Using Recurrence Relations to Evaluate the Running Time of Recursive Programs

by Peter Strazdins, Computer Systems Group

Overview:

- review from lectures 1–3:
  - recursive definition, recursive functions
  - revisit induction proof with recursive summation definition
  - relationship between induction, recursion and recurrences
- (review) big-O notation and running time for iterative programs
  - big-O as an *abstraction*
- using recurrence relations and induction for the running time of recursive:
  - logarithm, factorial, Fibonacci
  - list length and split, mergesort

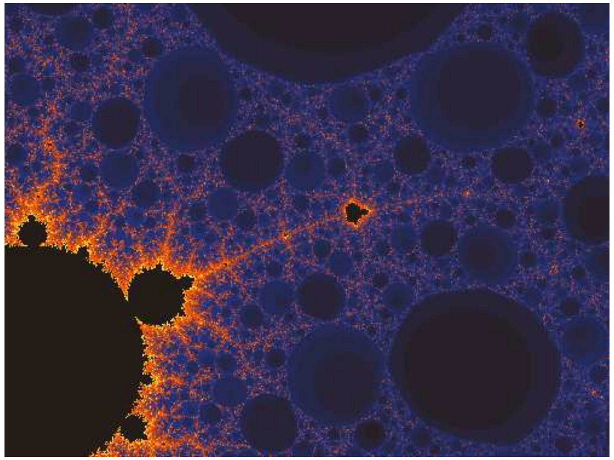
Recursive Definition and Recursive Functions

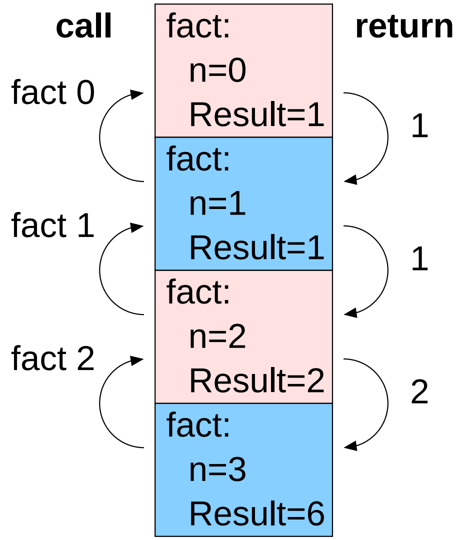
- recursion: (now rare or obsolete, 1616) a backward movement, return

  – The Shorter Oxford English Dictionary
- a recursive definition has one or more ‘base’ rules and one or more ‘inductive’ rules (lectures 1–3 p18)
- a recursive function is one that uses itself in its definition (i.e. it calls itself; see lectures 1–3 p21)
  - (to be well defined) it definition must have at least 2 parts Q: what kind?
- e.g. factorial function

```
fact 0 = 1
fact n = n * fact (n-1)
```

- how is recursion implemented on a computer? notion of a stack





Induction Proof with a Recursive Definition of Summation

- we can define the standard summation recursively:

$$\Sigma_{i=0}^{n} f(i) = \begin{cases} f(0) & \text{if } n = 0 \quad \text{-(S1)} \\ f(n) + (\Sigma_{i=0}^{n-1} f(i)) & \text{, otherwise} \quad \text{-(S2)} \end{cases}$$

- in the proof of $\Sigma_{i=0}^{n} 2^i = 2^{n+1} - 1$ (lectures 1–3 p4):

Base Case: show $S(0)$: $\Sigma_{i=0}^{0} 2^0 = 2^1 - 1$

$$\begin{aligned} \Sigma_{i=0}^{0} 2^i &= 2^0 && \text{, by -S(1)} \\ &= 2^1 - 1 \end{aligned}$$

Inductive Case: assuming $S(n)$: $\Sigma_{i=0}^{n} 2^i = 2^{n+1} - 1$, show $S(n+1)$:

