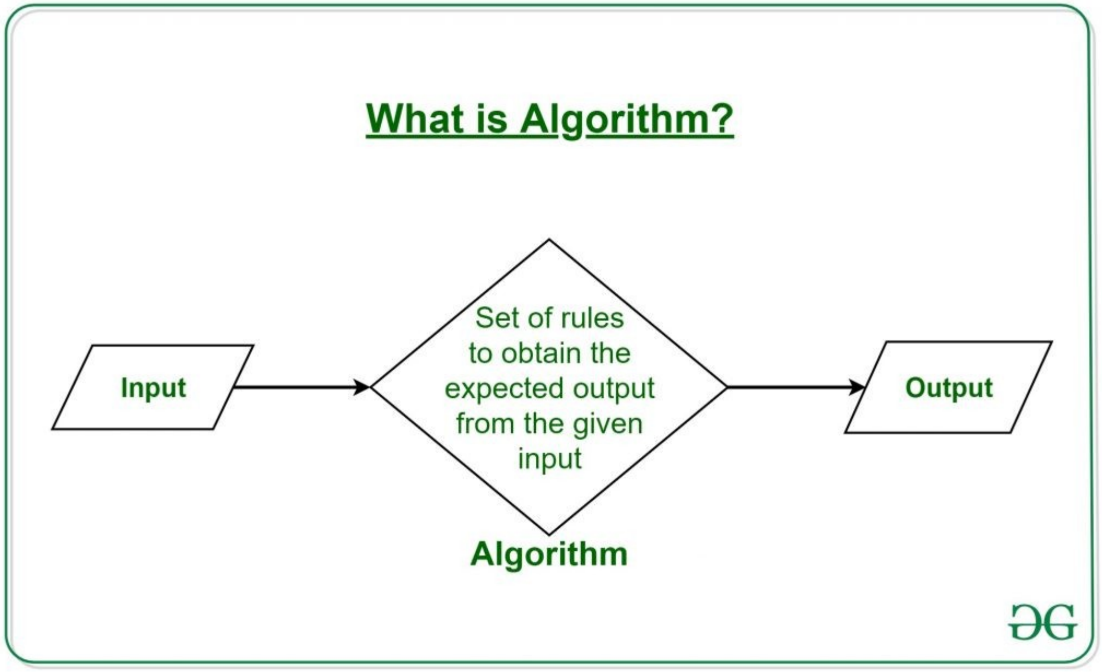
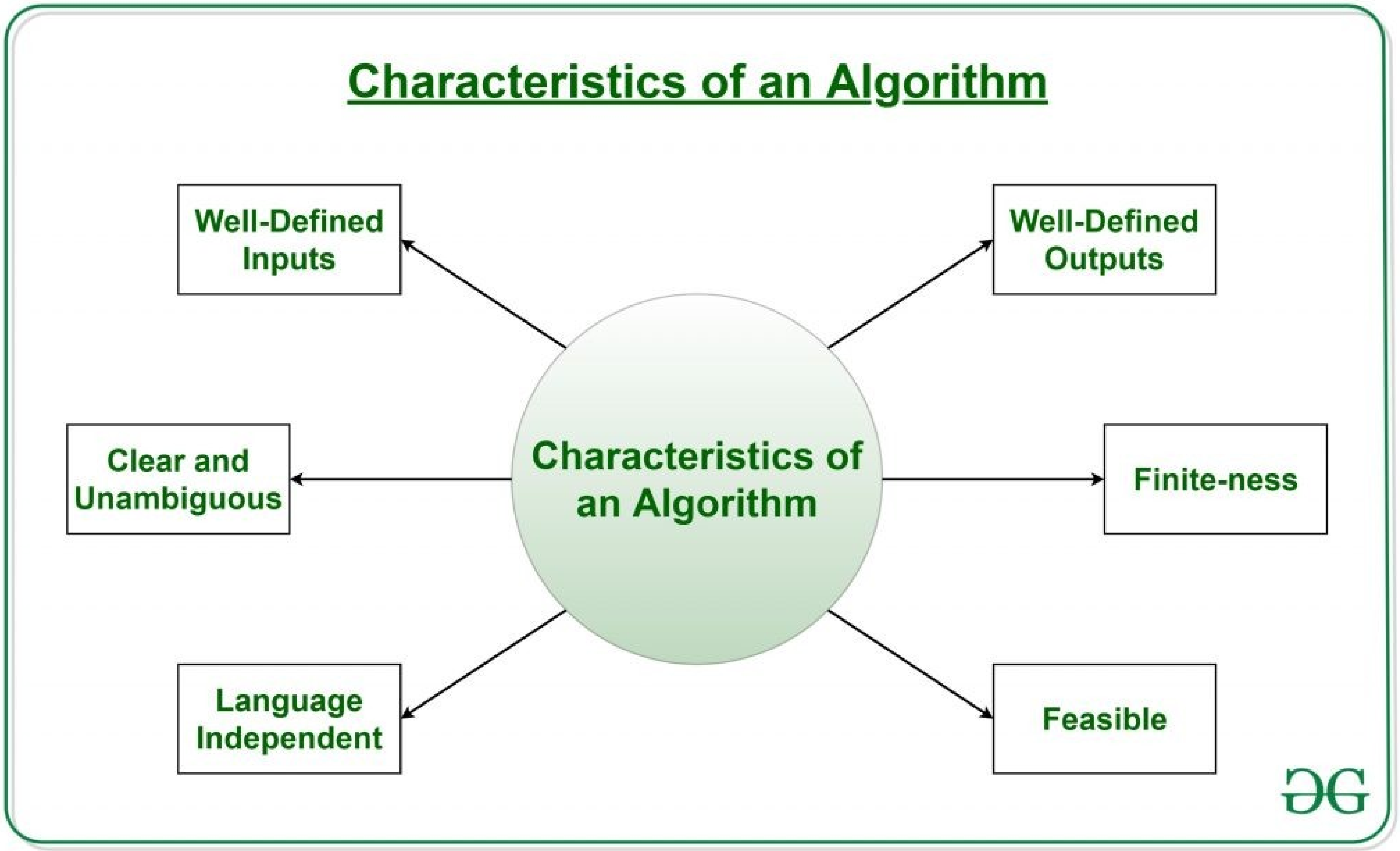
Asymptotic Notations

Algorithm

- An **algorithm** is a finite sequence of **well-defined**, **computer-implementable instructions**, typically to solve a problem or to perform a computation


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm


Characteristics of an Algorithm
Well-Defined Inputs
Well-Defined Outputs
Clear and Unambiguous
Characteristics of an Algorithm
Finite-ness
Language Independent
Feasible

Characteristics of Algorithm

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

Basic Terminologies

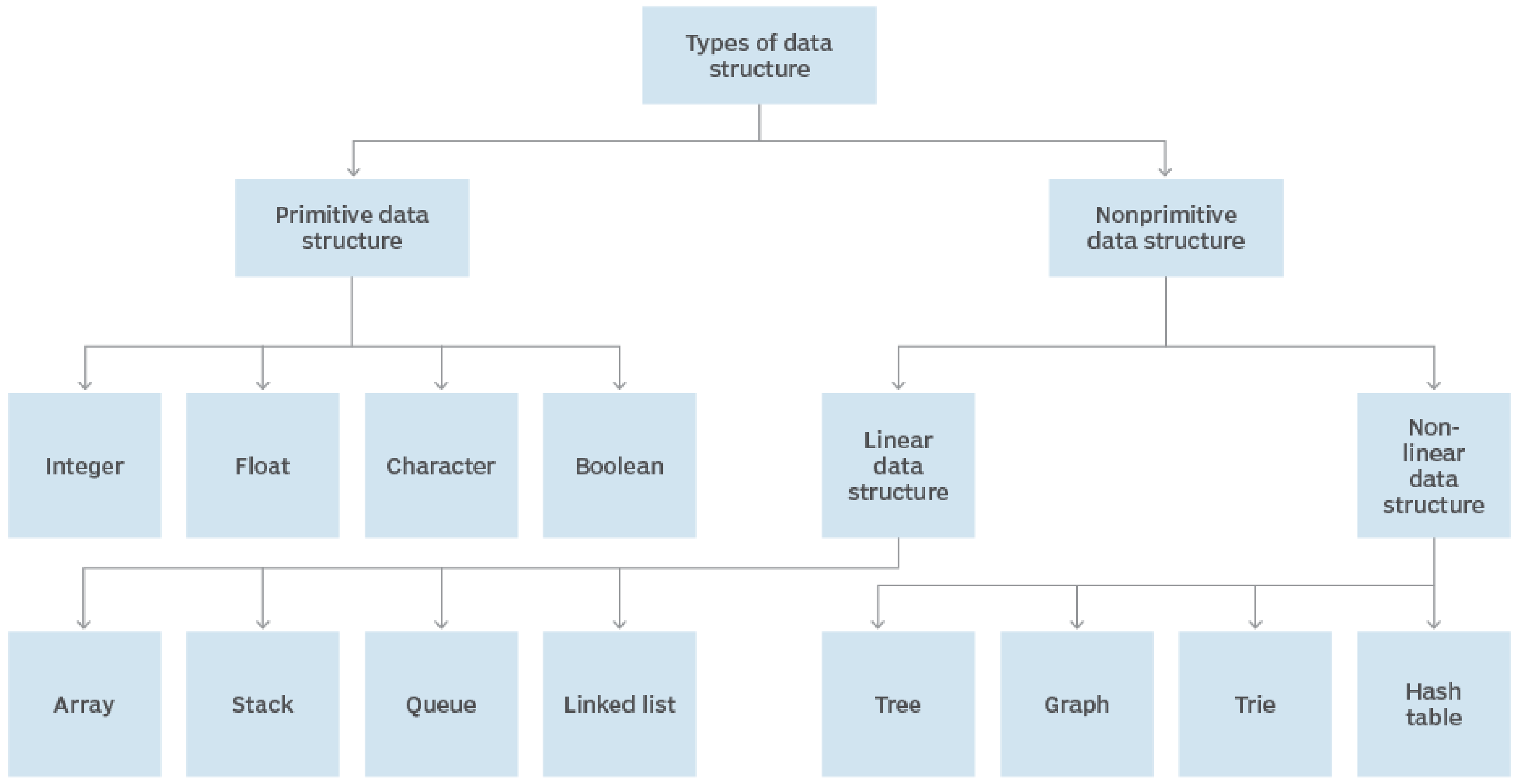
❑**Program**

Implementation of an algorithm in a programming language

❑**Data Structure**

It is a particular way of organizing a data in computer’s memory so that it can be used easily and effectively by a program to solve a given problem.

Data structure hierarchy



Algorithm Complexity

- For a given problem , there can be many algorithms possible to solve it.
- **The main issue is to determine which algorithm is the best among all to solve**
- All these algorithms differ in efficiencies.
- To measure the efficiency of an algorithm, we need to analyze the algorithm.
- We know that when a program implementing an algorithm is executed, it uses the resources of the computing system such as CPU, memory etc.
- The study of the algorithm is necessary to determine the amount of time and storage space an algorithm may require for execution.

Algorithm Complexity

- The study of algorithm is necessary to determine the amount of time and storage space an algorithm my require for execution. This study is called **algorithm complexity**.
- There are two types of complexity with any algorithm:
  - ❖ **Time Complexity**
  - ❖ **Space Complexity**
- How efficient a particular algorithm is of no concern when the amount od data to be processed is small.
- The efficiency of an algorithm varies if the amount of data to be processed is very large.

Algorithm Complexity

- The best algorithm will be the one which is most efficient with respect to time and space it requires to solve .
- An efficient algorithm is the one that makes the space requirement as low as possible.
- Although, space complexity is important but inexpensive memory has reduced its significance.
- Thus our main area of concern is the time complexity.

Time Complexity

- The main objective of time complexity is to compare the performance of different algorithms.
- How to measure time complexity?
- One simple way is to implement each algorithm using a PL one by one and determine which takes how much time it takes to solve the problem.
- But this method is not feasible because implementing each and every algorithm would waste huge amount of time.
- Secondly, the configuration of computing system on which the programs would be executed, would greatly influence its running time. Thus, **we are not interested in absolute time, i.e., how many seconds it takes to solve a particular problem**, as it is not a useful measure of an algorithm’s performance.

