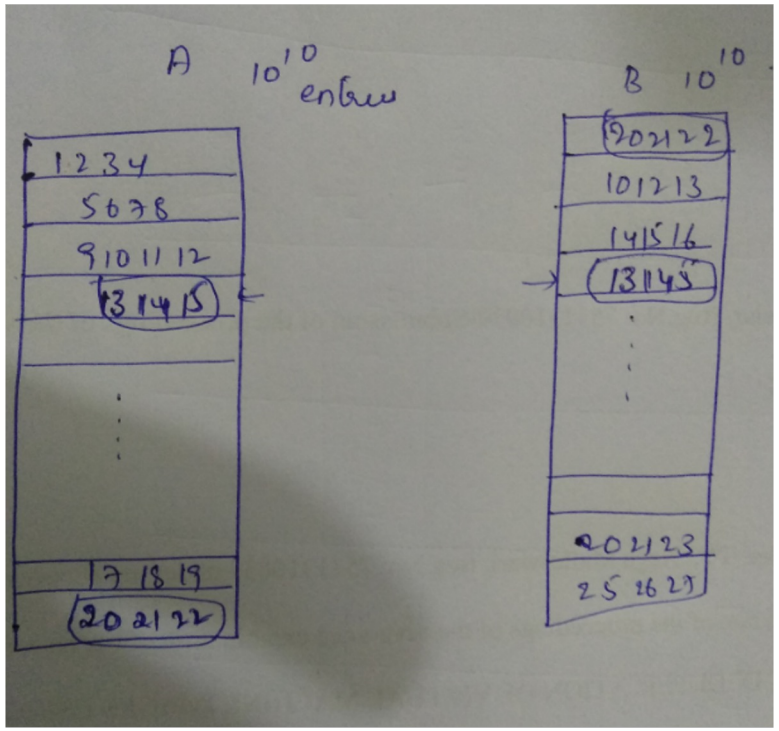
UNIT-I

Introduction-Algorithm Design

- **Session Learning Outcome-SLO**-To understand why algorithm design is important.
- **Example 1 :**

  Take 2 arrays A and B of 10^{10} entries each. Now write an algorithm to find common entries.



Algorithm:

Assume 2 arrays
k=1
for i=1 ton
    for j=1 to n
        if A[i]=B[j] then C[k] = A[i]; k=k+1
Output the array C

- Steps taken by the above algorithm is n^2 that is $(10^{10})^2 = 10^{20}$
- How many seconds 10^{20} steps take?
- 10^{10} Seconds = 317 years
- On a modern powerful computer it takes 3 years.

Motivation:

- Whether any way to write an algorithm for the above problem which takes less time than previously mentioned (Yes/No)

  Yes

- Now take example 1, finding common entries . Assume array B is arranged in increasing order ? Can I get a better algorithm. (yes / no)

  Yes

- If the example 2 concept (binary search) is used here. Then algorithm will be

  for i=1 to n

  look for A[i] in B by doing binary search and

  if found store in C

  Output the array C

Analysis:

- How many steps it takes. $n(\log_2 n)$
- $n = 10^{10}$
- $n(\log_2 n) = 10^{10} (\log_2 10^{10})$
- 10^{10}steps in 1 second then it for the above steps, it takes $10^{10} (\log_2 10^{10}) / 10^{10} <= 40$ secs

Example 2

A book contains page number from 1 and 1024, and I have to search 73 page

Maximum number of comparisons is

512, 256, 128, 64, 32, 16, 8, 4, 2, 1.

If Binary search is applied

For the range 1-1024 –10 comparisons

For the range 1-2048 – 11 comparisons

For the range 1-4096- 12 comparisons

For the range 1-N--- ? Comparisons

What is the relation

- It is log N comparisons ($1024 = 2^{10}$), $2048 = 2^{11}$ $4096 = 2^{12}$ …….
- $\log_2 N$ is the number of powers of 2 in N
- $\log_2 N$ is a slow growing function

Summary:

- More than the power of computers, an improvement in algorithm can make a difference between 317 years and 40 seconds.
- How much time to take for sorting?
- Can I choose $O(n^2)$?
- Quick sort, Mergesort…

Fundamentals of Algorithms

Session Learning Outcome-SLO:

- Able to know what is an algorithm and how to write algorithm / pseudocode.

What is an algorithm?

It is a finite set of instructions or a sequence of computational steps that if followed it accomplishes a task.

Characteristics of an algorithm:

- **Input :** 0 or more input
- **Output:** 1 or more output
- **Definiteness**: clear and unambiguous
- **Finiteness:** termination after finite number of steps
- **Effectiveness:** solved by pen and paper

Describing the Algorithm

The skills required to effectively design and analyze algorithms are entangled with the skills required to effectively describe algorithms. A complete description of any algorithm has four components:

- **What**: A precise specification of the problem that the algorithm solves.
- **How**: A precise description of the algorithm itself.
- **Why:** A proof that the algorithm solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm.

It is not necessary (or even advisable) to develop these four components in this particular order. Problem specifications, algorithm descriptions, correctness proofs, and time analyses usually evolve simultaneously, with the development of each component informing the development of the others.

Specification/ Design of Algorithm

- The algorithm can be specified either using:
  - Natural language
  - Pseudo code
  - Flow chart

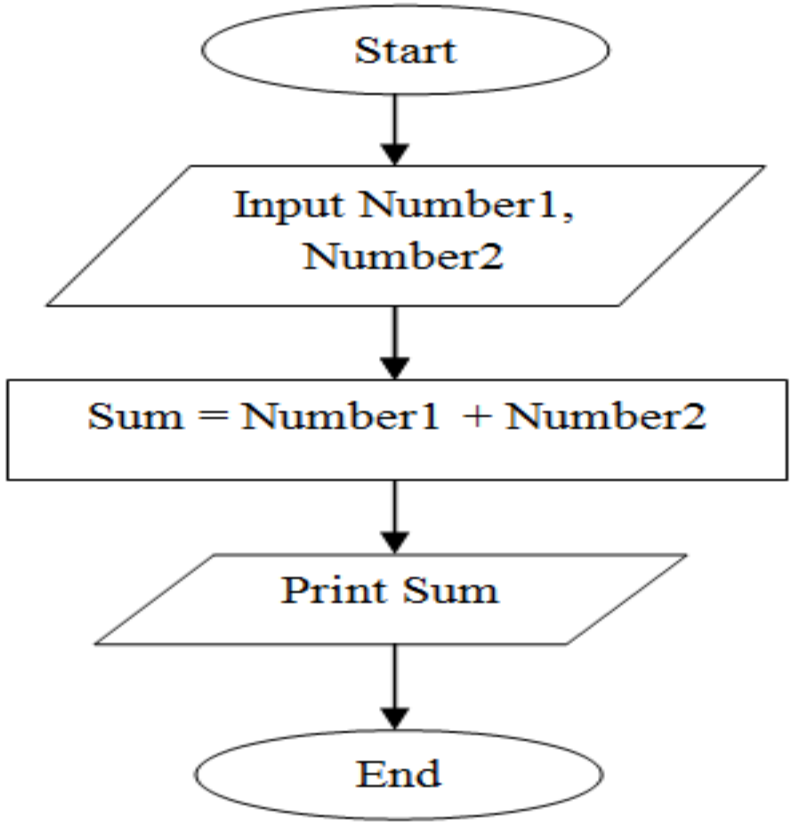
Natural Language

- Step 1: Start
- Step 2: Declare variables num1, num2 and sum.
- Step 3: Read values num1 and num2.
- Step 4: Add num1 and num2 and assign the result to sum. sum←num1+num2
- Step 5: Display sum
- Step 6: Stop

Pseudo code

- BEGIN.
- **NUMBER** s1, s2, **sum**.
- OUTPUT("Input number1:")
- INPUT s1.
- OUTPUT("Input number2:")
- INPUT s2.
- **sum**=s1+s2.
- OUTPUT **sum**.

Flowchart


Start
Input Number1, Number2
Sum = Number1 + Number2
Print Sum
End

Specifying the Problem

- To describe an algorithm, the problem is described that the algorithm is supposed to solve.
- Algorithmic problems are often presented using **standard English**, in terms of real-world objects. The algorithm designers, restate these problems in terms of formal, abstract, mathematical objects—numbers, arrays, lists, graphs, trees, and so on
- If the problem statement carries any hidden assumptions, state those assumptions explicitly;
- Specification to be refined as we develop the algorithm. For example, an algorithm may require a particular input representation, or produce a particular output representation, that was left unspecified in the original informal problem description.

- The specification should include just enough detail that someone else could use our algorithm as a black box, without knowing how or why the algorithm actually works.
- In particular, the type and meaning of each input parameter, and exactly how the eventual output depends on the input parameters are described. On the other hand, specification should deliberately hide any details that are not necessary to use the algorithm as a black box.
- **Example:** Given two non-negative integers x and y, each represented as an array of digits, compute the product $x \cdot y$, also represented as an array of digits. To someone using these algorithms, the choice of algorithm is completely irrelevant.

Describing Algorithms

- The clearest way to present an algorithm is using a combination of pseudocode and structured English.
- Pseudocode uses the structure of formal programming languages and mathematics to break algorithms into primitive steps; the primitive steps themselves can be written using mathematical notation, pure English, or an appropriate mixture of the two, whatever is clearest.
- Well written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm easier to understand, analyze, debug, and implement.

Pseudocode conventions

1. **Comments** //
2. **Blocks** {and }.
3. An **identifier** begins with a letter
4. ; (dot), -> operator

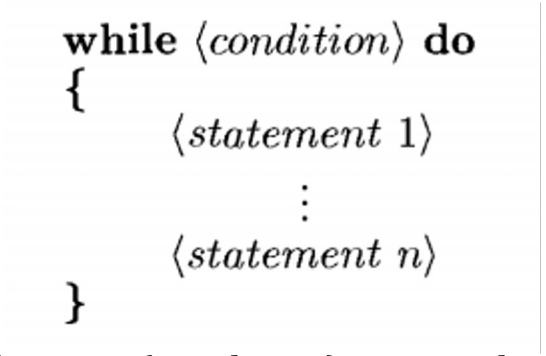
```
node = record
       {   datatype_1   data_1;
                ⋮
           datatype_n   data_n;
           node         *link;
       }
```

1. **Assignment**--- $\langle variable \rangle := \langle expression \rangle;$
2. Two **Boolean values *true*** and ***false***, logical operators and, or, and not, relational operators <, ≤, ≥,and >

7. Elements of **multi-dimensional arrays** are accessed using[and]. For example, if A is a two dimensional array, the (i,j)th element of the array is denoted as - A[i, j]. Array indices start at zero.

8. **Looping statements- for, while and repeat-until**

- while loop written as



As long as (condition)is true, the statements get executed. When (condition)becomes false, the loop is exited. The value of (condition) is evaluated at the top of the loop.

- The general form of a for loop is

```
for variable := value1 to value2  step step do
{
        ⟨statement 1⟩
                ⋮
        ⟨statement n⟩
}
```

Here valuel, value2, and *step* are arithmetic expression The clause "step *step*" is optional and taken as +1 if it does not occur. *Step* could either be positive or negative

- A repeat-until statement is constructed as follows:

```
repeat
        ⟨statement 1⟩
                ⋮
        ⟨statement n⟩
until ⟨condition⟩
```

9. **break –**
10. **return**
11. **Conditional statement has the following forms**

if ⟨*condition*⟩ **then** ⟨*statement*⟩
if ⟨*condition*⟩ **then** ⟨*statement 1*⟩ **else** ⟨*statement 2*⟩

9. **Multiple-decision statement as:**

case
{
:⟨*condition 1*⟩: ⟨*statement 1*⟩
⋮
:⟨*condition n*⟩: ⟨*statement n*⟩
:else: ⟨*statement n + 1*⟩
}

9. **Input, output :** read, write

14. Procedure: *Algorithm*. An algorithm consists of a heading and a body.

Algorithm *Name* (⟨*parameter list*⟩)

Where *Name* is procedure name and (<parameter list>) is a listing of the procedure parameters. The body has one or more (simple or compound)statements enclosed within braces {and } .

Example of pseudocode:

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

In this algorithm (named Max), A and n are procedure parameters. *Result* and i are local variables.

Analysing Algorithms:

We have to prove that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

- In some application settings, it is acceptable for programs to behave correctly most of the time, on all "reasonable" inputs.
- But we require algorithms that are always correct, for all possible inputs.
- We must prove that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough.
- In particular, correctness proofs usually involve induction.

Analyzing Algorithms:

Running Time

- The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular problem.

Summary:

- Every problem is to be specified, for which an algorithm is described in the form of pseudocode and is analysed to test its efficiency.

Home assignment:

1. Write the pseudocode for finding matrix multiplication of 2 two-dimensional arrays.
2. Write the pseudocode for linear search.

Correctness of Algorithm

Session Learning Outcome-SLO:

- Able to prove the correctness of an algorithm by any of the methods.

A proof of correctness requires that the solution be stated in two forms.

- One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus.
- The second form is called a specification, and this may also be expressed in the predicate calculus.

- A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct.
- A proof of correctness is much more valuable than a thousand tests(if that proof is correct),since it guarantees that the program will work correctly for all possible inputs.

Methods of proving correctness

How to prove that an algorithm is correct?

Proof by:

- Counterexample (indirect proof)
- Induction (direct proof)
- Loop Invariant

- **Other approaches:**
  - proof by cases/enumeration
  - proof by chain of iffs
  - proof by contradiction
  - proof by contrapositive

- For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem. For sorting, this means even if the input is already sorted or it contains repeated elements.

Assertions

- To prove correctness we associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm.
  - E.g., *A*[1], …, *A*[k] form an increasing sequence
- **Preconditions** – assertions that must be valid *before* the execution of an algorithm or a subroutine
- **Postconditions** – assertions that must be valid *after* the execution of an algorithm or a subroutine

Loop Invariants

- **Invariants** – assertions that are valid any time they are reached (many times during the execution of an algorithm, e.g., in loops)

- We must show three things about loop invariants:

  - **Initialization** – it is true prior to the first iteration
  - **Maintenance** – if it is true before an iteration, it remains true before the next iteration
  - **Termination** – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

Proof by mathematical induction

Mathematical induction (MI) is an essential tool for proving the statement that proves an algorithm's correctness. The general idea of MI is to prove that a statement is true for every natural number n.

It contains 3 steps:

1. **Induction Hypothesis**: Define the rule we want to prove for every n, let's call the rule f(n).

2. **Induction Base**: Proving the rule is valid for an initial value, or rather a starting point - this is often proven by solving the Induction Hypothesis f(n) for n=1 or whatever initial value is appropriate

3. **Induction Step**: Proving that if we know that f(n) is true, we can step one step forward and assume f(n+1) is correct

Example

If we define S(n) as the sum of the first n natural numbers, for example S(3) = 3+2+1, prove that the following formula can be applied to any n:

$$S(n) = \frac{(n+1) * n}{2}$$

Let's trace our steps:

1. **Induction Hypothesis**: S(n) defined with the formula above

1. **Induction Base**: $S(1) = \frac{(1+1) * 1}{2} = \frac{2}{2} = 1$ (1) = 1

3. **Induction Step**: In this step we need to prove that if the formula applies to S(n), it also applies to S(n+1) as follows:

$$S(n+1) = \frac{(n+1+1)*(n+1)}{2} = \frac{(n+2)*(n+1)}{2}$$

This is known as an **implication** (a=>b), which just means that we have to prove b is correct prov

$$S(n+1) = S(n) + (n+1) = \frac{(n+1)*n}{2} + (n+1) = \frac{n^2+n+2n+2}{2}$$

$$\boxed{= \frac{n^2+3n+2}{2} = \frac{(n+2)*(n+1)}{2}}$$

Note that S(n+1) = S(n) + (n+1) just means we are recursively calculating the sum.

Example with literals:

S(3) = S(2) + 3= S(1) + 2 + 3 = 1 + 2 + 3 = 6 Hence proved

Summary:

For every algorithm proof of correctness is very important to show that the program is correct in all cases.

Home Assignment

Prove by induction:

(a) $\sum_{i=1}^{n} i = n(n+1)/2,\ n \geq 1$

(b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6,\ n \geq 1$

(c) $\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x-1),\ x \neq 1,\ n \geq 0$

Direct method of Proof

Suppose, the hypothesis P is true. Then the implication P $\rightarrow$ Q can be proved if we can prove that Q is true by using the rules of inference and some other theorems.

A counter-example is an example that shows that a statement is not always true. It is sufficient to just give one example.

Show by counter-example that the following statements are not always true:

1) $4n+4$ is always a multiple of 8 for all positive integer values of n.

2) If $x>y$ then $\frac{x}{y}>1$

3) $p-q \leq p^2-q^2$ for all values of p and q.

4) $2n^2-16n+31$ is always positive for all values of n.

① when $n=2$, $4n+4=12$

not a multiple of 8

② If $x=6$, $y=-3$, ie. $6>-3$

then $\frac{x}{y}=\frac{6}{-3}=-2$

③ when $p=4$, $q=-5$

$p-q=4--5$ | $p^2-q^2=(4)^2-(-5)^2$

$=9$ | $=-9$

So $p-q$ is not less than p^2-q^2 for all values of p and q.

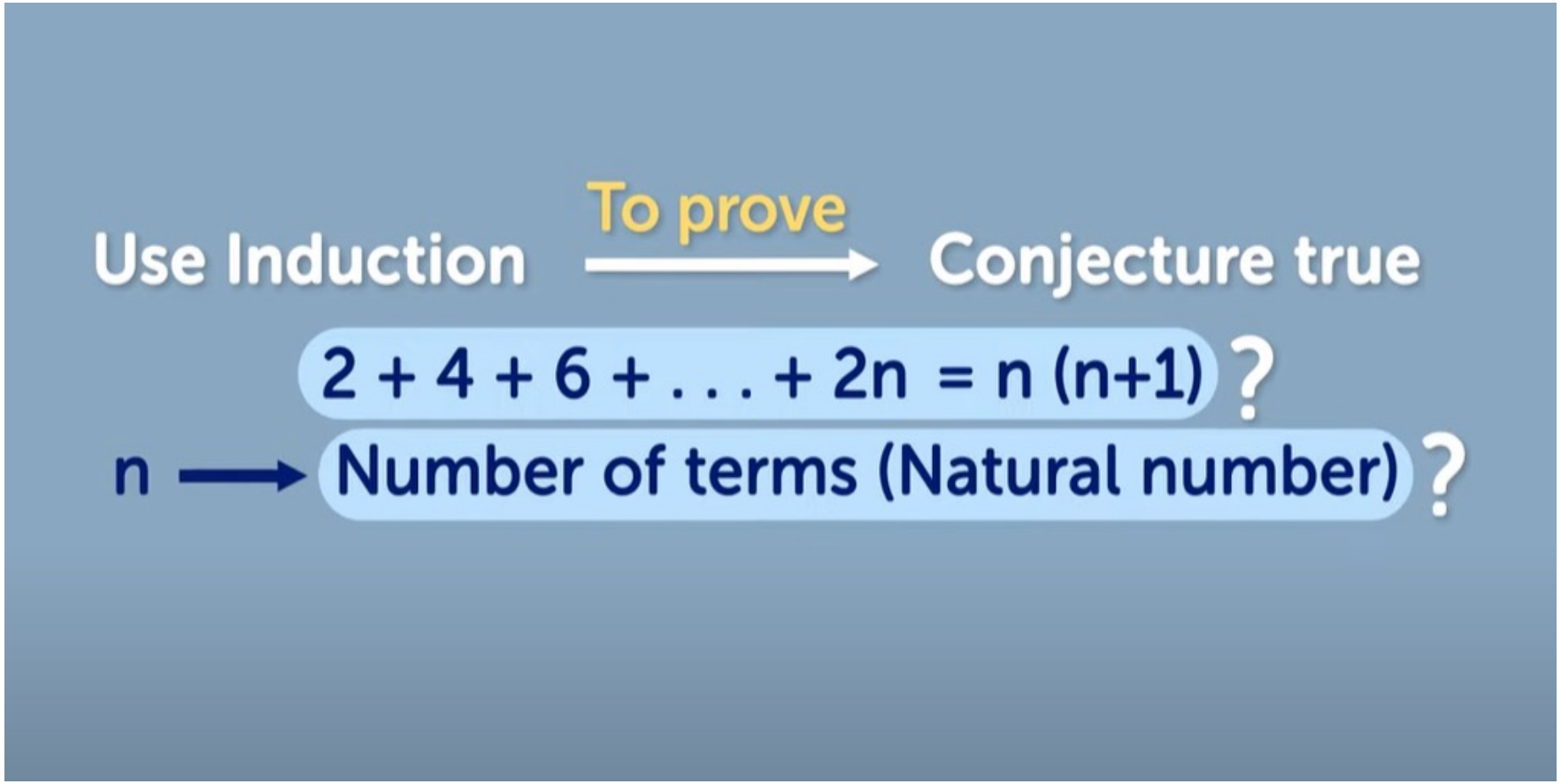
④ when $n=4$

$2n^2-16n+31=2(4)^2-16(4)+31$

$=32-64+31$

$=-1$

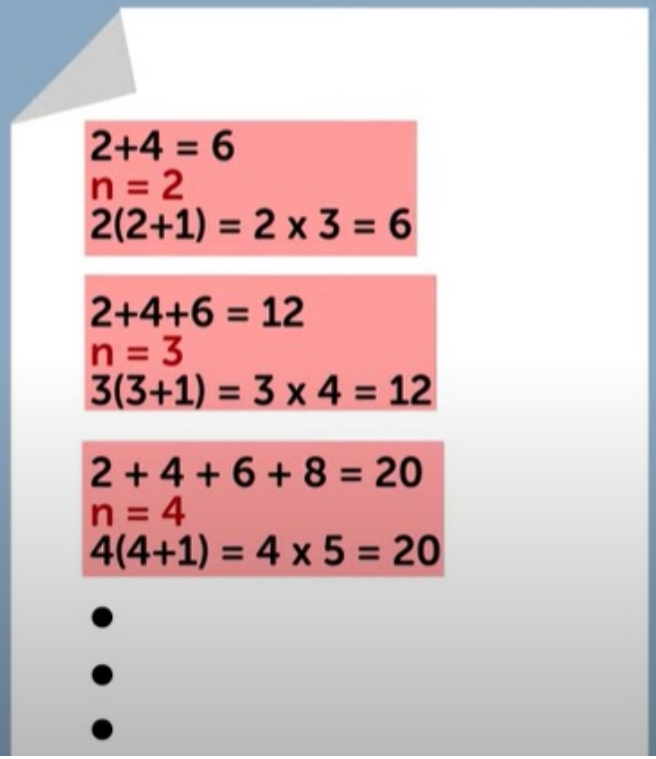
Conjecture : inferring

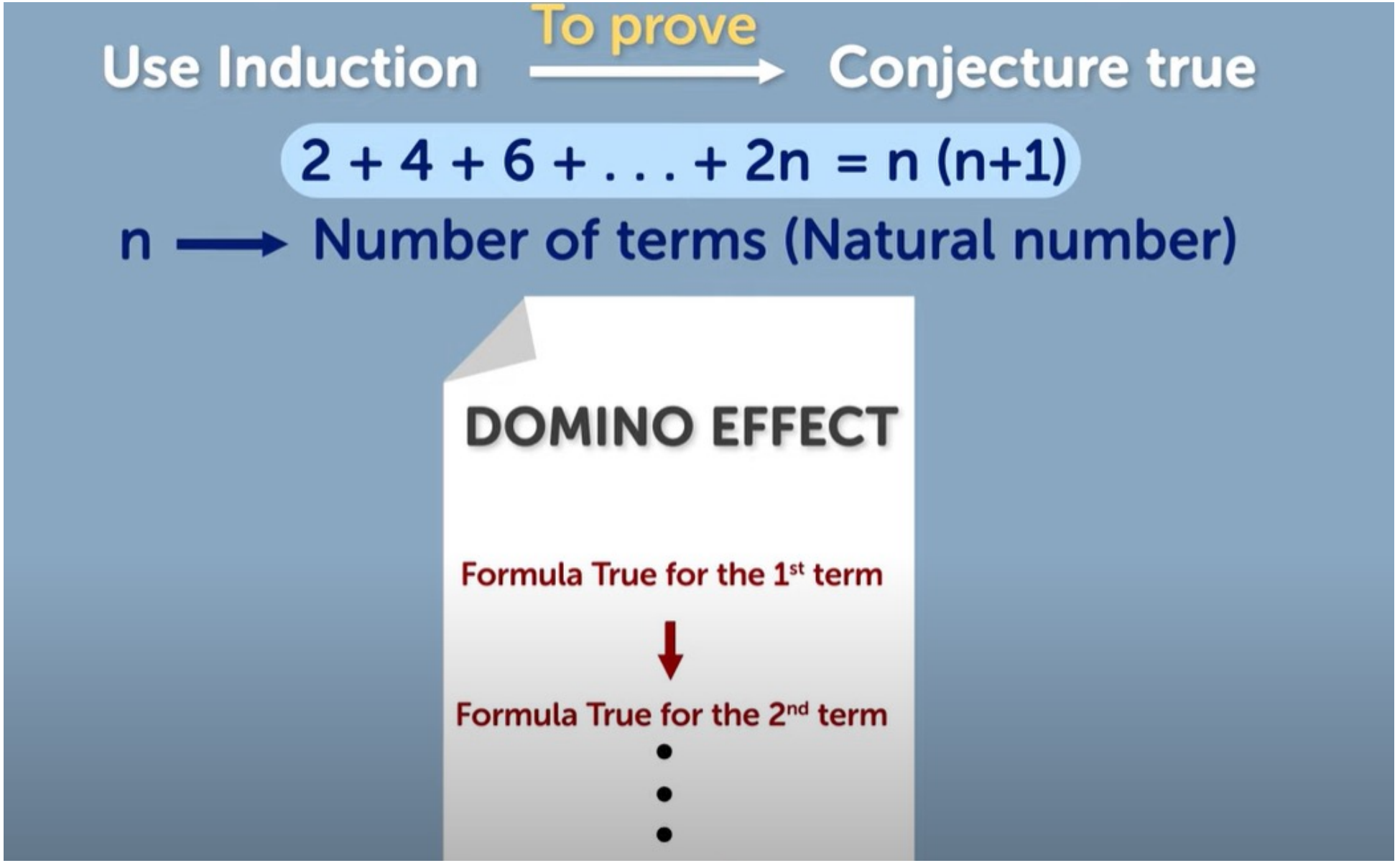


Use Induction → To prove → Conjecture true

$$2 + 4 + 6 + \ldots + 2n = n(n+1)$$

n → Number of terms (Natural number)




Use Induction
To prove
Conjecture true
2 + 4 + 6 + . . . + 2n = n (n+1)
n
Number of terms (Natural number)
DOMINO EFFECT
Formula True for the 1st term
Formula True for the 2nd term

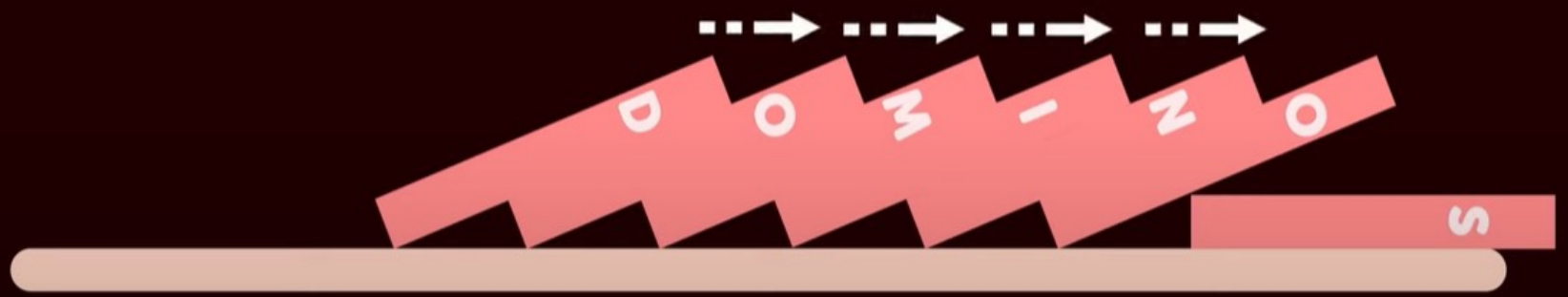
DOMINO EFFECT

Condition 1 :

When the first domino falls, it will hit the second one

Condition 2 :

Each domino will hit the next one and each domino that is hit will fall



Goal → All the dominos fall

Use Induction → To prove → Conjecture true

$$2 + 4 + 6 + \ldots + 2n = n(n+1)$$

Base case :

Condition 1 : P(1) is True for n = 1

n = 1 → First term

P → Propositional statement

Eg :- Statement : x is Cold

P(x) → True or False (depends on x)

x is ice → P(x) is TRUE

x is fire → P(x) is FALSE

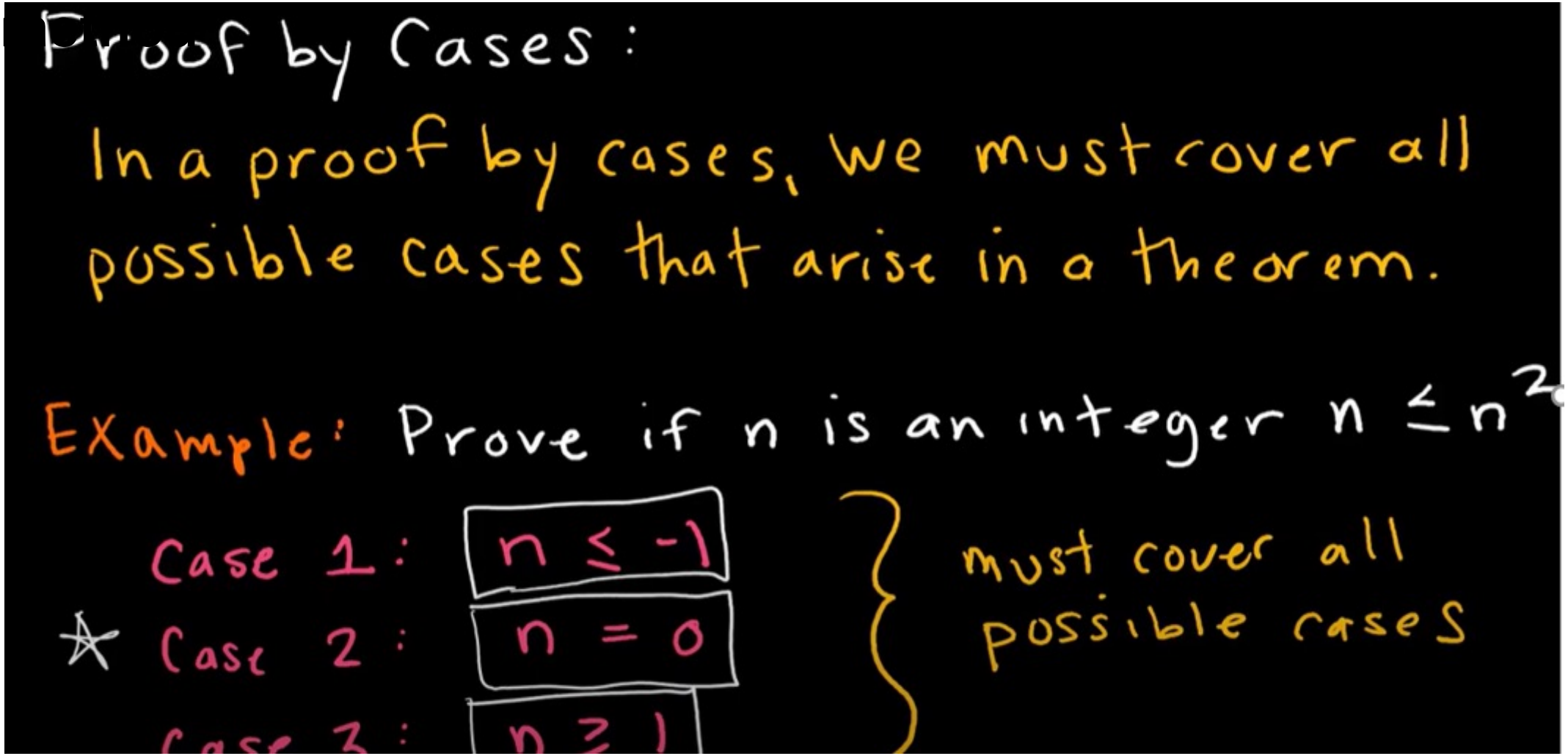
Inductive case :

Condition 2 : If P(n) true for n = k, then P(n) true for n = k+1

{P(k) true → P(k+1) true}

Inductive hypothesis

Proof by Case



Prove "If n is an integer, $n \leq n^2$"

Case 1: $n \leq -1$ $n^2 \geq 0$, so it follows that $n \leq n^2$

Case 2: $n = 0$ $0 \leq 0^2$, $0 \leq 0$, so it follows that $n \leq n^2$

Case 3: $n \geq 1$ $n \geq 1$, $\cdot n \quad \cdot n$, $n^2 \geq n$, so it follows that $n \leq n^2$.

