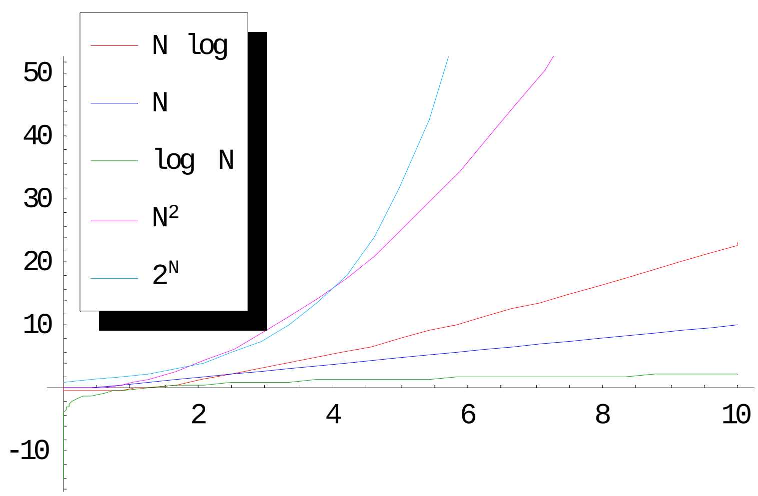
Benjamin Fenster
CS-104 (Lee)
14 September 2003

Notes – Overview, Analysis of Algorithms

I. Course Themes
  a. Data Structures
  b. Abstract Data Types
  c. Algorithms
  d. Analysis of Algorithms
  e. Relationships
    i. ADTs and Data Structures
      1. ADTs represent Data Structures
      2. Alternative data structures can represent an ADT.
      3. Set and List can both represent many data structures. Data structures can represent many ADTs.
    ii. Data Structures and Algorithms
      1. An algorithm is: “A mechanical or recursive computational procedure.”
      2. Algorithms and DSs are co-related.
      3. Writing working code is easy. Getting run time within limits is hard.

II. Analysis of Algorithms
  a. Overview
    i. We want to show that one algorithm is more efficient than another.
    ii. Could be done empirically but we’re interested in analytic methods.
    iii. We have a data structure implementing operations on an ADT.
    iv. We want to write an algorithm to implement a particular operation, so we need some analysis.
  b. Basics
    i. Growth Rate of Functions
      1. Consider functions of N
      2. N log N, N, 2^N, N^2, log N
      3.



  c. Example
    i. See slides
    ii. $(a_0 + a_1k) + a_2k$, s[k] = x
      1. Scans linearly for x, finds at position k. Then finds partial sum.
      2. Some fixed overhead plus amount of time to step through array, plus amount of time to sum array.
  d. Run-Time
    i. Worst Case (*never* longer than this)
    ii. Average (statistically predicted)

iii. Best (“in your dreams” time)
iv. We’re usually interested in the worst-case time since we can be sure it will never take longer than that.

III. Relative Growth
  a. T(N), f(N) functions
  b. $\zeta(T(N)), \zeta(f(n))$
  c. $T(n) = O(f(N))$ iff $\zeta(T(N)) <= \zeta(f(N))$
  d. $T(N) = \Omega(f(N))$ iff $\zeta(T(N)) >= \zeta(f(N))$
  e. $T(N) = \theta(f(N))$ iff $\zeta(T(N)) = \zeta(f(N))$
  f. $T(N) = o(f(N))$ iff $\zeta(T(N)) < \zeta(f(N))$

IV. Asymptotic Expressions of Run-Time
  a. $T(N) = O(f(N))$ iff $\exists\ c_0 > 0,\ n_0 > 0 \ni T(N) \le c_0 f(N)$ for all $N > n_0$.
  b. That means the same thing as the previous definition.
  c. $T(N) = \theta(f(N))$ iff $T(N) = O(F(N))$ and $T(N) = \Omega(f(N))$
  d. Example
    i. $T(N) = 3 + 8N + 5N^2$
    ii. $T(N) = O(N^2)$
    iii. Proof:
    iv. Need $c_0 > 0,\ n_0 > 0 \ni T(N) \le c_0N^2$
    v. Choose $n_0 = 1$ arbitrarily
    vi. $T(N) < c_0N^2$ for any $c_0 > 16$ (substitute 1 for N and solve). $3 + 5(1) + 8(1) = 16$
  e. Run-Time Bounds
    i. O(f(N)) means “upper-bound” (worst case)
    ii. $\Omega(f(N))$ means “lower bound” (best case)
    iii. $\theta(f(N))$ means “tight bound” (best and worst cases are the same)

