Recursion and DML Programming

R. Sekar

Recursion

- One of the most fundamental techniques in programming
  - exceptionally versatile
- Closely related to induction:
  - Like induction, has a base case and inductive (recursive) case.
  - Correctness of recursive algorithms proved by induction!
- Example 1: Factorial:

  Base case(s): $fac(0) = 1$

  Recursive case: $fac(n) = n * fac(n-1)$
- Example 2: Fibonacci numbers:

  Base case(s): $fib(0) = 0, fib(1) = 1$

  Recursive case: $fib(n) = fib(n-1) + fib(n-2)$

Uses of Recursion

Recurrences: Typically used in the context of algorithm analysis

- **Base Case:** $T(1) = 1$
- **Recursive Case:** $T(n) = 2T(n/2) + n$

Recursive functions: Used in programming

- **Base Case:** $sum(0) = 0$
- **Recursive Case:** $sum(n) = n + sum(n-1)$

Estimating Runtime of Fibonacci

- For runtime, we don't look for exact numbers
  - Approximation will do, as long as they are not too far off
- Each call of *fib* performs just 3 operations, other than recursive calls
  - So, number of calls is a good estimate of runtime of *fib*
- Start with exact number of calls:

  Base case(s): $N(0) = 0, N(1) = 0$

  Recursive case: $N(n) = N(n-1) + N(n-2) + 2$
- Now, approximate: $N(n) \approx 2N(n-1) + 2$
  - Approximation also removes the need for the second base case of n=1
- Often, such recurrences are solvable to obtain a closed form expression
  - Later on, we will learn how. Meanwhile, DML can help!

Why is Fibonacci So Slow?

- In a call to $fib(n)$, only n distinct Fibonacci numbers are computed.
  - But there are 2^n calls to fib!
    - We are calling $fib(k)$ for the same k many, many times!
- Idea: What if we "remember" Fibonacci numbers we computed already?
  - This avoids repeated calls to the same $fib(k)$
  - This technique is called *dynamic programming* or *memoing*
- Approach: define $ffib(n)$ to return $[fib(0), fib(1), \ldots, fib(n)]$

```
fun ffib(n) = if n = 0
                 then [0]
              else if n = 1
                 then [0, 1]
              else let prev = ffib(n-1)
                 in prev ++ [prev[-1]+prev[-2]]
```

What do these functions do?

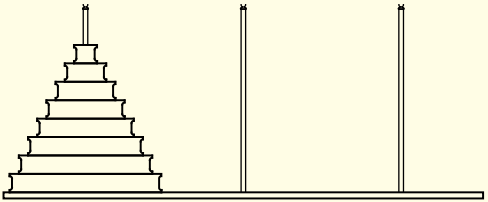
Base case(s): $h(0) = 1$

Recursive case: $h(n) = 2 * h(n/2)$

Base case(s): $m(n) = n - 10$, if $n > 100$

Recursive case: $m(n) = m(m(n + 11))$, otherwise

Tower of Hanoi Problem



Goal: Move all disks from one post to another.

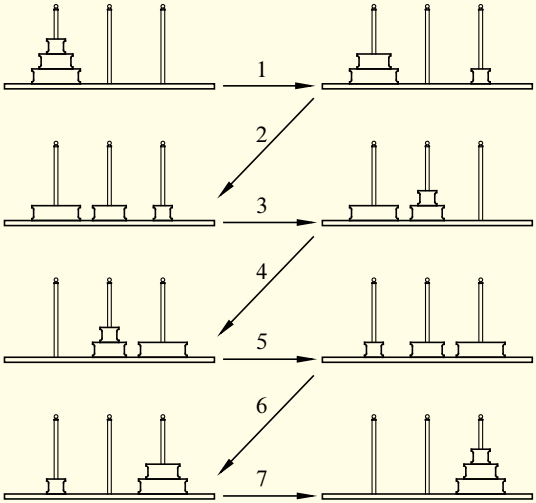
Rules:

- Only the top-most disk can be moved.
- No disk can be placed on a smaller disk.