Time Complexity

- A better method is to employ a mathematical model to analyze algorithms independent of specific implementations, computers, programming languages etc.
- This method analyze the performance by **counting the number of key operations in an algorithm**. They key operations can be easily identified using a pseudo code.
- For example, in searching, the key operations are the number of comparisons made and in sorting, the key operations are the number of swapping and comparisons.
- The time complexity is measured using key operations because time involved in performing other operations is much less or atmost proportional to time for key operations.
- **The number of key operations performed by the algorithm is itself a function of the input size.**

	Cost	Frequency
1. **for** i = 1 to N – 1 **do**	c1	N
2. sum = 0	c2	N – 1
3. **for** j = 0 to i **do**	c3	$\sum_{i=1}^{N-1}(i+2)$
4. sum = sum + A[j]	c4	$\sum_{i=1}^{N-1}(i+1)$
5. A[i] = sum	c5	N – 1

	Cost	Frequency
1. **for** i = 1 to N – 1 **do**	c1	N
2. sum = 0	c2	N – 1
3. **for** j = 0 to i **do**	c3	$\sum_{i=1}^{N-1}(i+2)$
4. sum = sum + A[j]	c4	$\sum_{i=1}^{N-1}(i+1)$
5. A[i] = sum	c5	N – 1

$$c1N + c2(N-1) + c3\sum_{i=1}^{N-1}(i+2) + c4\sum_{i=1}^{N-1}(i+1) + c5(N-1)$$

$$c1N + c2N - c2 + c3\left(\frac{N(N-1)}{2}\right) + 2.c3.N - 2.c3 + c4\left(\frac{N(N-1)}{2}\right) + c4N - c4 + c5N - c5)$$

$$N^2\left(\frac{c3}{2}+\frac{c4}{2}\right) + N\left(c1 + c2 + \frac{3}{2}c3 + \frac{c4}{2} + c5\right) - (c2 + 2.c3 + c4 + c5)$$

Time Complexity

$$f(n) = n^2 + 27n + 1005$$

			Contribution
10	1375	100	7.27%
100	13705	10000	72.96%
1000	1028005	1000000	97.27%
10000	100271005	100000000	99.72%

Asymptotic Complexity

- The simplified form of time complexity function by discarding all the terms that do not substantially contribute to the function’s magnitude is called **asymptotic complexity**.
- The resultant function gives only an approximate time complexity of the original function.
- However, this approximation is sufficiently close to the original one, especially for large amount of data.
- For processing large amount of data we are only concerned with the dominant term in the complexity function, i.e., the term with the largest order of magnitude.

Rate of Growth

The rate at which the running time increases as a function of input is called rate of growth.

The input can be categorized as:

- Size of array
- Degree of polynomial
- Number of elements in matrix
- Vertices and edges in a graph
- Number of bits in binary representation of the input

Big-Oh Notation (O)

For a given function $g(n) \geq 0$, denoted by $\text{O}(g(n))$ the set of functions,

$\text{O}(g(n)) = \{ f(n)$: there exist positive constants c and n_o such that

$0 \leq f(n) \leq cg(n)$, for all $n \geq n_o \}$

$f(n) = \text{O}(g(n))$ means function $g(n)$ is an asymptotically

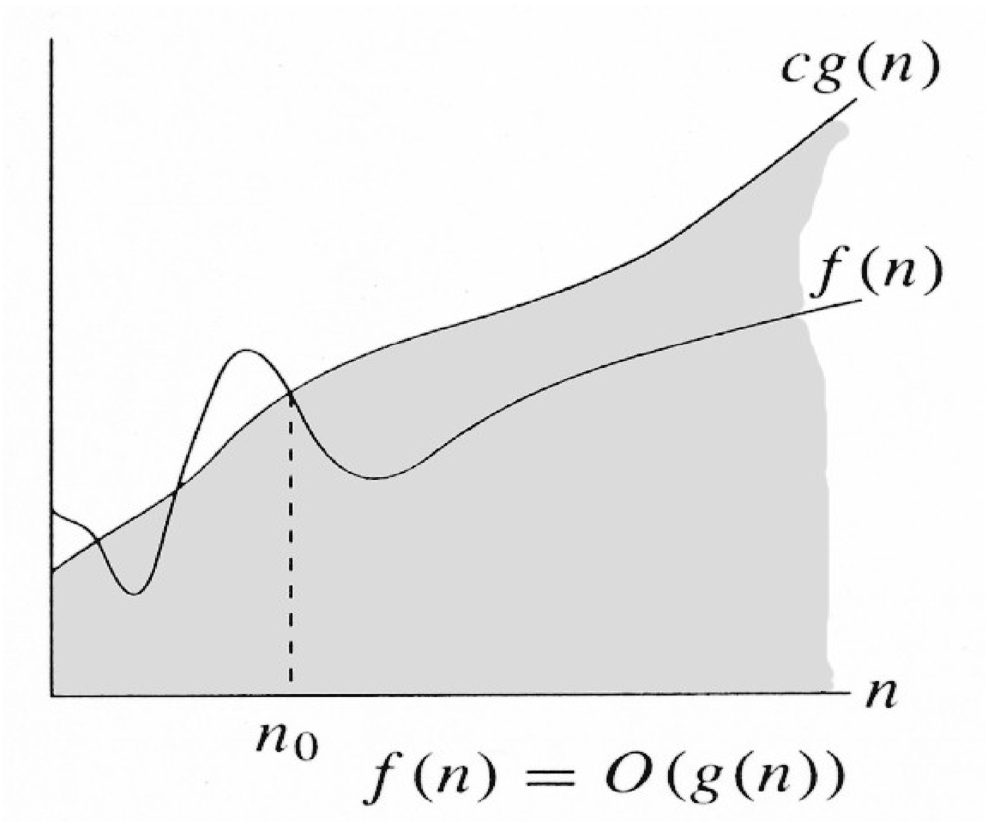
upper bound for $f(n)$.

We may write f(n) = O(g(n)) OR f(n) $\in$ O(g(n))

Intuitively:

Set of all functions whose *rate of growth* is the same as or lower than that of *g*(*n*).

Big-Oh Notation

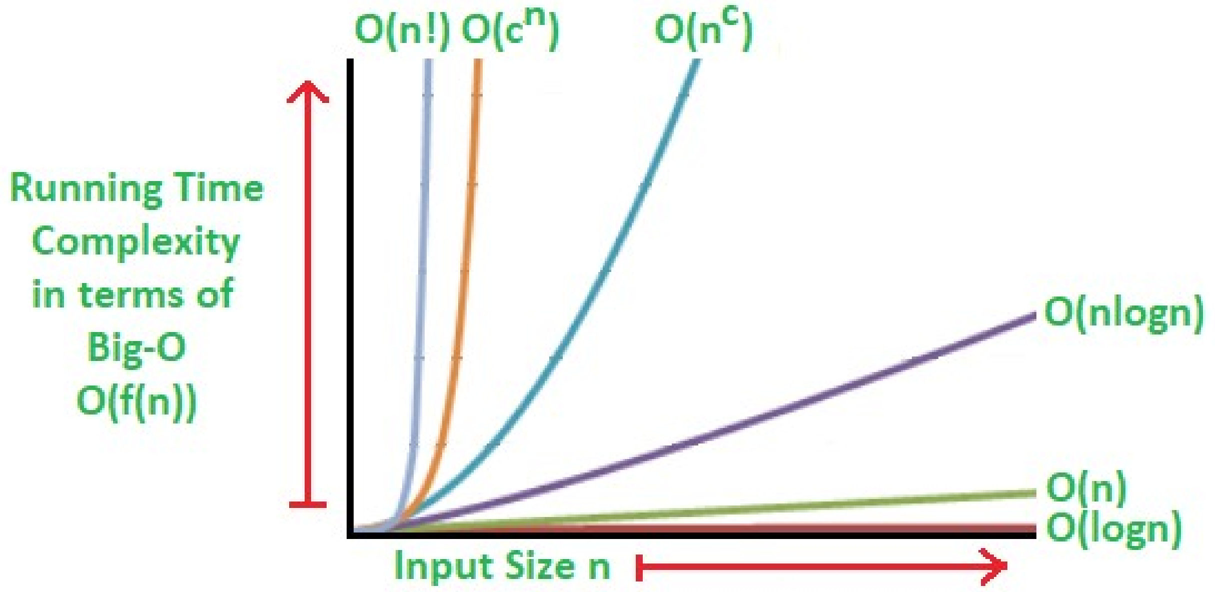


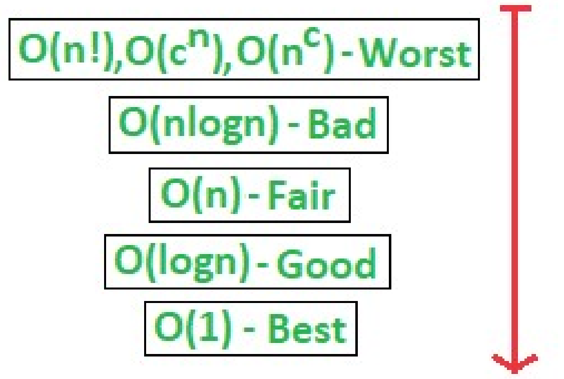
Intuitively:
Set of all functions whose *rate of growth* is the same as or lower than that of *g*(*n*).

f(n) ∈ O(g(n))

$\exists\, c > 0, \exists\, n_0 \geq 0$ **and** $\forall n \geq n_0,\ 0 \leq f(n) \leq c.g(n)$

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





$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^c) < O(2^n) < O(c^n) < O(n!)$$

Examples

Example 1: Prove that $2n^2 \in O(n^3)$

Proof:

Assume that $f(n) = 2n^2$, and $g(n) = n^3$

$f(n) \in O(g(n))$?

Now we have to find the existence of c and n_0

$f(n) \leq c.g(n) \Leftrightarrow 2n^2 \leq c.n^3 \Leftrightarrow 2 \leq c.n$

if we take, $c = 1$ and $n_0 = 2$ OR

$c = 2$ and $n_0 = 1$ then

$2n^2 \leq c.n^3$

Hence $f(n) \in O(g(n))$, $c = 1$ and $n_0 = 2$

Examples

Example 2: Prove that $n^2 \in O(n^2)$

Proof:

Assume that $f(n) = n^2$, and $g(n) = n^2$

Now we have to show that $f(n) \in O(g(n))$

Since

$f(n) \leq c.g(n) \Leftrightarrow n^2 \leq c.n^2 \Leftrightarrow 1 \leq c$, take, $c = 1$, $n_0= 1$

Then

$n^2 \leq c.n^2$ for $c = 1$ and $n \geq 1$

Hence, $2n^2 \in O(n^2)$, where $c = 1$ and $n_0= 1$

Examples

Example 3: Prove that $1000.n^2 + 1000.n \in O(n^2)$

Proof:

Assume that $f(n) = 1000.n^2 + 1000.n$, and $g(n) = n^2$

We have to find existence of c and n_0 such that

$0 \leq f(n) \leq c.g(n)$ A $n \geq n_0$

$1000.n^2 + 1000.n \leq c.n^2 = 1001.n^2$, for $c = 1001$

$1000.n^2 + 1000.n \leq 1001.n^2$

Û $1000.n \leq n^2 \Leftrightarrow n^2 \geq 1000.n \Leftrightarrow n^2 - 1000.n \geq 0$

Û $n (n-1000) \geq 0$, this true for $n \geq 1000$

$f(n) \leq c.g(n)$ A $n \geq n_0$ and $c = 1001$

Hence $f(n) \in O(g(n))$ for $c = 1001$ and $n_0 = 1000$

Examples

Example 4: Prove that n^3 📋 $O(n^2)$

Proof:

On contrary we assume that there exist some positive constants c and n_0 such that

$0 \le n^3 \le c.n^2 \quad A\ n \ge n_0$

$0 \le n^3 \le c.n^2 \Leftrightarrow n \le c$

Since c is any fixed number and n is any arbitrary constant, therefore $n \le c$ is not possible in general.

Hence our supposition is wrong and $n^3 \le c.n^2$,

$A\ n \ge n_0$ is not true for any combination of c and n_0.