Since our inequality $n \leq n^2$ is true for all possible cases, we can conclude $n \leq n^2$ for all integers.

□

Prove " If n is an integer, $n^2 + 3n + 2$ is even."

Cases: n is odd, n is even

Case 1: n is odd

$n = 2k+1$

$= (2k+1)^2 + 3(2k+1) + 2$

$= 4k^2 + 4k + 1 + 6k + 3 + 2$

$= 4k^2 + 10k + 6$

$= 2(2k^2 + 5k + 3)$

Case 2: n is even

$n = 2k$

$= (2k)^2 + 3(2k) + 2$

$= 4k^2 + 6k + 2$

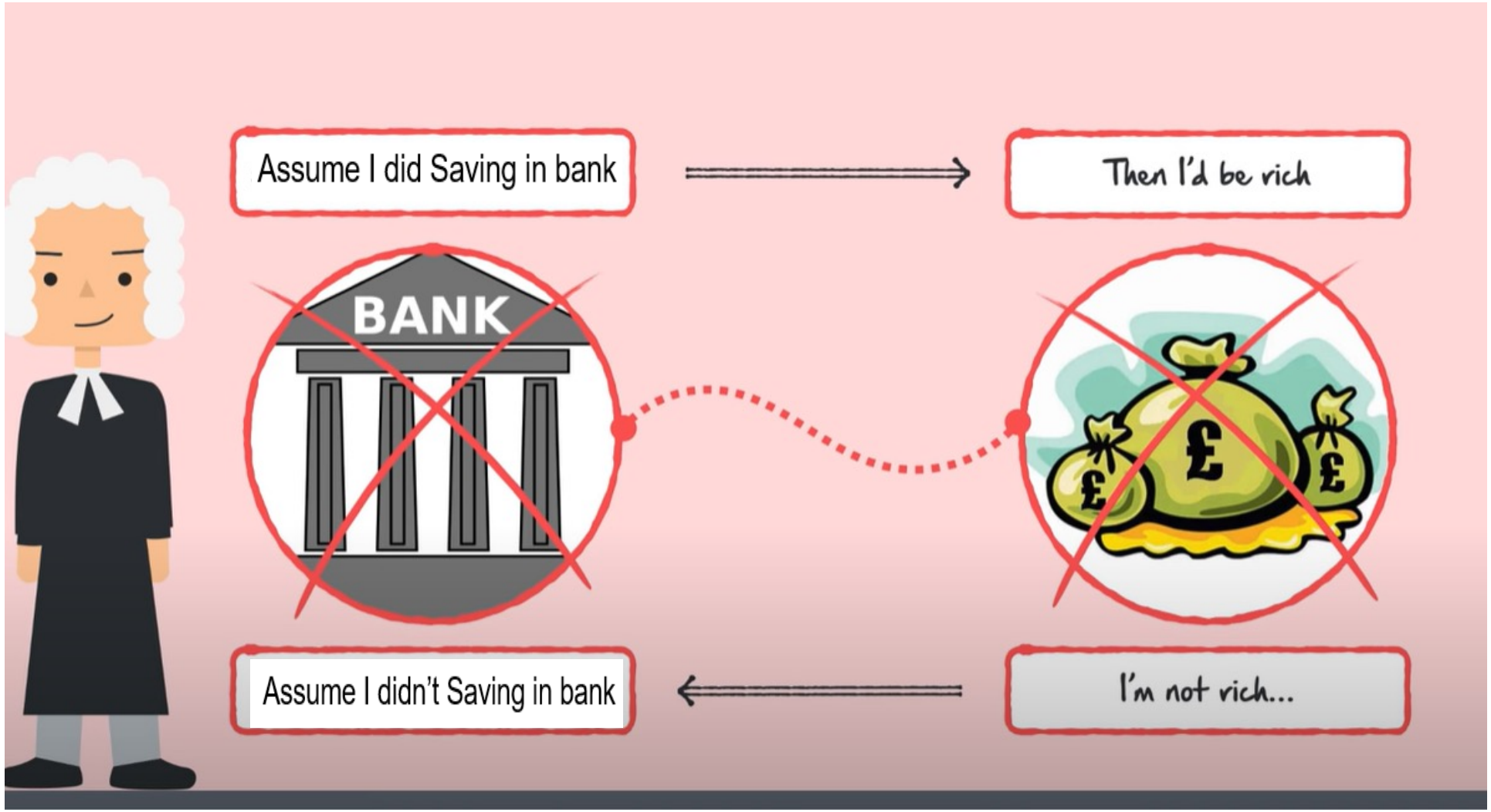
$= 2(2k^2 + 3k + 1)$

Since both cases result in even values, then $n^2 + 3n + 2$ is even if n is an integer.

□

Proof by chain of IFFs?

Assume I didn’t Saving in bank



Proof By Contradiction:

What is it?

Proof By Contradiction: A way to prove something is true by showing that if it wasn't true, that would lead to a logical error

So that's basically Latin for reducing your assumption to something absurd

not at all logical

REDUCTIO AD ABSURDUM

Proof by Contradiction

We want to prove that a statement P is true

- we assume that P is false
- then we arrive at an incorrect conclusion
- therefore, statement P must be true

Example

Theorem: $\sqrt{2}$ is not rational

Proof:

Assume by contradiction that it is rational

$\sqrt{2} = n/m$

n and m have no common factors

We will show that this is impossible

1. Prove that $\sqrt{2}$ is irrational by giving a proof using contradiction.

Solution:

Assume the contradiction that $\sqrt{2}$ is rational.

$\therefore \sqrt{2} = \frac{a}{b}$ where a and b have no common factors.

$$\frac{a}{b} = \sqrt{2}$$

$$\therefore \Rightarrow \frac{a^2}{b^2} = 2 \Rightarrow a^2 = 2b^2 \quad ---(1)$$

This means that a^2 is even $\Rightarrow a$ is even. Let $a = 2c$.

$$
\begin{aligned}
(1) \Rightarrow \quad & a^2 = 2b^2 \\
\Rightarrow \quad & (2c)^2 = 2b^2 \\
\Rightarrow \quad & 4c^2 = 2b^2 \\
\Rightarrow \quad & 2c^2 = b^2 \\
\Rightarrow \quad & b^2 \text{ is even } \Rightarrow \ b \text{ is even.}
\end{aligned}
$$

Thus, a and b are even. Hence, they have a common factor 2, which is a contradiction to our assumption. $\sqrt{2}$ is irrational.

$\sqrt{2} = n/m \Rightarrow 2m^2 = n^2$

Therefore, n^2 is even $\Rightarrow$ n is even
$n = 2k$

$2m^2 = 4k^2 \Rightarrow m^2 = 2k^2 \Rightarrow$ m is even
$m = 2p$

Thus, m and n have common factor 2

Contradiction!

Contrapositive

Statement: $p \rightarrow q$

Contrapositive: $\sim q \rightarrow \sim p$

If Johnny likes apples,
then Johnny eats fruit.

If Johnny doesn't eat fruit,
then Johnny doesn't like
apples.

If $7x+9$ is even, then x is odd.

Suppose x is not odd. Thus x is even, so $x = 2a$ for an integer a. So then

$$7x+9 = 7(2a)+9 = 14a+9 = 14a+8+1$$
$$= 2(7a+4)+1.$$

Thus $7x+9 = 2b+1$ where b is the integer $7a+4$.

Therefore $7x+9$ is odd.

2.Prove that if $3n + 2$ is odd then n is odd.

Solution:

We prove this problem by the method of contrapositive.

Suppose n is even $\Rightarrow n = 2k$.

Now $3n + 2 = 3(2k) + 2$

$$3n + 2 = 2(3k + 1) \ldots(1)$$

$\therefore$ Equation (1) is a multiple of 2. $\therefore$ $3n + 2$ is even number.

$\Rightarrow 3n + 2$ is not a odd number.

$\therefore$ If $3n + 2$ is odd then n is odd.

Loop Invariant

Definition:

A loop invariant is a condition [among program variables] that is necessarily true immediately before and immediately after each iteration of a loop. (Note that this says nothing about its truth or falsity part way through an iteration.)

A loop invariant is some predicate (condition) that holds for every iteration of the loop.

For example, let's look at a simple for loop that looks like this:

```
int j = 9;
for(int i=0; i<10; i++)
j--;
```

The loop invariant must be true:

- before the loop starts
- before each iteration of the loop
- after the loop terminates

(although it can temporarily be false during the body of the loop).

On the other hand the loop conditional must be false after the loop terminates, otherwise, the loop would never terminate.

```
for (i = 0 to n-1)
  if (A[i] > max)
      max = A[i]
```

In the above example after the 3rd iteration of the loop max value is 7, which holds true for the first 3 elements of array A. Here, the loop invariant condition is that max is always maximum among the first i elements of array A.

Performance Analysis

- Time Complexity
- Space Complexity
- Communication Bandwidth

Space Complexity Analysis

- The space complexity of an algorithm is the amount of memory it needs to run to completion.

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

Example

- #include<stdio.h>

```
int main()
 {
   int a = 5, b = 5, c;
   c = a + b;
   printf("%d", c);
 }
```

Space Complexity: size(a)+size(b)+size(c)

=>let sizeof(int)=2 bytes=>2+2+2=6 bytes

=>**O(1) or constant**

Example

```
#include <stdio.h>
   int main()
  {
    int n, i, sum = 0;
    scanf("%d", &n);
    int arr[n];
   for(i = 0; i < n; i++)
    {
       scanf("%d", &arr[i]); sum = sum + arr[i];
     }
    printf("%d", sum);
  }
```

Space Complexity:

- **The array consists of n integer elements.**
- **So, the space occupied by the array is 4 * n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes.**
- **Since the highest order of n in the equation 4n + 12 is n, so the space complexity is O(n) or linear.**

Time complexity analysis

Session Learning Outcome-SLO:

Able to write algorithms and analyze its complexity

- The time T(P) taken by a program P is

  T(P) =compile time + run (or execution) time.

- The compile time does not depend on the **instance** characteristics.
- Also, the compiled program will be run several times without recompilation. Consequently, the run time of a program only considered. This run time is denoted by t_p (instance characteristics)
- As the factors that influence t_p , when the program run it is better to estimate t_p
- If the compiler characteristics and the time taken for addition, subtraction, division, multiplication etc.. are known then we can find t_p as

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

- Where n denotes the instance characteristics, and c_a,c_s,c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on.

- ADD, SUB,MUL,DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

- It is difficult to get such an exact formula because the time taken for addition and other operations depends on numbers that are being added.

- So alternative is to do all activities such as program typing, compiling and running on a machine and the execution time is physically clocked to get $t_p(n)$.

- Drawbacks of experimental approach is the execution time depends on the other programs running on the computer at the time the program P run. It also depends on machine architecture.

Solution:

- Obtain a count for the total number of operations in the algorithm. We can go one step further and count only the number of **program steps.**

- A **program step** is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement is a single program step.

$$\textbf{return } a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

- The number of steps any program statement is assigned depends on the kind of statement.
- Comments count as zero steps;
- an assignment statement which does not involve any calls to other algorithms is counted as one step;
- In an iterative statement such as the for, while, and repeat-until statements, we consider the step counts only for the control part of the statement.

Two methods to find the time complexity for an algorithm

1. Count variable method
2. Table method

Using Count Method

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N){
    int s=0;
    for (int i=0; i< N; i++)
        s = s + A[i];
    return s;
}
```

1,2,8: Once
3,4,5,6,7: Once per each iteration of for loop, N iteration
Total: 5N + 3
The *complexity function* of the algorithm is : $f(N) = 5N + 3$

Table method

- The table contains s/e and frequency.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- Frequency is defined as the total number of times each statement is executed.
- Combining these two, the step count for the entire algorithm is obtained.

Example 1

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

Example 2

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a,n-1)+a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Class Activity- Find Time Complexity

1.
```
for (let i = 0; i < n; ++i)
    {
            console.log(i);
     }
```

O(n) ---------- Linear Time Complexity

2.
```
for (let i = 0; i < n; ++i)
  {
     for (let j = 0; j < n; ++j)
   {
     console.log(i, j);
   }
  }
```

$O(n^2)$ --------- Exponential Time Complexity

3.
```
for (let i = 1; i < n; i *= 2)
     {
     console.log(i);
     }
```

O(log2n) ----------Logarithmic Time Complexity

Frequency count Method

1. Algorithm Sum(A, n)

```
{
    S = 0;                          — 1
    for ( i = 0; i < n; i++)        — n+1
    {                                  (1)  (n+1)  (n)
        S = S + A[i];               — n
    }
    return S;                       — 1
}
```

$f(n) = 2n+2$

A | 2 | 3 | 4 | 5 | 6 | (where n = 5)
0 1 2 3 4

* Assign 1 unit of time for each Statement

* If any Statement is Repeated for Some number of times, the frequency of execution ~~will~~ of Statement will be Calculated to find the taken time by the Algorithm.

$f(n) = 2n+3 \Rightarrow O(n)$

∴ Degree of polynomial = 1
∴ $O(n)$

SPACE COMPLEXITY

Size of A is

A	—	n words
n	—	1 word
S	—	1 word
i	—	1 word

$S(n) = n+3$ words

Degree of polynomial is (1)

∴ $O(n)$

Ex:2: n×n matrix eg:- 3×3

Algorithm Add(A, B, n)

```
{
  for (i = 0; i < n; i++)           —— n+1
  {
    for (j = 0; j < n; j++)         — n x (n+1)
    {
      C[i,j] = A[i,j] + B[i,j];     — n×n
    }
  }
}
```

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

Degree of polynomial is n^2

∴ $O(n^2)$

SPACE COMPLEXITY

A — n^2
B — n^2
C — n^2
n — 1 word
i — 1 word
j — 1 word

$S(n) = 3n^2 + 3$

Degree of polynomial is n^2

∴ $O(n^2)$

Ex: 3:

```
Algorithm Multiply (A,B,n)
{
(n+1)   for (i=0; i<n; i++)
        {
(n+1)xn     for (j=0; j<n; j++)
            {
n x n           C[i,j] = 0;
(n+1)x n x n        for (k=0; k<n; k++)
                    {
n x nx n                C[i,j] = c[i,j] + A[i,k] * B[k,j];
                    }
            }
        }
```

$\begin{bmatrix} 6 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 5 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & \\ & \end{bmatrix}$

$n+1$	=	$n+1$
$(n+1) \times n$	=	n^2+n
$n \times n$	=	n^2
$(n+1) \times n \times n$	=	n^3+n^2
$n \times n \times n$	=	n^3

$f(n) = 2n^3 + 3n^2 + 2n + 1$

Degree of polynomial = 3

$O(n^3)$

SPACE COMPLEXITY

A — n^2
B — n^2
C — n^2
n — 1
i — 1
j — 1
k — 1

$S(n) = 3n^2 + 4$

Degree of Polynomial = n^2

$O(n^2)$

Ex : 4 :

```
for(i=0; i<n; i++)        ___ n+1
{
  Stmt;                   ___ n
}
```

$f(n) = 2n+1$

Degree of polynomial = 1 $\Rightarrow$ (n)

$\therefore O(n)$

```
for(i= n; i>0; i--)       ___ n+1
{
  Stmt;                   ___ n
}
```

$f(n) = 2n+1$

$O(n)$

```
for(i=1; i<n; i=i+2)
{
  Stmt;                   ___ n/2
}
```

$f(n) = n/2$

Degree of polynomial = 1 $\Rightarrow n$

$\therefore O(n)$

```
for(i=1; i<n; i=i+20)
{
  Stmt;                   ___ n/20
}
```

$f(n) = n/20$

$O(n)$

Ex 5 :

```
for(i=0; i<n; i++)        ___ n+1
{
  for(j=0; j<n; j++)      ___ n(n+1)
  {
    Stmt;                 ___ n x n
  }
}
```

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

$O(n^2)$

Ex 6:

```
for (i=0; i<n; i++)
{
    for (j=0; j<i; j++)
    {
        Stmt;
    }
}
```

i	j	Stmt [NO. OF TIMES]
0	0 [NE]	0
1	0 [E]	1
	1 [NE]	
2	0 [E]	2
	1 [E]	
	2 [NE]	
3	0 [E]	3
	1 [E]	
	2 [E]	
	3 [NE]	

E => Executed
NE => Not Executed.

$1+2+3+\cdots+n$

$\Rightarrow \frac{n(n+1)}{2}$

$\Rightarrow \frac{n^2+1}{2}$

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

$O(n^2)$

$\cdots > n.$

Ex: 7:

```
P=0;
for(i=1; P<=n; i++)
{
   P= P+i;
}
```

i	P
1	0+1 = 1
2	1+2 = 3 ie 1+2
3	3+3 = 6 ie 1+2+3
4	6+4 = 10 ie 1+2+3+4
⋮	
k	1+2+3+4+5+....+k

Assume $P > n$ then the loop terminates.

$$P = \frac{k(k+1)}{2}$$

Stops when $\frac{k(k+1)}{2} > n$

$$\frac{k^2}{2} + k > n$$

$$k^2 \geq n$$

$$k > \sqrt{n}$$

$\therefore f(n) = \sqrt{n}$

$O(\sqrt{n})$

Ex 8:

```
for (i = 1; i < n; i = i * 2)
{
    Stmt;
}
```

i
1
$1 \times 2 = 2$
$2 \times 2 = 2^2$
$2 \times 2 \times 2 = 2^3$
⋮
2^k

Assume $i >= n$ Stops

$i = 2^k$

which means

$2^k >= n$

$$\boxed{2^k = n}$$

$$k = \log_2 n \Rightarrow O(\log_2 n)$$

Ex : 9

```
for(i=1; i<n; i=i*2)
  {
    Stmt;
  }
```

$i = 1 \times 2 \times 2 \times \ldots = n$

$2^k = n$

$\boxed{k = \log_2 n}$

$\Rightarrow O(\log_2 n)$

```
for(i=1; i<=n; i++)
  {
    Stmt;
  }
```

$i = 1 + 1 + 1 + 1 \ldots + 1 = n$

$k = n.$

$\Rightarrow O(n)$

Ex: 10

```
for(i=1; i<n; i=i*2)
{
  Stmt;
}
```

$\longrightarrow \log_2 n$

$\log n$ is a floating point value

which one to consider

either ceil or floor ?

$n = 8$

i

1
2
4
8

3 times executed

$\log_2 8 = 3$

ie $\log_2 2^3 = 3 \log_2 2$

$\Rightarrow 3 \times 1$

$\Rightarrow 3$

$n = 10$

i

1
2
4
8
16

4 times executed

$\log_2 10$

$\log_2 10$

No exact 2^{Powers} for 10

$\log_2 10 \Rightarrow$ Assume 3.2

Consider ceil

$\lceil \log n \rceil$

Ex: 11

```
for(i=n; i>=1; i=i/2)
{
   Stmt;
}
```

```
for(i=n; i>=1; i=i/2)
{
   Stmt;
}
```

i
n
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
$\frac{n}{2^k}$

Assume $i<1$

$$\frac{n}{2^k} < 1$$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$k = \log_2 n \Rightarrow O(\log_2 n)$$

Ex No: 12

```
for (i=0; i*i<n; i++)
{
    Stmt;
}
```

$i * i < n$

$i * i >= n \Rightarrow$ Stops.

$i^2 = n$

$i = \sqrt{n} \Rightarrow O(\sqrt{n})$

Ex : 13 Independent Loop.

```
for (i=0; i<n; i++)
{
  Stmt;
}
```

——— n

```
for (j=0; j<n; j++)
{
  Stmt;
}
```

——— n

$f(n) = 2n$

$O(n)$

Ex : 14

```
P=0
for (i=1; i<n; i=i*2)
{
    P++;          ---- logn
}

for (j=1; j<P; j=j*2)
{
    Stmt;         ---- log P
}
```

P is evaluated first then used in next loop ∴ P = logn ∴ for log P its log logn ⇒ O(log log n)

Ex: 15

```
for(i=0; i<n; i++)          ____ n+1
{
  for(j=1; j<n; j=j*2)      ____ n*logn
    {
      stmt;                 ____ n x logn
    }
}
```

$f(n) = 2n\log n + n + 1$

$O(n \log n)$

GENERAL

for(i=0; i<n; i++) ⟶ O(n)

for(i=0; i<n; i=i+2) → $\frac{n}{2}$ O(n)

for(i=n; i>1; i--) ⟶ O(n)

for(i=1; i<n; i=i*2) = $O(\log_2 n)$

for(i=1; i<n; i=i*3) = $O(\log_3 n)$

for(i=n; i>1; i=i/2) = $O(\log_2 n)$

$\frac{n}{2}$ => O(n) $\frac{n}{200}$ => O(n)

Home assignment

Calculate the time complexity for the below algorithms using table method

1.

```
1   Algorithm Fibonacci(n)
2   // Compute the nth Fibonacci number.
3   {
4       if (n ≤ 1) then
5           write (n);
6       else
7       {
8           fnm2 := 0; fnm1 := 1;
9           for i := 2 to n do
10          {
11              fn := fnm1 + fnm2;
12              fnm2 := fnm1; fnm1 := fn;
13          }
14          write (fn);
15      }
16  }
```

2. Determine the frequency counts for all statements in the following two algorithm segments:

```
1  for i := 1 to n do
2      for j := 1 to i do
3          for k := 1 to j do
4              x := x + 1;
```

(a)

```
1  i := 1;
2  while (i ≤ n) do
3  {
4      x := x + 1;
5      i := i + 1;
6  }
```

(b)

3.

