Design and Analysis of Algorithms

- ***Course Code***: BCSE304L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 1

Module:1	Design Paradigms: Greedy, Divide and Conquer Techniques	6 hours

Overview and Importance of Algorithms - Stages of algorithm development: Describing the problem, Identifying a suitable technique, Design of an algorithm, Derive Time Complexity, Proof of Correctness of the algorithm, Illustration of Design Stages -

Greedy techniques: Fractional Knapsack Problem, and Huffman coding

Divide and Conquer: Maximum Subarray, Karatsuba faster integer multiplication algorithm.

Stages of algorithm development

- Describing the problem (Problem Definition),
- Identifying a suitable technique
- Design of an algorithm (Design and Implementation)
- Derive Time Complexity (Optimization),
- Proof of Correctness of the algorithm,
- 6)Illustration of Design Stages

Stages of algorithm development

Derive Time Complexity (analysis)

- Analyze and estimate the computational efficiency of the algorithm.
- Determine the time complexity by analyzing the number of operations as a function of input size.
- Evaluate best-case, average-case, and worst-case scenarios.
- Consider Big O notation or other complexity measures.

Analyse Algorithms or performance analysis

To **analyze an algorithm** is to determine the amount of resources (such as time and storage) necessary to execute it.

If an algorithm is executed, it used the

- Computer's CPU → to perform the operations.
- Memory (both RAM and ROM) → to hold the program & data.

Analysis of algorithms is the task of determining how much computing time and storage memory required for an algorithm.

Analyse Algorithms or performance analysis

Space Complexity:

- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

- The space requirement S(P) of any algorithm P can be written as
  - $S(P)=C+ S_P$

- C→ Constant
- S_P→ Instance characteristics.

Find space complexity of Algorithm of sum of ‘n’ numbers.

```
Algorithm sum(a, n)
        //Here a is array of Size n
        {
        S:=0;
        if(n≤0)) then return 0;
        for i:=1 to n do
                s:=s+a[i];
        return s;
        }
```

Space complexity S(P):

s→1 word
i→1 word
n→1 word
a→n word

$$S(P) \geq n+3$$

Word is a collection of bits stored in computer memory, its size is a 4 to 64 bits.

Memory Units:

- Bit (Binary Digit): A binary digit is logical 0 and 1 representing a passive or an active state of a component in an electric circuit.
- Nibble : A group of 4 bits is called nibble
- Byte : A group of 8 bits is called byte

- **Word** : A computer word, like a byte, is a group of fixed number of bits processed as a unit, which varies from computer to computer but is fixed for each computer.

- Bit (Binary Digit) : A binary digit is logical 0 and 1
- Nibble : A group of 4 bits is called nibble
- Byte : A group of 8 bits is called byte
- Kilobyte (KB) : 1 KB = 1024 Bytes
- Megabyte (MB) : 1 MB = 1024 KB
- GigaByte (GB) : 1 GB = 1024 MB
- TeraByte (TB) : 1 TB = 1024 GB
- PetaByte (PB) : 1 PB = 1024 TB
- Exa Byte (EB) : 1 EB = 1024 PB
- Zeta Byte(ZB) : 1 ZB = 1024 EB
- Yotta Byte(YB) : 1 YB = 1024 ZB
- Bronto Byte : 1 Bronto Byte= 1024 YB
- Geop Byte : 1 Geop Byte = 1024 Bronto Byte

Analyse Algorithms or performance analysis

Time Complexity:

- Time complexity of an algorithm is the amount of computer time it needs to run to completion.

➢ Here RUN means Compile + Execution.

➢ Time Complexity of a algorithm P can be written as

  - ➢ $T(P)= t_P+ t_c$

- neglecting t_c, because
  - The compile time does not depends on the instance characteristics.
  - The compiled program will be run several times without recompilation.
- $T(P)= t_p$
- Here t_p → instance characteristics.

Performance of a computer (optional)

- The performance of a computer is determined by:
- The hardware:
  - processor used (type and speed).
  - memory available (cache and RAM).
  - disk available.
- The programming language in which the algorithm is specified.
- The language compiler/interpreter used.
- The computer operating system software.

