Analysis of Algorithms
CSC 402
2022-23

Subject Incharge

Bidisha Roy

Associate Professor

Room No. 401

email: bidisharoy@sfit.ac.in

Module 1 - Introduction

Analyzing Algorithms

- Why
  - Measuring efficiency of an Algorithm.
- Efficiency checked by
  - Correctness
  - Implementation
  - Simplicity
  - Execution time and Memory space req.
  - New ways of doing same task better.

Time and Space

- Time Complexity
  - Amount of computer time an algorithm needs to execute the program and get the intended result.
- Space Complexity
  - Amount of memory required for running an algorithm.

Types of Analysis

- Priori Analysis
  - Machine independent
  - Done before implementation

- Posteriori Analysis
  - Target Machine dependent
  - Done after implementation

Order of Magnitude

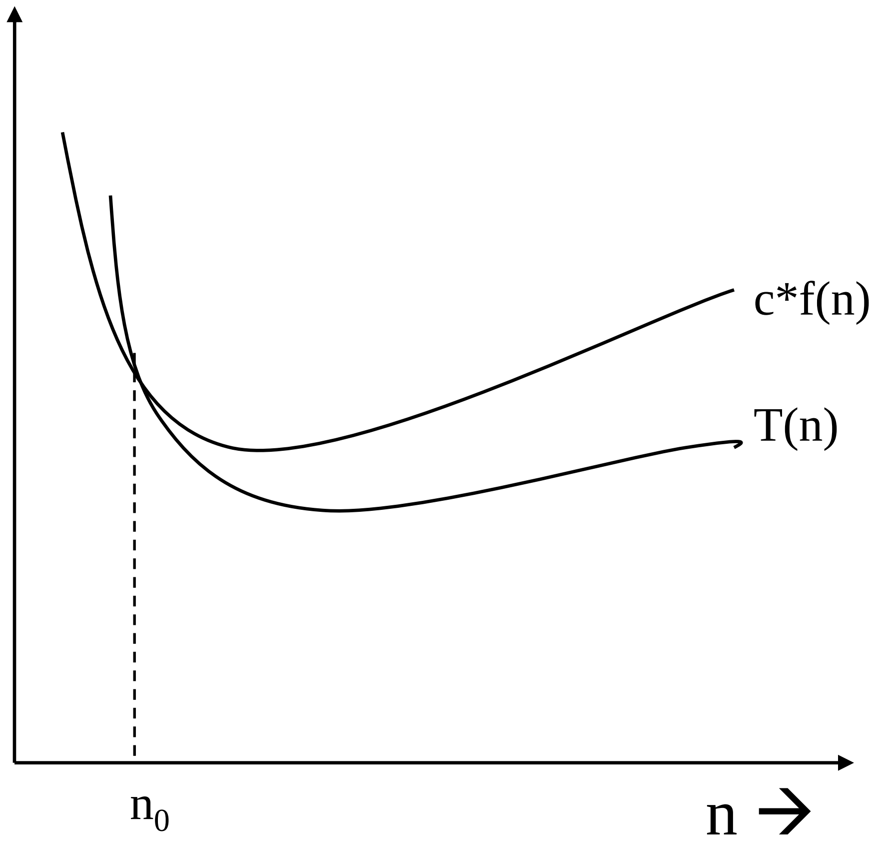
- Order of Magnitude of an Algorithm is the sum of number of occurrences of statements contained in it.
- Other terms
  - Worst case Running Time
  - Best case Running Time
  - Average case Running Time

Asymptotic Notations

- Concerned with how the running time of an algorithm increases with the size of input in the limit, as the size of input increases without bound.
- Notations
  - O notation
  - Ω notation
  - Θ notation

