EECS 477: Introduction to algorithms.

Lecture 3

Prof. Igor Guskov
guskov@eecs.umich.edu

September 10, 2002

Lecture outline

- Combinatorics
- Algorithmics
- Computing mathematical functions
- Computational models
- Problems and instances

Combinatorics

- set of N elements: subsets *ordered* and *unordered*
- equivalence relation: 2^N unordered subsets
- subsets with exactly k elements $C(N,k) = N!/(N-k)!k!$, need to compute without overflows
- rectilinear paths: 1 is up, 0 is right

Top-down design and analysis

- identify relevant invariants and abstractions, reason about these: use known generic techniques/theorems
- remove unnecessary level of detail
- translate results back into domain-specific terms, fill in details
- example: build fence around towns (convex hull, D&C)

Top-down design and analysis II

- improves reuse

- distinguish coding mistakes from fundamental flaws: who's responsible? CEO: ensure perspective and delegate details. In practice requires experience with application area

- improves learning curve, documentation

- Sample abstraction: graphs, ordering relations

- Sample approach: divide and conquer, greedy, lazy, ...

Design reuse

- just another aspect of top-down
- primitives that appear over and over again: sorting, searching, string matching
- research resulted in efficient algorithms: implementations took years of work, available in software libraries
- e.g. reformulate your problem in graph terms
- reuse is a good idea: implementations, analysis

Algorithmics

Need to distinguish for complexity analysis:

- Problem: given a function, how can we efficiently compute it?

- Algorithms: recipes to compute a given function

- Programs: formalized recipes understandable by computers

Algorithmics II

- Problem complexity: the best possible algorithm complexity

- Implementation: may not implement a given algorithm correctly – that would be a way to find bugs: $O(n \log n)$ algorithm complexity and $O(n^2)$ running times may indicate a bug

Goals of algorithm analysis

- evaluate and compare existing algorithms: empirically and theoretically, determine winners under different circumstances (e.g. quicksort, mergesort, pigeon-hole-sort)

- evaluate, compare and diagnose given implementations

- analyze the complexity of algorithmic problems: sometimes it is hopeless to look for an algorithm

- can prevent lots of useless programming

Algorihm evaluation

Various parameters

- Time (efficiency, performance): asymptotic, on actual inputs
- Memory (size)
- Ease of programming
- Worst, best, average, typical case
- all of these mean potentially no single winner

Principle of invariance

- Challenge for time complexity analysis: different hardware, different number of instructions per second, but we need some uniform measure of time complexity

- Solution: measure constant time steps – we will be off at most by a constant

- Principle of invariance: any two implementations of an algorithm (when executed on actual computers) will not differ in time complexity by more than a constant (this is not a theorem)

Computing mathematical functions

- $C(N,k) = N!/((N-k)!k!)$ so we need to be careful with $N!$ try doing 20! – you'll get overflow. On the other hand $C(20,19) = 20$ so it's not a problem and $C(20,10) = 184756$ is not that big either.

- another try $C(N,k) = (N(N-1)\ldots(N-k+1)/k!$.

- the largest value we get for $k = floor(N/2)$

- so we not very successful

N choose k

- $C(N, k+1) = C(N, k)(N-k)/(k+1)$ is easy to prove so:

  ```
  C(N,1) = N;
  for i=1 to k-1 do
    C(N,i+1) = ( C(N,i)*(N-i) ) / (i+1);
  ```

- note that we get integers all the way: otherwise we would be wrong (can also reduce $(N-k)/(k+1)$ to p/q and divide by q first)

- for $k \leq N/2$ result of each step does not exceed the final result, and for the rest of k we use $C(N, k) = C(N, N-k)$

Greatest common divisor

- $GCD(m, n)$ is the greatest integer that divides both m and n. So we can use it to maximally simplify the fraction m/n

- how to compute $GCD(m, n)$?

  I. By definition: try all integers from 1 to m/n – too slow.

  II. Decompose m and n into prime factors and collect the common portion and multiply to get GCD – but factoring large numbers is a very hard computational problem!

Greatest common divisor

- Euclid to the rescue (more than 2000 years ago)

```
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 $\log \min(m, n)$.

- Another form uses subtractions only

Determinants

- Direct computation from definition: at least $N!$ steps

- via Gaussian elimination: get upper-triangular matrix with the same determinant

- so we get on the order of N^3 steps – that is much faster

Computing mathematical functions

Conclusions:

- sometimes following definition leads to: numerical overflows or a hopelessly slow algorithm

- to find a better one: analyze worst/better cases, use alternate definitions, target the worst case, reduce it to the best case

- Fibonacci sequence – another example.

Computational models

- Elementary operations: execution time is bounded by a constant that does not depend on input values. Actual seconds per operation may be disregarded. Selection of elementary operations is hardware dependent.

- Arithmetic operations: +, -, *, etcetera

Yes: if integers have bounded number of bits

No: otherwise (unbounded)

- Function calls, memory accesses, e.g. a[i]?

Computational models II

- Computational model is determined by: data representation and storage mechanisms, available elementary operations.

  Examples: logic circuits, deterministic finite automata, push-down automata, Turing machines, C/C++ programs

  External memory: two-level disk model – count I/O operations

- Computational models originate in technologies, physics, biology, etcetera

- Parallel and distributed computing, optical and DNA computing, analog computing, quantum computing

Problems vs instances

- Problem: in the *functions domain*

- Instance: one possible argument – function input

- For instance, C(N,k) vs C(20, 3)

- Instances can be different: best and worst case, average (ex: sorting insertion (sensitive) vs selection(insensitive))

- Instances that require average resources are called *average case instances*