$\Sigma_{i=0}^{n+1} 2^i = 2^{n+2} - 1$

$$\begin{aligned} \Sigma_{i=0}^{n+1} 2^i &= 2^{n+1} + (\Sigma_{i=0}^{n} 2^i) && \text{, by -S(2)} \\ &= 2^{n+1} + (2^{n+1} - 1) && \text{, by the Induction Hypothesis} \\ &= 2^{n+2} - 1 \end{aligned}$$

- i.e. we have used the definition of summation to formally make the step in our inductive proof

Relationship between Induction, Recursion and Recurrences

- a recurrence relation is simply a (mathematical) function (or relation) defined in terms of itself
  - e.g. $f(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + f(n-1) & \text{, otherwise} \end{cases}$
  - also, our definition of summation
  - not all formulations yield meaningful definitions, e.g. $f(n) = f(n)+1$, $f(n) = f(2n)+1$
- recurrence relations on the natural numbers ($\mathbb{N}$) can be used to characterized running times of programs with some (possibly derived) numerical input parameter (n)
- induction shares the same structure, but with a proposition instead: from $S(0)$, and $S(n) \Rightarrow S(n+1)$ we establish $S(n)$ for all $n \in \mathbb{N}$
  - note: we could equivalently define $f(n)$ above as $f(0) = 1$, $f(n+1) = 1 + f(n)$

Big-O Notation and Running Time for Programs

- recall $T(n) \in O(f(n))$ means $\exists$ constants c and $n_0 > 0$ s.t. $\forall\, n > n_0$: $T(n) \le c f(n)$
  - e.g. $T(n) = 3n+5$, $f(n) = n$, we can choose $c = 4$ and $n_0 = 3$
    for $n > 3$, $4n > 2n+2*3 > 2n+5$
- let $T(n)$ represent the running time of a program
- is the number of statements executed a realistic estimate of actual running time?

```
s = 0;  t  = 0;
for (i=0; i < n; i++) {
  s = s + sqrt(a[i]);  t = t + s;
}
printf("t=", sqrt(t));
```

  Are all of the executed $2n+3$ statements equal? We can at least say $T(n) \in O(n)$
- principles of deriving (an upper bound estimate to) $T(n)$
  - composition: if a program has (sequential) parts `A; B`, we can write
    $T(n) = T_A(n) + T_B(n)$
  - abstraction: we approximate any term (not including $T(\ldots)$ by its (simplest) big-O order (e.g. 3 becomes 1, $2n+5$ becomes n etc)
- the big-O notation provides an *abstraction* from both the (structural) complexities in computer programs, and the (complex) details of modern computer architectures

Example: the Factorial Program

- the classic example:

```
fact 0 = 1
fact n = n * fact (n-1)
```

- for Haskell programs, we take elementary operations as having a running time of 1

  e.g. $*$, $-$, access constant/variable, apply function definition, `:`, take head/tail of list

- then the execution time follows the recurrence relation:

$$\begin{array}{lcll} T(0) & = & 1 & \text{-(T1)} \\ T(n) & = & 1+T(n-1) & \text{-(T2)} \end{array}$$

- we can prove by induction $S(n)$: $T(n) = n+1$, and hence $T(n) \in O(n)$
  - Base Case: show: $S(0)$: $T(0) = 1$

    follows immediately from -(T1)
  - Inductive Case: given $S(n)$, show $S(n+1)$: $T(n+1) = n+2$:

$$\begin{array}{lcll} T(n+1) & = & 1+T(n) & \text{, by -(T2)} \\ & = & 1+n+1 & \text{, by the Induction Hypothesis} \\ & = & n+2 & \end{array}$$

Example: the Logarithm Program

- ```
  log2 1 = 0
  log2 n = 1 + log2 (n `div` 2)
  ```
- here we can similarly derive:

  $$\begin{array}{lcll} T(1) & = & 1 & \text{-(T1)} \\ T(n) & = & 1 + T(n/2) & \text{-(T2)} \end{array}$$

