EECS 477: Introduction to algorithms.

Lecture 7

Prof. Igor Guskov
guskov@eecs.umich.edu

September 26, 2002

Lecture outline

- Recursion issues
- Recurrence relations
- Examples

Recursion: factorial

- ```
  unsigned fact_rec(unsigned n) {
      return ( n==0 ? 1 : n*fact_rec(n-1) );
  }
  ```
- ```
  unsigned fact_iter(unsigned n) {
      unsigned result = 1;
      for(unsigned k=0; k<n; ++k)
          result *= k;
      return result;
  }
  ```

Recursion

- Recursive algorithms invoke themselves (*maybe indirectly*)
- May never terminate (need initial conditions/base case), then the result is not defined
- Similar to induction proofs – lend themselves nicely for things defined recursively
- Challenging for algorithm analysis: cannot use sequential composition/loops, cannot inline function calls

Recursion: advantages

- Algorithmic versions of inductive definitions *and proofs* are easy: for instance many algorithms on trees or mathematical functions on integers
- Especially, algorithm correctness: induction proofs
- GCD, factorial, tree traversal, etcetera
- Easy and fast implementation: directly from description, when you have little precious time, exam/deadline for noncritical parts

Recursion: usual drawbacks

- Complexity analysis is not as clear as correctness
  - Time: they call themselves
  - Memory: implicit program stack usage
- What actually happens when a function is called:
  - Caller: pushes local variables and arguments onto the program stack
  - Callee start-up: pops its arguments from program stack
  - Callee return: pushes return value onto the program stack
  - Caller: pops return value and local variables
  - More data may be pushed onto the program stack: registers

Recursion

- Time complexity handled by recurrencies
- Memory complexity handled by careful accounting
  - Statically allocated memory goes onto program stack
  - but not dynamically allocated
- Rule of thumb: limit static memory allocation in functions that are called many times, e.g. recursive

Recursion drawbacks

- In practice, recursive versions are often slower
  - Overhead of static variables and registers
  - Any recursive algorithm can be implemented without recursion using a single explicit stack – without slowdown (sometimes more convenient to use several stacks). Ex: think of evaluating arithmetic expressions – calculator
  - Rule of thumb: remove recursion from time-critical sections of your code
- Program stack is very limited and may overflow: so do not rely on it when implementing recursive algorithms for large datasets, e.g. graphs (*unless you have explicit support for that kind of adventures in your system*)

Recursion: misc

- Tail recursion: a *single* recursion call at the end – each call will reuse the same memory frame for local variables and pass the result *directly* to the caller
- Example:

```
int fact(int n) { return fact_tail(n, 1); }
int fact_tail(int n, int f) {
    if(n==0) return 1;
    return fact_tail(n-1, n*f);
}
```

- compare it to the simple recursive version: no need to keep local vars
- Another idea: memorize results on frequent instances, like dynamic programming

Recurrencies

- Equations where functions are unknowns (like difference eqs)
- *Time complexity* as unknown is of interest
- Ex: Factorial $t(n) = t(n-1) + c_1, t(0) = c_0$
- Solution: $t(n) = c_1 * n + c_0 = O(n)$
- A similar recurrence $t(n) = 3 * t(n-1) + c_1, t(0) = c_0$ but the solution is $\Omega(3^n)$, beware that *some constants matter*!!!
- A basic method: guess the answer and prove by induction

Intelligent guesswork

- $t(0) = 0$, $t(n) = 4t(n/3) + n$
- $t(0) = 0, t(1) = 1, t(3) = 7, t(9) = 37, t(27) = 175$, hmmm...
- $t(1) = 1, t(3) = 4 * 1 + 3, t(9) = 4^2 + 4 * 3 + 3^2, \ldots$
- Here's the pattern: $t(3^k) = \sum_{i=0}^{k} 3^i 4^{k-i} = 4^k \sum_{i=0}^{k} (3/4)^i = 4^{k+1} - 43^k$
- So for $n = 3^k$ we get $t(n) = 4(n^{\log_3 4} - n) = \Theta(n^{\log_3 4})$ then use smoothness rule

Linear recurrencies

- Recursive Fibonacci $t(n) = c_1 + t(n-1) + t(n-2)$ and $t(0) = t(1) = c_0$
- Need closed form solution
- And this is a linear recurrence!!! makes sense for recursion
- General form: $a_0 t(n) + a_1 t(n-1) + \ldots + a_k t(n-k) = f(n)$ plus initial conditions on $t(0), \ldots, t(k-1)$
- a_k are constants
- Start with homogeneous case: $f(n) = 0$: solutions form linear space (can add and scale them)

Linear recurrencies: characteristic polynomial

- Consider solution of exponential kind $t(n) = x^n$, substitute into equation to get

$$a_0 x^n + a_1 x^{n-1} + \ldots a_k x^{n-k} = 0$$

or

$$a_0 x^k + a_1 x^{k-1} + \ldots a_k = 0$$

- Find roots of the above and assume they are different $r_1, \ldots, r_k$. Then

$$t(n) = c_1 r_1^n + c_2 r_2^n + \ldots + c_k r_k^n$$

is a general solution form, find constants from initial conditions

Linear recurrencies: multiple roots

- for a root r of multiplicity m we get m fundamental solutions

$$r^n,\ nr^n, \ldots,\ n^{m-1}r^n$$

- Again, find constants from initial conditions

Linear recurrencies: inhomogeneity

- Inhomogeneous are important!

$$a_0 t(n) + a_1 t(n-1) + \ldots + a_k t(n-k) = b^n p(n),$$

  restricted version where $p(n)$ is a polynomial of degree d.

- Solution involves forming the *implied homogeneous recurrence*:

$$(a_0 x^k + a_1 x^{k-1} + \ldots a_k)(x-b)^{d+1} = 0$$

- General solution is $t(n) = \sum_i \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$ then substitute into the original recurrence and initial condition

- Example: 4.7.8 on page 128