Algorithms

Chapter 3

With Question/Answer Animations

Chapter Summary

- Algorithms
  - Example Algorithms
  - Algorithmic Paradigms
- Growth of Functions
  - Big-O and other Notation
- Complexity of Algorithms

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 problem, algorithm and input of a particular size, how efficient is algorithm for solving problem?
- In particular
  1. how much time does algorithm use?
  2. how much computer memory?
- When we analyze
  1. time, we are finding *time complexity;*
  2. computer memory, we find its *space complexity*.

Our focus is *Time* Complexity

- In both 2440 and 2540, we focus on time complexity.

  (You may find space complexity treated in other courses.)
- We measure time complexity as #operations needed & use big-O, big-Θ notation.
- We may use this analysis to
  1. to see how practical it is to use algorithm with input of a particular size.
  2. compare efficiency of different algorithms for solving same problem.
- We ignore implementation details, e.g.,
  - data structures
  - hardware and software platforms

Time Complexity

- To analyze time complexity, we determine #operations, e.g.,
  - comparisons;
  - arithmetic operations (addition, multiplication, etc.).
- We ignore "house keeping" aspects.
- If we determine
  - #operations needed
  - time needed for basic operations
- then we could estimate actual time a computer needs.
- We focus on *worst-case* time complexity of an algorithm which provides an upper bound on #operations needed.
- On occasion we find *average-case* complexity:
  - average #operations used over all inputs of particular size.

Example: analysis of “max”

Describe time complexity of “max”, which finds the maximum element in a finite sequence.

procedure *max*(a_1, a_2, …., 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 #comparisons.

- The $max < a_i$ comparison is made $n - 1$ 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 $\Theta(n)$.

Example: Linear Search

Determine **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 #comparisons.

- If x is in list, then loop will exit early $\Rightarrow$ $\exists$fewer comparisons
- Thus, we assume that x is **not** in the list.
- For each pass of loop, 2 comparisons are made; $i \leq n$ and $x \neq a_i$.
- To exit loop, comparison $n + 1 \leq n$ is made.
- After the loop, one more $n + 1 \leq n$ comparison is made.

So in worst case, $2n + 2$ comparisons are made.
$\therefore$ complexity is $\Theta(n)$.

Example: Linear Search (cont.)

Determine **average-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: This time we assume that x is in list, say in j^{th} location, i.e., $x = a_j$

- For each pass of loop, 2 comparisons are made: $i \leq n$ and $x \neq a_i$.
- To exit loop, comparison $j \leq n$ (T) is made as well as $x \neq a_j$ (F).
- After the loop, one more $j \leq n$ comparison is made.

So $2j + 1$ comparisons are made. j could be any of 1,.., n and hence,

$$\text{ave} = \frac{1}{n}\sum_{j=1}^{n} 2j + 1 = \frac{1}{n}\left[2\left(\sum_{j=1}^{n} j\right) + n\right] = \frac{1}{n}\left[2\frac{n(n+1)}{2} + n\right] = n + 2$$

$\therefore$ average-case complexity is $\Theta(n)$. (For those don't like Σ, see next slide.)

Example: Linear Search (cont.)

- For those who do not like to work with Σ:

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

- From now on, “complexity” will mean

  “worst-case time complexity”

unless otherwise stated.

Example: Binary Search

Find complexity of binary search in terms of #comparisons.

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 moment that $n = 2^k$ elements. Note that $k = \log n$.

- Two comparisons are made for each pass: $i < j$, and $x > a_m$.
- Before 1st iteration, size of list is 2^k, then 2^{k-1} , 2^{k-2} & so on until size of list is $2^1 = 2$.
- Loop exits when |list| is $2^0 = 1$ & then x is compared with single remaining element.
- Hence, $2k + 2 = 2 \log n + 2$ comparisons are made.
- For non-powers of 2, #comparisons is $2\lceil \log n \rceil + 2$

$\therefore$ complexity is Θ ($\log n$), which is better than linear search.

Example: Bubble Sort

Find complexity of bubble sort in terms of #comparisons

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 list.
On i^{th} pass $n-i$ comparisons are made.
Hence #comparisons is

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

$\therefore$ complexity of bubble sort is $\Theta(n^2)$.

