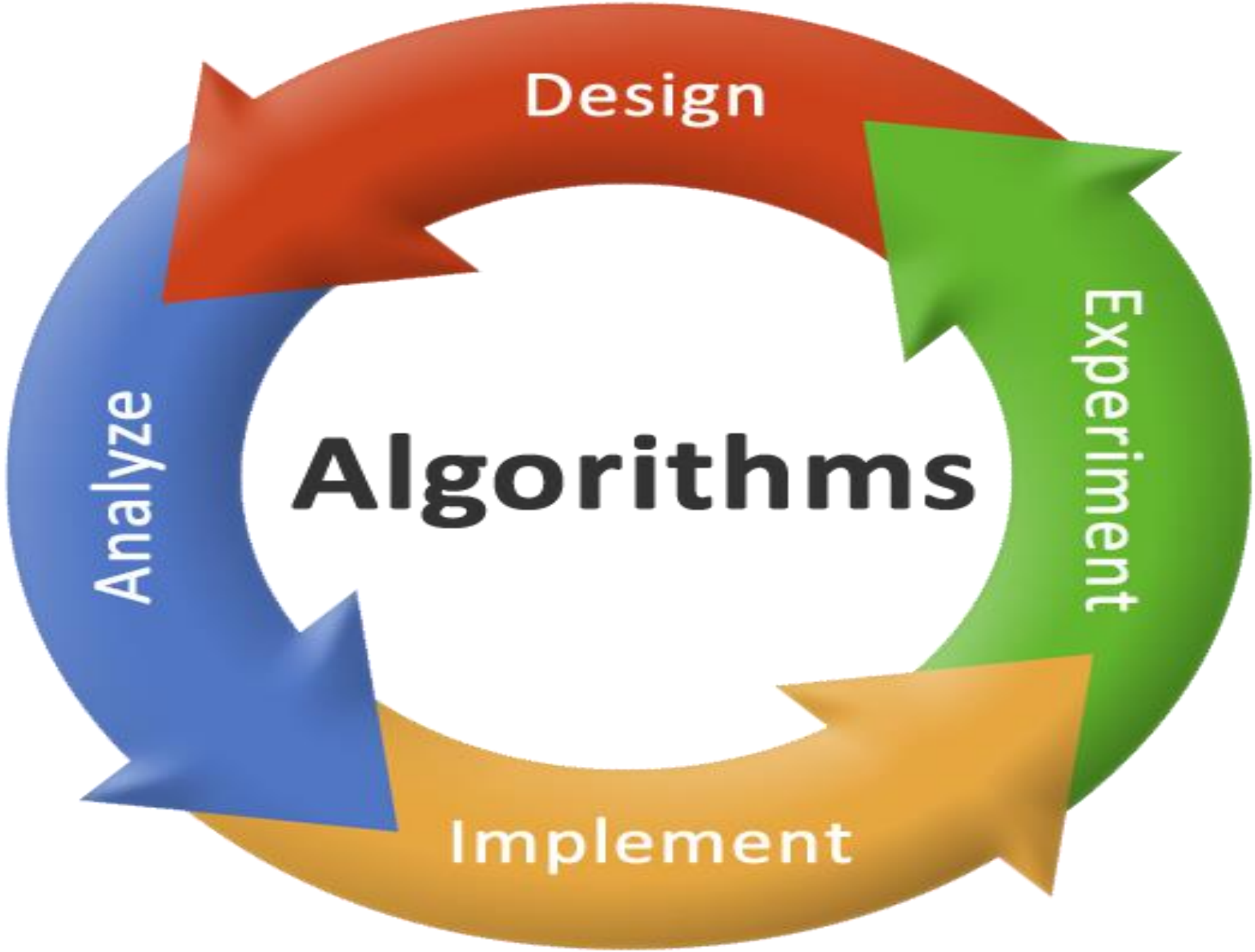
DESIGN AND ANALYSIS OF ALGORITHM

Prof. Sumitra Jakhete


Design
Experiment
Implement
Analyze
Algorithms

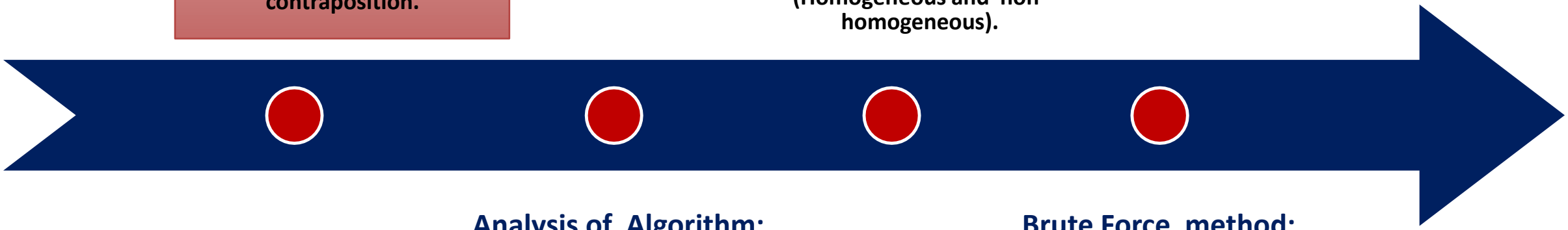
UNIT - I INTRODUCTION 6 HOURS

- **Proof Techniques:** Minimum 2 examples of each: Contradiction, Mathematical Induction, Direct proofs, Proof by counterexample, Proof by contraposition.
- **Analysis of Algorithm:** Efficiency- Analysis framework, asymptotic notations – big O, theta and omega.
- **Analysis of Non-recursive and recursive algorithms**: Solving Recurrence Equations (Homogeneous and non homogeneous).
- **Brute Force method:** Introduction to Brute Force method & Exhaustive search, Brute Force solution to queens' problem

UNIT - I INTRODUCTION 7 HOURS

Proof Techniques:
Minimum 2 of each: Contradiction, Mathematical Induction, Direct proofs, Proof by counterexample, Proof by contraposition.

Analysis of Non- recursive and recursive algorithms:
Solving Recurrence Equations (Homogeneous and non-homogeneous).



Analysis of Algorithm:
Efficiency- Analysis framework, asymptotic notations –big O, theta and omega.

Brute Force method:
Introduction to Brute Force method & Exhaustive search, Brute Force solution to queens' problem.

ALGORITHMS

WHY DO YOU NEED TO STUDY ALGORITHMS?

- Able to design new algorithms and analyze their efficiency for which more than one solutions exist.

The study of algorithms include many important and active areas of research.

- **How to design algorithms:** e.g. dynamic programming.
- **How to validate algorithms**: verifying correct output for every input e.g. mathematical induction etc.
- **How to analyze algorithms:** study of time and space complexity
- **How to test a program: debugging –**It is the process of executing programs on sample data sets to determine whether faulty results occur and if so to correct them**.**

WHAT IS AN ALGORITHM?

- An Algorithm is a **finite ordered** set of unambiguous and effective steps which are when followed ,accomplices a particular task ,by accepting zero or more input quantities and generate at least one output.
- input----→Algorithm---→output

All the algorithms must satisfy the following five characteristics:

- **INPUT:** Zero or set of values supplied to the algorithm.
- **OUTPUT:** After processing input it produces at least one quantity as output.
- **DEFINITENESS:** Each instruction must be clear and certain i.e. unambiguous.(e.g. add 6 or 7 to x, 5/0 –not allowed)
- **FINITENESS:** Algorithm must terminate after some finite number of steps for all cases.
- **EFFECTIVENESS:** The human being can trace the instructions by using paper and pencil.

Program is an expression of an algorithm in a programming language.

TYPES OF ANALYSIS

- **Worst case**
  - Provides an **upper bound on running time**
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- **Best case**
  - Provides a **lower bound on running time**
  - Input is the one for which the algorithm runs the fastest

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

- **Average case**
  - Provides a **prediction about the running time**
  - Assumes that the input is random

HOW DO WE COMPARE ALGORITHMS?

- We need to define a number of objective measures.

(1) Compare **execution times?**

Not good: times are specific to a particular computer !!

(2) **Count the number of statements executed?**

Not good: number of statements vary with the **programming language** as well as the **style** of the individual programmer.

IDEAL SOLUTION

Express **running time** as a function of the **input size** n (i.e., f(n)).

Compare different functions corresponding to running times.

Such an analysis is independent of machine time, programming style, etc.

PERFORMANCE CRITERIA TO JUDGE/TO ANALYZE AN ALGORITHM

Time Complexity

- The **running time** of an algorithm is defined to be an **estimate** of the **number of operations performed by it given a particular number of input values.**
- Let *T(n) be a* ***measure*** of the time required to execute an algorithm of problem size *n*.
- *We call T(n) the* ***time complexity*** **function** of the algorithm.
- If *n is* ***sufficiently*** **small** then the algorithm will not have a long running time.

TIME COMPLEXITY

- **Compile time (C) : time needed for a single execution of the statement** independent of instance characteristics
- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- It is machine dependent.
- **run (execution) time T_P is I** dependent Dependents on the number of instances.
  - It is depends on the no of times the statement is executes, **Frequency count**

EXAMPLE

Algorithm Sum(a,n)	*Cost (s/e)*	*Frequency*
{		
s:=0.0;	1	1
for i=1 to n do	1	n+1
s:=s+ a[i];	1	n
return s;	1	1
}		

Total time required for this algorithm is : f(n) =2n + 3

COMMON TIME COMPLEXITIES

BETTER

WORSE

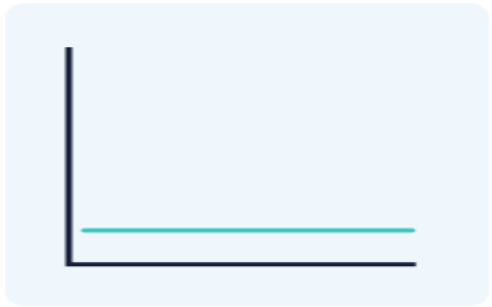
- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

CLASSIFICATION OF TIME COMPLEXITY

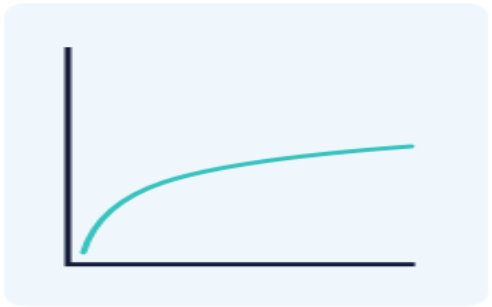
Notation	Complexity	Description	Example
O(1)	Constant	Simple statement	Addition
O(log(n))	Logarithmic	Divide in half	Binary search
O(n)	Linear	loop	Linear search
O(n*log(n))	Linearithmic	Divide & Conquer	Merge sort
$O(n^2)$	Quadratic	Double loop	Check all pairs
$O(n^3)$	Cubic	Triple loop	Check all triples
$O(2^n)$	Exponential	Exhaustive search	Check all subsets
O(n!)	Factorial	Recursive function	Factorial

Common Runtimes

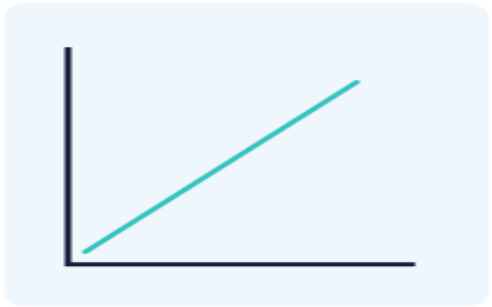
$\Theta(1)$



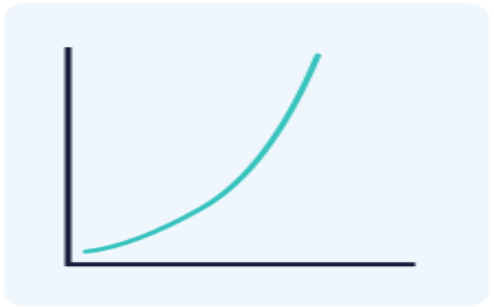
$\Theta(\log N)$



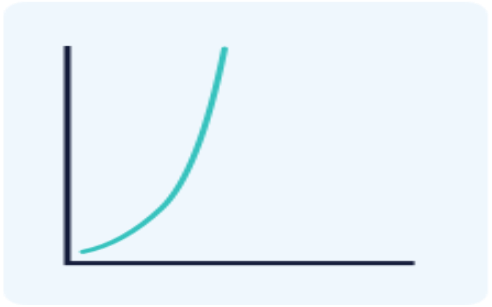
$\Theta(N)$



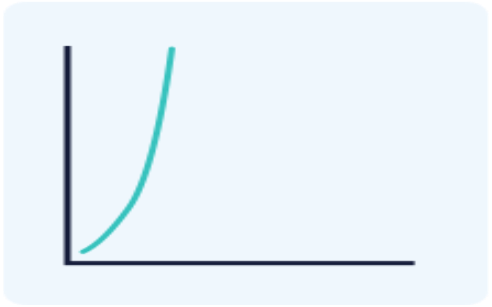
$\Theta(N \log N)$



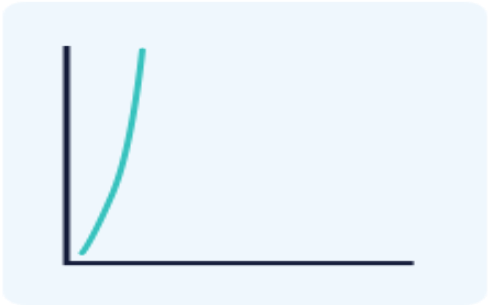
$\Theta(N^2)$



$\Theta(2^N)$



$\Theta(N!)$

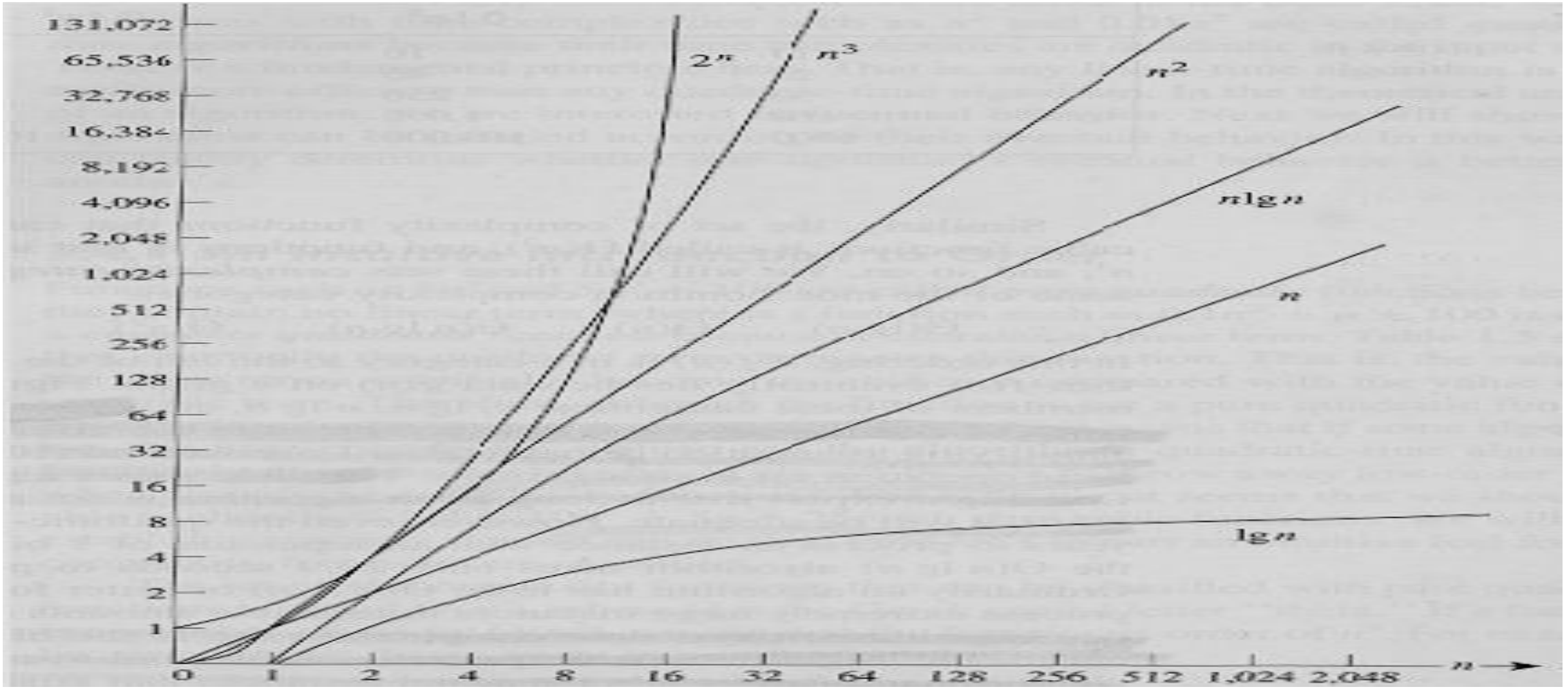


Time complexity

The functions shown in the table below are mainly used to express the running time in terms of the input size (n).

