RAMAIAH
Institute of Technology

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of CSE (AIML) and CSE (Cyber Security)

Course Name: Design and Analysis of Algorithms
Course Code: CI43
Credits: 3:0:0
UNIT 1

Reference:
Jon Kleinberg and Eva Tardos
Algorithm Design, Pearson (1st Edition),2013

RAMAIAH
Institute of Technology

Basics of Algorithm Analysis

- What is an algorithm?
  - An algorithm is a *set of steps to complete a task*.
  - It provides a step-by-step procedure that converts an input into a desired output.
  - Example - Task: to make a cup of tea.
    - An algorithm for the above task goes as follows:
      1. Add water and milk to the kettle,
      2. Boil it, add tea leaves,
      3. Add sugar, and then serve it in a cup.
  - So, the above algorithm typically follows a logical structure, where it receives few inputs (I/P data), performs a series of operations (Processing), and produces the desired result (O/P data).

RAMAIAH Institute of Technology

Basics of Algorithm Analysis

- What is a Computer Algorithm?
  - A set of steps to accomplish or complete a task that is, described precisely enough, such that a computer can run it.
  - described precisely enough – In the previous task example (Tea making algorithm), it is very *difficult for a machine to know how much water or milk needs to be added*.
  - Similarly, there are several computer algorithm examples as well.
    - GPS in our smartphones - uses ***Shortest Path Algorithm***
    - Secure online shopping applications/websites - uses a technique known as Cryptography that employs the ***RSA Algorithm***
  - So, algorithms run on computers or computational devices **should be described precisely** with a focus on
    1. Solving problems
    2. Optimizing solutions
    3. Automating tasks

RAMAIAH
Institute of Technology

Basics of Algorithm Analysis

- What should be the characteristics of a Computer Algorithm?
  - It must take input (one or more)
  - It must produce an output / result
  - An algorithm should possess the properties of
    1. Definiteness – Instructions in an algorithm should be clear and unambiguous
    2. Finiteness – An algorithm should terminate after a finite number of steps
    3. Effectiveness – Every instruction in an algorithm must be basic i.e, simple instruction
    4. Efficiency – How much computational resources (time and space) an algorithm uses to accomplish a task.

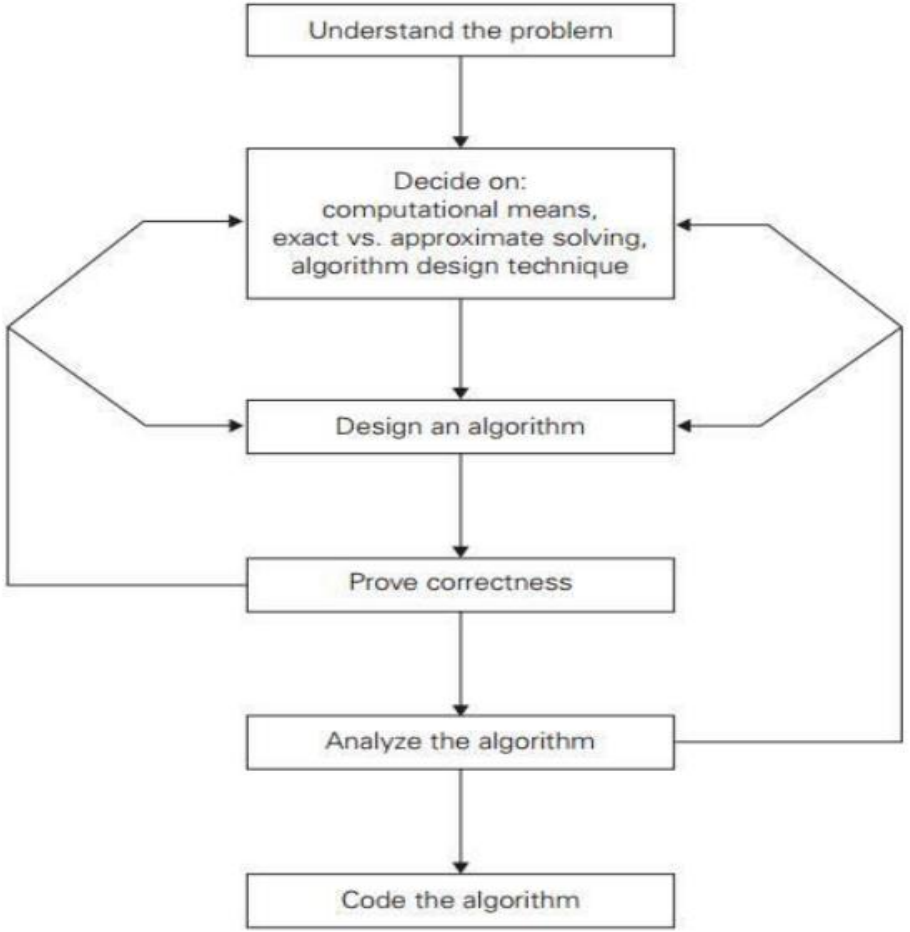
RAMAIAH
Institute of Technology

Basics of Algorithm Analysis

- What is meant by Design and Analysis of Algorithms?
  - Analysis of Algorithms is the *process of evaluating the efficiency of algorithms*, focusing mainly on the *time and space complexity.*
  - Analyzing an algorithm helps in evaluating
    - how the algorithm's running time or space requirements grow as the size of input increases.
    - The efficiency of an algorithm is measured in terms of **TIME** and **SPACE or MEMORY**.
  - Why Analysis of the Algorithm important?
    - It is an essential part of the **computational complexity theory** - provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem
    - It is essential for determining the amount of time and space resources required to execute a problem statement or task.

Basics of Algorithm Analysis

- This figure depicts the fundamentals of algorithmic problem solving.
- It describes the sequence of steps in the process of design and analysis of algorithms.



RAMAIAH
Institute of Technology

Basics of Algorithm Analysis

1. **Understanding the problem**
   - Ask questions, do a few small examples by hand, think about special cases, etc.
   - An input is an instance of the problem the algorithm solves
   - Specify exactly the set of instances the algorithm needs to handle
   - Decide on
   - Exact vs. approximate solution
   - Approximate algorithm: Cannot solve exactly, e.g., extracting square roots,
   - solving nonlinear equations, etc.
   - Appropriate Data Structure

Basics of Algorithm Analysis

2. Analyze the Algorithm

- *Time efficiency:* How fast it runs?
- *Space efficiency:* How much extra memory it uses?
- *Simplicity:* Easier to understand, usually contains fewer bugs, sometimes simpler is more efficient, but not always.

3. Coding algorithm

- Write in a programming language for a real machine
- Standard tricks:
  - Compute loop invariant (which does not change value in the loop) outside loop
  - Replace expensive operation by cheap ones.

Why do we need to analyze an algorithm?

