CS 2305: Discrete Mathematics for Computing I

Lecture 21

- KP Bhat

Analogy for reasoning often used in Big-O problems

- Consider a weighing scale where on the one side you have an apple, a pear and a banana. On the other side you replace the apple by a bigger apple, the banana by a bigger banana and the pear by a bigger pear
- Obviously the first side of the scale will be lighter than the second side of the scale
- In our case we will look at an expression and replace smaller terms by larger terms. Obviously the original expression will be less than the resultant expression

Big-Omega Notation [1]

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are constants *C* and *k* such that

$$|f(x)| \geq C|g(x)|$$ when $x > k$.

We say that "*f*(*x*) is big-Omega of *g*(*x*)."

Big-*O* gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-Omega tells us that a function grows at least as fast as another.

Big-Omega Notation 2

Example: Show that $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$ where $g(x) = x^3$.

Solution: $f(x) = 8x^3 + 5x^2 + 7 \ \geq \ 8x^3$ for all positive real numbers *x*.

Big-Theta Notation [1]

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. The function

$$f(x) \text{ is } \Theta(g(x)) \text{ if } f(x) \text{ is } O(g(x)) \text{ and } f(x) \text{ is } \Omega(g(x)).$$

We say that "f is big-Theta of *g*(*x*)" and also that "*f*(*x*) is of *order* *g*(*x*)" and also that "*f*(*x*) and *g*(*x*) are of the *same order*."

$f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1, C_2 and *k* such that $C_1 g(x) < f(x) < C_2 g(x)$ if $x > k$. This follows from the definitions of big-*O* and big-Omega.

Big-Theta Notation 2

Example: Show that the sum of the first *n* positive integers is $\Theta(n^2)$.

Solution:

We have already shown that $f(n)$ is $O(n^2)$.

To show that $f(n)$ is $\Omega(n^2)$, we need a positive constant C such that $f(n) > Cn^2$ for sufficiently large n. Summing only the terms greater than $n/2$ we obtain the inequality

$$
\begin{aligned}
1+2+\cdots+n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\
&\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\
&= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\
&\geq (n/2)(n/2) = n^2/4
\end{aligned}
$$

Sum of n terms is greater than sum of ($n - \lceil n/2 \rceil + 1$) terms

Replacing each of the ($n - \lceil n/2 \rceil + 1$) terms in the range $\lceil n/2 \rceil$ through n by $\lceil n/2 \rceil$

Inequality holds for both even (e.g. $n = 2k$) and odd cases (e.g. $n = 2k + 1$)

Hence, $f(n)$ is $\Omega(n^2)$, and we can conclude that $f(n)$ is $\Theta(n^2)$.

Big-Theta Notation $_3$

Example: Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$.

Solution:

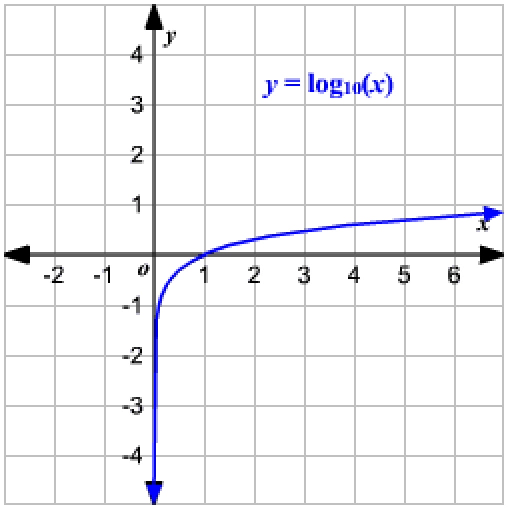
Part 1:

For x >1, log x is +ve and log x < x

$8x \log x < 8x^2$

$3x^2 + 8x \log x < 11x^2$

$\therefore 3x^2 + 8x \log x$ is $O(x^2)$



Part 2:

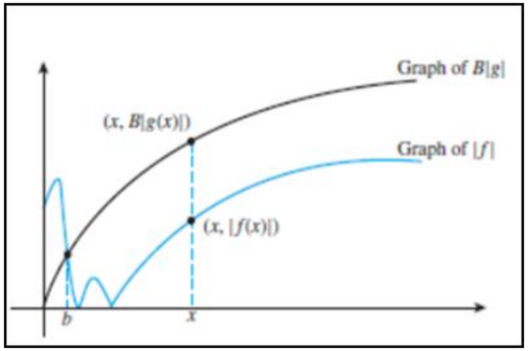
$3x^2 + 8x \log x > x^2$

$\therefore 3x^2 + 8x \log x$ is $\Omega(x^2)$

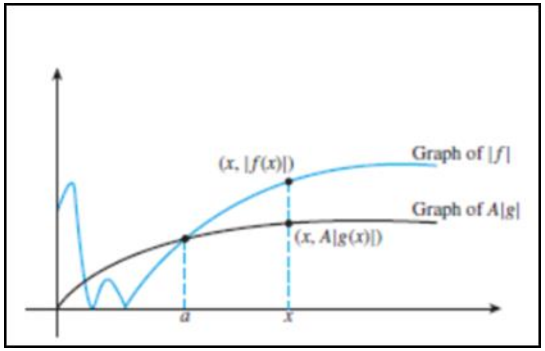
Hence $3x^2 + 8x \log x$ is $\Theta(x^2)$