```
1  Algorithm Transpose(a, n)
2  {
3      for i := 1 to n − 1 do
4          for j := i + 1 to n do
5          {
6              t := a[i, j]; a[i, j] := a[j, i]; a[j, i] := t;
7          }
8  }
```

Insertion Sort

INTRODUCTION

- Session Learning Outcome - SLO – To Understand the concept of Insertion sort, the time complexity of an algorithm and Algorithm design paradigms.

ALGORITHM DESIGN AND ANALYSIS

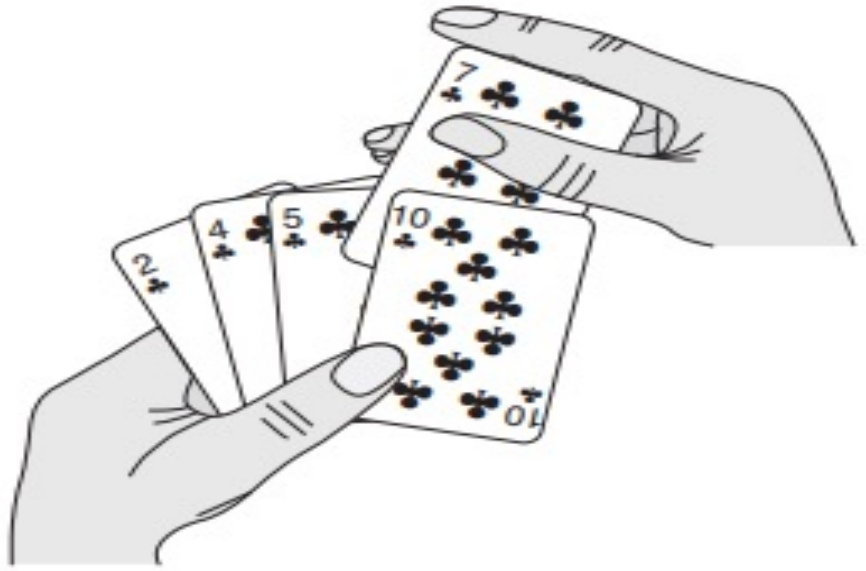
Efficiency of the algorithm's design is totally dependent on the understanding of the problem. The important parameters to be considered in understanding the problem are:

- Input
- Output
- Order of Instructions
- Check for repetition of Instructions
- Check for the decisions based on conditions

INSERTION SORT

- Insertion Sort Algorithm sorts array by shifting elements one by one and inserting the right element at the right position

- It works similar to the way you sort playing cards in your hands



PROCEDURE

- We start by making the second element of the given array, i.e. element at index 1, the key.
- We compare the key element with the element(s) before it, i.e., element at index 0:
  - If the key element is less than the first element, we insert the key element before the first element.
  - If the key element is greater than the first element, then we insert it after the first element.
- Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
- And we go on repeating this, until the array is sorted.

EXAMPLE

- Step 1:

7	2	8	1	5	3

- Step 2: (A[1] is compared with A[0], Since 2<7, its swapped)

2	7	8	1	5	3

- Step 3: (A[2] is compared with A[1] and A[0]. As 8 is the largest of all 3, no swapping is done)

2	7	8	1	5	3

EXAMPLE (Contd..)

- Step 4: (A[3] is compared with A[2], A[1] and A[0]. Since 1 is the smallest of all, its is swapped)

1	2	7	8	5	3

- Step 5: (A[4] is compared with A[3], A[2], A[1] and A[0]. Since 5 is lesser than 8 and 7 and greater than 2, it is swapped and placed between 2 and 7)

1	2	5	7	8	3

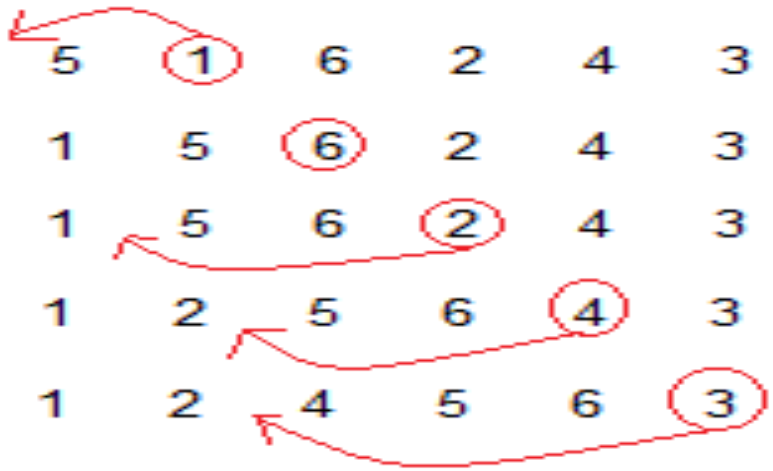
EXAMPLE (Contd..)

- Step 6: (A[5] is compared with A[4], A[3], A[2], A[1] and A[0]. Since 3 is lesser than 8,7,5 and greater than 1 &2, it is placed between 2 and 5).

1	2	3	5	7	8

- Array is sorted.

5	1	6	2	4	3



(Always we start with the second element as key.)

Lets take this Array.

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

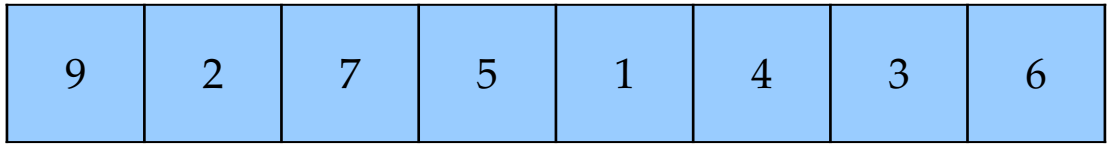
6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

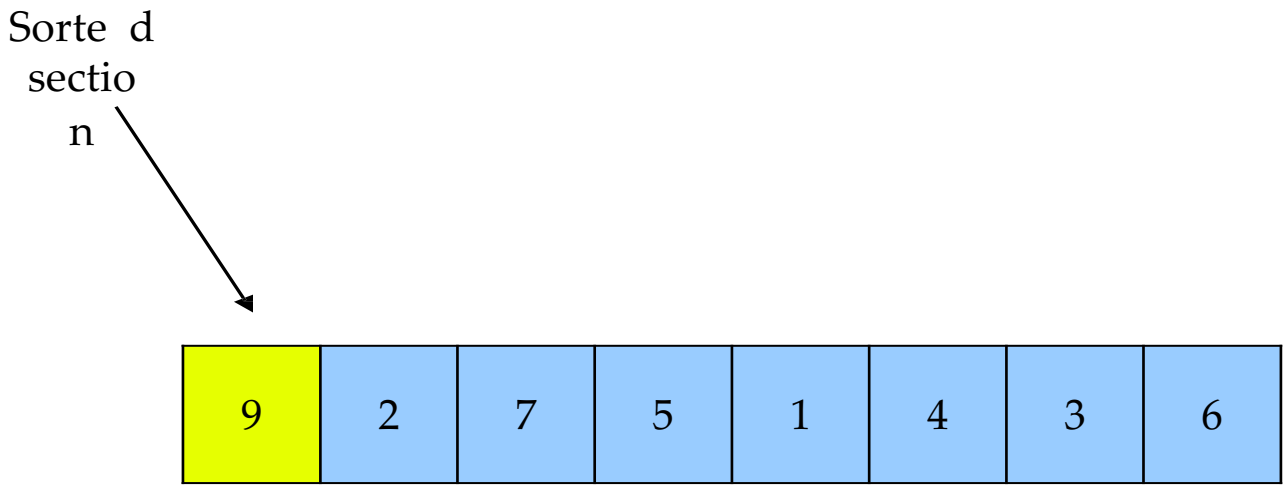
And this goes on...

Insertion Sort

Example :

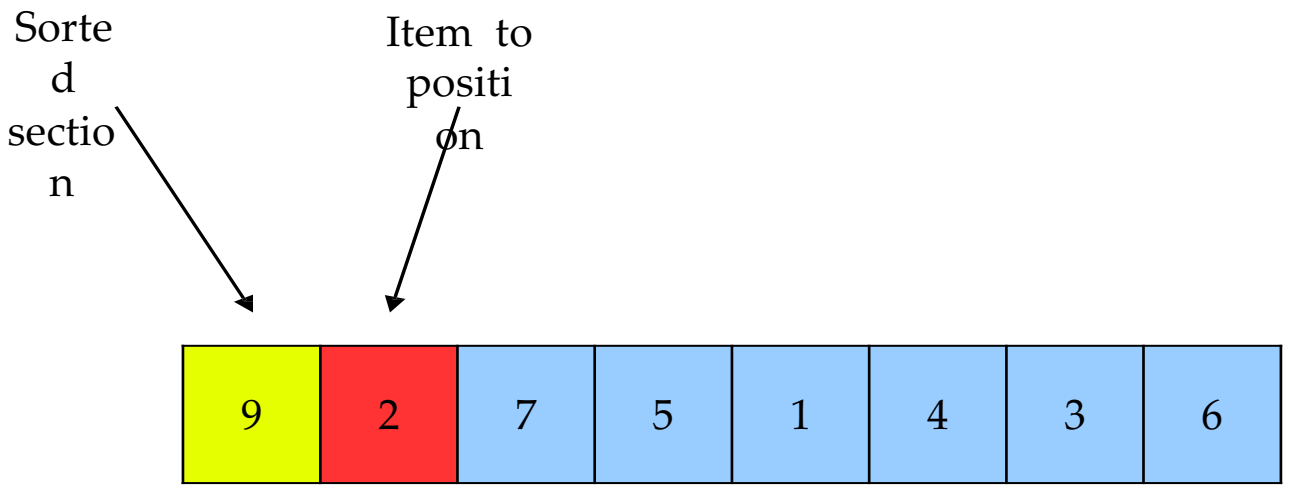


Insertion Sort



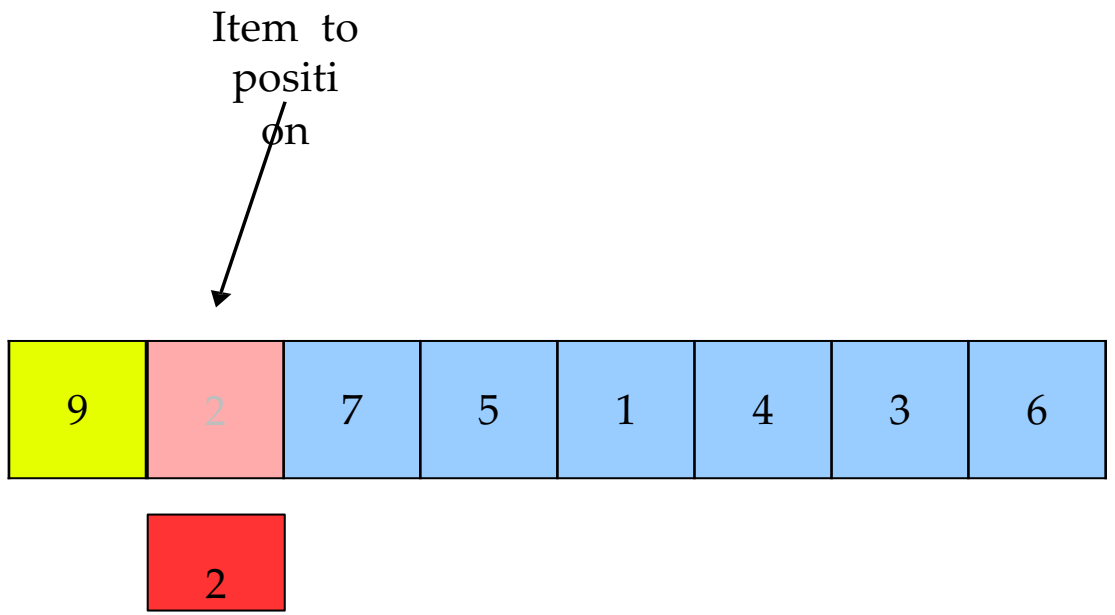
We start by dividing the array in a sorted section and an unsorted section. We put the first element as the only element in the sorted section, and the rest of the array is the unsorted section.

Insertion Sort



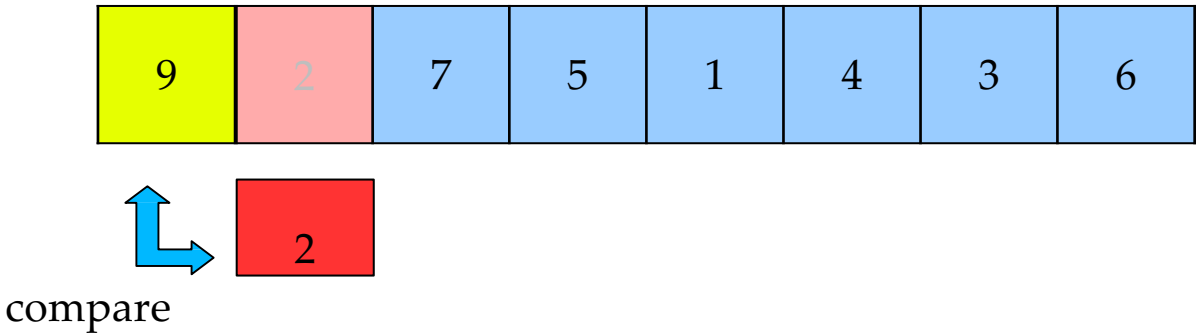
The first element in the unsorted section is the next element to be put into the correct position.

Insertion Sort



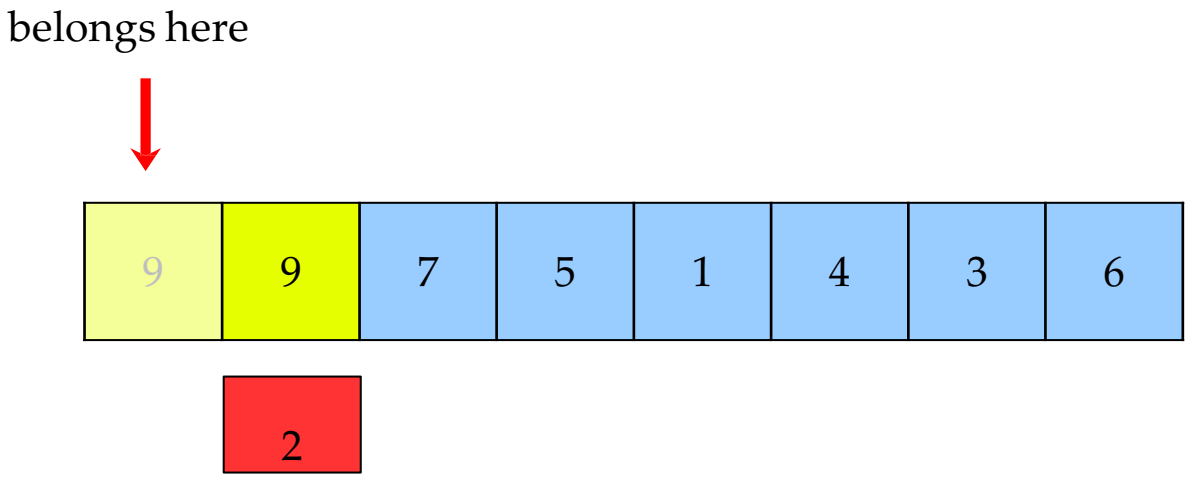