Input Size	(1)	log (n)	n*log(n)	n	n^2	n^3	2^n
5	1	3	15	5	25	125	2^5
1000	1	10	10^4	1000	10^6	10^9	2^{1000}
10000	1	13	10^5	10000	10^8	10^{12}	2^{10000}

COMMON ORDERS OF MAGNITUDE



ASYMPTOTIC ANALYSIS

- To compare two algorithms with running times *f(n)* and *g(n)*, we need a **rough measure** that characterizes **how fast each function grows.**
- *Hint:* **use *rate of growth***
- Compare functions in the limit, that is, **asymptotically!**

  (i.e., for large values of ***n*** *it means* Focus on the growth of complexity w.r.t input size n)

RATE OF GROWTH

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_gold

  Cost ~ cost_of_elephants (approximation)

- The low order terms in a function are relatively insignificant for **large** n

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

RATE OF GROWTH, ANOTHER VIEW

Function growth and weight of terms as a percentage of all terms as n increases for

$$f(n) = n^2 + 80n + 500$$

n	f(n)	n^2	80n	500
10	1,400	100 (7%)	800 (57%)	500 (36%)
100	18,500	10,000 (54%)	8,000 (43%)	500 (3%)
1000	1,0805,000	1,000,000 (93%)	80,000 (7%)	500 (0%)
10000	100,800,500	100,000,000 (99%)	800,000 (1%)	500 (0%)

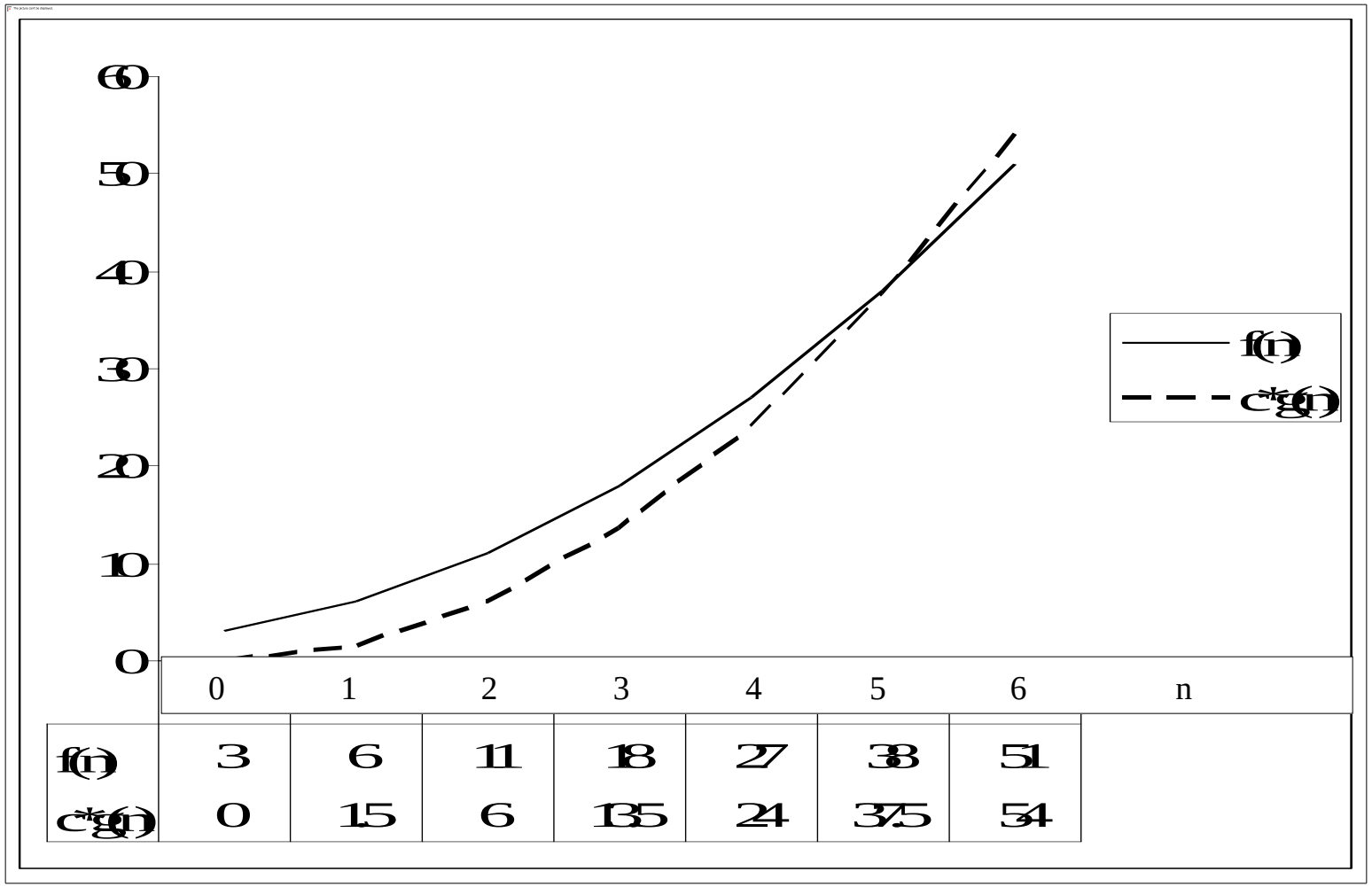
Conclusion: consider highest order term with the **coefficient dropped**, also drop all lower order terms

BIG-O DEFINITION

- **Definition of big-O:**
  - f(n) = **O(g(n))** if and only if there exist **two positive constants c and N** such that
  - 0<= **f(n) <= c*g(n)** for all n > N i.e **g(n) is UPPER BOUND on f(n)**

- Example: $f(n) = n^2 + 2n + 3$ is $O(n^2)$ because when **c = 1.5**, $n^2 + 2n + 3 \leq 1.5N^2$
- when $N \geq 5.1625$: $39.976 \leq 39.977$
- When N = 6, $51 \leq 54$ and when N = 8, $83 \leq 96$

C*G(N) BIGGER THAN F(N) AT 5.1625



f(n)	3	6	11	18	27	38	51
c*g(n)	0	15	6	135	24	375	54

Another View

n	f(n)	c*g(n)
1	6	1.5
2	11	6
3	18	13.5
4	27	24
5	38	37.5
5.1625	39.97641	39.97711
6	51	54
7	66	73.5
8	83	96
9	102	121.5
10	123	150

THE BIG ‘O’ NOTATION

To show that $f(n) = 4n + 5 = O(n)$, we need to produce a constant C such that:

$f(n) \leq C * n$ for all n.

If we try C = 4, this doesn't work because 4n + 5 is not less than 4n. We need C to be at least 9 to cover *all* n. If n = 1, C has to be 9, but C can be smaller for greater values of n (if n = 100, C can be 5). Since the chosen C must work for all n, we must use 9:

$4n + 5 \leq 4n + 5n = 9n$

Since we have produced a constant C that works for all n, we can conclude: $T(4n + 5) = O(n)$.

THE BIG 'O' NOTATION

Find the values of 'C' for the big 'O' notations for the following functions:

1) $f(n) = n^2 + 5n - 1$

2) $f(n) = 2n^7 - 6n^5 + 10n^2 - 5$

THE BIG ‘O’ NOTATION

1: Suppose $f(n) = n^2 + 5n - 1$.

$f(n) = n^2 + 5n - 1 < n^2 + 5n$

$\leq n^2 + 5n^2$ (since $n <= n^2$ for all integers n)

$= 6n^2$

Therefore, if C = 6, we have shown that **$f(n) = O(n^2)$**.

2: Suppose $f(n) = 2n^7 - 6n^5 + 10n^2 – 5$.

$f(n) < 2n^7 + 10n^2$

$\leq 2n^7 + 10n^7$

$= 12n^7$

Thus, with C = 12 and we have shown that **$f(n) = O(n^7)$**

ASYMPTOTIC NOTATIONS

- O notation: asymptotic“less than” **(Asymptotic Upper Bound)**

  f(n)= O (g(n)) implies: f(n) “<=” c* g(n)

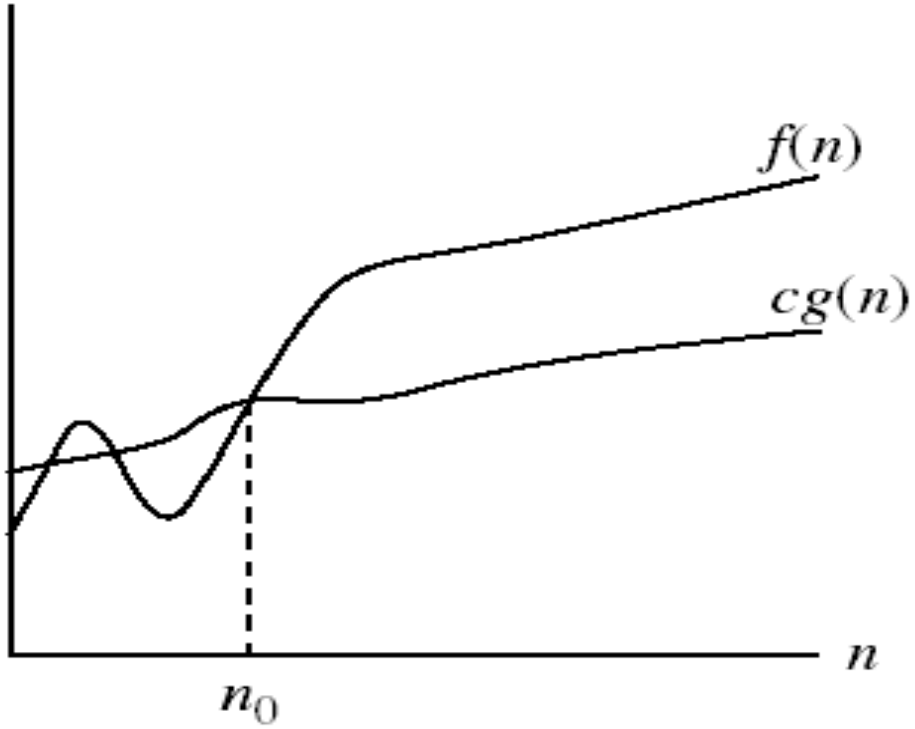
- **Ω notation**: asymptotic“ greater than”:(**Asymptotic Lower Bound)** f(n)= Ω (g(n))

- implies: f(n) “>=” c* g(n)

- **Θ notation**: asymptotic “equality”: **(Asymptotic Tight Bound)** f(n)= Θ (g(n))

  implies: f(n) “=” c* g(n)

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



g(n) is LOWER BOUND on f(n)

f(n) >= c * g(n) for all n,n>= n0

Ω(g(n)) is the set of functions with larger or same order of growth as g(n)

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

- $3n + 2 = \Omega(n)$ as $3n + 2 >= 3n$ for $n >= 1$
- $10\ n^2 + 4n + 2 = \Omega(n^2)$ as $10\ n^2 + 4n + 2 >= n^2$ for $n >= 1$
- We can also observe that $3n + 2 = \Omega(1)$ and

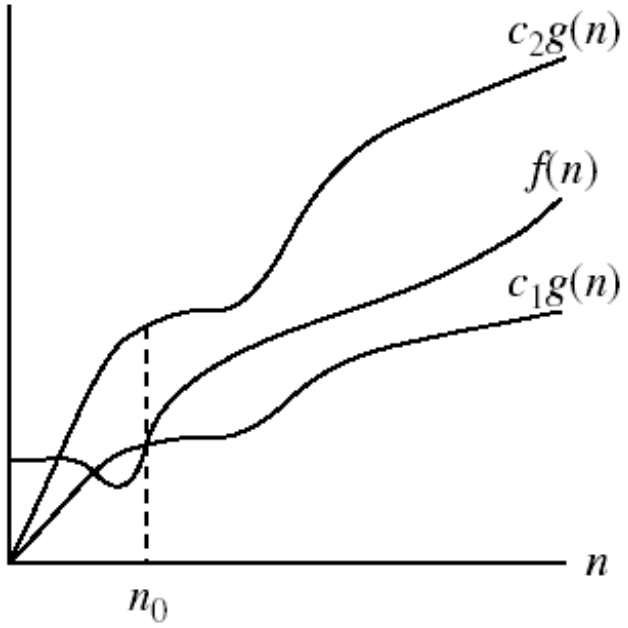
  $10\ n^2 + 4n + 2 = \Omega(n)$

But the function g(n) here is only **LOWER BOUND** on f(n) so while we say that f(n) $3n + 2 = \Omega(n)$ we almost never say that $3n + 2 = \Omega(1)$

ASYMPTOTIC NOTATIONS (CONT.)

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

- *Θ-notation*



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Ex. 3n+2= Θn as 3n+2>=3n and 3n+2<=4n for all n>=2 for c1=3, c2=4, n0=2

Θ(g(n)) is the set of functions with the same order of growth as g(n)

ASYMPTOTIC NOTATIONS

This notation is called "asymptotic" because it deals with the behavior of functions in the limit ,that is for sufficiently large values of its parameter.

Big-"O" notation: Definition :

- The function f(n) = O(g(n)) iff there exists positive constants c > 0 and n0 > 0 such that 0 <= f (n)<= c* g(n) for all n,n >= n0
- O(g(n)) is an upper-bound on the growth of a function, f (n).

- **Prove that if T(n) = 15n^3 + n^2 + 4, T(n) = O(n^3)**.
- **Proof:.** Let c = 20 and n0 = 1.
- Must show that 0 <= f (n) and f (n) <= cg(n).
- 0 <= 15n3 + n2 + 4 for all n>= n0 = 1.
- f (n) = 15n3 + n2 + 4 <= 15n3 + n3 + 4n3
- 15n3 + n3 + 4n3 = 20n3 = 20g(n) = cg(n)

ASYMPTOTIC NOTATIONS:

- **Big-Omega Ω of n**
- The function f(n) =Ω(g(n)) iff there exists positive constants c > 0 and n0 > 0 such that 0 <=c*g(n)<= f (n) for all n,n>= n0.

- Ω(g(n)) is lower-bound on the growth of a function, f(n).
- E.g. **Prove that if T(n) = 15n^3 + n^2 + 4, T(n) = (n^3).**
- **Proof:.**
- Let c = 15 and n0 = 1.
- Must show that 0 <=cg(n) and cg(n)<=f(n).
- 0<=15n^3 for all n>= n0 = 1.
- cg(n)=15n^3<=15n^3 + n^2 + 4 = f(n)

ASYMPTOTIC NOTATIONS:

Big Theta Θ of n:

- The function f(n)=Θ(g(n)) iff there exist positive constants c1>0, c2>0 and n0>0 such that c1g(n)<=f(n)<=c2g(n) for all n,n>=n0.
- Θ(g(n)) is tight bound on the growth of a function, f (n).
- We can also say, f(n) = Θ(g(n)) iff f(n) = O(g(n)) and f(n)=Ω(g(n)).

- **Prove that if T(n) = 15n^3 + n^2 + 4, T(n) = (n^3).**
- **Proof.**
- Let c1=15, c2=20 and n0 = 1.
- Must show that c1g(n)<= f(n) and f (n)<=c2g(n).
- c1g(n) = 15n^3<=15n^3 + n^2 + 4 = f (n).
- f (n)=15n^3 + n^2 + 4<=15n^3 + n^3 + 4n^3 =20n^3=c2g(n).

SPACE COMPLEXITY

- **Space complexity**: The amount of memory it needs to run to completion of algorithm

- The space needed to the algorithms is obtain by the sum of following components

1. **A fixed part**: that is independent of the instance characteristics like input size. this part includes space for instruction i.e code space, space for variables, space for constants etc.
2. **A Variable part**: consists space needed for component variables whose size is dependent on the problem instance being solved, space needed by referenced variables and the recursion stack space,

CONTD…

Thus space requirement

S(P) = c + Sp (instance characteristics)

Ex:

```
Algorithm Sum(a,n)
{
   s:=0.0;
    for i=1 to n do
      s:=s+ a[i];
    return s;
}
```

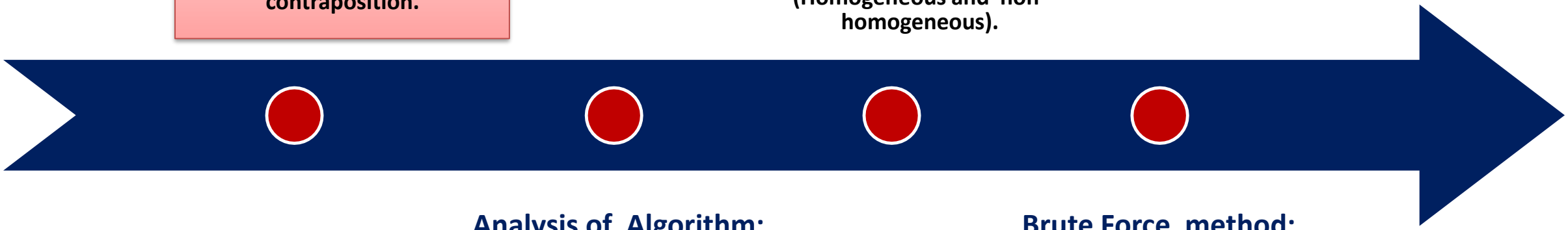
The problem instance for above algorithm is characterized by n, the number of elements to be summed the space needed by this algorithm is obtained as

Sum(n) >= (n+3) (n for a[] , one for n, i & s)

UNIT - I INTRODUCTION 7 HOURS

Proof Techniques:
Minimum 2 of each: Contradiction, Mathematical Induction, Direct proofs, Proof by counterexample, Proof by contraposition.

Analysis of Non- recursive and recursive algorithms:
Solving Recurrence Equations (Homogeneous and non-homogeneous).



Analysis of Algorithm:
Efficiency- Analysis framework, asymptotic notations –big O, theta and omega.

Brute Force method:
Introduction to Brute Force method & Exhaustive search, Brute Force solution to queens' problem.

PROOF TECHNIQUES

Why to study proof techniques?

- To solve a **problem**, **choice** of **algorithms** are available
- To decide which is **best suited** to our specific application by applying some parameters.
- We ***need*** to show that our ***algorithm*** produces a feasible solution.

Various proof techniques: by Contradiction, by Mathematical Induction, direct proofs, proof by counterexample, proof by contraposition

DIRECT PROOF METHOD

- In this method we prove the given statement by making use of existing theorem, algorithm, universally accepted facts. **OR**
- A *direct proof* is a conclusion established by logically combining the laws/rules, definitions, and earlier theorems.

- **E.g.1) The sum of two even integers is itself an even number.**
- Consider two even integers x and y. Since they are even, they can be written as $x = 2a$ and $y = 2b$ respectively for integers a and b. Then the sum $x + y = 2a + 2b = 2(a + b)$. From this it is clear $x + y$ has 2 as a factor and therefore is even, so the sum of any two even integers is even.

