MIPS Assembly: Recursion, factorial, fibonacci

CptS 260
Introduction to Computer Architecture
Week 2.3
Wed 2014/06/18

Recursion: Guidelines

- **Function calls itself**
  - **Not a leaf function** *([P&H14] §2.8 p.100)*
    - **Must have a stack frame (for $ra)**
  - **1 call to self ≈ a loop** *~ O(n)*
    - **Must have base cases**
    - **Must make the problem "smaller"**
    ("smaller" ≡ "closer to base case")
  - **2+ calls ≈ worse than loop!** *~ $O(c^n)$*

Handle Base Cases First
Get Your Stack Frame Correct

```
void LL( a, … )
{
    …
    LL( a – 1, … );
    …
}
```

```
void TT( a )
{
    …
    TT(a–1) + TT(a/2);
    …
}
```

MIPS Example: factorial

1 argument: numeric
1 base case/call

```
# int factorial($a0 :  int n)       // returns n! if n > 0, or 1 if n ≤ 0
factorial:
# base case
     bgt     $a0, 0, recur        # if (n <= 0)

     li      $v0, 1               #           … 1;
     jr      $ra                  #     return …
recur:                            # // else
     addi    $sp, $sp, –[ ]       # stack push
     ???
     sw      $ra, 0($sp)          # 0: |_ ra _|

     addi    $a0, $a0, –1         #                  (n – 1)
     jal     factorial            #     v0 = factorial(n – 1) …
     lw      $t0, 4($sp)          #                          … n;
     mul     $v0, $v0, $t0        #                        … *

     lw      $ra, 0($sp)
     addi    $sp, $sp, [ ]        # stack pop
     jr      $ra                  #     return …
```

MIPS Example: factorial

1 argument: numeric
1 base case/call

```
# int factorial($a0 :  int n)        // returns n! if n > 0, or 1 if n ≤ 0
factorial:
# base case
      bgt     $a0, 0, recur       # if (n <= 0)


      li      $v0, 1              #          … 1;
      jr      $ra                 #    return …
recur:                            # // else
      addi    $sp, $sp, –8        # stack push
      sw      $a0, 4($sp)         # 4: |_ n _|
      sw      $ra, 0($sp)         # 0: |_ ra _|

      addi    $a0, $a0, –1        #                    (n – 1)
      jal     factorial           #    v0 = factorial(n – 1) …
      lw      $t0, 4($sp)         #                             … n;
      mul     $v0, $v0, $t0       #                           … *

      lw      $ra, 0($sp)
      addi    $sp, $sp,  8        # stack pop
      jr      $ra                 #    return …
```

Preserving `n`: Way 1. Explicit Restore

```
# int factorial($a0 :  int n)        // returns n! if n > 0, or 1 if n ≤ 0
factorial:
# base case
        bgt     $a0, 0, recur        # if (n <= 0)

        li      $v0, 1               #           … 1;
        jr      $ra                  #     return …
recur:                               # // else
        addi    $sp, $sp, –8         # stack push
        sw      $a0, 4($sp)          # 4: |_ n _|
        sw      $ra, 0($sp)          # 0: |_ ra _|

        addi    $a0, $a0, –1         #                  (n – 1)
        jal     factorial            #     v0 = factorial(n – 1) …
        lw      $t0, 4($sp)          #                          … n;
        mul     $v0, $v0, $t0        #                        … *

        lw      $ra, 0($sp)
        addi    $sp, $sp,  8         # stack pop
        jr      $ra                  #     return …
```