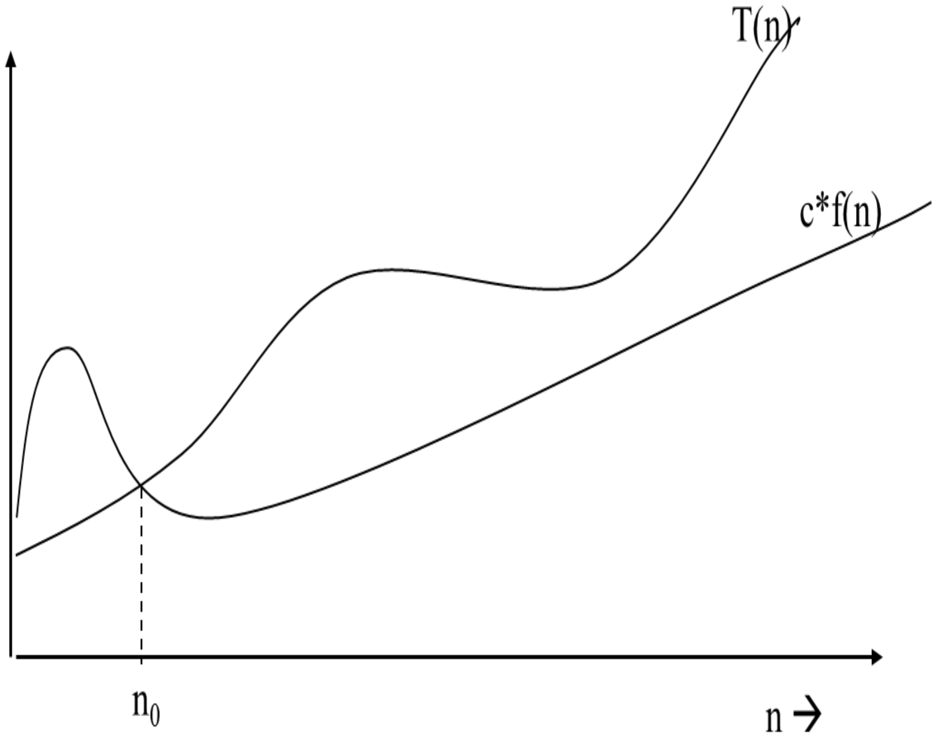
O Notation

- Formally defined as
  - For non-negative functions T(n) and f(n), the function **T(n) = O(f(n))** if there are positive constants **c** and $\mathbf{n_0}$ such that T(n) ≤ c*f(n) for all **n**, $\mathbf{n \geq n_0}$ **(c>0,** $\mathbf{n_0 \geq 1}$**)**
- Known as the ***Big-Oh***
- If graphed, **f(n)** serves as an upper bound to the curve you are analysing.
- Describes the ***worst*** that can happen.



