Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture – 1 : Introduction

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Prerequisites and Objectives

- **Analysis of Algorithms Prerequisites:**
  - Discrete Mathematics (counting arguments, induction, recurrence relations and discrete probability)
- **Objectives:**
  This is an introductory course in the analysis and design of combinatorial algorithms. Emphasis is given on
  1. familiarizing the students with fundamental algorithmic paradigms and
  2. Rigorous analysis of combinatorial algorithms.

  This is a modern introduction to combinatorial algorithms and it maintains some consistency with previous courses.

Course Description

- This course teaches techniques for the analysis of efficient algorithms, emphasizing methods useful in practice.
- Algorithms are recipes for solving computational problems.
- In this course we will study fundamental algorithms for solving a variety of problems, including sorting, searching, divide and-conquer, dynamic programming, greediness, and probabilistic approaches.
- Algorithms are judged not only by how well they solve a problem, but also by how effectively they use resources like time and space.
- Techniques for analyzing time and space complexity of algorithms and to evaluate tradeoffs between different algorithms.
- Analysis of algorithms is studied - worst case, average case, and amortized - with an emphasis on the close connection between the time complexity of an algorithm and the underlying data structures.
- NP-Completeness theory is examined along with methods of coping with intractability, such as approximation and probabilistic algorithms.
- A basic understanding of mathematical functions and data structures is a prerequisite for the subject.
- A lab course is associated with it to strengthen the concepts

Course Outline (To be covered in 30 lectures)

1. Introduction, Review of basic concepts, advanced data structures like Binomial Heaps, Fibonacci Heaps (5)
2. Divide and Conquer with examples such as Sorting, Matrix Multiplication, Convex hull etc(6)
3. Dynamic programming with examples such as Kanpsack, All pair shortest paths etc (4)
4. Backtracking, Branch and Bound with examples such as Travelling Salesman Problem etc (6)
5. Algorithms involving Computational Geometry (4)
6. Selected topics such as NP-completeness, Approximation algorithms, Randomized algorithms, String Matching (5)

Text Books

1. Introduction to Algorithms by Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest
2. Fundamentals of Computer Algorithms by E. Horowitz & S Sahni
3. The Design and Analysis of Computer Algorithms by Aho, Hopcraft, Ullman

MS Team Code

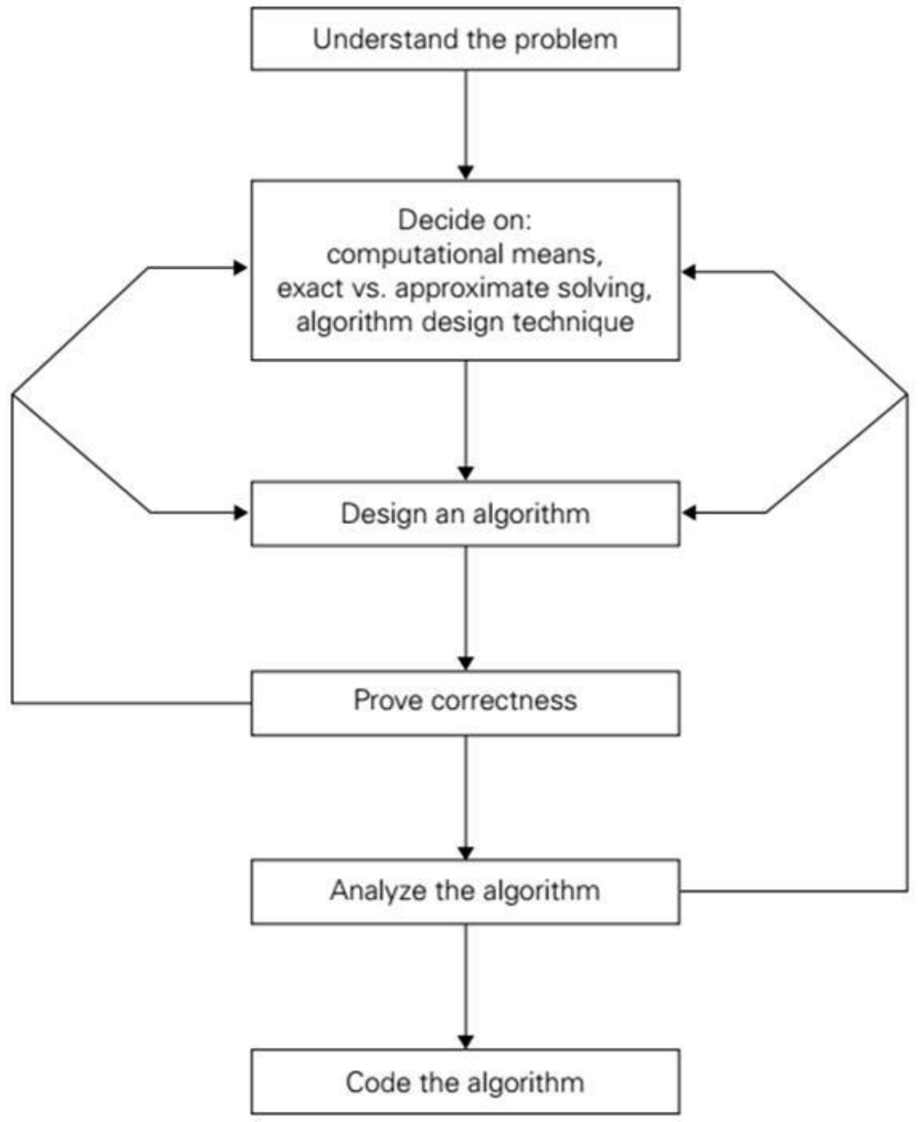
- MS Team Code for submitting the assignments, quiz and feedback forms:

y9ppwbd

Algorithms

- An algorithm specifies a procedure for solving a problem in a finite number of steps.
- An algorithm is a well-defined list of steps for solving a particular problem.
- Computer scientist Niklaus Wirth stated that, ***Program = Algorithms + Data.***

Algorithm design and analysis process.

