CompSci 102
Discrete Math for Computer Science

February 14, 2012

Prof. Rodger

Slides modified from Rosen

Announcements

- Read for next time Chap. 4.4-4.6
- Finish Chapter 3 first, then start Chapter 4, number theory

Chap 3.3 - The Complexity of Algorithms

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size?
  - How much time does this algorithm use to solve a problem?
  - How much computer memory does this algorithm use to solve a problem?
- *time complexity* - analyze the time the algorithm uses to solve the problem given input of a particular size
- *space complexity* - analyze the computer memory the algorithm uses to solve the problem, given input of a particular size

The Complexity of Algorithms

- In this course, focus on time complexity.
- Measure time complexity in terms of the number of operations an algorithm uses
- Use big-*O* and big-Theta notation to estimate the time complexity
- Is it practical to use this algorithm to solve problems with input of a particular size?
- Compare the efficiency of different algorithms for solving the same problem.

Time Complexity

- For time complexity, determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.).
- Ignore minor details, such as the "house keeping" aspects of the algorithm.
- Focus on the *worst-case time* complexity of an algorithm. Provides an upper bound.
- More difficult to determine the *average case time complexity* of an algorithm (average number of operations over all inputs of a particular size)

Complexity Analysis of Algorithms

Example: Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

procedure *max*($a_1, a_2, \ldots, a_n$: integers)
    $max := a_1$
    for $i := 2$ to n
        if $max < a_i$ then $max := a_i$
    return *max*{*max* is the largest element}

Solution: Count the number of comparisons.

Complexity Analysis of Algorithms

Example: Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

procedure *max*($a_1, a_2, \ldots, a_n$: integers)
    $max := a_1$
    for $i := 2$ to n
        if $max < a_i$ then $max := a_i$
    return *max*{*max* is the largest element}

Solution: Count the number of comparisons.

- Compare $max < a_i$ $n-1$ times.
- when i incremented, compare if $i \leq n$. $n-1$ times
- One last comparison for $i > n$.
- $2(n-1)+1 = 2n-1$ comparisons are made.

Hence, the time complexity of the algorithm is $\Theta(n)$.

Worst-Case Complexity of Linear Search

procedure *linear search*(x:integer,
        $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
if $i \leq n$ **then** *location* := i
else *location* := 0
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

Solution: Count the number of comparisons.

-

Worst-Case Complexity of Linear Search

procedure *linear search*(x:integer,
$a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
  $i := i + 1$
if $i \leq n$ **then** *location* := i
else *location* := 0
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

Solution: Count the number of comparisons.

- At each step two comparisons are made; $i \leq n$ and $x \neq a_i$.
- end of loop, one comparison $i \leq n$ is made.
- After loop, one more $i \leq n$ comparison is made.

If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made. One comparison to exit loop.
Worst case $2n + 2$ comparisons, complexity is $\Theta(n)$.

Average-Case Complexity of Linear Search

Example: average case performance of linear search

Solution: Assume the element is in the list and that the possible positions are equally likely.

Average-Case Complexity of Linear Search

Example: average case performance of linear search

Solution: Assume the element is in the list and that the possible positions are equally likely.

By the argument on the previous slide, if $x = a_i$, the number of comparisons is $2i + 1$.

$$\frac{3+5+7+\ldots+(2n+1)}{n} = \frac{2(1+2+3+\ldots+n)+n}{n} = \frac{2\left[\frac{n(n+1)}{2}\right]}{n} + 1 = n + 2$$

Hence, the average-case complexity of linear search is $\Theta(n)$.

Worst-Case Complexity of Binary Search

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is the left endpoint of interval}
$j := n$ {j is right endpoint of interval}
while $i < j$
  $m := \lfloor (i + j)/2 \rfloor$
  if $x > a_m$ then $i := m + 1$
  else $j := m$
if $x = a_i$ **then** *location* := i
else *location* := 0
return *location*{location is the subscript i of the term a_i equal to x, or 0 if x is not found}

Solution: Assume $n = 2^k$ elements. Note that $k = \log n$.

Worst-Case Complexity of Binary Search

```
procedure binary search(x: integer, a_1,a_2,…, a_n: increasing integers)
 i := 1 {i is the left endpoint of interval}
 j := n {j is right endpoint of interval}
 while i < j
       m := ⌊(i + j)/2⌋
       if x > a_m then i := m + 1
       else j := m
  if x = a_i then location := i
  else location := 0
  return location{location is the subscript i of the term a_i  equal to x, or
 0 if x is not found}
```

Solution: Assume $n = 2^k$ elements. Note that $k = \log n$.

- Two comparisons are made at each stage; $i < j$, and $x > a_m$.
- Size of list is 2^k, then 2^{k-1}. then 2^{k-2} , … then $2^1 = 2$.
- At the last step, list size is $2^0 = 1$ and single last element compared.
- Hence, at most $2k + 2 = 2 \log n + 2$ comparisons are made.
- Therefore, the time complexity is $\Theta(\log n)$, better than linear search.