  (this program is evidently faster!)
- using the example $T(8) = 1 + T(4) = 2 + T(2) = 3 + T(1)$, we conjecture $S(n)$:
  $T(n) = 1 + \log_2(n)$, i.e. $T(n) \in O(\log_2(n))$
- proof by induction is similar to before, except we restrict n to powers of 2:
  - Base Case: prove $S(1)$: $T(1) = 1 + \log_2(1)$
    $T(1) = 1 = 1 + 0 = 1 + \log_2(1)$
  - Inductive Case: given $S(n)$, show $S(2n)$: $T(2n) = 1 + \log_2(2n)$

    $$\begin{array}{lcll} T(2n) & = & 1 + T(n) & \text{, by -(T2)} \\ & = & 1 + 1 + \log_2(n) & \text{, by the Induction Hypothesis} \\ & = & 1 + \log_2(2n) & \text{, using } \log_2(2x) = 1 + \log_2(x) \end{array}$$
- is this a valid form of induction? Why (not)?
- how do we show $T(n) = 1 + \log_2(n)$ for all n?
  ```

Example: the Fibonacci Program

- direct Haskell implementation of the Fibonacci recurrence:

```
fib 0 = 1
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

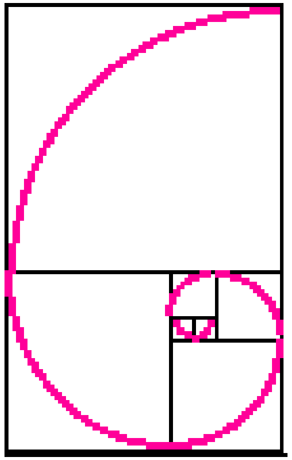
- as before, we can similarly derive:

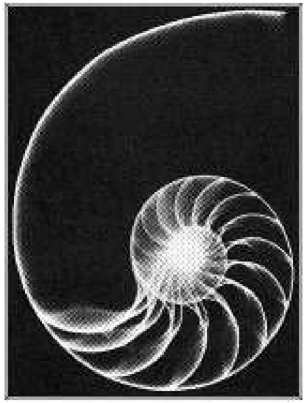
$$T(0) = T(1) = 1 \quad \text{-(T1)}$$
$$T(n) = 1 + T(n-1) + T(n-2) \quad \text{-(T2)}$$

- this time, we expect an exponential running time:

$$T(8) = 1 + T(7) + T(6) = 2 + 2T(6) + T(5) = 4 + 3T(5) + 2T(4)$$

and conjecture $S(n)$: $T(n) \le 2^n$, which will mean that $T(n) = O(2^n)$





  - Base Case: show $S(0)$: $T(0) \le 2^0$ (also $S(1)$)

    Follows directly from -(T1).

  - Inductive Case: given $S(n)$ (*and* $S(n-1)$ and $n > 0$), show

    $S(n+1)$: $T(n) \le 2^{n+1}$

$$\begin{aligned} T(n+1) &= 1 + T(n) + T(n-1) && \text{, by -(T2), as } n+1 > 1 \\ &\le 1 + 2^n + 2^{n-1} && \text{, by the Induction Hypothesis} \\ &\le 2^n + 2^n && \text{, as } 2^n \ge 1 + 2^{n-1} \text{ for } n > 0 \\ &= 2^{n+1} \end{aligned}$$

  - note use of Generalized Principle of Induction, with inequalities

Running Times of Programs Operating on Lists

- lists are a recursive data structure; can model running time on n =`length xs`
- e.g. the length of a list function itself can be defined as

```
length [] = 0
length (x:xs) = 1 + length xs
```

- we can similarly derive the running time $T(n)$ for this program:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= 1 + T(n-1) \end{aligned}$$