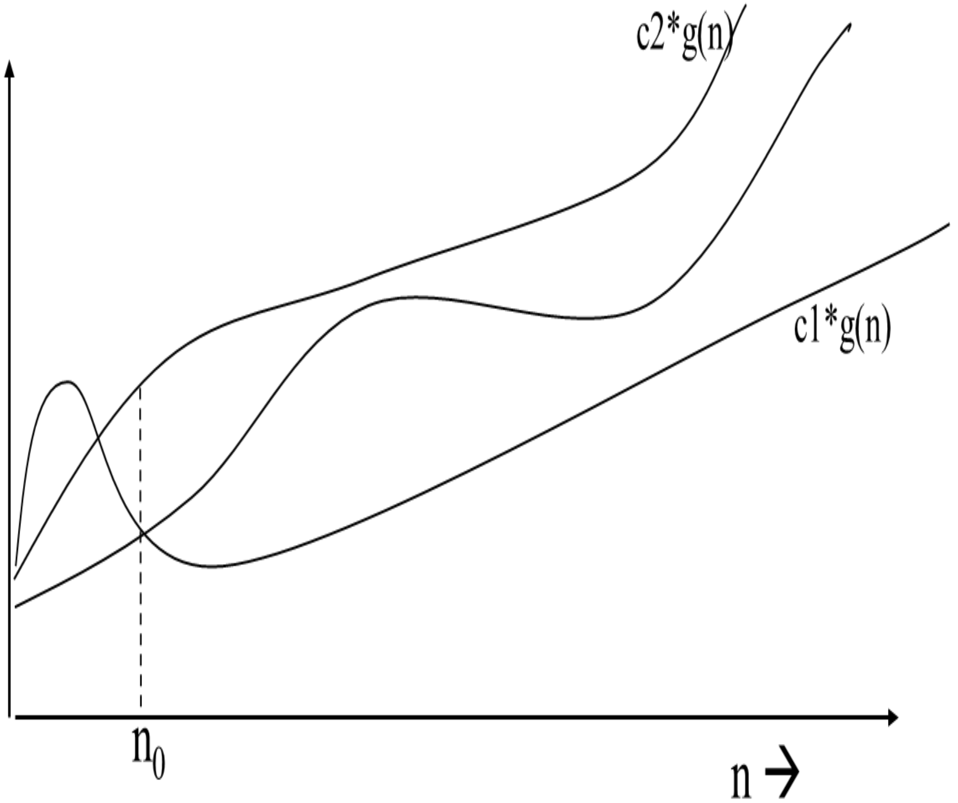
Ω Notation

- Formally defined as
  - For non-negative functions, **T(n)** and **f(n)**, the function **T(n) = Ω(f(n))** if there are positive constants **c** and $\mathbf{n_0}$ such that **T(n) ≥ c*f(n)** for all n, n ≥ $\mathbf{n_0}$. **(c>0,** $\mathbf{n_0}$ **≥ 1)**
- **f(n)** is the lower bound for **T(n).**
- Describes ***best*** that can happen for a given data input size.



Θ Notation

- Formally defined as
  - For non-negative functions, **T(n)** and **g(n),** the function **T(n) = Θ(g(n))** if there exist positive constants **c1, c2** and $\mathbf{n_0}$ such that $\mathbf{c_1 * g(n) \le T(n) \le c_2 * g(n)}$ for all $\mathbf{n, n \ge n_0}$. $\mathbf{(c_1, c_2 > 0, n_0 \ge 1)}$
- Describes the ***average*** case for the input data size **n.**



How to approximate time complexity?

- Algorithms of two types
  - Iterative Algorithms
  - Recursive Algorithms
- Iterative Algorithms
  - Count number of times instructions are executed
- Recursive Algorithms
  - Recursive/recurrence equations

Insertion Sort Pseudocode

```
Insertion_Sort(A, n)

for j ← 2 to n
    key ← A[j]
    i ← j-1
    while i>0 and A[i] >key
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

http://liveexample.pearsoncmg.com/dsanimation/InsertionSortNeweBook.html

```
for j ← 2 to n
    key ← A[j]
    i ← j-1
    while i>0 and A[i] >key
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

Index		1	2	3	4	5	6	7
List		8	6	9	5	1	2	7
Pass 1	j=2, key=6, i=1	6	8					
Pass 2	j=3, key=9, i=2	6	8	9				
Pass 3	j=4, key=5, i=,3,2,1,0	5	6	8	9			
Pass 4	j=5, key=1, i=4,3,2,1,0	1	5	6	8	9		
Pass 5	j=6, key=2, i=5,4,3,2,1	1	2	5	6	8	9	
Pass 6	j=7, key=7, i=6,5,4	1	2	5	6	7	8	9

Insertion Sort (A, n)

Ln No.	Pseudo Code	Cost	Times
1.	for j ← 2 to n	c1	n
2.	key ← A[j]	c2	n-1
3.	i ← j-1	c3	n-1
4.	while i>0 and A[i] >key	c4	$\sum_{j=2}^{n} t_j$
5.	A[i+1] ← A[i]	c5	$\sum_{j=2}^{n} (t_j - 1)$
6.	i ← i-1	c6	$\sum_{j=2}^{n} (t_j - 1)$
7.	A[i+1] ← key	c7	n-1

Assumptions while finding Time Complexity

- The leading constant of highest power of **n** and all lower powers of **n** are ignored in f(n)
- Example for Insertion sort
  - T(n) = O(f(n))
    - Best case f(n) = (c1+c2+c3+c4+c7)n – (c2+c3+c7)
    - Therefore **T(n) = O(n)**

