Recursion isn’t so hard

Recursion: Basis and Progress

A recursive method may call itself on a smaller problem

Always divided into basis and progress:
- **Basis:** Solve a small problem directly
- **Progress:** Break off a smaller problem to solve with a recursive call

Factorial

\[ n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n \]

**Recursive definition:**
- \( n! = 1 \quad \text{if } n < 2 \)
- \( n! = n \times (n-1)! \quad \text{if } n \geq 2 \)

From definition to method ...

**Recursive definition:**
- \( n! = 1 \quad \text{if } n < 2 \)
- \( n! = n \times (n-1)! \quad \text{if } n \geq 2 \)

**Java code:**
```java
static int fact(int n) {
    if (n < 2) { return 1; }
    return n * fact(n - 1);
}
```
fact(5):
  return 5 * fact(5 - 1)

fact(4):
  return 4 * fact(4 - 1)

fact(3):
  return 3 * fact(3 - 1)

fact(2):
  return 2 * fact(2 - 1)

fact(1):
  return 1
Recursion vs Loop

```java
static int fact(int n) {
    if (n < 2) {
        return 1;
    }
    return n * fact(n - 1);
}
```

```java
static int factLp(int n) {
    int prod=1;
    for (int i=2; i <= n; ++i) {
        prod = prod * i;
    }
    return prod;
}
```

Recall Inductive Loop Design

```java
while (not basis case) {
    make the problem a little smaller;
}
solve the basis case;
```

Almost the same

```java
foo(problem) {
    if (basis case) {
        return the solution;
    } else {
        foo(smaller problem);
    }
}
```

Recall facts for computing gcd(a,b)

```java
gcd(n,0) = n
    because n × 0 = 0, for all n

gcd(a,b) = gcd(b,a)
gcd(a, b) = gcd(b,a mod b) if a > b
    progress case for loop or recursion
```
Recursive Method

```java
int gcd(a, b) {
    if (b == 0) { // base case
        return a;
    }
    // progress case
    return gcd(b, a % b);
}
```

Compare to the loop

```java
int gcd(a, b) {
    while (b > 0) {
        if (b == 0) { // gcd(a, b) = (a mod b, a)
            return a;
        }
        // gcd(a, b) = (a mod b, a)
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

Palindrome (assignment)

How does the induction work?
What does “make the problem smaller” mean?
What are the basis cases?
What is the progress case?

Edit Distance (Assignment Part 2)

One way spelling “suggestions” work:
Find “close” dictionary words: A small number of edits (mistakes) changes one to another

Example: edit distance fleagle => league is 2

```
U
X L E A G X E
```
Edit Distance as Search

Problem size: The whole string starting from the first character

Progress: Consider the suffix starting at the next character

Mismatched characters: At least one change required
Consider three possible edits:
- Insert a letter
- Delete a letter
- Substitute a letter

What’s the recursion?

We want to determine if badWord is within edit distance k of goodWord.

What are the basis cases?

What are the progress cases?

Each change just moves one index or both
Edit Distance k: Progress

The base cases will be more obvious if we consider progress cases first.

Break down each non-empty word into a first character (prefix) and the rest (suffix)

What if the prefixes match?
What if the prefixes don’t match?

Edit distance: Progress

Each word is prefix char, suffix string

What if the prefixes don’t match?

Edit distance is the 1 + edit distance after forcing a match by changing a character

Apply one edit to consume a character, then check edit distance of suffixes

Edit distance: Progress

Each word is prefix char, suffix string

What if the prefixes match?

Edit distance is the same as edit distance of suffixes. (Progress: shorter strings to check)

If edit distance from “T” to “OAT” ≤ k

At each point we’re looking at the suffix beyond the grey box. With each move, we expand the grey box.
Basis cases

Is edit distance from goodWord to badWord < k?

if k < 0, no

if goodWord and badWord are both empty, yes

Basis cases?

For progress, we reduce length of suffix possibly reduce edit distance (on mismatch)

When are we done?

Basis Case: Successful Match

Distance ≤ 2?

Distance ≤ 0?

Distance ≤ 1?

{substitute A for i}

YES!
The little trick with “##”

We could do this without “#” at the ends of words
  • But the logic would be complicated ... always checking if one or both indexes were beyond the end of the string

“Sentinel!” or “guard characters” make it easier
  • With enough #### at the end, we never have to make a special check for the end of the string.
  • When both prefixes are ‘#’, we have matched

Edit distance is depth first search

// basis cases
if (edit distance < 0) return false;
if (prefix of both strings is “#”) return true;

// progress cases
if (prefixes match) try suffixes, same edit distance;
try suffixes after insertion, deletion, substitution with edit distance reduced by 1.

Depth First Search

A classic algorithm, natural for recursion:
if (basis case) {
    return the answer;
}
while (some progress case possible) {
    if (progress case succeeds) return true;
}
return false;
In Java ...

```java
boolean nearMatch( int maxEdits,
    String fromWord, int fromPos,
    String toWord,  int toPos ) {
    // Basis case: Too many edits, fail
    if (maxEdits < 0) { return false; }
    // Basis case: End of both words, succeed
    if (fromWord.charAt(fromPos) == GuardChar &&
        toWord.charAt(toPos) == GuardChar) {
        return true;
    }
    // Progress cases
    ...
```