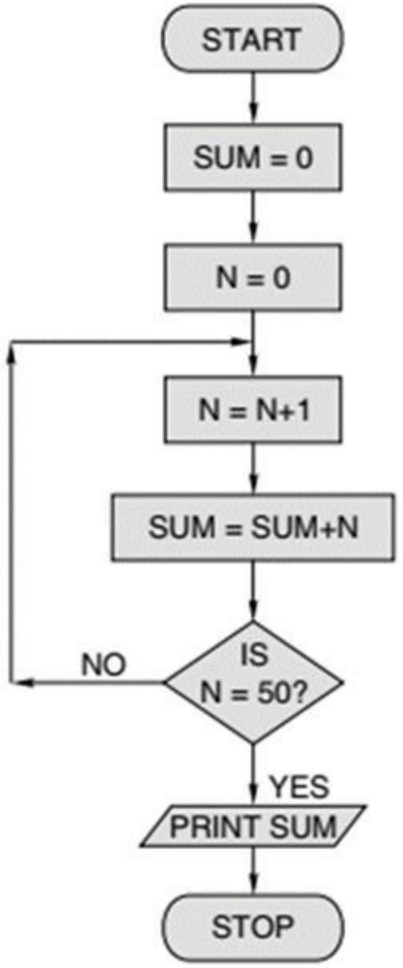


Different Ways of stating Algorithms

- Step-form
  - In the step-form representation, the procedure of solving a problem is stated with written statements.
  - Each statement solves a part of the problem and these together complete the solution.
  - The step-form uses just normal language to define each procedure.
  - Every statement, that defines an action, is logically related to the preceding statement.
  - An example of an algorithm, for making a pot of tea:
    1. If the kettle does not contain water, then fill the kettle.
    2. Plug the kettle into the power point and switch it on.
    3. If the teapot is not empty, then empty the teapot.
    4. Place tea leaves in the teapot.
    5. If the water in the kettle is not boiling, then go to step 5.
    6. Switch off the kettle.
    7. Pour water from the kettle into the teapot

Different Ways of stating Algorithms

- Flowchart
  - Flowchart use symbols and language to represent sequence, decision, and repetition actions.
  - Example to find sum of first 50 natural numbers.



Different Ways of stating Algorithms

- Pseudo-code
  - The pseudo-code is a written form representation of the algorithm.
  - However, it differs from the step form as it uses a restricted vocabulary to define its action of solving the problem.
  - One problem with human language is that it can seem to be imprecise. But the pseudo-code, which is in human language, tends toward more precision by using a limited vocabulary.
  - Example to find maximum of three numbers

```
1. START
2. PRINT “ENTER THREE NUMBERS”
3. INPUT A, B, C
4. MAX ← A
5. IF B > MAX THEN MAX ← B
6. IF C > MAX THEN MAX ← C
7. PRINT MAX
8. STOP
```

Different Ways of stating Algorithms

- In order to express the algorithm for a given operation, we will assume different control structures and notations as stated below

Algorithm <Name of the Operation>(Input parameters:; Output parameters)

Input: <Specification of input data for the operation>

Output: <Specification of output after the successful performance of the operation>

Data Structure: <If the operation assumes other data structure for its implementation>

Steps:

```
1.........
2. If <condition> then
        1. ..........
        2. ..........
3. Else
        1. .........
        2. ........
4. End if
5. .....
6. While <Condition> do
        1. .........
        2. ........
        ........
7. End While
.......
8. Stop
```

Efficient Algorithm!!!

- Sometimes, there are more than one way to solve a problem.
- We need to learn how to compare the performance different algorithms and choose the best one to solve a particular problem.
- While analyzing an algorithm, we mostly consider time complexity and space complexity.
- Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
- Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

