Week 3 part 1: review of recursion
an essential technique for AI

Simple examples
- Some textbooks show the factorial function as a typical illustration.
- But factorial isn't typical at all.
- And a recursive implementation of factorial is ridiculous!

Why?

A central criterion
- Each level of the factorial(n) reduces n by just 1.
- Therefore it will take 50 recursive function invocations to compute 50!
  - That's monstrously inefficient
  - There must be a better way.
- Effective recursive algorithms should reduce the size of the sought solution geometrically, typically by half.

Here's one recursive version of the power function

```cpp
double power(double x, int n)
{
    if (n==0)        return 1.0; /* Base case */
    if (n < 0)      /* Negative power */
        return power(1.0/x, -n);
    return (x * power(x, n-1);
}
```

What do we think of that?
Will it work? Is it efficient?
An insight (just using mathematics)

- We know from the laws of exponents that:
  \[ x^{2n} = (x^n)^2 \]
  \[ x^{2n+1} = x(x^n) \]

  The big saving comes from the even power cases.

- How many multiplications are needed to compute \( x^{24} \)?

We saw a better recursive power implementation last week

```c
static double power(double x, int n)
{
    if (n == 0) return 1.0;  // Base case
    if (n < 0)                // Negative
        return power(1.0/x,-n); //   power
    if (n % 2 == 1) //  Odd    return x * power(x,n-1); //    power
    double tempo = power(x, n/2);//  Even
    return tempo * tempo;    //    power
}
```

Is there a still better way?

Example of recursive integer exponentiation

- \( x^{24} = (x^{12})^2 \)
  = ((x^6)^2)^2
  = (((x^3)^2)^2)^2
  = ((((x(x^2))^2)^2)^2)^2
  = ((((x(x(x))))^2)^2)^2

- Just five multiplications to compute \( x^{24} \).

Tail recursion

- When a recursive function call is the last executable statement in a function, the compiler can better optimize the code, since it needn’t restore the environment after the recursive invocation returns.

- A recursive invocation inside a return statement is a typical way of doing tail recursion.

- What construct in the C-family supports conditional tail recursion?
Here's a start

- static double power(double x, int n)
  {return n==0 ? 1.0 : . . . . }

Now what? What comes after the colon?

- Fortunately C and its descendants chose the right precedence rules, so we don't have to put a nested ?: clause inside another level of parentheses.

Here's the whole thing: just a return statement!

- public static double power (double x, int n)
  {return
    n   == 0 ? 1.0   // Base
    : n   == 1 ? x   // cases
    : n < 0 ? 1.0 / power(x,-n) // Neg. power
    : n%2 == 1 ? power(x,n-1)  * x // Odd power
    : (x = power(x,n/2)) * x:// Even power
  }

- Is that harder or easier to understand?
- Is there a bug? Do we care?
- What does that have to do with artificial intelligence?

Forward chaining inference

- In inference engines we often break a problem down into a sequence of condition tests.
  - The first one that's satisfied determines the result.
- Lisp and Clojure have a construct that's very similar to the C-family's ?: operator.
- What would the integer power function look like in Clojure?

Unification

- Definition: an algorithm that establishes whether two expressions (which may contain variables) have the same form.
  - Arthur Nunes-Hewitt
- i.e. given two expressions, is there a substitution for their free variables that will make them identical?
- Example: In these two expressions:
  - \*x + z
  - \*x + 1
  - The substitution \(z=1\) is a unifier.

COMP 378, Spring 2016
What about this?

- $x + z$
- $x + y$

- We could arbitrarily pick the following unifiers:
  $z = 6, \ y = 6$. Then the expressions would unify.
- However, this arbitrariness is troubling. We'd really like the 'most general unifier'. The most general unifier is the equation $y = z$. This unifier allows us to unify the expressions without making any arbitrary assumptions.
  - Nunes-Hewitt

Could we just as well have chosen $z = y$?

Unification algorithms

- The job of the unification algorithm is to determine whether two expressions are the same given that some assumptions can be made. The algorithm reports whether the expressions can be made to look identical, and it also reports what assumptions were required (i.e. the unifiers) if it is possible.
  - Nunes-Hewitt

That sounds simple, but how would we automate it?
- Not so simple!
- But suppose the expressions are rendered as lists?
  (Clojure, Lisp, etc.)

Is that always possible?

Comments on students' assignment #1

- Everyone's function got the right answers.
  - That's expected in this advanced course.
- But:
  - One student disregarded the constraint:
    "Forget that the standard library \texttt{pow} function exists!"

  Why did we say that?

  What was the purpose of the exercise?
  - Several showed a very inefficient algorithm (repeated multiplication), even though we had shown one recursive version in class.
  - One used recursion for $x^{n-1}$, even more inefficient!
  - (more) . . .

More notes on assignment #1

- Test driver issues:
  - Dialog with a console user throws away test cases.
  - Better to build cases into the test driver (or get them from a file) so test can be exactly repeated if we need to validate further. ("test automation")
  - In any case \texttt{Please enter} dialog is unnecessary (rather silly) for interaction with yourself.
    \url{http://www.idinews.com/pleaseEnter.html}
  - One student instantiated a Java class with \texttt{new}; no need for \texttt{static} functions.
Finally

- Optimization issue
  - if (n % 2 == 0)
    return power(x n/2) * power(x,n/2);

  Depends on compiler recognizing common repeated expression.

- Better:
  if (n % 2 == 0)
  return (tempo=power(x,n/2)) * tempo;