Using Recurrence Relations to Evaluate the Running Time of Recursive Programs

by Peter Strazdins, Computer Systems Group

Overview:

bullet review from lectures 1–3:
  - recursive definition, recursive functions
  - revisit induction proof with recursive summation definition
  - relationship between induction, recursion and recurrences
bullet (review) big-O notation and running time for iterative programs
  - big-O as an abstraction
bullet 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

● e.g. factorial function

\[
\begin{align*}
\text{fact } 0 &= 1 \\
\text{fact } n &= n \times \text{fact } (n-1)
\end{align*}
\]

● 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:
  \[ \sum_{i=0}^{n} f(i) = \begin{cases} 
  f(0) & \text{if } n = 0 \quad \text{-(S1)} \\
  f(n) + \left( \sum_{i=0}^{n-1} f(i) \right) & , \text{otherwise} \quad \text{-(S2)}
  \end{cases} \]

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

  Base Case: show \( S(0) \): \( \sum_{i=0}^{0} 2^i = 2^1 - 1 \)
  \[ \sum_{i=0}^{0} 2^i = 2^0 , \text{ by -(S1)} \]
  \[ = 2^1 - 1 \]

  Inductive Case: assuming \( S(n) \): \( \sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \), show \( S(n+1) \):
  \[ \sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1 \]
  \[ \sum_{i=0}^{n+1} 2^i = 2^{n+1} + \left( \sum_{i=0}^{n} 2^i \right) , \text{ by -(S2)} \]
  \[ = 2^{n+1} + (2^{n+1} - 1) , \text{ by the Induction Hypothesis} \]
  \[ = 2^{n+2} - 1 \]

- 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 characterize 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. \( \forall n > n_0: T(n) \leq cf(n) \)
  - e.g. \( T(n) = 3n + 5, f(n) = n \), we can choose \( c = 4 \) and \( n_0 = 3 \) for \( n > 3, 4n > 2n + 2 \times 3 > 2n + 5 \)

- let \( T(n) \) represent the running time of a program

- is the number of statements executed a realistic estimate of actual running time?

```c
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:

\[
\text{fact } 0 = 1 \\
\text{fact } n = n \times \text{fact } (n-1)
\]

- 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:

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

- 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 \(S(n+1): T(n+1) = n + 2\):
    \[
    T(n+1) = 1 + T(n) \quad \text{, by -(T2)} \\
    = 1 + n + 1 \quad \text{, by the Induction Hypothesis} \\
    = n + 2
    \]
Example: the Logarithm Program

- \( \log_2 1 = 0 \)
- \( \log_2 n = 1 + \log_2 (n \div 2) \)

- Here we can similarly derive:
  - \( T(1) = 1 \) -(T1)
  - \( T(n) = 1 + T(n/2) \) -(T2)

- (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:
  - \( S(1) \): \( T(1) = 1 + \log_2(1) \)
    - \( T(1) = 1 = 1 + 0 = 1 + \log_2(1) \)
  - \( S(2n) \): \( T(2n) = 1 + \log_2(2n) \)
    - \( T(2n) = 1 + T(n) \), by -(T2)
    - \( = 1 + 1 + \log_2(n) \), by the Induction Hypothesis
    - \( = 1 + \log_2(2n) \), using \( \log_2(2x) = 1 + \log_2(x) \)

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

- as before, we can similarly derive:
  \[
  T(0) = T(1) = 1 - (T1)
  \]
  \[
  T(n) = 1 + T(n-1) + T(n-2) - (T2)
  \]

- 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 \( S(n): T(n) \leq 2^n \), which will mean that \( T(n) = O(2^n) \)

  - Base Case: show \( S(0): T(0) \leq 2^0 \) (also \( S(1) \))
    Follows directly from -(T1).
  - Inductive Case: given \( S(n) \) (and \( S(n-1) \) and \( n > 0 \)), show
    \[
    S(n+1): T(n) \leq 2^{n+1}
    \]
    \[
    T(n+1) = 1 + T(n) + T(n-1) , \text{ by -(T2), as } n+1 > 1
    \]
    \[
    \leq 1 + 2^n + 2^{n-1} , \text{ by the Induction Hypothesis}
    \]
    \[
    \leq 2^n + 2^n , \text{ as } 2^n \geq 1 + 2^{n-1} \text{ for } n > 0
    \]
    \[
    = 2^{n+1}
    \]

  - 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 \( n = \text{length } \text{xs} \)
- e.g. the length of a list function itself can be defined as
  
  \[
  \text{length } [\vphantom{a}] = 0 \\
  \text{length } (x:xs) = 1 + \text{length } \text{xs}
  \]

- we can similarly derive the running time \( T(n) \) for this program:
  \[
  T(0) = 1 \\
  T(n) = 1 + T(n - 1)
  \]

  which we can solve as \( T(n) = n \)

- we can ‘split’ a list into sublist of odd and even elements:
  
  \[
  \text{split } [\vphantom{a}] = ([], []) \\
  \text{split } [x] = ([x], []) \\
  \text{split } (x1:x2:xs) = (x1:x1s, x2:x2s) \\
  \text{where } (x1s,x2s) = \text{split } \text{xs}
  \]

- noting that \( \text{length } (x1:x2:xs) = 2 = \text{length } \text{xs} \), we can derive:
  \[
  T(0) = 1 \\
  T(1) = 1 \\
  T(n) = 1 + T(n - 2)
  \]
The Mergesort: An Efficient Sorting Algorithm
The Mergesort in Haskell

- **note:** the use `split` ‘shuffles’ items before merge begins

```haskell
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 $T_S(n) = T_M(n) = n/2$:

\[
T(0) = T(1) = 1 \quad \text{-(T1)}
\]
\[
T(n) = T_S(n) + 2T(n/2) + T_M(n) \quad \text{(we ignore } O(1) \text{ terms)}
= n + 2T(n/2) \quad \text{-(T2)}
\]

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

(\text{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:

<table>
<thead>
<tr>
<th>example</th>
<th>recurrence:</th>
<th>running time:</th>
</tr>
</thead>
<tbody>
<tr>
<td>logarithm</td>
<td>$T(n) = 1 + T(n/2)$</td>
<td>$O(\log_2(n))$</td>
</tr>
<tr>
<td>factorial</td>
<td>$T(n) = 1 + T(n - 1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$T(n) = n + 2T(n/2)$</td>
<td>$O(n \log_2(n))$</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>$T(n) = 1 + 2T(n - 1)$</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>

(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 $10^3$ s 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 $\log_2$ 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 $S(1): T(1) = 1 \log_2(1) + 1$
  $T(1) = 1 = 0 + 1 = 1 \times 0 + 1 = 1 \log_2(1) + 1$

■ Inductive Case: given $S(n)$, show $S(2n)$:
  $T(2n) = 2n + 2T(n)\quad$, by -(T2)
  $= 2n + 2(n \log_2(n) + n)\quad$, by the Inductive Hypothesis
  $= 2n(1 + \log_2(n)) + 2n\quad$, rearranging terms
  $= 2n \log_2(2n) + 2n\quad$, using $\log_2(2x) = 1 + \log_2(x)$

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