Chapter 2: Fundamentals of the Analysis of Algorithm Efficiency

The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency. (Bill Gates)

Analysis of Algorithms

- Issues:
  - Correctness. Is the algorithm correct?
  - Time Efficiency. How much time does the algorithm use?
  - Space Efficiency. How much extra space (in addition to the space needed for the input and output) does the algorithm use?
- Approaches:
  - Theoretical Analysis. Proof of correctness and big-Oh and related notations.
  - Empirical Analysis. Testing and measurement over a range of instances.

Efficiency as a Function of Input Size

Typically, more time and space are needed to run an algorithm on bigger inputs (e.g., more numbers, longer strings, larger graphs).

Analyze efficiency as a function of n=*size of input*.

- Searching/sorting. n=number of items in list.
- String processing. n = length of string(s).
- Matrix operations, n = dimension of matrix. $n \times n$ matrix has n^2 elements.
- Graph processing. n_V = number of vertices and n_E = number of edges.

Measuring Running Time

Analyze time efficiency by:

- identifying the *basic operation(s)*, the operation(s) contributing the most to running time,
- characterizing the number of times it is performed as a function of input size.

We can crudely estimate running time by $T(n) \approx c_{op} * C(n)$

- $T(n)$: running time as a function of n.
- c_{op}: running time of a single basic operation.
- $C(n)$: number of basic operations as a function of n.

Worst-Case, Best-Case, and Average-Case

algorithm $SequentialSearch(A[0..n-1], K)$
  // Searches for a value in an array
  // Input: An array A and a search key K
  // Output: The index where K is found or -1
  for $i \leftarrow 0$ **to** $n-1$ **do**
    if $A[i] = K$ **then return** i
  return -1

- Basic Operation: The comparison in the loop
- Worst-Case: n comparisons
- Best-Case: 1 comparison
- Average-Case: $(n+1)/2$ comparisons assuming each element equally likely to be searched.

Order of Growth

Typically, the basic operation count can be approximated as $c\,g(n)$, where $g(n)$ is the *order of growth*, often some "simple" function such as:

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Asymptotic Order of Growth

A way of comparing functions that ignores constant factors and small input sizes.

- Big-Oh $O(g(n))$: functions $\leq c\,g(n)$.

  $t(n) \in O(g(n))$ if there are positive constants c and n_0 such that $t(n) \leq c\,g(n)$ for all $n \geq n_0$

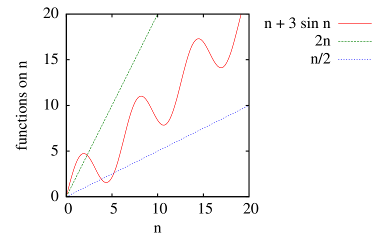
- Big-Omega $\Omega(g(n))$: functions $\geq c\,g(n)$.

  $t(n) \in \Omega(g(n))$ if there are positive constants c and n_0 such that $t(n) \geq c\,g(n)$ for all $n \geq n_0$

- Big-Theta $\Theta(g(n))$: functions $\approx c\,g(n)$.

  Both $t(n) \in O(g(n))$ and $t(n) \in \Omega(g(n))$.

Illustration of Asymptotic Order



Graph suggests (and we can prove) $n + 3\sin n$ is:
$O(n)$: $n + 3\sin n \leq 2n$ when $n \geq 3$.
$\Omega(n)$: $n + 3\sin n \geq n/2$ when $n \geq 6$.

Examples of Asymptotic Order

$t(n)$	$O(n)$	$O(n^2)$	$O(n^3)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$
$\log_2 n$	T	T	T	F	F	F
$10n + 5$	T	T	T	T	F	F
$n(n-1)/2$	F	T	T	T	T	F
$(n+1)^3$	F	F	T	T	T	T
2^n	F	F	F	T	T	T

For example, $10n + 5$ is $\Theta(n)$. Assuming $n \geq 5$:
$10n+5 \in O(n)$ because $10n+5 \leq 10n+n = 11n$.
$10n + 5 \in \Omega(n)$ because $10n + 5 \geq 10n$.