EXAMPLES::::

- **Prove that**
- The Sum of any two consecutive integers is odd.
- If the difference of any two integers is even, then so is their sum
- For Integers x,y and z if x divides y and x divides z then x divides y+z

PROOF BY COUNTER EXAMPLE

- Counter example is nothing but the **exception** to the **proposed** or **given statement**.

  eg. All Politicians are corrupted which is related to politician but this statement becomes false if any or a single politician is not corrupted or honest

- In this example uncorrupted or honest politician is counter example to the statement "all politicians are corrupted".

PROOF TECHNIQUES BY COUNTER EXAMPLE…CONT…

- *In math's , by using counter example we can prove complicated theorems very easily and efficiently.*

  Eg.prove the following statement by making use of counter example. If X and Y are even integers then X + y is also an even integer. show that the converse is not true for above statement by making use of counter example.

- (Consider, If x then y and its converse is if y then x).
- If x and y are even integers then x+y is also an even integer its converse is if x+y is an even integer then x,y are also even.
- Here we want to prove that whenever x+y gives even integer then x & y both are even.

PROOF TECHNIQUES

E.g. x+y=18(now there are many possibilities for case of even numbers as follows;

x=2, y=16 or x=16,y=2

X=4,y=14 x=14,y=4

X=6,y=12 x=12,y=6

X=8, y=10 x=10,y=8 ….case I

These cases are true but following cases are also true,

x=1, y=17 or x=17,y=1

X=3,y=15 x=15,y=3

X=5,y=13 x=13,y=5

X=7, y=11 x=11,y=7

X=9,y=9 …..case II

From above case I and II we can say that if x and y both are even integers then x+y is an even integer. But its converse is not true.

PROOF BY CONTRAPOSITION

- In mathematical words we prove the statement by contraposition that x →y then we prove that ¬y→ ¬x i.e negation of y is equivalent to negation of x
- In this method of proof we take advantage of logical equivalence between x→y and ¬y→ ¬x.
- E.g.
- 1) if it is my pen then it is blue --
- If that pen is not blue then it is not my pen.

PROOF BY CONTRAPOSITION

- Prove the following statement by method of contraposition. “If there are two integers x and y and x+y is result in even integer then both x and y are having same parity.
- Sol.:=>if x and y are two integer with apposite parity then x+y gives result as odd integer.
- When we assume that x and y with apposite parity then consider that x is having even value and y is having odd value.
- Let these are two integer P and Q which is related with x and y as follows:
- X=2P and y=2Q +1.
- Now we have to find out value of x+y
- i.e. x+y =2P +2Q +1
- Therefore x+y =2(P+Q)+1
- By definition of odd integer, we say that above result of x+y gives an odd value.
- And hence the poof by contraposition method.

PROOF BY CONTRADICTION (INDIRECT PROOF)

- It consists of demonstrating the truth of a statement by proving that its negation yields a contradiction.
- To prove some statement ‘T’ by contradiction, assume that given statement ‘T’ is false. E.g. Prove the following statement by method of contradiction:
- For all integers n, if n^2 is odd then n is odd.
- The contradiction of above statement is “for all integer n if n^2 is odd then n is even. By definition of an even integer we can say that
- n=2x for some integer value x. So by substitution we get following equation. n*n =2x*2x
  n*n=2(2* x*x)We know product of two integer value is also an integer value i.e. 2.x.x is an integer value. Since 2 & x are two integer values. Hence from this consideration we derive following equation.
- n.n =2.some integer value n^n=2.some integer value.
- So by definition of even no n^2 is even because n is even i.e. n^2 means product of two integers n and n, But this contradicts our assumption that n^2 odd when n is even. Hence our assumption becomes false, so it is proved.

PROOF BY MATHEMATICAL INDUCTION

- **Mathematical induction** is method of mathematical proof typically used to **establish** that a given statement is **true** of all natural numbers
- Steps of MI
- **1)Induction base:** n=1,2,3,… using basic values prove the case.
- **2)Induction hypothesis :** assume the result to be true for some intermediate values.(<=n)
- **3)Induction step:** prove the result for n+1 i.e show the result of left hand side equals to right hand side. (result should be known to us)
- E.g. Solve the following example by MI method:
- ∑ i=n(n+1)/2 , where (1<=i<=n)

PROOF TECHNIQUES: MI

Solution: given eq. is ∑ i=n(n+1)/2 where (1<=i<=n)

step 1 : induction base: let n=1, lhs=∑ i where (1<=i<=n), put values of n=1 in above lhs eq., LHS=∑ i=1 …..(1)

Now put the values of n=1 in RHS eq. RHS =n(n+1)/2=1(1+1)/2=2/2=1… (2)

From eq. (1) and (2) we can say that LHS=RHS.

Similarly for n=2,3,… we can show that lhs=rhs

Step 2: Induction hypothesis:

Asume that the result is true for all the values of i<=n

∑ i=n(n+1)/2 , where (1<=i<=n)

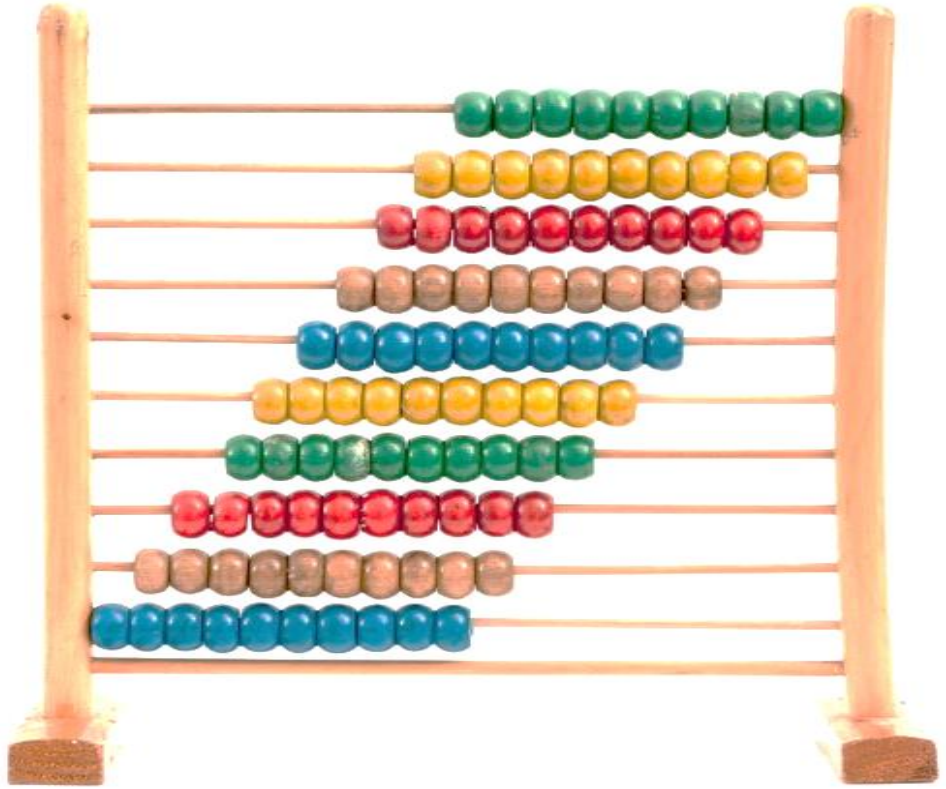
Step 3:Induction step: prove the result for i=k+1

- ∑ i=(k+1)(k+2)/2 where (1<=i<=k+1<=n)
- LHS =∑ i, where (1<=i<=k+1<=n)
- LHS =1+2+3+4+…+k+k+1
- LHS =k(k+1)/2 +k+1
- LHS =(k+1)(k/2+1)
- LHS =(k+1)(k+2)/2
- So LHS=RHS, hence the proof.

SUMMARIZING PROOF TECHNIQUES

Proof Technique	Approach to prove $P \rightarrow Q$	Remarks
Exhaustive Proof	Demonstrate $P \rightarrow Q$ for all cases	May only be used for finite number of cases
Direct Proof	Assume P, deduce Q	The standard approach- usually the thing to try
Proof by Contraposition	Assume Q', derive P'	Use this Q' if as a hypothesis seems to give more ammunition then P would
Proof by Contradiction	Assume $P \wedge Q'$, deduce a contradiction	Use this when Q says something is not true
Serendipity	Not really a proof technique	Fun to know

Brute Force Algorithms



Brute Force Algorithms – trying every possibility rather than advanced techniques to improve efficiency.

Also called –
“Exhaustive search”

Basic Idea

- A straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved.
- **Example**

  Computing $\mathbf{a^n}$ based on the definition of exponentiation:

  $$\mathbf{a^n = a^* a^* a^* \ldots . * a} \qquad (a > 0, n \text{ a nonnegative integer})$$

Brute-Force Approach

Find all the possible solutions and then select the feasible solutions out of them.

Exhaustive search: definition

- A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as a permutations, combinations, or subsets of a set.

- Given an initial state, a goal state, and a set of operations, find a sequence of operations that transforms the initial state to the goal state.

Exhaustive search: method

- Construct a way of listing all potential solutions to the problem in a systematic manner
  - all solutions are eventually listed
  - no solution is repeated
- Evaluate solutions one by one, perhaps disqualifying infeasible ones and keeping track of the best one found so far
- When search ends, announce the winner

Examples of brute force approach

1. Sequential search
2. String matching algorithm
3. Travelling sales man problem
4. Knapsack problem

Brute Force Method

You are given a string “s” and s pattern “p”, you need to check if the pattern is there in the string.

S = “prodevelopertutorial”

P = “rial”

We need to check if “rial” is present in “prodevelopertutorial” string.

We shall use brute force approach to solve this problem.

https://www.prodevelopertutorial.com/introduction-to-brute-force-approach-with-example/

```
#include<iostream>
#include<string>
using namespace std;
bool search(string str, string pattern)
{
	int n = str.length();
	int m = pattern.length();
	for (int i = 0; i <= n - m; i++)
	{
		int j;
		for (j = 0; j < m; j++)
		{
		return true;
	}	if (str[i+j] != pattern[j])
				b==reak;
		}
		if (j m)

		return false;
}
```

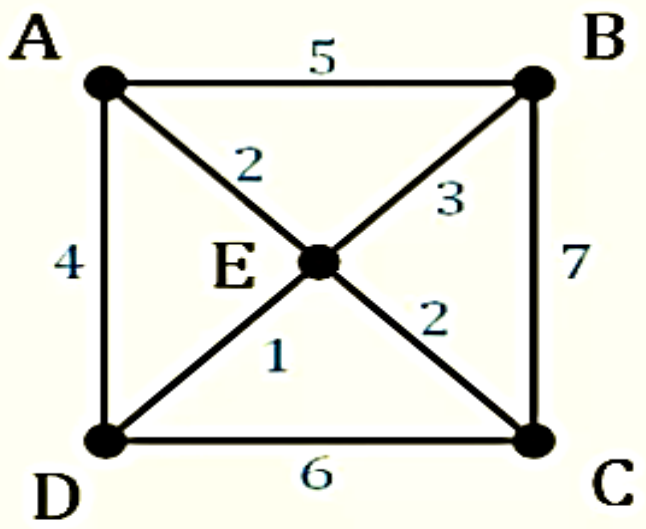
```
int main()
{
	string str = "prodevelopertutrial";
	string pattern = "rial";

	if(search(str, pattern))
	{
		cout<<"The substring is present"<<endl;
	}
	else
	{
		cout<<"The substring is NOT present"<<endl;

	}

	return 0;
}
```

Brute Force – Example 2



Circuit	Weight
ABCDEA	5+7+6+1+2=21
ABCEDA	5+7+2+1+4=19
ABECDA	5+3+2+6+4=20
ADCBEA	4+6+7+3+2=22
ADCEBA	4+6+2+3+5=20
ADECBA	4+1+2+7+5=19
AEBCDA	2+3+7+6+4=22
AEDCBA	2+1+6+7+5=21

- **Find lowest cost Hamiltonian circuit.**

Brute Force – Example 3

- A classic example in computer science is the traveling salesman problem (TSP). Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which those cities should be visited such that the total distance travelled is minimized?

- The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.

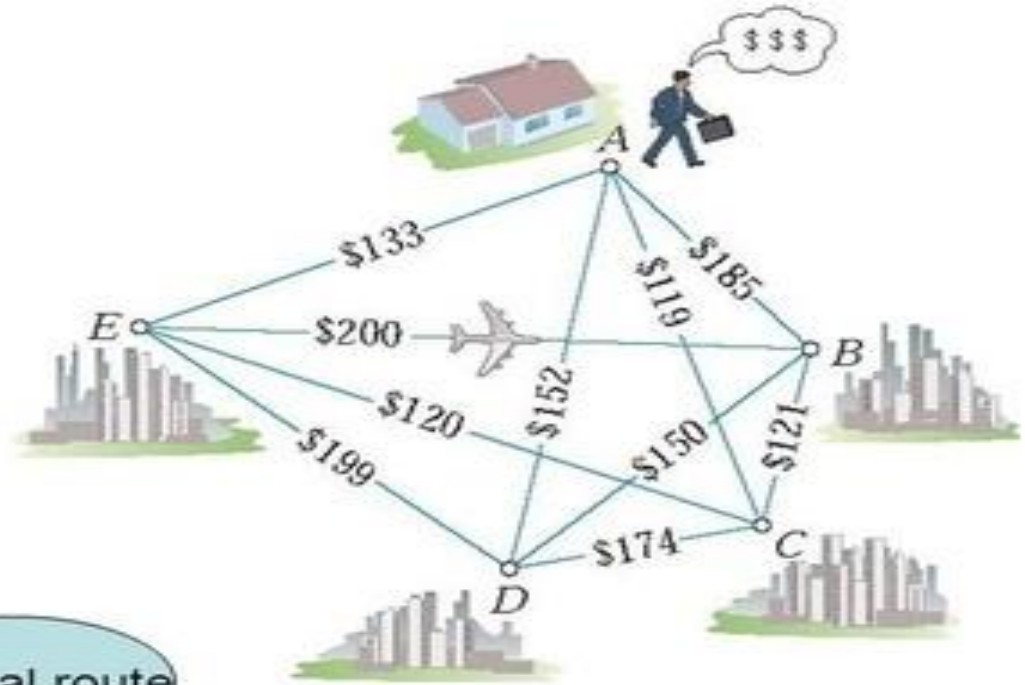
The Brute-Force Algorithms

There are 5 vertices so we have (5-1)! = 24 Hamilton circuits

1) A,B,C,D,E,A = 812
2) A,B,C,E,D,A = 777
3) A,B,D,C,E,A = 762
4) A,B,D,E,C,A = 773
5) A,B,E,C,D,A = 831
6) A,B,E,D,C,A = 877
7) A,C,B,D,E,A = 722
8) A,C,B,E,D,A = 791
9) A,C,D,B,E,A = 776
10) A,C,E,B,D,A = 741
11) A,D,B,C,E,A = 676
12) A,D,C,B,E,A = 780

Plus 12 mirror images

Optimal route



Brute Force – Example 4

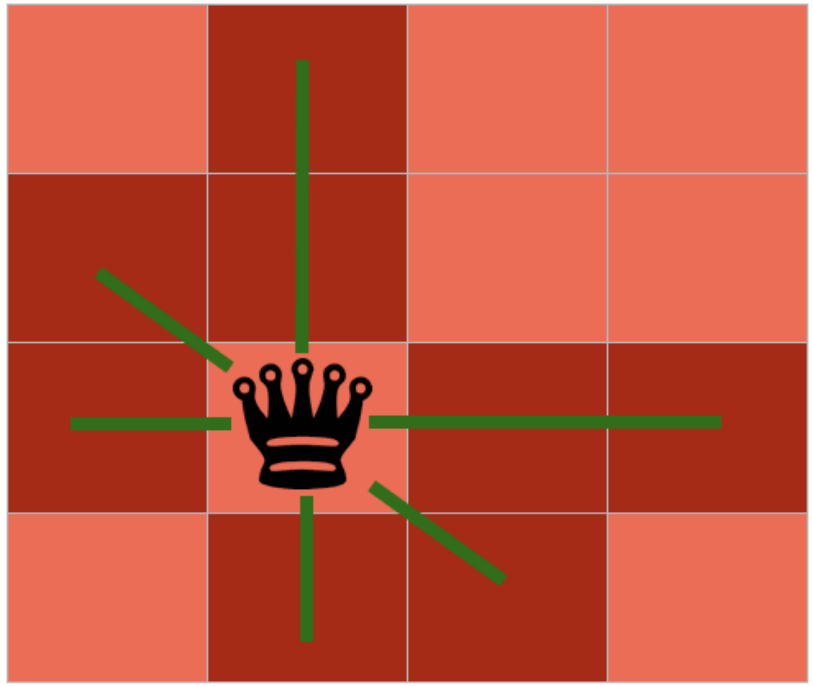
- In cryptography, a brute-force attack involves systematically checking all possible keys until the correct key is found.

- This strategy can in theory be used against any **encrypted data** (except a one-time pad) by an **attacker** who is unable to take advantage of any weakness in an encryption system that would otherwise make his or her task easier.

Brute Force – Example 5

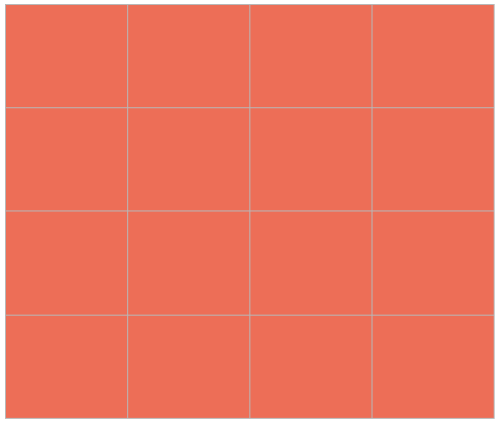
- Example: Consider a chess playing program, if this checks all the possible moves and then checks simulates for each possibility of opponent moves and does this so on for each move , this would be a Brute Force approach, but this might take approximately years to make a single move.

program to find the solution of placing n queens on the chessboard so that no two queens attack each other.