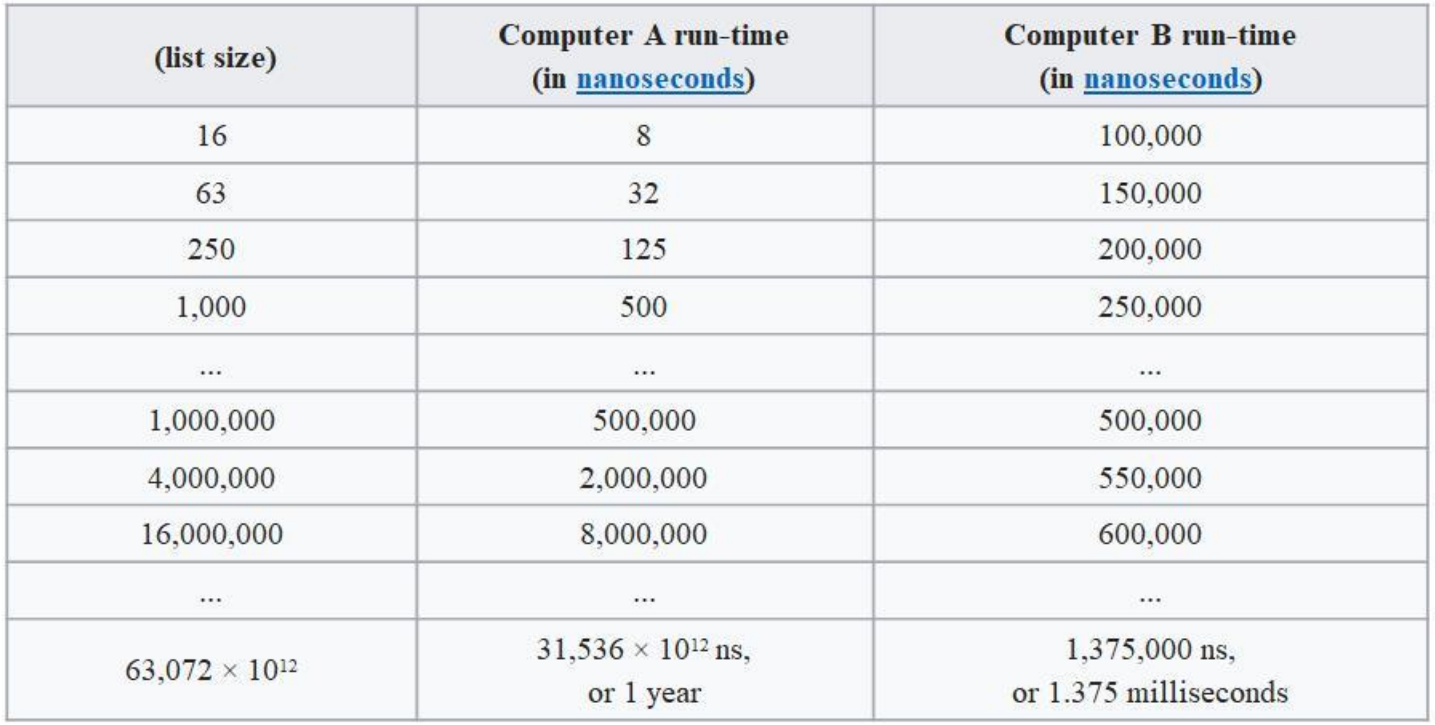
Time Complexity

- Take as an example of a program that looks up a specific entry in a sorted list of size n.
  - Suppose this program were implemented on Computer A, a state-of-the-art machine, using a linear search algorithm, and on Computer B, a much slower machine, using a binary search algorithm.
  - Benchmark testing on the two computers running their respective programs might look something as follows:

n **(list size)**	**Computer A run-time (in nanoseconds)**	**Computer B run-time (in nanoseconds)**
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000

Time Complexity

- Based on these metrics, it would be easy to jump to the conclusion that *Computer A* is running an algorithm that is far superior in efficiency to that of *Computer B*. However, if the size of the input-list is increased to a sufficient number, that conclusion is dramatically demonstrated to be in error:
- Hence, we measure time complexity of an algorithm using Asymptotic Analysis



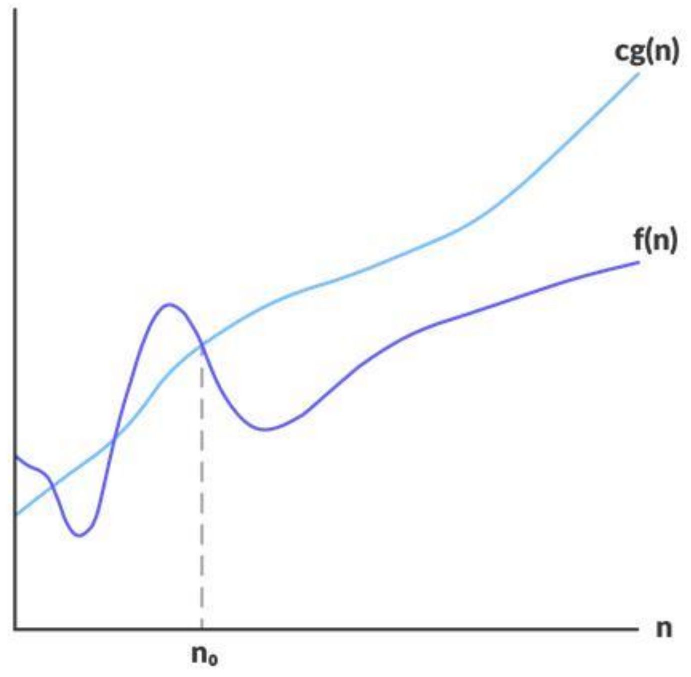
(list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000
...	...	...
1,000,000	500,000	500,000
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000
...	...	...
$63{,}072 \times 10^{12}$	$31{,}536 \times 10^{12}$ ns, or 1 year	1,375,000 ns, or 1.375 milliseconds

Asymptotic Analysis: Time Complexity

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation /framing of its run-time performance.
- Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Usually, the time required by an algorithm falls under three types −
  - **Best Case** − Minimum time required for program execution.
  - **Average Case** − Average time required for program execution.
  - **Worst Case** − Maximum time required for program execution.
- Asymptotic Notations
  - Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  - Ο Notation
  - Ω Notation
  - θ Notation