Selection Sort

- Successive elements are selected in order and placed in their proper position.
- An in-place sort.
- Simple to implement
- Works as follows
  - Find the minimum value in the list
  - Swap it with the value in the first position
  - Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Selection Sort Pseudocode

```
Selection_Sort (A,n)

for i← 1 to n
     j ← i
     for k← i+1 to n
          if (A[k]<A[j]) then
             j←k
      swap (A[i],A[j])
```

https://liveexample.pearsoncmg.com/dsanimation13ejava/SelectionSorteBook.html

Index		1	2	3	4	5	6	7
Initial List		8	6	9	5	1	2	7
Pass 1	i=1, j=1, 2, 4, 5 Swap (A[1], A[5])	1	6	9	5	8	2	7
Pass 2	i=2, j=2, 5, 6 Swap (A[2], A[6])	1	2	9	5	8	6	7
Pass 3	i=3, j=3, 4 Swap (A[3], A[4])	1	2	5	9	8	6	7
Pass 4	i=4, j=4, 5, 6 Swap (A[4], A[6])	1	2	5	6	8	9	7
Pass 5	i=5, j=5, 7 Swap (A[5], A[7])	1	2	5	6	7	9	8
Pass 6	i=6, j=6, 7 Swap (A[6], A[7])	1	2	5	6	7	8	9
Pass 7	i=7, j=7	1	2	5	6	7	8	9

```
for i← 1 to n
    j ← i
    for k← i+1 to n
        if (A[k]<A[j]) then
            j←k
    swap (A[i],A[j])
```

Selection Sort (A,n)

Ln No.	Pseudo Code	Cost	Times
1.	for i← 1 to n	c1	n+1
2.	j ← i	c2	n
3.	for k← i+1 to n	c3	n(n+1)/2
4.	if (A[k]<A[j]) then	c4	n(n+1)/2 - 1
5.	j←k	c5	n(n+1)/2 - 1
6.	swap (A[i],A[j])	c6	n

Some Problems

Recursive Algorithms

- Bin_Search(A, target, low, high, n)
  - If (high < low)
    - return not found
  - mid ← low + ((high - low) / 2)
  - if (A[mid] > target)
    - return Bin_Search(A, target, low, mid-1)
  - if (A[mid] < target)
    - return Bin_Search(A, target, mid+1, high)
  - else
    - return mid

Solving Recursive Algorithms

- Recurrence Relations/Substitution Method
  - Substitute the equation for earlier instances
- Recursion Tree
  - Draw a recurrence tree and calculate time taken for each level of tree
- Master's Method
  - Direct Method

Substitution/Recurrence Relation Method

Example 1

- A(n)
  - if (n>1)
    - Return (A(n-1))
  - if (n==1)
    - Return 1
- T(n) = 1 + T(n-1)
- Where
  - T(n-1) – Time taken to execute n-1 inputs

Example 2

- A(n)
  - if (n>1)
    - Return (A(n/2) + A(n/2))
  - if (n==1)
    - Return 1
- T(n) = c + 2T(n/2)
- where
  - c- time taken for constant actions
  - T(n/2) – Time taken to execute A for n/2 inputs

Master's Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size *n* into *a* subproblems, each of size *n*/*b*
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function *f*(n)
- Then, the Master Theorem gives us a cookbook for the algorithm's running time:

Master's Theorem

- If $T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$, then
  - Case I:
    - If **$f(n) = O(n^{\log_b a-\varepsilon})$**, for some constant $\varepsilon > 0$, then **$T(n) = \Theta(n^{\log_b a})$**
  - Case II:
    - If **$f(n) = O(n^{\log_b a})$**, then **$T(n) = \Theta(n^{\log_b a} \log n)$**
  - Case III:
    - If **$f(n) = \Omega(n^{\log_b a+\varepsilon})$** for some constant $\varepsilon > 0$, and if **$af(n/b) \leq cf(n)$** for some constant $c < 1$ and all sufficiently large n, then **$T(n) = \Theta(f(n))$**