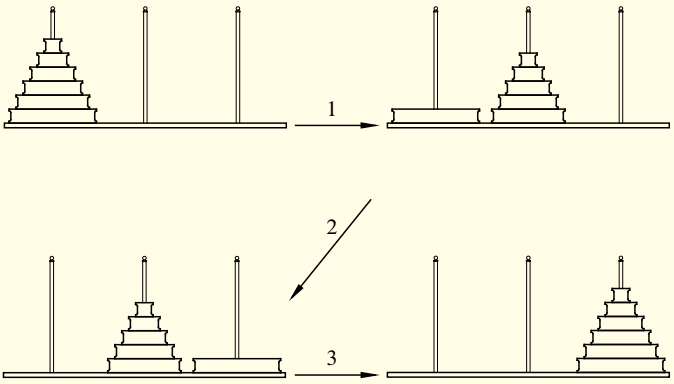
Questions:

- How do you solve the puzzle?
- How many moves will be needed?

Tower of Hanoi Problem: Example with Three Disks



A Recursive Algorithm for Tower of Hanoi Problem



MoveStack$(n, frm, to, spare)$:

Base Case: $n = 0$

- Nothing needs to be done

Recursive Case: $n > 0$

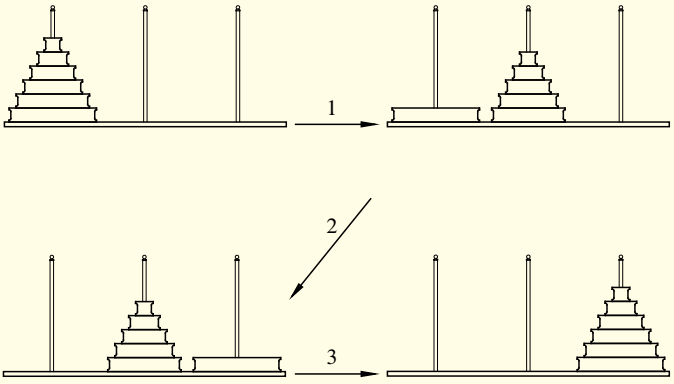
1. *MoveStack*$(n - 1, frm, spare, to)$
2. Move disk n from frm to to
3. *MoveStack*$(n - 1, spare, to, frm)$

DML program for Towers of Hanoi

```
fun MoveStack(n, frm, to, spr) =
  if n = 0
    then []
  else MoveStack(n-1, frm, spr, to) ++
       [("Move disk", n, "from stack", frm, "to", to)] ++
       MoveStack(n-1, spr, to, frm)

fun pp(l) = {print(e) for e in l}
```

A Recurrence for the Runtime of Towers of Hanoi Algorithm



MoveStack$(n, frm, to, spare)$:

Base Case: $n = 0$

$$T(0) = 0$$

Recursive Case: $n > 0$

1. *MoveStack*$(n-1, frm, spare, to)$
2. Move disk n from *frm* to *to*
3. *MoveStack*$(n-1, spare, to, frm)$

$$T(n) = 2T(n-1) + 1$$

More Runtime Recurrence Examples: Exponentiation

$$exp(x, n) = \begin{cases} 1, & \text{if } n = 0 \\ x * exp(x, n-1), & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + 1 \end{aligned}$$

A Divide-and-Conquer Algorithm for Exponentiation

$$fexp(x, n) = \begin{cases} 1, & \text{if } n = 0 \\ fexp(x * x, n/2), & \text{if } n \text{ is even} \\ fexp(x * x, n/2) * x, & \text{if } n \text{ is odd} \end{cases}$$

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/2) + 1 \end{aligned}$$

Prime Numbers

- Simple version:

```
fun is_prime(x) = forall (y in 2..x-1) x % y != 0
fun primes(N) = {x for x in 2..N if is_prime(x)}
```

- Faster version:

```
fun is_prime2(x) = forall (y in 2..sqrt(x)) x % y != 0
fun primes2(N) = {x for x in 2..N if is_prime2(x)}
```

Sieve of Eratosthenes (`sieve`)

1. Start with integer lists L:
   - $L = [2, 3, ..., n]$ for some $n \in \mathbb{N}$.
2. *Invariant:* At all times:
   - L contains no multiples of primes discovered so far.
3. Output the first number m in L as the next prime.
4. Remove multiples of m from L
   - Use a helper function `rm_mult`.
5. Repeat Steps 3 and 4 until L is empty
   - Implemented by calling `sieve` recursively

```
fun rm_mult(L, m) =
  [x for x in L if x % m != 0]

fun sieve(L) =
   if (L = [])
     then []
   else let
          next = L[0]
          newL = rm_mult(L, next)
        in
          sieve(newL) ++ [next]

fun primes3(N) =
  sieve(range(2,N+1))
```

