Mathematical Analysis of Nonrecursive and Recursive Algorithms

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Reference: Chapter 2: Section 2.3 to 2.5 of Introduction to Design and Analysis of Algorithms 3rd Edition (Levitin, 2012)

Overview

Algorithm Analysis Framework

Analysis of Nonrecursive Algorithms

Analysis of Recursive Algorithms

Algorithm Analysis Framework

Time efficiency indicates how fast an algorithm in question runs.

What to Count and Consider

- Algorithm's efficiency as a function of the algorithms input size
- Efficiency is analyzed by determining $C(n)$, the number of repetitions of the **basic operation** as function of algorithm's input size n
- Estimated running time:

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

  where c_{op} is the execution time of the basic operation
- Worst-Case, Best-Case and Average-Case Efficiency Analyses

Overview

Algorithm Analysis Framework

Analysis of Nonrecursive Algorithms

Analysis of Recursive Algorithms

Analysis of Nonrecursive Algorithms

Plan for Analyzing Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter(s) indicating input's size.
2. Identify the algorithm's basic operation.
3. Determine the worst-case, average-case, and best-case efficiencies of the algorithm separately (if necessary).
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Analysis of Nonrecursive Algorithms

List of summation formulas and rules that are useful in analysis of algorithms.

1. $\sum_{i=l}^{u} 1 = u - l + 1$
2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
3. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
4. $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}, (a \neq 1)$
5. $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$
6. $\sum_{i=1}^{n} i^k \approx \frac{1}{k+1} n^{k+1}$
7. $\sum_{i=2}^{n} \log i \approx n \log n$
8. $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + \gamma$, where
   $\gamma \approx 0.5772\ldots$ (Euler's constant)

Analysis of Nonrecursive Algorithms

Sum Manipulation Rules

1. $\displaystyle\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$
2. $\displaystyle\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\displaystyle\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$ where $l \leq m < u$
4. $\displaystyle\sum_{i=l}^{u}(a_i - a_{i-1}) = a_u - a_{l-1}$

Analysis of Nonrecursive Algorithms

Example

Compute the following sums.

a. $1+3+5+7+\ldots+999$

b. $\sum_{i=3}^{n+1} 1$

c. $\sum_{j=0}^{n} 3^{j+1}$

Homework

Find the order of growth of $\sum_{i=2}^{n-1} \left(i^2+1\right)$. Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

Problem 1: Determining Sample Variance

The sample variance s^2 of n measurements $x_1, \ldots, x_n$ can be computed in two ways:

1. $s^2 = \dfrac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}$ where $\bar{x} = \dfrac{\sum_{i=1}^{n} x_i}{n}$
2. $s^2 = \dfrac{\sum_{i=1}^{n} x_i^2 - \dfrac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}$

Find and compare the number of divisions, multiplications, and additions/subtractions that are required for computing s^2 according to each of these formulas.

Problem 2: Maximal Element

Consider the following pseudocode of a standard algorithm for determining the largest element in a list of n numbers.

```
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
```

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Problem 3: Element Uniqueness Problem

Consider the algorithm below that checks whether all the elements in a given array are distinct.

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 $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Problem 3: Element Uniqueness Problem

For the average-case efficiency of the algorithm, let

- $X + 1 = {}_nC_2 + 1 = \dfrac{n(n-1)}{2} + 1$: no. of possible input groups
- $0 \leq p \leq 1$: probability that random input has repeated elements (so $1 - p$ is the probability that input has unique elements). Moreover, let $0 \leq p_i \leq 1$ be the probability that a random input with nonunique elements has repeated elements at the pair of indices (j, k), where $0 \leq j, k \leq n-1$, for $1 \leq i \leq X$ (**note:** groups i are ordered according to the no. of comparisons made after succesful search of first repeated element)

  Assume that p_i are the same for all input groups i so that $p_i = \dfrac{p}{X} = \dfrac{p}{{}_nC_2}$.
- $t_i = i$: no. of comparisons made when input comes from group i
- $t_{distinct} = X$: no. of comparisons made when array elements are distinct

Problem 3: Element Uniqueness Problem

Then, the average-case efficiency of the algorithm is

$$
\begin{aligned}
C_{ave}(n) &= \sum_{i=1}^{{}_nC_2} p_i t_i + (1-p)t_{distinct} = \frac{p}{{}_nC_2}\sum_{i=1}^{{}_nC_2} i \;+\; (1-p)\,{}_nC_2 \\
&= \frac{p}{{}_nC_2}\left(\frac{{}_nC_2({}_nC_2+1)}{2}\right) + (1-p)\frac{n(n-1)}{2} \\
&= \frac{p}{4}(n^2-n+2) + (1-p)\frac{n(n-1)}{2}.
\end{aligned}
$$

Problem 4: Matrix Multiplication Problem

Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$.