Factors that influence the running time of an Algorithm (Optional)

- The running time of an algorithm or data structure method increases with the input size, although it may also vary for different inputs of the same size.
- The running time is affected by the
  - hardware environment (as reflected in the processor, clock rate, memory, disk, etc.) and
  - software environment (as reflected in the operating system, programming language, compiler, interpreter, etc.)

  in which the algorithm is implemented, compiled, and executed.
- All other factors being equal, the running time of the same algorithm on the same input data is smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation run on a virtual machine.
- Like to focus on the relationship between the running time of an algorithm and the size of its input. But what is the proper way of measuring it?

But what can we analyze?

- Determine the running time of a program as a function of its inputs.
- Determine the total or maximum memory space needed for program data.
- Determine the total size of the program code.
- Determine whether the program correctly computes the desired result.
- Determine the complexity of the program- e.g., how easy is it to read, understand, and modify.
- Determine the robustness of the program- e.g., how well does it deal with unexpected or erroneous inputs?

Analyzing the running times of Algorithms

Takes into account all possible inputs.

- Allows us to evaluate the relative efficiency of any two algorithms in a way that is
  - Independent from the hardware and software environment.
  - Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

Asymptotic Notation:

- A problem may have numerous (many) algorithmic solutions.
  - In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run.
- Asymptotic notation is used to judge the best algorithm among numerous algorithms for a particular problem.
- Asymptotic complexity is a way of expressing the main component of algorithms like
  - Time complexity
  - Space complexity

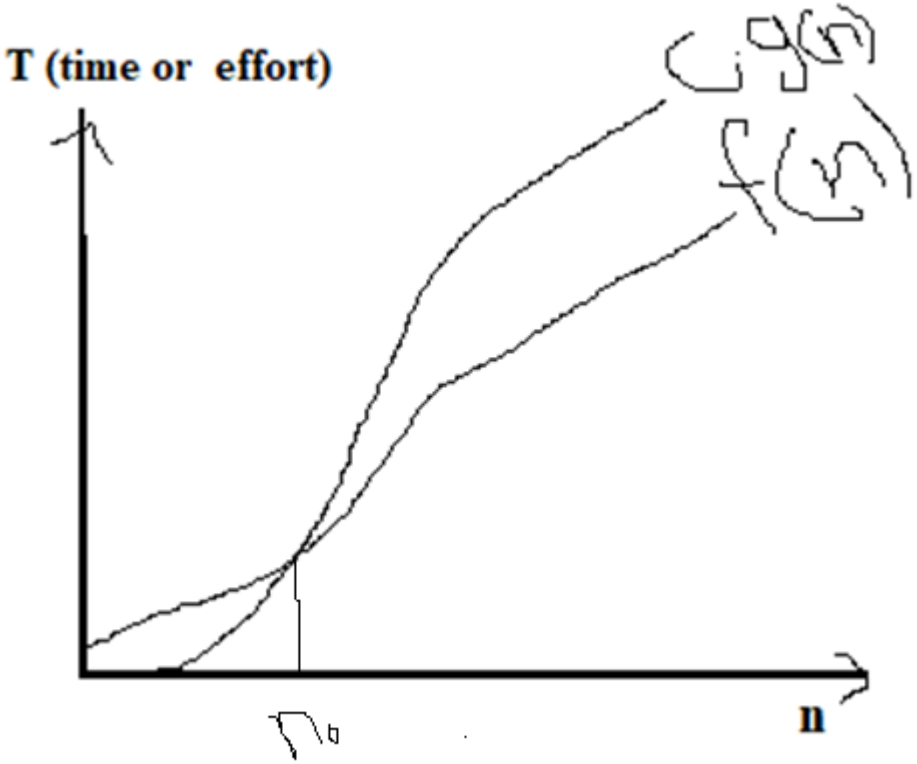
Important Note: The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants.

Some Asymptotic notations are

- Big oh → O
- Omega → Ω
- Theta → θ
- Little oh →o
- Little Omega → ω

Big oh notation: O

- Computational complexity (time or memory) of an algorithm is the amount of resources required to run it.
- As the amount of resources required to run an algorithm generally varies with the size of the input.
  - The complexity is typically expressed as a function f(n), where n is the size of the input.



If there is any **upper bound** that function by some other function g(n) after some input n0, then

$f(n) \leq C * g(n)$ for all $n \geq n0$, $n0 \geq 1$ and $c>0$

then it can be written as **$f(n)=O(g(n))$**

(read as "f of n is big oh of g of n")

Big oh notation: O

In Big oh O, f(n)≤c*g(n), what is f(n), c and g(n)? explain about each parameter clearly?