2. Analyze the Algorithm - Reasons

i. Predict Performance

ii. Compare Algorithms

iii. Provide Guarantees

iv. Understand theoretical basis.

v. *Primary Practical Reason:* Avoid Performance Bugs

Basics of Algorithm Analysis

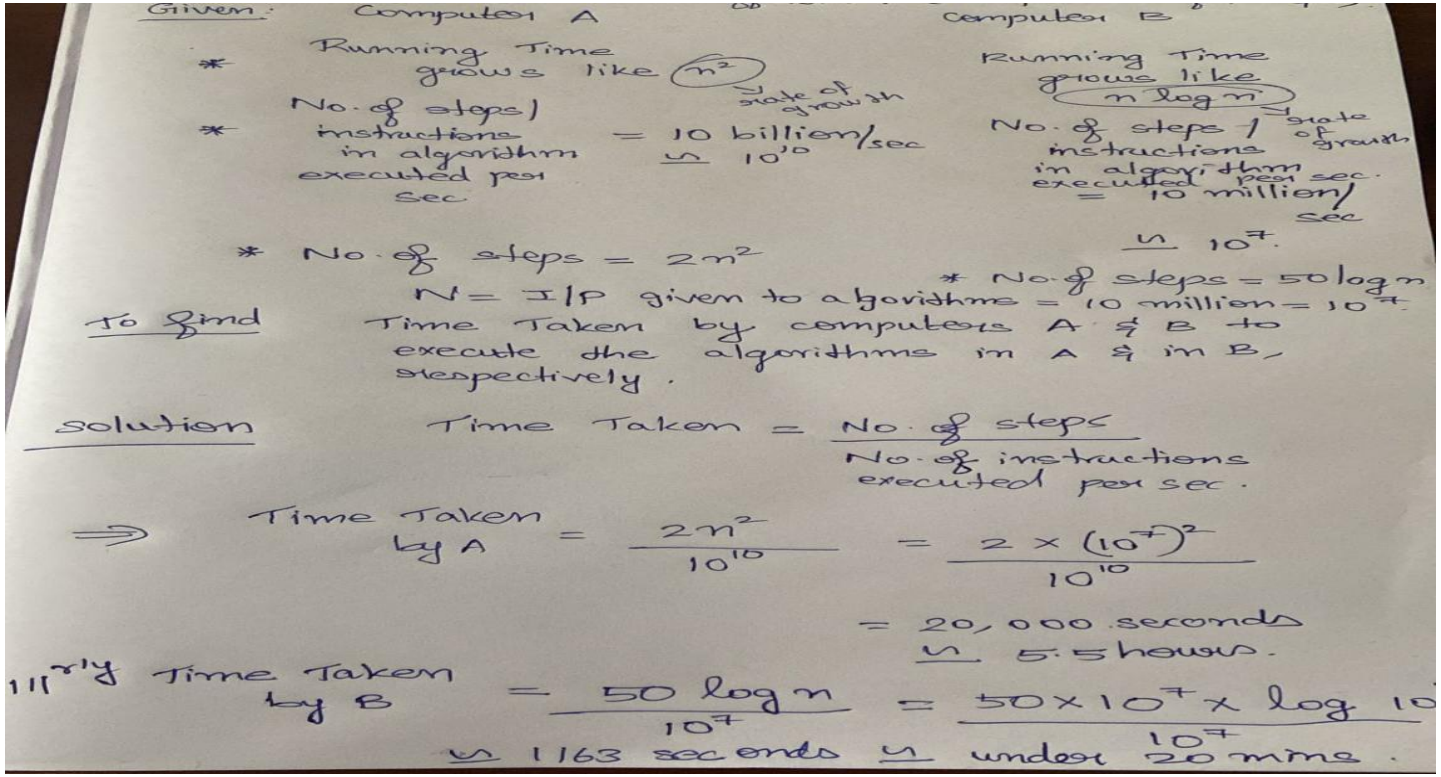
Example Scenario – compute the time taken and conclude the efficiency. Provide justifications for the conclusions drawn

A faster computer(computer A) running a sorting algorithm whose running time on n values grows like $\mathbf{n^2}$ against a slower computer (computer B) running a sorting algorithm whose running time grows like **nlogn**. They each **must sort an array of 10 million numbers**. Suppose that computer A executes **10 billion instructions per second** (faster than any single sequential computer at the time of this writing) and computer B executes only **10 million instructions per second**, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer writes for computer B, using a high-level language with an inefficient compiler, with the resulting code taking 50 nlog n instructions.

Basics of Algorithm Analysis

Example Scenario – compute the time taken and conclude the efficiency

Solution:



Unit I

Computational Tractability

Some Initial Attempts at Defining Efficiency

Worst-case Running Times and Brute-Force Search

Polynomial Time as a Definition of Efficiency

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates, Asymptotic Bounds for Some Common Functions

A Survey of Common Running Times – Linear Time, O(n log n) Time, $O(n^k)$ Time, Beyond Polynomial Time

Some Representative Problems –

A First Problem: Stable Matching

RAMAIAH
Institute of Technology

Performance Measures of the Algorithm

- Two kinds of efficiency:

1. Space Efficiency or Space Complexity

2. Time Efficiency or Time Complexity

Two kinds of Algorithm Efficiency

- Analyzing the efficiency of an algorithm (or the complexity of an algorithm) means establishing the amount of computing resources needed to execute the algorithm.
- There are two types of resources:
  - Memory space. It means the amount of space used to store all data processed by the algorithm.
  - Running time. It means the time needed to execute all the operations specified in the algorithm.
- Space efficiency: Deals with the space required by the algorithm
- Time efficiency: It indicates how fast an algorithm runs.

What is space complexity?

- For any algorithm, memory is required for the following purposes
  - Memory required to store program instructions
  - Memory required to store constant values
  - Memory required to store variable values

- Space complexity of an algorithm can be defined as follows
  - Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

What is space complexity?

- Generally, when a program is under execution it uses the computer memory for THREE reasons.
- They are:

  i. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.

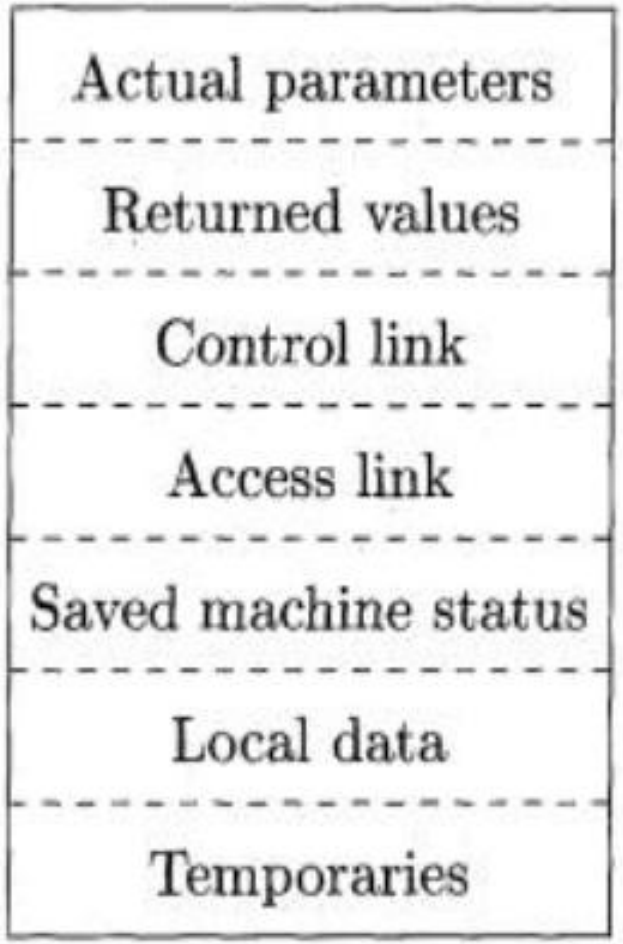
  ii. **Data Space:** It is the amount of memory used to store all the variables and constants.

  iii. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.

Space Complexity = Instruction space + Data space + Stack space

What is stack space or stack allocation of space?

- **Temporary values**, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
- A **saved machine status**, with information about the state of the machine just before the call to the procedure. This information typically includes the return address (value of the program counter, to which the called procedure must return).
- An **access link** may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
- A **control link**, pointing to the activation record of the caller.
- The **actual parameters** used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency.