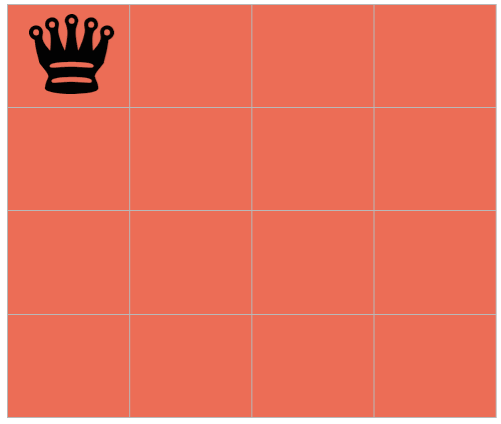
Cells attacked by the queen

- The basic idea of the brute force algorithm is to place the queens on all possible positions and check each time if the queens cannot capture each other.
- If not then it has found a solution.
- Because of the vast amount of possible positions (N^N for a table of size N while each row has 1 queen), this algorithm is not practical even for small table sizes (like N=12).

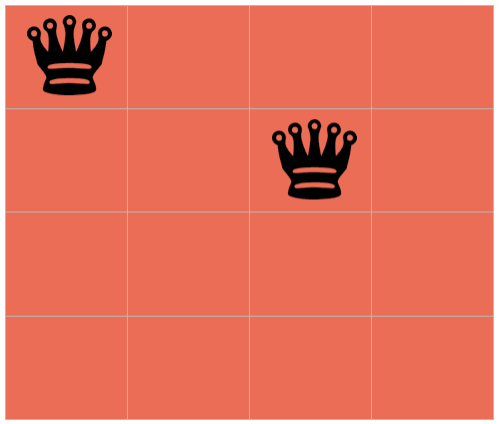


4x4 Chessboard

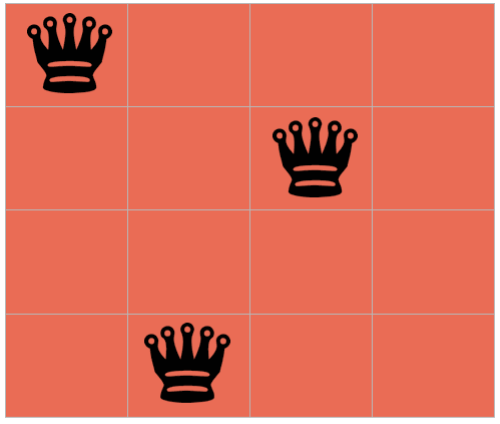
Queens to be placed



Queens to be placed

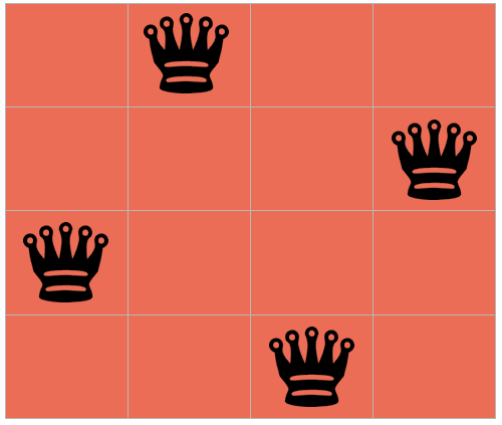


Queens to be placed



No safe position left

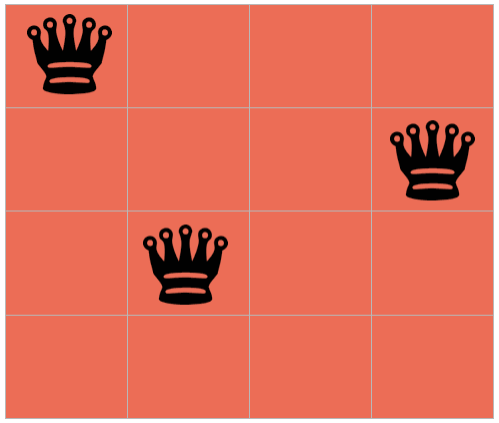
Queens to be placed



Removed

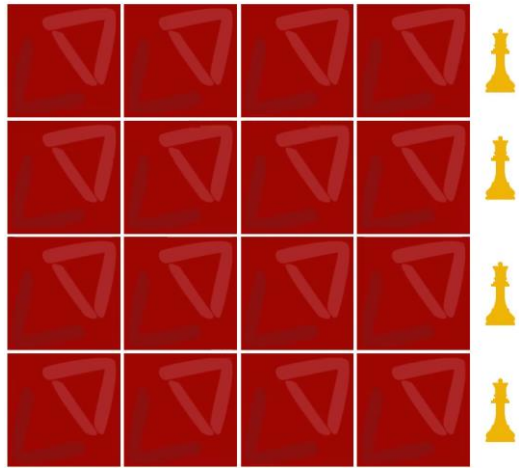
Queens to be placed

Position Changed



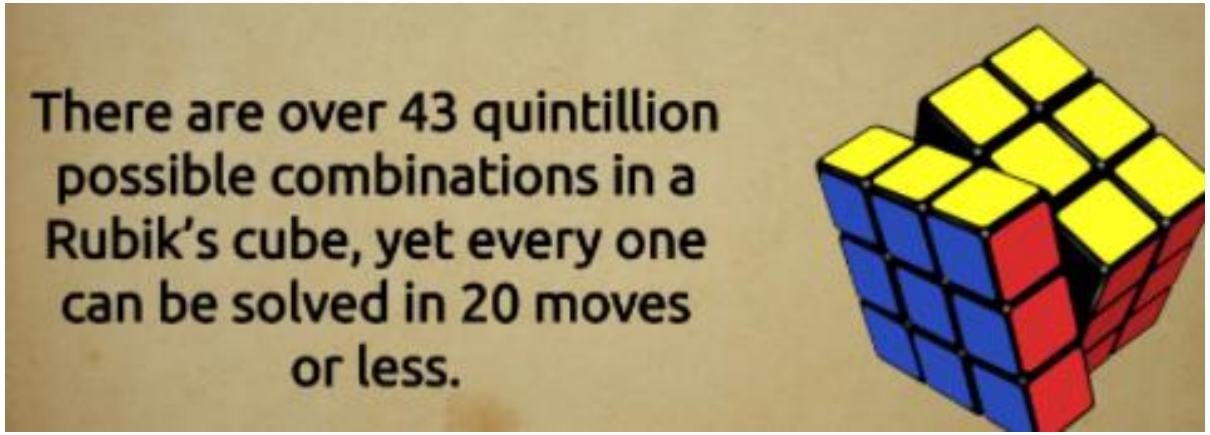
Queens to be placed

Ref. https://www.codesdope.com/course/algorithms-backtracking/

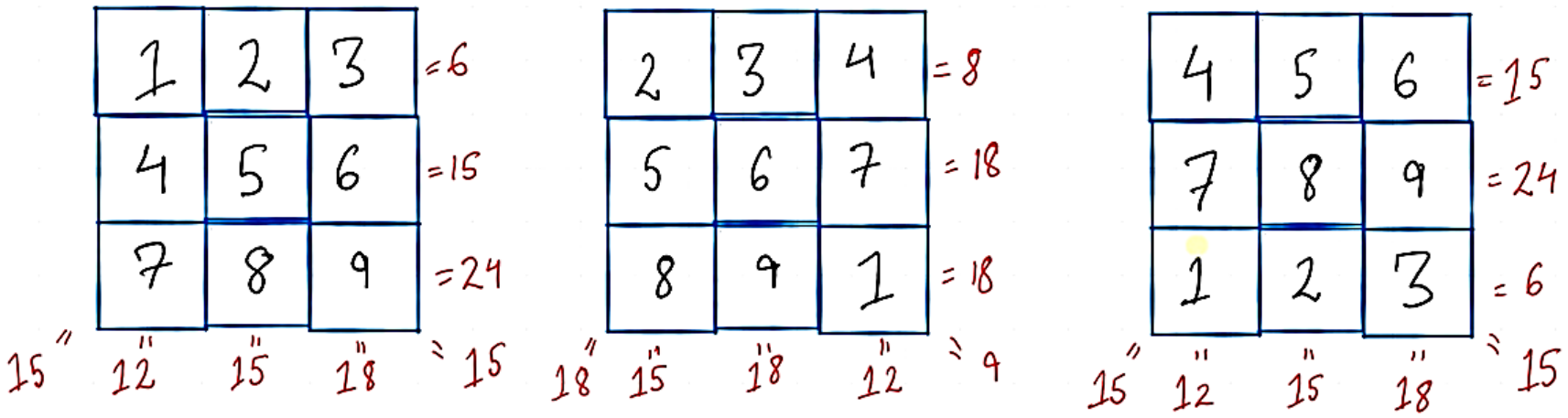


Brute Force – Example 6

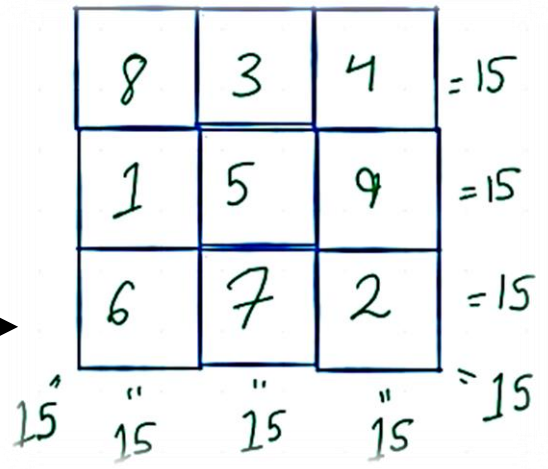
- Rubic Cube



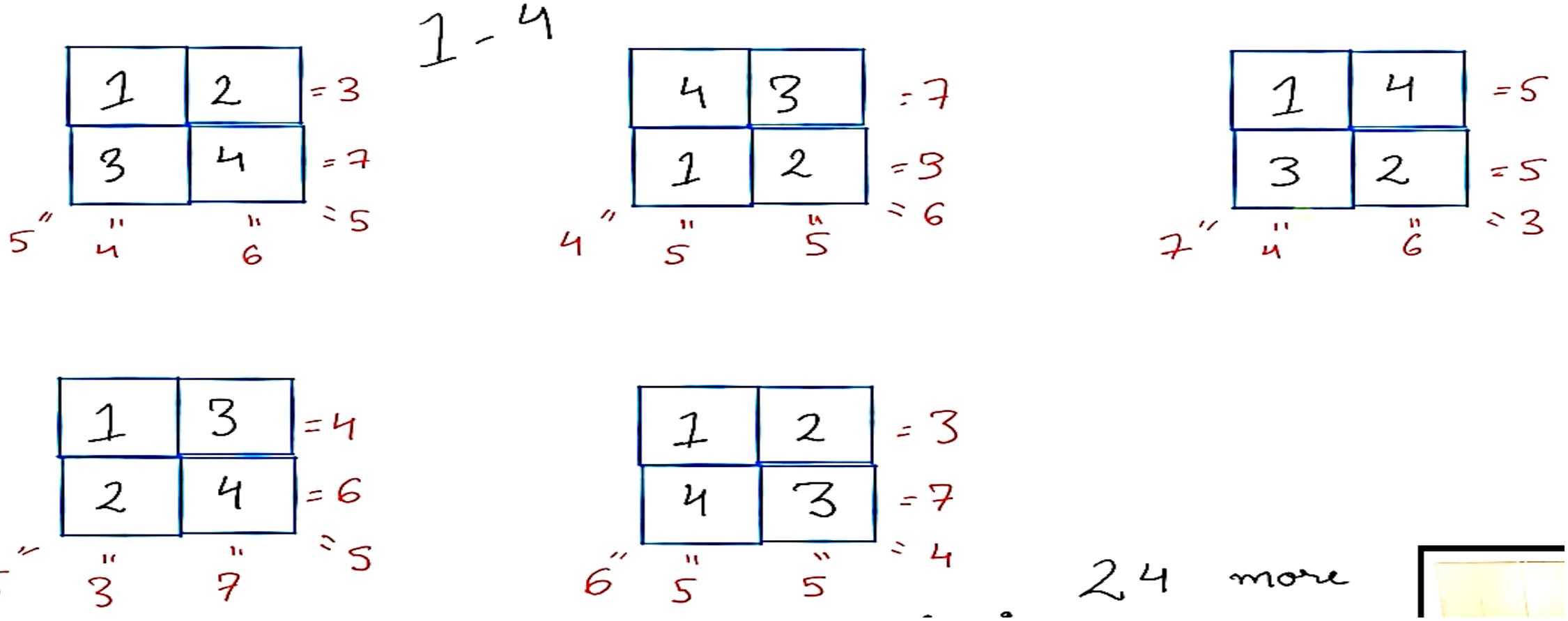
Brute Force – Example 7



Takes 32,000 possibilities



Always try to apply brute force method first to find solution.



Time complexity of Brute Force

- The time complexity of brute force is **O(mn)**, which is sometimes written as **O(n*m)** .
- So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

```
c ← first(P)
while c ≠ Λ do
    if valid(P,c) then
        output(P, c)
    c ← next(P, c)
end while
```

Pros of Brute Force Algorithm

Pros:

- The brute force approach is a guaranteed way to **find** the **correct solution** by listing all the possible candidate solutions for the problem.
- It is a **generic method** and **not limited to any specific** domain of problems.
- The brute force method is **ideal for solving small and simpler** problems.
- It is known for its **simplicity** and can serve as a **comparison** benchmark.

Cons of Brute Force Algorithm

- The brute force approach is **inefficient**. For real-time problems, algorithm analysis often goes above the ***O(N!)*** **order of growth**.
- This method relies more on **compromising** the **power** of a computer system for solving a problem than on a good algorithm design.
- Brute force algorithms are **slow**.
- Brute force algorithms are **not constructive or creative** compared to algorithms that are constructed using some other design paradigms

Solving Recurrence Relations (Homogeneous and Non- homogeneous)

Outline

- What is a Recurrence Relation?
- Forming Recurrence Relations
- Solving Recurrence Relations
- Examples
  - Analysis Of Recursive Factorial Method
  - Analysis Of Recursive Selection Sort
  - Analysis Of Recursive Binary Search
  - Analysis Of Recursive Towers of Hanoi Algorithm

Recurrences and Running Time

- An equation or inequality that describes a function in terms of its value on smaller inputs.

$$T(n) = T(n-1) + n$$

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
  - Find an explicit formula of the expression
  - Bound the recurrence by an expression that involves n

Example Recurrences

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$ $\Theta(\lg n)$
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$ $\Theta(n)$
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

RECURRENCE RELATIONS

Recurrence Relations can take many forms, and most forms are hard, if not impossible to solve. There are however a certain subset that can be solved explicitly. They are of the form:

This is a linear, homogeneous recurrence relation of degree k with constant coefficients

-linear because we don't have terms: $F(a_{n-k})$ with F(.) a nonlinear function.
-homogeneous because we don't have terms: $G(n)$.
-degree k, because it depends on k terms in the past a(n-1) ... a(n-k).
- Note that the equation: $a_n = c_k a_{n-k}$ is also of degree k (some c's are zero).

-constant coefficients because c(k) does not depend on n.

RECURRENCE RELATIONS

Examples:

$a_n = a_{n-1} + (a_{n-2})^2$ non-linear

$b_n = rb_{n-1}$ no constant coefficients

$H_n = 2H_{n-1} + 1$ non-homogeneous

Reminder: *There can be more than one solution to a recurrence relation! Only when the initial conditions are specified is the solution unique.*

For a recurrence relation of degree k, you need k contiguous initial conditions.

What is a Recurrence Relation?

- A recurrence relation, T(n), is a recursive function of integer variable n.
- Like all recursive functions, it has both recursive case and base case.
- Example:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn + c & \text{if } n > 1 \end{cases}$$

- The portion of the definition that does not contain T is called the **base case** of the recurrence relation; the portion that contains T is called the **recurrent or recursive case**.

- Recurrence relations are useful for expressing the running times (i.e., the number of basic operations executed) of recursive algorithms

Forming Recurrence Relations

- For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- Example 1: Write the recurrence relation for the following method.

```
public void f (int n) {
   if (n > 0) {
       System.out.println(n);
       f(n-1);
   }
}
```

- The base case is reached when n == 0. The method performs one comparison. Thus, the number of operations when n == 0, T(0), is some constant a.
- When n > 0, the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter n – 1.
- Therefore the recurrence relation is:

$$T(0) = a \quad \text{where a is constant}$$
$$T(n) = b + T(n-1) \quad \text{where b is constant, } n>0$$

Forming Recurrence Relations

- Example 2: Write the recurrence relation for the following method.

```
public int g(int n) {
   if (n == 1)
      return 2;
   else
      return 3 * g(n / 2) + g( n / 2) + 5;
}
```

- The base case is reached when n == 1. The method performs one comparison and one return statement. Therefore, T(1), is constant **c**.
- When n>1, the method performs TWO recursive calls, each with the parameter n/2, and some constant # of basic operations.
- Hence, the recurrence relation is:

$$T(1) = c \qquad \text{for some constant } c$$
$$T(n) = b + 2T(n/2) \qquad \text{for a constant } b$$

Solving Recurrence Relations

- To solve a recurrence relation T(n) we need to derive a form of T(n) that is not a recurrence relation. Such a form is called a "*closed form*" of the recurrence relation.

- There are four methods to solve recurrence relations that represent the running time of recursive methods:
  - Substitution method (*unrolling and summing)*
  - Recursion tree method
  - Master method

Solving Recurrence Relations –Substitution Method

- Steps:
  - Expand the recurrence
  - Express the expansion as a summation by plugging the recurrence back into itself until you see a pattern.
  - Evaluate the summation
- In evaluating the summation one or more of the following summation formulae may be used:
- Arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- Geometric Series:

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} (x \neq 1)$$

•Special Cases of Geometric Series:

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \quad \text{if } x < 1$$

Solving Recurrence Relations

- Harmonic Series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- Others:

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$

Analysis Of Recursive Factorial Method

- Example1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

```
long factorial (int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial (n – 1);
}
```

T(0) = c

T(n) = b + T(n - 1)

= b + b + T(n - 2)

= b +b +b + T(n - 3)

…

= kb + T(n - k)

When k = n, we have:

T(n) = nb + T(n - n)

= bn + T(0)

= bn + c.