V. Big-O Rules
  a. If $T(N) = O(c\ f(N))$ then $T(N) = O(f(N))$ where c is a constant.
    i. This means we’re worried only about scale / growth rate.
    ii. Constants are irrelevant.
  b. If $T_1(N) = O(f_1(N)$ and $T_2(N) = O(f_2(N))$ then…
    i. $T_1(N) * T_2(N) = O(f_1(N) * f_2(N))$
    ii. and $T_1(N) + T_2(N) = \max(O(f_1(N)), O(f_2(N)))$
  c. If T(N) is a polynomial of degree k then $T(N) = \theta(N^k)$
  d. $\log^k(N) = O(N)$ for any k
    i. This shows that logs grow much slower than linear equations.
    ii. Logarithm to any power will never exceed linear.
    iii. This rule isn’t terribly important. See slides for its proof.

VI. Basic Rules for Asymptotic Algorithm Analysis
  a. Non-Recursive
    i. Loop
      1. for I from 1 to N, j from 1 to M
      2. O(MN) (constant runtime for each innermost instruction, so pull it out by the first Big-Oh rule)
      3. If M = cN for some constant then $O(MN) = O(M^2)$ or $O(N^2)$
    ii. Sequence
      1. One block followed by another.
      2. First, O(f(N)), second O(g(N))
      3. Total: max(O(f(N)), O(g(N))
    iii. Conditional Branching
      1. If … then O(f(N)) else O(g(N))
      2. Total: Take max
      3. We want Big-O, worst case, so take worst side of the ‘if’
  b. Recursive
    i. Harder

ii. Need a Recurrence Relation: A mathematical relationship expressing f_n as some combination of f_i with $i < N$
iii. When formulated as an equation to be solved, called a recurrence equation.
iv. Example
    1. Binary Search
    2. Run Time Metric: Number of comparisons performed
    3. Problem Size: Number of elements in the search
    4. $T(N) = T(N/2) + 2$ if $N \geq 2$
       $= 1$ if $N = 1$
    5. Define $2^n = N$
    6. $T(2^n) = T(2^{n-1}) + 2$
    7. $T(2^{n-1}) = T(2^{n-2}) + 2$
    8. …
    9. $T(2) = T(1) + 2$
    10. Left with $T(N) = T(1) + 2n$
    11. $T(N) = 2 \log N + 1 = O(\log N)$
    12. We've turned a potentially nasty division problem into subtraction.
v. Example
    1. MinMax
    2. Runtime = Number of Comparisons
    3. Problem Size = Number of Elements
    4. $T(N) = 2(T(N/2)) + 2$ if $N \geq 2$
       $= 0$ if $N = 0$
    5. $T(N) = T(2^n) = 2^1T(2^{n-1}) + 2$
    6. $2^1T(2^{n-1}) = 2^2T(2^{n-2}) + 2^2$
    7. $2^2T(2^{n-2}) = 2^3T(2^{n-3}) + 2^3$
    8. …
    9. $2^{n-1}T(2) = 2^nT(1) + 2^n$
    10. Collapses to: $T(2^n = 2^nT(1) + 2^n + 2^{n-1} + \ldots + 2^2 + 2$
    11. $= 2^n + 2^{n-1} + \ldots + 2^2 + 2$
    12. $= 2^n ((1 - ½^n) / (1 - ½)) = 2^{n+1} - 2 = 2(2^n) - 2 = 2N - 2 = O(N)$
    13. We get there using the geometric sum formula.
vi. Example
    1. Fibonacci Numbers
    2. $T(N) = T(N - 1) + T(N - 2)$ if $N \geq 2$
       $= 0$ if $N < 2$
    3. Note that the equation looks just like the function itself.
    4. $Fib(N) = \phi^n / \sqrt{5}$ where $\phi = (1 + \sqrt{5}) / 2$
    5. So let's say $T(N) = Fib(N)$
    6. In other words, $T(N)$ itself behaves like Fibonacci numbers
    7. So $T(N) = O(\phi^n)$ -- exponential!
    8. Exercise: Now solve this the same way as the Min-Max numbers.