Calculating Space Complexity

- To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler).
- For example, the **C Programming Language compiler** requires the following:
    1. 1 byte to store Character value,
    2. 2 bytes to store Integer value,
    3. 4 bytes to store Floating Point value,
    4. 6 or 8 bytes to store double value

Calculating Space Complexity

- Calculating the Data Space required for the following given code

```
int square(int a)
{
return a*a;
}
```

Data Space Required:

For int a -> 2 Bytes

For returning a*a -> 2 Bytes

Total 4 Bytes

Calculating Space Complexity

- Data Space Required:
  - This code requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.
  - That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be Constant Space Complexity.
  - If any algorithm requires a fixed amount of space for all input values, then that space complexity is said to be **Constant Space Complexity**.

What is Time Complexity?

- Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.
- Time complexity of an algorithm can be defined as follows:
  - The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution
- Generally, running time of an algorithm depends upon the following:
  - ✓ Whether it is running on Single processor machine or Multi processor machine.
  - ✓ Whether it is a 32 bit machine or 64 bit machine
  - ✓ Read and Write speed of the machine.
  - ✓ The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
  - ✓ Input data

Calculating Time Complexity

	Cost or Number Operations in the Statement
int sum = 0, i;	1 (initializing zero to sum)
for(i = 0; i < n; i++)	1+1+1 (i=0, i<n, i++)
sum = sum + A[i];	1+ 1 (Addition and Assigning result to sum)
return sum;	1 (returning sum)

Calculating Time Complexity

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**
int sum = 0, i;	1	1
for(i = 0; i < n; i++)	1+1+1	1+(n+1)+n (i=0 gets executed one time, i<n gets executed (n+1) times, i++ gets executed n times
sum = sum + A[i];	1+ 1	
return sum;	1	

Calculating Time Complexity

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1+1+1	1+(n+1)+n	2n+2
sum = sum + A[i];	1+ 1	n + n	2n
return sum;	1	1	1
		Running Time T(n)	4n+4

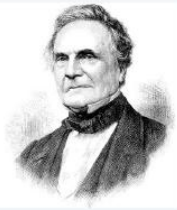
Calculating Time Complexity

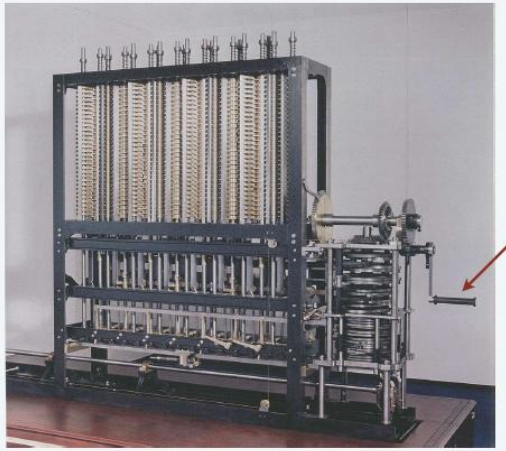
- Cost is the amount of computer time required for a single operation in each line.
- Repetition is the amount of computer time required by each operation for all its repetitions.
- Total is the amount of computer time required by each operation to execute.
- So above code requires '4n+4' Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.
- Totally it takes '4n+4' units of time to complete its execution and it is Linear Time Complexity.
- If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity

Computational Tractability

A strikingly modern thought

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





Analytic Engine

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

Computational Tractability

- Why are we looking at computational tractability?
  - Computational tractability is a factor that is *used to assess or identify if a specific problem statement at hand can be solved with an algorithm*.
  - This factor analysis a problem and focuses on answering
    1. Is the **problem statement** or task *tractable* (easily dealt with)?
    2. If tractable, *can an algorithm (course of calculation) be designed* to find the result of the task?
    3. How can an algorithm *be designed with computational tractability*?

Computational Tractability

- What is computational tractability?
  - It is a measure that is used to categorize if an algorithm is tractable or intractable *by determining the resource requirements* – the amount of time and space (memory or storage) needed for execution of the algorithm.
  - In other words, **the scale at which the resource requirements increases as the size of the input increases**.
  - Meaning, the rate at which the **running time** and **memory utilization GROWS** w.r.t the input size. In this context, two parameter **Time Complexity and Space Complexity** are introduced.
  - So, computational tractability for an algorithm measures the performance of an algorithm in terms of its time and space complexity.

Some Initial Attempts at Defining Efficiency

- Proposed definition for the Efficiency of an Algorithm [1] – An algorithm is efficient if, when implemented, it runs quickly on real input instances.
- GOAL - To solve real problem instances quickly on real computers.
- Unanswered questions in the above definition:
  1. Answers are required to “Where” and “How well” an algorithm is implemented.
  2. Answers are required to “What is a ‘real input instance’?”
  3. Answers are required to “How good or bad an algorithm is capable of scaling as the problem size grows?”
- Refined definition for the efficiency of an algorithm – An algorithm is efficient, if, it is platform-independent, instance-independent, and of predictive value w.r.t. increasing input sizes
- Mathematical definition for the efficiency of an algorithm – Given a problem statement with input size ‘N’, an algorithm can be described and then analyzed for performance (efficiency) by computing the running time mathematically as a function of the input size ‘N’.

Worst-Case Running Times and Brute-Force Search

- The performance of an algorithm can be categorized into Worst-Case, Average Case and Best Case.
- Worst-Case (Mostly used):
  - *maximum amount of time* or resources *an algorithm will take* to complete a task, considering the most challenging input scenario.
- Average Case (rarely used):
  - *expected running time of an algorithm* when considering all possible inputs with a typical distribution.
  - performance of an algorithm averaged over “random” input instances.
- Best Case (very rarely used):
  - *swiftest time an algorithm* can complete a task under optimal conditions.

Worst-Case Running Times and Brute-Force Search

- Why should we go for analyzing the worst-case running time for an algorithm?
  - Average case analysis uses “random” input instances to calculate the algorithm’s performance.
  - The disadvantage of using average case analysis approach is that
    1. an algorithm may perform well on one class of I/P instances and underperform on another class.
    2. Real time i/p is not produced from a random distribution
    3. As a result of point 2, average-case analysis risks telling us more about the means by which the random inputs were generated than about the algorithm itself
  - In practice, generally worst-case analysis of an algorithm reasonably captures the efficiency of an algorithm.

Worst-Case Running Times and Brute-Force Search

- How to analyze the Worst-Case Running Times or Worst-Case Estimate or Pessimistic Estimate?
  - **Step 1:** Calculate the upper bound on the running time of an algorithm.

    i.e., the largest possible running time an algorithm can have over all inputs of a given size ‘N’ is calculated.

    This can be done by identifying the case that causes the execution of maximum number of operations or the longest time
  - **Step 2:** Observe how the running time scales with the input size ‘N’

Worst-Case Running Times and Brute-Force Search

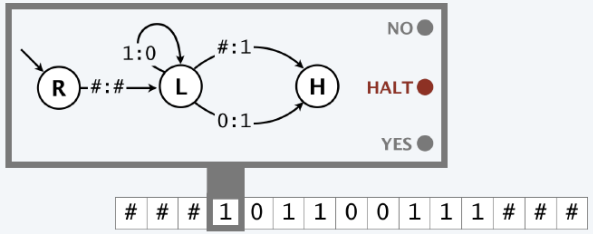
- What is a running-time bound?
  - upper, lower, or tightest limit on the time complexity of the algorithm as a function of input size n.
- How to decide if a running-time bound is impressive or weak?
  - We can decide this with the help of few models of computation.
  - These models helps us understand the limitations of computation – meaning what can be solved with algorithms.
    1. Turing Machines
    2. Word RAM
    3. Brute Force
  - After which we can redefine the term “Efficiency of an algorithm” w.r.t the terminology worst-case performance parameters.

Worst-Case Running Times and Brute-Force Search

1. Turing Machines

Models of computation: Turing machines

Deterministic Turing machine. Simple and idealistic model.



Running time. Number of steps.

Memory. Number of tape cells utilized.

Caveat. No random access of memory.