Asymptotic Analysis: Time Complexity

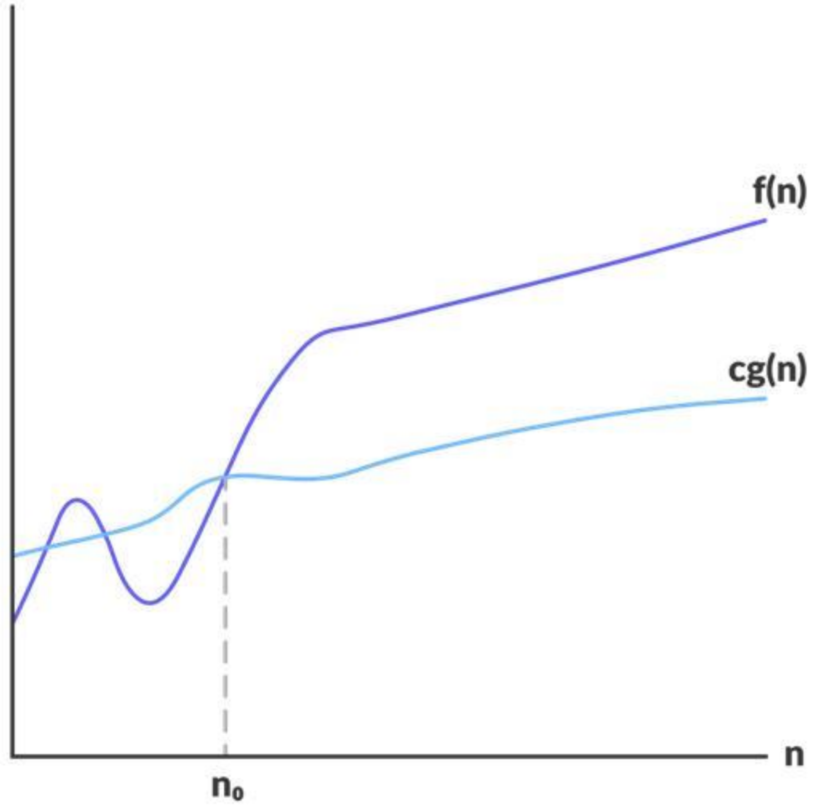
- Big Oh Notation, O
- The notation O(n) is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
- For example, for a function f(n)
    - $f(n) = \{ O(g(n)) :$ there exists $c > 0$ and n_0 such that $f(n) \le c.g(n)$ for all $n > n_0$



f(n) = O(g(n))

Asymptotic Analysis: Time Complexity

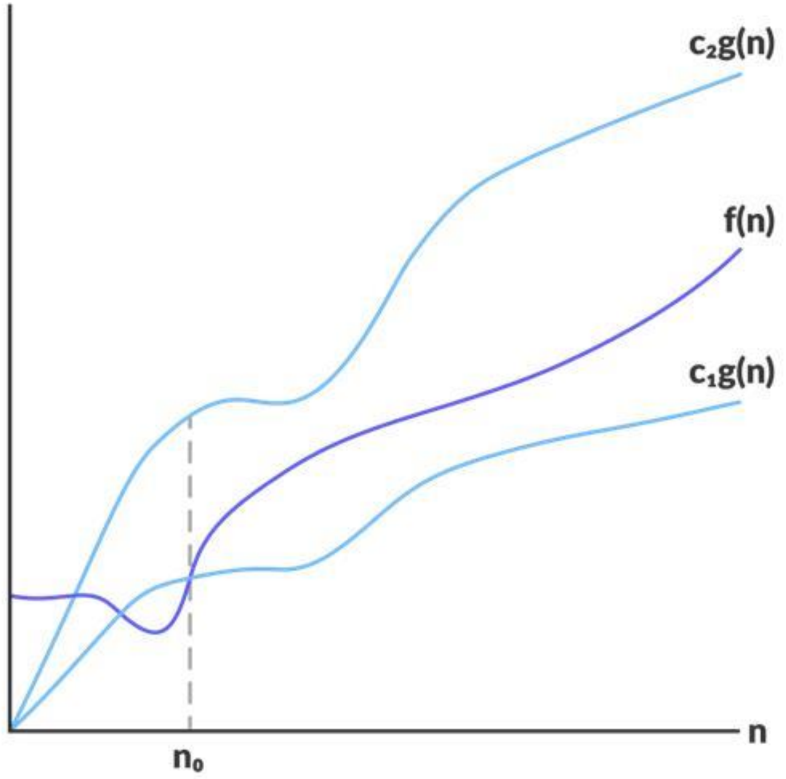
- Omega Notation, Ω
- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- For example, for a function f(n)
  - $f(n) \geq \{\Omega(g(n)) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq f(n) \text{ for all } n > n_0.\}$



f(n) = Ω(g(n))

Asymptotic Analysis: Time Complexity

- Theta Notation, θ
- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- It is represented as follows −
    - $f(n) = \{\theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ for all $n > n_0$. i.e. $C_1 g(n) \le f(n) \le C_2 g(n)$ for all $n > n_0$. }



$f(n) = \Theta(g(n))$

Asymptotic Analysis: Time Complexity

- Example of how we measure asymptotic analysis
  - a. printf("Hello World");
    - In above code “Hello World!!!” print only once on a screen. So, time complexity is constant: O(1) i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.
  - b. Loop: 1 to n times

    printf("Hello World");
    - In above code “Hello World!!!” will print N times. So, time complexity of above code is O(1).

Asymptotic Analysis: Time Complexity

- Example of how we measure asymptotic analysis
  - c. Pseudocode:

```
list_Sum(A,n)
{                              //A->array and n->number of elements in the array
        total =0                        // cost=1 no of times=1
        loop: i=0 to n-1   // cost=n+1 (+1 for the end false condition)
                sum = sum + A[i]  // cost=n
        return sum                      // cost=1 no of times=1
}
```

  - **Tsum=1 + (n+1) + n + 1 = 2n + 3 =C1 * n + C2 = O(n)**
- The complexity is a polynomial equation (quadratic equation for a square matrix)
  - Matrix (nxn) => Tsum= $an^2 + bn + c$
  - For this Tsum if in order of $n^2 = O(n^2)$
  - The above O -> is called Big – O which is an asymptotic notation.

