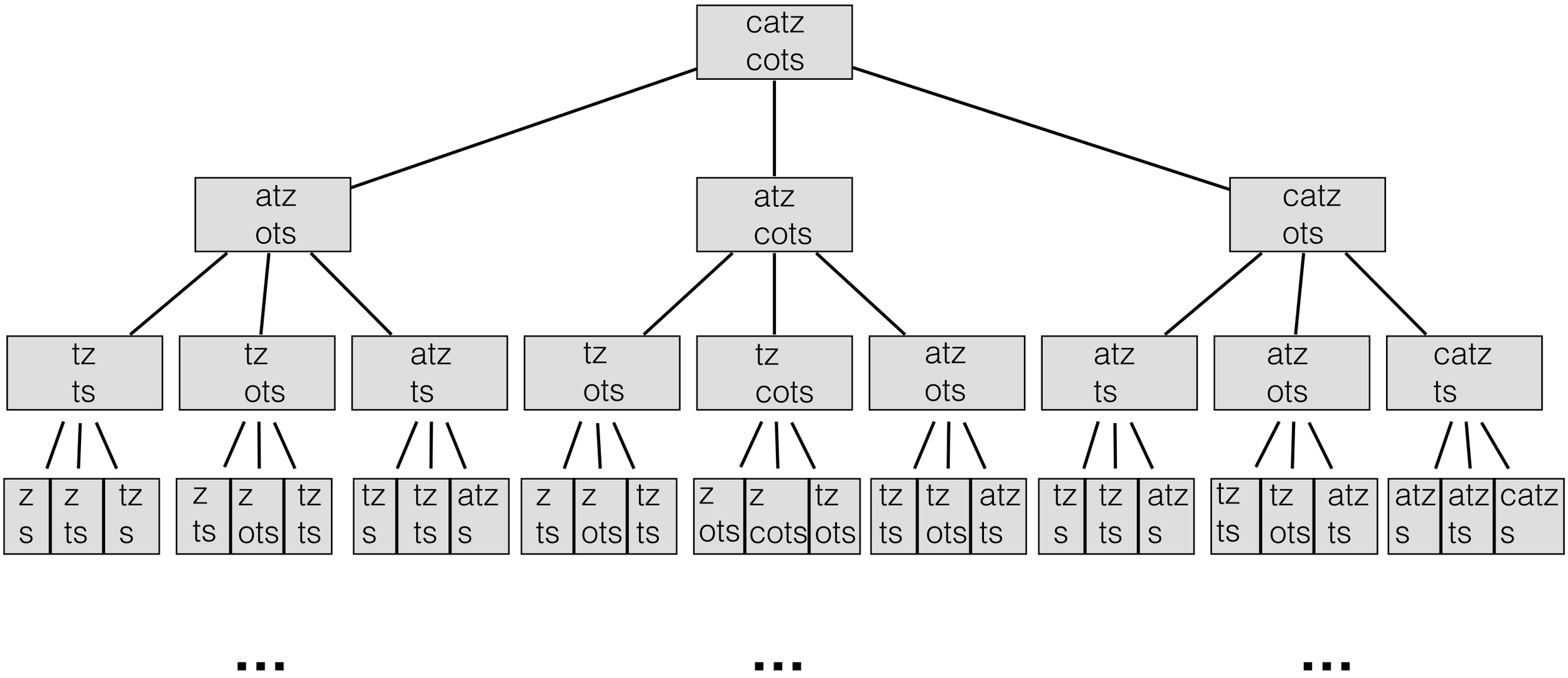
cats

cots

“swap”

(not relevant in this example)

EditDistance.java (naive)



Question: *Can we do better?! Lots of repeated computations…*

Note: *this tree only shows match/replace, insert, and delete. The swap op. has been left out since it is a more specific, special case operation. It would, however, add a possible 4th branch to each recursive step.*

Memoization

- Better approach: Memoization.
- “Dynamic Programming”; benefits/motivation:
  - A special type of algorithmic design pattern (simpler code)
  - Produces polynomial-time algorithms to solve problems that would seem to take exponential-time.
- Basic Idea:

  Keep a matrix of subproblem answers and fill it in using an order that guarantees the subproblem’s solution is known before you use it to fill in another place in the table.

  - Key: Identify/reuse overlapping subproblems.
  - Memory/Runtime tradeoff: use more memory, but save on computation (i.e., store results of already solved subproblems).
- Runtime:
  - *O(m*n)* were *m* and *n* are the length of *s1* and *s2*, respectively.

EditDistance.java (main)

```
public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     String s1, s2; // The strings to find the edit distance between
     EditDistance calc = new EditDistance();

     System.out.println("Enter two strings, one per line");
     s1 = input.nextLine();
     s2 = input.nextLine();

     while (s1.length() > 0) {
       System.out.println("The edit distance between \"" + s1 + "\" and \"" + s2 +
                                                          "\" is " + calc.memoizedEditDist(s1, s2) + "\n");

         // Get more input.
         System.out.println("Enter two strings, one per line");
         s1 = input.nextLine();
         s2 = input.nextLine();
     }
}
```

EditDistance.java (memoized)

```
private Map<StringPair, Integer> solvedProblems;

...

public int memoizedEditDist(String s1, String s2) {
    solvedProblems = new HashMap<StringPair, Integer>();
    return editDist(s1, s2);
}

...

private int editDist(String s1, String s2) {
    ...
    else {
        StringPair pair = new StringPair(s1, s2);
        Integer result = solvedProblems.get(pair);

        if (result != null) // Did we find the subproblem in the map?
            return result;  // If so, return the answer
        else {
            ...
            int dist = Math.min(matchDist, Math.min(insertDist, Math.min(deleteDist, swapDist)));
            solvedProblems.put(pair, dist); // Save the result for future
            return dist;
        }
    }
}
```