- Single-tape TM requires $\geq n^2$ steps to detect n-bit palindromes.
- Easy to detect palindromes in $\leq cn$ steps on a real computer.

RAM Institute

W ...orce Search

2. w

Models of computation: word RAM

Word RAM.

assume $w \geq \log_2 n$

- Each memory location and input/output cell stores a w-bit integer.
- Primitive operations: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, ...

constant-time C-style operations ($w = 64$)

Running time. Number of primitive operations.

Memory. Number of memory cells utilized.

Caveat. At times, need more refined model (e.g., multiplying n-bit integers).

RAMAIA
Institute of Tech

Wor ce Search

3. Brute

Brute force

Brute force. For many nontrivial problems, there is a natural brute-force search algorithm that checks every possible solution.

- Typically takes 2^n steps (or worse) for inputs of size n.
- Unacceptable in practice.

Ex. Stable matching problem: test all $n!$ perfect matchings for stability.

Worst-Case Running Times and Brute-Force Search

- Thus, we can redefine the term "Efficiency of an algorithm" w.r.t the terminology worst-case performance parameters.
- Proposed definition for the Efficiency of an Algorithm [2] – An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.
- Meaning, an algorithm is efficient if
  - ✓ Even in the worst case, if it solves a problem faster than just trying every possible solution (brute force)
  - ✓ Brute-force search checks all possible answers one by one – slow for big problems
  - ✓ A better algorithm - finds the answer faster with smart techniques
- Example : looking for a name in a **phone book** with **1 million names**
  - Brute force: Start from page 1 and check every name one by one
  - Efficient search: Use binary search (open the book in the middle, go left or right, and repeat)

Polynomial Time as a Definition of Efficiency

- So, now we are left with the idea of understanding what "qualitatively better performance" really means.
- Meaning to find out a reasonable running time for an algorithm.
- How to quantify the idea of a "reasonable running time"?
- **Problem statement** – The search space or the number of possible solutions for any combinatorial problem grows exponentially in the size N of the input.

  i.e., When the i/p size **increases** by 1,

  (N+1) => Number of possible solutions **increases** multiplicatively

- **Needed solution** – A good algorithm with a better scaling property.

Polynomial Time as a Definition of Efficiency

- So, now what is this idea of a "better scaling property"?
- The running time of an algorithm is controlled by a mathematical rule that limits how many steps the algorithm can take to accomplish a task.
- Arithmetically, this scaling property can be explained as:

> There are absolute constants $c > 0$ and $d > 0$ so that on every input instance of size N,
>
> its running time is bounded by cN^d primitive computational steps.
>
> (In other words, its running time is atmost proportional to N^d.)

- Assumptions:
  - ✓ c and d are constants
  - ✓ N is the size of the input

Polynomial Time as a Definition of Efficiency

- Assumptions:
  - ✓ c and d are constants (Scaling Factor and Growth Rate/Exponent) – does not depend on the input size ‘N’
    - ✓ c = a fixed number
    - ✓ d = exponent number
  - ✓ N is the size of the input
- Definitions:
  - ✓ Scaling Factor (c) : adjusts the number of steps but does not change the overall growth pattern.
  - ✓ Growth Rate (d) : Determines how the algorithm's running time increases as the input N grows.
    - ✓ If d = 1, the algorithm runs in O(N) (linear time).
    - ✓ If d = 2, the algorithm runs in $O(N^2)$ (quadratic time).
    - ✓ If d = 3, the algorithm runs in $O(N^3)$ (cubic time).

Polynomial Time as a Definition of Efficiency

- Inferences from the scaling property of the algorithm:
  - ✓ The algorithm takes atmost **cN^d steps** = Maximum no. steps
  - ✓ Running time is **atmost proportional to N^d** - meaning that as 'N' gets bigger, the running time grows at the rate of N^d , but not faster
  - ✓ So, the running time of an algorithm does not go slow all of a sudden but in this case follows a predictable pattern of N^d

Polynomial Time as a Definition of Efficiency

- Example:
  - ✓ Let's say an algorithm has
    - ✓ Growth Rate / Exponent (d) = 2
    - ✓ Scaling Factor(c) = 5
    - ✓ Input size (N) = 10
    - ✓ then the maximum steps or the running time = $5 \times 10^2 = 500$ steps
- Take aways:
  - ✓ c affects the scaling (makes the running time larger or smaller).
  - ✓ d affects the growth rate (controls how fast the time increases as input size grows).
  - ✓ These constants help define the **worst-case running time** of the algorithm.

Polynomial Time as a Definition of Efficiency

- Thus, in any case, if this running-time bound holds, for some c and d, then we say that the **algorithm** has a ***polynomial running time (or) polynomial-time algorithm***
  - ✓ *Suppose if input size (N) = 2N,*
  - ✓ *The bound on the running time =* $c(2N)^d = C \cdot 2^d N^d$ *where,* 2^d *is the slow down factor*
  - ✓ *lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials*

- Proposed definition for the Efficiency of an Algorithm [3] – An algorithm is efficient if it has a polynomial running time.
- An algorithm that follows a polynomial running time would follow a time complexity = $O(N^c)$

Polynomial Time as a Definition of Efficiency

- Challenge with this definition –
  - Consider the below scenarios
    1. The running time of the algorithm is proportional to N^{100} (a polynomial).
    2. The running time of the algorithm is proportional to $n^{1+.02(\log n)}$
  - Which would be efficient and which would be inefficient / efficient and why?
  - Now, N^{100} , even though is technically a polynomial function, it is extremely slow in practice.
  - Why?
    - For large values of N, raising it to the 100th power results in an enormous number of operations, making the algorithm impractical for real-world use - inefficient

Polynomial Time as a Definition of Efficiency

- **Challenge with this definition –**
  - Consider the below scenarios
    1. The running time of the algorithm is proportional to N^{100} (a polynomial).
    2. The running time of the algorithm is proportional to $n^{1+.02(\log n)}$
  - Which would be efficient and which would be inefficient / efficient and why?
  - On the other hand, considering $n^{1+.02(\log n)}$
    1. It is a non-polynomial function because the exponent itself is growing with n (**Key** - yes, growth is slow!)
    2. Growth is much smaller than raising N to the 100th power – can be considered relatively efficient
  - Thus, we can conclude that defining the efficiency in terms of polynomial running time is not actually efficient

Other types of analyses

Probabilistic. Expected running time of a randomized algorithm.

Ex. The expected number of compares to quicksort n elements is $\sim 2n \ln n$.

Amortized. Worst-case running time for any sequence of n operations.

Ex. Starting from an empty stack, any sequence of n push and pop operations takes $O(n)$ primitive computational steps using a resizing array.

Also. Average-case analysis, smoothed analysis, competitive analysis, ...

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What is Asymptotic Analysis?
  - Consider 2 algorithms that are running on a computer. The performance of an algorithm or the comparison of time efficiency of one algorithm over another algorithm is referred to as Asymptotic Analysis.
- How are we evaluating the performance using the concept of asymptotic analysis?
  - We are evaluating the performance of an algorithm in terms of the input size (N)
  - We consider the **order of growth** of time or space taken by the algorithm in terms of the **input size - (or) the growth rate**
  - So, Asymptotic analysis is is a method of evaluating the efficiency of an algorithm by analyzing its **growth rate (running time of an algorithm)** as the input size (n) approaches infinity.
  - Instead of calculating the exact running time or exact limit, asymptotic analysis focuses on the order of growth or considers the input size (N) as a function of n ie., f(n).

Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic Notations?
  - Asymptotic notations is the mathematical way of representing the **time complexity**.
  - To compare and rank the order of growth of an algorithm's basic operation count (**no. of times** a statement is executed – iteration, frequency, or the no.of times a function is calling itself), computer scientists use three notations : O (big oh), Ω (big omega), and θ (big theta).
  - Apart from these notations we also have a o(little oh) and a ω (little omega).
  - Mostly the time complexity of algorithms is compared in terms of **O (big oh)**, Ω (big omega), and θ (big theta) notations.