Properties of Order of Growth

- If $f_1(n)$ is order $g_1(n)$, and $f_2(n)$ is order $g_2(n)$, then $f_1(n) + f_2(n)$ is order $\max(g_1(n), g_2(n))$.
- If $b > 1$, then $\log_b n \in \Theta(\log n)$.
- Polynomials of degree $k \in \Theta(n^k)$, that is, $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 \in \Theta(n^k)$.
- Exponential functions a^n have different orders of growth for different a's.
- order $\log n <$ order n^k where $k > 0 <$ order a^n where $a > 1 <$ order $n! <$ order n^n

CS 3343 Analysis of Algorithms Chapter 2: Slide – 10

Basic Asymptotic Efficiency Classes

Class	Name	Example
1	constant	access array element
$\log n$	logarithmic	binary search
n	linear	find median
$n \log n$	"n-log-n"	mergesort
n^2	quadratic	insertion sort
n^3	cubic	matrix multiplication
a^n	exponential	generating all subsets
$n!$	factorial	generating all permutations

CS 3343 Analysis of Algorithms Chapter 2: Slide – 11

Mathematical Analysis of Nonrecursive Algorithms

Plan for Analyzing Nonrecursive Algorithms

- Decide on parameter n indicating input size.
- Identify algorithm's basic operation(s).
- Determine worst, average, and best cases for input of size n.
- Set up a sum for the number of times the basic operation is executed.
- Simplify the sum using standard formulas and rules (see Appendix A).

CS 3343 Analysis of Algorithms Chapter 2: Slide – 12

Useful Summation Formulas and Rules

- $\sum_{i=1}^{n} 1 = 1 + 1 + \cdots + 1 = n \in \Theta(n)$
- $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \in \Theta(n^2)$
- $\sum_{i=1}^{n} i^k = 1 + 2^k + \cdots + n^k \approx \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$
- $\sum_{i=1}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \in \Theta(a^n)$
- $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$ $\qquad$ $\sum_{i=1}^{n} c\, a_i = c \sum_{i=1}^{n} a_i$

CS 3343 Analysis of Algorithms Chapter 2: Slide – 13

Finding the Maximum

algorithm $MaxElement(A[0..n-1])$
  // Returns the maximum value in an array
  // Input: A nonempty array A of real numbers
  // Output: The maximum value in A
  $maxval \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$ **then**
      $maxval \leftarrow A[i]$
  return $maxval$

Basic Operation: comparison in loop

Performs $\sum_{i=1}^{n-1} 1 = n-1$ comparisons

Element Uniqueness

algorithm $UniqueElements(A[0..n-1])$
  // Returns true if all elements are different
  // Input: An array A
  // Output: Returns true if there are no duplicates
  for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
      if $A[i] = A[j]$ **then return false**
  return true

- Basic Operation: inner loop comparison
- Best/Worst-Case: from 1 to $n(n-1)/2$

$$\sum_{i=0}^{n-2} n-(i+1) = (n-1)+(n-2)+\cdots+1 = \sum_{k=1}^{n-1} k = n(n-1)/2$$

Matrix Multiplication

algorithm $MatrixMultiply(A, B : n \times n$matrices)
  // Multiplies matrix A times matrix B
  // Input: Two $n \times n$ matrices A and B
  // Output: Matrix $C = AB$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
      $C[i,j] \leftarrow 0$
      for $k \leftarrow 0$ **to** $n-1$ **do**
        $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$
  return C

Basic Operation: inner loop multiplications

Performs $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = n^3$ multiplications

Number of Bits in an Integer

algorithm $BitCount(m)$
  // Input: A positive integer m
  // Output: The number of bits to encode m
  $count \leftarrow 1$
  while $m > 1$ **do**
    $count \leftarrow count + 1$
    $m \leftarrow \lfloor m/2 \rfloor$
  return $count$

n = length of m in bits
Basic Operation: The division by 2 in loop
Performs $n-1$ operations because $n = count$.

Mathematical Analysis of Recursive Algorithms

Plan for Analyzing Recursive Algorithms

- Decide on parameter n indicating input size.
- Identify algorithm's basic operation(s).
- Determine worst, average, and best cases for input of size n.
- Set up a recurrence relation expressing the basic operation count.
- Solve the recurrence (at least determine it's order of growth).

Factorial Function

algorithm $Factorial(n)$
 // Computes $n!$ recursively
 // Input: A nonnegative integer n
 // Output: The value of $n!$
 if $n = 0$ **then return** 1
 else return $Factorial(n-1) * n$

Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: multiplication

Recurrence for Factorial Function

- Let $M(n)$ = multiplication count to compute $Factorial(n)$.
- $M(0) = 0$ because no multiplications are performed to compute $Factorial(0)$.
- If $n > 0$, then $Factorial(n)$ performs recursive call plus one multiplication.