Using Master's Theorem

- $T(n) = 9T(n/3) + n$
  - $a=9$, $b=3$, $f(n) = n$
  - $n^{\log b\ a} = n^{\log 3\ 9} = \Theta(n^2)$
  - Since $f(n) = O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon>1$, case 1 applies:

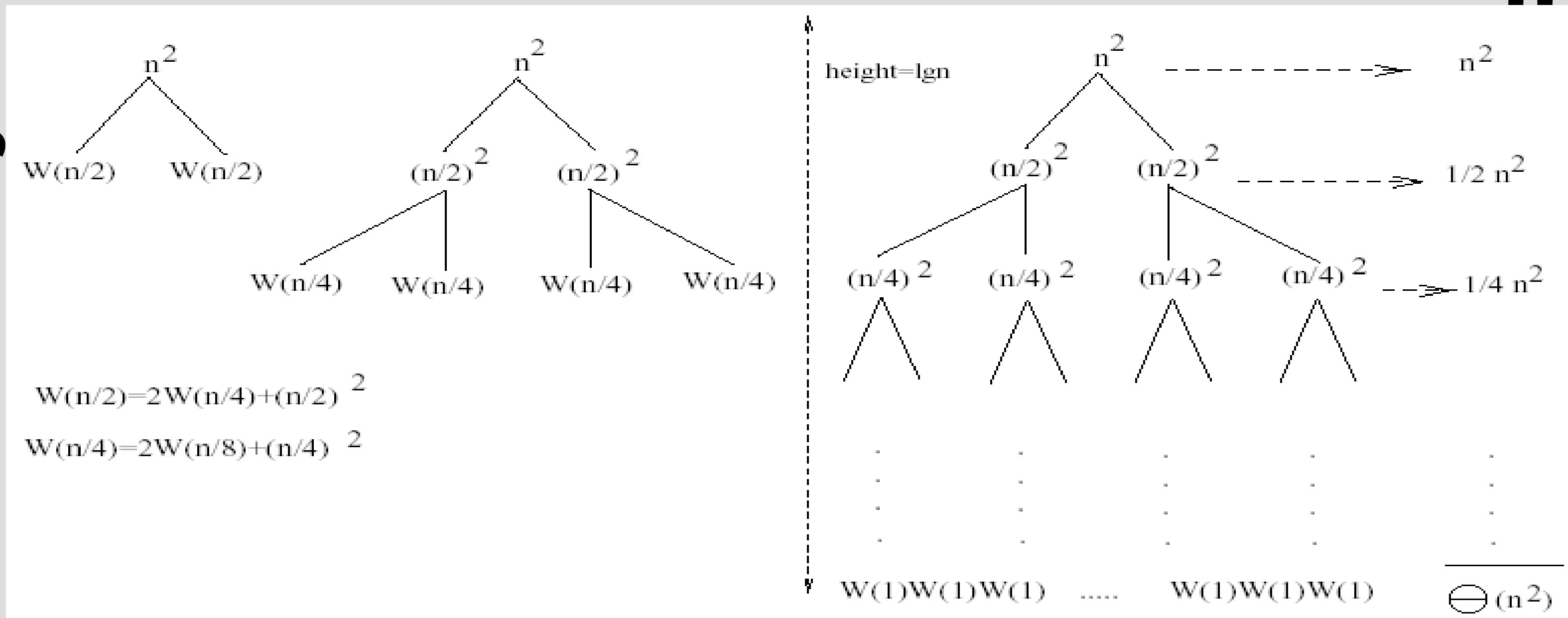
$$T(n) = \Theta\left(n^{\log_b a}\right) \text{when } f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

  - Thus the solution is $T(n) = \Theta(n^2)$

Recurrence Tree Method

- Convert the recurrence into a tree:
  - Each node represents the cost incurred at various levels of recursion
  - Sum up the costs of all levels
  - To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels.
  - The pattern is typically a arithmetic or geometric series.