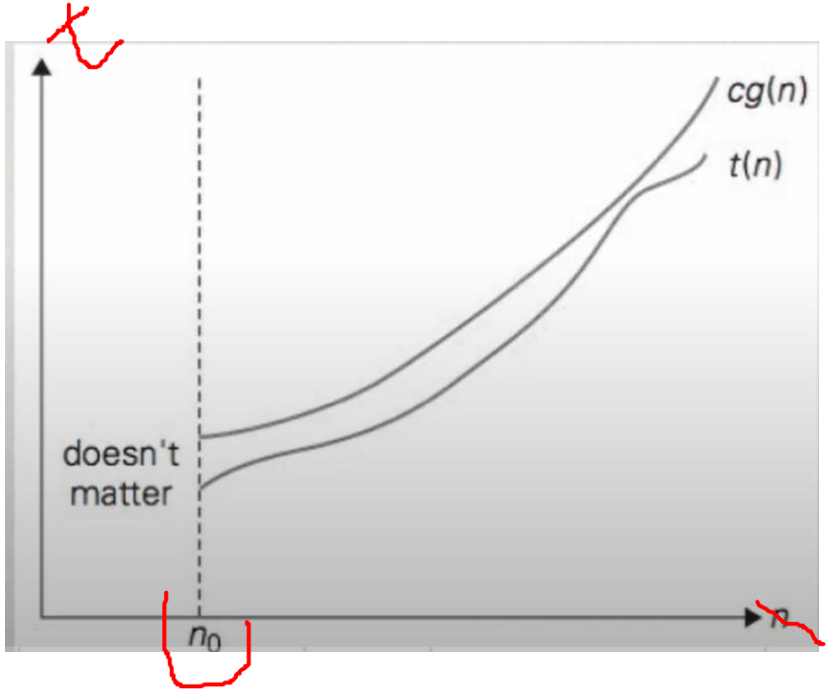
Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic Notations?
  - To understand the notation usage and context, we need to represent and understand the meaning of few more variables
    1. The **input size** = **n**
    2. Any problem at hand represented in terms of a function. The function represents the **growth of the input size(n) w.r.t the time(t)** is denoted as **f(n)**
    3. Any algorithm's **worst case running time w.r.t its input size** is denoted as **T(n) = Time complexity**
    4. Any algorithm's **basic operation count** is denoted as **C(n)**
    5. Simple **function to compare the count** or the **order of magnitude** is denoted as **g(n)**

Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic **Upper Bounds**?
  - Represented by the notation O (big oh).
  - So, this is the graphical representation of a O (big oh) notation.
  - t(n) is said to be O(g(n)) if we find suitable constants c and n0

    so that c(g(n) is an upper bound for t(n) for n beyond n0.
  - t(n) <= cg(n) for every n>=n0



Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic **Upper Bounds**?
  - To understand this, let's look at an example,
    - Suppose that t(n) = 100n + 5 which we claim as $O(n^2)$ where n is the i/p size and i/p size for a problem is always going to be >=1 (atleast 1)
    - Now for t(n) if we choose n>=5 (bigger than 5) then we can say 100n + 5 is smaller than or equal to
    - 100n + n
    - i.e,
    - 100 n + 5 <=100n + n, for n>=5
    - Ie., = 101n <=$101n^2$ because since n>=1, n^2 is bigger than n
    - Hence by choosing n0= 1 and c = 101, we have established that n^2 is an upper bound to 100n+5, which means 100n+5 is $O(n^2)$

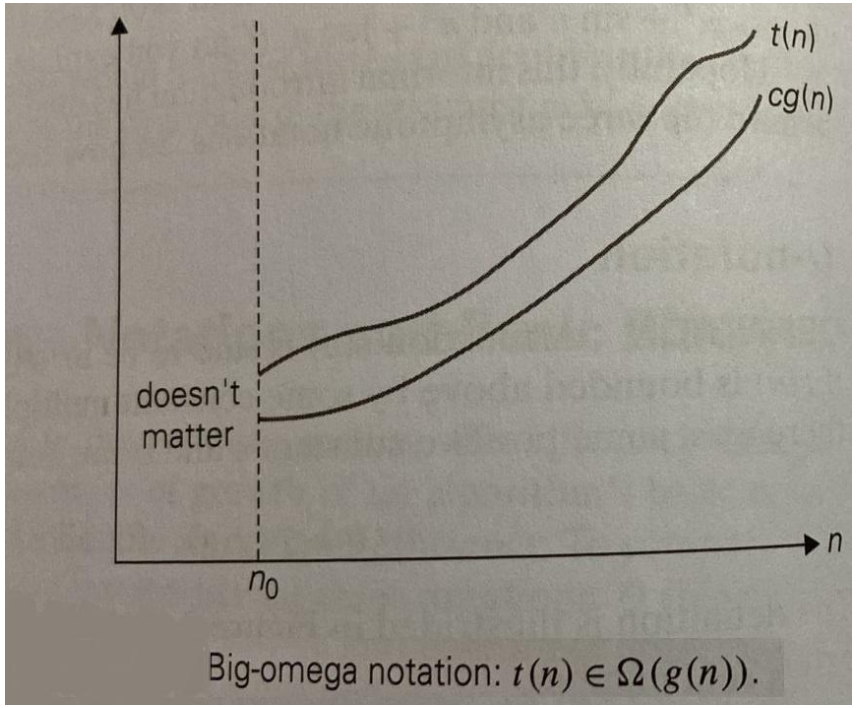
Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic **Upper Bounds**?
  - To conclude this example, we can infer
  1. n0 and c are not unique. Depending on how we do the calculation, there might be different n0 and different c
  2. So it means beyond a certain n0 there is a **uniform constant** c such that c(g(n) dominates t(n)

Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic **Lower Bounds**?
  - Represented by the notation Ω (big omega).
  - So, this is the graphical representation of a Ω (big omega) notation.
  - t(n) is said to be Ω(g(n)) denoted by t(n) belongs to Ω(g(n)), if t(n) is bounded below by some positive constant multiple of g(n) for all large n
  - i.e., if there exists some positive constant c and some non-negative integer n0 such that

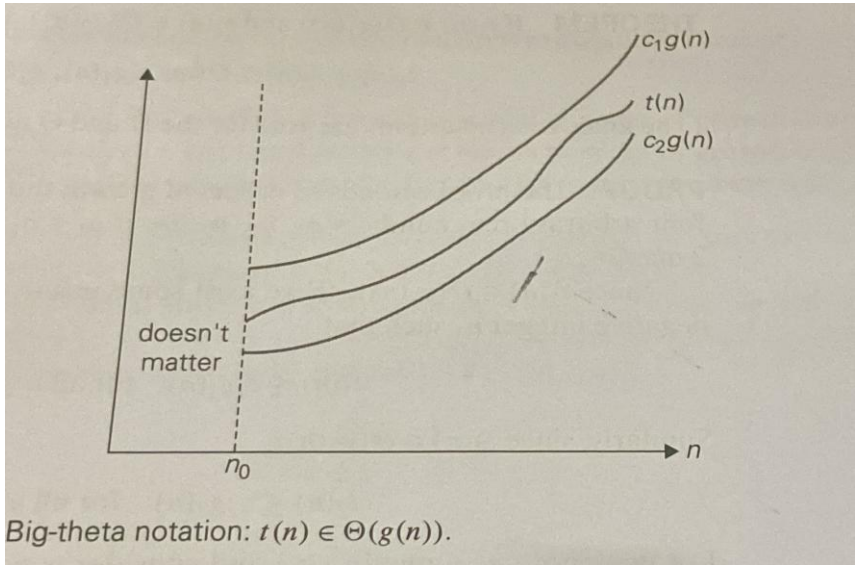
T(n) >=cg(n) for all n>=n0



Asymptotic Order of Growth –Asymptotic Analysis / Asymptotic Notations

- What are Asymptotic **Tight Bounds**?
  - Represented by the notation θ (big theta).
  - So, this is the graphical representation of a θ (big theta) notation.
  - t(n) is said to be θ g(n)) denoted by t(n) belongs to θ (g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n
  - i.e., if there exists some positive constants c1 and c2 and some non-negative integer n0 such that

C2g(n) <=t(n)<=c1g(n) for all n>=n0



Big-theta notation: $t(n) \in \Theta(g(n))$.

