Complexity of Algorithms

Section 3.3

Section Summary

- Time Complexity
- Worst-Case Complexity
- Algorithmic Paradigms
- Understanding the Complexity of Algorithms

The Complexity of Algorithms

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size? To answer this question, we ask:
  - How much time does this algorithm use to solve a problem?
  - How much computer memory does this algorithm use to solve a problem?
- When we analyze the time the algorithm uses to solve the problem given input of a particular size, we are studying the *time complexity* of the algorithm.
- When we analyze the computer memory the algorithm uses to solve the problem given input of a particular size, we are studying the *space complexity* of the algorithm.

The Complexity of Algorithms

- In this course, we focus on time complexity. The space complexity of algorithms is studied in later courses.
- We will measure time complexity in terms of the number of operations an algorithm uses and we will use big-*O* and big-Theta notation to estimate the time complexity.
- We can use this analysis to see whether it is practical to use this algorithm to solve problems with input of a particular size. We can also compare the efficiency of different algorithms for solving the same problem.
- We ignore implementation details (including the data structures used and both the hardware and software platforms) because it is extremely complicated to consider them.

Time Complexity

- To analyze the time complexity of algorithms, we determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.). We can estimate the time a computer may actually use to solve a problem using the amount of time required to do basic operations.
- We ignore minor details, such as the "house keeping" aspects of the algorithm.
- We will focus on the *worst-case time* complexity of an algorithm. This provides an upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size.
- It is usually much more difficult to determine the *average case time complexity* of an algorithm. This is the average number of operations an algorithm uses to solve a problem 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.

- The $max < a_i$ comparison is made $n - 2$ times.
- Each time i is incremented, a test is made to see if $i \leq n$.
- One last comparison determines that $i > n$.
- Exactly $2(n - 1) + 1 = 2n - 1$ comparisons are made.

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

Worst-Case Complexity of Linear Search

Example: Determine the time complexity of the linear search algorithm.

procedure *linear search*(x:integer,
$a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \le n$ and $x \ne a_i$)
$i := i + 1$
if $i \le n$ **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; $i \le n$ and $x \ne a_i$.
- To end the loop, one comparison $i \le n$ is made.
- After the loop, one more $i \le n$ comparison is made.

If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case $2n + 2$ comparisons are made. Hence, the complexity is $\Theta(n)$.

Average-Case Complexity of Linear Search

Example: Describe the average case performance of the linear search algorithm. (Although usually it is very difficult to determine average-case complexity, it is easy for 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 $2i + 1$.

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

$$\frac{2[\frac{n(n+1)}{2}]}{n} + 1 = n + 2$$

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

Worst-Case Complexity of Binary Search

Example: Describe the time complexity of binary search in terms of the number of comparisons used.

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 (for simplicity) $n = 2^k$ elements. Note that $k = \log n$.

- Two comparisons are made at each stage; $i < j$, and $x > a_m$.
- At the first iteration the size of the list is 2^k and after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$.
- At the last step, a comparison tells us that the size of the list is the size is $2^0 = 1$ and the element is compared with the single remaining element.
- Hence, at most $2k + 2 = 2 \log n + 2$ comparisons are made.
- Therefore, the time complexity is Θ ($\log n$), better than linear search.

Worst-Case Complexity of Bubble Sort

Example: What is the worst-case complexity of bubble sort in terms of the number of comparisons made?

procedure *bubblesort*($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  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}
{$a_1,\ldots, a_n$ is now in increasing order}

Solution: A sequence of $n-1$ passes is made through the list. On each pass $n - i$ comparisons are made.

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

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

Example: What is the worst-case complexity of insertion sort in terms of the number of comparisons made?

Solution: The total number of comparisons are:

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

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

procedure *insertion sort*($a_1,\ldots,a_n$:
real numbers with $n \geq 2$)
for $j := 2$ to n
$i := 1$
while $a_j > a_i$
$i := i + 1$
$m := a_j$
for $k := 0$ to $j - i - 1$
$a_{j-k} := a_{j-k-1}$
$a_i := m$

Matrix Multiplication Algorithm

- The definition for matrix multiplication can be expressed as an 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 $\mathbf{B}$.
- This algorithm carries out matrix multiplication based on its definition.

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{$\mathbf{C} = [c_{ij}]$ 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: There are n^2 entries in the product. Finding each entry requires n multiplications and $n - 1$ additions. Hence, n^3 multiplications and $n^2(n - 1)$ additions are used.

Hence, the complexity of matrix multiplication is $O(n^3)$.