And hence, n^3 📋 $O(n^2)$

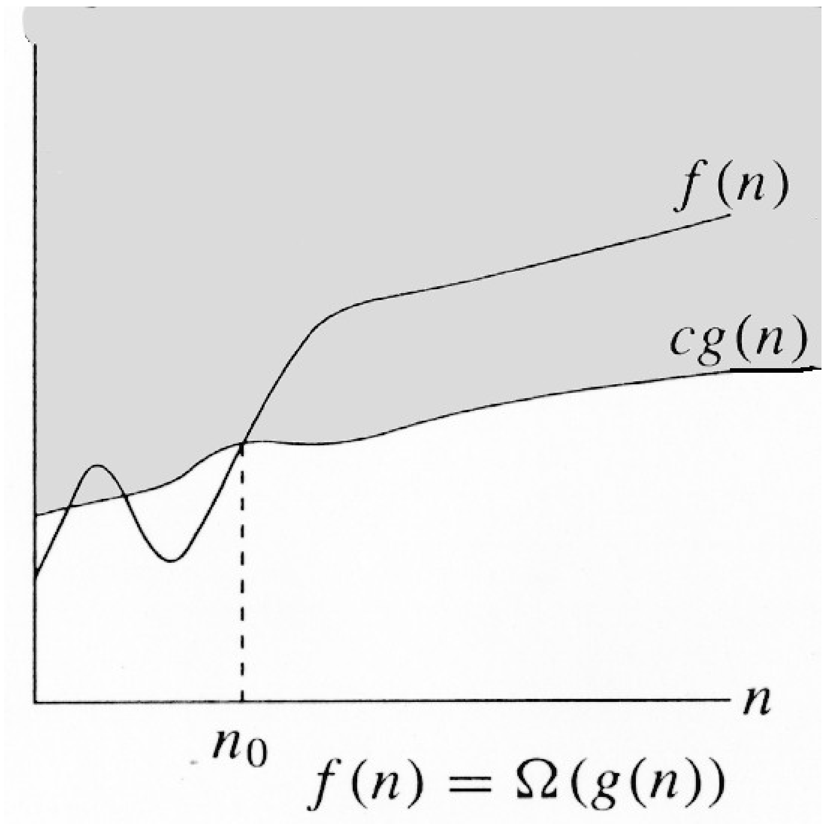
Big-Omega Notation (Ω)

For a given function $g(n)$ denote by $\Omega(g(n))$ the set of functions,
$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_o such that
$0 \le cg(n) \le f(n)$ for all $n \ge n_o \}$
$f(n) = \Omega(g(n))$, means that function $g(n)$ is an asymptotically lower bound for $f(n)$.

We may write $f(n) = \Omega(g(n))$ OR $f(n) \in \Omega(g(n))$

Intuitively:
Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

Big-Omega Notation



Intuitively:
Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$$\mathbf{f(n) \in \Omega(g(n))}$$

$\exists\, c > 0, \exists\, n_0 \geq 0,\ \forall n \geq n_0,\ f(n) \geq c.g(n)$

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

Examples

Example 1: Prove that $5.n^2 \in \Omega(n)$

Proof:

Assume that $f(n) = 5.n^2$, and $g(n) = n$

$f(n) \in \Omega(g(n))$?

We have to find the existence of c and n_0 s.t.

$c.g(n) \leq f(n) \quad \forall\, n \geq n_0$

$c.n \leq 5.n^2 \Leftrightarrow c \leq 5.n$

if we take, $c = 5$ and $n_0 = 1$ then

$c.n \leq 5.n^2 \quad \forall\, n \geq n_0$

And hence $f(n) \in \Omega(g(n))$, for $c = 5$ and $n_0 = 1$

Examples

Example 2: Prove that $5.n + 10 \in \Omega(n)$

Proof:

Assume that $f(n) = 5.n + 10$, and $g(n) = n$

$f(n) \in \Omega(g(n))$?

We have to find the existence of c and n_0 s.t.

$c.g(n) \leq f(n) \;\; A \; n \geq n_0$

$c.n \leq 5.n + 10 \Leftrightarrow c.n \leq 5.n + 10.n \Leftrightarrow c \leq 15.n$

if we take, $c = 15$ and $n_0 = 1$ then

$c.n \leq 5.n + 10 \;\; A \; n \geq n_0$

And hence $f(n) \in \Omega(g(n))$, for $c = 15$ and $n_0 = 1$

Examples

Example 3: Prove that $100.n + 5 \notin \Omega(n^2)$

Proof:

Let $f(n) = 100.n + 5$, and $g(n) = n^2$

Assume that $f(n) \in \Omega(g(n))$?

Now if $f(n) \in \Omega(g(n))$ then there exist c and n_0 s.t.

$c.g(n) \leq f(n) \quad \text{A } n \geq n_0 \Leftrightarrow$

$c.n^2 \leq 100.n + 5 \quad \Leftrightarrow$

$c.n \leq 100 + 5/n \quad \Leftrightarrow$

$n \leq 100/c$, for a very large n, which is not possible

And hence f(n) 📋 $\Omega(g(n))$

Theta Notation (Θ)

For a given function $g(n)$ denoted by $\Theta(g(n))$ the set of functions,

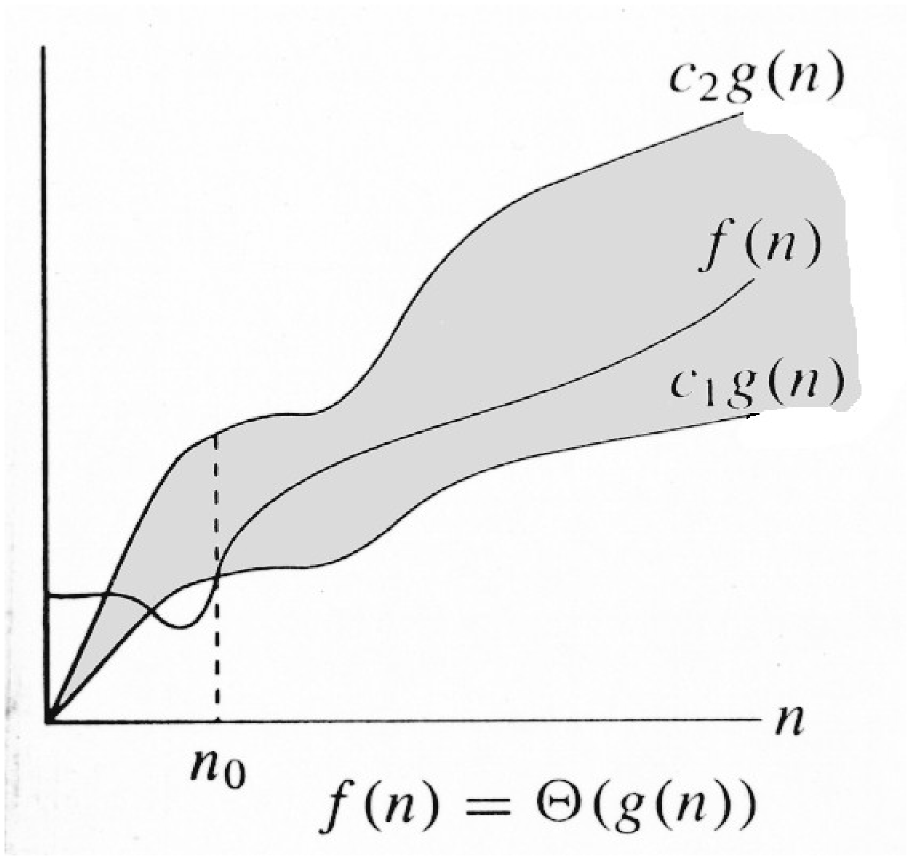
$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_o such that

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_o \}$

We may write f(n) = Θ(g(n)) OR f(n) $\in$ Θ(g(n))

Intuitively: Set of all functions that have same *rate of growth* as g(*n*).

Theta Notation



Intuitively: Set of all functions that have same *rate of growth* as g(*n*).

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

$\exists\ c_1 > 0,\ c_2 > 0,\ \exists\ n_0 \geq 0,\ \forall\ n \geq n_0,\ c_{2.}g(n) \leq f(n) \leq c_{1.}g(n)$

***We say that g(n)* is an *asymptotically tight bound* for *f(n)*.**

Theta Notation

Example 1: Prove that $\frac{1}{2}.n^2 - \frac{1}{2}.n = \Theta(n^2)$

Proof

Assume that $f(n) = \frac{1}{2}.n^2 - \frac{1}{2}.n$, and $g(n) = n^2$

$f(n) \in \Theta(g(n))$?

We have to find the existence of c_1, c_2 and n_0 s.t.

$c_1.g(n) \leq f(n) \leq c_2.g(n) \quad A\ n \geq n_0$

Since, $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \ \forall n \geq 0$ if $c_2 = \frac{1}{2}$ and

$\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n . \frac{1}{2} n \ (\forall n \geq 2) = \frac{1}{4} n^2, \ c_1 = \frac{1}{4}$

Hence $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \leq \frac{1}{2} n^2 - \frac{1}{2} n$

$c_1.g(n) \leq f(n) \leq c_2.g(n) \quad \forall n \geq 2, c_1 = \frac{1}{4}, c_2 = \frac{1}{2}$

Hence $f(n) \in \Theta(g(n)) \Rightarrow \frac{1}{2}.n^2 - \frac{1}{2}.n = \Theta(n^2)$

Theta Notation

Example 1: Prove that $2.n^2 + 3.n + 6$ 📋 $\Theta(n^3)$

Proof: Let $f(n) = 2.n^2 + 3.n + 6$, and $g(n) = n^3$

we have to show that f(n) 📋 $\Theta(g(n))$

On contrary assume that $f(n) \in \Theta(g(n))$ i.e.

there exist some positive constants c_1, c_2 and n_0

such that: $c_1.g(n) \leq f(n) \leq c_2.g(n)$

$c_1.g(n) \leq f(n) \leq c_2.g(n) \Leftrightarrow c_1.n^3 \leq 2.n^2 + 3.n + 6 \leq c_2.\ n^3 \Leftrightarrow$

$c_1.n \leq 2 + 3/n + 6/n^2 \leq c_2.\ n \Rightarrow$

$c_1.n \leq 2 \leq c_2.\ n$, for large $n \Rightarrow$

$n \leq 2/c_1 \leq c_2/c_1.n$ which is not possible

Hence f(n) 📋 $\Theta(g(n)) \Rightarrow 2.n^2 + 3.n + 6$ 📋 $\Theta(n^3)$

Time Complexity

- Time complexity not only depends upon the input size, but also on the type of the input.

❖ **Best Case:** Not preferable. It represents lower bound.

❖ **Worst case** is usually used: It is an upper bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance.

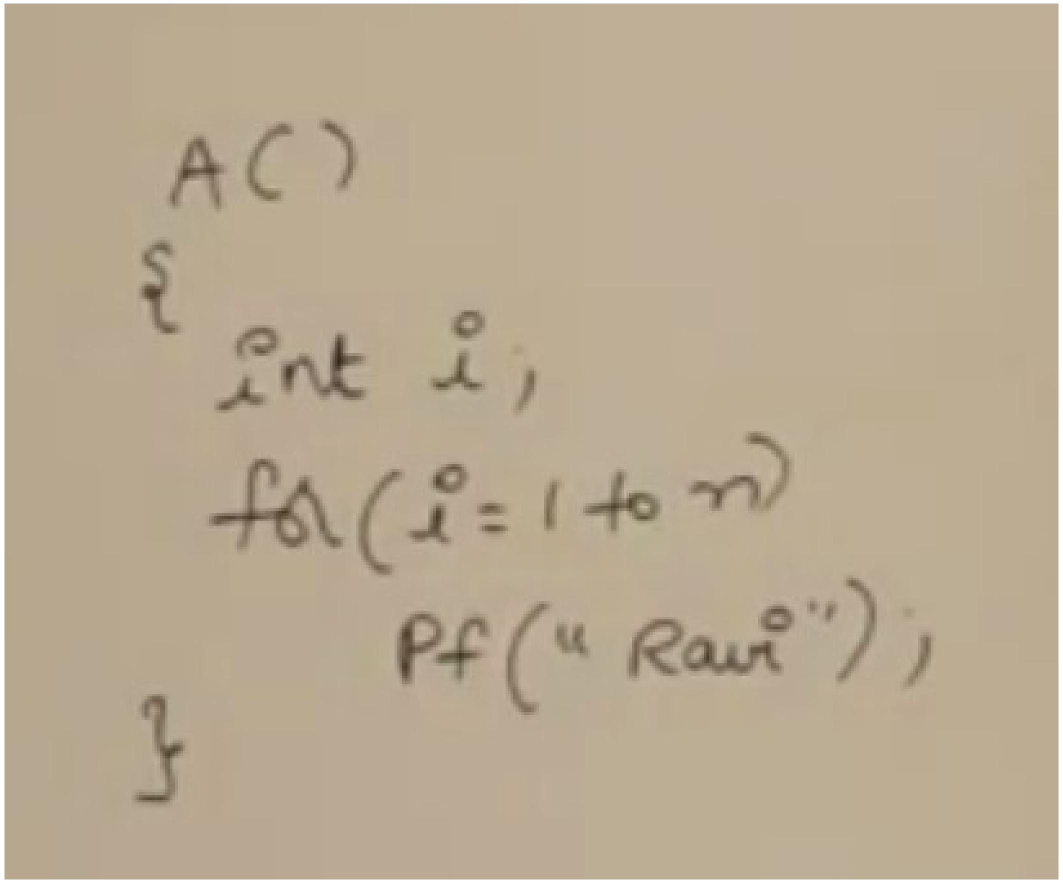
❖ For some algorithms **worst case** occurs fairly often.