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ Factorial(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ Factorial(n-1) \text{ by } n}}$$

Solving the Factorial Recurrence

Make a reasonable guess.

- Forward substitution. $M(1) = M(0) + 1 = 1$
 $M(2) = M(1) + 1 = 2$
 $M(3) = M(2) + 1 = 3$
- Backward substitution. $M(n) = M(n-1) + 1$
 $= [M(n-2) + 1] + 1 = M(n-2) + 2$
 $= [M(n-3) + 1] + 2 = M(n-3) + 3$

Prove $M(n) = n$ by mathematical induction.

- Basis: if $n = 0$, then $M(n) = M(0) = 0 = n$
- Induction: if $M(n-1) = n-1$, then
 $M(n) = M(n-1) + 1 = (n-1) + 1 = n$

Towers of Hanoi Algorithm

algorithm $Towers(n, i, j)$
 // Moves n disks from peg i to peg j
 // Input: Integers $n > 0$, $1 \le i, j \le 3$, $i \ne j$
 // Output: Specifies disk moves in correct order
 if $n = 1$ **then**
  move disk 1 from peg i to peg j
 else
  $Towers(n-1,\ i,\ 6-i-j)$
  move disk n from peg i to peg j
  $Towers(n-1,\ 6-i-j,\ j)$

Input Size: Use number n
Basic Operation: moving a disk

Recurrence for Towers of Hanoi

- Let $M(n)$ = move count to compute $Towers(n, \cdot, \cdot)$.
- $M(1) = 1$ because 1 move is needed for $Towers(1, \cdot, \cdot)$.
- If $n > 1$, then $Towers(n, \cdot, \cdot)$ performs 2 recursive calls plus one move.

$$M(n) = \underbrace{2M(n-1)}_{\substack{\text{to compute} \\ Towers(n-1,\ \cdot,\ \cdot) \\ \text{twice}}} + \underbrace{1}_{\substack{\text{to move} \\ \text{disk } n}}$$

Solving the Towers of Hanoi Recurrence

Make a reasonable guess.

- Forward substitution. $M(2) = 2M(1) + 1 = 3$
  $M(3) = 2M(2) + 1 = 7$
  $M(4) = 2M(3) + 1 = 15$
- Backward substitution. $M(n) = 2M(n-1) + 1$
  $= 2[2M(n-2) + 1] + 1 = 4M(n-2) + 3$
  $= 4[2M(n-3) + 1] + 2 = 8M(n-3) + 7$

Prove $M(n) = 2^n - 1$ by mathematical induction.

- Basis: if $n = 1$, then $M(n) = 1 = 2^n - 1$
- Induction: if $M(n-1) = 2^{n-1} - 1$, then
  $M(n) = 2M(n-1) + 1 = 2 * (2^{n-1} - 1) + 1 = (2^n - 2) + 1 = 2^n - 1$

Number of Bits in an Integer

algorithm $BitCount(n)$
  // Input: A positive integer n
  // Output: The number of bits to encode n
  if $m = 1$ **then return** 1
  else return $BitCount(\lfloor n/2 \rfloor) + 1$

Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: division by 2

Recurrence for BitCount Function

- Let $D(n)$ = division count to compute $BitCount(n)$.
- $D(1) = 0$ because no divisions are performed to compute $BitCount(1)$.
- If $n > 1$, then $BitCount(n)$ performs recursive call on $\lfloor n/2 \rfloor$ plus one division.

$$D(n) = \underbrace{D(\lfloor n/2 \rfloor)}_{\substack{\text{to compute} \\ BitCount(\lfloor n/2 \rfloor)}} + \underbrace{1}_{\substack{\text{to compute} \\ \lfloor n/2 \rfloor}}$$

Solving the BitCount Recurrence

Make a reasonable guess using powers of 2.

- Forward substitution. $D(2) = D(1) + 1 = 1$
  $D(4) = D(2) + 1 = 2$
  $D(8) = D(4) + 1 = 3$
- Backward substitution. $D(n) = D(n/2) + 1$
  $= [D(n/4) + 1] + 1 = D(n/4) + 2$
  $= [D(n/8) + 1] + 2 = D(n/8) + 3$

Prove $D(2^k) = k$ by mathematical induction.