Asymptotic Order of Growth – Tight Bound Proof

- **Proof 1: To obtain an asymptotically tight bound for a function using limits**
  - **Statement :** An asymptotically tight bound can be obtained directly **by computing a limit** as n goes to infinity. If the ratio of functions f(n) and g(n) converges to a positive constant as n goes to infinity, then

$$f(n) = \theta\, g(n)$$

  - **Given:** Mathematically the statement can be rewritten as:
    - Let f and g be two functions such that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

    - Exists and is equal to some number c>0, then $f(n) = \theta\, g(n)$

Asymptotic Order of Growth – Tight Bound Proof

- **Proof 1: To obtain an asymptotically tight bound**
  - **Proof :** This can be proved by using the fact that **there exists a limit and it is positive.**

    Now to show that f(n) = O(g(n)) and f(n) = Ω g(n) as per the definition of θ.

    Since, $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = c > 0 , from the statement we also understand that there is some n_0

    Beyond which the ratio is always **between ½ c and 2c**.

    Thus, f(n) <= 2 c.g(n) for all n>= n_0 => f(n) = O(g(n))

    Also, f(n) > = ½ c.g(n) for all n>= n_0 => f(n) = Ω g(n)

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

1. **Transitivity**

Statement: If a function *f* is asymptotically upper-bounded by a function *g*, and if *g* in turn is asymptotically upper-bounded by a function *h*, then *f* is asymptotically upper-bounded by *h*.
Similarly, If a function *f* is asymptotically lower-bounded by a function *g*, and if *g* in turn is asymptotically lower-bounded by a function *h*, then *f* is asymptotically lower-bounded by *h.*

Mathematically, this statement can be rewritten as,

$$(a)\ \text{If } f = O(g) \text{ and } g = O(h), \text{ then } f = O(h).$$
$$(b)\ \text{If } f = \Omega(g) \text{ and } g = \Omega(h), \text{ then } f = \Omega(h).$$

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

Proof:

1. As per the upper bound definition, we know that for some constants c and n0, we have f (n) ≤ cg(n) for all n ≥ n0
2. Also, for some constants c' and n_0', g(n) ≤ c'h(n) for all n ≥ n_0'
3. So we need to consider any number n that is at least as large as both n_0 and n_0'
4. So now we have f (n) ≤ cg(n) ≤ cc' h(n), and so f (n) ≤ cc' h(n) for all n ≥ max(n_0 , n_0')

Hence, we have proved that the property of transitivity holds for asymptotically upper and lower bounds, given 3 functions f, g, and h.

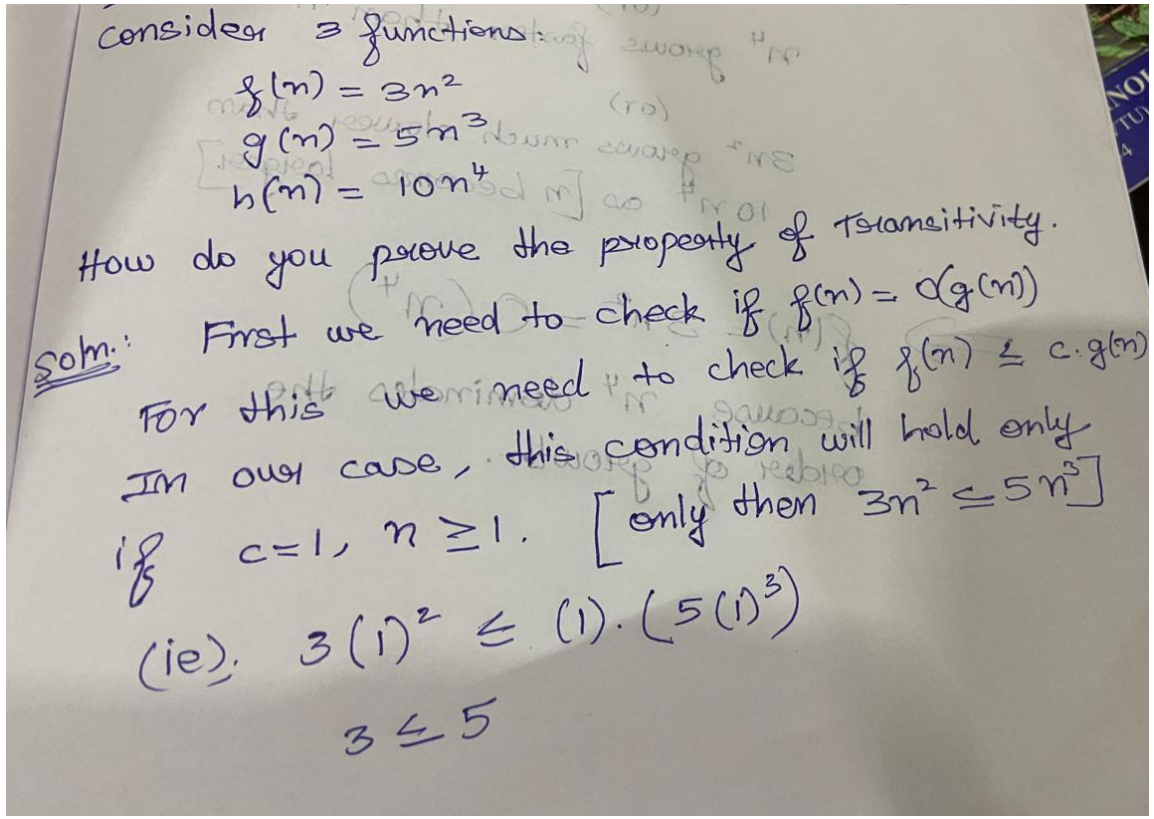
Similarly, transitivity property holds good for asymptotically tight bounds as well.