We copy the element to be placed into another variable so it doesn't get overwritten.

Insertion Sort



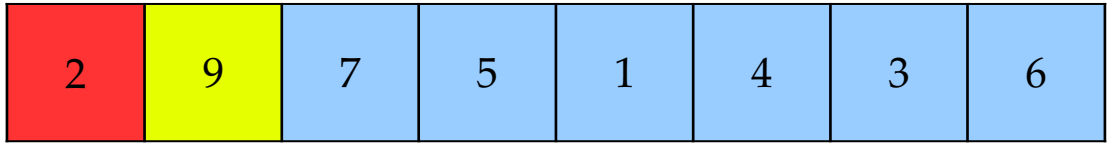
If the previous position is more than the item being placed, copy the value into the next position

Insertion Sort

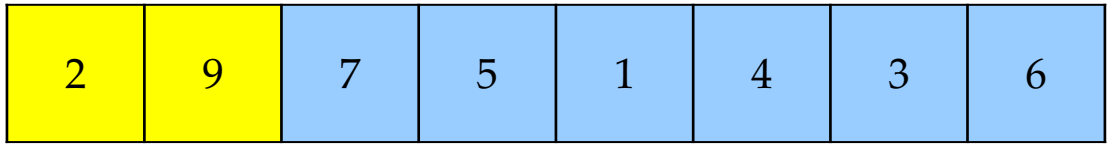


If there are no more items in the sorted section to compare with, the item to be placed must go at the front.

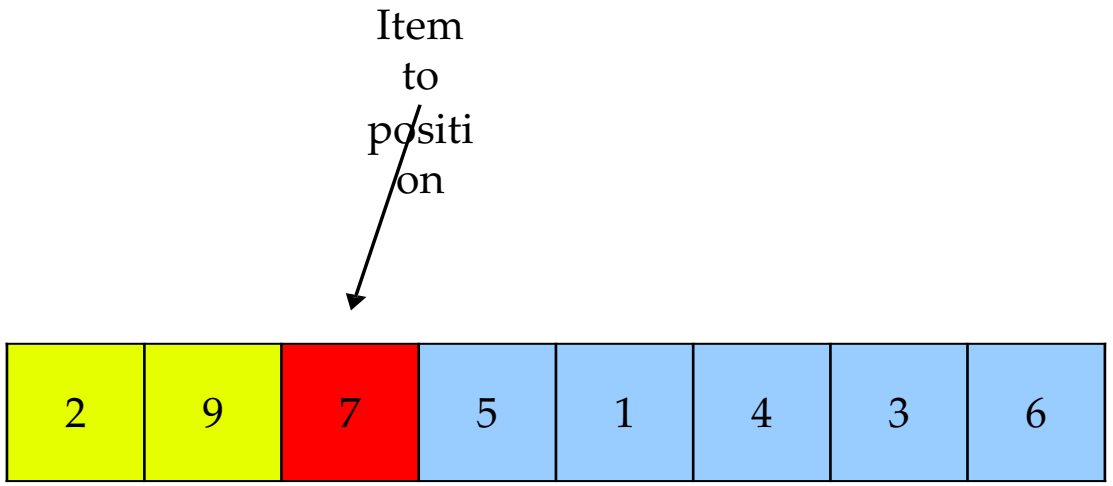
Insertion Sort



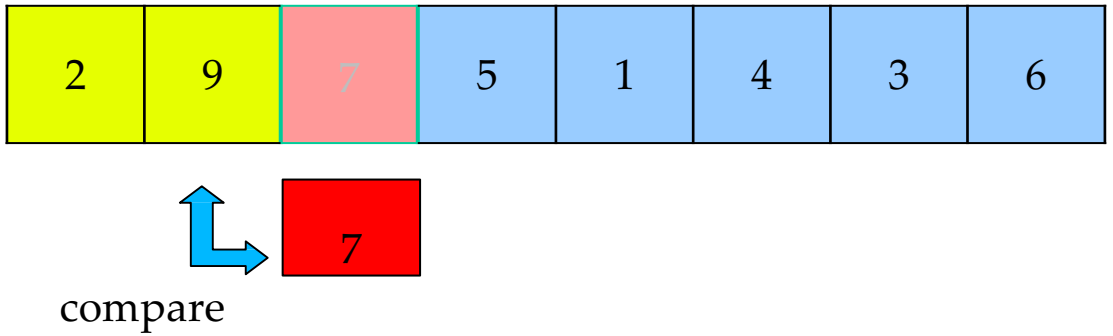
Insertion Sort



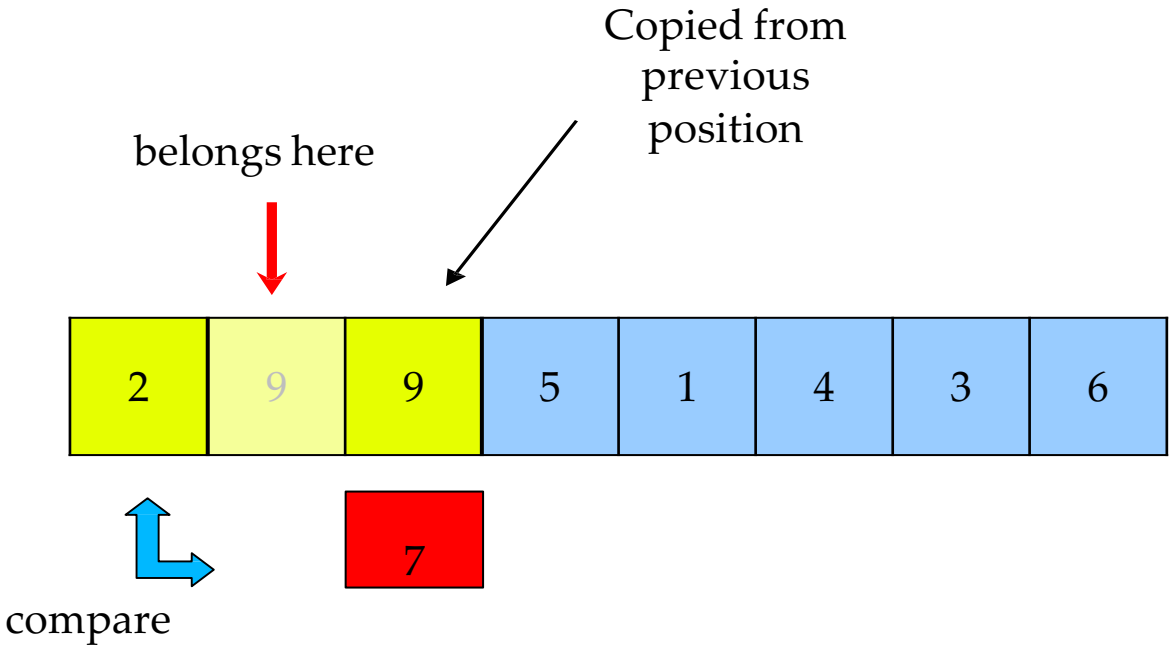
Insertion Sort



Insertion Sort

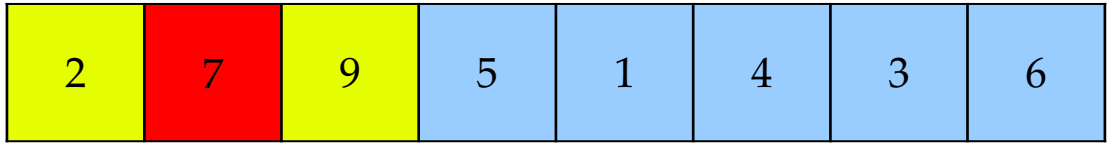


Insertion Sort

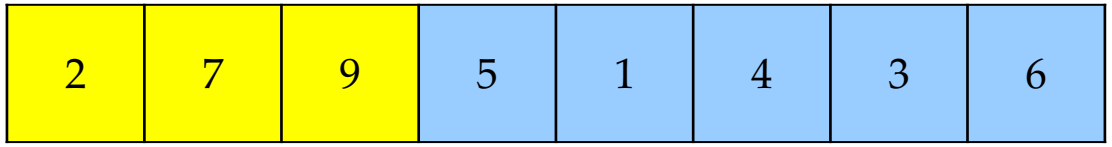


If the item in the sorted section is less than the item to place, the item to place goes *after* it in the array.

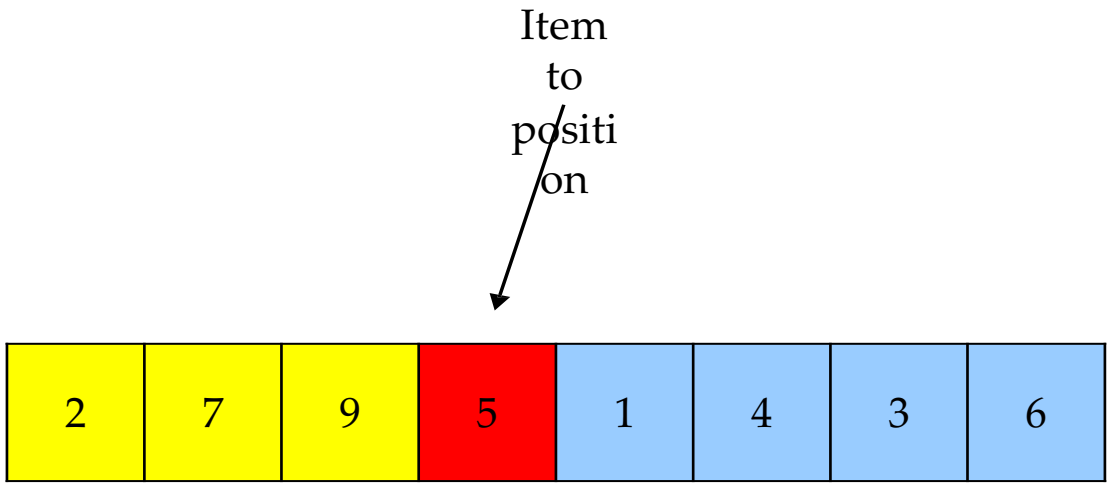
Insertion Sort



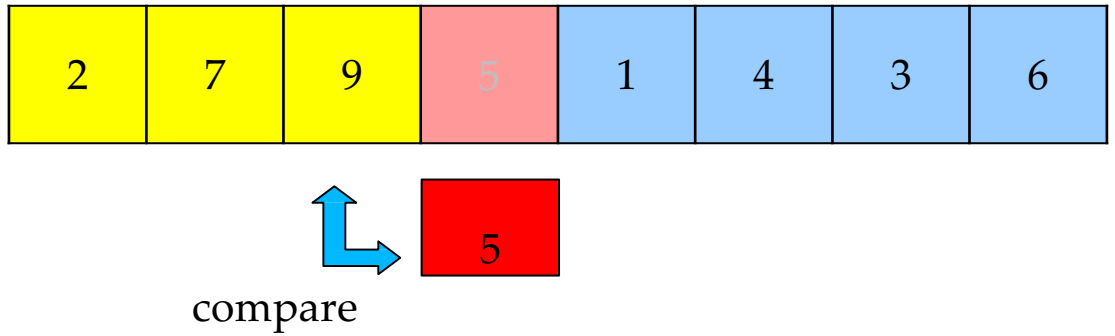
Insertion Sort



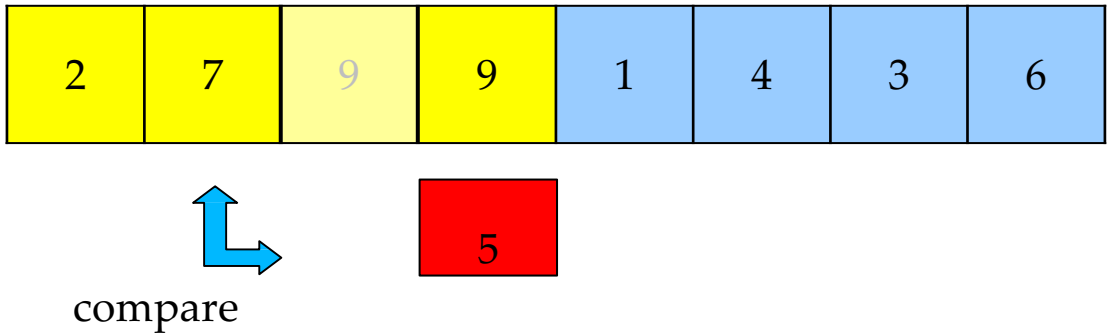
Insertion Sort



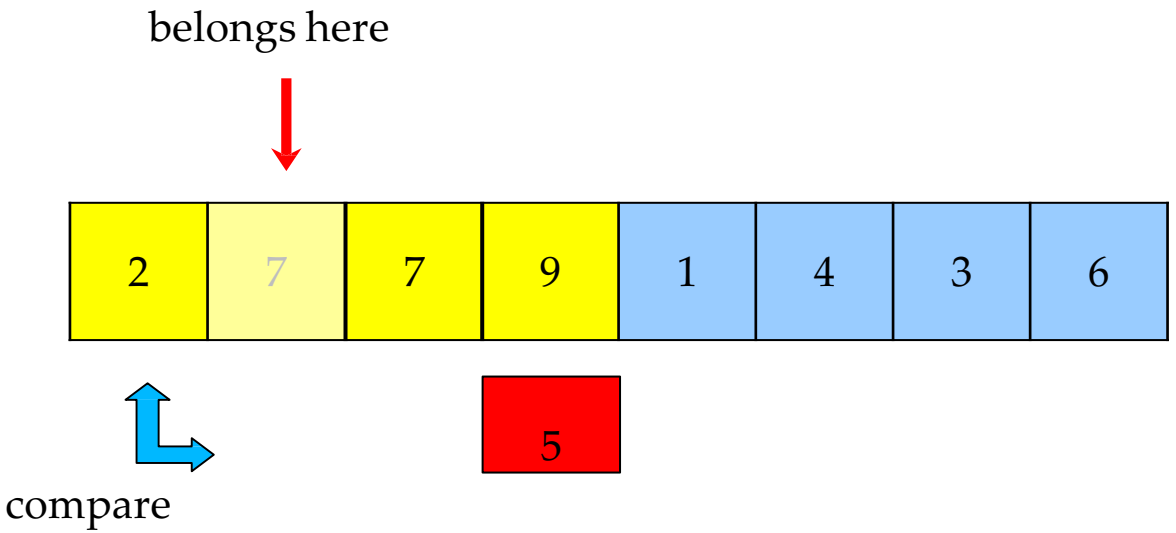
Insertion Sort



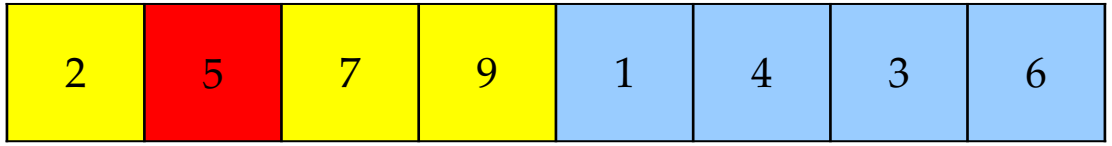
Insertion Sort



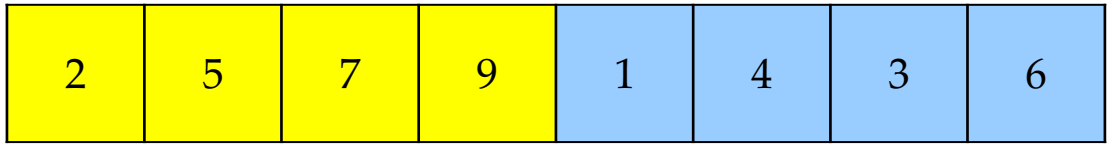
Insertion Sort



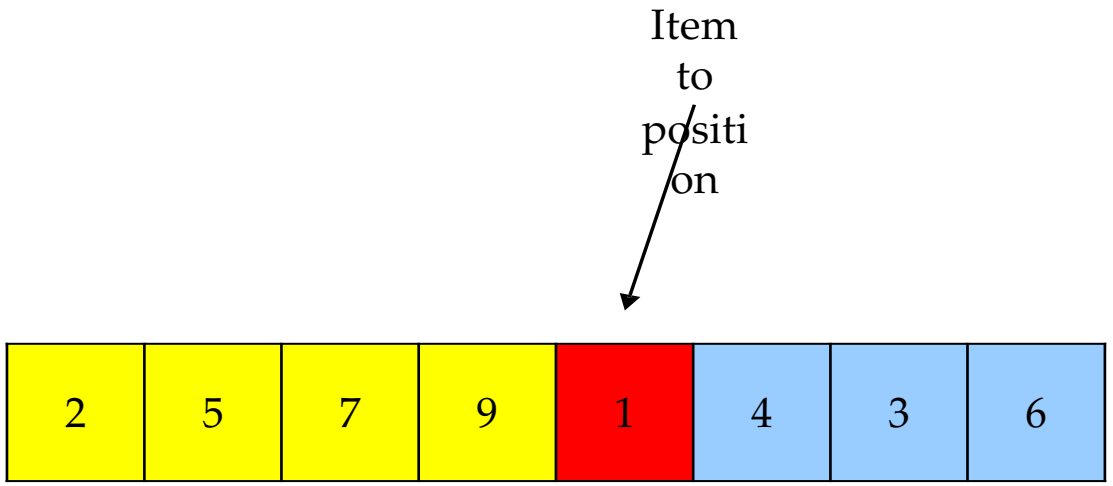
Insertion Sort



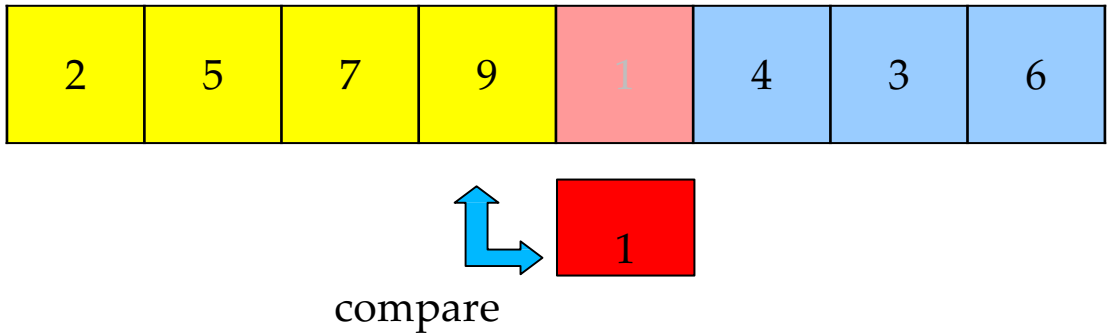
Insertion Sort



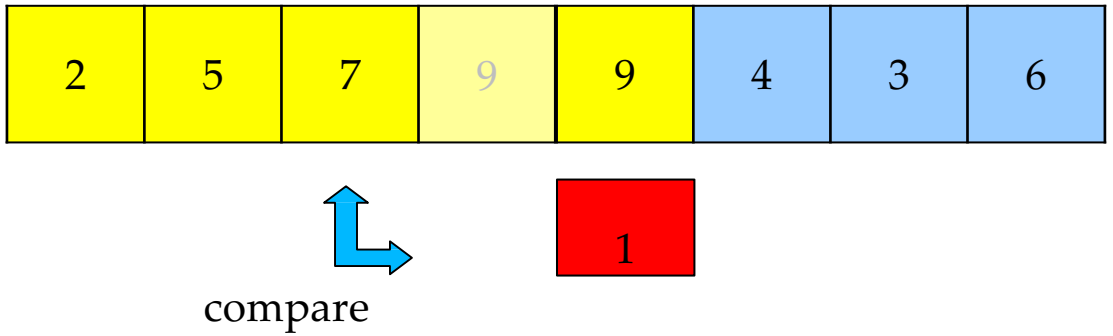
Insertion Sort



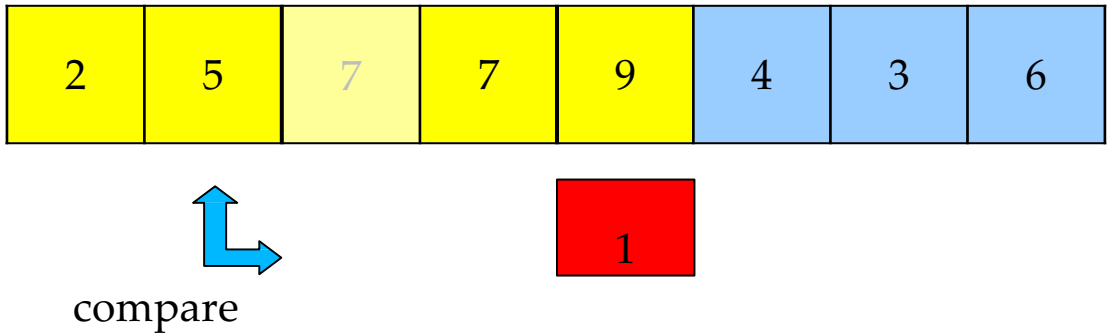
Insertion Sort



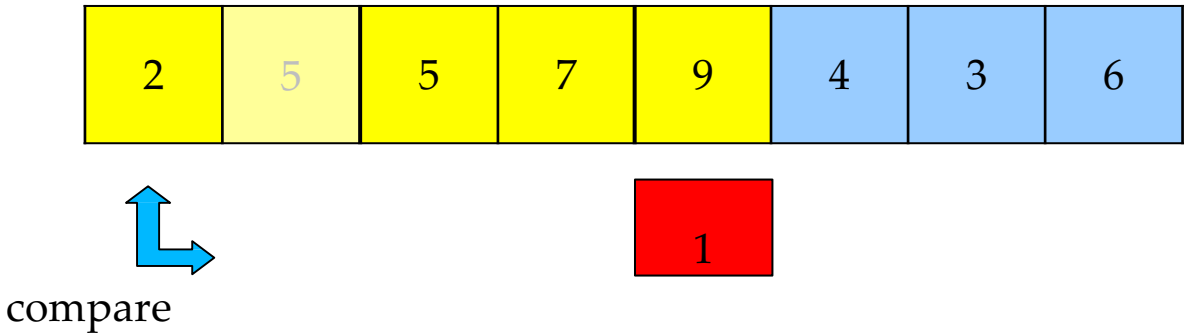
Insertion Sort



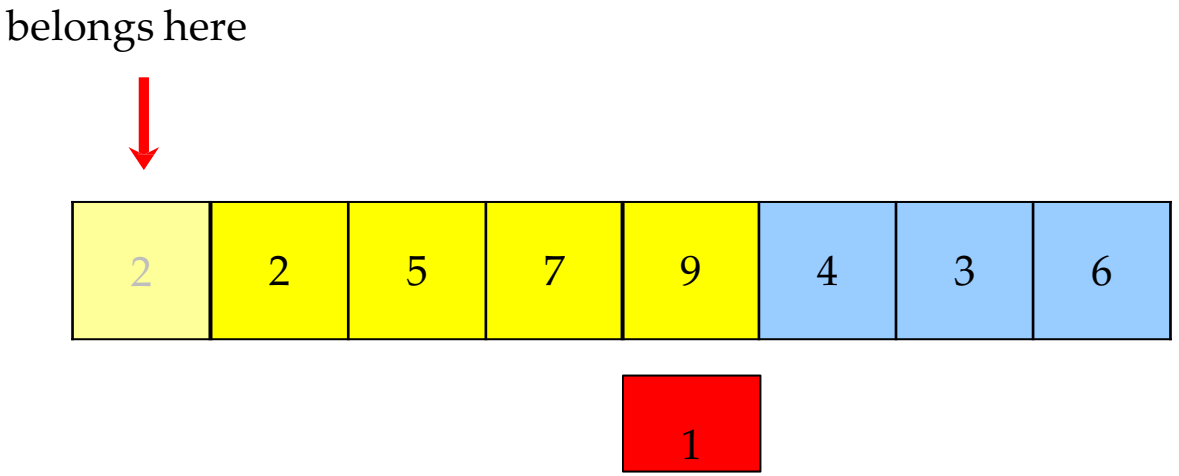
Insertion Sort



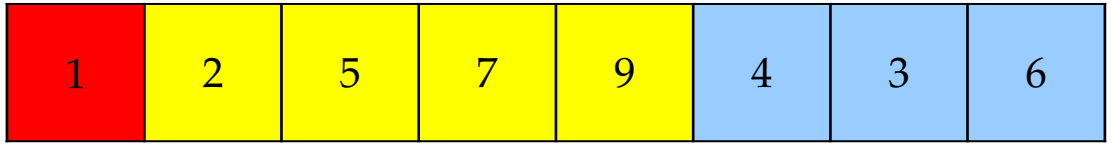
Insertion Sort



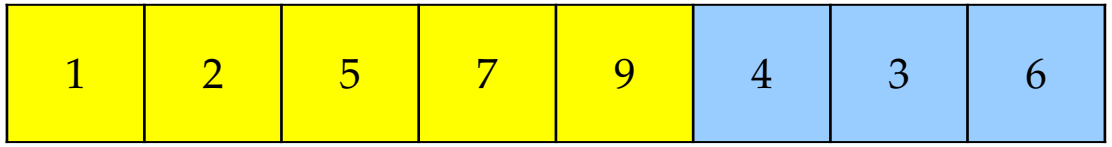
Insertion Sort



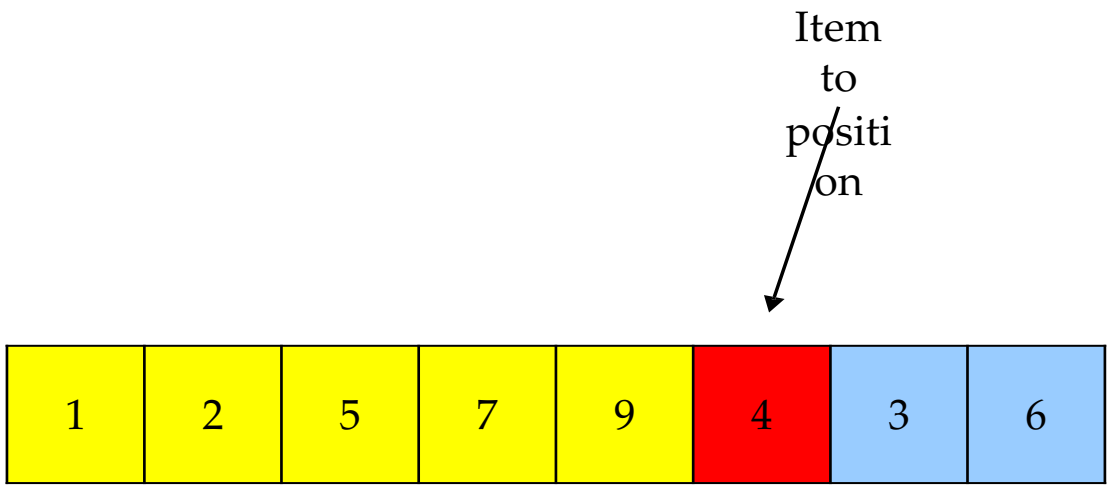
Insertion Sort



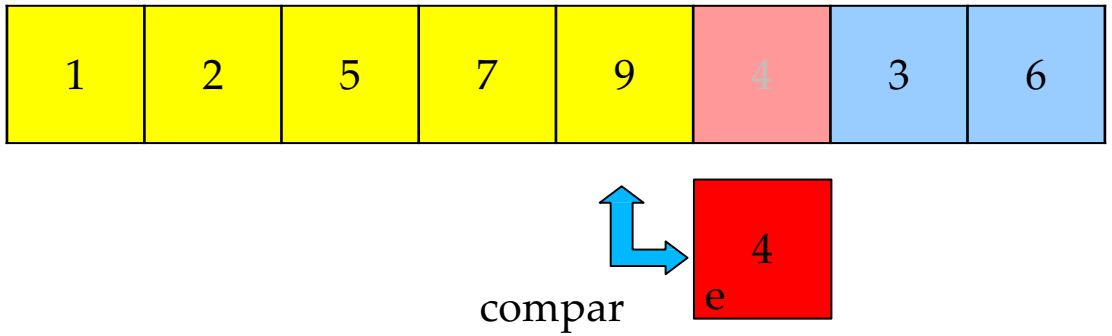
Insertion Sort



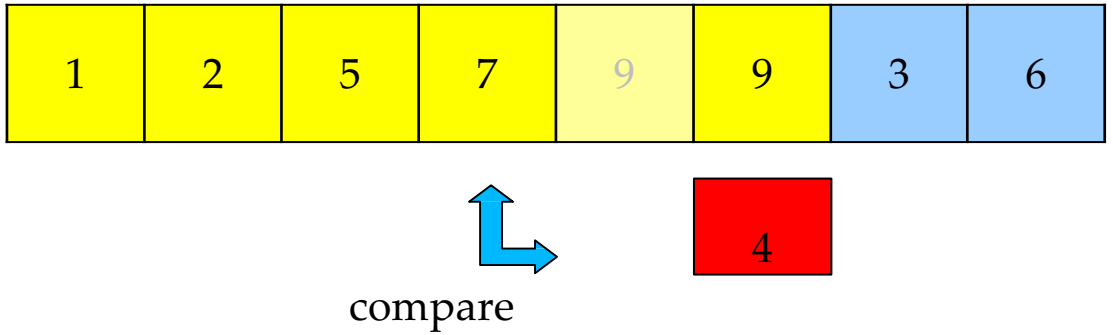
Insertion Sort



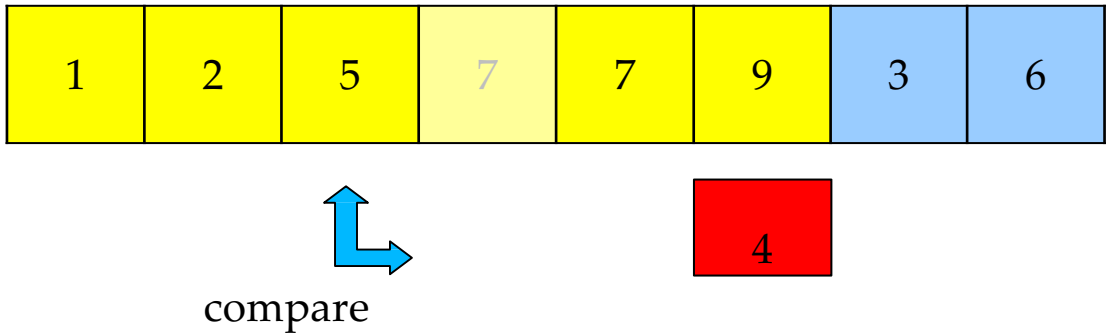
Insertion Sort



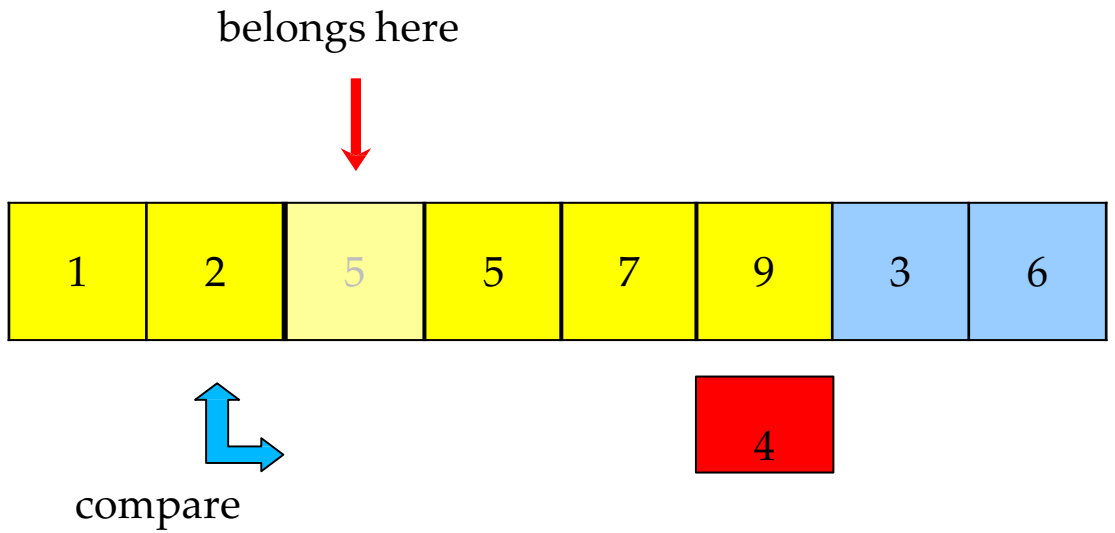
Insertion Sort



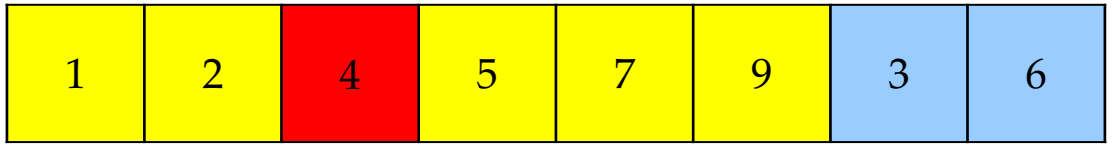
Insertion Sort



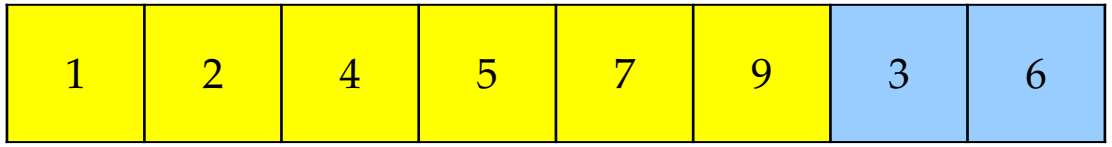
Insertion Sort



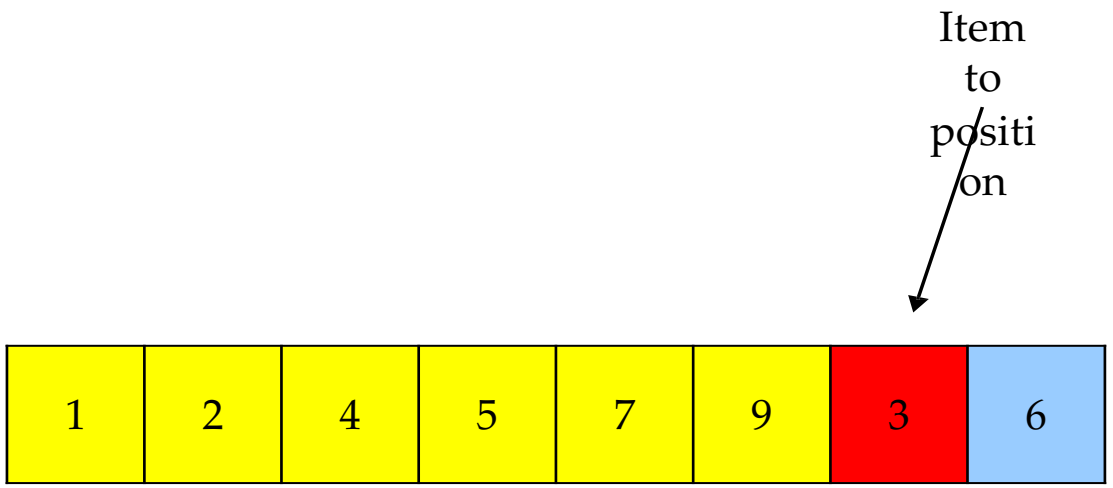
Insertion Sort



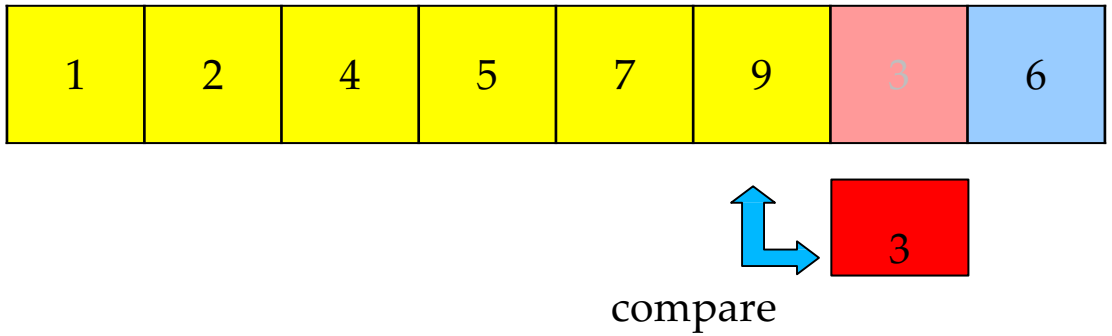
Insertion Sort



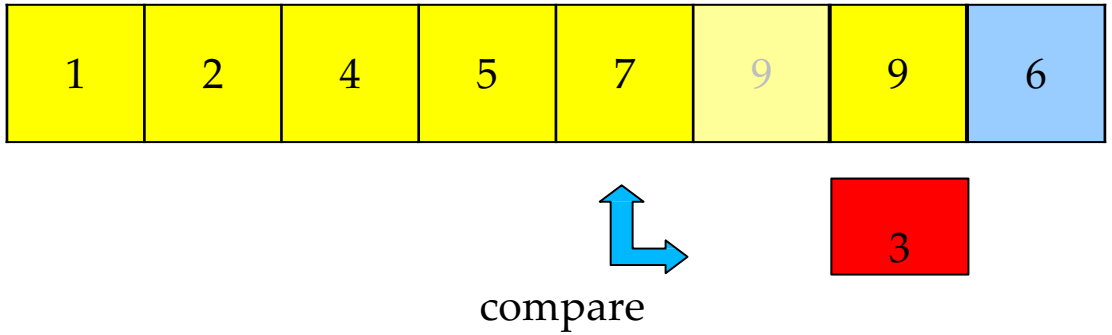
Insertion Sort



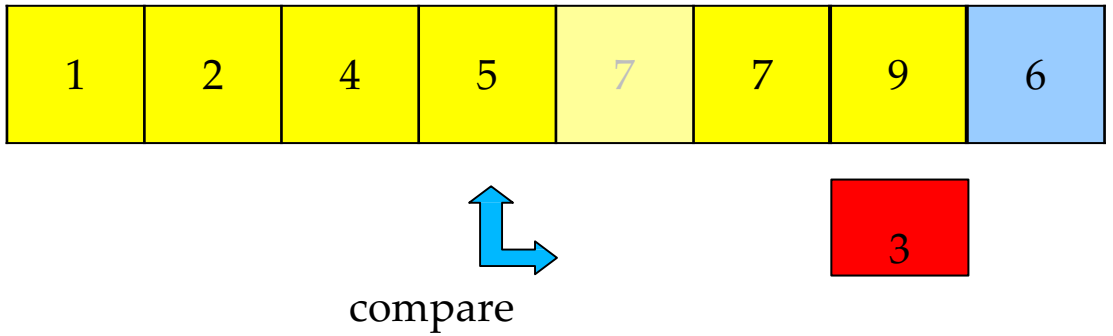
Insertion Sort



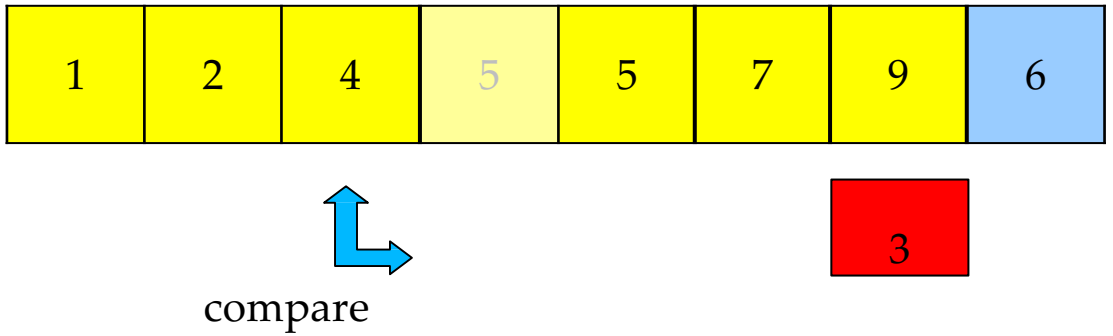
Insertion Sort



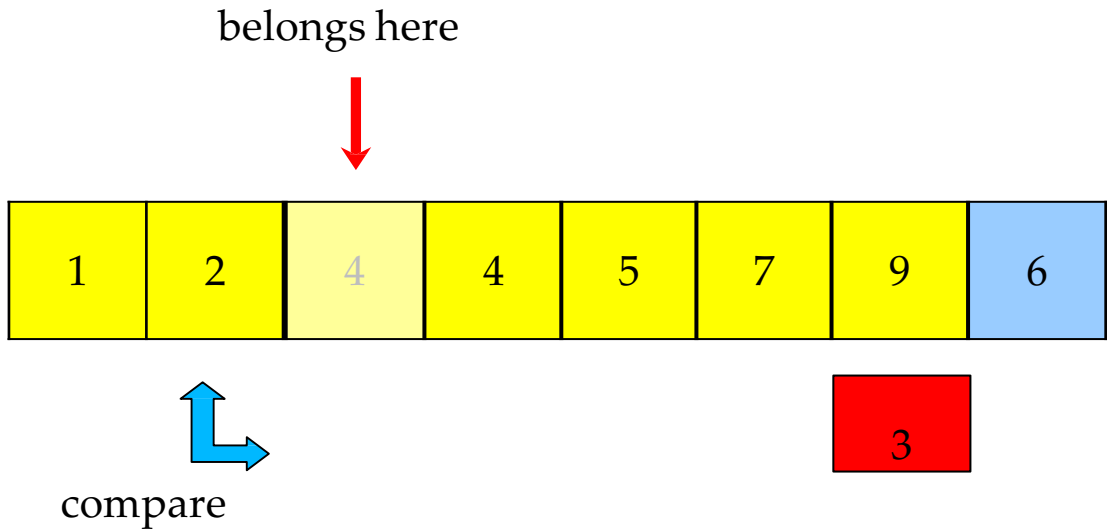
Insertion Sort



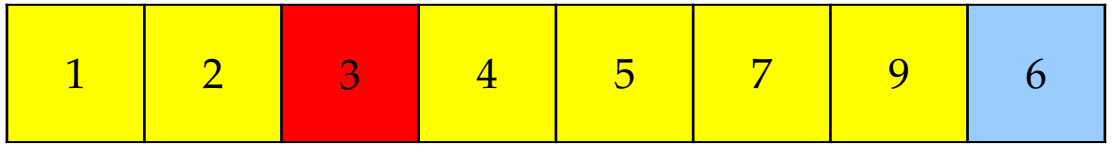
Insertion Sort



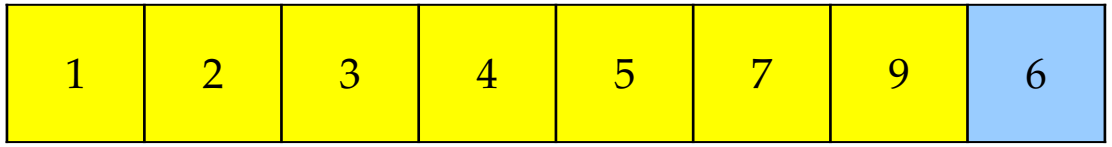
Insertion Sort



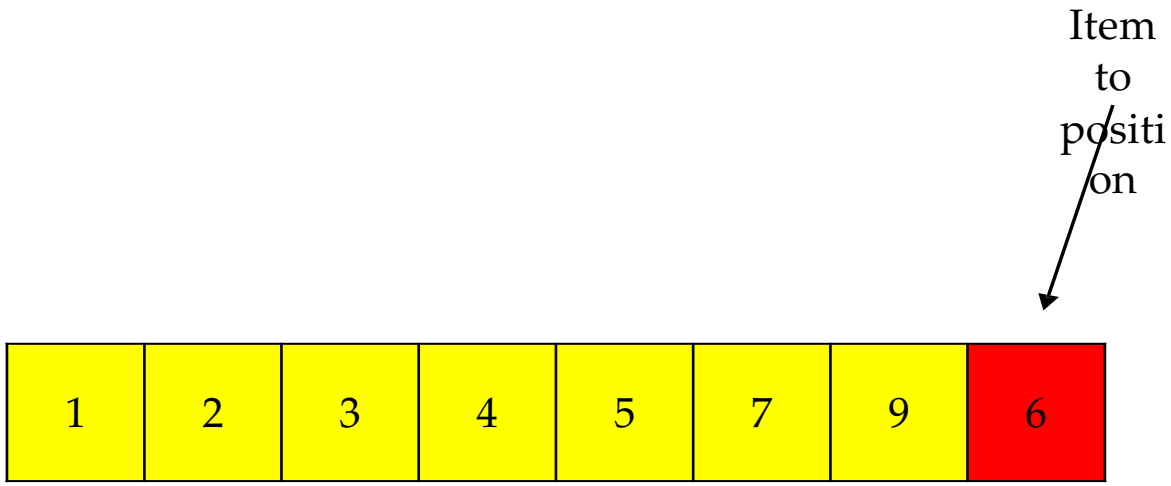
Insertion Sort



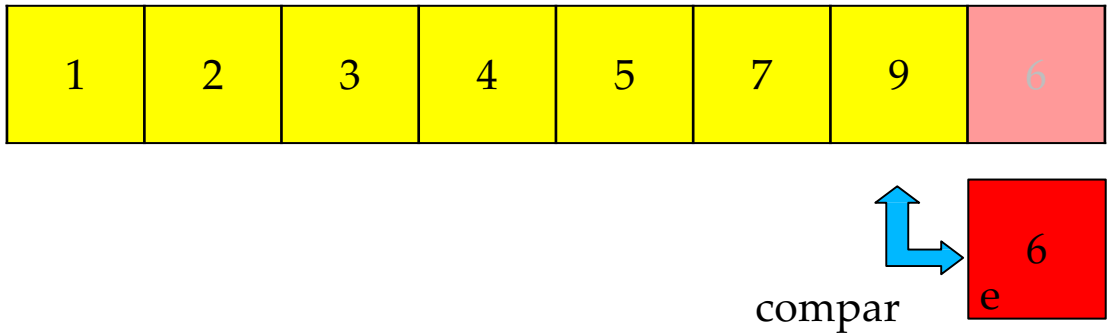
Insertion Sort



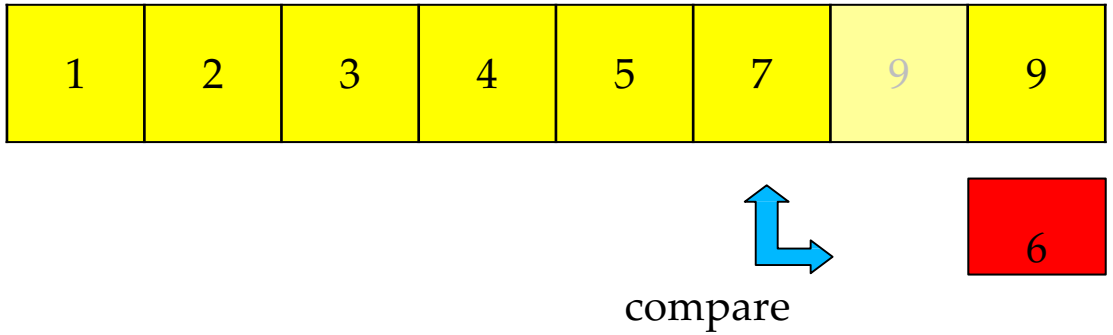
Insertion Sort



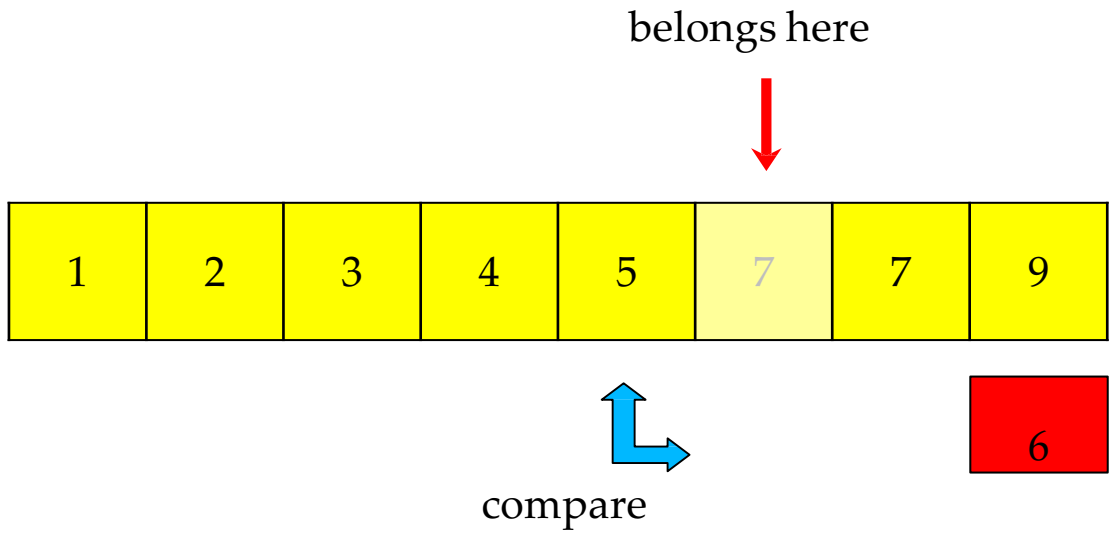
Insertion Sort



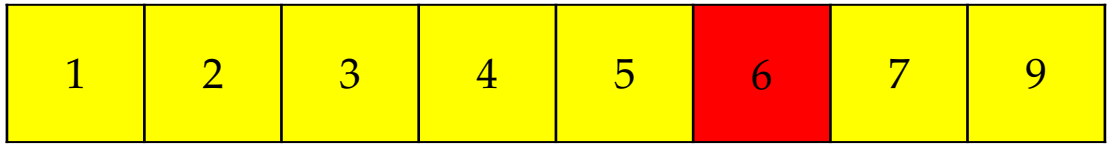
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

1	2	3	4	5	6	7	9

SORTED!

(Way of working)

Select – Compare – Shift - Insert

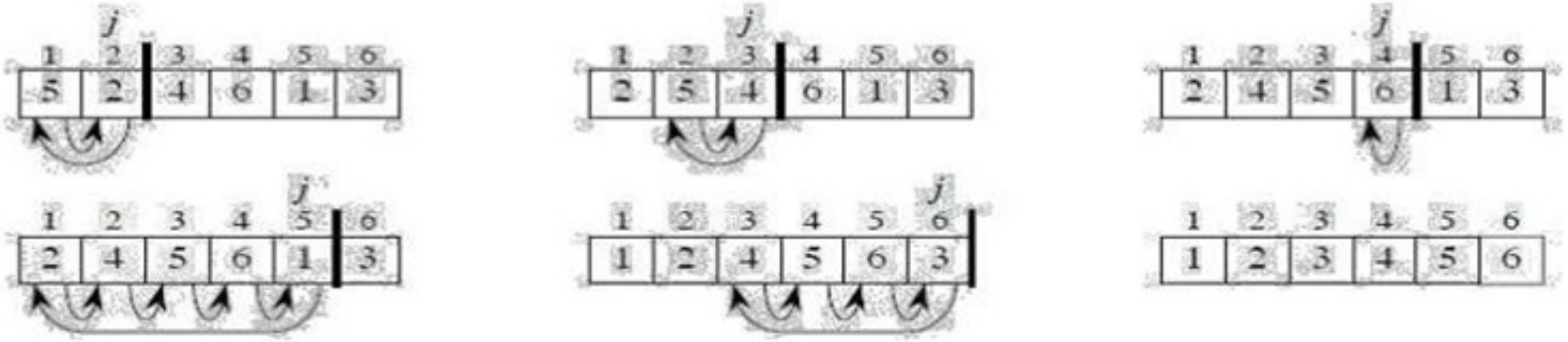


Figure 1: Insertion sort working model

Algorithm: Insertion Sort