Relating O, Ω and Θ[1]

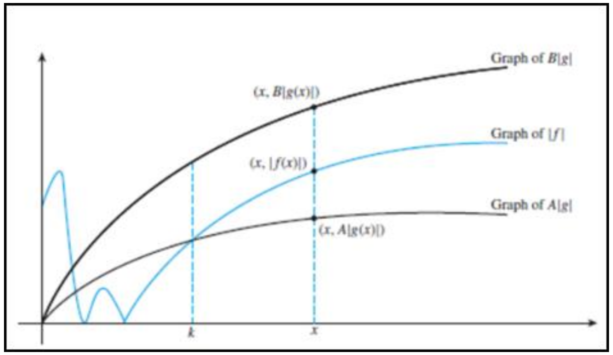
- If f(x) is O(g(x)) then f is of order at most g
  - g is the upper-bound for f
- If f(x) is Ω(g(x)) then f is of order at least g
  - g is the lower-bound for f
- If f(x) is Θ(g(x)) then f is of order g
  - f is bounded both above and below by some multiple of g



f(x) is O(g(x))

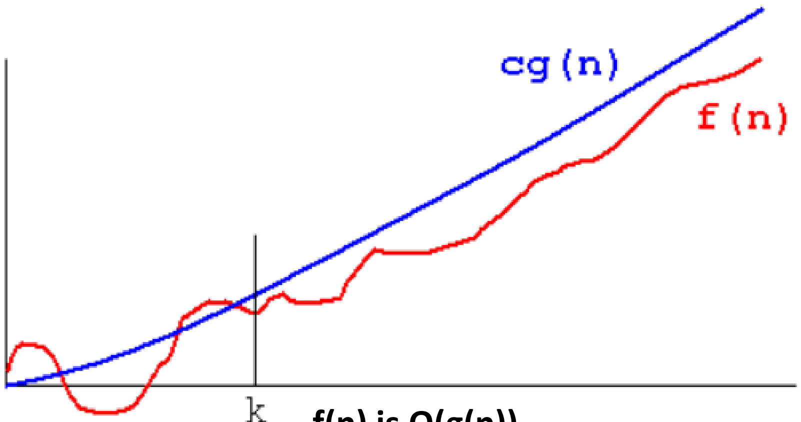


f(x) is Ω(g(x))

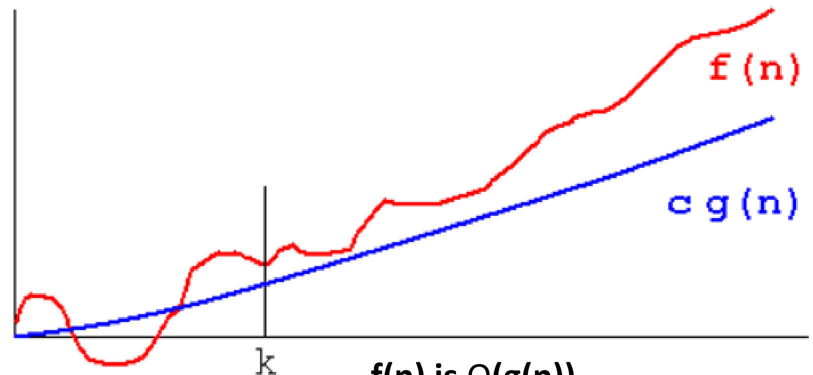


f(x) is Θ(g(x))

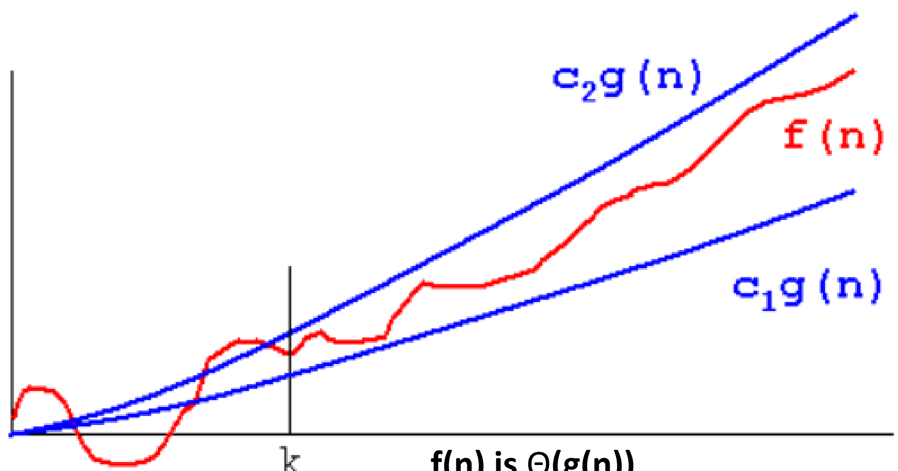
Relating O, Ω and Θ 2



f(n) is O(g(n))



f(n) is Ω(g(n))



f(n) is Θ(g(n))

Big-Theta Notation 4

When $f(x)$ is $\Theta(g(x))$ it must also be the case that

$g(x)$ is $\Theta(f(x))$.