Therefore method factorial is O(n).

Analysis Of Recursive Binary Search

```
public int binarySearch (int target, int[] array,
                         int low, int high) {
   if (low > high)
      return -1;
   else {
      int middle = (low + high)/2;
      if (array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
     }
}
```

- The recurrence relation for the running time of the method is:

  $T(1) = a$ if $n = 1$ (one element array)

  $T(n) = T(n / 2) + b$ if $n > 1$

Analysis Of Recursive Binary Search

Expanding:

$T(n) = T(n / 2) + b$

$= [T(n / 4) + b] + b = T(n / 2^2) + 2b$

$= [T(n / 8) + b] + 2b = T(n / 2^3) + 3b$

$= \ldots\ldots\ldots$

$= T(n / 2^k) + kb$

When $n / 2^k = 1 \rightarrow n = 2^k \rightarrow k = \log_2 n$, we have:

$T(n) = T(1) + b \log_2 n$

$= a + b \log_2 n$

Therefore, Recursive Binary Search is **O(log n)**

Analysis Of Recursive Selection Sort

```
public static void selectionSort(int[] x) {
   selectionSort(x, x.length - 1);}
private static void selectionSort(int[] x, int n) {
   int minPos;
   if (n > 0) {
      minPos = findMinPos(x, n);
      swap(x, minPos, n);
      selectionSort(x, n - 1);
   }
}
private static int findMinPos (int[] x, int n) {
   int k = n;
   for(int i = 0; i < n; i++)
      if(x[i] < x[k])  k = i;
   return k;
}
private static void swap(int[] x, int minPos, int n) {
   int temp=x[n]; x[n]=x[minPos]; x[minPos]=temp;
}
```

Analysis Of Recursive Selection Sort

- findMinPos is O(n), and swap is O(1), therefore the recurrence relation for the running time of the selectionSort method is:

  T(0) = a

  T(n) = T(n – 1) + n + c n > 0

  = [T(n-2) +(n-1) + c] + n + c = T(n-2) + (n-1) + n + 2c

  = [T(n-3) + (n-2) + c] +(n-1) + n + 2c= T(n-3) + (n-2) + (n-1) + n + 3c

  = T(n-4) + (n-3) + (n-2) + (n-1) + n + 4c

  = ……

  = T(n-k) + (n-k + 1) + (n-k + 2) + …….+ n + kc

  When k = n, we have : $T(n) = T(0) + 1 + 2 + \ldots.. + n + nc$

$$= a + \sum_{i=0}^{n} i + cn$$

$$= a + (\frac{n(n+1)}{2}) + cn$$

$$= \frac{n^2}{2} + (c + \frac{1}{2})n + a$$

Therefore, Recursive Selection Sort is **$O(n^2)$**

Analysis Of Recursive Towers of Hanoi Algorithm

```
public static void hanoi(int n, char from, char to, char temp){
   if (n == 1)
      System.out.println(from + " --------> " + to);
   else{
      hanoi(n - 1, from, temp, to);
      System.out.println(from + " --------> " + to);
      hanoi(n - 1, temp, to, from);
   }
}
```

- The recurrence relation for the running time of the method **hanoi** is:

$$T(n) = a \quad \text{if } n = 1$$

$$T(n) = 2T(n - 1) + b \quad \text{if } n > 1$$

The **Tower of Hanoi** (also called **The problem of Benares Temple** or **Tower of Brahma** or **Lucas' Tower** and sometimes pluralized as **Towers**, or simply **pyramid puzzle**) is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:

1.Only one disk may be moved at a time.

2.Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.

3.No disk may be placed on top of a disk that is smaller than it.

Analysis Of Recursive Towers of Hanoi Algorithm

Expanding:

$T(n) = 2T(n-1) + b$

$= 2[2T(n-2) + b] + b \quad = 2^2 T(n-2) + 2b + b$

$= 2^2 [2T(n-3) + b] + 2b + b \quad = 2^3 T(n-3) + 2^2 b + 2b + b$

$= 2^3 [2T(n-4) + b] + 2^2 b + 2b + b \quad = 2^4 T(n-4) + 2^3 b + 2^2 b + 2^1 b + 2^0 b$

$= \ldots\ldots$

$= 2^k T(n-k) + b[2^{k-1} + 2^{k-2} + \ldots 2^1 + 2^0]$

$$= 2^k T(n-k) + b\sum_{i=0}^{k-1} 2^i$$

$$= 2^k T(n-k) + b(2^k - 1)$$

When k = n – 1, we have:

$$T(n) = 2^{n-1} T(1) + b(2^{n-1} - 1)$$
$$= (a+b)\, 2^{n-1} - b$$
$$= \left(\frac{a+b}{2}\right) 2^n - b$$

Therefore, The method **hanoi** is **$O(2^n)$**

SOLVE T(N)=2T(N/2)+N USING SUBSTITUTION METHOD

- **Sol :** T(n)=2T(n/2)+n….
- Given recurrence
- = 2[2T(n/4)+n/2] +n
- =4T(n/4)+2n
- = 4[2T(n/8) +n/4] +2n
- =8T(n/8) +3n
- =8[2T(n/16)+n/8] +3n
- =16T(n/16) +4n

In general ,T(n)=2^i.T(n/2^i)+in for any log(n) >=1 putting i=log(n) we get

T(n)=2^log(n).T(n/2^log(n)) +nlog(n)

T(n)=nT(1)+nlog(n)

But T(1)=2,so T(n)=2n+nlog(n)

Thus T(n)=θ(nlog(n))

MASTER THEORM

- Master's Theorem is a popular method for solving the recurrence relations
- It is also known as a **table-look up method**.
- Master's theorem solves recurrence relations of the form

$$T(n) = a\,T\left(\frac{n}{b}\right) + \theta(n^k \log^p n)$$

Master's Theorem

- Here, a >= 1, b > 1, k >= 0 and p is a real number.

T(n)=aT(n/b)+f(n)

- If f(n) is polynomial of degree d
- $T(n)= \theta(n^d \log n)$, if $a=b^d$
- $T(n)= \theta(n^{\log_b a})$, if $a>b^d$
- $T(n)= \theta(n^d)$, if $a<b^d$

Master Threom Cases

- To solve recurrence relations using Master's theorem, we compare **a with b^k.**
- Then, we follow the following case
- **Case-01:**
- If **$a > b^k$**, then $T(n) = \theta (n^{\log_b a})$
- **Case-02:**
- If **$a = b^k$** and
  - If **$p < -1$**, then $T(n) = \theta (n^{\log_b a})$
  - If **$p = -1$**, then $T(n) = \theta (n^{\log_b a} . \log\log n)$
  - If **$p > -1$**, then $T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$
- **Case-03:**
- If $a < b^k$ and
  - If $p < 0$, then $T(n) = O (n^k)$
  - If $p >= 0$, then $T(n) = \theta (n^k \log^p n)$

Problem-01:

Solve the following recurrence relation using Master's theorem- $T(n) = 3T(n/2) + n^2$

- **Solution-**
- We compare the given recurrence relation with

  $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
- Then, we have- **a = 3, b = 2, k = 2, p = 0**
- Now, $a = 3$ and $b^k = 2^2 = 4$.
- Clearly, $a < b^k$.
- So, we follow **case-03**.
- Since p = 0, so we have-
- $T(n) = \theta (n^k \log^p n)$
- $T(n) = \theta (n^2 \log^0 n)$
- Thus,

$$\mathbf{T(n) = \theta (n^2)}$$

Problem-02:

Solve the following recurrence relation using Master's theorem-

$T(n) = 2T(n/2) + n\log n$

- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta(n^k \log^p n)$.
- Then, we have- **a = 2 b = 2 k = 1 p = 1**
- Now, $a = 2$ and $b^k = 2^1 = 2$.
- Clearly, $a = b^k$.
- So, we follow case-02.
- Since $p = 1$, so we have-
- $T(n) = \theta(n^{\log_b a} . \log^{p+1} n)$
- $T(n) = \theta(n^{\log_2 2} . \log^{1+1} n)$
-
- Thus

$T(n) = \theta(n\log^2 n)$

Problem-03:

Solve the following recurrence relation using Master's theorem-

$T(n) = 2T(n/4) + n^{0.51}$

- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (nklogpn)$.
- Then, we have- $a = 2$ $b = 4$ $k = 0.51$ $p = 0$
- Now, $a = 2$ and $bk = 40.51 = 2.0279$.
- Clearly, $a < bk$.
- So, we follow case-03.
- Since $p = 0$, so we have-
- $T(n) = \theta (nklogpn)$
- $T(n) = \theta (n0.51log0n)$
-
- Thus,

$$T(n) = \theta (n0.51)$$

Problem-04:

Solve the following recurrence relation using Master's theorem-

$T(n) = \sqrt{2}T(n/2) + \log n$

- **Solution-**
- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
- Then, we have- $a = \sqrt{2}$ $b = 2$ $k = 0$ $p = 1$
- Now, $a = \sqrt{2} = 1.414$ and $b^k = 2^0 = 1$.
- Clearly, $a > b^k$.
- So, we follow case-01.
- So, we have-
- $T(n) = \theta (n^{\log_b a})$
- $T(n) = \theta (n^{\log_2 \sqrt{2}})$
- $T(n) = \theta (n^{1/2})$
- Thus,

$$\mathbf{T(n) = \theta (\sqrt{n})}$$

Problem-05:

Solve the following recurrence relation using Master's theorem-

$T(n) = 3T(n/3) + n/2$

- **Solution-**
- We write the given recurrence relation as $T(n) = 3T(n/3) + n$.
- This is because in the general form, we have θ for function f(n) which hides constants in it.
- Now, we can easily apply Master's theorem.
- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
- Then, we have- $a = 3$ $b = 3$ $k = 1$ $p = 0$
- Now, $a = 3$ and $b^k = 3^1 = 3$.
- Clearly, $a = b^k$.
- So, we follow case-02.
- Since $p = 0$, so we have-
- $T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$
- $T(n) = \theta (n^{\log_3 3} . \log^{0+1} n)$
- $T(n) = \theta (n^1 . \log^1 n)$
- Thus, **$T(n) = \theta (n \log n)$**

Examples

- $T(n) = T(n/2) + 1 \Rightarrow T(n) \in \Theta(log(n))$
  Here a = 1, b = 2, d = 0, $a = b^d$

- $T(n) = 4T(n/2) + n \Rightarrow T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$
  Here a = 4, b = 2, d = 1, $a > b^d$

- $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \Theta(n^2 \log n)$
  Here a = 4, b = 2, d = 2, $a = b^d$

- $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \Theta(n^3)$
  Here a = 4, b = 2, d = 3, $a < b^d$

The recursion-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

Used to “guess” a solution for the recurrence

Recall

- ***Introduction***
- ***Control Abstraction***
- ***Master Theorm***

Recursion Trees

- A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.
- A recursion tree is a tree where each node represents the cost of a certain recursive sub-problem.
- We sum up the values in each node to get the cost of the entire algorithm.

Steps to Solve Recurrence Relations Using Recursion Tree Method-

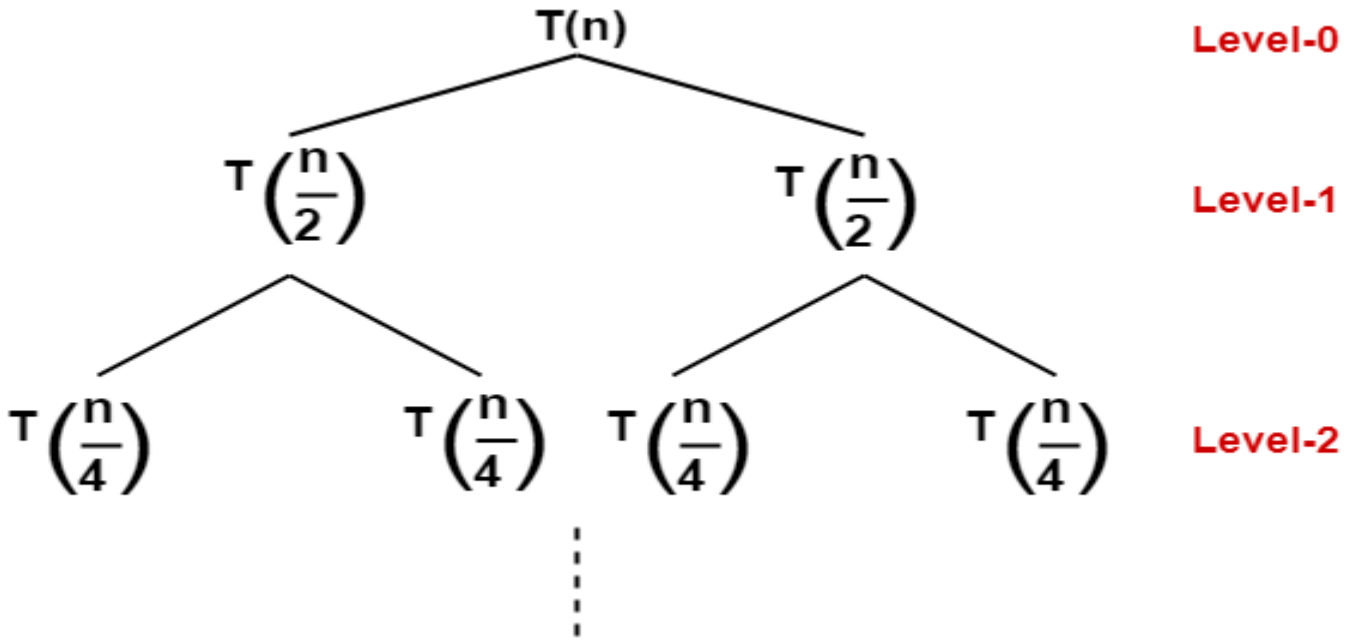
- **Step-01:**
  - Draw a recursion tree based on the given recurrence relation
- **Step-02:**
- Determine-
  - Cost of each level
  - Total number of levels in the recursion tree
  - Number of nodes in the last level
  - Cost of the last level
- **Step-03:**
  - Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

Problem-01:

Solve the following recurrence relation using recursion tree method- $T(n) = 2T(n/2) + n$

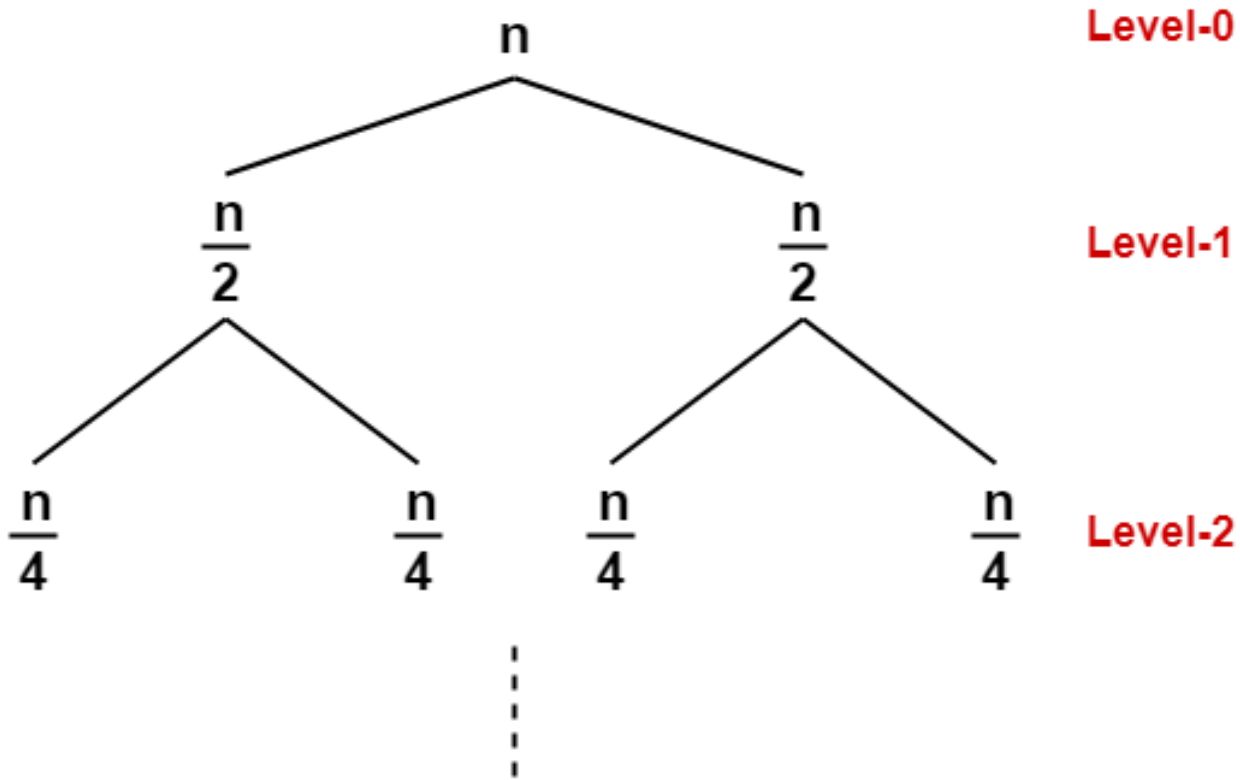
- **Step-01:**
  - Draw a recursion tree based on the given recurrence relation

  - The given recurrence relation shows-
  - A problem of size n will get divided into 2 sub-problems of size n/2.
  - Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
  - At the bottom most layer, the size of sub-problems will reduce to 1.



- The given recurrence relation shows-
- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/2 into its 2 sub-problems and then combining its solution is n/2 and so on.