❖ **Average Case:** Corresponds to complexities obtained by each possible combination of input and then dividing by those number of cases.

❖ **Average case** is often as bad as the **worst case.** Finding **average case** can be very difficult.

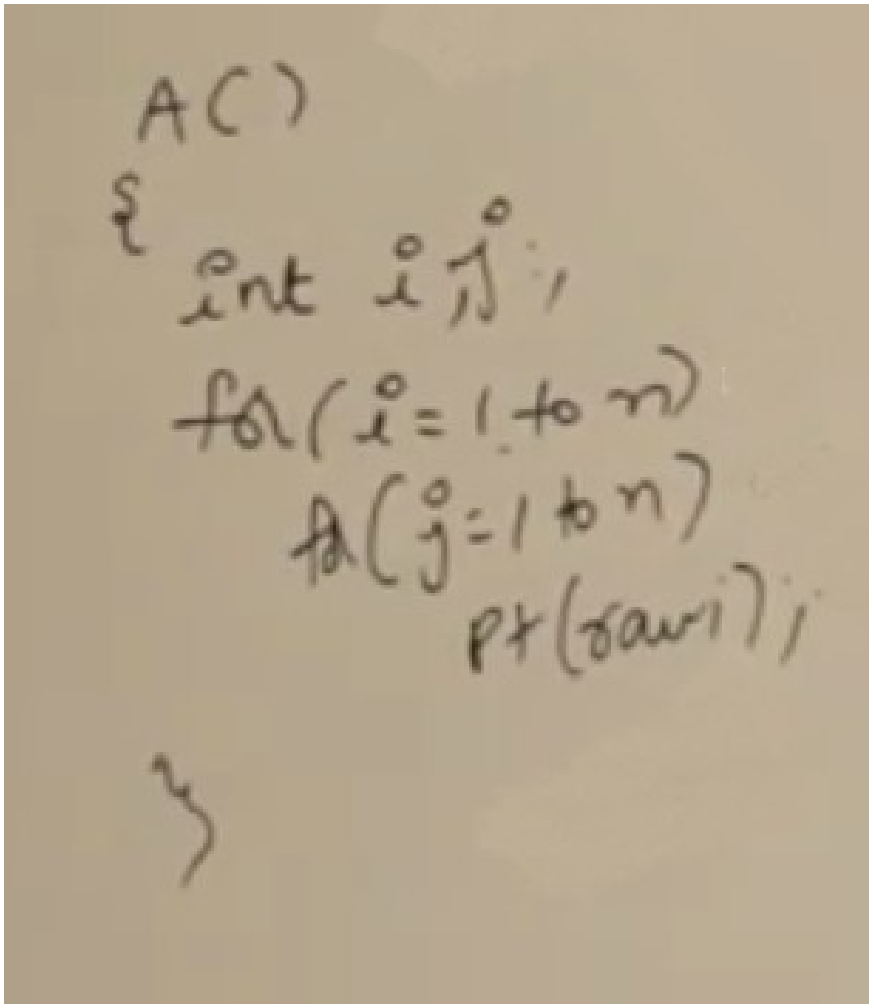
EXAMPLES

Example 1



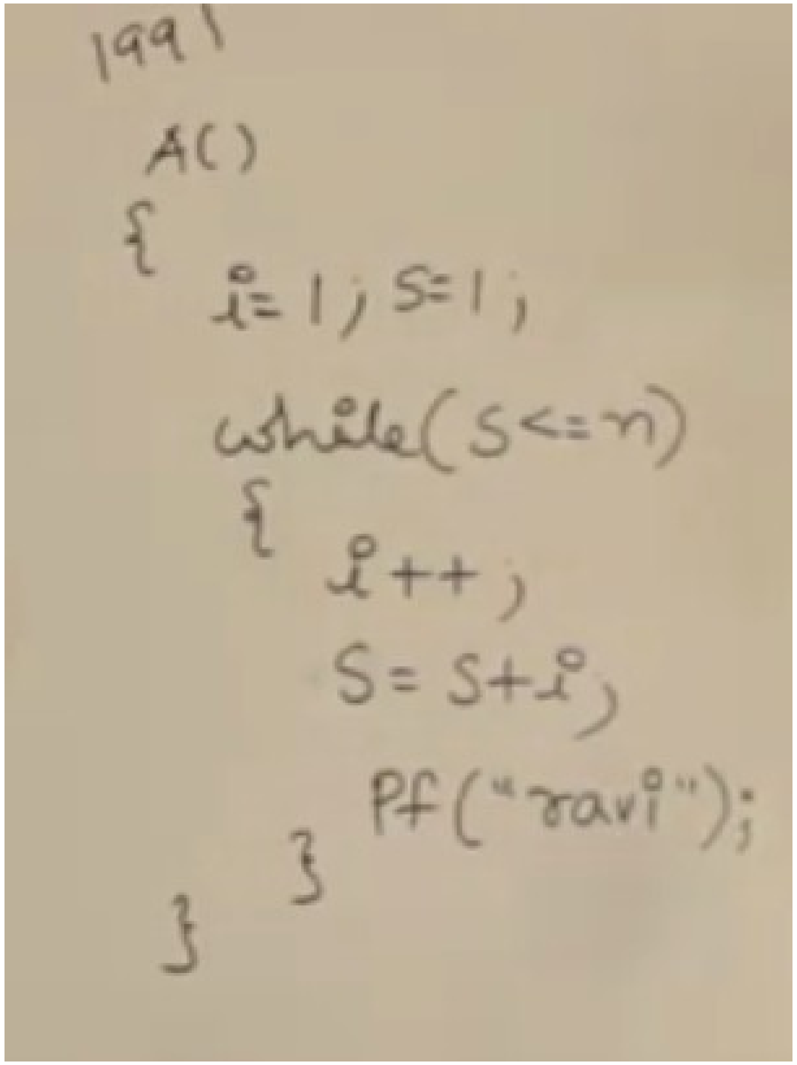
```
A()
{
  int i;
  for(i=1 to n)
      Pf("Ravi");
}
```

Example 2



```
A()
{
  int i,j;
  for(i=1 to n)
     for(j=1 to n)
        pf(ravi);
}
```

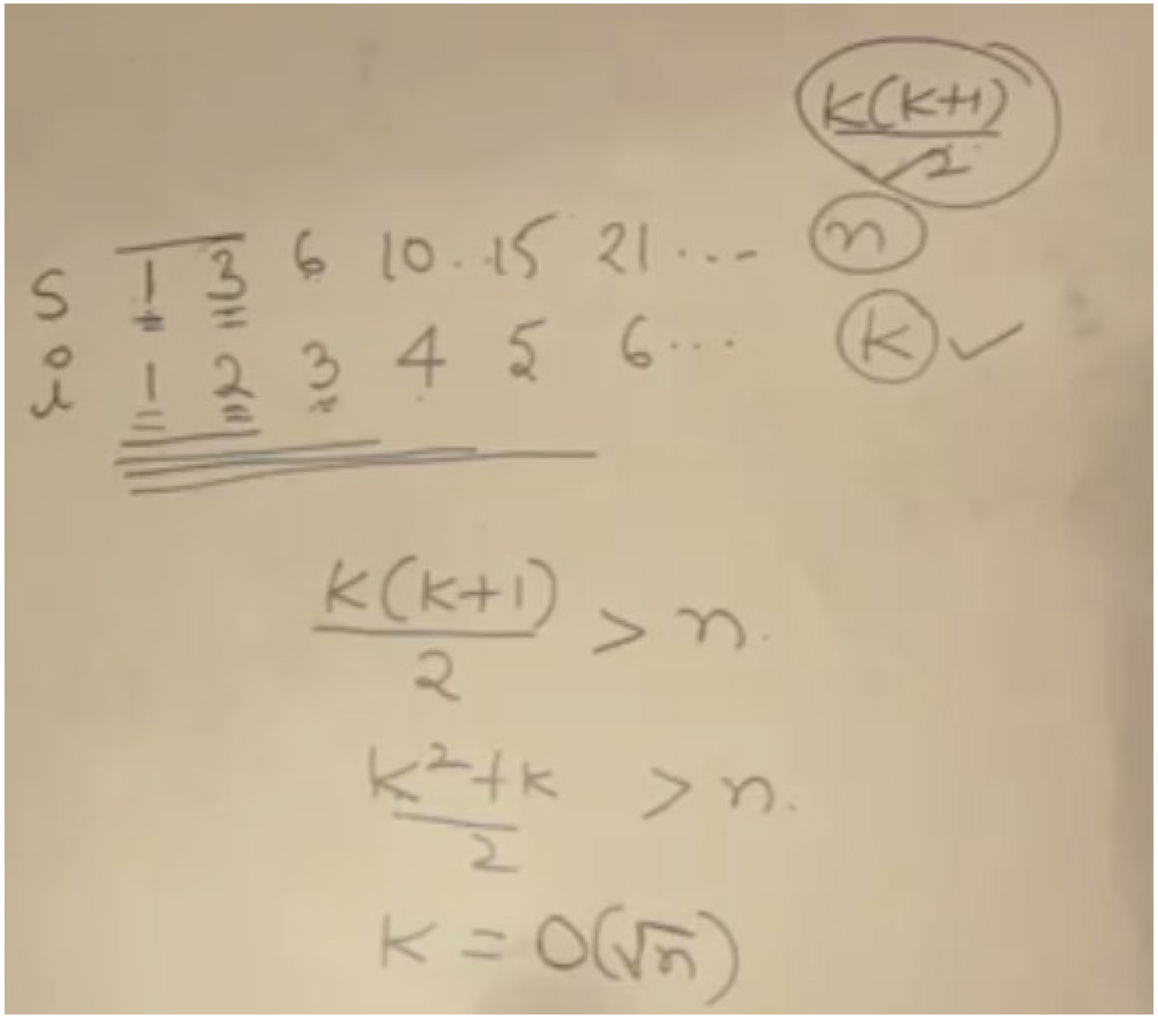
Example 3



```
1991

A()
{
    i=1; s=1;
    while(s<=n)
    {
        i++;
        s=s+i;
        Pf("ravi");
    }
}
```

Solution



$$\boxed{\frac{k(k+1)}{2}}$$

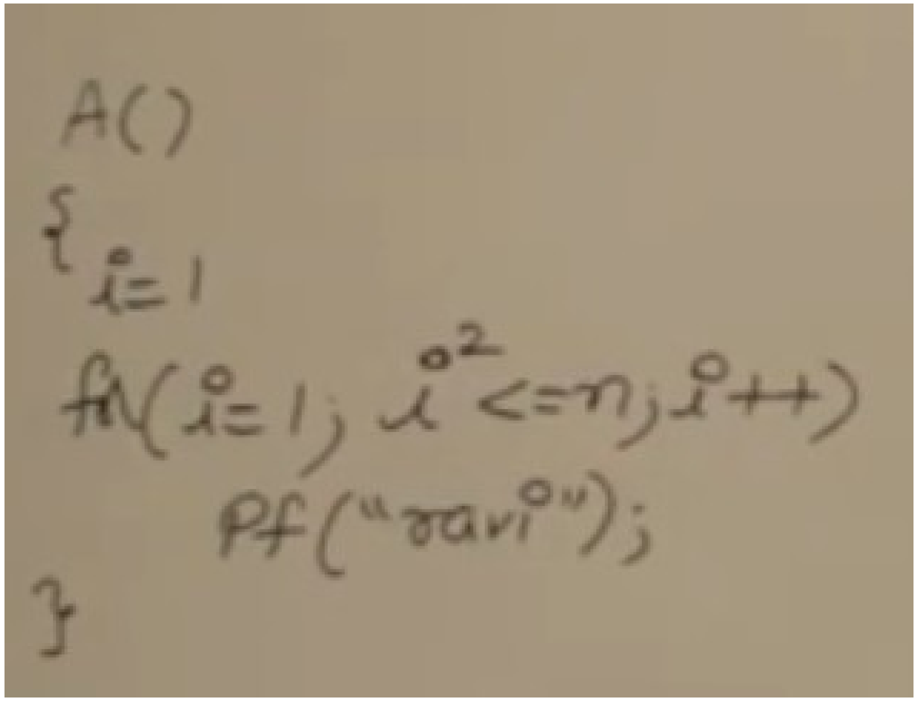
s	1	3	6	10	15	21	…	$\boxed{n}$
i	1	2	3	4	5	6	…	$\boxed{k}$ ✓