Example: Insertion Sort

Find complexity of insertion sort in terms of #comparisons

Solution: If we study # passes for 2 serially constructed while and for-loops inside the outer for-loop, we see that #comparisons is exactly j, the index of the outer loop.
Hence #comparisons in total is:

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

$\therefore$ 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; **C** = **A B** where **C** is an $m \times n$ matrix that is the product of the $m \times k$ matrix **A** and the $k \times n$ matrix **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{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

Example: Matrix Multiplication

Find complexity in terms of #arithmetic operations

Solution: Product of 2 $n \times n$ matrices is itself an $n \times n$ matrix and so has n^2 entries.

Finding an entry requires n mults & $n - 1$ additions.

[$(i,j)^{\text{th}}$ entry is the dot product of i^{th} row and j^{th} column.]

Hence, n^3 multiplications & $n^2(n-1)$ additions are used.

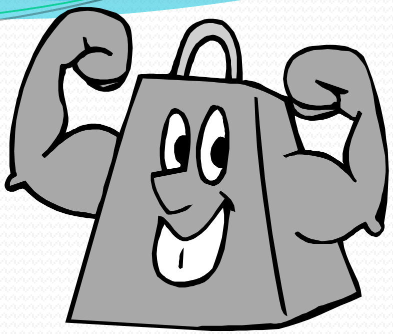
$\therefore$ complexity is $O(n^3)$.

Algorithmic Paradigms

An *algorithmic paradigm* is general approach based on particular concept for constructing algorithms to solve variety of problems.

- Greedy algorithms were introduced in Section 3.1.
- We discuss brute-force algorithms in this section.
- Elsewhere in text you can find:
  - probabilistic algorithms (Chapter 7)
  - divide-and-conquer algorithms (Chapter 8)
  - dynamic programming (Chapter 8)
  - backtracking (Chapter 11)
- $\exists$many other paradigms that you may see in other courses.

Brute-Force Paradigm



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:
  - sequential search
  - bubble and insertion sort

Example: Closest Pair of Points

Construct algorithm for finding closest pair of points in a set of n points & find complexity in terms of # operations.

Solution: Recall that distance betwn pts (x_i,y_i) & (x_j, y_j) is

$$\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$

Our brute-force algorithm computes distance between all pairs of points & picks pair with smallest distance.

Note: In our procedure, we do not compute the square root, since the square of the distance between two points is smallest when the distance is smallest.

Procedure and estimate →

Closest Pair of Points (cont.)

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

procedure *closest pair*($(x_1, y_1), (x_2, y_2), \ldots ,(x_n, y_n)$: 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 our algorithm uses $\Theta(n^2)$ arithmetic and comparison operations.
- For algorithm with $O(\log n)$ worst-case complexity, see Section 8.3.

Terminology for Complexity

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

Comparison table for Complexity

Problem Size	**Computer Time Used by Algorithms.**				
	Bit Operations Used				
n	$\log n$	n	$n \log n$	n^2	2^n
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} 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 *.
- We assume each operation takes 10^{-11}s = 0.01 nano s = 10 pico s
- 10^{11} yr is 100 billion yrs (BY). In contrast age of universe is 13.8 BY

Complexity of Problems

- *Tractable*: ∃polynomial time alg. to solve problem (*Class P*).
- *Intractable*: ∄polynomial time alg. to solve problem
- *Unsolvable*: ∄alg. to solve problem, e.g., halting problem.
- *Class NP*: Solution can be checked in polynomial time. But no polynomial time alg. has been found for finding solution.
- *NP Complete Class*: If you find polynomial time alg. for one member of class, it can be used to solve all problems in class.

P Versus NP Problem

Stephen Cook
(Born 1939)

P versus NP problem asks whether class $P \neq NP$?

- I.e., do $\exists$problems whose solutions can be *checked* in poly. time, but can not be *solved* in poly. time?
- If polynomial time algorithm for *any* problem 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 $P \neq NP$ since no one has been able to find a polynomial time algorithm for any problem in the NP complete class.
- P versus NP is the most famous **unsolved** problems in theoretical CS.
- **The Clay Math Institute has $1,000,000 prize for solution!**