Class Exercise

- $2n+6$
- $O(n)$
- $50 n \log n + 60n+2$
- $O(n \log n)$
- $8n^2 \log n + 5n^2 + n$
- $O(n^2 \log n)$
- Increasing order of Asymptotic analysis
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^2 \log n) < O(n^3) < O(n^m) < O(m^n) < O(n!)$

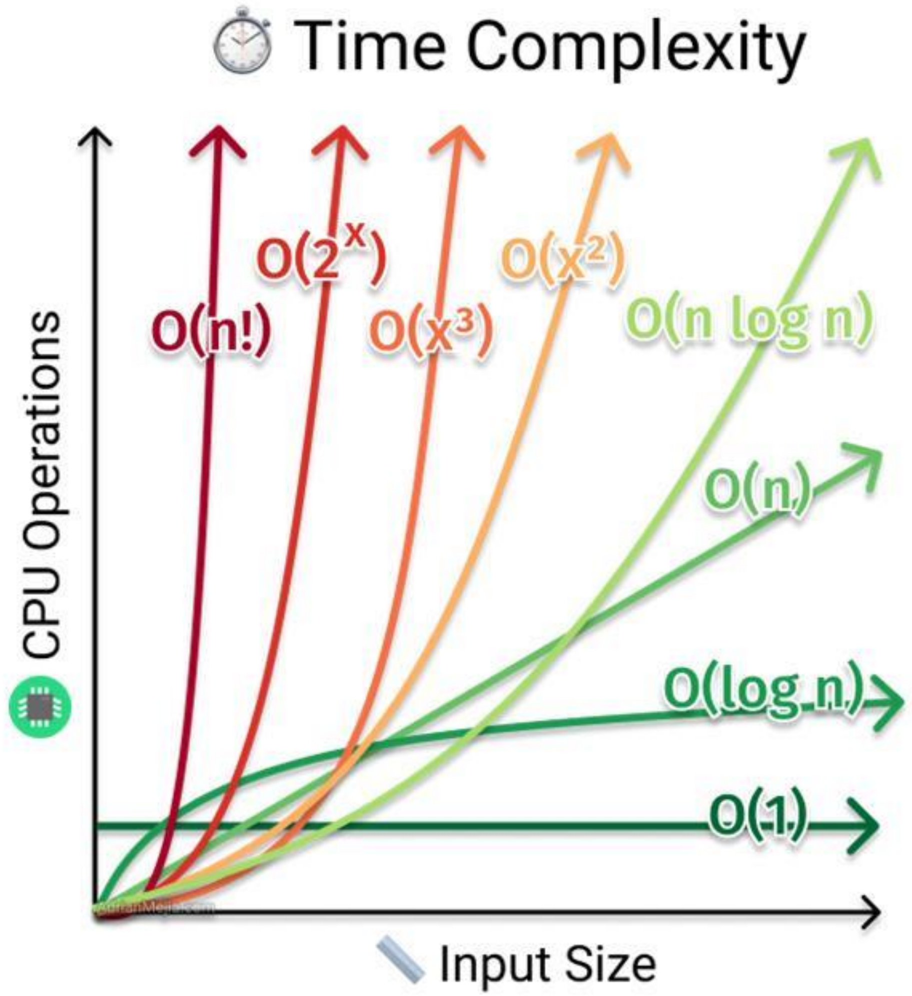
Orders of Growth

- Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Asymptotic Notation (terminology)

- Special classes of algorithms:
  - Logarithmic: O(log n)
  - Linear: O(n)
  - Quadratic: O(n^2)
  - Polynomial: O(n^k), $k \geq 1$
  - Exponential: O(a^n), $a > 1$



Class Exercise #2

```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Class Exercise #2