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

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

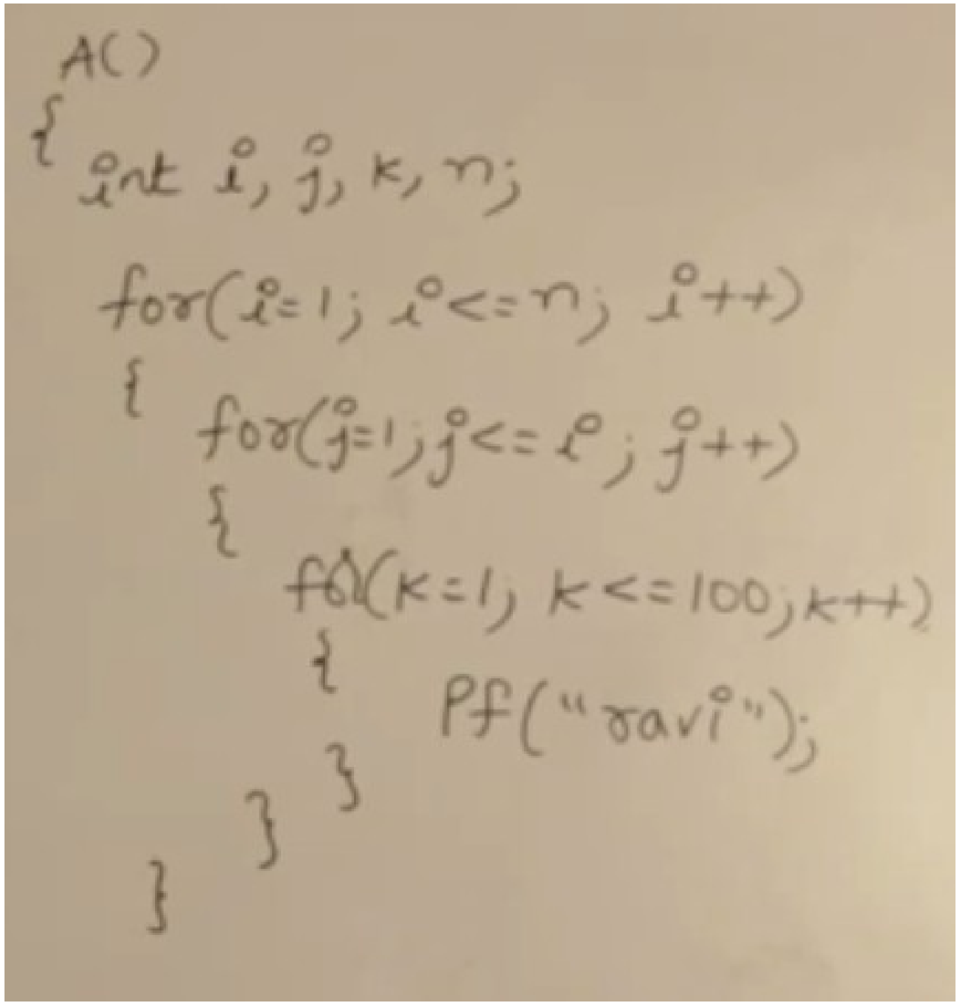
$$k = O(\sqrt{n})$$

Example 4



```
A()
{
  i=1
  for(i=1; i^2<=n; i++)
      pf("ravi");
}
```

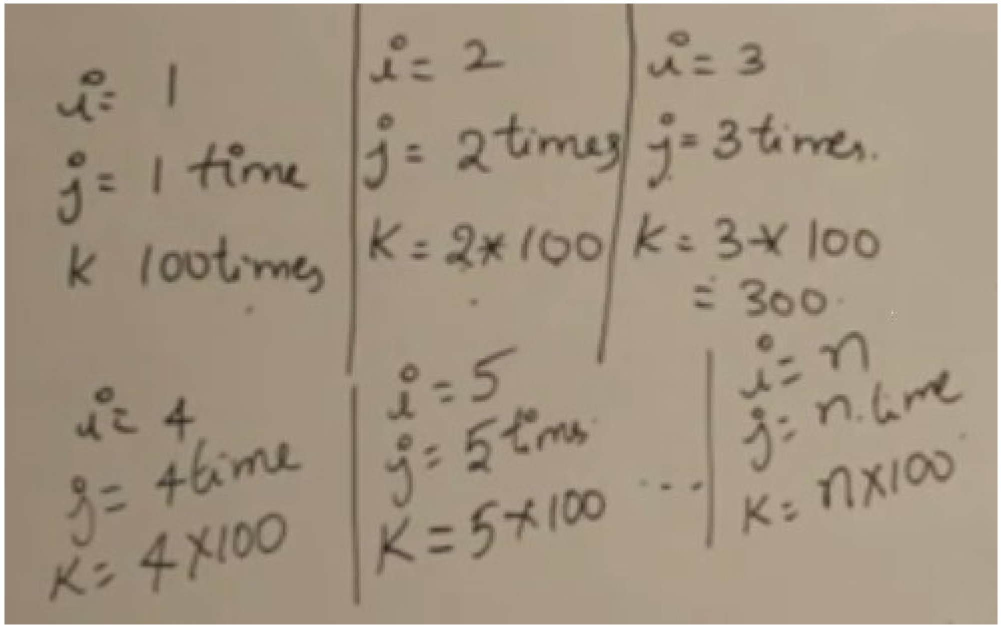
Example 5



```
A()
{
 int i, j, k, n;

  for(i=1; i<=n; i++)
  {
     for(j=1; j<=i; j++)
     {
        for(k=1, k<=100; k++)
        {
              Pf("ravi");
        }
     }
  }
```

Solution



i = 1
j = 1 time
k 100times

i = 2
j = 2 times
K = 2 * 100

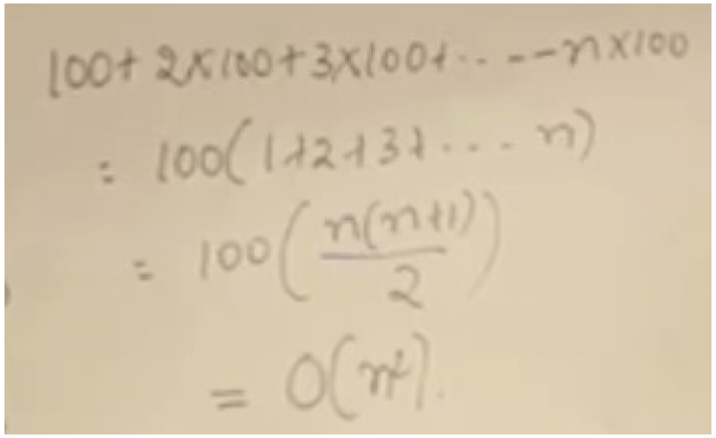
i = 3
j = 3 times.
k = 3 * 100
= 300.

i = 4
j = 4 time
K = 4 × 100

i = 5
j = 5 times
K = 5 * 100

. . .

i = n
j = n.time
K = n × 100



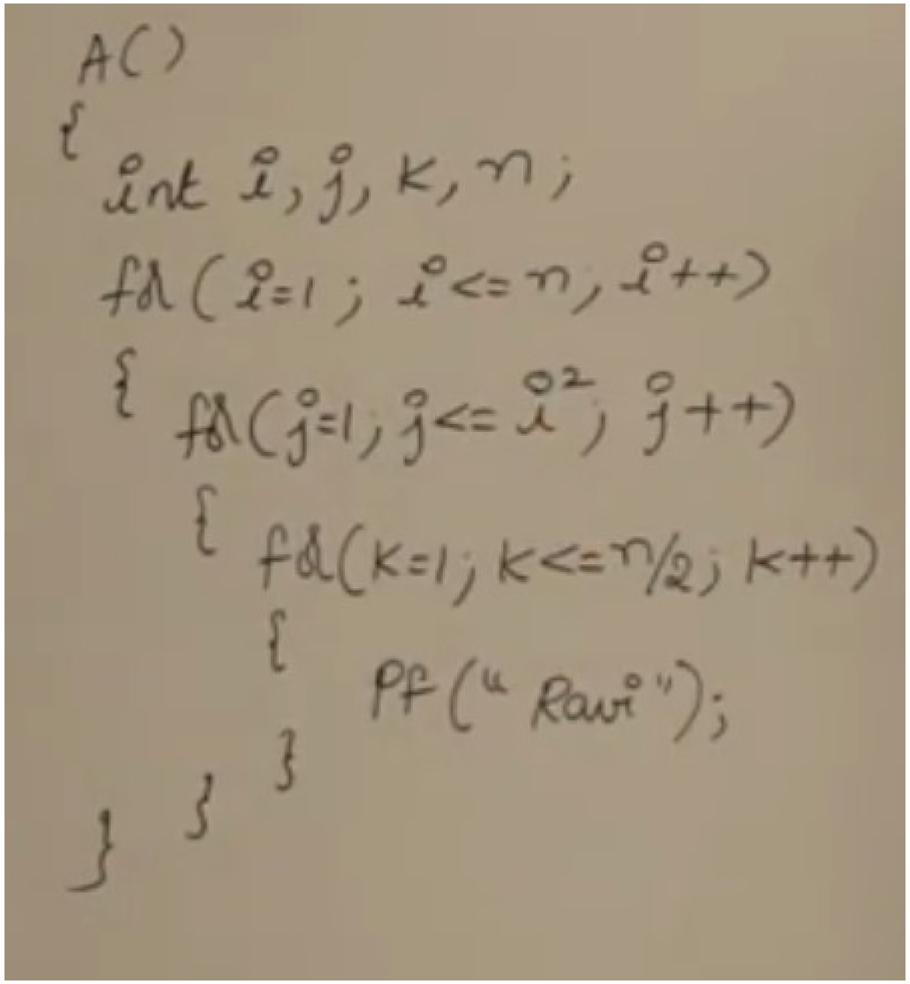
$100 + 2 \times 100 + 3 \times 100 + \cdots - - n \times 100$

$= 100(1 + 2 + 3 + \cdots n)$

$= 100\left(\frac{n(n+1)}{2}\right)$

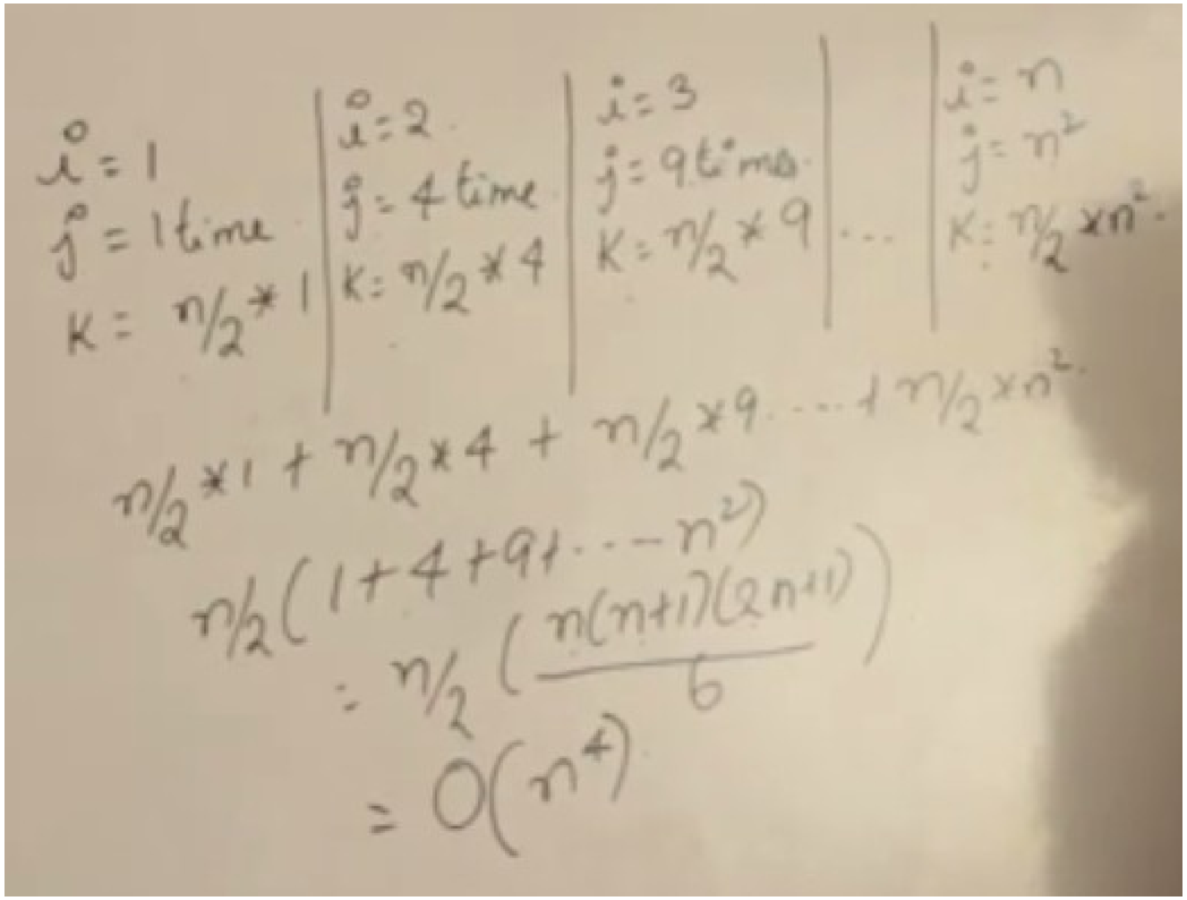
$= O(n^2)$

Example 6



```
A()
{
  int i, j, k, n;
  for(i=1; i<=n, i++)
  {
    for(j=1, j<= i^2, j++)
    {
      for(k=1; k<=n/2; k++)
      {
        pf("Ravi");
      }
    }
}
```