```
INSERTION-SORT(A)
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1 .. j − 1].
4       i = j − 1
5       while i > 0 and A[i] > key
6           A[i + 1] = A[i]
7           i = i − 1
8       A[i + 1] = key
```

Loop Invariants and Correctness of Insertion Sort

- **Initialization:** Before the first loop starts, j=1. So, A[0] is an array of single element and so is trivially sorted.

- **Maintenance:** The outer for loop has its index moving like j=1,2,…,n-1 (if A has n elements). At the beginning of the for loop assume that the array is sorted from A[0..j-1]. The inner while loop of the jth iteration places A[j] at its correct position. Thus at the end of the jth iteration, the array is sorted from A[0..j]. Thus, the invariance is maintained. Then j becomes j+1.
  - ***Also, using the same inductive reasoning the elements are also the same as in the original array in the locations A[0..j].***

Loop Invariants and Correctness of Insertion Sort

- **Termination:** The for loop terminates when j=n, thus by the previous observations the array is sorted from A[0..n-1] and the elements are also the same as in the original array.

Thus, the algorithm indeed sorts and is thus correct!

Insertion Sort: Line and Operation Counts

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	$n-1$

Running time of the Insertion sort

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Let $T(n)$ = running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

The running time depends on the values of t_j. These vary according to the input.

Home assignment

1. Using Figure 1 as a model, illustrate the operation of INSERTION-SORT on the array A [31, 41, 59, 26, 41, 58].

2. Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

ALGORITHM DESIGN PARADIGMS

- **SLO: To understand the different algorithm design paradigms.**

- Specifies the pattern to write or design an algorithm

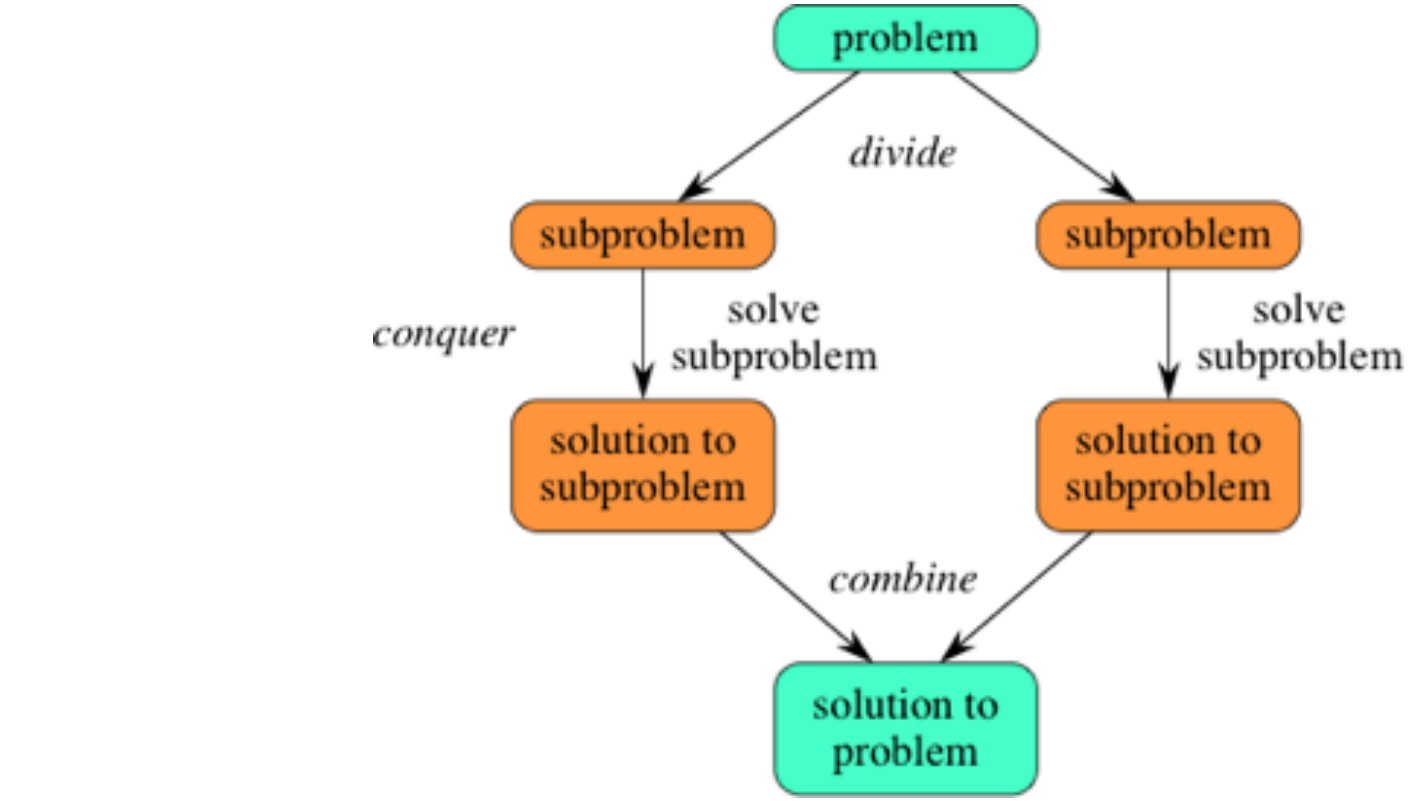
- Various algorithm paradigms are
  - Divide and Conquer
  - Dynamic programming
  - Backtracking
  - Greedy Approach
  - Branch and Bound

- Selection of the paradigms depends upon the problem to be addressed

DIVIDE AND CONQUER

- The Divide and Conquer Paradigm is an algorithm design paradigm which uses this simple process: It Divides the problem into smaller sub-parts until these sub-parts become simple enough to be solved, and then the sub parts are solved recursively, and then the solutions to these sub-parts can be combined to give a solution to the original problem.

- Examples :
  - Binary search
  - Merge sort
  - Quick sort

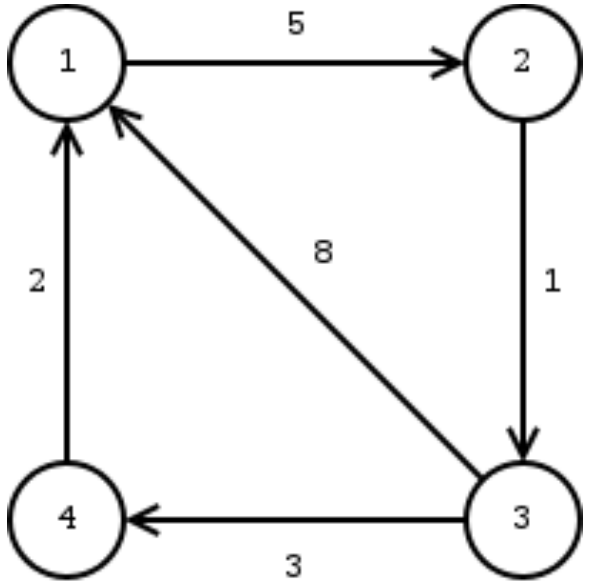


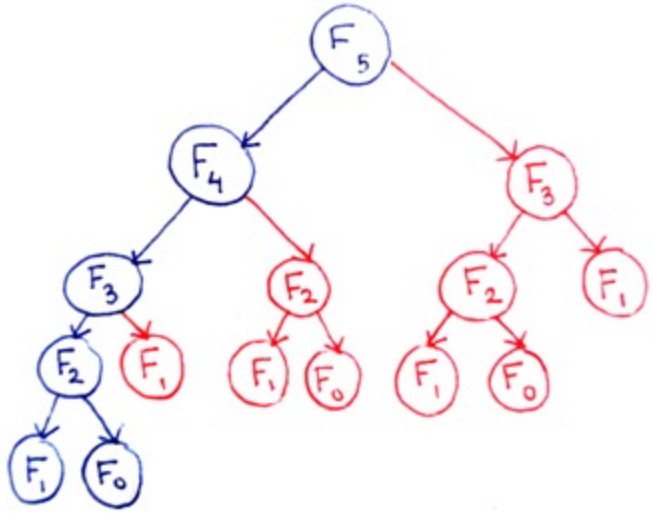
DYNAMIC PROGRAMMING

- Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub-problems and stores the results of sub-problems to avoid computing the same results again.

- It is an optimization technique

- Examples :
  - All pairs of shortest path
  - Fibonacci series



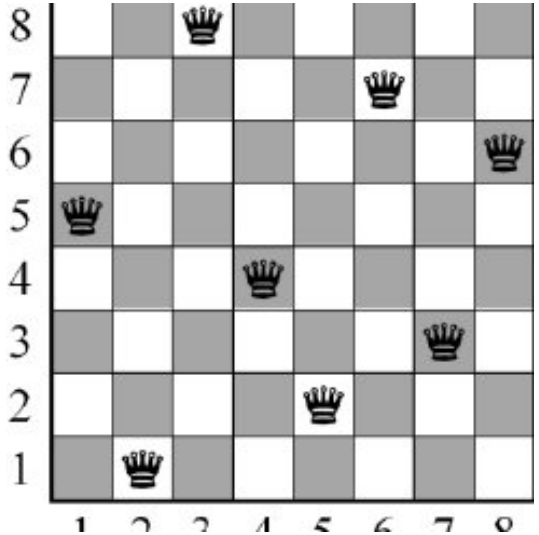


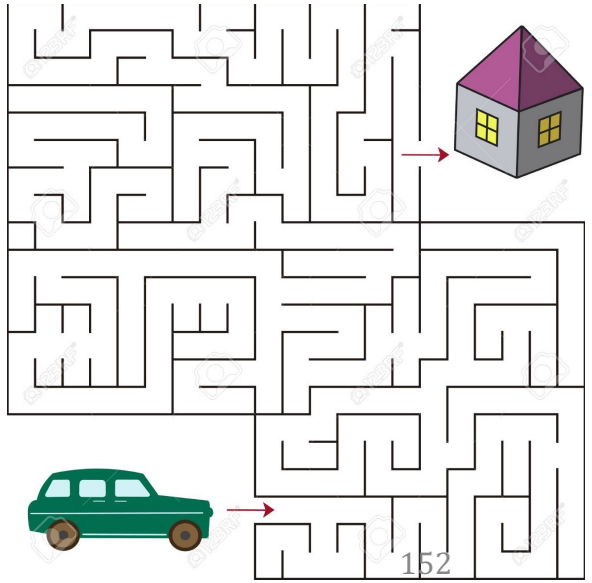
BACKTRACKING

- Backtracking is an algorithmic paradigm aimed at improving the time complexity of the exhaustive search technique if possible. Backtracking does not generate all possible solutions first and checks later. It tries to generate a solution and as soon as even one constraint fails, the solution is rejected and the next solution is tried.

- A backtracking algorithm tries to construct a solution incrementally, one small piece at a time. It's a systematic way of trying out different sequences of decisions until we find one that works.

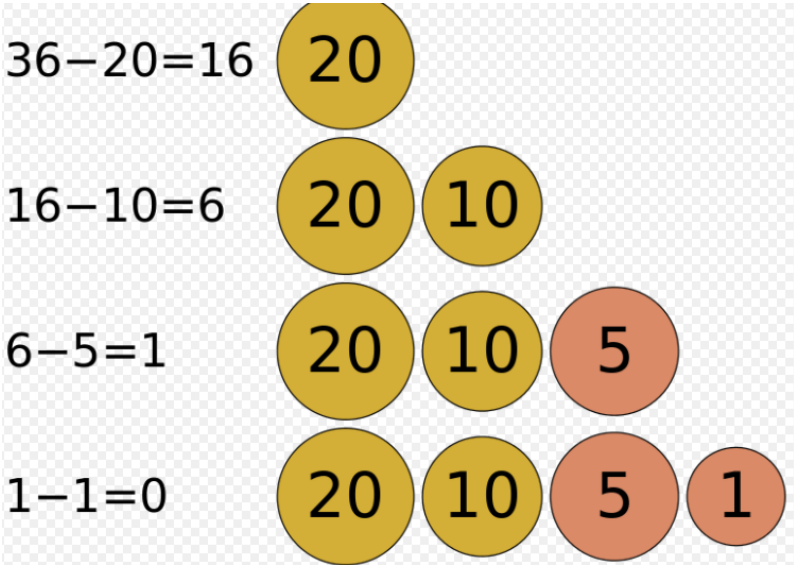
- Examples :
  - 8 Queens problem
  - Sum of subsets





GREEDY APPROACH

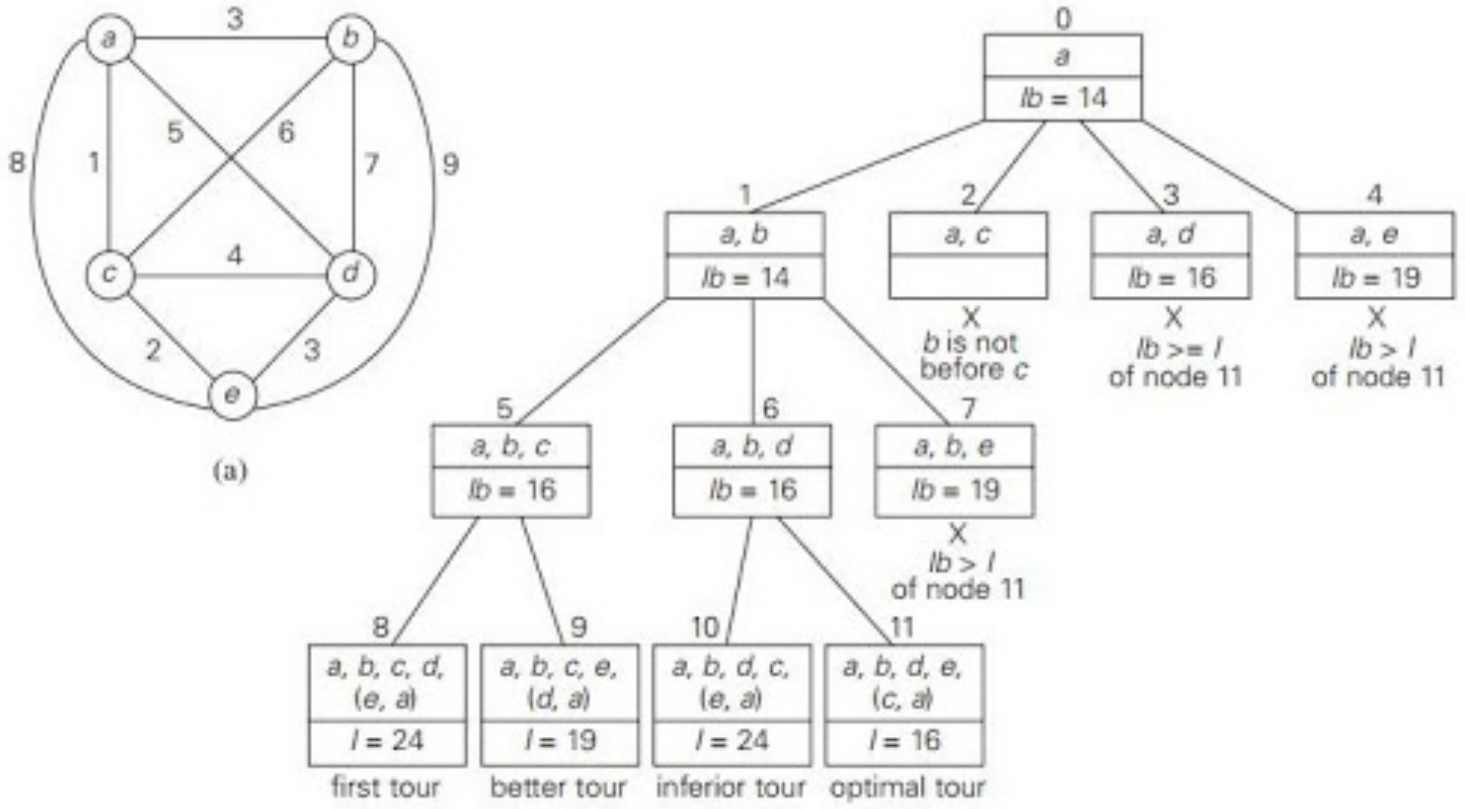
- Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.
- This approach is mainly used to solve optimization problems.
- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Examples
  - Coin exchange problem
  - Prim's
  - Kruskal's algorithm
  - Travelling salesman problem
  - Graph - map coloring



BRANCH AND BOUND

- Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems.
- These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- Examples :

  Travelling Salesman problem



HOME ASSIGNMENT

- Calculate the time complexity of binary search.

Asymptotic Analysis

Session Learning Outcome-SLO

- Estimate algorithmic complexity
- Learn approximation tool
- Specify the behaviour of algorithm

Asymptotic Analysis

- 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 Analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Dervive the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound.
  - Specify the behaviour of the algorithm when the input size increases
- Theory of approximation.
- Asymptote of a curve is a line that closely approximates a curve but does not touch the curve at any point of time.

Asymptotic notations

- Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.
- ***Asymptotic order*** is concerned with how the running time of an algorithm increases with the size of the input, if input increases from small value to large values

1. Big-Oh notation (O)
2. Big-Omega notation (Ω)
3. Theta notation (θ)
4. Little-oh notation (o)
5. Little-omega notation (ω)

Big-Oh Notation (O)

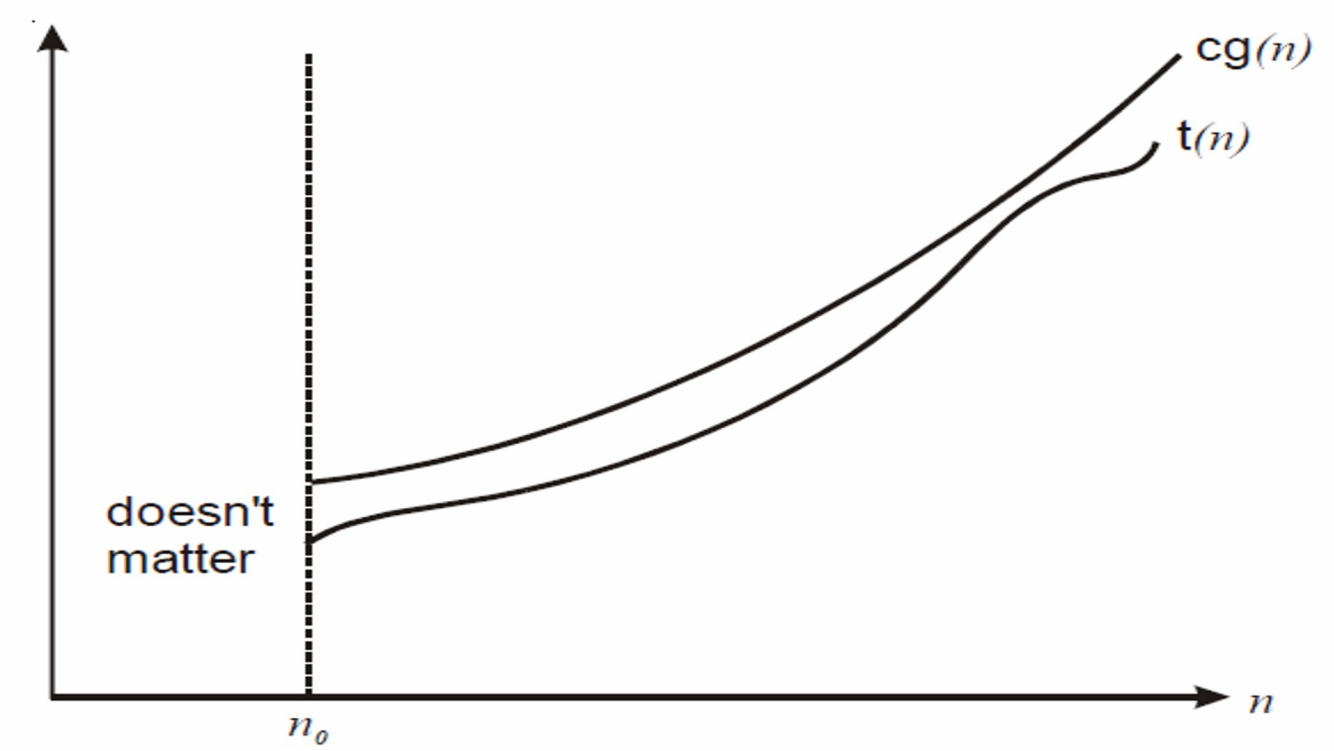
- Big-oh notation is used to define the worst-case running time of an algorithm and concerned with large values of n.

- ***Definition:*** A function $t(n)$ is said to be in $O(g(n))$, denoted as $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n. i.e., if there exist some positive constant c and some non-negative integer $n0$ such that

$$t(n) \leq cg(n) \text{ for all } n \geq n0$$

- **O(*g*(*n*)):** Class of functions $t(n)$ that grow no faster than $g(n)$.
- Big-oh puts asymptotic ***upper bound*** on a function.

Big-Oh Notation (O)



Big-Oh Notation (O)

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < 2^n < 3^n < \ldots. < n^n$

- Let $t(n) = 2n + 3$ upper bound

  $2n + 3 \leq$ _____??

  $2n + 3 \leq 5n \quad n \geq 1$

  here $c = 5$ and $g(n) = n$

  $t(n) = O(n)$

  $2n + 3 \leq 5n^2 \quad n \geq 1$

  here $c = 5$ and $g(n) = n^2$

  $t(n) = O(n^2)$

Big-Omega notation (Ω)

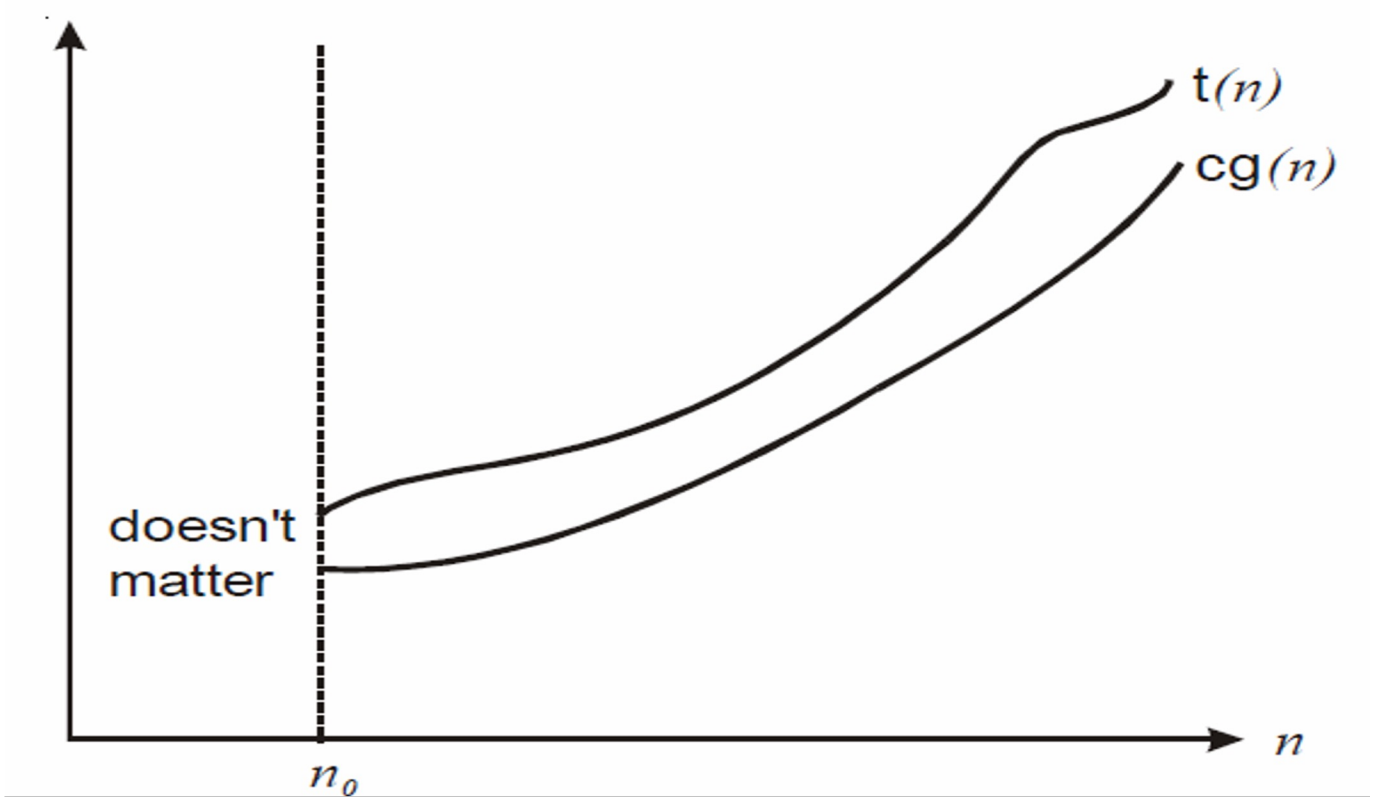
- This notation is used to describe the best case running time of algorithms and concerned with large values of n.

- ***Definition:*** A function $t(n)$ is said to be in $\Omega(g(n))$, denoted as $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n. i.e., there exist some positive constant c and some non-negative integer $n0$. Such that

$$t(n) \geq cg(n) \text{ for all } n \geq n0$$

- It represents the ***lower bound*** of the resources required to solve a problem.

Big-Omega notation (Ω)



Big-Omega notation (Ω)

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < 2^n < 3^n < \ldots. < n^n$

lower bound

- Let $t(n) = 2n + 3$

  $2n + 3 \geq$ _____??

  $2n + 3 \geq 1n \quad n \geq 1$

  here $c = 1$ and $g(n) = n$

  $t(n) = \Omega(n)$

  $2n + 3 \geq 1 \log n \quad n \geq 1$

  here $c = 1$ and $g(n) = \log n$

  $t(n) = \Omega(\log n)$

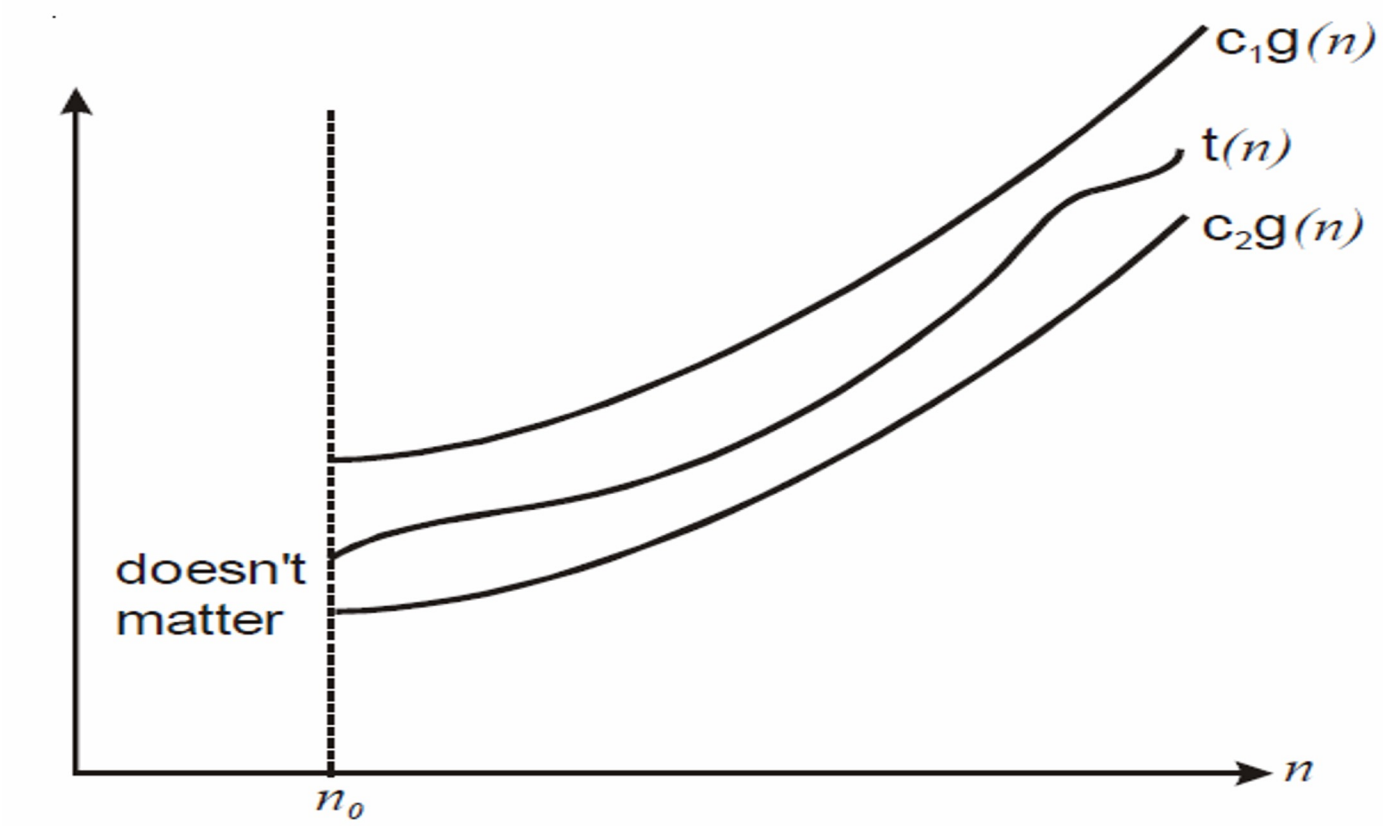
Theta notation (θ)

- ***Definition:*** A function $t(n)$ is said to be in $\theta(g(n))$, denoted $t(n) \in \theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n. i.e., if there exist some positive constant $c1$ and $c2$ and some non-negative integer $n0$ such that

$$c2g(n) \leq t(n) \leq c1g(n) \text{ for all } n > n0$$

$$\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Theta notation (θ)



Theta notation (θ)

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < 2^n < 3^n < \ldots. < n^n$

- Let $t(n) = 2n + 3$ average bound

  $\_c2.g(n)\_\_\_ \leq 2n + 3 \leq \_c1.\ g(n)\_\_\_\_$??

  $1n \leq 2n + 3 \leq 5n \quad n \geq 1$

  here $c1 = 5$, $c2 = 1$ and $g(n) = n$

  $t(n) = \theta(n)$

Little-oh notation (o)

- This notation is used to describe the worst case analysis of algorithms and concerned with small values of n.

- ***Definition :*** A function $t(n)$ is said to be in o($g(n)$), denoted $t(n) \in$ o($g(n)$), if there exist some positive constant c and some non-negative integer such that

$$t(n) \leq cg(n)$$

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = 0$$

Little-omega notation (ω)

- This notation is used to describe the best case analysis of algorithms and concerned with small values of *n*.
- The function t(*n*) = ω(g(*n*)) iff

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty \qquad \text{(or)} \qquad \lim_{n\to\infty}\frac{g(n)}{f(n)}=0$$

