EECS 477: Introduction to algorithms.

Lecture 6

Prof. Igor Guskov
guskov@eecs.umich.edu

September 24, 2002

Lecture outline

- Finish up with asymptotic notation
- Asymptotic analysis of programs
- Analyzing control structures: sequencing, for-loops, recursive calls, while- and repeat-loops
- Using a barometer
- Examples
- Average case analysis vs amortized analysis

Conditional notation

- Initially useful to do a simpler restricted case
- Long integer multiplication assume that the sizes are powers of two
- Or for binary search – can claim complexity $O(\log n | n = 2^p)$ (note the notation!)
- Once the special case is handled, generalize it. This is often easy because complexity is an eventually non-decreasing function often. Thus $O(\log n)$ propagates to all values of n
- This is easy for smooth eventually non-decreasing functions
- $f(n)$ is b-smooth iff $f(bn) = O(f(n))$
- n^k is smooth, 2^n is not – prove!

Smoothness

- If $f(n)$ is b-smooth for some integer $b \geq 2$ then it is smooth (for all other such b's). That is, $f(n) \leq cf(bn)$ and $f(n) \leq f(n+1)$ imply $f(n) \leq c'f(an)$
- *Smoothness rule:* If $t(n) \in \Theta(f(n)|n = b^k)$ and f is smooth and t is eventually non-decreasing, then $t(n) \in \Theta(f(n))$ unconditionally.
- e.g. if $t(n) \in \Theta(n^2|n = 2^k)$ and ev. non-decreasing then $t(n) \in \Theta(n^2)$ unconditionally (same for O and Ω.
- Take $n' = b^{\lfloor \log_b n \rfloor}$ then

$$t(n) \leq t(bn') \leq af(bn') = af(bn') \leq acf(n') \leq acf(n);$$

that is $t(n) \in O(f(n))$ unconditionally.

Multiple parameters

- Two sorted arrays of size K and M
- Problem: Count all repetitions and sort the result
- I: scan both $O(\max(K, M))$
- II: binary-search elements of the smaller array in the larger one $O(\min(M, K) \log(\max(M, K))$
- Formally:
  $\exists c > 0 \; \exists m_0 \in \mathbf{N} \; k_0 \in \mathbf{N} \; \forall k > k_0 \; \forall m > m_0 \; g(k, m) \leq c f(k, m)$.

Operations on asymptotic notation

- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
- also works for other operations
- $n^{O(1)}$ denotes all the functions dominated by Cn^k, this is basically polynomial growth functions
- $f(n) \in n^{O(1)}$ means that $\exists \alpha(n) \in O(1)$ such that $f(n) = n^{\alpha(n)}$

Asymptotic analysis of programs

- We assume that our algorithms are represented by programs, e.g. in pseudocode, C, C++ etc.
- Memory and runtime
  - Traditionally, runtime analysis comes first
  - If $\Omega(f(n))$ memory is used then $\Omega(f(n))$ is required (UNLESS we allocating memory without initialization)
- Basic idea: bottom-up analysis: from elementary operations to control structures and further upwards
- Pitfalls: insufficient specification, hidden complexity, unstructuredness (*goto*'s)

Example: hiding complexity

```
struct prime_iterator {
  int operator++(int) {
    do { ++m_k; } while(!is_prime(m_k));
    return m_k;
  }
  int m_k;
}
bool is_prime() { // check primality }

int count_primes(int n) {
  prime_iterator i;
  int count = 0;
  for(i=2; i<n; ++i) ++count;
  return count;
}
```

Analyzing control structures

- Time/memory complexity of simple steps
- Assemble elementary steps into structures
  - Sequencing
  - for-loops
  - recursive calls
  - while-loops
- Go on to more involved steps

Sequencing

- Sequential composition: assume two steps, assume they are *independent*
- Runtime of a sequential composition of two steps is the sum of runtimes
- Memory taken by sequential composition: anywhere from max to sum: depends on whether memory allocated by the first step can be reused by the second
- Recall $O(\max(f, g)) = O(f + g)$
- Compare to "parallel composition" where memory is a sum and runtime can be min to max

For-loops

- `for(i=1; i<n; ++i) P(i)`
- Does at least as much work as `P(i)` repeated n times
- Complexity *at least* n times that of `P(i)`
- Additional work: counter maintenance
- Counter maintenance can be ignored if the body of the loop is asymptotically more or as expensive
- C++ iterator's example is not rare: for complex containers
- For dynamic loop conditions analysis may be harder

For-loops

- ```
  unsigned FibIter(unsigned n) {
    unsigned j=1, i=0, k;
    for(k=0; k<n; ++k) {
      j += i;
      i = j - i;
    }
    return j;
  }
  ```
- Prove by induction that `j` is n-th Fibonacci number, there are n loop iterations, $O(n)$?
  ```

For-loops

- Complexity of each step
- k-th Fibonacci number has $O(k)$ digits
- Each step takes at least linear time
- Hence, the whole procedure takes quadratic time:

$$\sum_{k=0}^{n} k = O(n^2).$$

Recursive calls

- Complexity analyzed by composing an equation and solving it
-

```
unsigned Fibrec(unsigned n) {
  if(n<2) return n;
  else return Fibrec(n-1) + Fibrec(n-2);
}
```

- Recurrence:
  $T(n) = a$ for $n < 2$
  $T(n) = T(n-1) + T(n-2) + h(n)$ otherwise
- Later we'll see that this takes exponential time!

While-loops

- Difficult analyse the number of iterations
- Trick: find a decreasing function: if it decreases by more than one every time then look at the value
- Binary search: distance between the right and the left index decreases with every step until the end by 2x
- page 103 for more details

Barometer

- *Barometer* is a step that is taken at least as many times as any other step.
- Asymptotic complexity allows to drop constant factors and ignore all the steps except barometers
- ```
  unsigned FibIter(unsigned n) {
    unsigned j=1, i=0, k;
    for(k=0; k<n; ++k) {
      j += i; // barometer
      i = j - i;
    }
    return j;
  }
  ```
  ```

Barometer

- Especially convenient for the analysis of nested loops
- Last phase of pigeonhole sorting

```
i = 0;
for(k=1; k<s; ++k) {
  while(U[k]!=0) {
    ++i;
    T[i] = k;
    --U[k];
  }
}
```

- Cannot use inner instructions as barometers because sometimes they are not taken
- Complexity of the above is $O(n + s)$

Greatest common divisor

- Example of while loop analysis

```
unsigned GCD(unsigned m, unsigned n) {
  while(m>0) {
    n = n%m; swap(m,n);
  }
  return n;
}
```

- Runtime – number of iterations: in the worst case $2 \log \min(m, n)$ – progress occurs every second iteration

Projects

- Individual or teams of up to three people
- You can suggest a topic – need to be approved
- Involves: algorithm design/analysis, implementation likely, should not be entirely subsumed by published results
- Template project: choose NP-hard problem (say from Garey and Johnson), implement/analyze exact algorithm (time/memory complexity), design/implement heuristics/online algorithm with $O(n^d)$ and analyse its memory complexity