👍 **Way 1**
$a0 does not persist

save here

n

restore here

don't need to restore here

👍 can restore to any temp

👎 *can't trust $a0*

Preserving `n`: Way 2. Make it Persist

```
# int factorial($a0 :  int n)       // returns n! if n > 0, or 1 if n ≤ 0
factorial:
# base case
      bgt       $a0, 0, recur       # if (n <= 0)

      li        $v0, 1              #          … 1;
      jr        $ra                 #     return …
recur:                              # // else
      addi      $sp, $sp, –8        # stack push
      sw        $a0, 4($sp)         # 4: |_  n _|
      sw        $ra, 0($sp)         # 0: |_ ra _|

      addi      $a0, $a0, –1        #                  (n – 1)
      jal       factorial           #     v0 = factorial(n – 1) …
      addi      $a0, $a0,  1        #                        … n;      (again)
      mul       $v0, $v0, $a0       #                      … *

      lw        $ra, 0($sp)
      lw        $a0, 4($sp)         #
      addi      $sp, $sp,  8        # stack pop
      jr        $ra                 #     return …
```

☝ **Way 2**
$a0 does persist

push here

n

pop here

☝ can trust $a0!

✋ must restore to $a0

Fibonacci Sequence

- **[Leonardo of Pisa] Rabbit pairs in month *n***
  - **New rabbits start breeding at 2 months**
  - ***(Rabbits are immortal?)***

Binet's formula

$$F_n = \frac{\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n}{2^n \sqrt{5}}.$$

***F*(1) = 1**
***F*(2) = 1**
F(n) = F(n-1) + F(n-2)

- **Growth rate is: exponential!**
  - ***binary tree of recursive calls …***

$$F_n = \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right]$$

MIPS Example: Fibonacci

1 argument: numeric
2 base cases/calls

```
F:      # int F($a0 :  int n)   // F(0) = 0;   F(1) = 1;     F(n) = F(n–1) + F(n–2)
        bgt   $a0, 0, base1     # if (n <= 0)
        li    $v0, 0
        jr    $ra               #     return 0;
base1:  bgt   $a0, 1, recur     # if (n == 1)

        li    $v0, 1
        jr    $ra               #     return 1;
recur:  # // else
        addi  $sp, $sp, –12     # stack push
                                # 8: |_ F(n–1) _|
        sw    $a0, 4($sp)       # 4: |_    n     _|
        sw    $ra, 0($sp)       # 0: |_   ra     _|

        addi  $a0, $a0, –1      #            (n – 1)
        jal   F                 #     v0 = F(n – 1)
        sw    $v0, 8($sp)

        addi  $a0, $a0, –1      #            (n – 2)
        jal   F                 #     v0 = F(n – 2) ..
        lw    $t0, 8($sp)
        add   $v0, $v0, $t0     #                 … + F(n – 1)
```

in series!

base cases first

pre-allocate space

save it

F(n–1)

restore it

```
        lw    $a0, 4($sp)
        lw    $ra, 0($sp)
        addi  $sp, $sp,  12
        jr    $ra
```

Families of Recursive Functions

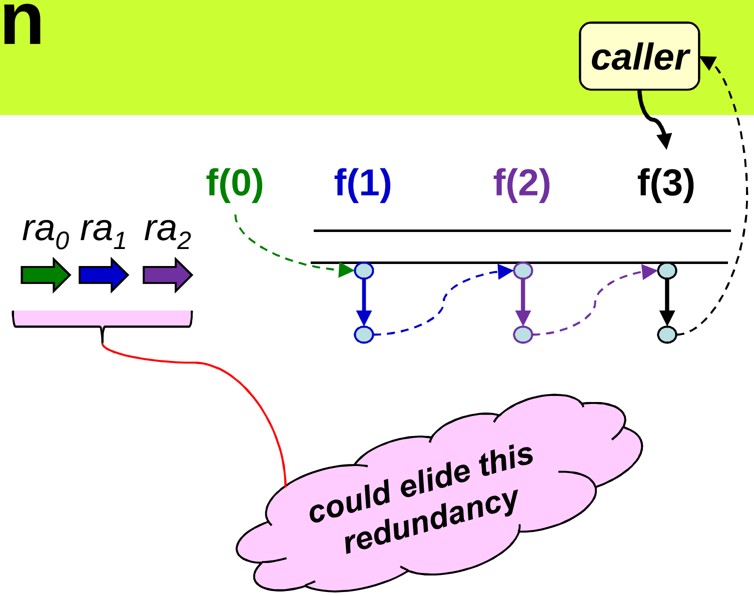
- Parameterize by (arguments, base cases & calls) = *(a, bc)*: *type*