Quicksort (qs): A Divide-and-Conquer Sort Algorithm

1. Pick a *pivot* element at random
2. Divide input list L into 3 parts:
   - Use `qsplit` to divide into elements less, equal to or greater than *pivot*
3. Sort each part
   - Invoke recursion!
4. Concatenate the sorted parts
   - If each part is already sorted, then the concatenation will be sorted too!
5. Recurrence:
   - $T(1) = 1$
   - $T(n) = n + 2T(n/2)$ (On average)

```
fun qsplit(L, pivot) =
  let less = [ x for x in L if x < pivot ]
      more = [ x for x in L if x > pivot ]
        eq = [ x for x in L if x = pivot ]
  in (less, eq, more)

fun qs(L) =
  if len(L) <= 1 then L
  else let p = rand() % (len(L) - 1)
           split = qsplit(L, L[p])
    in qs(split[0]) ++ split[1] ++ qs(split[2])
```

Merge Sort: Another Divide-and-Conquer Sort Algorithm

1. Divide input list L into two halves
   - Use helper function `split`
   - It does not matter how you split.
2. Sort each half
   - Invoke recursion!
3. Merge the sorted halves into one
   - Use helper `merge`
   - Takes time linear in $len(L)$
4. Recurrence:
   - $T(1) = 1$
   - $T(n) = n + 2T(n/2)$

```
fun split(L) = #split into odd, even-numbered elements
  ([L[i] for i in range(1,len(L),2)],
   [L[i] for i in range(0,len(L),2)])

fun ms(L) =
  if len(L) <= 1 then L
  else
    let halves = split(L)
        i = len(halves[0])
        j = len(halves[1])
    in merge(ms(halves[0]), i, ms(halves[1]), j)

fun merge(L1, i, L2, j) =
  if i=0 then [L2[k] for k in 0..j-1]
  else if j=0 then [L1[k] for k in 0..i-1]
  else let m1 = L1[i-1]; m2 = L2[j-1]
       in
       if m1 < m2
         then merge(L1, i, L2, j-1) ++ [m2]
       else merge(L1, i-1, L2, j) ++ [m1]
```

DML Overview

Integers

Integers can range from -2^{62} to $2^{62} - 1$ and use the familiar decimal representation, e.g., `15` for the number fifteen. Integers may also be specified in binary, octal or hexadecimal form using conventions typically used in C, C++ and Python.

Common arithmetic operations on integers use the same operators as in math. The full set of operations supported on integers is shown in the table below:

DML Symbol	Math equivalent	Explanation
`+`, `-`	$+$, $-$	Addition, subtraction, or unary minus
`*`, `/`, `%`	$\times$, $/$, **mod**	Multiplication, division, and modulo
`^`	superscript	exponentiation[1]
`=`, `==`, `!=`	$=$, $\neq$	Equality and disequality
`>`, `<`, `>=`, `<=`	$>$, $<$, $\geq$, $\leq$	Inequality

Reals

- Can be written in:
  - fixed point, e.g., `-1.53`, or
  - scientific format, e.g., `1.53e-27` for 1.53×10^{-27}.
- Support the same set of operations as integers, except for **mod**
- Conversion to integers
  - `round` rounds its argument, e.g., `round(1.49)` is 1, while `round(1.5)` is 2.
  - `ceil` computes the closest integer greater than or equal to the argument, e.g., `ceil(1.49)` is 2.
  - `floor` computes the closest integer less than or equal to the argument, e.g., `floor(1.99)` is 1.

Booleans

- Distinct from integers, and can't be intermixed
- Arise mainly from comparisons
  - but can also define variables and constants of boolean types
- Boolean operators

DML Symbol	Math equivalent	Explanation
<->	$\leftrightarrow$	Logical equivalence ("if and only if")
->	$\rightarrow$	Logical implication
||	$\vee$	Disjunction ("or" operator)
&&	$\wedge$	Conjunction ("and" operator)
!	$\neg$	Logical negation

Strings

String literals can be enclosed in single or double quotes, e.g., `"This is a string"` and `'This is another string'`. Common escape sequences for special characters are supported in a manner compatible with languages such as C and Python. For example, `'a\nb'` is a string that includes a newline character, denoted `\n`. DML will use these escape sequences when it shows the values of string variables and constants, but then such a string is provided as input to the `print` function, it is printed as a newline. Here is a DML session illustrating this.

```
dml> x="a\nb"
x:string = "a\nb"
dml> print(x)
a
b
dml>
```