Solution



$i=1$	$i=2$	$i=3$		$i=n$
$j=1$ time	$j=4$ time	$j=9$ times		$j=n^2$
$K=n/2 * 1$	$K=n/2 * 4$	$K=n/2 * 9$	...	$K=n/2 \times n^2$

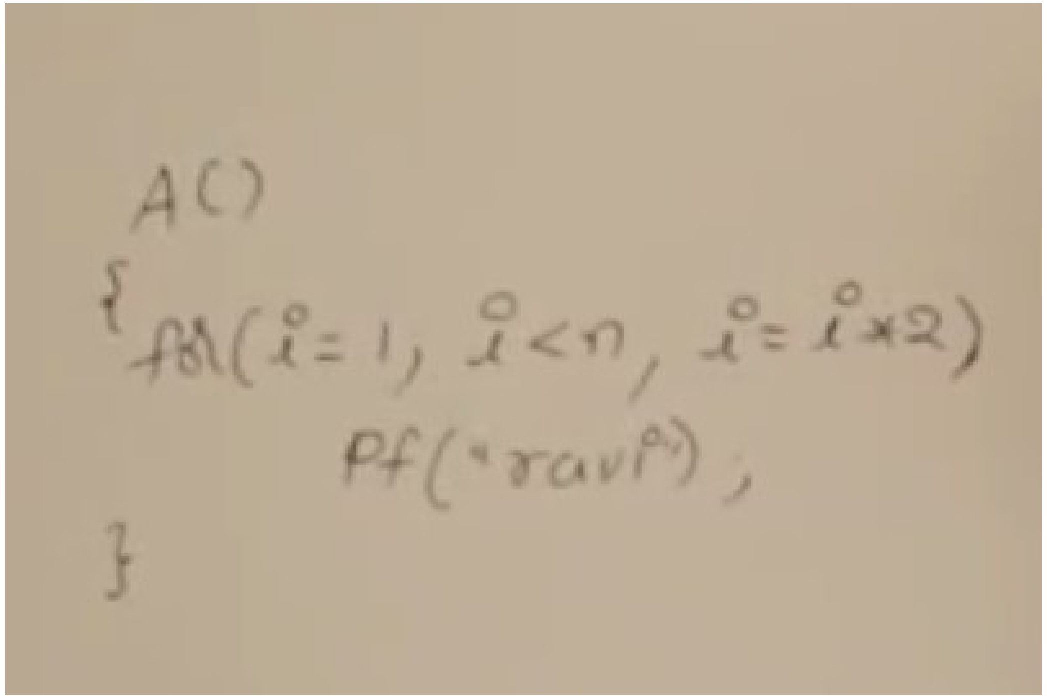
$$n/2 * 1 + n/2 * 4 + n/2 * 9 \ldots + n/2 \times n^2$$

$$n/2(1+4+9+\ldots n^2)$$

$$= n/2\left(\frac{n(n+1)(2n+1)}{6}\right)$$

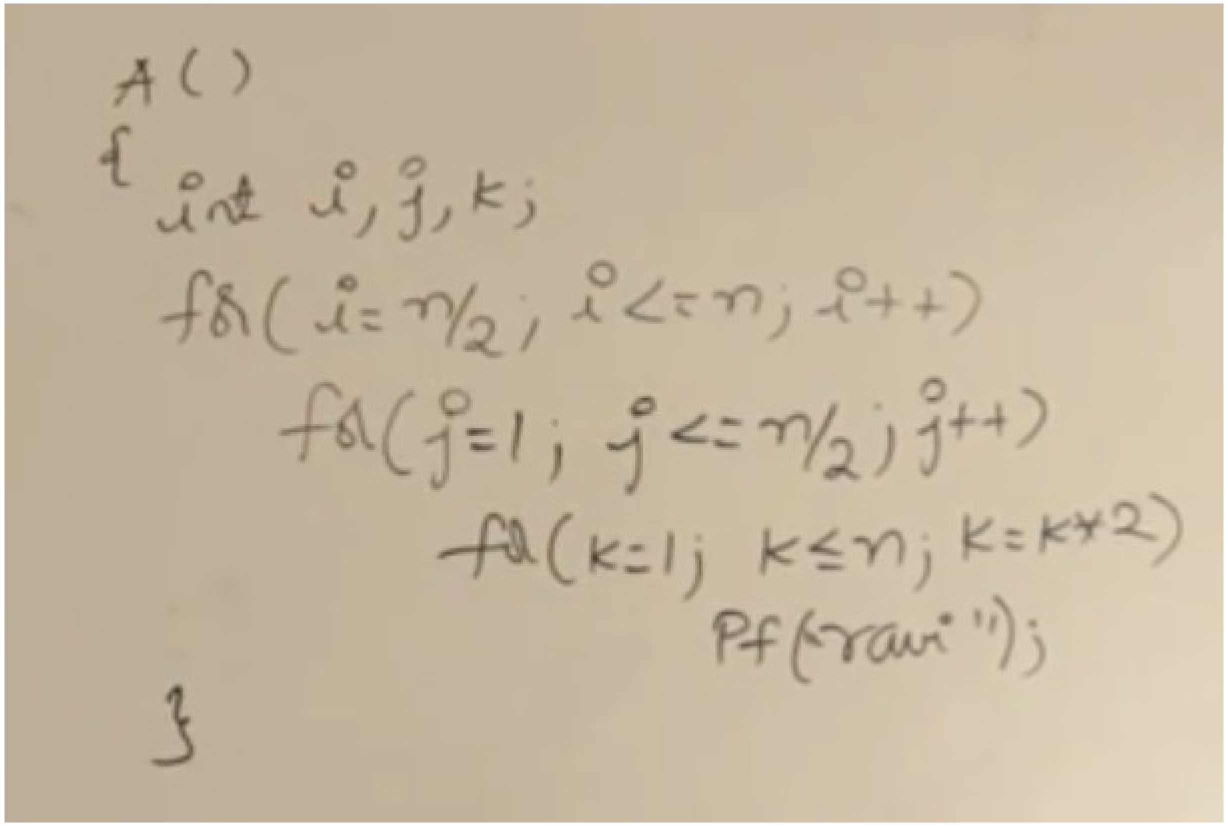
$$= O(n^4)$$

Example 7



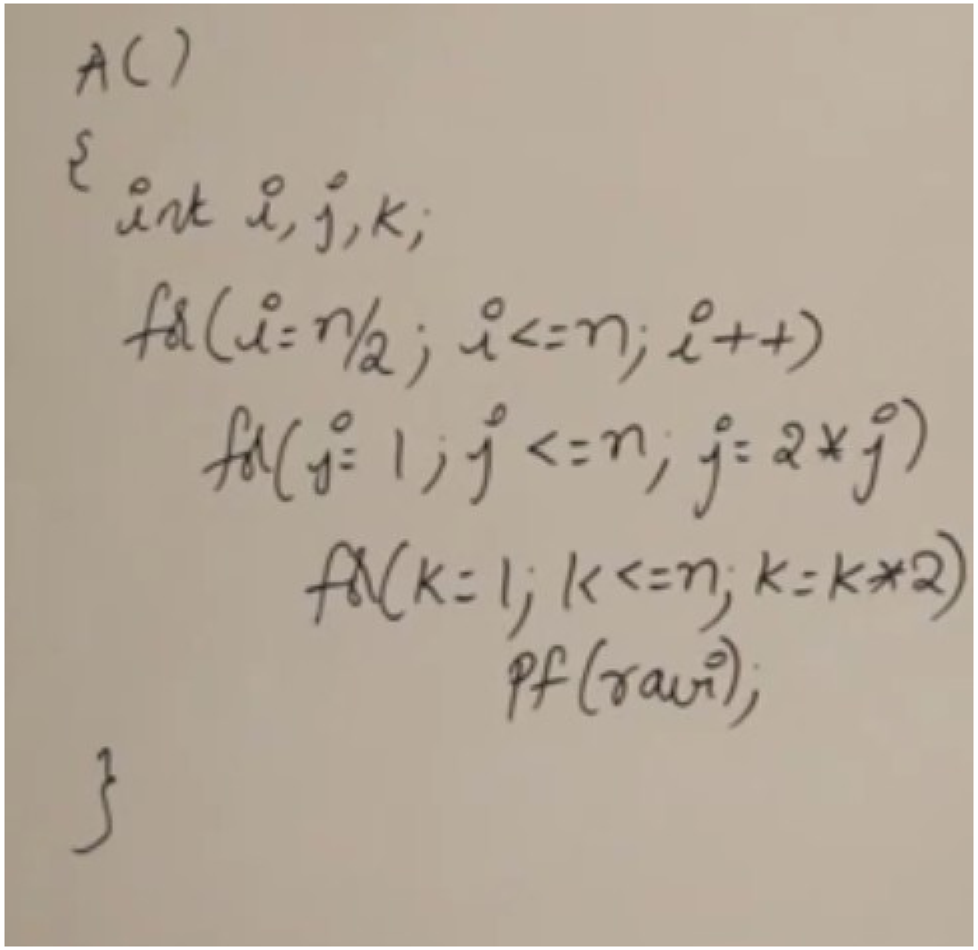
```
A()
{
  for(i=1, i<n, i=i*2)
        pf("ravi");
}
```

Example 8



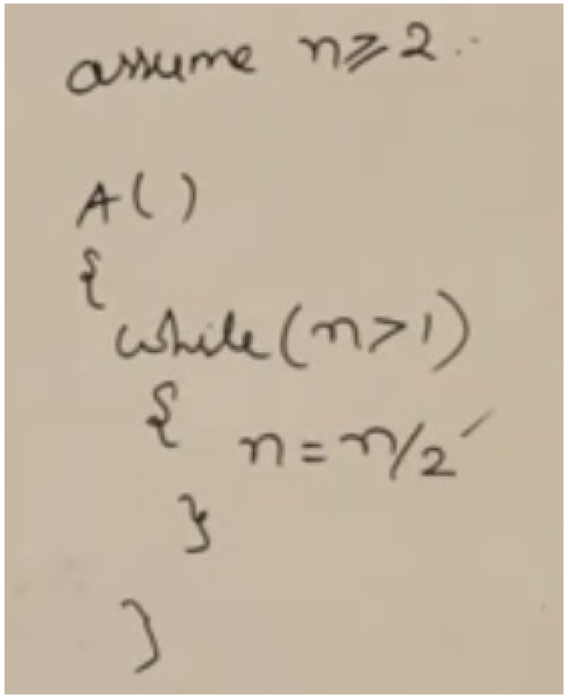
```
A()
{
 int i,j,k;
  for(i=n/2; i<=n; i++)
      for(j=1; j<=n/2; j++)
          for(k=1; k≤n; k=k*2)
               Pf("ravi");
 }
```

Example 9



```
A()
{
  int i, j, k;
   for(i=n/2; i<=n; i++)
      for(j=1; j<=n; j=2*j)
          for(k=1; k<=n; k=k*2)
               pf(ravi);
}
```