- This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



- **Step-02:**
- Determine cost of each level-
  - Cost of level-0 = n
  - Cost of level-1 = n/2 + n/2 = n
  - Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n and so on.
- **Step-03:**
- Determine total number of levels in the recursion tree-
  - Size of sub-problem at level-0 = n/2 ^ 0
  - Size of sub-problem at level-1 = n/2 ^ 1
  - Size of sub-problem at level-2 = n/2 ^ 2
- Continuing in similar manner, we have-
  - Size of sub-problem at level-i = n/2 ^ i
  - Suppose at level-x (last level), size of sub-problem becomes 1. Then-
  - n / 2 ^ x = 1
  - 2^x = n
- Taking log on both sides, we get-
  - xlog2 = logn
  - x = log2n
- ∴ Total number of levels in the recursion tree = log2n + 1

- **Step-04:**
- Determine number of nodes in the last level-
  - Level-0 has 2^0 nodes i.e. 1 node
  - Level-1 has 2^1 nodes i.e. 2 nodes
  - Level-2 has 2^2 nodes i.e. 4 nodes
- Continuing in similar manner, we have-
  - Level-$\log_2 n$ has $2^{\log_2 n}$ nodes i.e. n nodes
- **Step-05:**
- Determine cost of last level-
  - Cost of last level = n x T(1) = θ(n)
- **Step-06:**
- Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

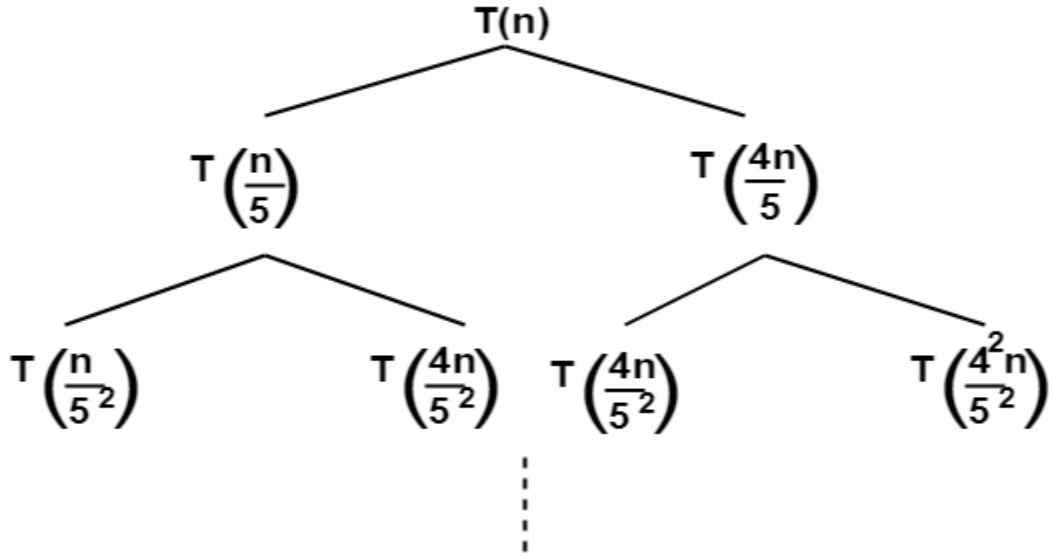
$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_2 n \text{ levels}} + \theta (n)$$

- = n x $\log_2 n$ + θ (n)
- = n$\log_2 n$ + θ (n)
- = **θ (n$\log_2 n$)**

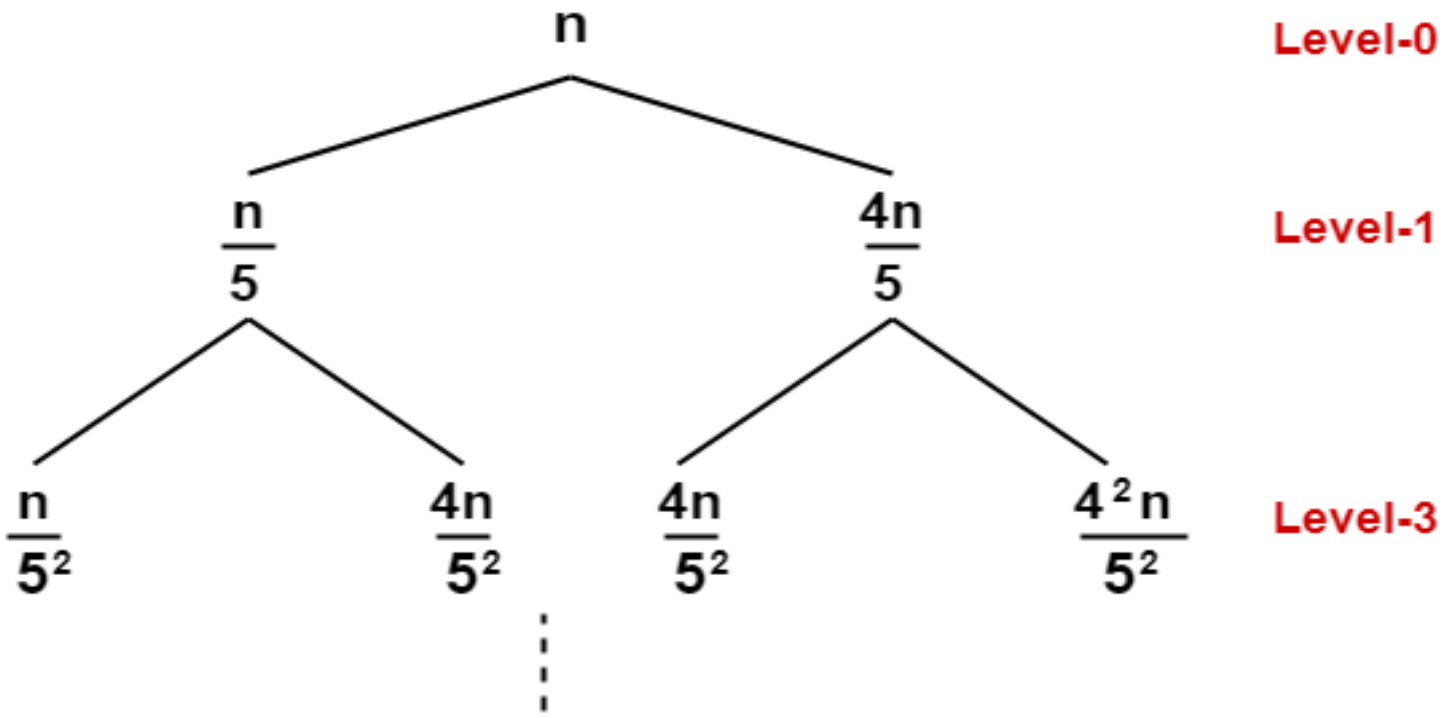
Problem-02:

Solve the following recurrence relation using recursion tree method- $T(n) = T(n/5) + T(4n/5) + n$

- **Step-01:**
  - Draw a recursion tree based on the given recurrence relation
  - The given recurrence relation shows-
  - A problem of size n will get divided into 2 sub-problems- one of size n/5 and another of size 4n/5.
  - Then, sub-problem of size n/5 will get divided into 2 sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
  - On the other side, sub-problem of size 4n/5 will get divided into 2 sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$ and so on.
  - At the bottom most layer, the size of sub-problems will reduce to 1.



- The given recurrence relation shows-
- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/5 into its 2 sub-problems and then combining its solution is n/5.
- The cost of dividing a problem of size 4n/5 into its 2 sub-problems and then combining its solution is 4n/5 and so on.
- This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



- **Step-02:**
- Determine cost of each level-
- Cost of level-0 = n
- Cost of level-1 = n/5 + 4n/5 = n
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$
- **Step-03:**
- Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-
- Size of sub-problem at level-0 = $(4/5)^0 n$
- Size of sub-problem at level-1 = $(4/5)^1 n$
- Size of sub-problem at level-2 = $(4/5)^2 n$
- Continuing in similar manner, we have-
- Size of sub-problem at level-i = $(4/5)^i n$
- Suppose at level-x (last level), size of sub-problem becomes 1. Then-
- $(4/5)^x n = 1$
- $(4/5)^x = 1/n$
- Taking log on both sides, we get-
- $x\log(4/5) = \log(1/n)$
- $x = \log_{5/4} n$
- ∴ Total number of levels in the recursion tree = $\log_{5/4} n + 1$

- **Step-04:**
- Determine number of nodes in the last level-
  - Level-0 has 2^0 nodes i.e. 1 node
  - Level-1 has 2^1 nodes i.e. 2 nodes
  - Level-2 has 2^2 nodes i.e. 4 nodes
- Continuing in similar manner, we have-
  - Level-$\log_{5/4}n$ has $2^{\log_{5/4}n}$ nodes
- **Step-05:**
- Determine cost of last level-
  - Cost of last level = $2^{\log_{5/4}n} \times T(1) = \theta(2^{\log_{5/4}n}) = \theta(n^{\log_{5/4}2})$
- **Step-06:**
- Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

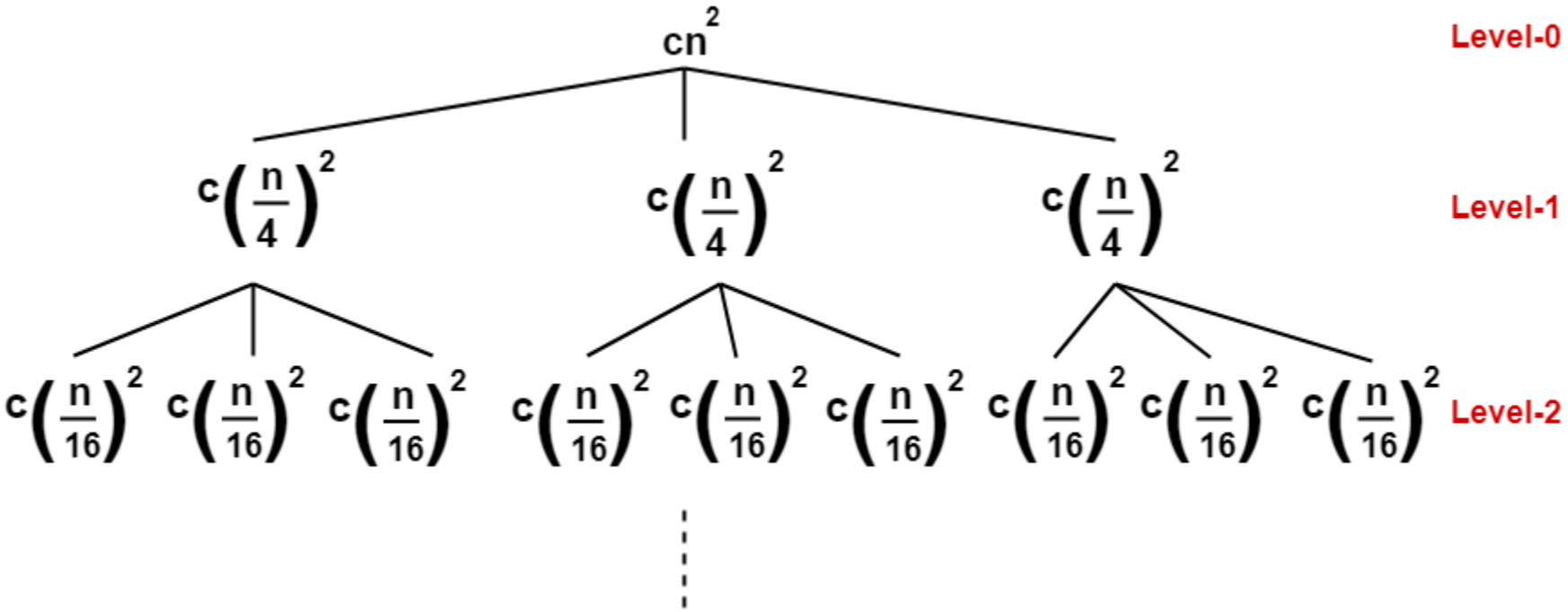
$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_{5/4}n \text{ levels}} + \theta(n^{\log_{5/4}2})$$

- $= n\log_{5/4}n + \theta(n^{\log_{5/4}2})$
- $= \theta(n\log_{5/4}n)$

Problem-02:

Solve the following recurrence relation using recursion tree method- $T(n) = 3T(n/4) + cn2$

- **Step-01:**
  - Draw a recursion tree based on the given recurrence relation



- **Step-02:**
- Determine cost of each level-
- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$
- **Step-03:**
- Determine total number of levels in the recursion tree-
- Size of sub-problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$
- Continuing in similar manner, we have- Size of sub-problem at level-i = $n/4^i$
- Suppose at level-x (last level), size of sub-problem becomes 1. Then-
- $n/4^x = 1$
- $4^x = n$
- Taking log on both sides, we get-
- $x\log 4 = \log n$
- $x = \log_4 n$
- $\therefore$ Total number of levels in the recursion tree = $\log_4 n + 1$

- **Step-04:**
- Determine number of nodes in the last level-
- Level-0 has 3^0 nodes i.e. 1 node
- Level-1 has 3^1 nodes i.e. 3 nodes
- Level-2 has 3^2 nodes i.e. 9 nodes Continuing in similar manner, we have-
- Level-$\log_4 n$ has $3^{\log_4 n}$ nodes i.e. $n^{\log_4 3}$ nodes
- **Step-05:**
- Determine cost of last level-
- Cost of last level = $n^{\log_4 3} \times T(1) = \theta(n^{\log_4 3})$
- **Step-06:**
- Add costs of all the levels of the recursion tree and simplify the expression so obtained

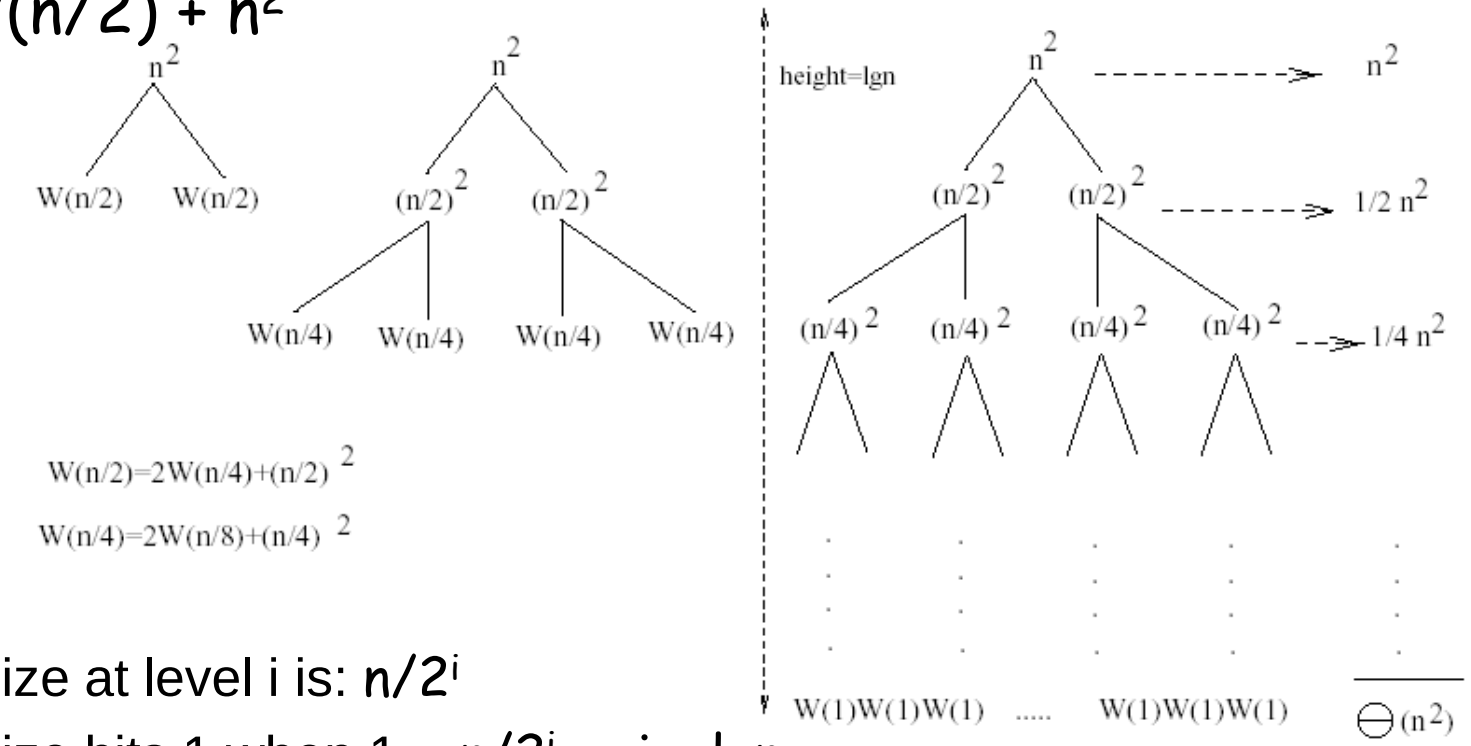
$$T(n) = \underbrace{\left\{ cn^2 + \frac{3}{16} cn^2 + \frac{9}{(16)^2} cn^2 + \ldots\ldots\ldots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta\left(n^{\log_4 3}\right)$$