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
 //Multiplies two square matrices of order n by the definition-based algorithm
 //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**
   $C[i, j] \leftarrow 0.0$
   for $k \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
 return C

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Matrix Multiplication Problem

Remarks

1. The estimated running time of the algorithm on a particular machine is

$$T(n) \approx c_m M(n) = c_m n^3$$

   where c_m is the time of one multiplication on the machine.

2. A more accurate estimate is given by

$$T(n) \approx c_m M(n) + c_a A(n) = (c_m + c_a) n^3$$

   where it took into account the time spent on the additions and c_a is the time of one addition.

Problem 5: Counting Binary Digits Problem

Algorithm that finds the number of binary digits in the binary representation of a positive decimal number n:

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 $count$

- The loop's variable takes only a few values between its lower and upper limits.
- The number of comparisons is $C(n) = \lfloor \log_2 n \rfloor + 1$.

Overview

Algorithm Analysis Framework

Analysis of Nonrecursive Algorithms

Analysis of Recursive Algorithms

Analysis of Recursive Algorithms

General Plan for Analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Determine the worst-case, average-case, and best-case efficiencies of the algorithm separately (if necessary).
4. Set up a **recurrence relation**, with an appropriate **initial condition**, for the number of times the basic operation is executed.
5. Solve the recurrence OR *at least ascertain the order of growth of its solution*.

Factorial of a Number Problem

The following recursive algorithm computes the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n.

ALGORITHM $F(n)$

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

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Solving Recurrence Relations

- One way to define a sequence, say $\{x_n\}$, is using a **recurrence relation** and a given **initial condition**. Some examples are as follows:

1. $x(n) = x(n-1) + n$ when $n > 0$; $x(0) = 0$
2. $F(n) = F(n-1) + F(n-2)$ when $n > 2$; $F(0) = 0$ and $F(1) = 1$
3. $T(n) = 2T(n-1) + 1$ when $n > 1$; $T(1) = 1$

Methods for Solving Recurrence Relations

1. Method of Forward Substitutions
2. Method of Backward Substitutions
3. Linear k^{th}-Degree Recurrences with Constant Coefficients

Tower of Hanoi Problem

In this puzzle, there are n disks of different sizes and three pegs.

- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
- **Goal**: Move all the disks to the third peg, using the second one as an auxiliary, if necessary, subject to the following rules:
  - (i) You can move only one disk at a time.
  - (ii) It is forbidden to place a larger disk on top of a smaller one.

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Tower of Hanoi Problem

Solution Idea:

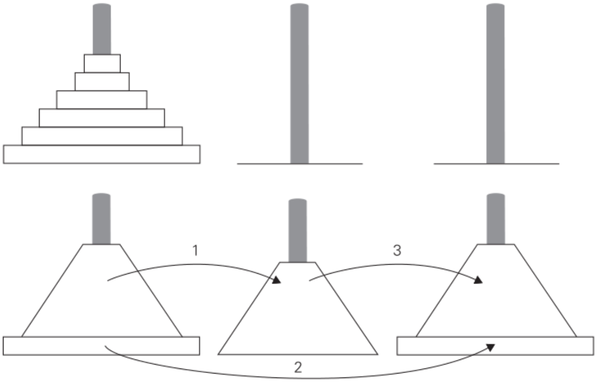
To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary)

- First, move recursively $n - 1$ disk from peg 1 to peg 2 (with peg 3 as auxiliary).
- Second, move the largest disk directly from peg 1 to peg 3.
- Third, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).

Note. When $n = 1$, simply move the single disk directly from the source peg to the destination peg. (*Base Case*)

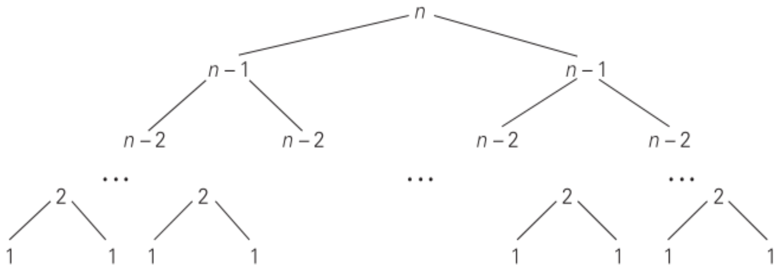
Tower of Hanoi Problem

Figure: Recursive Solution to the Tower of Hanoi



Tower of Hanoi Problem

Figure: Tree of Recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.



Counting Binary Digits Problem: Revisited

Below is a recursive algorithm that computes the number of binary digits in the number $n's$ binary representation.

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

"Solving" Recurrence Relations

- Analyzing algorithms $\Longrightarrow$ determine *asymptotic behavior*
- Rather than solving *exactly* the recurrence relation associated with the cost of an algorithm, it is sufficient to give an asymptotic characterization.