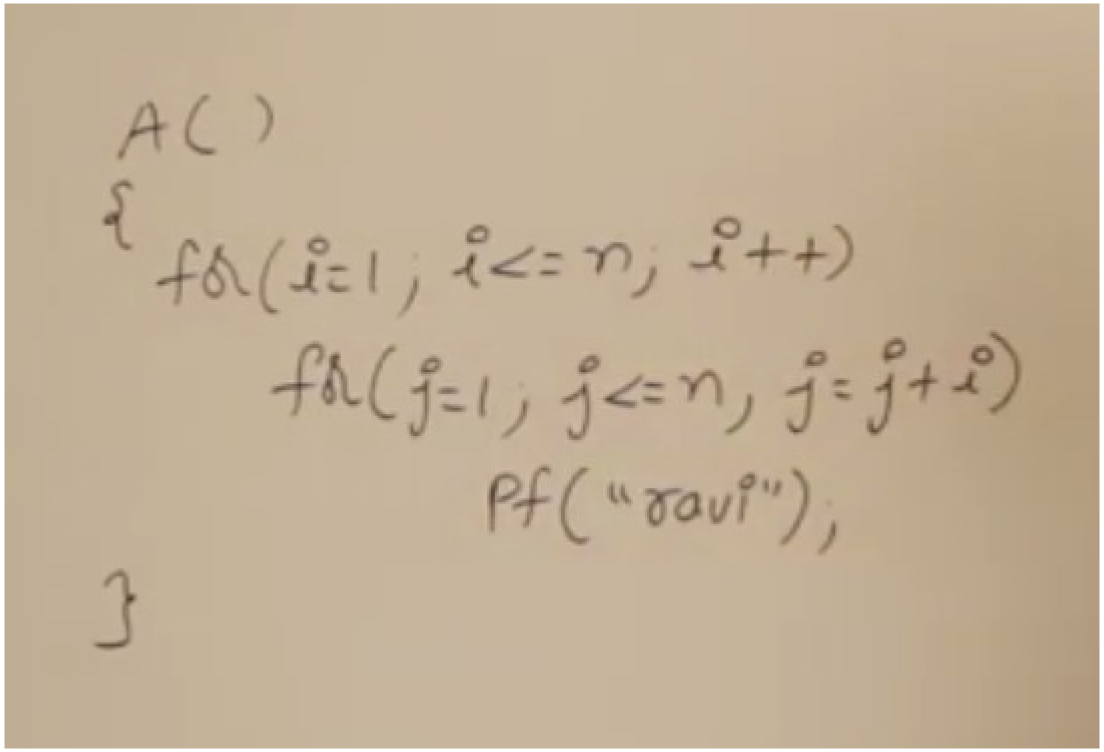
Example 10



assume $n \geq 2$.

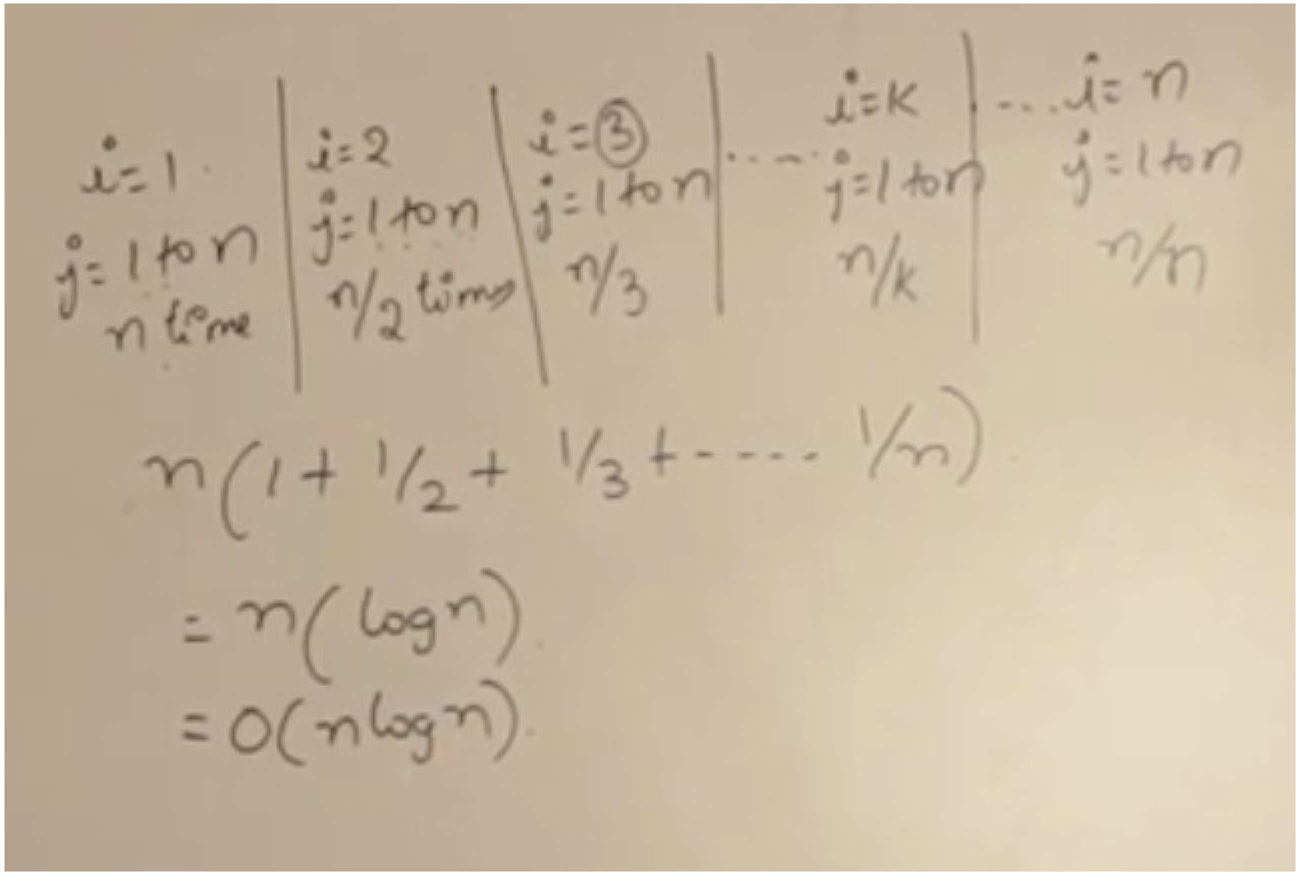
```
A()
{
  while(n>1)
   {  n=n/2
   }
}
```

Example 11



```
A( )
{
  for(i=1; i<=n; i++)
      for(j=1; j<=n, j=j+i)
          Pf("ravi");
}
```

Solution



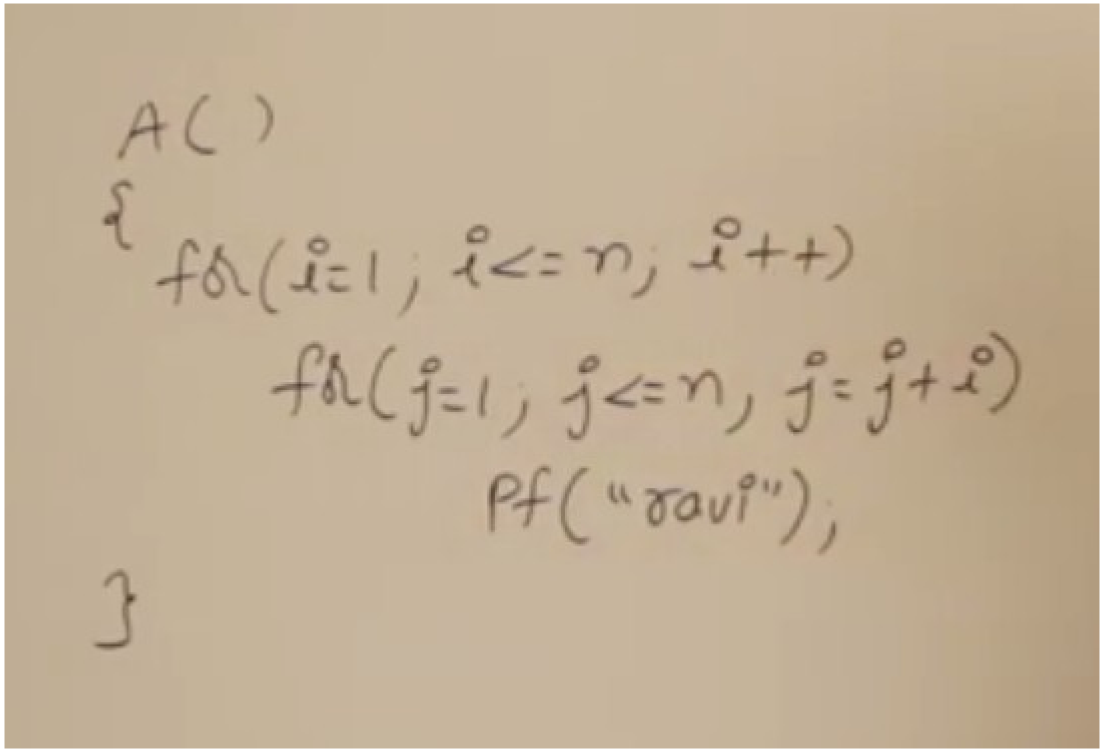
$i=1$	$i=2$	$i=\boxed{3}$	....	$i=k$	....	$i=n$
$j=1$ to n	$j=1$ to n	$j=1$ to n		$j=1$ to n		$j=1$ to n
n time	$n/2$ times	$n/3$		n/k		n/n

$$n\left(1+\frac{1}{2}+\frac{1}{3}+\cdots\frac{1}{n}\right)$$

$$= n(\log n)$$

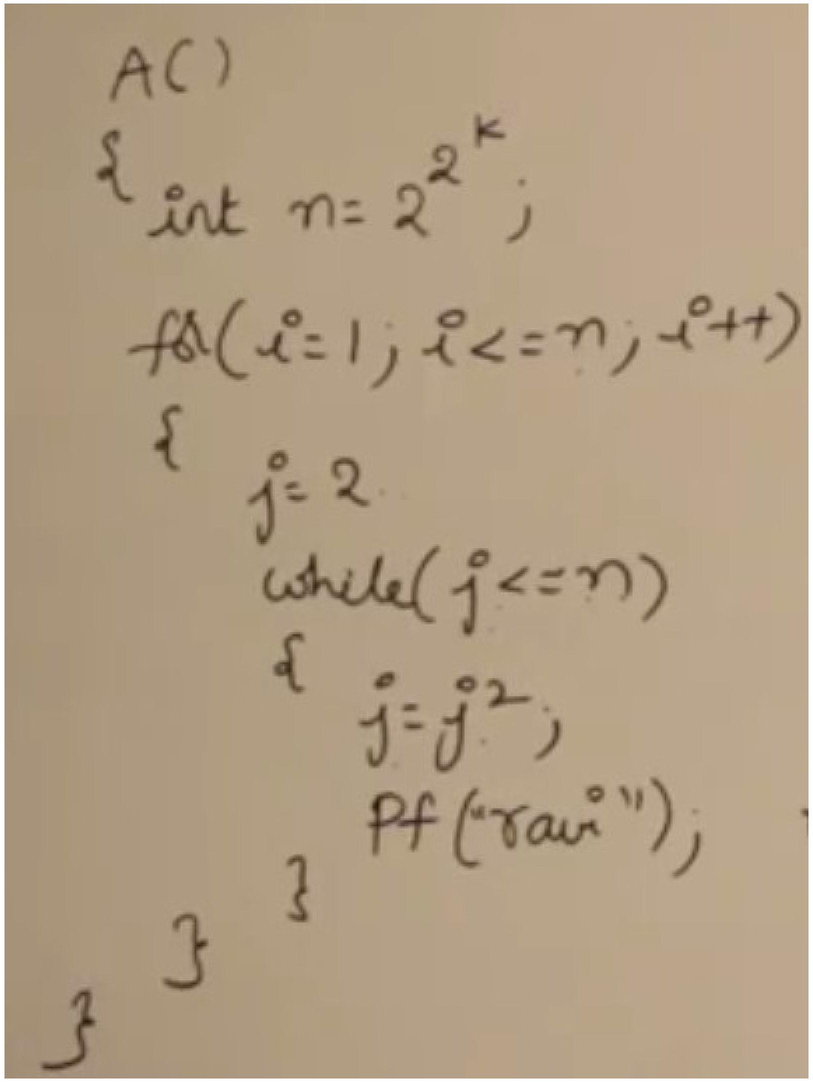
$$= O(n\log n)$$

Example 12



```
A( )
{
 for(i=1; i<=n; i++)
     for(j=1; j<=n, j=j+i)
          Pf("ravi");
}
```