Strings can be concatenated using the operator `++`, and can be compared with each other using any of the comparison operators discussed above.

```
dml> "Hello" ++ "World"
"HelloWorld"
dml>
```

Sets

- Set construction
  - Sets of consecutive integers: `7..100`
  - Sets with enumerated elements:
    - `{1, 7, 100, 33}`
    - `{"Alex", "Dana", "John", "Jennifer"}`
- DML directly supports most set operations

DML Symbol	Math equivalent	Explanation
`in`	$\in$	Membership check
`union`	$\cup$	Set union
`inter`	$\cap$	Set intersection
`subseteq`	$\subseteq$	Subset operators
`-`	$-$	Set difference
`*`	$\times$	Cartesian product
`pow`	$\wp$	Power Set

Set Builder Notation in Math and DML

- Set of odd numbers ≤ 100
  - Math: $E ::= \{n^2 \mid n \in \mathbb{N} \wedge (n < 100) \wedge (n \bmod 2 = 1)\}$
  - DML: `E = {n^2 for n in 0..99 if (n % 2 = 1)}`
- Set of numbers that satisfy a given condition
  - Math: $E ::= \{n \in \mathbb{N} \mid (n \leq 100) \wedge (n^2 - 41n - 40 > 0)\}$
  - DML: `E = {n for n in 0..100 if (n^2 - 41*n - 40 > 0)}`
- Set of Pythagorean triples ≤ 10
  - Math: $E ::= \{(x, y, z) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mid x^2 + y^2 = z^2\}$
  - DML:
```
{(x,y,z) for x in 1..10 for y in 1..10 for z in 1..10
        if x^2 + y^2 = z^2}
```

DML Lists

- Lists are enclosed in square brackets, and are *ordered*
  - `[1,2]` is different from `[2,1]`
- Use built-in function `range` to construct integer lists:
  - `range(7, 11) = [7, 8, 9, 10]`
    - Unlike sets, the largest element is one less than the higher limit
  - An optional third parameter specifies the step:
    - `range(7, 17, 5) = [7, 12]`
    - `range(7, 18, 5) = [7, 12, 17]`
- List builder notation is similar to set builder notation:

```
[ x for x in {1,2,3,2,1} ]  =  [1,2,3]
[ y for y in range(1,50,2) if y % 7 = 0]  =  [7,14,21,28,35,42,49]
[5*z+1 for z in 1..6]  =  [6, 11, 16, 21, 26, 31]
```

Tuples and Records

- Tuples in math (and DML) correspond to cartesian products of sets:

```
dml> x = (1, 'w', 2.0)
x:(int, string, real) = (1, "w", 2.0)
```

- Ordering matters in cartesian product (and in DML tuples)
  - Elements of a tuple can be accessed using a index

    ```
    dml> x[0]
    1
    ```

  - index starts from 0, can't be a variable or an expression
- *Records* are tuple variants that have field names

```
dml> y = {ival=1, sval='w', rval=2.0}
y:{ival:int, rval:real, sval:string} = {ival=1, rval=2.0, sval="w"}
```

Dictionaries

Dictionaries, also called *associative lists* or *hash tables* in some languages, are a very versatile and efficient data structure frequently used in programs. A dictionary can be viewed as a set of pairs of the form $(key, value)$. Its advantage is that the value associated with a given key can be located efficiently using the index operator. An example dictionary is as follows:

```
x = {'V':'violet', 'I':'indigo', 'B':'blue', 'G':'green',
     'Y':'yellow', 'O':'orange', 'R':'red'}
```

Note that the elements are listed within braces. Within each element, the key precedes the colon operator, while the value follows it. They can both be of arbitrary types. The value associated with a key can be looked up using the index operator, e.g., `x['B']` will return `'blue'`.

While the above notation is convenient for most uses, there can be situations where we want to generate a dictionary from a set of key-value pairs. A predefined function `dict` can do this. For instance, with

```
y = {('V','violet'), ('I','indigo'), ('B','blue'), ('G','green'),
     ('Y','yellow'), ('O','orange'), ('R','red')}
```

`dict(y)` will yield the exact same dictionary as `x`.