- $= cn^2 \{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \} + \theta(n^{\log_4 3})$
- Now, $\{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \}$ forms an infinite Geometric progression
- On solving, we get-$= (16/13)cn^2 \{ 1 - (3/16)^{\log_4 n} \} + \theta(n^{\log_4 3})$
- $= (16/13)cn^2 - (16/13)cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$

$$\mathbf{O(n^2)}$$

Example 1

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



Subproblem size at level i is: $n/2^i$

Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$

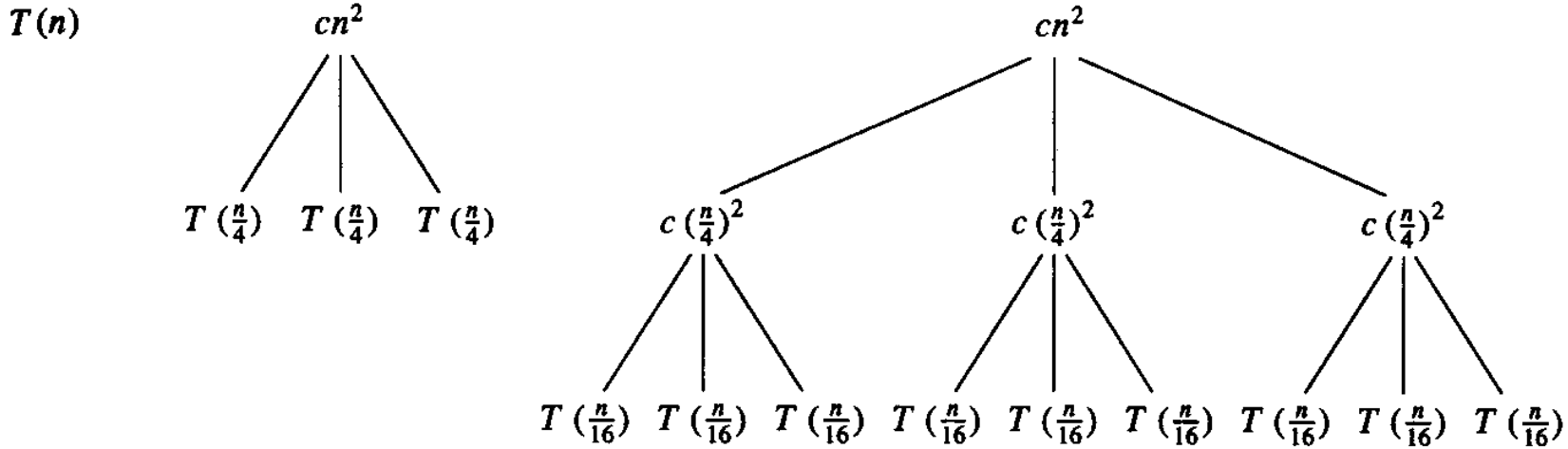
Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level i = 2^i

Total cost:

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

Example 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:

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

Example 2 - Substitution

$$T(n) = 3T(n/4) + cn^2$$

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \leq dn^2$, for some d and $n \geq n_0$
  - Induction hypothesis: $T(n/4) \leq d\,(n/4)^2$
- Proof of induction goal:

$$\begin{aligned} T(n) &= 3T(n/4) + cn^2 \\ &\leq 3d\,(n/4)^2 + cn^2 \\ &= (3/16)\,d\,n^2 + cn^2 \\ &\leq d\,n^2 \qquad \text{if: } d \geq (16/13)c \end{aligned}$$

- Therefore: $T(n) = O(n^2)$

Example 3 (simpler proof)

$$W(n) = W(n/3) + W(2n/3) + n$$

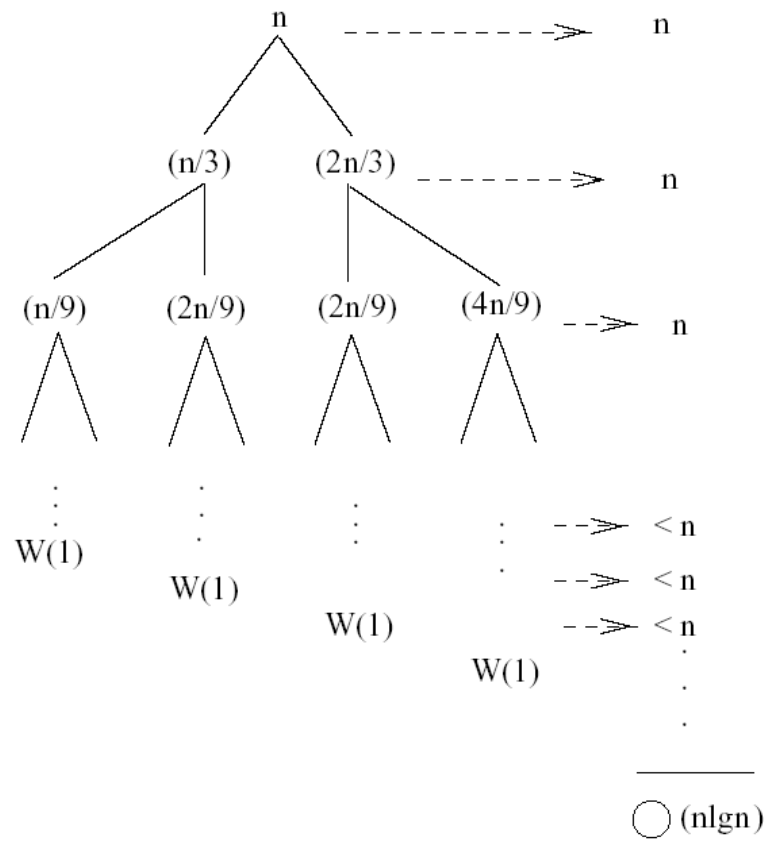
- The longest path from the root to a leaf is:

  $n \to (2/3)n \to (2/3)^2 n \to \ldots$

  $\to 1$

- Sub problem size hits 1 when $1 = (2/3)^i n \Leftrightarrow$ $i = \log_{3/2} n$
- Cost of the problem at level i = n
- Total cost:

$\Rightarrow W(n) = O(n \lg n)$



Example 3

$W(n) = W(n/3) + W(2n/3) + n$

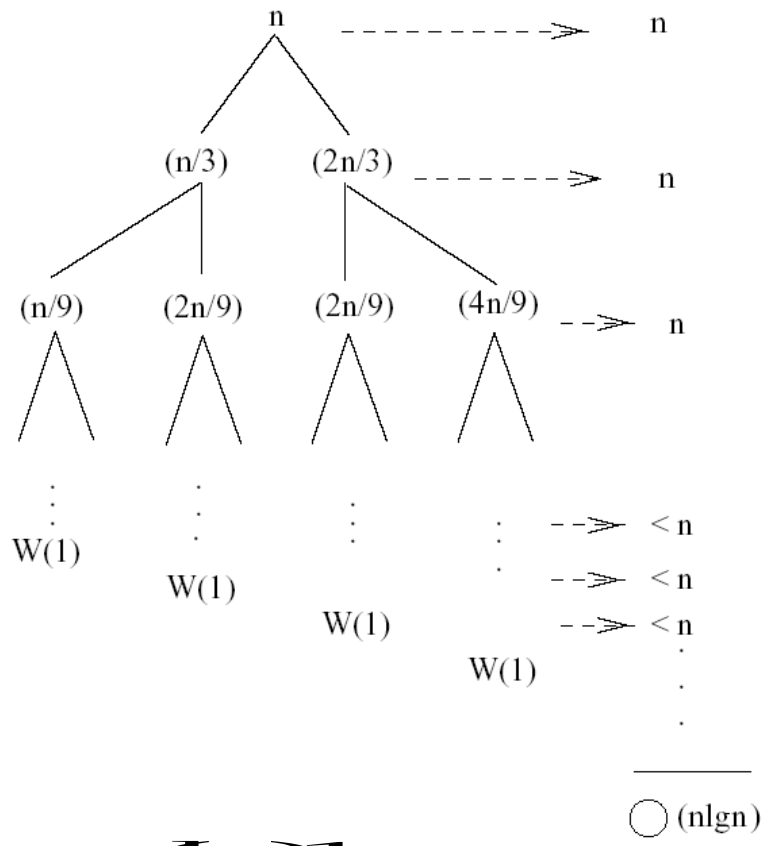
- The longest path from the root to a leaf is:

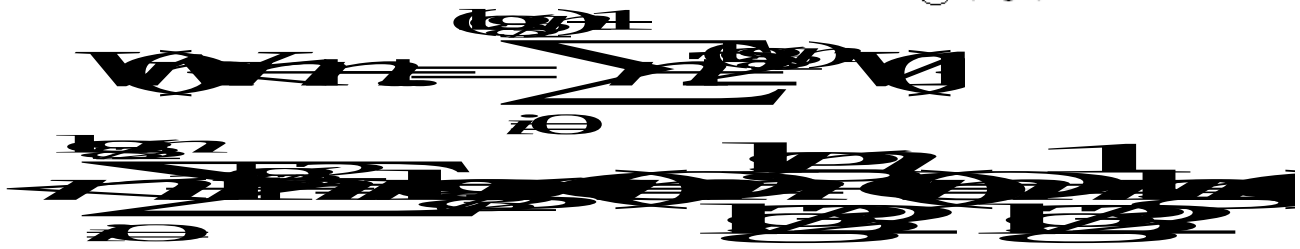
  $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$

- Subproblem size hits 1 when

  $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$

- Cost of the problem at level i = n
- Total cost:



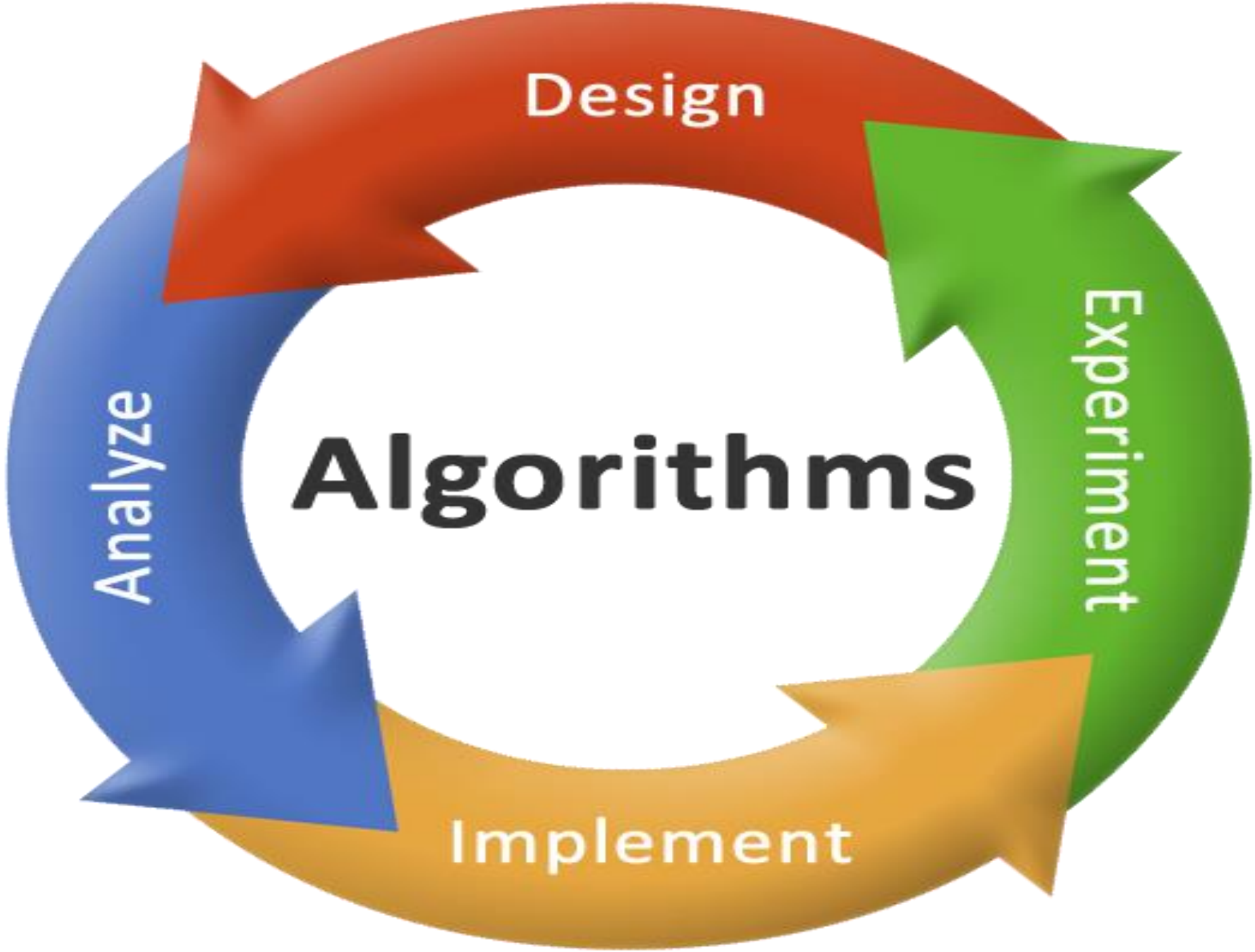


$\Rightarrow W(n) = O(n \lg n)$

Solve the following recurrence relation using Master's theorem- $T(n) = 2T(n/2) + n\log n$

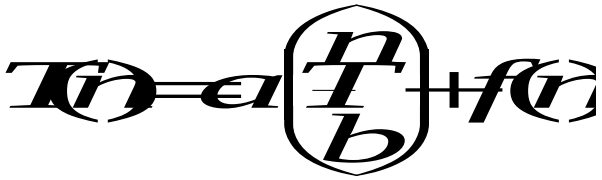
- **Solution-**
- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
- Then, we have- $a = 2 \quad b = 2 \quad k = 1 \quad p = 1$
- Now, $a = 2$ and $b^k = 2^1 = 2$.
- Clearly, $a = b^k$.
- So, we follow case-02.
- Since $p = 1$, so we have-
- $T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$
- $T(n) = \theta (n^{\log_2 2} . \log^{1+1} n)$
-
- Thus

$$\mathbf{T(n) = \theta (n \log^2 n)}$$


Design
Experiment
Implement
Analyze
Algorithms

Master's method

- "Cookbook" for solving recurrences of the form:



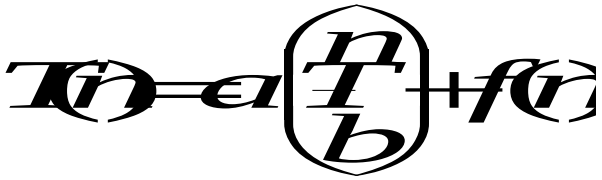
where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Idea: compare $f(n)$ with $n^{\log_b a}$

- $f(n)$ is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^{ε}
- $f(n)$ is asymptotically equal with $n^{\log_b a}$

Master's method

- "Cookbook" for solving recurrences of the form:



where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

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

regularity condition

Examples

$$T(n) = 2T(n/2) + n$$

$a = 2, b = 2, \log_2 2 = 1$

Compare $n^{\log_2 2}$ with $f(n) = n$

$\Rightarrow f(n) = \Theta(n) \Rightarrow$ Case 2

$\Rightarrow T(n) = \Theta(n \lg n)$

Examples

$$T(n) = 2T(n/2) + n^2$$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ Case 3 $\Rightarrow$ verify regularity cond.

$a\, f(n/b) \leq c\, f(n)$

$\Leftrightarrow 2\, n^2/4 \leq c\, n^2 \Rightarrow c = \frac{1}{2}$ is a solution ($c<1$)

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

Examples (cont.)

$$T(n) = 2T(n/2) + \sqrt{n}$$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^{1/2}$

$\Rightarrow f(n) = O(n^{1-\varepsilon})$ Case 1

$\Rightarrow T(n) = \Theta(n)$

Examples

$$T(n) = 3T(n/4) + n\lg n$$

$a = 3$, $b = 4$, $\log_4 3 = 0.793$

Compare $n^{0.793}$ with $f(n) = n\lg n$

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$ Case 3

Check regularity condition:

$$3*(n/4)\lg(n/4) \le (3/4)n\lg n = c*f(n),\ c=3/4$$

$\Rightarrow T(n) = \Theta(n\lg n)$

Examples

$$T(n) = 2T(n/2) + n\lg n$$

$a = 2, b = 2, \log_2 2 = 1$

- Compare n with $f(n) = n\lg n$
  - seems like case 3 should apply
- f(n) must be polynomially larger by a factor of n^{ε}
- In this case it is only larger by a factor of $\lg n$

SOLVING RECURSION USING POLYNOMIAL REDUCTION METHOD

ANALYSIS OF NON RECURSIVE ALGORITHM

- find different i/p parameters and their sizes.
- find basic operations of an algorithm generally present in inner most loop.
- efficiency should be checked separately for best, average & worst case, if basic operation is based on condition.
- find out how many times an algorithm executes its basic operations.
- use the standard formulae to represent the efficiency.

ANALYSIS OF RECURSIVE ALGORITHM

- find different i/p parameters and their sizes
- find basic operation of an algorithm.
- check that the basic operations varies with i/p size , if yes find best, average and worst cases separately
- Use recurrence equation with proper initial conditions to represent how many times the algorithm executes its basic operations.
- Solve the recurrence or establish the order or growth to get the solution.

A) HOMOGENEOUS RECURRENCES:

- A recurrences of the form a0tn+a1tn-1 +a2tn-2+….+aktn-k=0
- Where a and k are constants and is called a homogeneous linear recurrence eq. with const. coefficient. It is called a linear homogeneous because the linear combination of the term =0.
- **The homogeneous linear recurrences with const. coefficients can be solved by the following steps:**

1) first rewrite the recurrence in the following form :

a0tn +a1tn-1+…..aktn-k=0

2)then form the characteristic polynomial of degree k using const. as: a0x^k+a1x^k-1+…+ak

3)find the k roots of the characteristic poly. a) if k roots are distinct then general solution is of the form :tn=c1r1^n+c2r2^n+….+cnrk^n b) if k roots are not distinct ,then suppose root r1 has multiplicity m1 and r2 has multiplicity m2 then m1+m2=k the general solution is of the following form: tn=c11r1^n+c12nr1^n+c13n^2r1^n+….+ c1m1n^m1-1r1^n + c21r2^n+c22nr2^n + c23n^2r2^n+….+c2m2n^m2-1r2^n 4)now we can solve k eq. using k initial conditions to find values of const. ci. 5) after putting the const. values in the eq. of tn and solving it, we obtain the solution for the recurrence.

CHARACTERISTIC (POLYNOMIAL) EQUATION
B)NON-HOMOGENEOUS RECURRENCES:

The linear combination of the terms are not equal to zero.