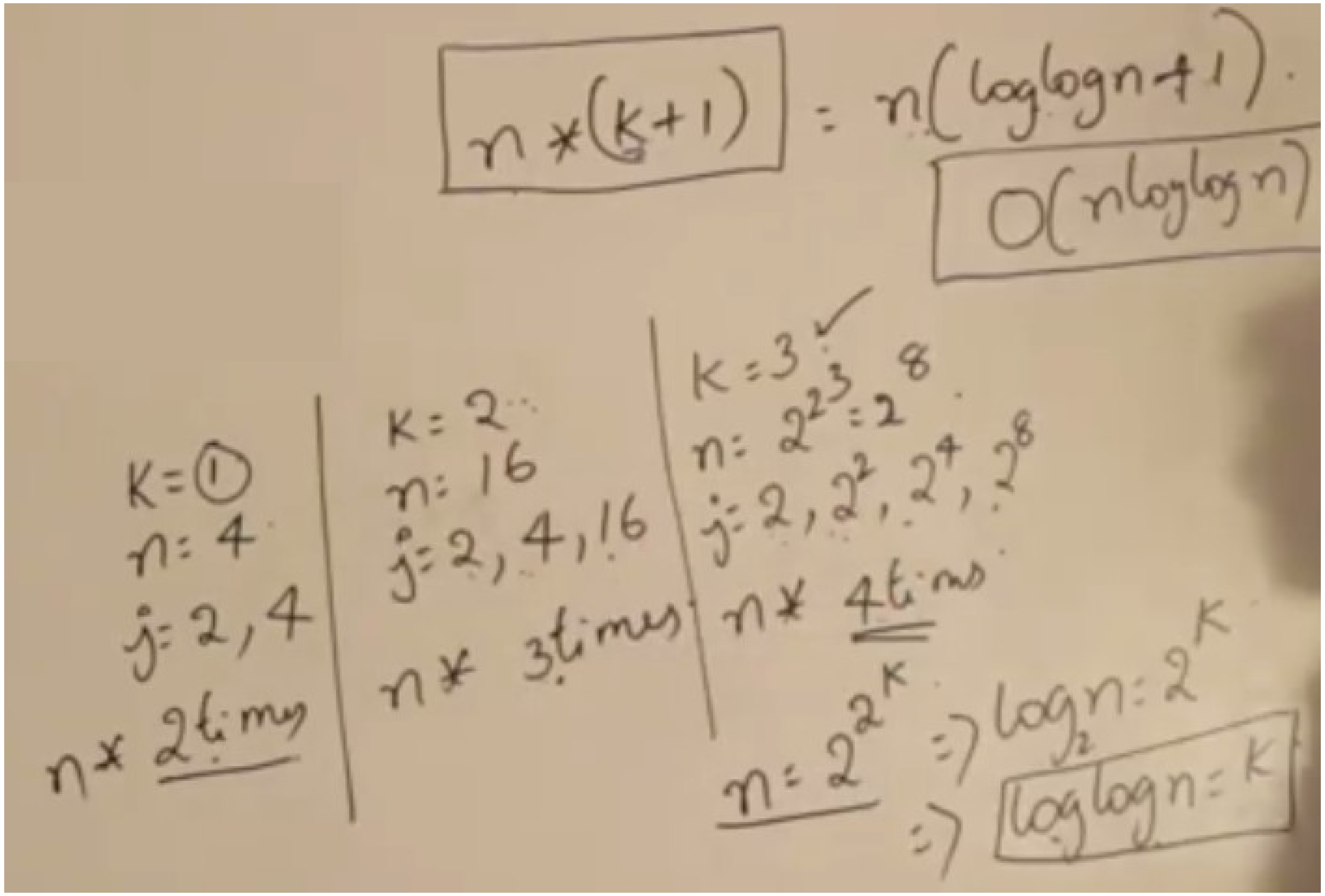
Example 13



```
A()
{ int n = 2^2^k;

  for(i=1; i<=n; i++)
  {
      j=2
      while(j<=n)
      {
          j=j^2;
          pf("ravi");
      }
  }
}
```

Solution



$\boxed{n * (k+1)} = n(\log\log n + 1)$.

$\boxed{O(n \log\log n)}$

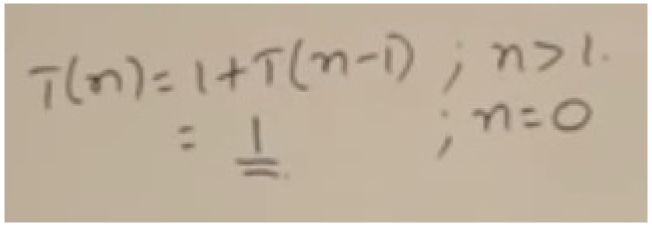
$k = 1$
$n = 4$
$j = 2, 4$
$n *$ 2 times

$k = 2$
$n = 16$
$j = 2, 4, 16$
$n *$ 3 times

$k = 3$ ✓
$n = 2^{2^3} = 2^8$
$j = 2, 2^2, 2^4, 2^8$
$n *$ 4 times

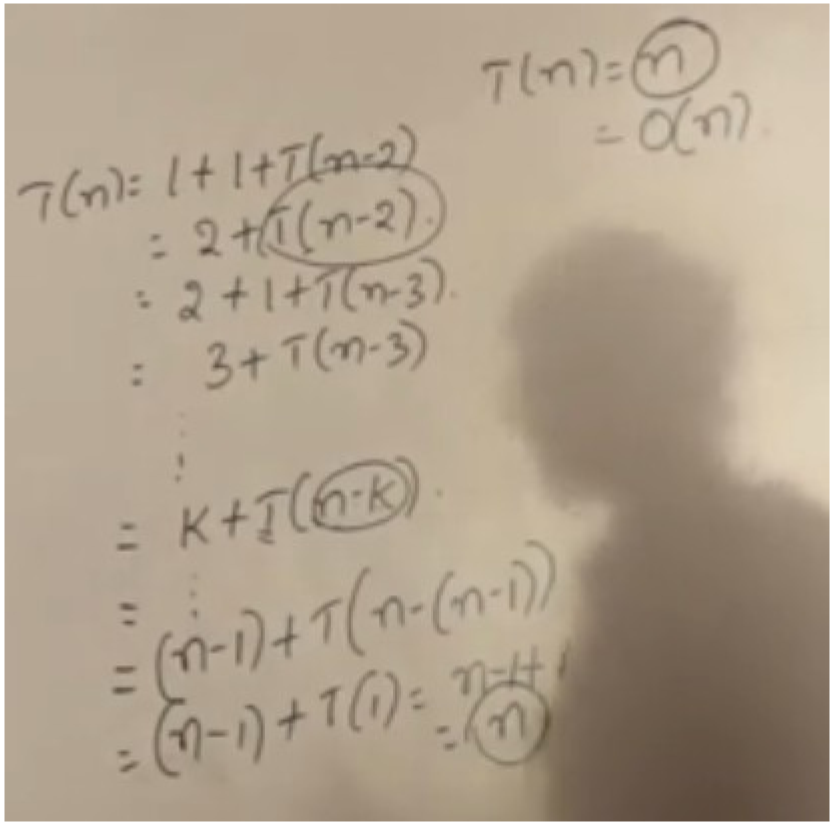
$n = 2^{2^k} \Rightarrow \log_2 n = 2^k$
$\Rightarrow \boxed{\log\log n = k}$

Example 14



$$T(n) = 1 + T(n-1) \quad ; n > 1.$$
$$= \underline{\underline{1}} \quad ; n = 0$$

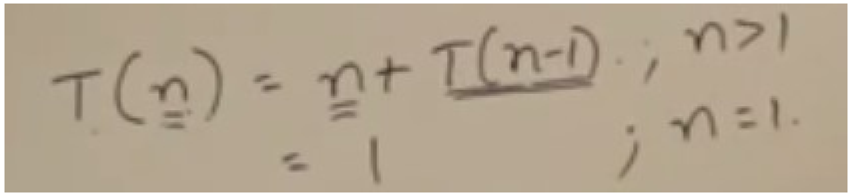
Solution



$$
\begin{aligned}
T(n) &= 1 + 1 + T(n-2) \\
&= 2 + \boxed{T(n-2)} \\
&= 2 + 1 + T(n-3) \\
&= 3 + T(n-3) \\
&\;\vdots \\
&= K + T(\boxed{n-K}) \\
&= \;\vdots \\
&= (n-1) + T(n-(n-1)) \\
&= (n-1) + T(1) = n-1+1 \\
&= \boxed{n}
\end{aligned}
$$

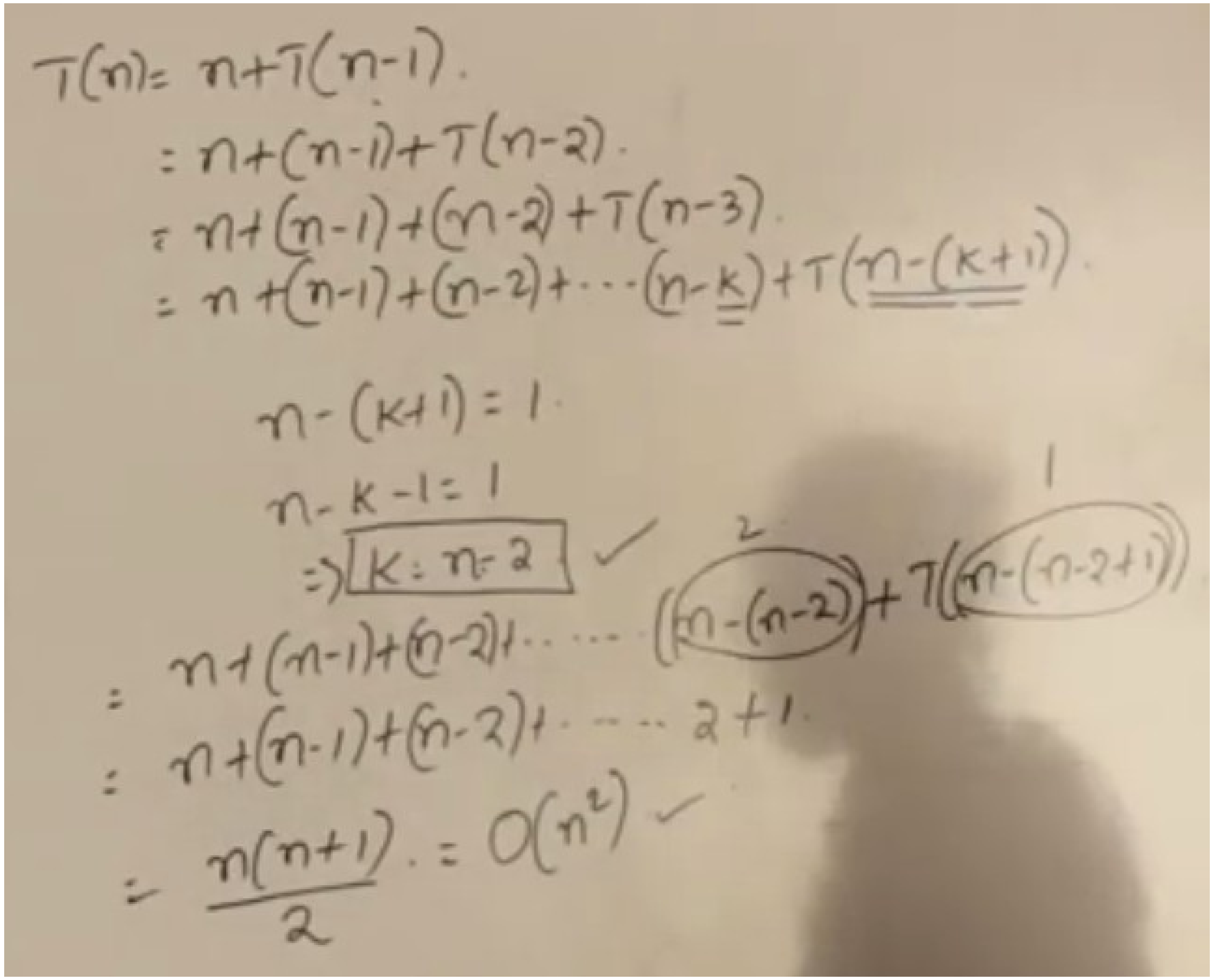
$$
\begin{aligned}
T(n) &= \boxed{n} \\
&= O(n)
\end{aligned}
$$

Example 15



$$T(\underline{\underline{n}}) = \underline{\underline{n}} + \underline{T(n-1)} \; ; \; n>1$$
$$= 1 \; ; \; n=1.$$

Solution



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

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

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

$= n + (n-1) + (n-2) + \cdots (n-\underline{\underline{k}}) + T(\underline{\underline{n-(k+1)}}).$

$n-(k+1) = 1.$

$n-k-1 = 1$

$\Rightarrow \boxed{k = n-2}$ ✓

$= n + (n-1) + (n-2) + \cdots\cdots (\overset{2}{(n-(n-2))}) + T(\overset{1}{(n-(n-2+1))})$

$= n + (n-1) + (n-2) + \cdots\cdots 2 + 1.$

$= \frac{n(n+1)}{2} = O(n^2)$ ✓