- *(a, bc)* = (1, 1): numeric
  - sum Σ
  - factorial, product Π
- *(a, bc)* = (1, 2): numeric
  - Fibonacci; mapping $\mathbb{N} \to \mathbb{Z}$
  - "2D" Fib-like: $\mathbb{N}^2 \to \mathbb{Z}$
- *(a, bc)* = (2, 2): numeric
  - Ackermann's function

    >> O(2^n)!!

- *(a, bc)* = (1, 1): pointer
  - strlen
  - *any single loop over array*
- *(a, bc)* = (2, 1*): pointer
  - (quick)sort
  - *divide-and-conquer an array*

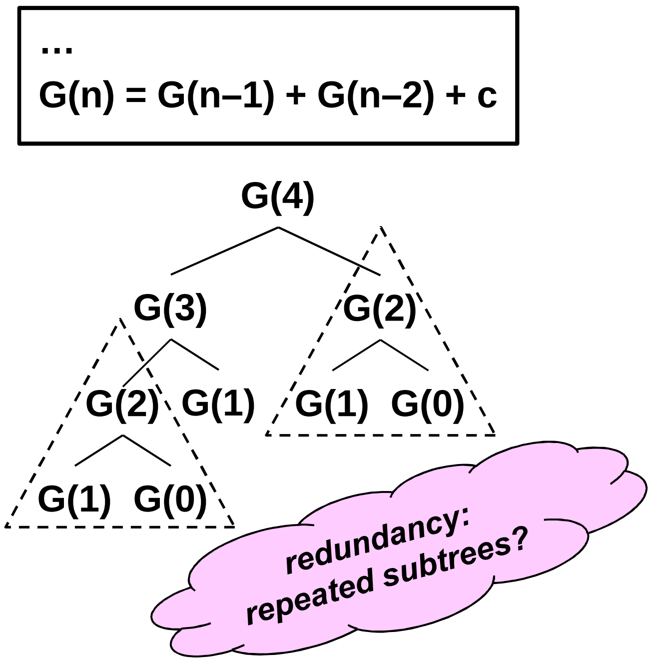
Perspectives on Recursion



- Singly-recursive can be a loop

  Tail-recursive + *stack frame elision*
  ➔ identical to a loop

- Doubly-recursive can't be a loop



- Binary tree of calls (➔ or n-ary)
  - Stack at any instant is one path from root to a leaf
  - Stack as a topological line that sweeps across the call tree

MIPS Stack Trace: factorial

```
# int factorial($a0 :  int n)       // // returns n! else 1
factorial:
# base case
     bgt     $a0, 0, recur       # if (n <= 0)


     li      $v0, 1              #          … 1;
     jr      $ra                 #    return …
recur:                           # // else
     addi    $sp, $sp, –8        # stack push
     sw      $a0, 4($sp)         # 4: |_ n _|
     sw      $ra, 0($sp)         # 0: |_ ra _|

     addi    $a0, $a0, –1        #            (n – 1)
     jal     factorial           #    v0 = f(n – 1) …
     lw      $t0, 4($sp)         #                    … n;
     mul     $v0, $v0, $t0       #                  … *

     lw      $ra, 0($sp)
     addi    $sp, $sp,   8       # stack pop
     jr      $ra                 #    return …
```