Asymptotic Analysis of Insertion sort

- Time Complexity:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

- Best Case: the best case occurs if the array is already sorted, tj=1 for j=2,3…n.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- Linear running time: O(n)

Asymptotic Analysis of Insertion sort

- Worst case : If the array is in reverse sorted order

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

and

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

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

- Quadratic Running time. O(n2)

Properties of O, Ω and θ

General property:

If t(n) is O(g(n)) then a * t(n) is O(g(n)). Similar for Ω and θ

Transitive Property :

If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$; that is O is transitive. Also Ω, θ, o and ω are transitive.

Reflexive Property

If f(n) is given then f(n) is O(f(n))

Symmetric Property

If f(n) is θ(g(n)) then g(n) is θ(f(n))

Transpose Property

If f(n) = O(g(n)) then g(n) is Ω(f(n))

Properties of Asymptotic Notations:-

① General properties.

if $f(n)$ is $O(g(n))$ then $a * f(n)$ is $O(g(n))$

eg: $f(n) = 2n^2+5$ is $O(n^2)$

then $= 7 \cdot f(n) = 7(2n^2+5)$

$= 14n^2 + 35$ is $O(n^2)$

Also true for Ω & Θ. i.e ∀ all three notation

$f(n) = \Omega(g(n)) \rightarrow a * f(n) \rightarrow \Omega(g$

② Reflexive property:-

if $f(n)$ is given then $f(n)$ is $O(f(n))$

eg:- $f(n) = n^2 \Rightarrow O(n^2)$

↑ $f(n)$

③ Transitive property: if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$

then $f(n) = O(h(n))$

eg:- $f(n) = n$ (n is upper bound of $g(n) = n^2$) ; $h(n) = n^3$ upper bound.

n is $O(n^2)$ & n^2 is $O(n^3)$

then n is $O(n^3)$.

∀ three notations O, Ω & Θ

④ Symmetric property :- (True for only 'Θ' notation)

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$

eg:- $f(n) = n^2 \rightarrow g(n^2) = n^2$

$f(n) = \Theta(n^2)$

$g(n) = \Theta(n^2)$

When both the fns/. are same, that they are symmetric.

⑤ Transpose symmetric :- (O & Ω) only

If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$
(UP) (LB)

eg:- $f(n) = n$ $g(n) = n^2$

then n is $O(n^2)$ and

n^2 is $\Omega(n)$

If one fny. forms an upper bound for other fny. then the other fny. will form a lower bound for the other fny.

If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

$g(n) \leq f(n) \leq g(n)$

$\boxed{f(n) = \Theta(g(n))}$ //.

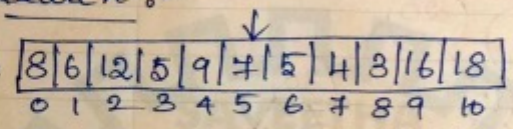
When same fny. is both upper & lower bound =

Cases - Types of analysis

Best case, and Worst Case & Average case analysis.

1. Linear Search
2. Binear Search.

Linear Search :-



Key = 7 ✓ Successful index @ 5

Key = 20 ✓ Unsuccessful index

Searching key present.

Best Case:- @ Very begining index.

Best Case Time:- Constant time — 1 $O(1)$

$\therefore B(n) = O(1)$ //.

Searching a Key

Worst Case:- @ the last index

Worst Case Time:- n

$w(n) = n$

$\therefore \boxed{w(n) = O(n)}$ //.

$B(n) = O(1)$
$w(n) = O(n)$
$A(n) = \frac{(n+1)}{2}$

Average Case:- $\frac{\text{All possible case time}}{\text{no. of cases}}$

$$\text{Avg. time} = \frac{1+2+3+\cdots+n}{n}$$

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

$\therefore A(n) = \frac{n+1}{2}$

Asymptotic Notation and its intuition

Notation	What it means	In terms of limit	Representation	Mathematically equivalent to
Big oh (O)	Growth of t(n) is ≤ the growth of g(n)	$\lim_{n\to\infty}\frac{t(n)}{g(n)} = c. c \geq 0$	t(n) = O(g(n))	t(n) ≤ g(n)
Big omega (Ω)	Growth of t(n) is ≥ the growth of g(n)	$\lim_{n\to\infty}\frac{t(n)}{g(n)} \neq 0$	t(n) = Ω(g(n))	t(n) ≥ g(n)
Theta notation (θ)	Growth of t(n) is ≈ the growth of g(n)	$\lim_{n\to\infty}\frac{t(n)}{g(n)} = c, c > 0$	t(n) = θ(g(n))	t(n) ≈ g(n)
Little oh (o)	Growth of t(n) is < the growth of g(n)	$\lim_{n\to\infty}\frac{t(n)}{g(n)} = 0$	t(n) = o(g(n))	t(n) < g(n)
Little omega (ω)	Growth of t(n) is > the growth of g(n)	$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \infty$	t(n) = ω(g(n))	t(n) > (g(n)

Activity

- Find the upper bound, lower bound and tight bound range for the following functions
  - $2n + 5$
  - $3n + 2$
  - $3n + 3$
  - $n^2 \log n$
  - $10\ n^2 + 4\ n + 2$
  - $20\ n^2 + 80\ n + 10$
  - $n!$
  - $\log n!$

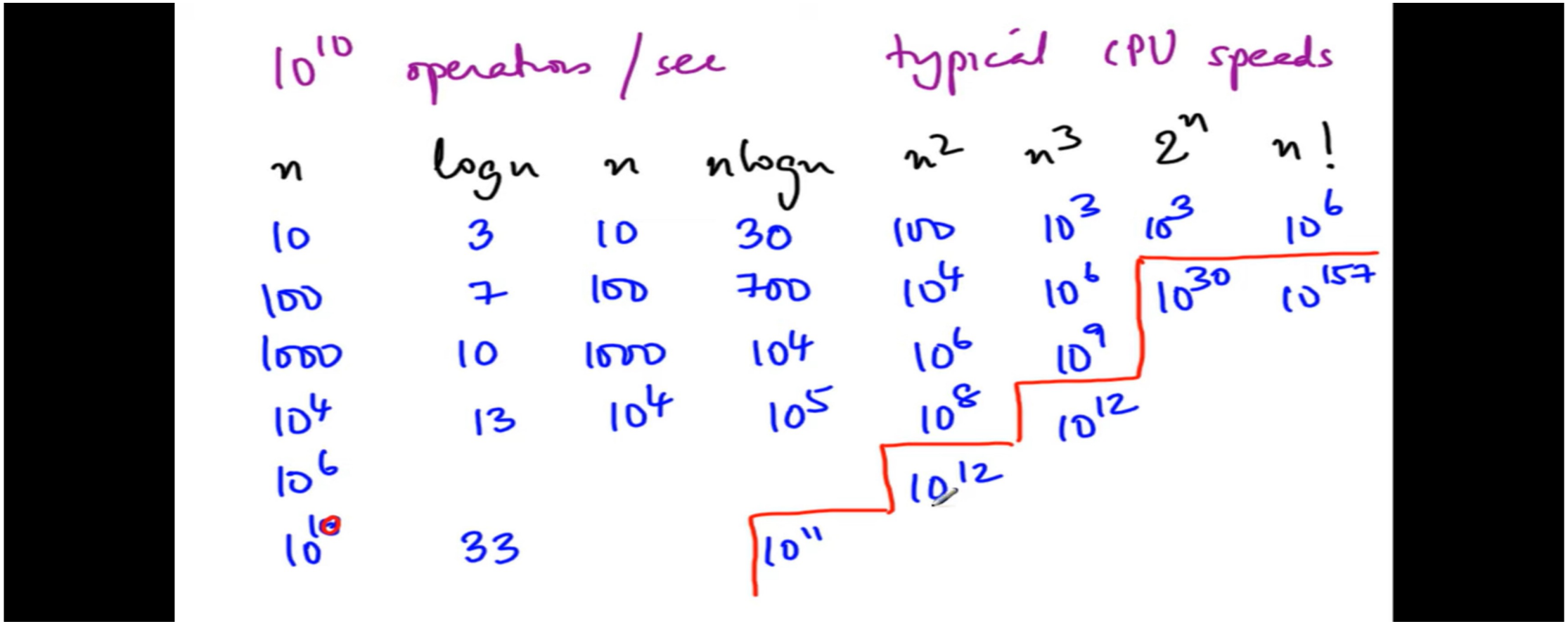
Orders of Growth

- Measuring the performance of an algorithm in relation with the input size, n is called order of growth.
- Order of growth rates are:
  - Constant
  - Logarithmic
  - Quadratic and
  - Exponential

Order of growth of functions

Time complexity		Example
O(1)	*constant*	Adding to the front of a linked list
O(log *N*)	*log*	Finding an entry in a sorted array
O(*N*)	*linear*	Finding an entry in an unsorted array
O(*N* log *N*)	*n-log-n*	Sorting n items by 'divide-and-conquer'
O(N^2)	*quadratic*	Shortest path between two nodes in a graph
O(N^3)	*cubic*	Simultaneous linear equations
O(2^N)	*exponential*	The Towers of Hanoi problem

Order of growth of functions



Summary

- Asymptotic analysis estimate an algorithmic complexity
- Based on theory of approximation.
- Effective in specifying the behaviour of algorithm when the input size increases
- Big – Oh notation – upper bound
- Big – Omega notation – lower bound
- Little – oh notation – tight bound

Mathematical Analysis

Induction

- Induction is a method for proving universally quantified propositions—statements about all elements of a (usually infinite) set.
- Induction is also the single most useful tool for reasoning about, developing, and analyzing algorithms.
- Steps:
- 1. Basis Step
- 2. Inductive Step

Induction- Example

- Use induction to prove each of the following for all natural numbers n.

$$4 + 9+14+19+\ldots+ (5n-1)=n/2(3+5n)$$

a) Basis Step: n=1

5(1)-1=1/2(3+5)

4=4 (true)

Induction

- b) Inductive step: Assume true for n=k, show that it is true for n=k+1
- Assume: 4+9+14+19+…+(5k-1)=k/2(3+5k)
- Show: 4+9+14+19+..+(5k-1)+(5(k+1)-1)=k+1/2(3+5(k+1))
  - k/2(3+5k)+(5(k+1)-1)=k+1/2(3+5(k+1))
  - k/2(3+5k)+(5k+4)=k+1/2(3+5k+5))
  - $3k/2+5k^2 + 5k+4 = k+1/2(8+5k)$
  - $13k/2+5k^{2+} +4=8(k+1)/2+5k(k+1)/2$
  - $13k/2+5k^2+4 = 4k+4+5k^2 +5k/2$
  - $13k/2+5k^2+4 = 13k/2+5k^2+4$ (true)

Recurrence Relation

Recurrence

- Any problem can be solved either by writing **recursive algorithm** or by writing **non-recursive algorithm**.

- A recursive algorithm is one which makes a recursive call to itself with smaller inputs. We often use a **recurrence relation to describe the running time of a recursive algorithm.**

- Recurrence relations often arise in calculating the time and space complexity of algorithms

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

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

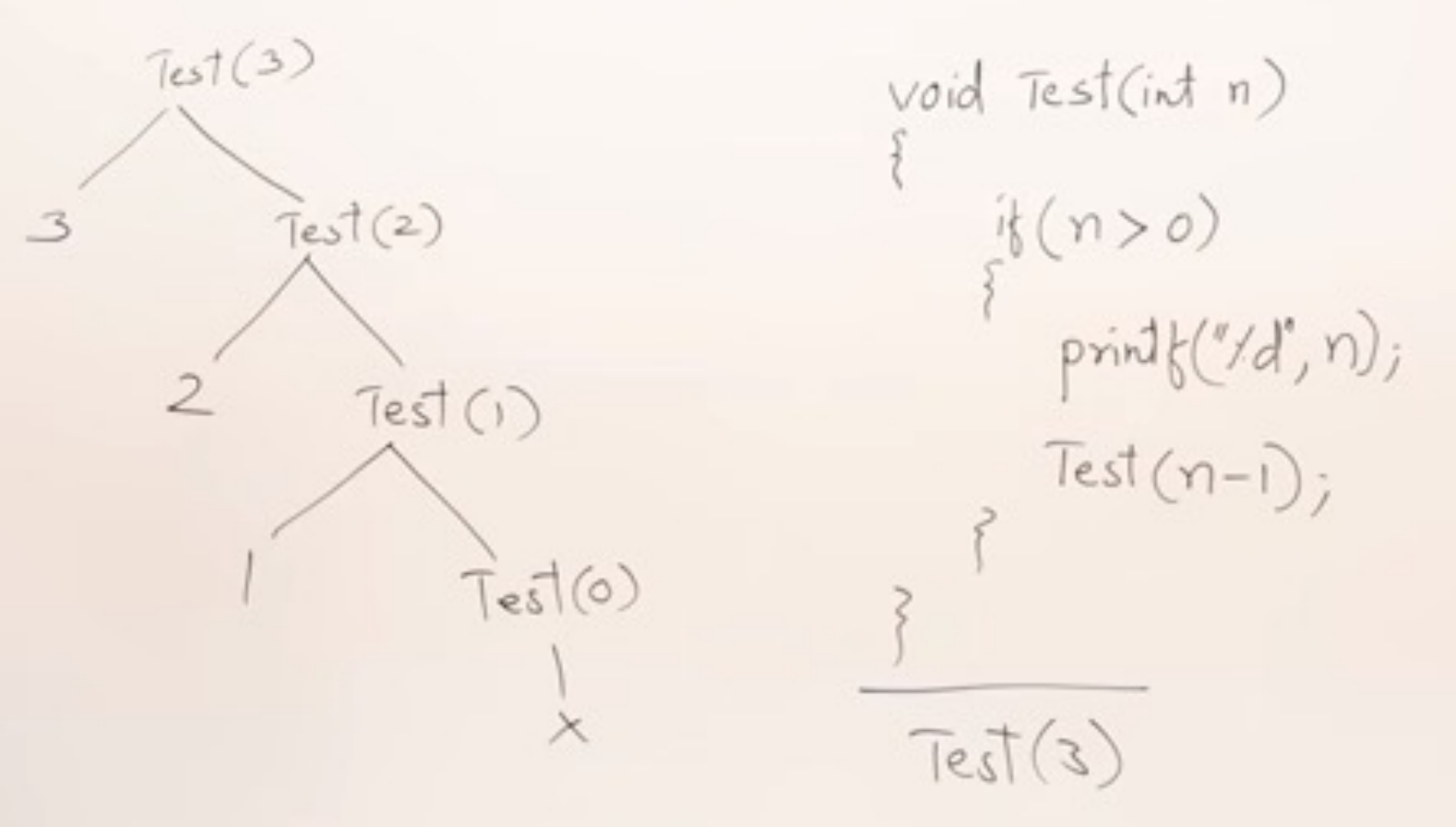
$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

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

Recursive Function and Tracing tree



Running time: Test(3)
3 (printf), 4 (recursive call)
n (printf), n+1 (recursive call)

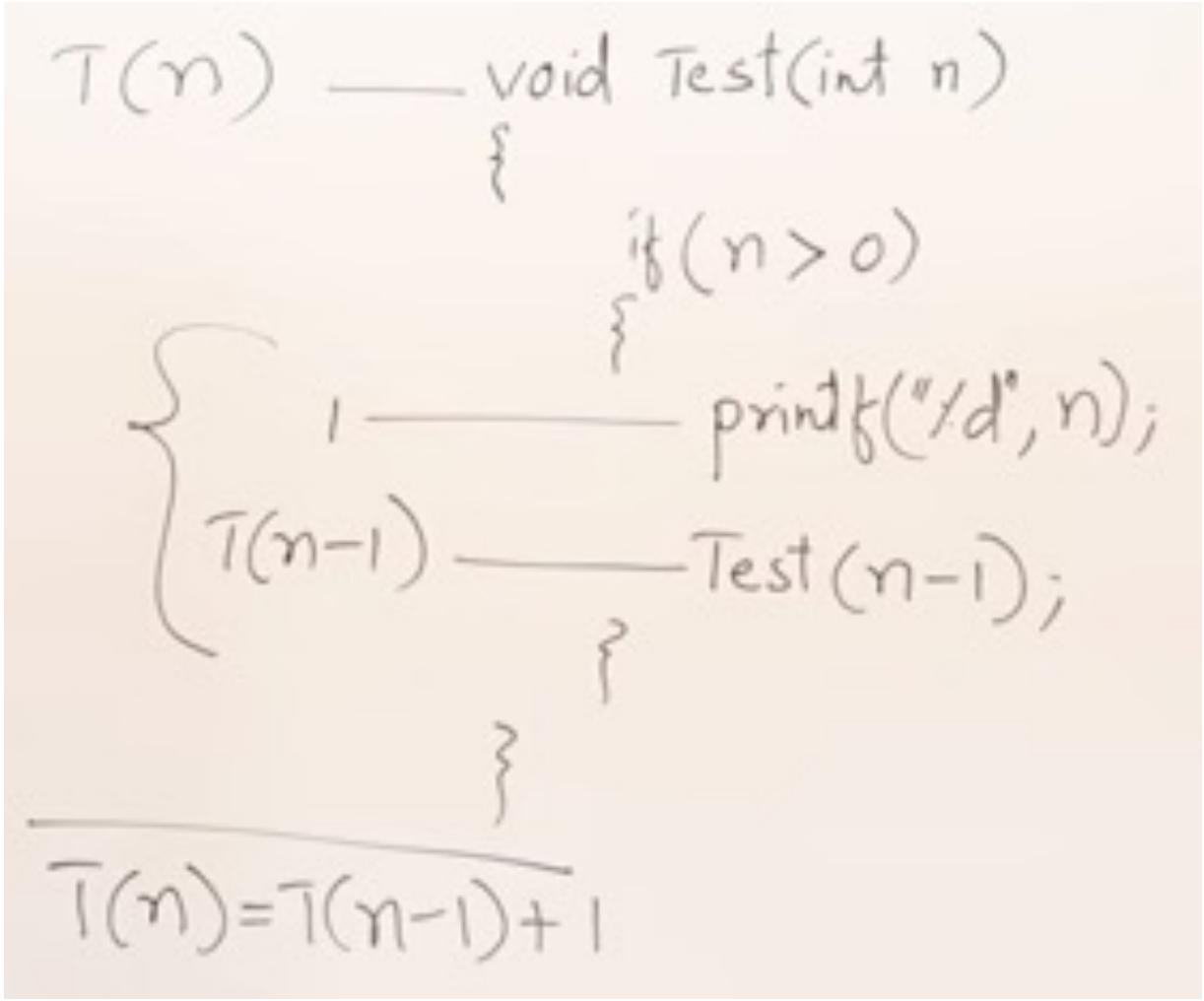
T(n)
n times print executes and n+1 function call occurs.

➔The amount of work done depends on the number of call so the time complexity is n+1

➔f(n)=n+1

➔Time complexity in notation= O(n)
➔ Big O(n), Omega(n),Theta(n)

Recurrence Relation – Example 1



$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

For this time value cannot be zero, so having some constant value

$T(n) = T(n-1)+1 \longrightarrow$ eq①

then what is $T(n-1)$

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

$= T(n-2)+1$

Substitute $T(n-1)$ in eq①.

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

$= T(n-2)+2 \longrightarrow$ eq②

then what is $T(n-2)$

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

$= T(n-3)+1$

Substitute $T(n-2)$ in eq②.

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

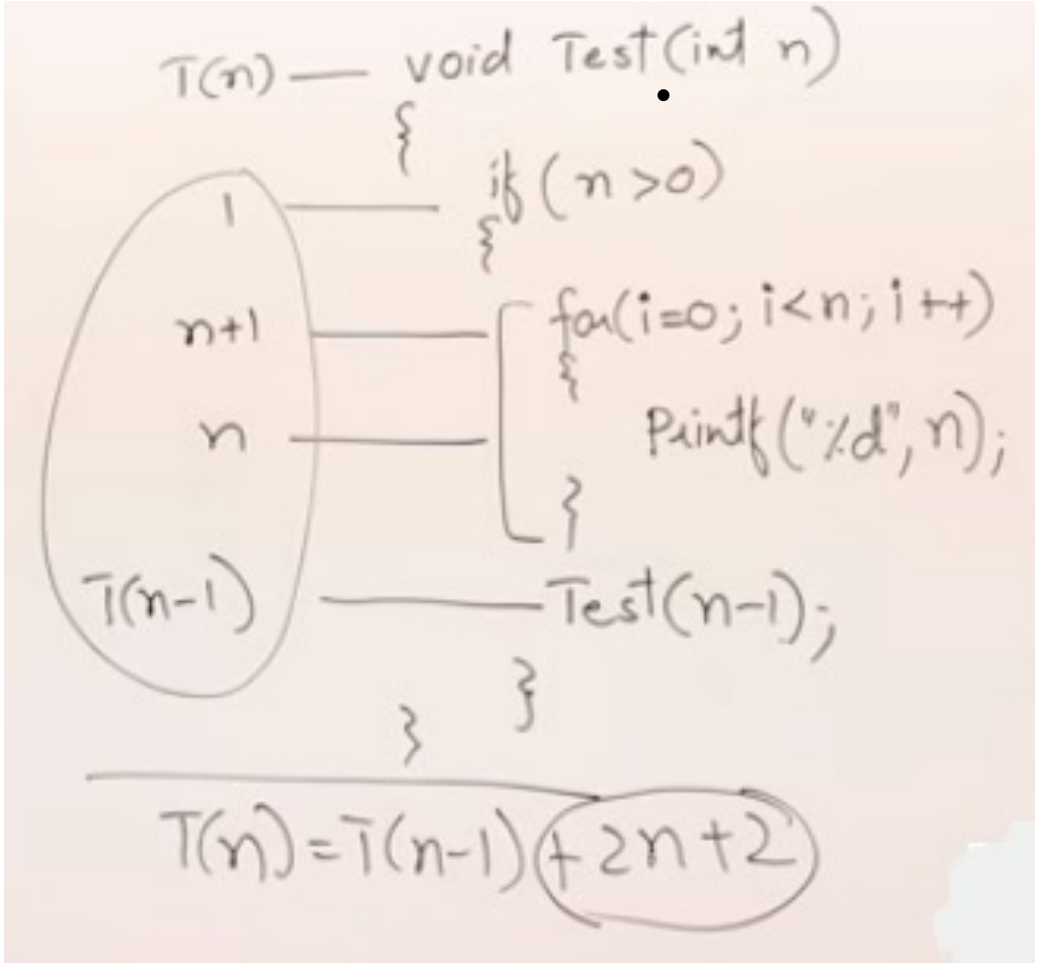
$= T[n-3]+3 \longrightarrow$ eq③

then like this

⋮

$T(n) = T(n-k)+k.$

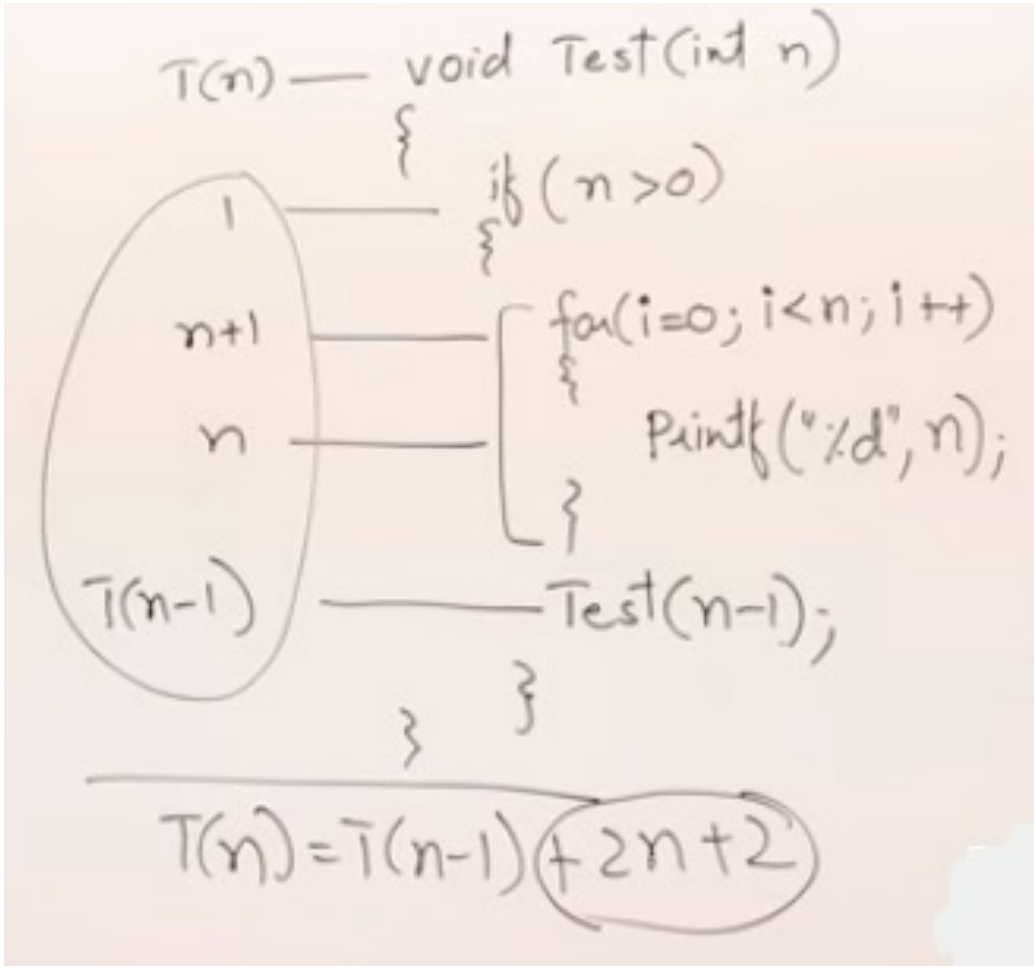
Recurrence Relation – Example 2



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

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

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + n & n>0 \end{cases}$$



TREE METHOD

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Time

n ← T(n)

n-1 ← loop takes → n T(n-1)

n-2 ← n-1 T(n-2)

n-3 ← n-2 T(n-3)

n-3 T(n-4)

⋮

Reduce by 1 at a time & reach Zero.

⋮

T(2)

2 T(1)

1 T(0)

1

0 ← X

$\Rightarrow 0+1+2+\cdots+n-1+n$

$= \frac{n(n+1)}{2} \Rightarrow T(n) = \frac{n(n+1)}{2}$

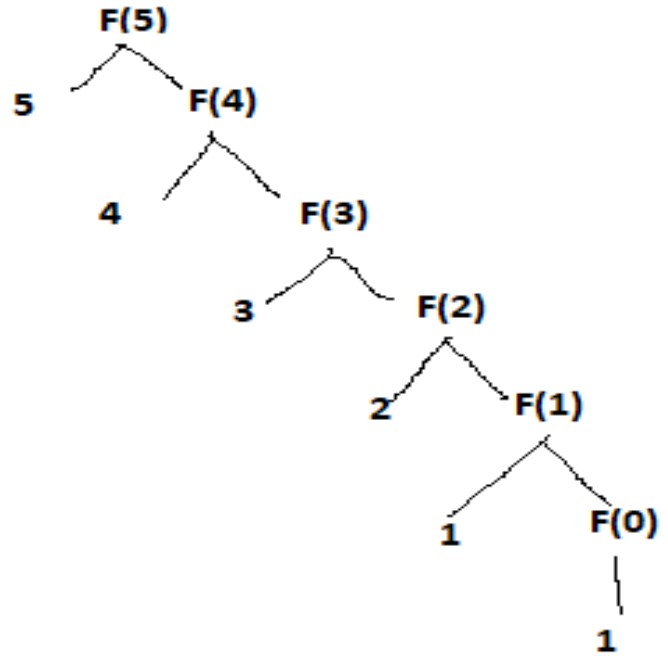
$\Theta(n^2)$

Recurrence Relation – Example 3