Boolean Product Algorithm

- The definition of Boolean product of zero-one matrices can also be converted to an algorithm.

procedure *Boolean product*(**A**,**B**: zero-one matrices)
  for i := 1 to m
    for j := 1 to n
      $c_{ij} := 0$
      for q := 1 to k
        $c_{ij} := c_{ij} \vee (a_{iq} \wedge b_{qj})$
return C{**C** = [c_{ij}] is the Boolean product of **A** and **B**}

Complexity of Boolean Product Algorithm

Example: How many bit operations are used to find $\mathbf{A} \odot \mathbf{B}$, where A and B are $n \times n$ zero-one matrices?

Solution: There are n^2 entries in the $\mathbf{A} \odot \mathbf{B}$. A total of n Ors and n ANDs are used to find each entry. Hence, each entry takes 2n bit operations. A total of $2n^3$ operations are used.

Therefore the complexity is $O(n^3)$

Matrix-Chain Multiplication

- How should the *matrix-chain* $\mathbf{A}_1\mathbf{A}_2 \cdots \mathbf{A}_n$ be computed using the fewest multiplications of integers, 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 (exercise in Section 2.6).

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 to use the least number of multiplications.

Solution: There are two possible ways to compute $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$.

- $\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)$: $\mathbf{A}_2\mathbf{A}_3$ takes $20 \cdot 40 \cdot 10 = 8000$ multiplications. Then multiplying $\mathbf{A}_1$ by the 20×10 matrix $\mathbf{A}_2\mathbf{A}_3$ takes $30 \times 20 \cdot 10 = 6000$ multiplications. So the total number is $8000 + 6000 = 14{,}000$.

- $(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3$: $\mathbf{A}_1\mathbf{A}_2$ takes $30 \cdot 20 \cdot 40 = 24{,}000$ multiplications. Then multiplying the 30×40 matrix $\mathbf{A}_1\mathbf{A}_2$ by $\mathbf{A}_3$ takes $30 \cdot 40 \cdot 10 = 12{,}000$ multiplications. So the total number is $24{,}000 + 12{,}000 = 36{,}000$.

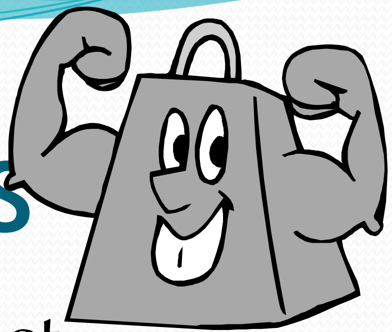
So the first method is best.

An efficient algorithm for finding the best order for matrix-chain multiplication can be based on the algorithmic paradigm known as *dynamic programming*. (see Ex. 57 in Section 8.1)

Algorithmic Paradigms

- An *algorithmic paradigm* is a a general approach based on a particular concept for constructing algorithms to solve a variety of problems.
  - Greedy algorithms were introduced in Section 3.1.
  - We discuss brute-force algorithms in this section.
  - We will see divide-and-conquer algorithms (Chapter 8), dynamic programming (Chapter 8), backtracking (Chapter 11), and probabilistic algorithms (Chapter 7). There are many other paradigms that you may see in later courses.

Brute-Force Algorithms



- A *brute-force* algorithm is solved in the most straightforward manner, without taking advantage of any ideas that can make the algorithm more efficient.
- Brute-force algorithms we have previously seen are sequential search, bubble sort, and insertion sort.

Computing the Closest Pair of Points by Brute-Force

Example: Construct a brute-force algorithm for finding the closest pair of points in a set of *n* points in the plane and provide a worst-case estimate of the number of arithmetic operations.

Solution: Recall that the distance between (x_i, y_i) and (x_j, y_j) is $\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$. A brute-force algorithm simply computes the distance between all pairs of points and picks the pair with the smallest distance.

Note: There is no need to compute the square root, since the square of the distance between two points is smallest when the distance is smallest.

Continued →

Computing the Closest Pair of Points by Brute-Force

- Algorithm for finding the closest pair in a set of n points.

```
procedure closest pair((x_1, y_1), (x_2, y_2), … ,(x_n, y_n): x_i, y_i real numbers)
min = ∞
   for i := 1 to n
      for j := 1 to i
           if (x_j − x_i)^2 + (y_j − y_i)^2 < min
             then  min := (x_j − x_i)^2 + (y_j − y_i)^2
                closest pair := (x_i, y_i), (x_j, y_j)
return closest pair
```

- The algorithm loops through $n(n-1)/2$ pairs of points, computes the value $(x_j - x_i)^2 + (y_j - y_i)^2$ and compares it with the minimum, etc. So, the algorithm uses $\Theta(n^2)$ arithmetic and comparison operations.
- We will develop an algorithm with $O(\log n)$ worst-case complexity in Section 8.3.

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.

Stephen Cook
(Born 1939)

P Versus NP Problem

- 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.