Theorem (Smoothness Rule)

Let $T(n)$ be an **eventually nondecreasing** function and $f(n)$ be a **smooth** function. If

$$T(n) \in \Theta(f(n)) \text{ for values of } n \text{ that are powers of } b,$$

where $b \geq 2$, then $T(n) \in \Theta(f(n))$.

(The analogous results hold for the cases of O and Ω as well.)

“Solving” Recurrence Relations

Theorem (Master Theorem)

Let $T(n)$ be an eventually nondecreasing function and $f(n)$ be a smooth function. If

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ for } n = b^k, 1, 2, \ldots \qquad \text{and} \qquad T(1) = c,$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

(Similar results hold for the O and Ω too.)

"Solving" Recurrence Relations

Remarks

1. The Master Theorem cannot be used if

   (i) $T(n)$ is not eventually nondecreasing, e.g. $T(n) = \sin(x)$

   (ii) $f(n)$ is not a polynomial, e.g., $T(n) = 2T\left(\frac{n}{2}\right) + 2^n$

   (iii) b cannot be expressed as a constant

2. The Master Method provides an estimate of the growth rate for recurrence relations.

“Solving” Recurrence Relations

Examples

Use the Master method to provide an estimate of the growth rate of the following recurrence relations.

1. $T(n) = T\left(\frac{n}{2}\right) + \frac{1}{2}n^2 + n$
2. $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} + 42$
3. $T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$

n^{th} Fibonacci Number Problem

- The Fibonacci numbers $F(n)$: `0,1,1,2,3,5,8,13,21,...`

$$\begin{cases} F(n) = F(n-1) + F(n-2) & \text{for } n > 1, \\ F(0) = 0, \;\; F(1) = 1. \end{cases}$$

- introduced by Leonardo Fibonacci in 1202 as a solution to a problem about the size of a rabbit population
- To find explicit solution $F(n)$, use technique in solving **homogeneous second-order linear recurrence with constant coefficients**

Recall. Homogeneous Linear Recurrence

Definition

Let $k \in \mathbb{Z}^+$ and $C_n, C_{n-1}, \ldots, C_{n-k}$ be real numbers such that C_n and C_{n-k} are nonzero. Then for $n \geq k \geq 0$

$$C_n a_n + C_{n-1} a_{n-1} + \ldots + C_{n-k} a_{n-k} = 0 \tag{1}$$

is a **homogeneous linear recurrence relation of order k**.

Remark. The linear recurrence relation of order k given in (1) is associated the equation

$$C_n r^k + C_{n-1} r_{k-1} + \ldots + C_{n-k} = 0.$$

called the *characteristic equation*.

Recall. Homogeneous Linear Recurrences

Theorem

Consider the *homogeneous linear recurrence relation of order k*

$$C_n a_n + C_{n-1} a_{n-1} + \ldots + C_{n-k} a_{n-k} = 0 \tag{2}$$

where $C_n, C_{n-1}, \ldots C_{n-k}$ are real constants such that C_n and C_{n-k} are nonzero. If the characteristic equation associated to (2) has **distinct roots** $r_1, r_2, \ldots r_k$, then the general solution of (2) is

$$a_n = c_1 r_1^n + c_2 r_2^n + \ldots + c_k r_k^n$$

where $c_1, c_2 \ldots, c_k$ are arbitrary constants.

n^{th} Fibonacci Number Problem

Below is a recursive algorithm for computing $F(n)$ following the recurrence for Fibonacci sequence.

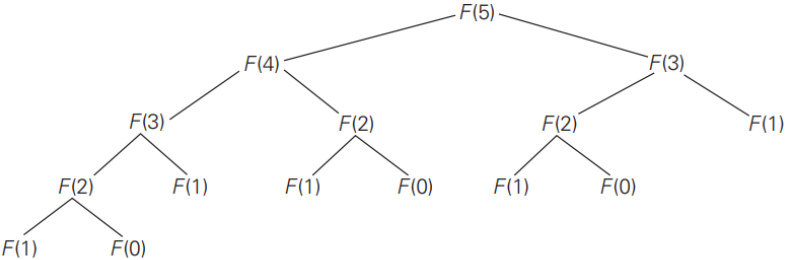
ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \leq 1$ **return** n
else return $F(n-1) + F(n-2)$

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

n^{th} Fibonacci Number Problem

Figure: Tree of Recursive calls for computing the $F(n)$ where $n = 5$ by the definition-based algorithm



n^{th} Fibonacci Number Problem

A much faster algorithm is given by simply computing the successive elements of the Fibonacci sequence *iteratively*.

ALGORITHM $Fib(n)$

//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
$F[0] \leftarrow 0;\ F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

n^{th} Fibonacci Number Problem

Remark

1. A third alternative for computing the n^{th} Fibonacci number:

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n \quad \textit{rounded to the nearest integer.}$$

The efficiency is determined by the exponentiation algorithm.

End of Lecture