Computational Complexity/ Tractable and Intractable Problems

ELEC 5402

Pavan Gunupudi

Dept. of Electronics, Carleton University

January 6, 2012

Computational Complexity

- Main concern of an algorithm is how fast it will run and how much memory it will take
- Theory of computational complexity helps us compare complexity of algorithms
- It characterizes the *rate of growth* of an algorithm with respect to (w.r.t) its *input size* (n)
- Input size or problem size is related to the number of symbols necessary to describe an input

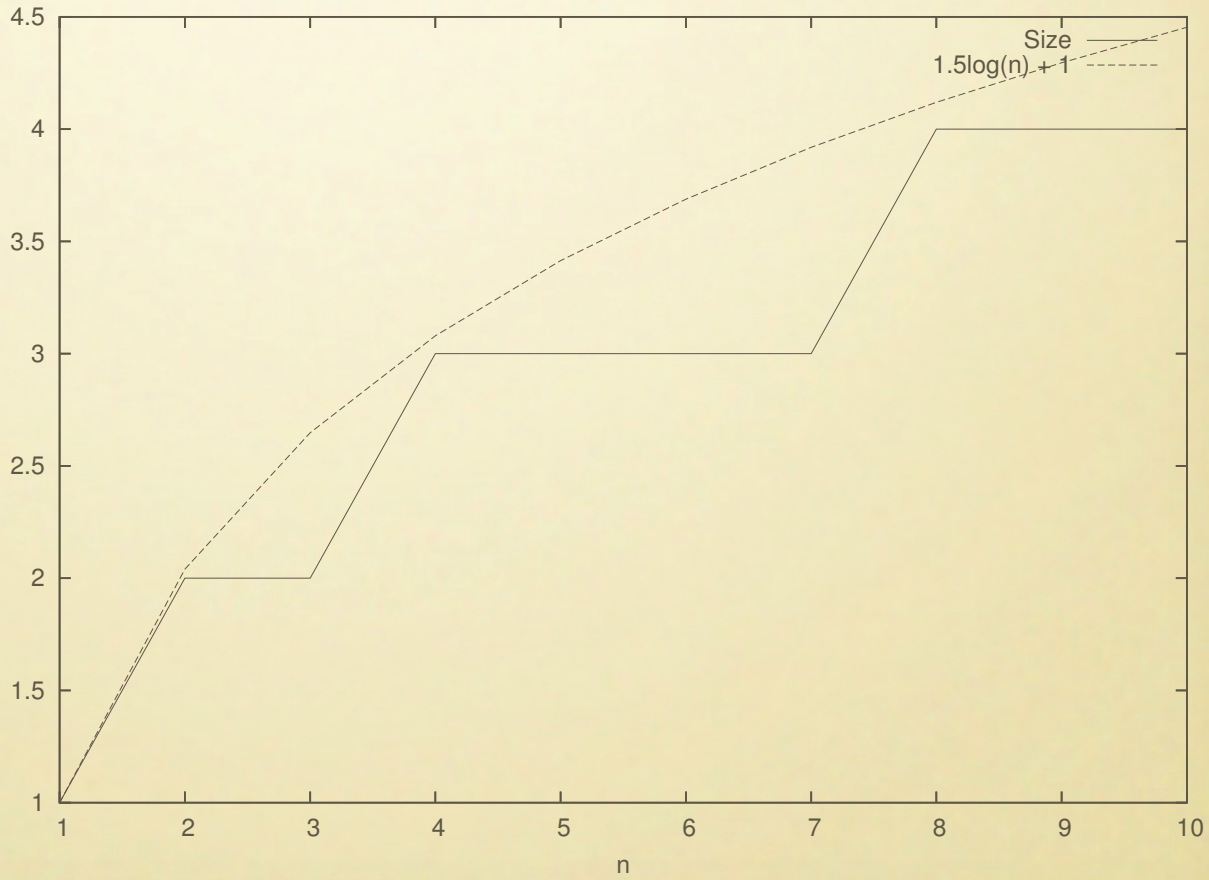
Example

- Sort list of n words; each with at most 10 letters
- Input size is bounded by 10n; if you include 8 bits for a letter we get 80n
- The constant is usually a function of the implementation: so we eliminate it
- The input size is said to be n for this algorithm

Example

- An algorithm takes input as a single natural number n using binary encoding

n	Bin	Siz
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4
10	1010	4



Time/Space Complexity

- Two types of computational complexity
  - Time complexity
  - Space complexity
- Space complexity given less importance than time complexity
- Caveat: An algorithm requiring more memory than required will not run at all
- Most comments made for time complexity also hold for space complexity
- For the rest of the course when we use the word "complexity" we refer to time complexity

Order of a function

Given two functions f and g that map a natural number n to a positive value

$f(n) = \mathcal{O}(g(n))$[1] if a K can be found such that

$$\forall n \geq n_0 : f(n) \leq K g(n)$$

The $=$ sign indicates set membership

[1] pronounced as "$f(n)$ is big O of $g(n)$" or as "$f(n)$ is of order at most $g(n)$"