```
INSERTION-SORT($A$)                                  cost     times
1  for $j = 2$ to $A.length$                         $c_1$    $n$
2      $key = A[j]$                                  $c_2$    $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1\,..\,j-1]$.                 $0$      $n-1$
4      $i = j - 1$                                   $c_4$    $n-1$
5      while $i > 0$ and $A[i] > key$                $c_5$    $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                           $c_6$    $\sum_{j=2}^{n}(t_j - 1)$
7          $i = i - 1$                               $c_7$    $\sum_{j=2}^{n}(t_j - 1)$
8      $A[i+1] = key$                                $c_8$    $n-1$
```

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

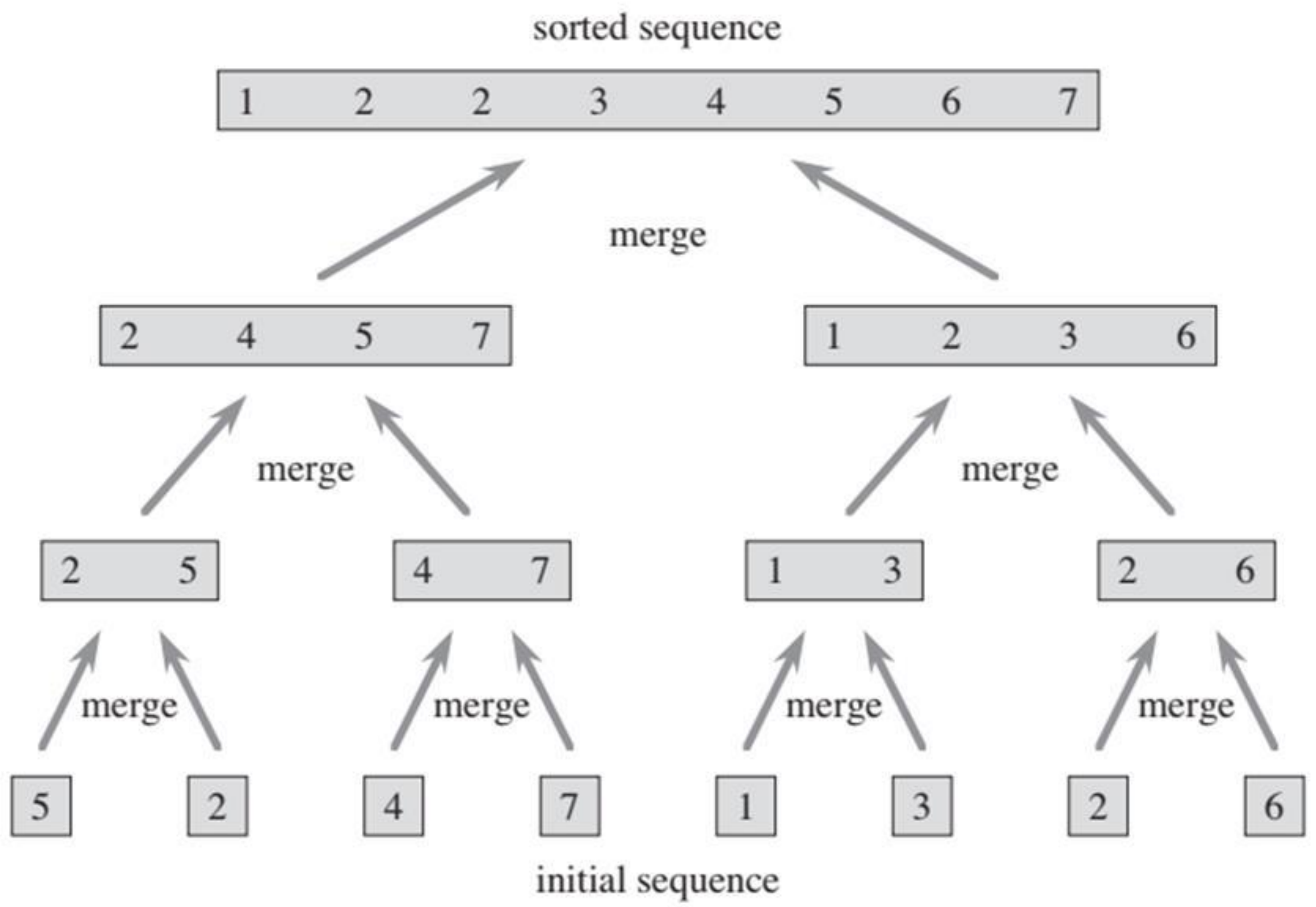
$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \end{aligned}$$

$O(n^2)$

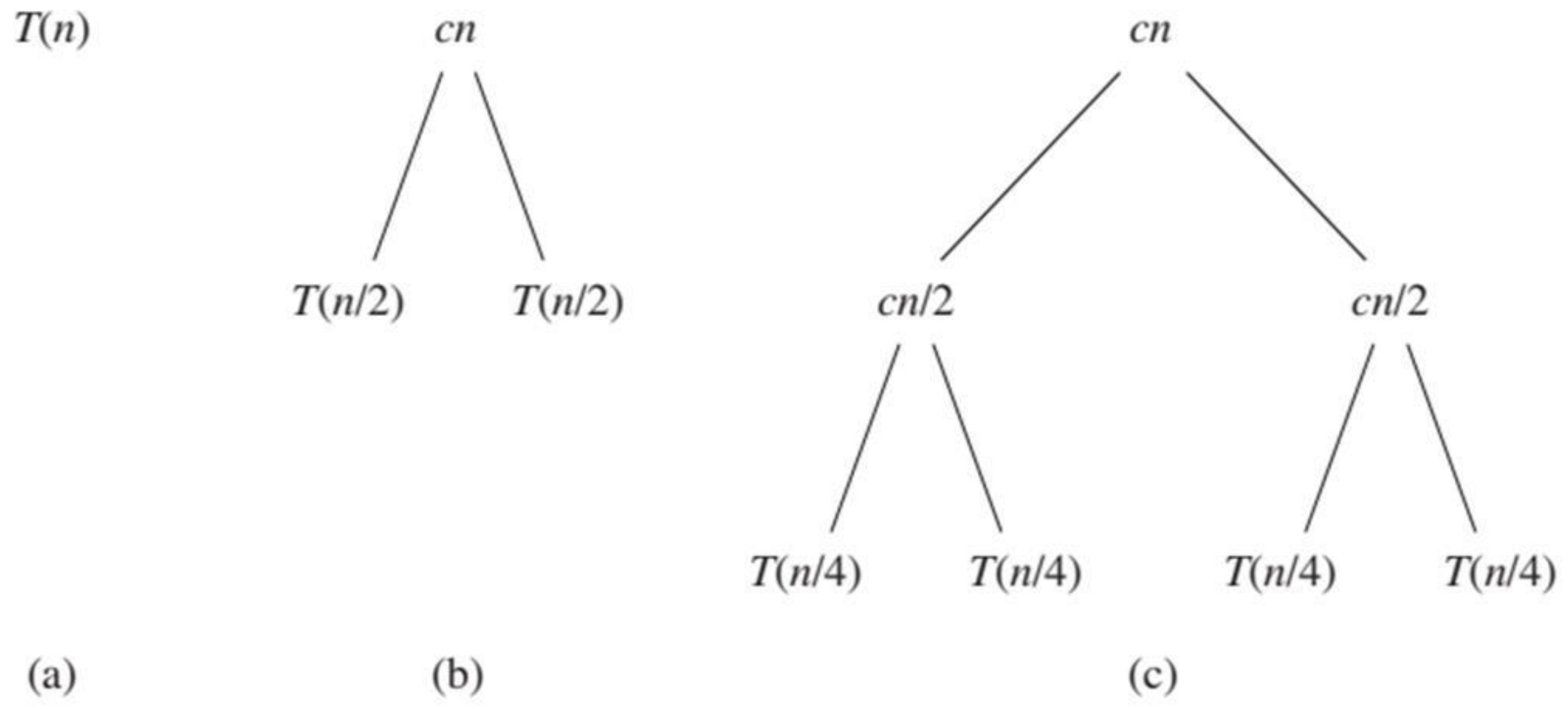
Class Exercise #3