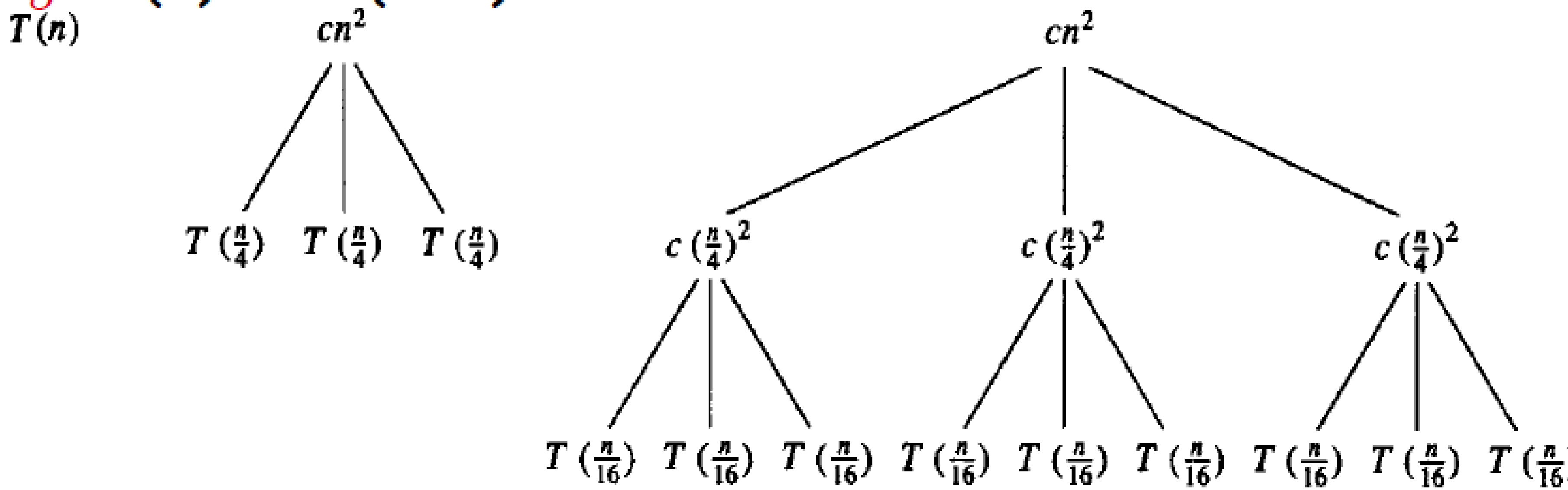
Example: $W(n) = 2W(n/2) + n^2$



- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $1 = n/2^i$ ➔ $i = \log n$
- Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level i = 2^i
- Total Cost

$$W(n) = \sum_{i=0}^{\lg n-1} \frac{n^2}{2^i} + 2^{\lg n} W(1) = n^2 \sum_{i=0}^{\lg n-1} \left(\frac{1}{2}\right)^i + n \le n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + O(n) = n^2 \frac{1}{1-\frac{1}{2}} + O(n) = 2n^2$$

E.g.: $T(n) = 3T(n/4) + cn^2$



- Subproblem size at level i is: $n/4^i$
- Subproblem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a node at level i = $c(n/4^i)^2$
- Number of nodes at level i = $3^i \Rightarrow$ last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes
- Total cost:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \le \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1-\frac{3}{16}} cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

$\Rightarrow T(n) = O(n^2)$

Some common recurrences

Recurrence	**Solution**
$T(n) = T(n/2) + d$	$T(n) = O(\log_2 n)$
$T(n) = T(n/2) + n$	$T(n) = O(n)$
$T(n) = 2T(n/2) + d$	$T(n) = O(n)$
$T(n) = 2T(n/2) + n$	$T(n) = O(n\log_2 n)$
$T(n) = T(n-1) + d$	$T(n) = O(n)$

Next

- **Divide and Conquer**