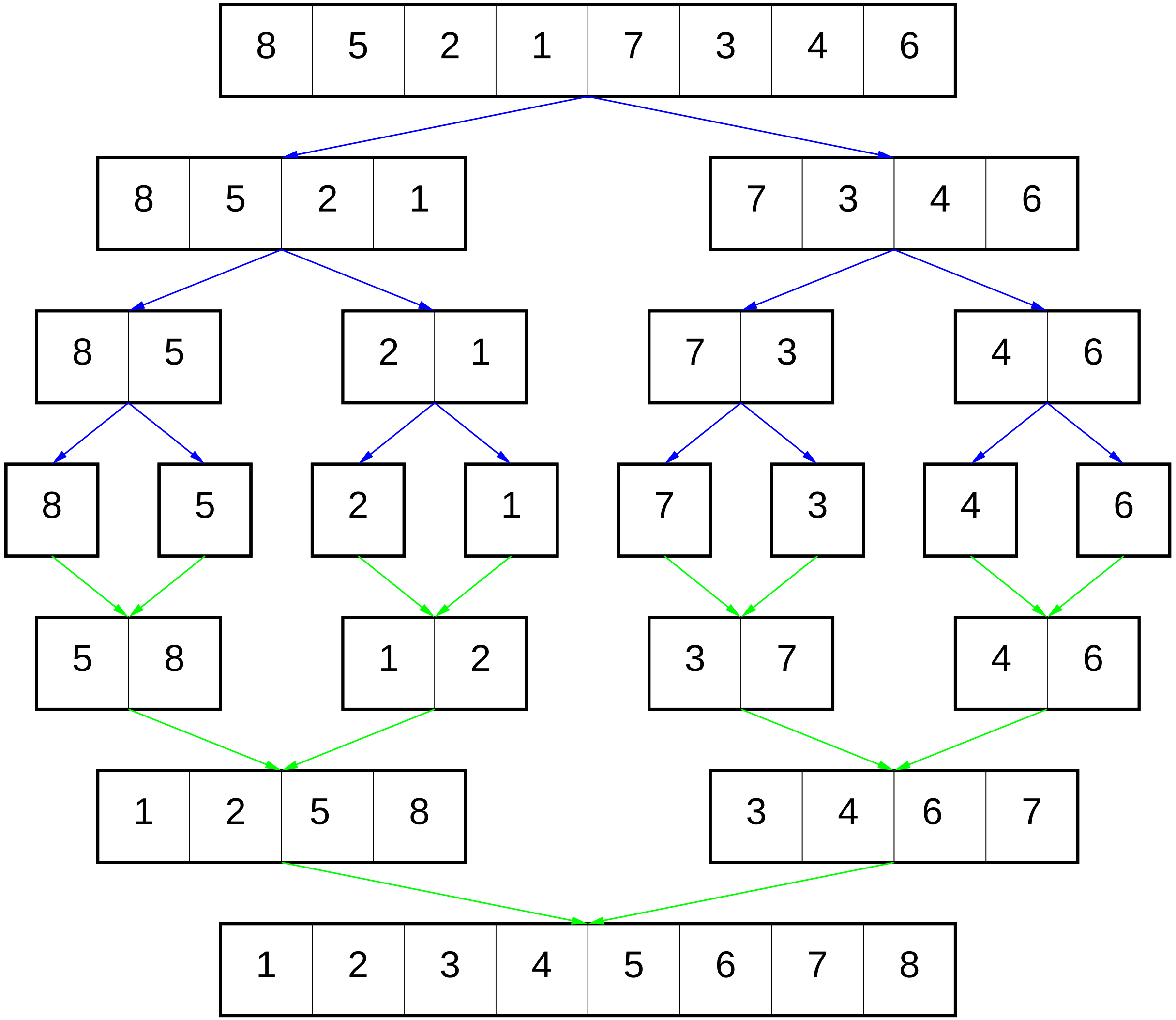
  which we can solve as $T(n) = n$
- we can 'split' a list into sublist of odd and even elements:

```
split [] = ([], [])
split [x] = ([x], [])
split (x1:x2:xs) = (x1:x1s, x2:x2s)
  where (x1s,x2s) = split xs
```