```
int factorial(unsigned int n)
{
   if (n == 0)   ------1
      return 1;  --------1
   return n * factorial(n - 1);  -------1 + T(n-1)
}
```

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

T(n)= O(n)



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

Where T(n-1) is the number of multiplications required to compute the F(n-1)
1 is one multiplication to multiply the F(n-1) by n.

Recurrence Relation is

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

Home Assignment

- Find the recurrence relation of the algorithm for

1. Fibonacci series.
2. Linear search
3. Binary Search
4. Insertion sort

Solution of Recurrence Relations

There are four methods for solving Recurrence:

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Method

Substitution Method

- The Substitution Method Consists of two main steps:
  1. Guess the Solution.
  2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

  The substitution method can be used to establish either upper or lower bounds on a recurrence.

Examples:

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

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

$T(n) = 2T(\lfloor n/2 \rfloor) + n$

Substitution Method

Forward Substitution

- Take the Recurrence equation and initial condition.
- Put the initial condition in equation and look for the pattern
- Guess the pattern
- Prove that the guess pattern is correct using induction.

Substitution Method

Forward Substitution

1. Take the equation and initial condition

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

   $T(1)=1$

2. Look for the pattern

   $T(1) = 1$

   $T(2) = T(2-1)+2 = T(1) +2=1+2=3$

   $T(3) = T(3-1)+3 = T(2) +3=3+3=6$

   $T(4)= T(4-1) + 4= T(3)+4= 6+4= 10$

   $T(5)= T(5-1) + 5= T(4)+5= 10+5= 15$

   …….

   $T(n)=1+3+6+10+\ldots\ldots n(n+1)/2 \rightarrow n(n+1)/2$ (summation of n numbers)

   $= n^2/2 + n/2 \rightarrow O(n^2)$

Forward Substitution

$T(1) = 1$

$T(2) = T(n-1) + n$
$= T(2-1) + 2$
$= T(1) + 2$
$= 1 + 2$
$= 3$

$T(3) = T(3-1) + 3$
$= T(2) + 3$
$= 3 + 3$
$= 6$

$T(4) = T(4-1) + 4$
$= T(3) + 4$
$= 6 + 4$
$= 10$

$T(5) = T(5-1) + 5$
$= T(4) + 5$
$= 10 + 5$
$= 15.$

$T(n) = 1 + 3 + 6 + 10 + 15 \cdots n\frac{(n+1)}{2}$

[Summation of n nos/.]

$= \frac{n^2+n}{2} = \frac{n^2}{2} + \frac{n}{2}$

$\Rightarrow O(n^2)$

Substitution Method

Forward Substitution

3. Guess the pattern as the above step.

$$T(n) = n(n+1)/2$$

4. Prove $T(n) = n(n+1)/2$ using induction

As per Induction for $T(n) = n (n + 1) / 2$ we can write

1) Let's prove that $T(1) = 1$

$T(n) = n (n + 1) / 2$

$T(1) = 1 (1+1) / 2$

$= 1(2)/2$

$= 1$

$T(1) = 1$ is proved.

2) Assume $T(n-1)$ is true, means

$T(n-1) = (n-1)(n-1+1) / 2$ is true

3) Now we will prove that $T(n)$ is also true

Proof:

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

$= (n-1)(n-1+1) / 2 + n$ (rule 2)

$= (n-1) (n) / 2 + n$

$= n (n-1)/ 2 + n$

$= n^2 / 2 - n / 2 + n$

$= n^2 / 2 + n / 2$

$T(n) = n (n +1) / 2$

Hence, it is proved that $T(n)$ is true.

So, as per the Induction, $T(n) = n (n+1) / 2$ is true and it is our solution

Also complexity is $O(n^2)$

Substitution Method

Forward Substitution

- This method make use of an initial condition in the initial term and value for the next term is generated.
- This process is continued until some formula is guessed.

1. T(n)=T(n-1) **+ 1**
2. T(1)=T(0)+1=1+1=2

   T(2)=T(1)+ 1=2+1=3

   T(3)=T(2)+1=3+1=4 ....

   By observing the above generated equations, we can derive a formula,
3. 1+2+3+...+(n+1)➔T(n)=n+1

4. Proof by Induction

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

1. n=1, T(1)=T(1-1)+1

   T(1)=T(0)+1=1+1=2
2. Assume T(n) is true for T(n-1)

T(n-1)= (n-1) +1 =n (rule 2)

Prove T(n) is true for T(n) also

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

T(n) =**n**+1 (true)

➔T(n)=O(n)

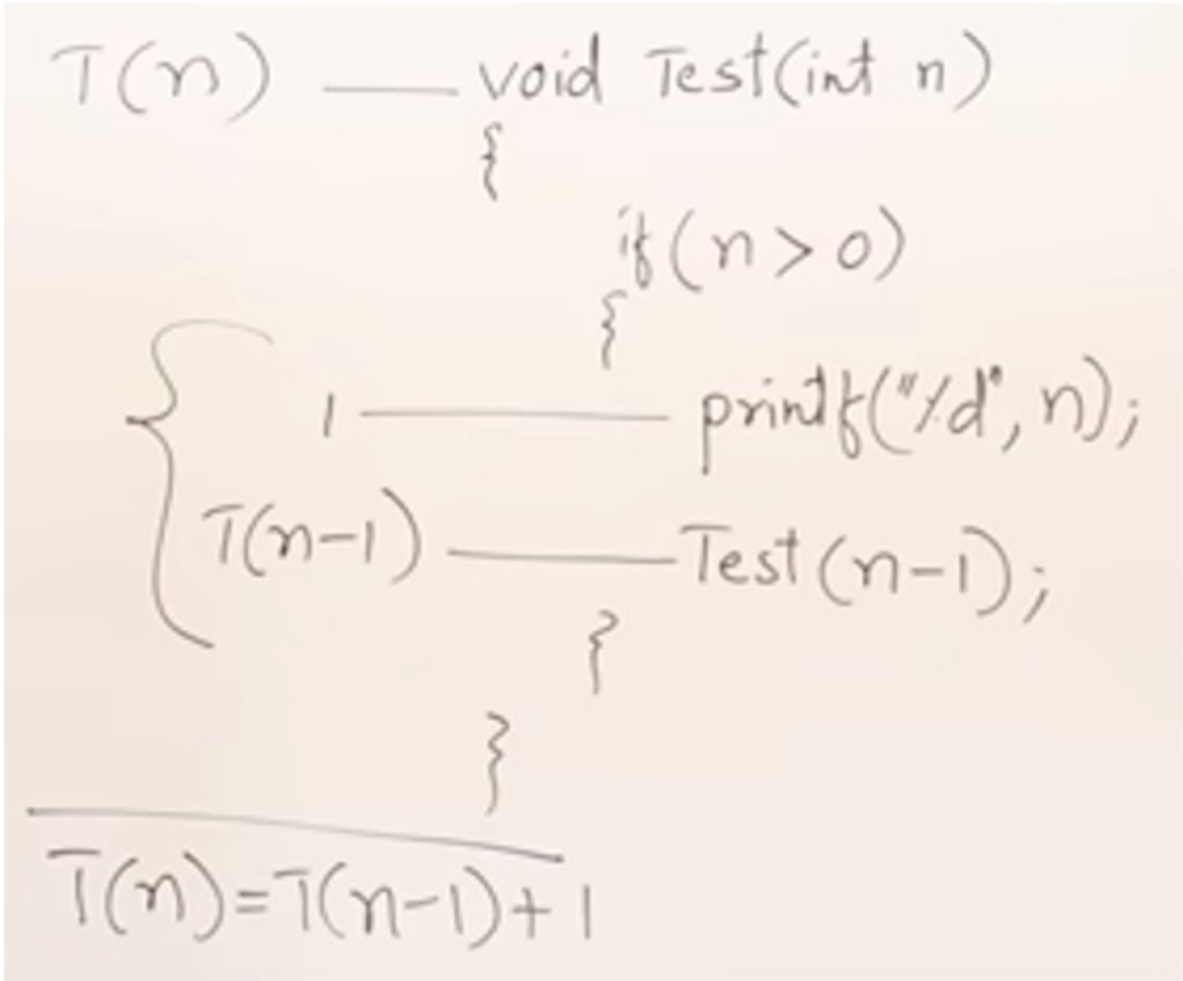
Substitution Method

Backward Substitution

Steps:

1. Take the recursive equation and initial condition
2. Guess the pattern
3. Prove that guess pattern using induction

Recurrence Relation – Example 1



$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

For this time value cannot be zero, so having some constant value

$T(n) = T(n-1)+1 \longrightarrow$ eq①.

then what is $T(n-1)$

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

$= T(n-2)+1$

Substitute $T(n-1)$ in eq①.

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

$= T(n-2)+2 \longrightarrow$ eq②

then what is $T(n-2)$

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

$= T(n-3)+1$

Substitute $T(n-2)$ in eq②.

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

$= T[n-3]+3 \longrightarrow$ eq③

then like this

:

$T(n) = T(n-k)+k$.

If $k=n$

$T(n) = T(0)+n$

$= 1+n$.

$T(n) = n$.

$T(n) = O(n)$

Substitution Method

Backward Substitution

1. Take the recursive equation and initial condition

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

2. Guess the pattern

Substitute $T(n-1)$

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

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

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

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

.

. Continue for k times

.

$$T(n) = T(n-k) + K$$

$T(n)= T(n-1) + 1$
$T(n-1) = T(n-2) +1$
$T(n-2) = T(n-3) +1$

3. Prove that guess pattern using induction

$T(n) = T(n-k) + k$
$n-k=0 \rightarrow n=k$
$T(n)=T(n-n)+n$
$T(n)=T(0) +n$
$T(n)=1+ n$
$T(n)=O(n)$

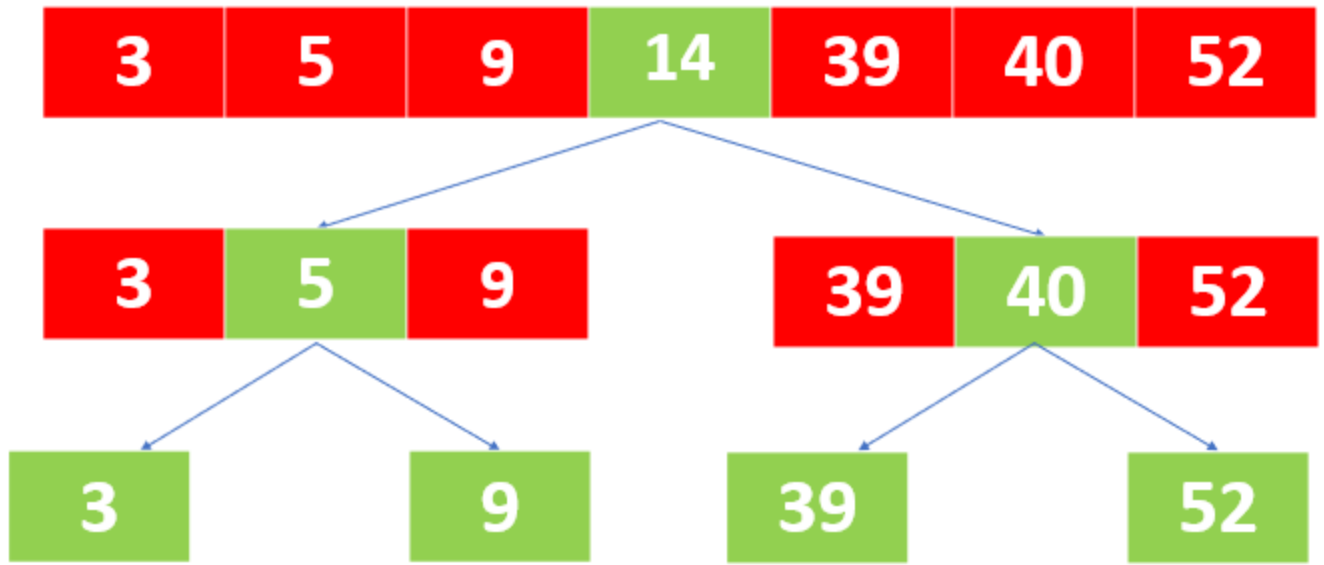
Substitution method

- 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

Binary Search – Recursion Tree



Green is marked as mid where division of array takes place


Actual number = 10
First guess = 8 < 10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Second guess = 12 > 10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Third guess = 10 = Actual number
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Binary Search time complexity

- search space size — iteration number

search space size	iteration number
n	1
n/2	2
n/4	3
...	...
1	x

- x = number of iterations
- Observe $2^x = n$, $\log 2^x = \log n$,
  x = log n iterations carried out
- Binary Search worst-case time complexity:
  O(log n)

Example: Binary Search

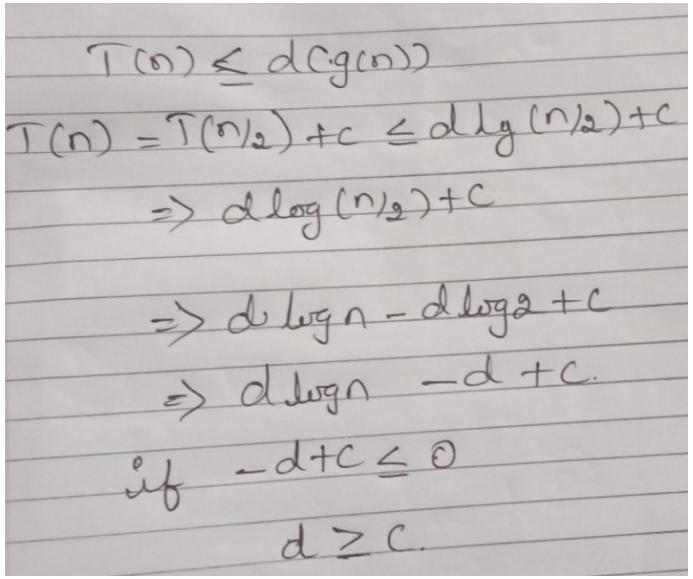
$T(n) = c + T(n/2)$

- 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:

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

$$= d \lg n - d + c \le d \lg n$$

if: $-d + c \le 0$, $d \ge c$



$$\log_b(x/y) = \log_b x - \log_b y$$

- Base case?

Example 2

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

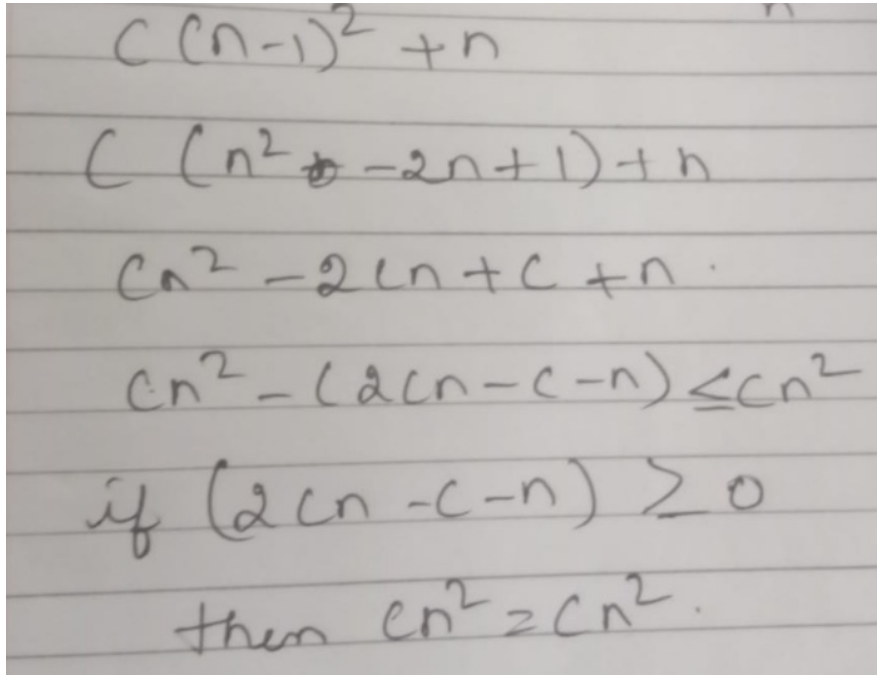
- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \leq c\, n^2$, for some c and $n \geq n_0$
  - Induction hypothesis: $T(n-1) \leq c(n-1)^2$ for all $k < n$
- Proof of induction goal:

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

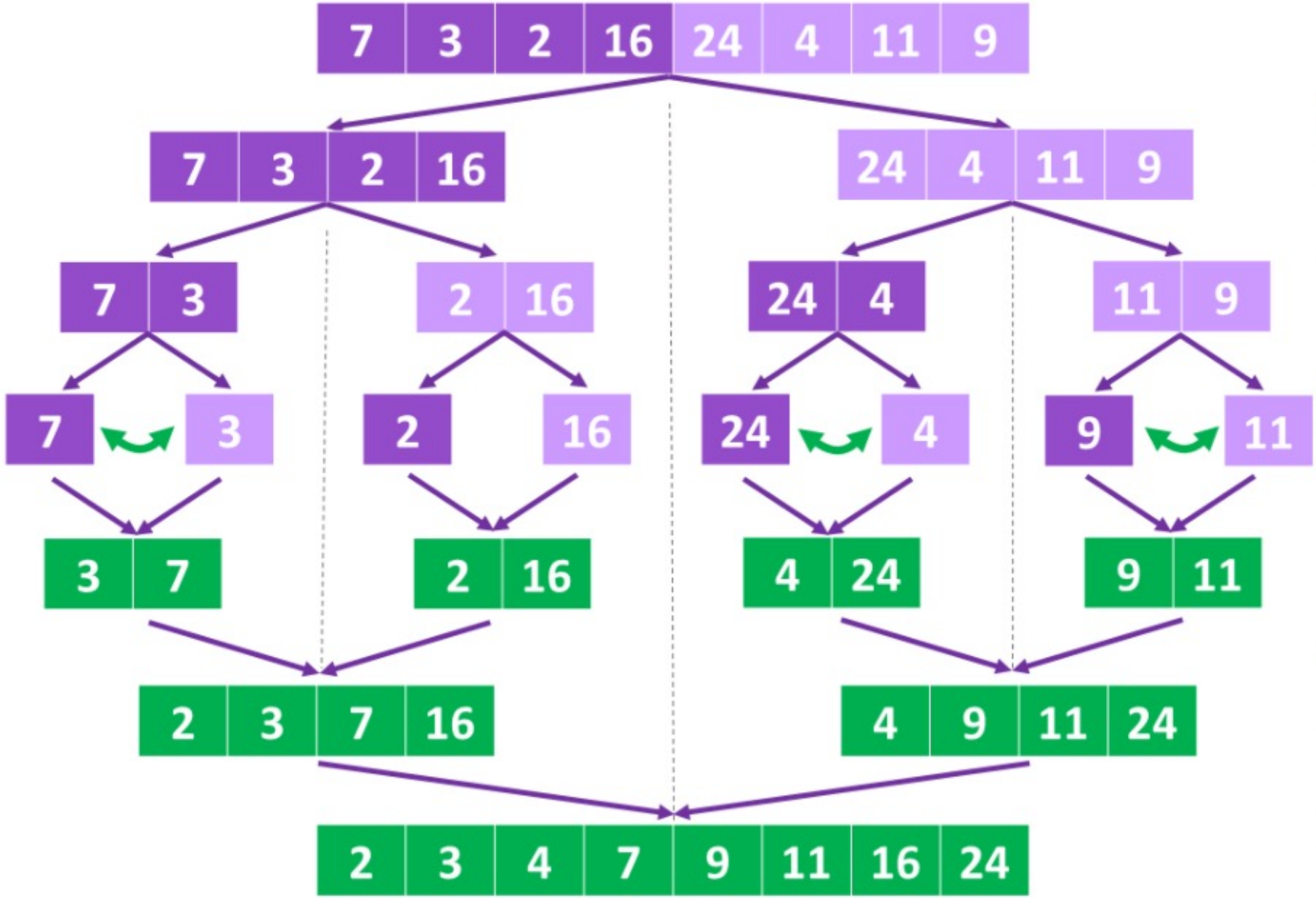
$$= cn^2 - (2cn - c - n) \leq cn^2$$

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



Merge Sort



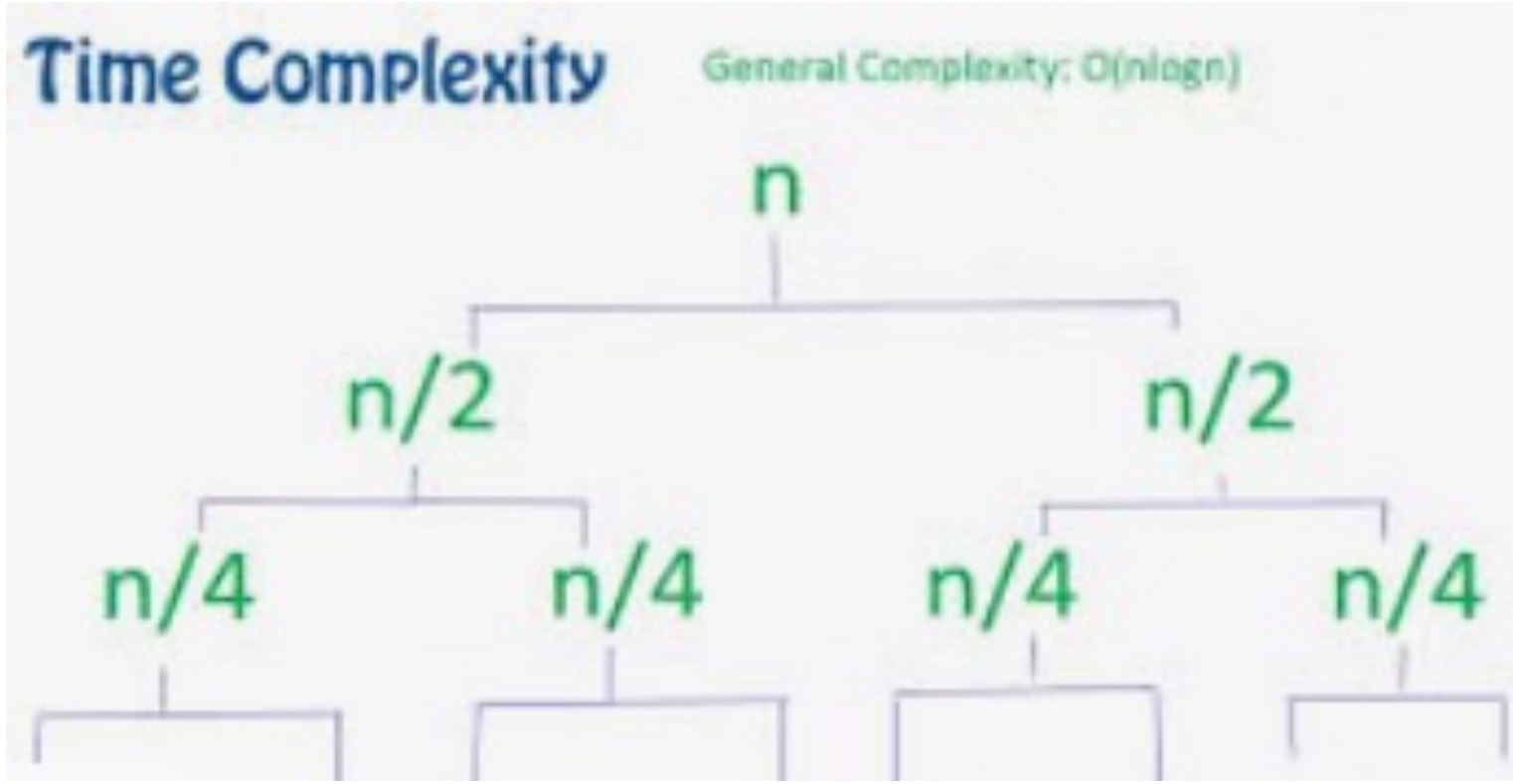
Step 1:
Split sub-lists in two until you reach pair of values.

Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Recurrence relation of the Merge sort

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



Example

Recurrence relation for Merge sort, T(n) = 2T(n/2) + n

- Guess: T(n) = O(nlgn)➔T(n)<=c.nlgn , for some c>0 and for all n>=n0
- Assume that it is true for all m<n
- **To prove:**
- T(n)<= c n lgn assuming T(m)<= c mlgm, for every m<n

  if m=n/2<n

Assuming T(n/2)<= c. n/2 lg n/2

Example

- T(n)= 2 T(n/2) + n
- <=2. c. n/2 lg n/2 +n
- =c. nlgn/2 +n
- = cn{ lgn- lg2} +1
- =cn lgn- cn +n
- <=cnlgn , -(c-1)n

Incorrect Guess

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

Guess T(n)=O(n)➔ T(n)<=c.n

Assume T(m) =O(m) <= c.m, for all m <n

To prove: T(n)<=cn assuming T(m)<= c.m, for all m <n

If m=n/2<n, T(n/2)<=cn/2

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

<= 2. c.n/2 +n

=cn+n

T(n)<=cn+n is not equal to T(n)<= cn (false,not proved)➔ guess incorrect

Substitution method

- Easy to prove
- Very fast prone to mistakes

Solving Recurrence Relation Using Recursion Tree

Step- 1:

- Draw a recursion tree based on the given recurrence relation.

Step- 2:

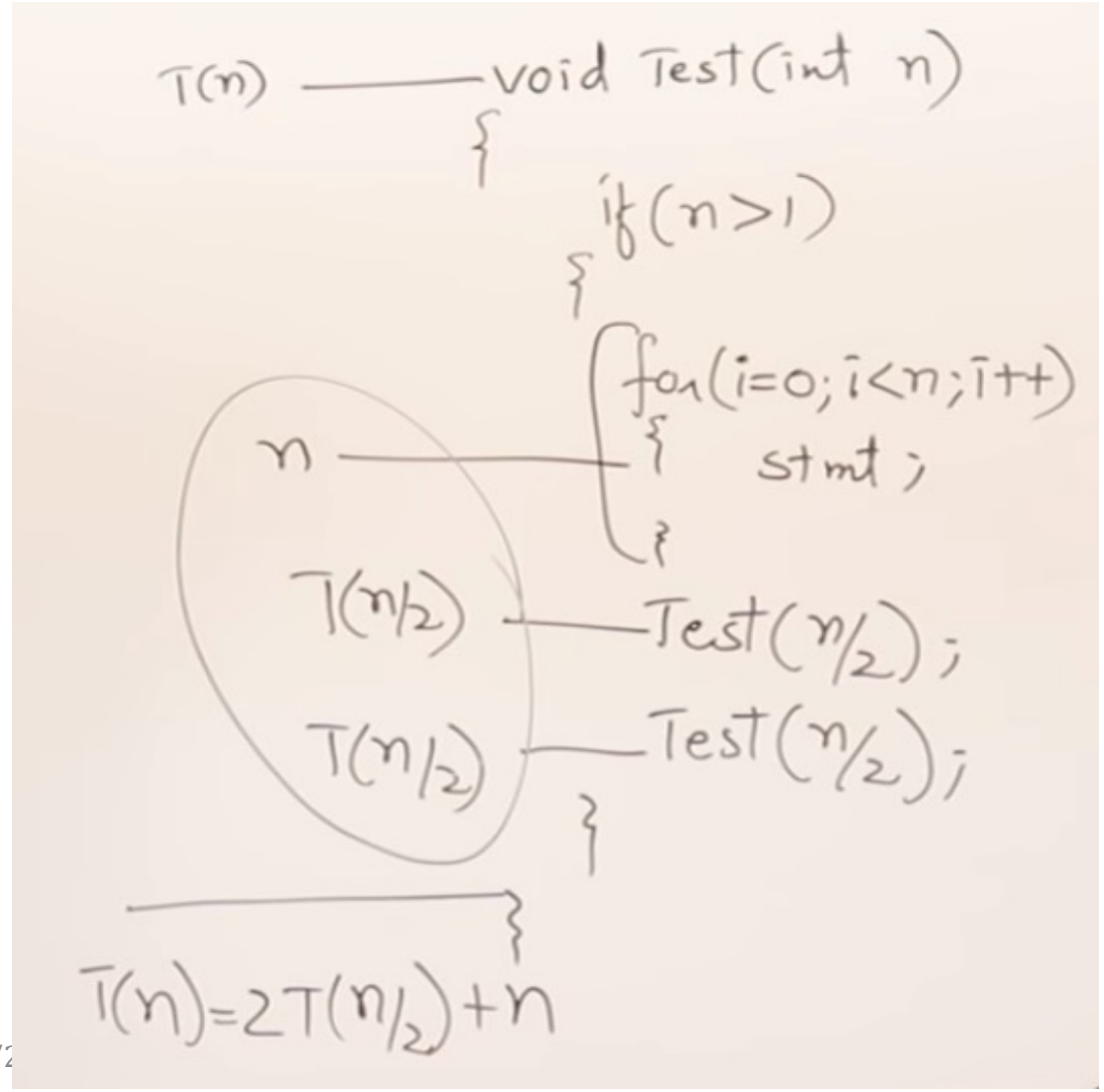
Determine-

- 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- 3:

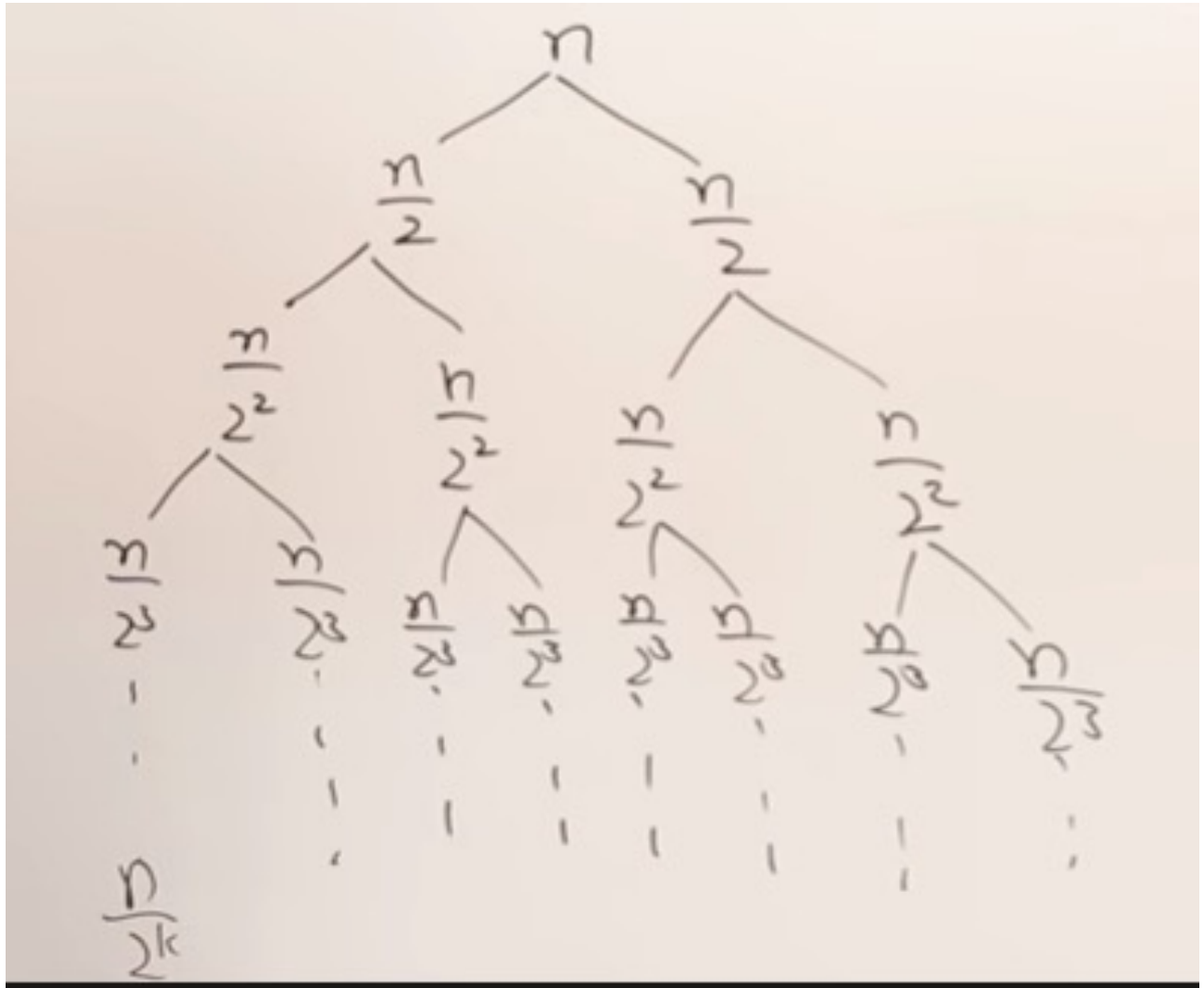
- Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

Solving Recurrence Relation Using Recursion Tree



