Basic Complexity Theory (Review)

CSE 373
Data Structures
Winter 2006

Agenda

• Goals: want to be able to analyze algorithm time (in particular) and space requirements
• Benchmarking
• Machine-independent characterizations
• Asymptotic complexity

Benchmarking

• Use stopwatch (or system clock) to time algorithms
• Repeat for inputs of different sizes
  › Graph results: time as function of input
• Maybe repeat for different algorithms
  › Compare results; which algorithm is “better”?

Benchmarking Pro

• Real numbers – often what the customer wants to know
• Reasonable results as long as machine, compiler, OS don’t change
• Reasonable if constraints are the same (e.g., all data fits in main memory vs some in main memory vs some on disk)

Benchmarking Con

• Depends on particular technology
  › Can’t make meaningful statements about results from different machines, compilers
• Too concrete – can’t answer questions like “is quicksort better than insertion sort” in a machine-independent way

Complexity Theory

• Idea: abstract away from particular machines, implementations
• Measure time/space in abstract “steps” or “cells”
  › Does not depend on particular implementations
• Analyze independent of particular input
  › In particular, analyze as function of large inputs – asymptotic analysis
Problem Size

- Want to analyze time/space as a function of the problem "size"
  - Want this to be relatively abstract
- Typical "sizes"
  - Amount of input – how much do we need to sort?
  - Size of data structure – number of items, nodes, edges
  - Magnitude of parameters – effort needed to compute n! as a function of n, for example

Execution Costs

- Basic steps
  - Initialization/assignment of scalar variables (int, double, char, boolean, pointer, reference)
  - Simple arithmetic operations (+, -, *, /, %)
  - Simple conditional tests (&&, ||, !)
  - Array subscripting (a[k])
  - Parameter passing/initialization
  - Method call/return (excluding cost of executing method body)

- Sequence of statements s1; s2; …; sn
  - Sum of costs of s1, s2, …, sn
- Loops
  - Loop overhead (constant) +
  - Sum of costs of iterations
    - Sometimes iteration cost * #iterations
    - Other times need to sum up iteration costs if they are not always the same (example: insertion or selection sort)

- if cond then s1 else s2:
  - cost of cond +
  - Max of cost s1, cost s2 (worst case), or
  - Weighted average of cost s1, cost s2 depending on probability that cond is true or false (expected case – harder)
- Typically we use worst-case analysis

Method calls

- Cost of evaluating arguments +
- Argument passing/parameter initialization (usually constant, but more complex if copying large values) +
- Call/return overhead (usually constant) +
- Cost of executing method body

Comparing Algorithms

- Use cost measures to figure out the cost of each algorithm
  - Result can be complex
- Then abstract away from noise – small terms don’t matter
- Worry about cost as input becomes large
Asymptotic Complexity

• Def: If f(n) and g(n) are two (complexity) functions, we say that
  f(n) is O(g(n)) (pronounced "is order of")
  if there are constants c, n₀, such that
  f(n) ≤ c ⋅ g(n)
  for all n ≥ n₀.

Exercise

• Prove that 3n + 5n² + 373 is O(n²)
• Prove that it is O(n⁴)

Significance

• This measures asymptotic complexity
  › Low-order terms don’t matter
  › Small values of n don’t matter
• Tight bounds are better (prefer small functions to large, even if both are valid)
• This is a worst-case analysis
  › Generally useful in practice
  › Usually easier than average-case (expected-time) analysis, but
  › Sometimes want expected-time analysis (e.g., worst-case is pathologically worse than typical)

Complexity Classes

• Key complexity classes (know these!)
  › Constant: O(1) (or O(k) for any constant k)
  › Logarithmic: O(log n) (base doesn’t matter)
  › Linear: O(n)
  › n log n: O(n log n)
  › Quadratic: O(n²)
  › Cubic: O(n³)
  › Polynomial: O(nᵏ)
  › Exponential: O(kⁿ)

Comparing Algorithms

• Generally, lower asymptotic complexity is preferable
  › But constants and low-order terms may matter for problems of practical size, so don’t do this blindly
• Algorithms of polynomial size (nᵏ) or better are generally feasible
• Exponential algorithms (kⁿ) are usually not feasible – even if computers get a lot faster