Worst-Case Complexity of Bubble Sort

```
procedure bubblesort(a_1,…,a_n: real numbers
                    with n ≥ 2)
   for i := 1 to n−1
      for j := 1 to n − i
           if a_j > a_{j+1} then interchange a_j and a_{j+1}
{a_1,…, a_n is now in increasing order}
```

Solution

Worst-Case Complexity of Bubble Sort

```
procedure bubblesort(a_1,…,a_n: real numbers
                    with n ≥ 2)
   for i := 1 to n−1
      for j := 1 to n − i
           if a_j > a_{j+1} then interchange a_j and a_{j+1}
{a_1,…, a_n is now in increasing order}
```

Solution: $n-1$ passes through list. pass $n - i$ comparisons

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

The worst-case complexity of bubble sort is $\Theta(n^2)$ since

$$\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Worst-Case Complexity of Insertion Sort

```
procedure insertion sort(a_1,…,a_n:
              real numbers with n ≥ 2)
   for j := 2 to n
      i := 1
      while a_j > a_i
          i := i + 1
       m := a_j
       for k := 0 to j − i − 1
           a_{j-k} := a_{j-k-1}
        a_i := m
```

Solution:

Worst-Case Complexity of Insertion Sort

```
procedure insertion sort(a_1,…,a_n:
        real numbers with n ≥ 2)
  for j := 2 to n
    i := 1
    while a_j > a_i
      i := i + 1
    m := a_j
    for k := 0 to j − i − 1
      a_{j-k} := a_{j-k-1}
    a_i := m
```

Solution: The total number of comparisons are:

$$2 + 3 + \dots + n = \frac{n(n+1)}{2} - 1$$

Therefore the complexity is $\Theta(n^2)$.

Matrix Multiplication Algorithm

- matrix multiplication algorithm; $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ where $\mathbf{C}$ is an $m \times n$ matrix that is the product of the $m \times k$ matrix $\mathbf{A}$ and the $k \times n$ matrix $\mathbf{B}$.

```
procedure matrix multiplication(A,B: matrices)
  for i := 1 to m
    for j := 1 to n
      c_ij := 0
      for q := 1 to k
        c_ij := c_ij + a_iq b_qj
return C{C = [c_ij] is the product of A and B}
```

$\mathbf{A} = [a_{ij}]$ is a $m \times k$ matrix

$\mathbf{B} = [b_{ij}]$ is a $k \times n$ matrix

Complexity of Matrix Multiplication

Example: How many additions of integers and multiplications of integers are used by the matrix multiplication algorithm to multiply two $n \times n$ matrices.

Solution

Complexity of Matrix Multiplication

Example: How many additions of integers and multiplications of integers are used by the matrix multiplication algorithm to multiply two $n \times n$ matrices.

Solution: There are n^2 entries in the product.

Each entry requires n mults and $n - 1$ adds.

Hence, n^3 mults and $n^2(n-1)$ adds.

matrix multiplication is $O(n^3)$.

Matrix-Chain Multiplication

- Compute *matrix-chain* $\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_n$ with fewest multiplications, where $\mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_n$ are $m_1 \times m_2$, $m_2 \times m_3$, $\cdots$ $m_n \times m_{n+1}$ integer matrices. Matrix multiplication is associative.

Example: In which order should the integer matrices $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$ - where $\mathbf{A}_1$ is 30×20, $\mathbf{A}_2$ 20×40, $\mathbf{A}_3$ 40×10 - be multiplied? **Solution**: two possible ways for $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$.

Matrix-Chain Multiplication

- Compute *matrix-chain* $\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_n$ with fewest multiplications, where $\mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_n$ are $m_1 \times m_2$, $m_2 \times m_3$, $\cdots$ $m_n \times m_{n+1}$ integer matrices. Matrix multiplication is associative.

Example: In which order should the integer matrices $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$ - where $\mathbf{A}_1$ is 30×20, $\mathbf{A}_2$ 20×40, $\mathbf{A}_3$ 40×10 - be multiplied? **Solution**: two possible ways for $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$.

- $\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)$: $\mathbf{A}_2\mathbf{A}_3$ takes $20 \cdot 40 \cdot 10 = 8000$ mults.. $\mathbf{A}_1$ by the 20×10 matrix $\mathbf{A}_2\mathbf{A}_3$ takes $30 \cdot 20 \cdot 10 = 6000$ mults. Total number is $8000 + 6000 = 14{,}000$.
- $(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3$: $\mathbf{A}_1\mathbf{A}_2$ takes $30 \cdot 20 \cdot 40 = 24{,}000$ mults. $\mathbf{A}_1\mathbf{A}_2$ by $\mathbf{A}_3$ takes $30 \cdot 40 \cdot 10 = 12{,}000$ mults. Total is $24{,}000 + 12{,}000 = 36{,}000$.

So the first method is best.

Understanding the Complexity of Algorithms

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Understanding the Complexity of Algorithms

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

Times of more than 10^{100} years are indicated with an *.

Complexity of Problems

- *Tractable Problem*: There exists a polynomial time algorithm to solve this problem. These problems are said to belong to the *Class P*.
- *Intractable Problem*: There does not exist a polynomial time algorithm to solve this problem
- *Unsolvable Problem* : No algorithm exists to solve this problem, e.g., halting problem.
- *Class NP*: Solution can be checked in polynomial time. But no polynomial time algorithm has been found for finding a solution to problems in this class.
- *NP Complete Class*: If you find a polynomial time algorithm for one member of the class, it can be used to solve all the problems in the class.

P Versus NP Problem

Stephen Cook
(Born 1939)

- The *P versus NP problem* asks whether the class P = NP? Are there problems whose solutions can be checked in polynomial time, but can not be solved in polynomial time?
  - Note that just because no one has found a polynomial time algorithm is different from showing that the problem can not be solved by a polynomial time algorithm.
- If a polynomial time algorithm for any of the problems in the NP complete class were found, then that algorithm could be used to obtain a polynomial time algorithm for every problem in the NP complete class.
  - Satisfiability (in Section 1.3) is an NP complete problem.
- It is generally believed that P≠NP since no one has been able to find a polynomial time algorithm for any of the problems in the NP complete class.
- The problem of P versus NP remains one of the most famous unsolved problems in mathematics (including theoretical computer science). The Clay Mathematics Institute has offered a prize of $1,000,000 for a solution.