$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+n & n>1\end{cases}$$

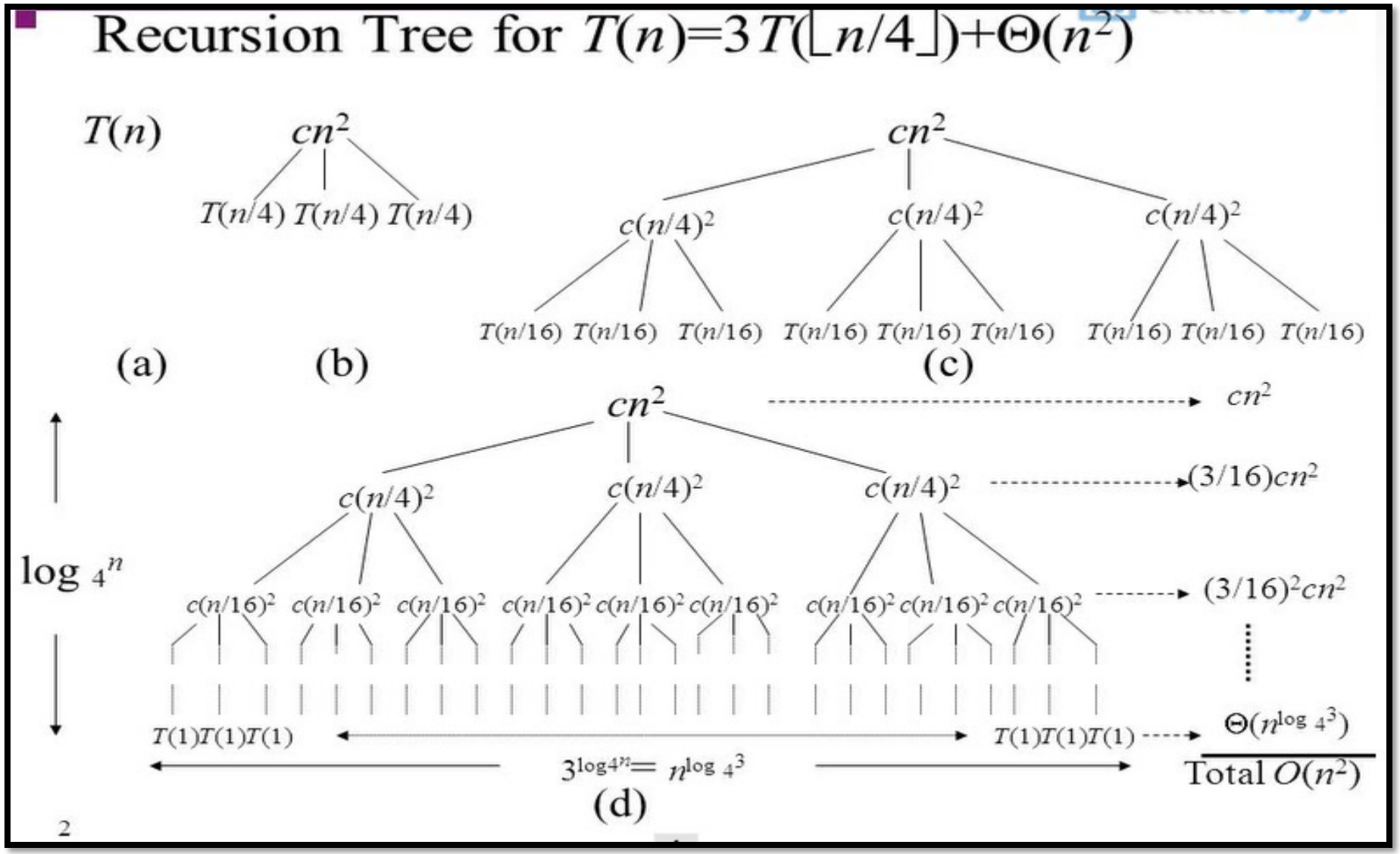
Solving Recurrence Relation Using Recursion Tree



- Total steps is k.
- Cost of each step is n
- Total cost is n *k
- k?
- Assume $\frac{n}{2^k} = 1$
- ➔ $n = 2^k$
- k=log n
- Total cost=n*k=n*log n
- ➔O(n log n)

Example 2

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$



GENERAL FORMS

$T(n) = T(n-1) + 1 \rightarrow O(n)$

$T(n) = T(n-1) + n \rightarrow O(n^2)$

$T(n) = T(n-1) + \log n \rightarrow O(n \log n)$

$T(n) = T(n-1) + n^2 \rightarrow O(n^3)$

$T(n) = T(n-2) + 1 \rightarrow \frac{n}{2} \rightarrow O(n)$

$T(n) = T(n-100) + n = O(n^2)$

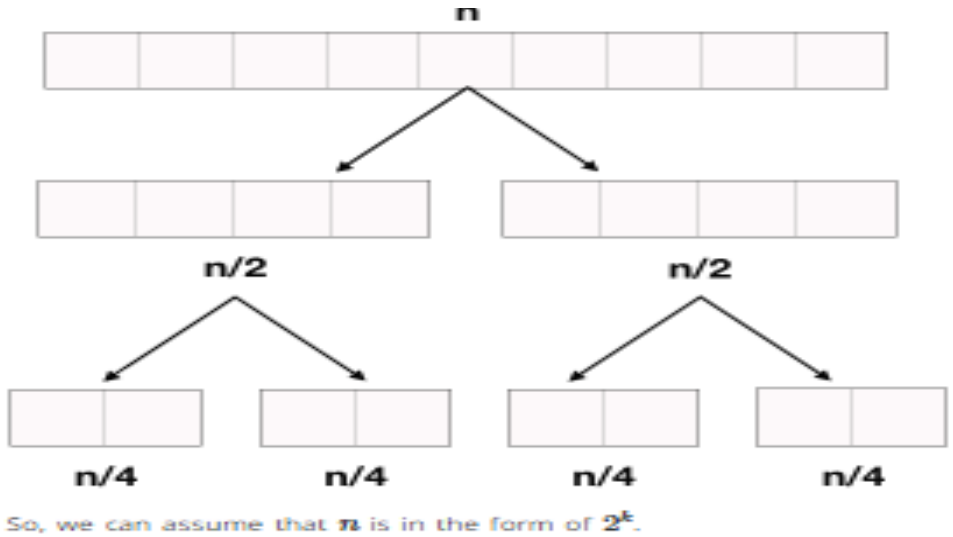
$T(n) = 2T(n-1) + 1 = ? \quad O(2^n)$

Iterative Method

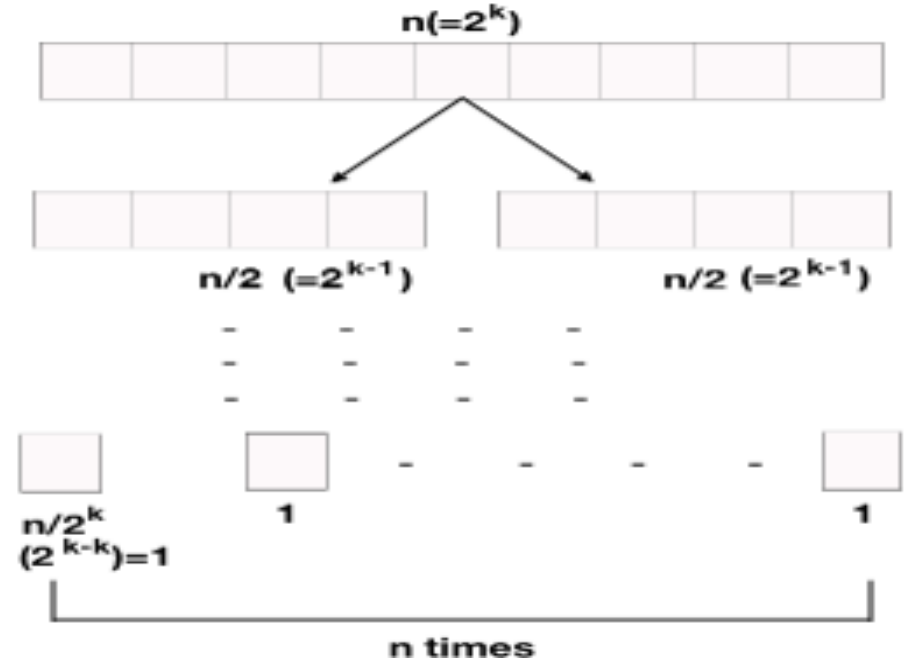
- To solve recurrence relation:
  - convert the recurrence into a summation by iterating the recurrence until the initial condition is reached.
  - break T(n) into T(n/2) and then into T(n/4) and so on.

Iterative Method

- convert the recurrence into a summation. We do so by iterating the recurrence until the initial condition is reached.



So, we can assume that n is in the form of 2^k.



Steps:

. break down the problem into n→n/2→n/4→n/8→...

. After reaching the base case, we back-substituted the equation (value of k) to express the equation in the form of n and initial boundary condition

$$T(n) = \begin{cases} T(1), & n = 1 \\ c + T\left(\frac{n}{2}\right), & \text{if } n > 1 \end{cases} \quad (1)$$

Let's replace n with $n/2$ in the previous equation.

$$T\left(\frac{n}{2}\right) = c + T\left(\frac{n}{4}\right) \quad (2)$$

where, c is a constant

Now, put the value of $T\left(\frac{n}{2}\right)$ from $eq(2)$ in the $eq(1)$, we get:

$$T(n) = c + T\left(\frac{n}{2}\right)$$

$$=> T(n) = c + \underbrace{\left(c + T\left(\frac{n}{4}\right)\right)}_{\text{value of } T\left(\frac{n}{2}\right)} \quad (3)$$

Again, let's use $T\left(\frac{n}{4}\right)$ in place of n in the $eq(1)$.

$$T\left(\frac{n}{4}\right) = c + T\left(\frac{n}{8}\right)$$

Putting this value in $eq(3)$, we get,

$$T(n) = c + c + c + T\left(\frac{n}{8}\right) \quad (4)$$

Recalling the $eq(4)$.

$$T(n) = c + c + c + T\left(\frac{n}{8}\right)$$

We can also write the above equation as

$$T(n) = 3c + T\left(\frac{n}{2^3}\right)$$

Similarly, we can write

$$T(n) = \underbrace{c + c + \ldots + c}_{k \text{ times}} + T\left(\frac{n}{2^k}\right)$$

$$\text{Since } n = 2^k,$$

$$T(n) = \underbrace{c + c + \ldots + c}_{k \text{ times}} + T\left(\frac{2^k}{2^k}\right)$$

$$= kc + T(1) \tag{5}$$

As mentioned above, $T(1)$ is the base case. So, we have reached the base case.

Also,

$$n = 2^k$$

$$=> \log_2(n) = log_2(2^k)$$

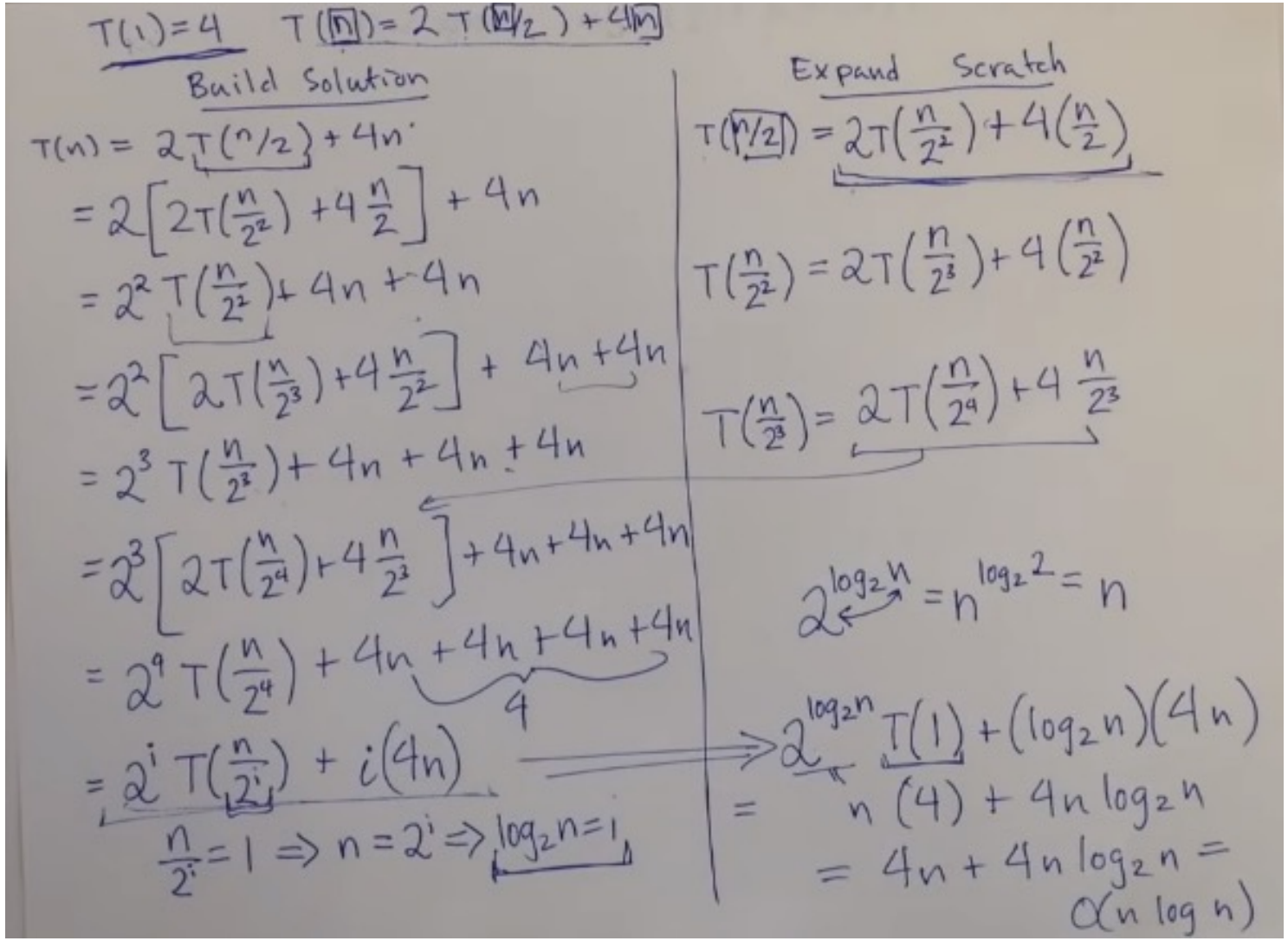
$$=> \lg(n) = k$$

Replacing the value of k in the $eq(5)$, we get

$$T(n) = c\lg(n) + T(1)$$

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

Iterative Method



Master's Method

- Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

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

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem.

All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions Here, $a \geq 1$ and $b > 1$ are constants, and $f(n) > 0$.

Master's Theorem

- If $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function, then the time complexity of a recursive relation is given by

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

where,

$T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$. $\epsilon > 0$ is a constant.

Master's Theorem

Each of the above conditions can be interpreted as:

- If the cost of solving the sub-problems at each level increases by a certain factor, the value of f(n) will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level ie. $n^{\log_b a}$
- If the cost of solving the sub-problem at each level is nearly equal, then the value of f(n) will be $n^{\log_b a}$. Thus, the time complexity will be f(n) times the total number of levels ie. $n^{\log_b a} * \log n$
- If the cost of solving the subproblems at each level decreases by a certain factor, the value of f(n) will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of f(n).

Master’s Theorem

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

Master's Theorem

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

Simplified three cases of master theorem

Master Theorem Cases-

To solve recurrence relations using Master's theorem, we compare a with b^k.

Case-01:

If $a > b^k$, or $\log_b a > k$ then $T(n) = \theta(n^{\log_b a})$

Case-02:

If $a = b^k$ or $\log_b a = k$ and

If $p < -1$, then $T(n) = \theta(n^{\log_b a})$

If $p = -1$, then $T(n) = \theta(n^{\log_b a}.\log^2 n)$

If $p > -1$, then $T(n) = \theta(n^{\log_b a}.\log^{p+1} n)$

Case-03:

If $a < b^k$ or $\log_b a < k$ and

If $p < 0$, then $T(n) = O(n^k)$

If $p >= 0$, then $T(n) = \theta(n^k \log^p n)$

Example 1- Case 1

Solve the following recurrence relation using Master's theorem-
$T(n) = 8T(n/2) + n \log n$

Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
Then, we have-
$a = 8$
$b = 2$
$k = 1$
$p = 1$

Now, $a = 8$ and $b^k = 2^0 = 1$.
Clearly, $a > b^k$.
So, we follow case-01.

So, we have-
$T(n) = \theta (n^{\log_b a})$
$T(n) = \theta (n^{\log_2 8})$
$T(n) = \theta (n^3)$

Thus, **$T(n) = \theta (n^3)$**

Case-01:
If $a > b^k$, or $\log_b a > k$ then $T(n) = \theta (n^{\log_b a})$

Case-02:
If $a = b^k$ or $\log_b a = k$ and
If $p < -1$, then $T(n) = \theta (n^{\log_b a})$
If $p = -1$, then $T(n) = \theta (n^{\log_b a} . \log^2 n)$
If $p > -1$, then $T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$

Case-03:
If $a < b^k$ or $\log_b a < k$ and
If $p < 0$, then $T(n) = O (n^k)$
If $p >= 0$, then $T(n) = \theta (n^k \log^p n)$

Example 1- Case 2

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

Case-01:
If $a > b^k$, or $\log_b a > k$ then $T(n) = \theta(n^{\log_b a})$

Case-02:
If $a = b^k$ or $\log_b a = k$ and
If $p < -1$, then $T(n) = \theta(n^{\log_b a})$
If $p = -1$, then $T(n) = \theta(n^{\log_b a}.\log^2 n)$
If $p > -1$, then $T(n) = \theta(n^{\log_b a}.\log^{p+1} n)$

Case-03:
If $a < b^k$ or $\log_b a < k$ and
If $p < 0$, then $T(n) = O(n^k)$
If $p >= 0$, then $T(n) = \theta(n^k \log^p n)$

Example 2- Case 2

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, **$T(n) = \theta (n \log n)$**

Case-01:
If $a > b^k$, or $\log_b a > k$ then $T(n) = \theta (n^{\log_b a})$

Case-02:
If $a = b^k$ or $\log_b a = k$ and
If $p < -1$, then $T(n) = \theta (n^{\log_b a})$
If $p = -1$, then $T(n) = \theta (n^{\log_b a}.\log^2 n)$
If $p > -1$, then $T(n) = \theta (n^{\log_b a}.\log^{p+1} n)$

Case-03:
If $a < b^k$ or $\log_b a < k$ and
If $p < 0$, then $T(n) = O (n^k)$
If $p >= 0$, then $T(n) = \theta (n^k \log^p n)$

Example 1- Case 3

Solve the following recurrence relation using Master's theorem-

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

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, **$T(n) = \theta (n^2)$**

Case-01:
If $a > b^k$, or $\log_b a > k$ then $T(n) = \theta (n^{\log_b a})$

Case-02:
If $a = b^k$ or $\log_b a = k$ and
If $p < -1$, then $T(n) = \theta (n^{\log_b a})$
If $p = -1$, then $T(n) = \theta (n^{\log_b a} . \log^2 n)$
If $p > -1$, then $T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$

Case-03:
If $a < b^k$ or $\log_b a < k$ and
If $p < 0$, then $T(n) = O (n^k)$
If $p >= 0$, then $T(n) = \theta (n^k \log^p n)$

Example 2-Case 3

Solve the following recurrence relation using Master's theorem-
$T(n) = 2T(n/4) + n^{0.51}$

Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
Then, we have-

a = 2
b = 4
k = 0.51
p = 0

Now, $a = 2$ and $b^k = 4^{0.51} = 2.0279$.
Clearly, $a < b^k$.
So, we follow case-03.

Since p = 0, so we have-
$T(n) = \theta (n^k \log^p n)$
$T(n) = \theta (n^{0.51} \log^0 n)$

Thus, **$T(n) = \theta (n^{0.51})$**

Case-01:
If $a > b^k$, or $\log_b a > k$ then $T(n) = \theta (n^{\log_b a})$

Case-02:
If $a = b^k$ or $\log_b a = k$ and
If $p < -1$, then $T(n) = \theta (n^{\log_b a})$
If $p = -1$, then $T(n) = \theta (n^{\log_b a} . \log^2 n)$
If $p > -1$, then $T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$

Case-03:
If $a < b^k$ or $\log_b a < k$ and
If $p < 0$, then $T(n) = O (n^k)$
If $p >= 0$, then $T(n) = \theta (n^k \log^p n)$

Home Assignments

- T(n)=2T(n/2)+1
- T(n)=4T(n/2)+n
- T(n)=8T(n/2)+ n^2
- T(n)=9T(n/3)+ n^2
- T(n)=2T(n/2)+n
- T(n)= 4T(n/2)+ n^2
- T(n)= 4T(n/2)+ n^2 log n
- T(n)= 8T(n/2) + n^3
- T(n)= T(n/2)+ n^2