```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT($A, p, q$)
4      MERGE-SORT($A, q + 1, r$)
5      MERGE($A, p, q, r$)
```

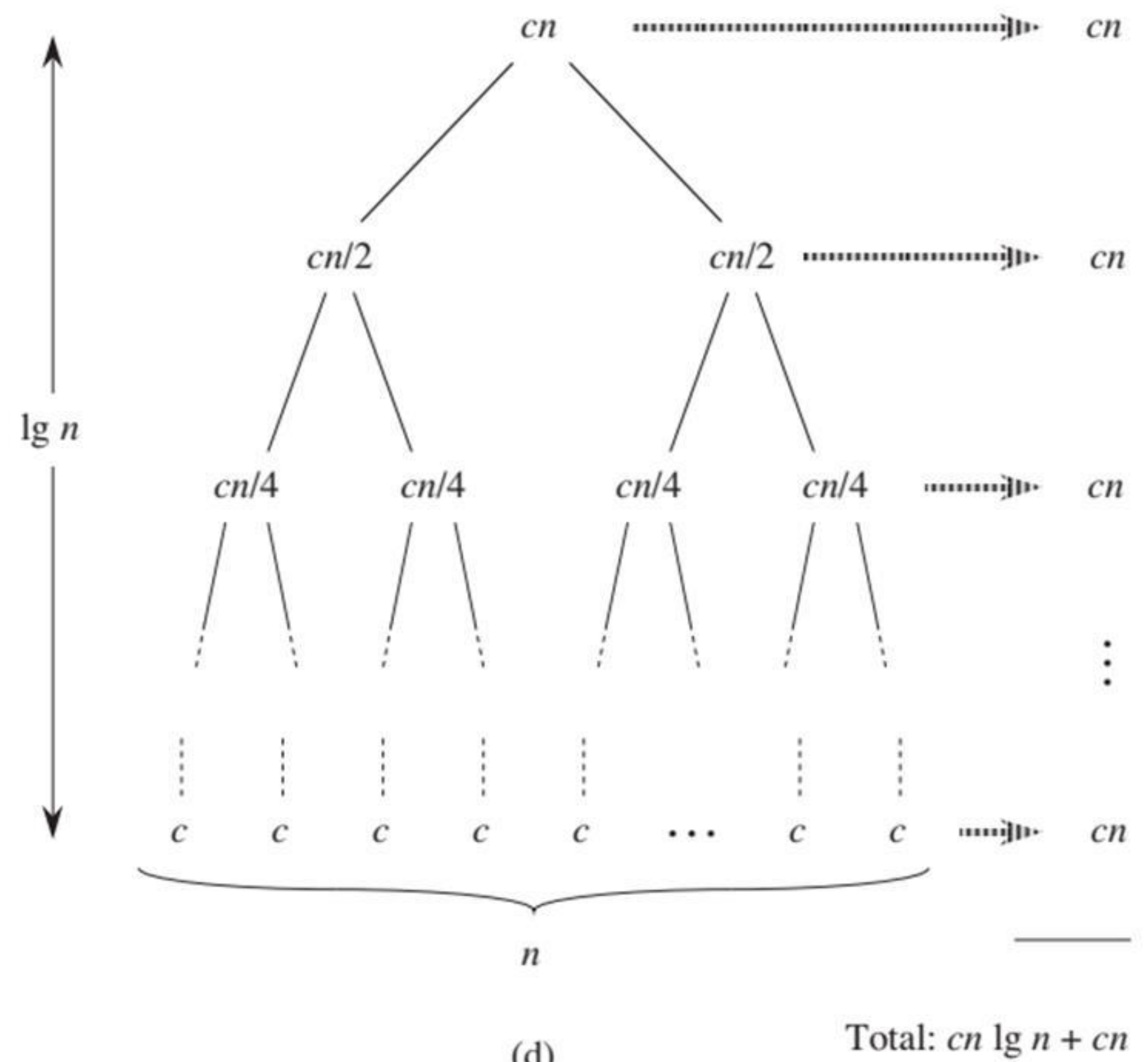
```
MERGE($A, p, q, r$)
 1  $n_1 = q - p + 1$
 2  $n_2 = r - q$
 3  let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
 4  for $i = 1$ to $n_1$
 5      $L[i] = A[p + i - 1]$
 6  for $j = 1$ to $n_2$
 7      $R[j] = A[q + j]$
 8  $L[n_1 + 1] = \infty$
 9  $R[n_2 + 1] = \infty$
10  $i = 1$
11  $j = 1$
12  for $k = p$ to $r$
13      if $L[i] \leq R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16      else $A[k] = R[j]$
17          $j = j + 1$
```



Class Exercise #3



Class Exercise #3



O(n log n)

Masters Theorem

- The ***master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n) \,,$$

where a ≥, b > 1, and f (n) is a given function.

- To use the master method, you will need to memorize three cases, but once you do that, you will easily be able to determine asymptotic bounds for many simple recurrences.

- **Case 1.** If $f(n) = O\left(n^{\log_b a-\epsilon}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
- **Case 2.** If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\log n\right)$.