Order.. Examples

Examples:

$$\frac{1}{3}n^2 = \mathcal{O}(n^2) \tag{1}$$

$$0.02n^2 + 127n + 1923 = \mathcal{O}(n^2) \tag{2}$$

$$3n \log n + n = \mathcal{O}(n \log n) \tag{3}$$

$$5n^3 = \mathcal{O}(\frac{1}{5}n^3) \tag{4}$$

$$5n^3 = \mathcal{O}(n^4) \tag{5}$$

$$5n^3 = \mathcal{O}(n^3 + n^2) \tag{6}$$

Other notations of bounds

Lower bound, big-Omega is used: $f(n) = \Omega(g(n))$ [1] if $g(n) = \mathcal{O}(f(n))$

Two functions have same rate of growth: $f(n) = \Theta(g(n))$ [2] if $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$

Generally big O is enough to describe computational complexity

[1] pronounced as "$f(n)$ is big Omega of $g(n)$" or as "$f(n)$ is of order at least $g(n)$"

[2] pronounced as "$f(n)$ is big Theta of $g(n)$" or as "$f(n)$ is of order exactly $g(n)$"

Elementary Computational Steps

- The big O does not give an idea of the time duration of the computation
- It just gives how the durations scales with input size
- The duration of a computation is expressed in terms of *elementary computational steps*
- Elementary computational step is a simple computation that is independent of input size e.g. multiplication of two 16 bit integers

Time Complexity

- Time complexity is a function that gives the number of elementary computational steps executed for inputs of a specific size e.g. number of floating point operations
- This can vary in an algorithm due to conditional branches
- *Worst-case time complexity* has to be considered
- *Average-case time complexity* can also be computed; it is very useful; however complicated to evaluate
- We consider worst case time complexity in general

Types of Complexity

- Linear vs quadratic
  - Say you have a circuit with a million nodes
  - Linear order methods are definitely better than quadratic order methods
- Sublinear order
  - Sometimes we may not have to process all elements to get the result we want; this leads to algorithms with sublinear order
  - e.g. Finding an element in a sorted list - $\mathcal{O}(\log n)$
  - Algorithm is independent of input size - $\mathcal{O}(1)$
- Polynomial order vs. exponential order
  - Exponential grows very fast w.r.t polynomial order
  - Defines tractable from intractable problems

Time Complexity of a Top-Level Algorithm

- Time complexities of entire algorithm are sum of time complexities of each part
- For conditional statements such as 'if'; complexity is sum of both the 'then' and 'else' part
- Some rules for multiplication and addition

$$f_1 = \mathcal{O}(g_1) \quad f_2 = \mathcal{O}(g_2)$$

$$f_1 \times f_2 = \mathcal{O}(g_1 \times g_2)$$

$$f_1 + f_2 = \mathcal{O}(g_1 + g_2)$$

- For addition it is followed by a simplification if g_1 or g_2 is of a lower order than the other

Tractable and Intractable problems

- Algorithms solvable in polynomial time complexity $\mathcal{O}(n^k)$ are called *tractable*
- If an algorithm cannot be solved with polynomial time complexity it is called *intractable*.

Nondeterministic computer

- *Nondeterministic computer:* A computer that allows for specification of multiple computations at a certain point in the program: the computer will make a *nondeterministic* choice on which of them to perform.
- This is not a *random choice* but that which will lead to the desired answer
- Another way to look at it: At the point where choice is to be made:
  - The computer splits itself into the number of choices present
  - Evaluates all choices in parallel
  - Then merges back into one machine choosing the correct choice to be picked
- Intuitively speaking, the nondeterministic computer has the ability to *guess* the right choice in constant time

Complexity Classes

- An algorithm that can be solved in a polynomial time complexity on a *deterministic computer* belongs to the class P (abbreviation for 'polynomial')
- An algorithm that can be solved in a polynomial time complexity on a *nondeterministic computer* belongs to the class NP (abbreviation for 'nondeterministic polynomial')
- Anything that can be solved in polynomial time on a deterministic computer can be solved in polynomial time on a nondeterministic computer
- Hence, $P \subset NP$

NP-Completeness

- Within NP, there is a special class called NPC (NP-Complete)
- Any algorithm in NPC can be reduced to another problem in NPC with polynomial time complexity
- If even one algorithm in NPC can be found to be solved in polynomial time complexity then all algorithms can be solved in polynomial time complexity
- If above is true: $NPC = P$ will hold
- Till now no one had found even one algorithm that is both in P and NPC

NP-Hard

- Algorithms with complexity at least as much as an NP-Complete problem are called *NP-hard*
- NP-complete problems form a lower bound in complexity for NP-hard problems
- Most problems in CAD for VLSI are NP-hard or NP-Complete

Classification of Decision Problems