- **Steps: case 1)**
- Consider the following recurrences which does not equates to zero. a0tn+a1tn-1+ …+aktn-k=b^np(n) here b is const. and p(n) is a polynomial in n of degree d. For such recurrences the characteristic polynomial is of the following form:
- (a0x^k +a1x^k-1 +…ak)(x-b)^d+1

  After getting this characteristic polynomial,remaining steps are same as that for solving the homogeneous recurrences ,because the initial conditions are not specified,they are obtained from the recurrences.

- **Case 2)** consider the recurrences of the following form:
- a0tn +a1tn-1+…+ aktn-k=b1^np1(n)+b2^np2(n)+… here b1 and b2 are const. and p1(n), p2(n) are polynomials in n of degree d1 and d2 resp. For such recurrences the characteristic polynomial is of the following form: (a0x^k +a1x^k-1 + ak)(x-b1)^d1+1(x-b2)^d2+1
- After getting this polynomial ,remaining steps are same as before.

Guess and Verify

- *Guess the form of the solution .*
- *Use mathematical induction to find the constants and show that the solution works .*

Guess and Verify

- Guess a solution
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation
    - $T(n) \leq d\ g(n)$, for some $d > 0$ and $n \geq n_0$
  - Induction hypothesis: $T(k) \leq d\ g(k)$ for all $k < n$ (strong induction)
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants d and n_0 for which the **induction goal** holds

Example: Binary Search

$$T(n) = c + T(n/2)$$

- Guess: $T(n) = O(\lg n)$
  - Induction goal: $T(n) \le d \lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le d \lg(n/2)$
- Proof of induction goal:

$$T(n) = T(n/2) + c \le d \lg(n/2) + c$$
$$= d \lg n - d + c \le d \lg n$$
$$\text{if: } -d + c \le 0,\ d \ge c$$

- Base case?

Example 2

$$T(n) = T(n-1) + n$$

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \leq c\, n^2$, for some c and $n \geq n_0$
  - Induction hypothesis: $T(n-1) \leq c(n-1)^2$ for all $k < n$
- Proof of induction goal:

$$T(n) = T(n-1) + n \leq c\,(n-1)^2 + n$$

$$= cn^2 - (2cn - c - n) \leq cn^2$$

$$\text{if: } 2cn - c - n \geq 0 \Leftrightarrow c \geq n/(2n-1) \Leftrightarrow c \geq 1/(2 - 1/n)$$

  - For $n \geq 1 \Rightarrow 2 - 1/n \geq 1 \Rightarrow$ any $c \geq 1$ will work

Example 3

$$T(n) = 2T(n/2) + n$$

- Guess: $T(n) = O(n \lg n)$
  - Induction goal: $T(n) \le cn \lg n$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le cn/2 \lg(n/2)$
- Proof of induction goal:

$$T(n) = 2T(n/2) + n \le 2c\,(n/2)\lg(n/2) + n$$

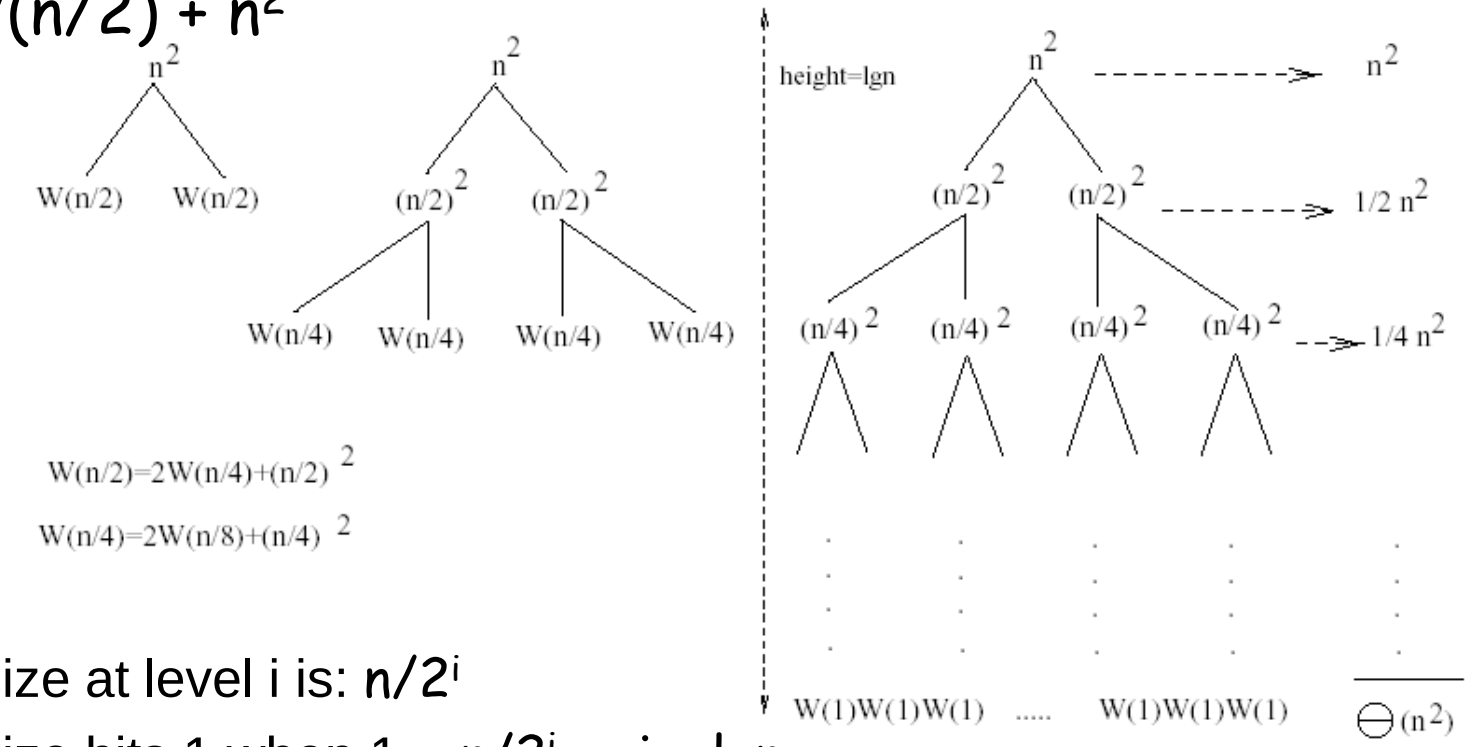
$$= cn \lg n - cn + n \le cn \lg n$$

$$\text{if: } -cn + n \le 0 \Rightarrow c \ge 1$$

- Base case?

Example 1

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



Subproblem size at level i is: $n/2^i$

Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$

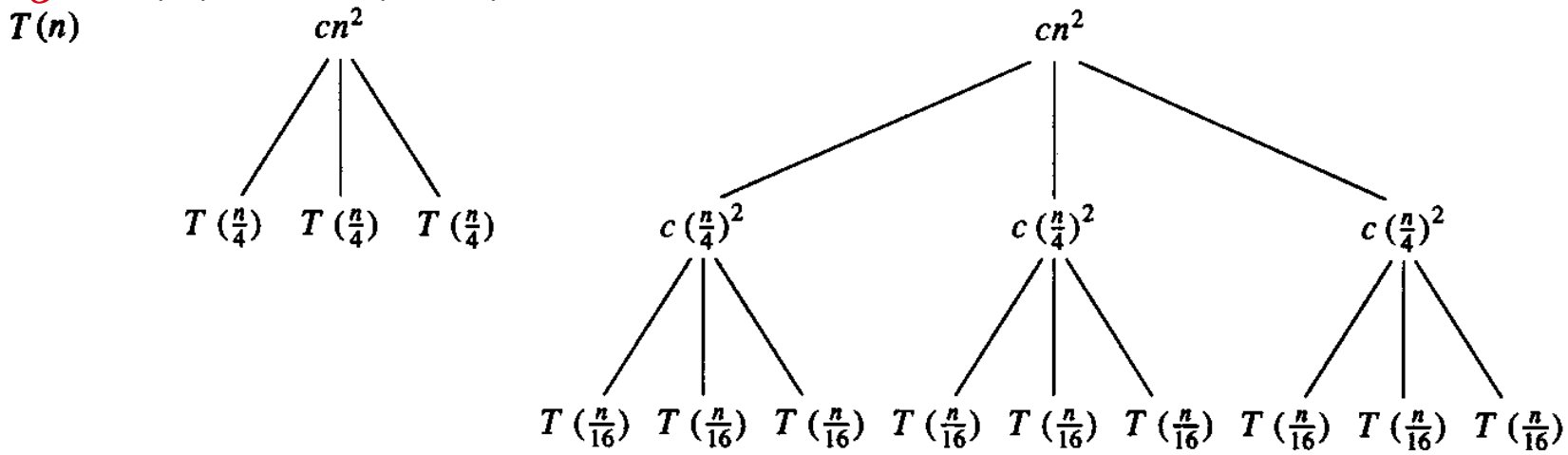
Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level $i = 2^i$

Total cost:

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

Example 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:

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

Example 2 –Proof

$$T(n) = 3T(n/4) + cn^2$$

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \leq dn^2$, for some d and $n \geq n_0$
  - Induction hypothesis: $T(n/4) \leq d\,(n/4)^2$
- Proof of induction goal:

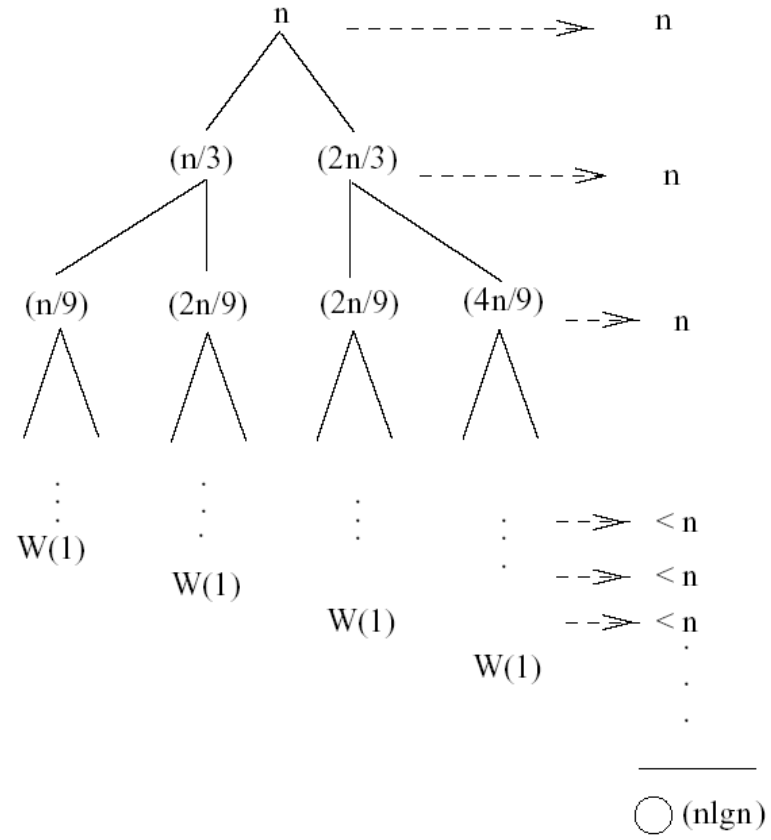
$$\begin{aligned} T(n) &= 3T(n/4) + cn^2 \\ &\leq 3d\,(n/4)^2 + cn^2 \\ &= (3/16)\,d\,n^2 + cn^2 \\ &\leq d\,n^2 \qquad \text{if: } d \geq (16/13)c \end{aligned}$$

- Therefore: $T(n) = O(n^2)$

Example 3

$W(n) = W(n/3) + W(2n/3) + n$

- The longest path from the root to a leaf is: $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \ldots \rightarrow 1$
- Subproblem size hits 1 when $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$
- Cost of the problem at level i = n
- Total cost:



$W(n) = O(n \lg n)$

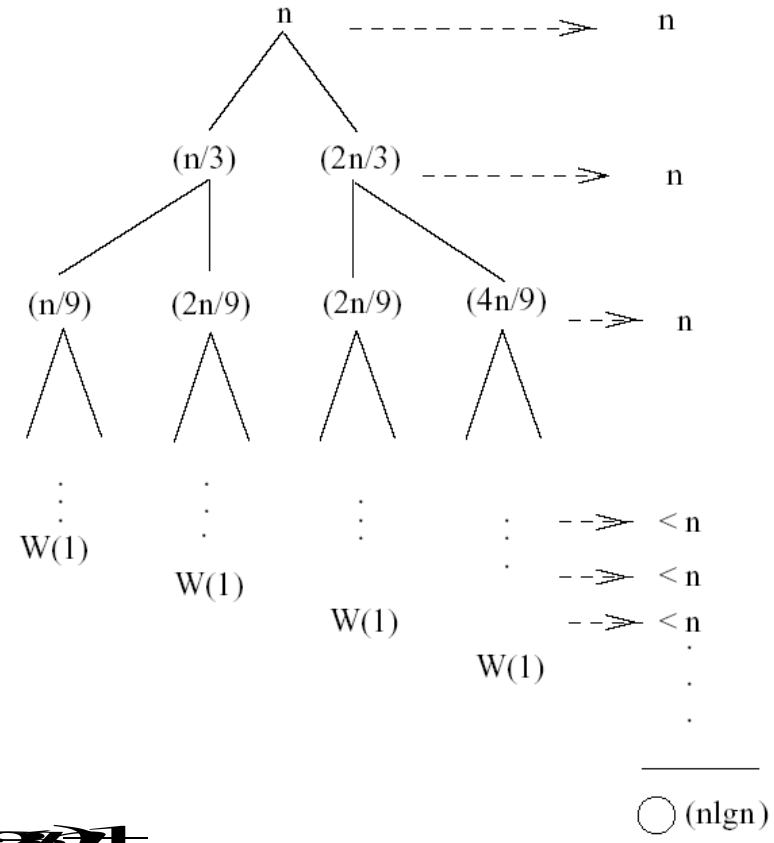
Example 3

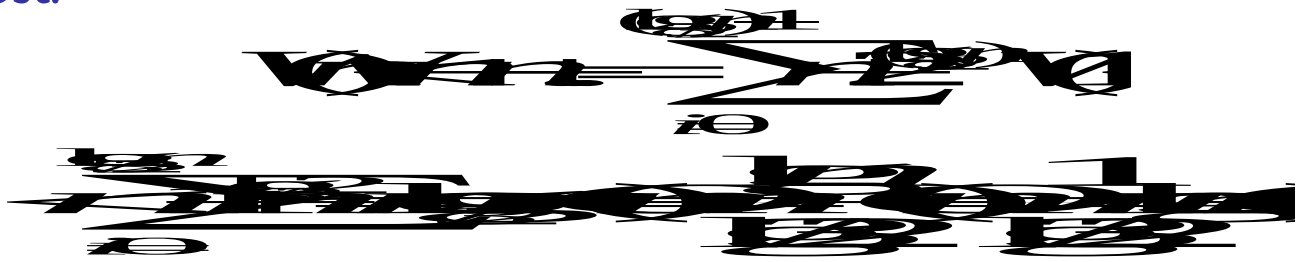
$$W(n) = W(n/3) + W(2n/3) + n$$

- The longest path from the root to a leaf is:

  $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$

- Subproblem size hits 1 when $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$
- Cost of the problem at level i = n
- Total cost:





$W(n) = O(n \lg n)$

Example 3 -Proof

$$W(n) = W(n/3) + W(2n/3) + O(n)$$

- Guess: $W(n) = O(n \lg n)$
  - Induction goal: $W(n) \leq dn \lg n$, for some d and $n \geq n_0$
  - Induction hypothesis: $W(k) \leq d\, k \lg k$ for any $K < n$ $(n/3, 2n/3)$
- Proof of induction goal:

  Try it out as an exercise!!
- $T(n) = O(n \lg n)$

Master's method

- "Cookbook" for solving recurrences of the form:

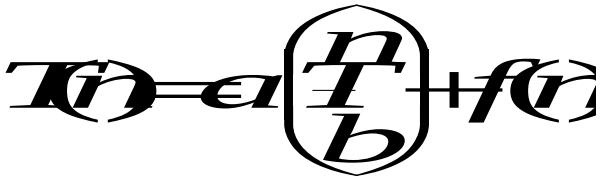
where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Idea: compare $f(n)$ with $n^{\log_b a}$

- $f(n)$ is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^{ε}
- $f(n)$ is asymptotically equal with $n^{\log_b a}$

Master's method

- "Cookbook" for solving recurrences of the form:



where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

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

regularity condition

Examples

$$T(n) = 2T(n/2) + n$$

$a = 2, b = 2, \log_2 2 = 1$

Compare $n^{\log_2 2}$ with $f(n) = n$

$\Rightarrow f(n) = \Theta(n) \Rightarrow$ Case 2

$\Rightarrow T(n) = \Theta(n \lg n)$

Examples

$$T(n) = 2T(n/2) + n^2$$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ Case 3 $\Rightarrow$ verify regularity cond.

$a\, f(n/b) \leq c\, f(n)$

$\Leftrightarrow 2\, n^2/4 \leq c\, n^2 \Rightarrow c = \frac{1}{2}$ is a solution ($c<1$)

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

Examples (cont.)

$$T(n) = 2T(n/2) + \sqrt{n}$$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^{1/2}$

$\Rightarrow f(n) = O(n^{1-\varepsilon})$ Case 1

$\Rightarrow T(n) = \Theta(n)$

Examples

$$T(n) = 3T(n/4) + n\lg n$$

$a = 3$, $b = 4$, $\log_4 3 = 0.793$

Compare $n^{0.793}$ with $f(n) = n\lg n$

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$ Case 3

Check regularity condition:

$$3*(n/4)\lg(n/4) \le (3/4)n\lg n = c*f(n),\ c=3/4$$

$\Rightarrow T(n) = \Theta(n\lg n)$

Examples

$$T(n) = 2T(n/2) + n\lg n$$

$a = 2, b = 2, \log_2 2 = 1$

- Compare n with $f(n) = n\lg n$
  - seems like case 3 should apply
- $f(n)$ must be polynomially larger by a factor of n^{ε}
- In this case it is only larger by a factor of $\lg n$