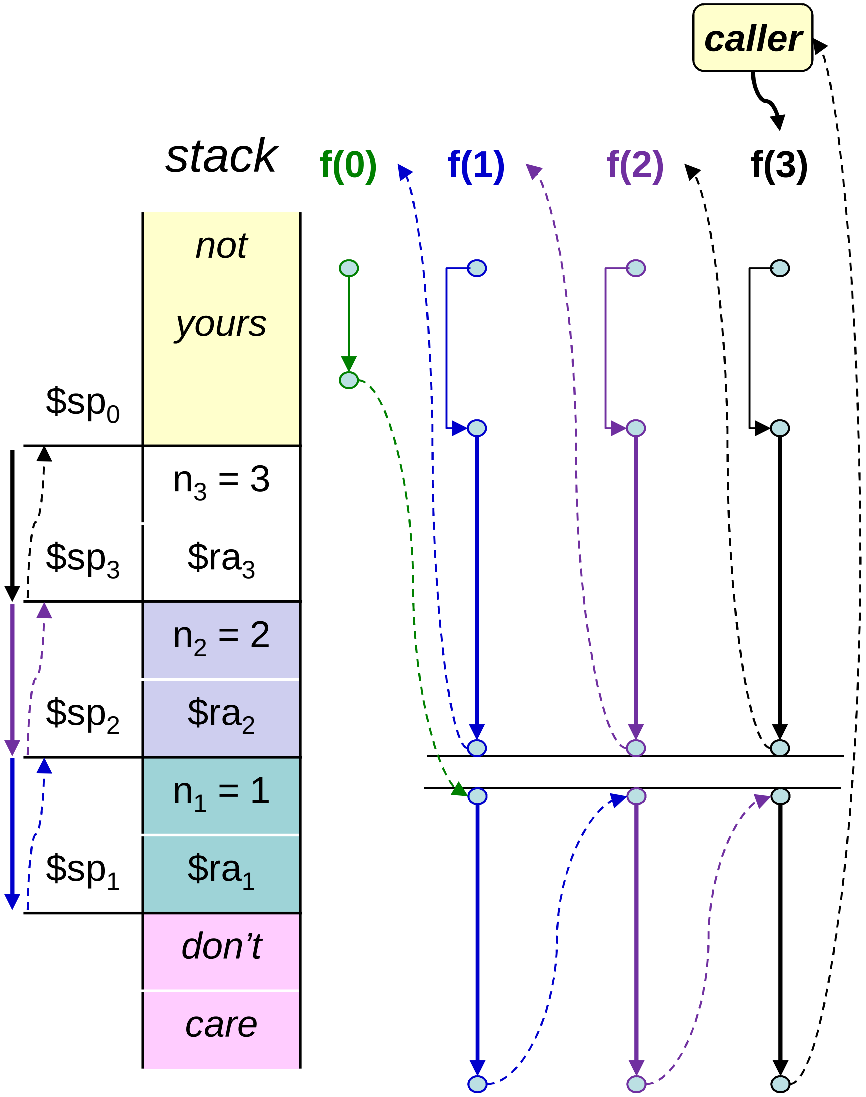
MIPS Stack Trace: factorial

```
# int factorial($a0 :  int n)        // // returns n! else 1
factorial:
# base case
     bgt       $a0, 0, recur         # if (n <= 0)

     li        $v0, 1                #          … 1;
     jr        $ra                   #     return …
recur:                               # // else
     addi      $sp, $sp, –8          # stack push
     sw        $a0, 4($sp)           # 4: |_ n _|
     sw        $ra, 0($sp)           # 0: |_ ra _|

     addi      $a0, $a0, –1          #          (n – 1)
     jal       factorial             #     v0 = f(n – 1) …
     lw        $t0, 4($sp)           #                  … n;
     mul       $v0, $v0, $t0         #                  … *

     lw        $ra, 0($sp)
     addi      $sp, $sp,   8         # stack pop
     jr        $ra                   #     return …
```



MIPS Stack Trace: factorial

- Recursive `jal` as a gap (discontinuity)
  - "Two-phase" processing
  - Contention for registers
  - ➔ "bottommost call wins"?!
    (or equally obscure)
- Similarities to:
  - message-passing / networked:
    send, …, recv
  - modal windows / multithreads:
    block, …, resume