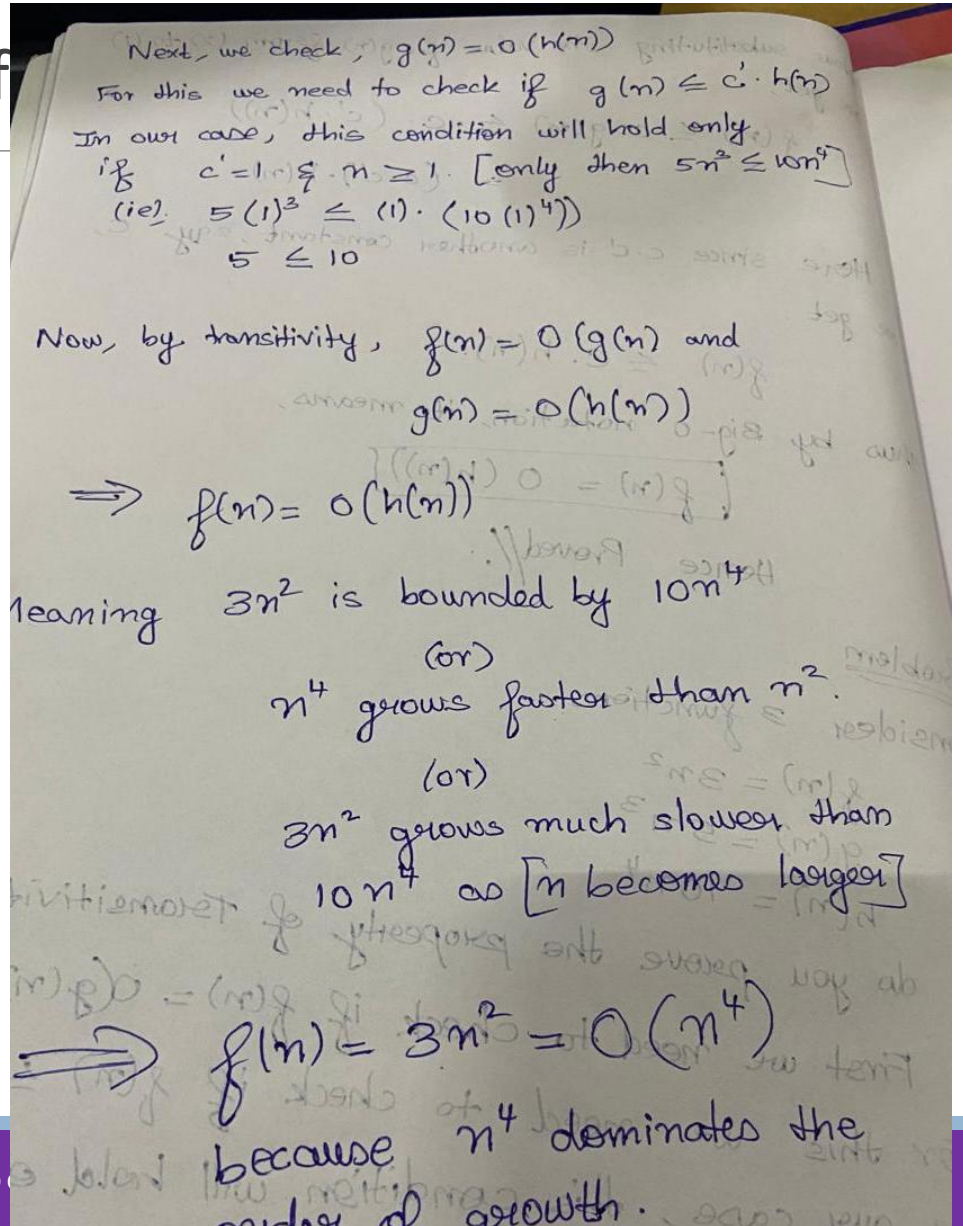
Task : Prove that transitivity property holds good for asymptotically tight bounds.

RAMAIAH
Institute of Technology

Asymptotic Order of Growth – Properties of

Example problem:

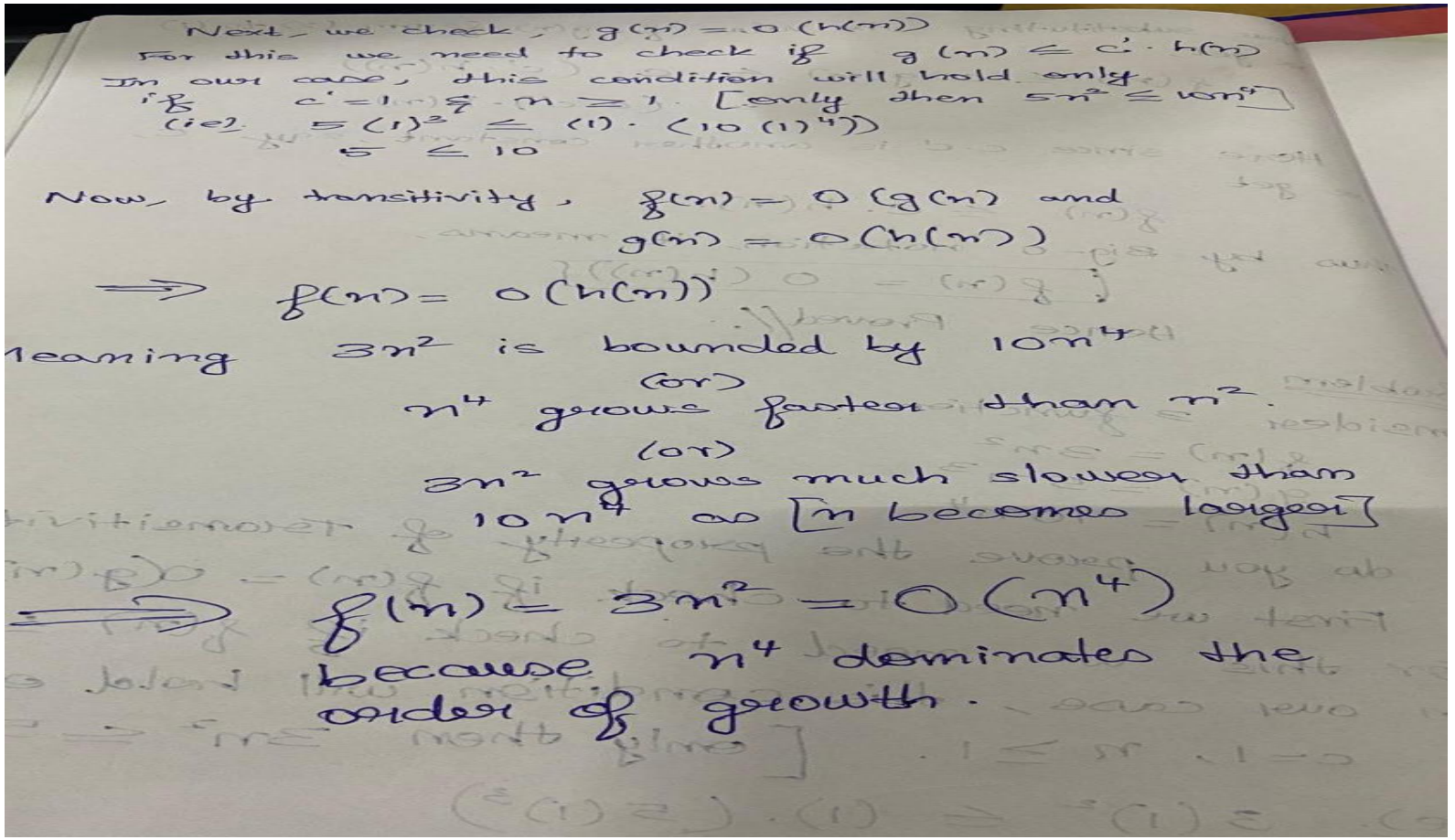




RAMAIAH
Institute of Technology

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

Example problem:



Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

2. Sum of Functions

Statement: if we have an asymptotic upper bound that applies to each of two functions f and g, then it applies to their sum.

Mathematically, this statement can be rewritten as,

Suppose that f and g are two functions such that for some other function h, we have f = O(h) and g = O(h).

Then we can say f + g = O(h).

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

Proof:

1. For some constants c and n0, we have f (n) ≤ c h(n) for all n ≥ n0
2. Also, for some constants c' and n_0', g(n) ≤ c'h(n) for all n ≥ n_0'
3. So we need to consider any number n that is at least as large as both n_0 and n_0'
4. So now we have f (n) + g(n) ≤ cg(n) ≤ c h(n) + c'h(n)
5. Thus, f (n) + g(n) <= (c + c') h(n) for all n ≥ max(n_0 , n_0'). Thus it shows that f + g = O(h)

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

3. Sum of Functions (Generalized)

- This property generalizes the sum of functions rule to any fixed number k of functions.