- Basis: if $k = 0$, then $D(2^k) = D(1) = 0 = k$
- Induction: if $D(2^{k-1}) = k - 1$, then $D(2^k) = D(2^{k-1}) + 1 = (k - 1) + 1 = k$

Fibonacci Function

algorithm $F(n)$
// Computes nth Fibonacci number recursively
// Input: A nonnegative integer n
// Output: The nth Fibonacci number
if $n = 0$ or $n = 1$ **then return** n
else return $F(n-1) + F(n-2)$

Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: addition in else statement

Recurrence for Fibonacci Function

- Let $A(n)$ = addition count to compute $F(n)$.
- $A(1) = A(0) = 0$ because no additions are performed to compute $F(0)$ or $F(1)$.
- If $n > 1$, then $F(n)$ performs two recursive calls plus one addition.

$$A(n) = \underbrace{A(n-1)}_{\text{to compute } F(n-1)} + \underbrace{A(n-2)}_{\text{to compute } F(n-2)} + \underbrace{1}_{\text{to add } F(n-1) \text{ and } F(n-2)}$$

Approximating the Fibonacci Recurrence

Make a reasonable guess at a lower bound.

- Forward substitution. $A(2) = 1$, $A(3) = 2$, $A(4) = 4$, $A(5) = 7$, $A(6) = 12$.
- Backward substitution.
  $A(n) = A(n-1) + A(n-2) + 1$
  $= [A(n-2) + A(n-3) + 1] + A(n-2) + 1$
  $= 2A(n-2) + A(n-3) + 2$

Prove $A(n) \geq 2^{n/2}$ when $n \geq 4$.

- Basis: True for $A(4)$ and $A(5)$.
- Induction: $A(n) = 2A(n-2) + A(n-3) + 2$
  $\geq 2A(n-2) \geq 2 * 2^{(n-2)/2} = 2^{n/2}$

Plan for Empirical Analysis of Algorithms

- Understand the experiment's purpose.
- Decide on the metric M and the measurement unit.
- Decide on characteristics of input.
- Prepare program implementing algorithm(s) and generating a sample of inputs.
- Run the algorithm(s) on the sample and record the data.
- Analyze the data.

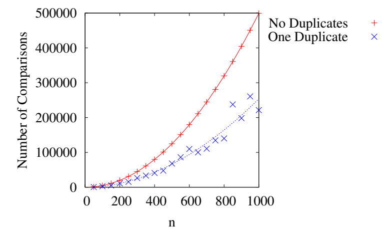
CS 3343 Analysis of Algorithms Chapter 2: Slide – 31

Empirical Analysis of UniqueElements

- Analyze *UniqueElements*, comparing arrays with unique elements vs. one duplicate.
- Measure number of comparisons on different input sizes, from 50 to 1000 by 50.
- For arrays with a duplicate, randomly choose a pair of positions that will have the same value. Also, run 10 times for each input size.
- Prepare `UniqueElementsExperiment.java`.
- Run program, record data, and create graph.
- Analyze the data.

CS 3343 Analysis of Algorithms Chapter 2: Slide – 32

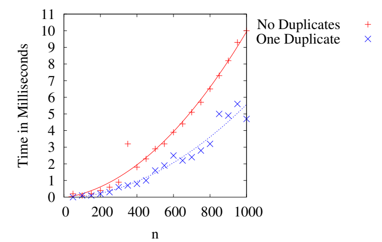
Results of Empirical Analysis: Comparisons



- No duplicates line is $n(n-1)/2$.
- One duplicate line is least squares fit, resulting in $0.24 * n^2 + 17 * n - 2042 \approx n^2/4$.
- Predict: Half the time with one duplicate.

CS 3343 Analysis of Algorithms Chapter 2: Slide – 33

Results of Empirical Analysis: Timings



- No duplicates and one duplicate lines are:
  $-0.037 + 0.0016 * n + 8.4 \times 10^{-6} * n^2$.
  $-0.073 + 0.00047 * n + 5.2 \times 10^{-6} * n^2$.

CS 3343 Analysis of Algorithms Chapter 2: Slide – 34

Algorithm Visualization

Visualization of Sorting Algorithms

http://homepages.dcc.ufmg.br/~dorgival/applets/SortingPoints/SortingPoints

http://maven.smith.edu/~thiebaut/java/sort/demo.html