- **Case 3.** If $f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$ for some $\epsilon > 0$ (and $af\left(\frac{n}{b}\right) \le cf(n)$ for some $c < 1$ for all n sufficiently large), then $T(n) = \Theta(f(n))$.

Masters Theorem

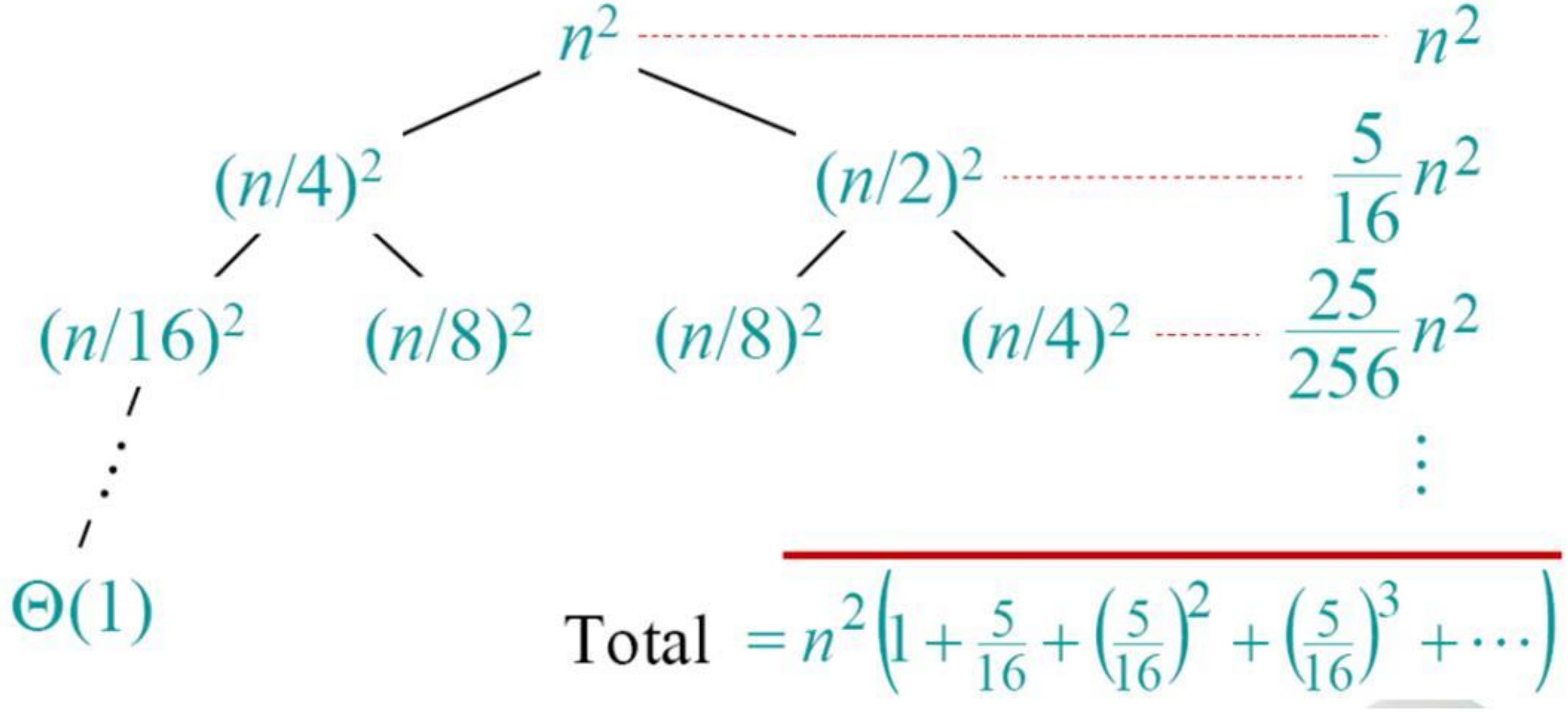
- $T(n)=2T(n/2)+n$.
- $n^{\log_a b}=n$ and f(n) = n, so case 2 of the master theorem gives O(nlogn)
- $T(n)=2T(n/2)+O(1)$
- $n^{\log_a b}$=n, and f(n)=1, $n^{\log_a b}$ which is asymptotically larger than a constant factor, so case 1 of the master theorem gives O(n)
- $T(n)=2T(n/2)+n^2$
- $n^{\log_a b}=n$ and f(n) = n^2, so case 3 of the master theorem gives T(n)= θ(f(n))= θ(n^2)

Masters Theorem

- $T(n)=8T(n/2)+(n^3/\log n)$.
- $n^{\log_a b}=n^3$ and f(n) is smaller than $n^{\log_a b}$ but by less than a polynomial factor. Therefore, the master theorem makes no claim about the solution to the recurrence.
- In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

Recursion tree: Example

Consider recurrence $T(n) = T(n/4) + T(n/2) + n^2$
with base case $T(1) = \Theta(1)$.



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

Recursion tree: Example

- Considering the geometric series

$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \text{ for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \text{ for } |x| < 1$$

we get $T(n) = \Theta(n^2)$