- noting that `length (x1:x2:xs)` $-$ 2 = `length xs`, we can derive:

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ T(n) &= 1 + T(n-2) \end{aligned}$$

The Mergesort: An Efficient Sorting Algorithm



The Mergesort in Haskell

- note: the use `split` 'shuffles' items before merge begins

```
mergesort []     = []
mergesort [x]    = [x]
mergesort (x:xs) = merge (mergesort l) (mergesort r)
  where (l, r) = split x:xs
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x<=y        = x : merge xs (y:ys)
  | otherwise   = y : merge (x:xs) ys
```

- assuming the execution time for `split` and `merge` are $T_S(n) = T_M(n) = n/2$:

$$
\begin{aligned}
T(0) &= T(1) = 1 && \text{-(T1)} \\
T(n) &= T_S(n) + 2T(n/2) + T_M(n) && \text{(we ignore } O(1) \text{ terms)} \\
&= n + 2T(n/2) && \text{-(T2)}
\end{aligned}
$$

- we conjecture that $T(n) \in O(n \log_2(n))$

  (the induction proof of this is not trivial!)

Review: Recursion Recurrences and Running Time

- induction is a valuable tool for solving the recurrences
  - why is form proving for powers of 2 valid?
  - sometimes need the Generalized Principle
  - forming the correct hypothesis takes experience (or attempting a proof)
  - using the big-O *abstraction* simplifies the proofs (validity?)
- review of algorithms and their (upper bounds) on running times:

example:	recurrence:	running time:
logarithm	$T(n) = 1 + T(n/2)$	$O(\log_2(n))$
factorial	$T(n) = 1 + T(n-1)$	$O(n)$
mergesort	$T(n) = n + 2T(n/2)$	$O(n\log_2(n))$
Fibonacci	$T(n) = 1 + 2T(n-1)$	$O(2^n)$

  (the Master Theorem gives a general relationship (COMP3600))
- why is $T(n) \in O(f(n))$ useful? Where is it not useful?
  - if a program with $T(n) = a2^n$ takes 10^3s for $n = 32$, what will it take for $n = 64$?
- recursion (and recurrence relations in the more general sense) often give a simple but powerful of algorithms (and many natural phenomena)

Addendum: Proof of $T(n) \in O(n \log_2(n))$ for the Mergesort

- the non-trivial part is actually in finding a workable inductive hypothesis . . .
  - initial guess $T(n) = n\log_2(n)$ works for the inductive but not the base case
  - attempted fix $T(n) = n\log_2(n) + 1$ no longer works for the inductive case!
  - strategy: try to do proof on $T(n) = n\log_2(n) + f(n)$ and see what properties of f are needed, yielding:
    - $f(1) = 1$ and $2f(n) = f(2n)$ – satisfiable by the humble identity function!
- as for the `log2` example, the induction proof is limited to powers of 2, with the hypothesis: $S(n)$: $T(n) = n\log_2(n) + n$
  - Base Case: prove $S(1)$: $T(1) = 1\log_2(1) + 1$

    $T(1) = 1 = 0 + 1 = 1 * 0 + 1 = 1\log_2(1) + 1$
  - Inductive Case: given $S(n)$, show $S(2n)$:

$$
\begin{aligned}
T(2n) &= 2n + 2T(n) && \text{, by -(T2)} \\
&= 2n + 2(n\log_2(n) + n) && \text{, by the Inductive Hypothesis} \\
&= 2n(1 + \log_2(n)) + 2n && \text{, rearranging terms} \\
&= 2n\log_2(2n) + 2n && \text{, using } \log_2(2x) = 1 + \log_2(x)
\end{aligned}
$$

- compare this with the proof in Aho & Ullman Ch 3.10!!