Function Definitions and Let Statements

- New functions are introduced using the `fun` keyword:
  - `fun square(x) = x*x`
- `let` statements enable a function definition to be broken up into simpler steps:
  - ```
    fun mypoly(x, y) =
        let t1 = x*x
            t2 = y*y
        in t1 + t2
    ```
- *There should be a newline or a semicolon before the* `in` *keyword*
  - `fun mypoly(x,y) = let t1 = x*x; t2 = y*y in t1 + t2`
    will result in an error message (that can be hard to understand)
    ```

Commonly Used DML Functions

- `abs`: Returns absolute value of a number
- `avg`, `sum`: Returns the average or sum of a set or list of numbers
- `concat`: Takes a list of lists or strings, concatenates them.
  - Note: ++ is the binary concatenation operator on lists/strings
- `len`: Returns the length of a set/list/dict/string
- `insert`: Insert new element into a set/list/string
- `min`, `max`: Returns the min or max element in a set or list.
- `print`: Prints all arguments with spaces between them
- `printr`: Raw print, does not implicitly print spaces or newlines

More Commonly Used DML Functions

- `rand`: Returns a pseudorandom number. Changes on each call.
- `remove`: Remove specified set element or dictionary key. Expensive.
- `removeat`: Remove list element at specified index. Expensive.
- `sortaz`, `sortza`: Input is a set or list, returns a list in sorted order.
- See the DML manual for the full list and additional explanation.

DML Application to CSE 150 Topics

DML and Summations and Series

- Numeric computation of any series or sum
  - Check if a closed form you derived produces the correct answer.
- Use numeric computation to identify a pattern in the summation
  - The pattern can suggest a closed form solution.
  - To be sure, you will need to prove the correctness of your guess using mathematical induction.
- Where DML does not help
  - Directly giving you those closed form solutions!
  - Perform symbolic operations in algebra

DML and Sets

- All set operations in math have a direct equivalent in DML
- What DML can't do:
  - Work with infinite sets
  - Draw Venn diagrams
  - Directly help with proofs

DML and Propositional Formulas

- DML can do pretty much anything on propositional formulas
  - Check implications for validity
  - Construct truth tables
  - Satisfiability of propositional formulas
  - Validity of propositional formulas
- Main limitations
  - Formulas have to be small, less than 30 or so variables.
  - It can check if a formula P is equivalent to another simpler formula Q, but it cannot produce Q from P
    - Again, no symbolic or algebraic simplification capability

DML and Predicate Logic Formulas

- Can check satisfiability of predicated logic formulas over finite (and relatively small) sets.
- Can give you counter examples when formulas are not true
  - Understand quantified formulas
  - Help you to translate from English to Logic or vice-versa
  - Debug your formulas
  - Use quantified formulas in computations

DML and Functions and Relations

- Define binary relations $R : X \to Y$ by enumerating the subset of $X \times Y$ included in R
  - Functions can also be defined as dictionaries
- Write simple functions to
  - Compute *support*, *range*, *image*, ...
  - Check for properties: *function*, *total*, *partial*, *injective*, *surjective*, *bijective*
  - Compute the *composition* of two relations or *inverse* of a relation
  - Check for properties such as *reflexivity*, *symmetry*, *transitiviity*, *partial order*, *linear order* and *equivalence relation*
  - Compute *reflexive*, *symmetric* or *transitive* closures.
  - Compute *walks*, *paths*, and *cycles* in graphs.

Counting and Probability

- Like summations, numerically solve counting and probability problems
  - Brute-force enumeration of sets or sequences, apply `len` on enumeration to count.
  - Very helpful for checking formulas
    - Brute-force nature of enumeration minimizes chances of error
- Brute-force enumeration can solve counting problems that require a variety of techniques
  - product rule, disjoint or overlapping set union rule, division rule, complement rule, permutation and combination rules, binary strings, birthday problem, and pigeonhole principle
- Use numeric computation to identify a pattern in counting results
  - The pattern can suggest a closed form solution.
- Where DML does not help
  - Directly giving you those closed form solutions!

Where DML does not help

- Dealing with infinite sets
  - Including counting problems on them
- Proofs
  - You can often use DML to check if a formula is true
    - If the formula involves finite domains
    - Or, by checking it on finite domains to get more insight
- Computing closed-form, symbolic solutions.