Note that $f(x)$ is $\Theta(g(x))$ if and only if it is the case that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

Note: Witnesses might be different in the two cases

Sometimes writers are careless and write as if big-*O* notation has the same meaning as big-Theta.

Big-Theta Estimates for Polynomials[1]

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$
where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is of order x^n (or $\Theta(x^n)$).

Example:

The polynomial $f(x) = 8x^5 + 5x^2 + 10$ is order of x^5 (or $\Theta(x^5)$).

The polynomial $f(x) = 8x^{199} + 7x^{100} + x^{99} + 5x^2 25$
is order of x^{199} (or $\Theta(x^{199})$).

Big-Theta Estimates for Polynomials[2]

Not on the exam

Proof

We need to show inequalities both ways. First, we show that $|f(x)| \le Cx^n$ for all $x \ge 1$, as follows, noting that $x^i \le x^n$ for such values of x whenever $i < n$. We have the following inequalities, where M is the largest of the absolute values of the coefficients and C is $M(n+1)$:

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0| \\
&\le |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0| \\
&\le |a_n|x^n + |a_{n-1}|x^n + \cdots + |a_1|x^n + |a_0|x^n \\
&\le Mx^n + Mx^n + \cdots + Mx^n + Mx^n = Cx^n
\end{aligned}
$$

For the other direction, which is a little messier, let k be chosen larger than 1 and larger than $2nm/|a_n|$, where m is the largest of the absolute values of the a_i's for $i < n$. Then each a_{n-i}/x^i will be smaller than $|a_n|/2n$ in absolute value for all $x > k$. Now we have for all $x > k$,

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0| \\
&= x^n \left| a_n + \frac{a_{n-1}}{x} + \cdots + \frac{a_1}{x^{n-1}} + \frac{a_0}{x^n} \right| \\
&\ge x^n \, |a_n/2| \, ,
\end{aligned}
$$

Complexity of Algorithms

Short Overview

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.

Space Complexity

To analyze the space complexity we often consider the "extra" memory needed i.e. not counting the memory needed to store the input itself (e.g. pointers, keys, scratch buffers for ex-situ processing etc.).

A big focus in space complexity is analyzing the memory requirements for data structures employed for processing the data. For recursive algorithms we also need to analyze the recursion stack space.

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 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, …., 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 $n - 1$ 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 $\Theta(n)$.

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 ($i \leq n$ 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 $x \neq a_i$.
- To end the loop, one comparison $i \leq n$ is made.
- After the loop, one more $i \leq n$ comparison is made.

In the worst case (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 $\Theta(n)$.

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 $\Theta(\log n)$, 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,…,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: 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 insertion sort inserts the j^{th} element into the correct position among the first $j - 1$ elements that have already been put into the correct order.
In the worst case, j comparisons are required to insert the jth element into the correct position.

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 $j - i - 1$
    $a_{j-k} := a_{j-k-1}$
  $a_i := m$

Time Complexity of Some Algorithms

- Linear search: $\Theta(n)$
- Binary search: $\Theta(\log n)$
- Bubble Sort: $\Theta(n^2)$
- Selection Sort: $\Theta(n^2)$
- Insertion Sort: $\Theta(n^2)$
- Matrix Multiplication $_{(n \times n)}$: $\Theta(n^3)$

Algorithmic Paradigms (1)

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

Not on the exam

- Brute-force algorithms
  - naive approach for solving problems; does not take advantage of any special structure of the problem or clever ideas
    - sequential search, bubble sort, selection sort, insertion sort, etc.
- Greedy algorithms
  - select the best choice at each step, instead of considering all sequences of steps; focus on local optimization and not overall optimization
- Divide-and-conquer algorithms
  - *divide* a problem into one or more instances of the same problem of smaller size and *conquer* the problem by using the solutions of the smaller problems to find a solution of the original problem
    - Quick sort, merge sort, fast matrix multiplication, etc.

Algorithmic Paradigms (2)

- Dynamic programming
  - recursively breaks down a problem into simpler overlapping subproblems, and stores the results of the subproblems in a table; computes the solution of the problem using the solutions of the subproblems

    Not on the exam

    - cutting stock, matrix-chain multiplication, longest common subsequence, etc.
- Backtracking
  - performs an exhaustive search of all possible solutions; once it is known that no solution can result from any further sequence of decisions, backtrack to a known point and work towards a solution with another series of decisions
    - n-Queens, graph coloring, sum of subsets, etc.
- Probabilistic algorithms
  - make random choices at one or more steps
    - Monte Carlo algorithms, Las Vegas algorithms, Sherwood algorithms, etc.
- …

Understanding the Complexity of Algorithms [1]

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 2

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

$P \stackrel{?}{=} NP$

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.

Some Well Known NP Complete Problems

- Boolean satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem (decision version)
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

Not on the exam