CSC 8301- Design and Analysis of Algorithms

Lecture 2

Techniques for efficiency analysis of nonrecursive algorithms

Analyzing time efficiency of an algorithm

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size.*

Basic operation: the operation that contributes most towards the running time of the algorithm.

input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time for basic operation

Number of times basic operation is executed

<u>Order of growth</u> of *C*(*n*) is of primary interest.

Two approaches to efficiency

- Theoretical – use mathematical tools such as
  - summations (mostly for nonrecursive algorithms)
  - recurrence relations (mostly for recursive algorithms)

- Empirical – select an input sample (e.g., randomly) and
  - measure running time in some physical unit (milliseconds)
    or
  - count actual number of basic operation executions by inserting counter(s) in appropriate places of the code

Math analysis of nonrecursive algorithms
General Plan

- Decide on parameter n indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size n
- Set up summation for $C(n)$, the basic operation count, reflecting algorithm's loop structure
- Simplify summation using standard formulas (see Appendix A)

Useful summation formulas

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$
In particular, $\Sigma_{1 \le i \le n} 1 = n\text{-}1+1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1) \in \Theta(a^n)$
In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma_{1 \le i \le n}\, 1/i = 1/1+1/2+\ldots+1/n \approx \ln n + 0.5772\ldots \in \Theta(\log n)$

$\Sigma_{1 \le i \le n} \lg i = \lg 1+\lg 2+\ldots+\lg n \in \Theta(n \log n)$

Useful summation rules

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$

$\Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

$\Sigma_{l \le i \le u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Approximation by definite integrals
$\int_{l-1}^{u} f(x)dx \le \Sigma_{l \le i \le u} f(i) \le \int_{l}^{u+1} f(x)dx$ for nondecreasing $f(x)$

$\int_{l}^{u+1} f(x)dx \le \Sigma_{l \le i \le u} f(i) \le \int_{l-1}^{u} f(x)dx$ for nonincreasing $f(x)$

Example 1: Maximum element

```
ALGORITHM MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

- *Input size*
- *Basic operation*
- *Worst, average, and best cases*
- *Summation for C(n)*

Example 2: Element uniqueness problem

```
ALGORITHM UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns "true" if all the elements in A are distinct
    //         and "false" otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

- *Input size*
- *Basic operation*
- *Best case*
- *Worst case – summation for C(n)*

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*($A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]$)
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

- *Input size*
- *Basic operation*
- *Worst, average, and best cases*
- *Summation for C(n)*

Example 4: Gaussian elimination

Algorithm *GaussianElimination*($A[0..n-1,0..n]$)
//Implements Gaussian elimination of an n-by-(n+1) matrix A
for $i \leftarrow$ 0 to n - 2 do
    for $j \leftarrow$ i + 1 to n - 1 do
    for $k \leftarrow n$ downto i do
        $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$

Find the efficiency class and a constant factor improvement.

Example 5: Counting binary digits

ALGORITHM *Binary*(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    $count \leftarrow 1$
    while $n > 1$ **do**
        $count \leftarrow count + 1$
        $n \leftarrow \lfloor n/2 \rfloor$
    return $count$

It cannot be investigated the way the previous examples are.

Homework

Exercises 2.3: 1–12 (you may skip 3 and 7)

Reading: Sections 2.3, 2.6 and 2.7

Watch the "Sorting Out Sorting Video" at http://www.youtube.com/watch?v=SJwEwA5gOkM

(note that today's computers are several orders of magnitude faster than those in the film; hence one needs much larger files to see a difference in the speed of different sorting algorithms)

Next: Sections 2.4, 2.5, and Appendix B