StringPair.java

```java
public class StringPair {

    private String s1, s2; // The pair of strings

    /**
     * Construct a new pair
     *
     * @param str1 the first string
     * @param str2 the second string
     */
    public StringPair(String str1, String str2) {
        s1 = str1;
        s2 = str2;
    }

    public boolean equals(Object other) {
        StringPair otherPair = (StringPair) other;
        return s1.equals(otherPair.s1) && s2.equals(otherPair.s2);
    }

    public int hashCode() {
        return s1.hashCode() + 31 * s2.hashCode();
    }
}
```

EditDistance.java (memoized)

		c	**a**	**t**	**s**
	c	0	1	2	3
	o	1	1	2	3
	t	2	2	1	3
	z	3	3	3	2

- [Do Demo again w/ memoized code].
  - *s1* = “catz”, *s2* = “cots”
  - *s1* = “Kate Blanchet”, *s2* = “Cate Blanchett”

Spelling Corrector

Spelling.java

- Some Notes:
  - A link to the spell corrector algorithm we will look at linked from the notes.
  - Initial implementation in Python — ported to Java (Spelling.java).
- Basic Idea for how it works:
  1. Use some *measure* to compute distance between two words.
     - *Edit Distance* is a reasonable measure; other measures work too.
  2. Only look at words with minimum edit distance (i.e., E.D. = 1).
  3. Pick most common word in English
     - Determined by “big.txt” (compiled from books and lists of most frequent words in Wiktionary, etc.).
     - “bigger.txt” also includes the Unix spelling dictionary (to incorporate more rarely used words).

Spelling.java

Algorithm:

- Find all words of minimum edit distance.
- Return the one that occurs most frequently in big.txt.

Spelling.java

- [Demo].

```
/**
 * Original version read a single word to correct from the command line. It
 * is commented out below
 *
 * @throws IOException
 */
public static void main(String args[]) throws IOException {
    //Spelling corrector = new Spelling("inputs/big.txt");
    Spelling corrector = new Spelling("inputs/bigger.txt");
    Scanner input = new Scanner(System.in);

    System.out.println("Enter words to correct");
    String word = input.next();

    while (true) {
        System.out.println(word + " is corrected to " + corrector.correct(word));
        word = input.next();
    }
}
```

Spelling.java

```
private HashMap<String, Integer> nWords;

/**
 * Constructs a new spell corrector. Builds up a map of correct words with
 * their frequencies, based on the words in the given file.
 *
 * @param file the text to process
 * @throws IOException
 */
public Spelling(String file) throws IOException {
    nWords = new HashMap<String, Integer>();
    BufferedReader in = new BufferedReader(new FileReader(file));

    // This pattern matches any word character (letters or digits)
    Pattern p = Pattern.compile("\\w+");
    for (String temp = ""; temp != null; temp = in.readLine()) {
        Matcher m = p.matcher(temp.toLowerCase());

        /*
         * find looks for next match for pattern p (in this case a word). True if found.
         * group then returns the last thing matched. The ? is a conditional expression.
         */
        while (m.find())
            nWords.put((temp = m.group()), nWords.containsKey(temp) ? nWords.get(temp) + 1 : 1);
    }
    in.close();
}
```

Spelling.java

```
public String correct(String word) {
    // If in the dictionary, return it as correctly spelled
    if (nWords.containsKey(word))
        return word;

    ArrayList<String> list = edits(word); // Everything edit distance 1 from word
    HashMap<Integer, String> candidates = new HashMap<Integer, String>();

    /*
     *  Find all things edit distance 1 that are in the dictionary. Also remember
     *  their frequency count from nWords.
     *
     *  (Note if equal frequencies the last one will be the one remembered.)
     */
    for (String s : list)
        if (nWords.containsKey(s))
            candidates.put(nWords.get(s), s);

    // If found something edit distance 1 return the most frequent word
    if (candidates.size() > 0)
        return candidates.get(Collections.max(candidates.keySet()));

    /*
     * Find all things edit distance 1 from everything of edit distance 1.
     * These will be all things of edit distance 2 (plus original word). Remember frequencies
     */
    for (String s : list)
        for (String w : edits(s))
            if (nWords.containsKey(w))
                candidates.put(nWords.get(w), w);

    /*
     * If found something edit distance 2 return the most frequent word.
     * If not return the word with a "?" prepended. (Original just returned
     * the word.)
     */
    return candidates.size() > 0 ? candidates.get(Collections.max(candidates.keySet())) : "?" + word;
}
```

Spelling.java

```
/**
 * Constructs a list of all words within edit distance 1 of the given word.
 *
 * @param word the word to construct the list from
 * @return a list of words with in edit distance 1 of word
 */
private ArrayList<String> edits(String word) {
    ArrayList<String> result = new ArrayList<String>();

    // All deletes of a single letter
    for (int i = 0; i < word.length(); ++i)
        result.add(word.substring(0, i) + word.substring(i + 1));

    // All swaps of adjacent letters
    for (int i = 0; i < word.length() - 1; ++i)
        result.add(word.substring(0, i) + word.substring(i + 1, i + 2) + word.substring(i, i + 1) +
                                                                          word.substring(i + 2));

    // All replacements of a letter
    for (int i = 0; i < word.length(); ++i)
        for (char c = 'a'; c <= 'z'; ++c)
            result.add(word.substring(0, i) + String.valueOf(c) + word.substring(i + 1));

    // All insertions of a letter
    for (int i = 0; i <= word.length(); ++i)
        for (char c = 'a'; c <= 'z'; ++c)
            result.add(word.substring(0, i) + String.valueOf(c) + word.substring(i));

    return result;
}
```