Statement : Let k be a fixed constant, and let f1, f2, …fk and h be functions such that fi = O(h) for all i. Then

f1+f2+…+fk = O(h)

Mathematically it can be written as This property states that if multiple functions f1,f2,...,fk are all asymptotically **bounded above (upper bound)** by another function h(n) (meaning if each function fi=O(h)), then their **sum** is also bounded above by h(n),

i.e., If $f1(n)=O(h(n)),f2(n)=O(h(n)),\ldots,fk(n)=O(h(n))$

Then $f1(n)+f2(n)+\cdots+fk(n)=O(h(n))$

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

Conclusions:

1. The sum of a finite number of functions with the same asymptotic bound remains within that bound.

2. When analyzing algorithms, we often approximate complexity using the dominant term.

Asymptotic Order of Growth – Properties of Asymptotic Growth Rates

4. Property:

Statement: Suppose that f and g are two functions (taking non-negative values) such that g(n) = O(f(n)) , then the sum f(n)+g(n) is asymptotically tight-bounded by f(n).

Mathematically it can be written as f(n) + g(n) = Θ(f(n))

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- In the analysis of algorithms, a number of functions come up repeatedly.
- Asymptotic properties of some basic functions are considered here, namely:
    1. Polynomials
    2. Logarithms
    3. Exponentials

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- **Polynomials:** A polynomial is a function that can be written in the form $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$ for some integer constant $d > 0$, where the final coefficient a_d is nonzero. This value d is called the *degree* of the polynomial.
  - Example - Functions of the form $pn^2 + qn + r$ (with p not equal to 0) are polynomials of degree 2.
- The major factor about polynomials is that their asymptotic order of growth or rate of growth is determined by their **"higher-order term" -** the one that determines the degree.

- Proof explained on board

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- **Polynomials:**

  Conclusions:

  - The dominant term (largest exponent, n^d) dictates the growth rate.
  - Lower-degree terms like n^{d-1},n^{d-2}, etc., grow much slower than n^d and become insignificant in Big-O notation.
  - Since Big-O ignores constant factors, we only consider the highest-degree term's growth.
  - The sum of multiple $O(n^d)$ terms is still $O(n^d)$.

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- **Logarithms:**
- **Growth Rate of Logarithmic Functions:** Logarithms are very slowly growing functions that are much smaller than very small polynomial functions. In other words, **even the smallest polynomial function grows faster than a logarithmic function** as 'n' becomes large.

  Now, $\log_b n$, is the number x such that $b^x = n$.

  Example : $\log_2 8 = 3$ because $2^3=8$.

  Statement: For every base *b*, the function $\log_b n$ is asymptotically upper bounded by every function of the form n^x, even for (non-integer) values of *x* arbitrary close to 0.

  Mathematically it can be rewritten as,

  $$\log_b n = O(n^x) \text{ for any } b>1 \text{ and } x>0$$

  Proof explained on board

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- **Logarithms:**

  Key take aways:

  1. log 2 n grows **slower than any polynomial function**, but the crossover happens at an extremely large n.
  2. For $n \approx 10^{300}$, $n^{0.01}$ finally surpasses $\log_2 n$.
  3. This confirms the Big-O bound:

     $\log_b n = O(n^x)$ for any $b>1$ and $x>0$

  Logarithms are **very slow-growing** functions, which is why algorithms like **binary search (O(logn))** are much faster than algorithms with even tiny polynomial complexity like **$O(n^{0.01})$**.

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- **Exponentials:**
- An exponential function is of the form:

  $f(n)=r^n$, where r is a constant base
- A polynomial function is of the form:

  $g(n)=n^d$, where $d > 0$ is a fixed exponent
- **Statement:** Every exponential function eventually outgrows every polynomial function, no matter how large the polynomial exponent d is, for larger values of n.
- **Key point:** If two exponential functions have **different bases** (say r^n and s^n where $r > s$), they are **not asymptotically equal**.
- Thus, when analyzing the running times of algorithms, the range of possible functions can either be logarithmic, polynomial, and exponential.

Asymptotic Order of Growth – Asymptotic Bounds for Some Common Functions

- **To conclude,**

  Exponential Growth Function / Exponential running time >

  Polynomial Growth Function / Polynomial running time >

  Logarithmic Growth Function / Logarithmic running time

A Survey of Common Running Times

- In computational complexity, algorithms have different running times based on how their execution grows with input size **n**. Below is an overview of common time complexities:

• O(1)	Constant	Accessing an array element
• O(log n)	Logarithmic	Binary Search
• O(n)	Linear	Finding max in an array
• O(n log n)	Near Linear	Merge Sort, Quick Sort
• $O(n^2)$	Quadratic	Bubble Sort, Floyd-Warshall
• $O(n^3)$	Cubic	Matrix Multiplication
• $O(2^n)$	Exponential	Recursive Fibonacci
• O(n!)	Factorial	Traveling Salesman Problem

$O(n^k)$ Time Complexity (Polynomial Time)

- **1. $O(n^k)$ Time Complexity (Polynomial Time)**
- **Definition**: Any algorithm where the number of operations is proportional to **n raised to a constant power (k)**.
- **Examples**:
  - Matrix multiplication (**$O(n^3)$**)
  - Floyd-Warshall algorithm (**$O(n^3)$**)
  - Dynamic programming solutions for problems like **Longest Common Subsequence ($O(n^2)$).**
  - **The above are Tractable Problems**

Beyond Polynomial Time (Intractable Problems): Factorial Time (O(n!))

- **Definition**: The number of operations is proportional to n! (factorial).
- **Examples:** ***Traveling Salesman Problem (TSP) using brute force***

Generating all permutations of a string

Brute Force TSP (O(n!))

```
int tsp(int graph[][N], int v[], int n, int pos, int count, int cost, int minCost) {
    if (count == n && graph[pos][0]) {
        return cost + graph[pos][0];
    }
    for (int i = 0; i < n; i++) {
        if (!v[i]) {
            v[i] = 1;
            minCost = tsp(graph, v, n, i, count + 1, cost + graph[pos][i], minCost);
            v[i] = 0;
        }
    }
    return minCost;
}
```

Some Representative Problems – A First Problem: Stable Matching

Stable Marriage Problem

There is a set Y = $\{m_1,\ldots,m_n\}$ of n men and a set X = $\{w_1,\ldots,w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).

A *marriage matching* M is a set of n pairs (m_i, w_j).

A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.

A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.

The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

Some Representative Problems – A First Problem: Stable Matching

Instance of the Stable Marriage Problem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable

{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable

Some Representative Problems – A First Problem: Stable Matching

Stable Marriage Algorithm (Gale-Shapley)

Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and do the following:
Proposal The selected free man m proposes to w, the next woman on his preference list
Response If w is free, she accepts the proposal to be matched with m. If she is not free, she compares m with her current mate. If she prefers m to him, she accepts m's proposal, making her former mate free; otherwise, she simply rejects m's proposal, leaving m free

Step 2 Return the set of n matched pairs

Some Representative Problems – A First Problem: Stable Matching

Example

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Free men:
Bob, Jim, Tom

Bob proposed to Lea
Lea accepted

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Free men:
Jim, Tom

Jim proposed to Lea
Lea rejected

Some Representative Problems – A First Problem: Stable Matching

Example (cont.)

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Sue
Sue accepted

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Tom proposed to Sue
Sue rejected

Some Representative Problems – A First Problem: Stable Matching

Example (cont.)

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Tom proposed to Lea
Lea replaced Bob
with Tom

Free men:
Bob

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Ann
Ann accepted

Some Representative Problems – A First Problem: Stable Matching

Analysis of the Gale-Shapley Algorithm

- The algorithm terminates after no more than n^2 iterations with a stable marriage output
- The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage. One can obtain the *woman-optimal* matching by making women propose to men
- A man (woman) optimal matching is unique for a given set of participant preferences
- The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training