In the Big O notation (O notation), the expression "**f(n) ≤ c * g(n)**" represents a way to formally describe the upper bound or worst-case time complexity of an algorithm

- **f(n)**: This represents the actual time complexity function of the algorithm, which is a mathematical function that describes how the algorithm's runtime grows concerning the size of the input (n). In simple terms, it quantifies the number of basic operations or steps the algorithm takes to solve a problem as a function of the input size.

- **g(n)**: This is another mathematical function, typically chosen as a simple and easily understandable function that represents an upper bound or an approximation of the actual time complexity. It is often a simplified version of f(n) that helps express the upper limit of the algorithm's time complexity.

- **c**: 'c' is a positive constant, and it represents a scaling factor. This constant is used to adjust and scale the growth rate of the function g(n) to match the function f(n) in the worst-case scenario. It indicates that, after a certain input size, the function f(n) does not grow faster than c times g(n).

In the context of Big O notation, the expression "**f(n) ≤ c * g(n)**" means that for sufficiently large values of 'n' (i.e., beyond a certain input size), the actual time complexity of the algorithm, represented by f(n), is bounded from above by a constant multiple of g(n).

Big oh notation: O

f(n)=3n+2

f(n)=O(g(n)) if f(n)<=c.g(n)

3n+2<=c.n This can be true if c=4

3n+2<=4n

2<=4n-3n

n>=2

From this n is bound in function for c=4 and n>=2

g(n) is can be : n, n2, n3, 2^n, n! (where n=2,3,...)

therefore, the f(n)=3n+2 can be written in case of big oh notation as 3n+2=O(n)

Note: Consider least upper bound.

The Big O notation for the function $f(n)=3n+2$ is $O(n)$.

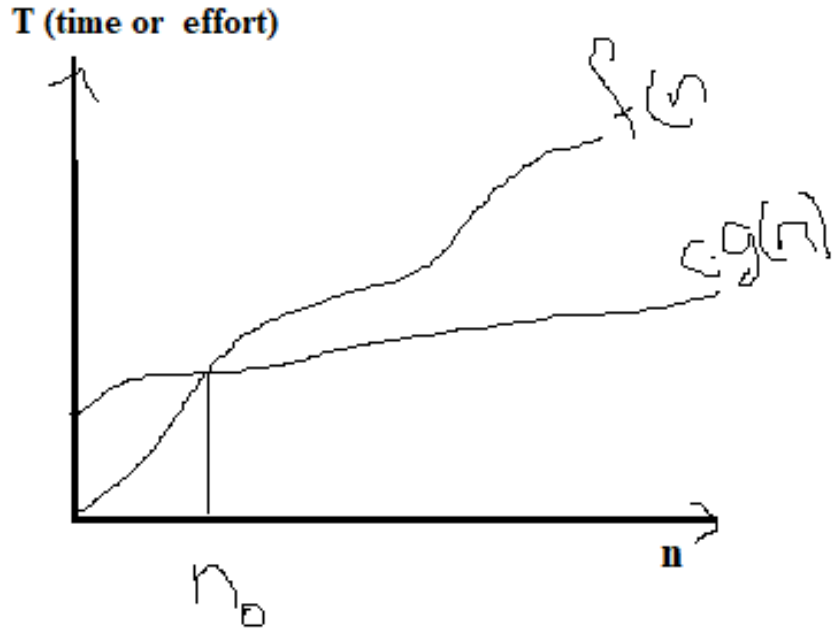
In Big O notation, we are concerned with the dominant term that grows the fastest as 'n' (the input size) increases.
In this case, the term $3n$ dominates, and the constant term 2 is insignificant when compared to the linear term $3n$ as 'n' becomes large.
Therefore, we express the time complexity as $O(n)$, indicating that the function grows linearly with the input size 'n'.

what is the Big oh notation for "1+n(n+1)"

- To find the Big O notation for the expression "1 + n(n + 1)," let's simplify the expression and then determine its growth rate as 'n' becomes very large.
- 1 + n(n + 1) = 1 + n^2 + n
- Now, let's consider the most significant term concerning 'n' as 'n' approaches infinity. In this case, the dominant term is n^2. Therefore, the Big O notation for the expression is O(n^2).
- So, the Big O notation for "1 + n(n + 1)" is O(n^2).

Omega notation: Ω

- As the amount of resources required to run an algorithm generally varies with the size of the input.
  - The complexity is typically expressed as a function f(n), where n is the size of the input.



If there is any **lower bound** that function by some other function g(n) after some input n0, then

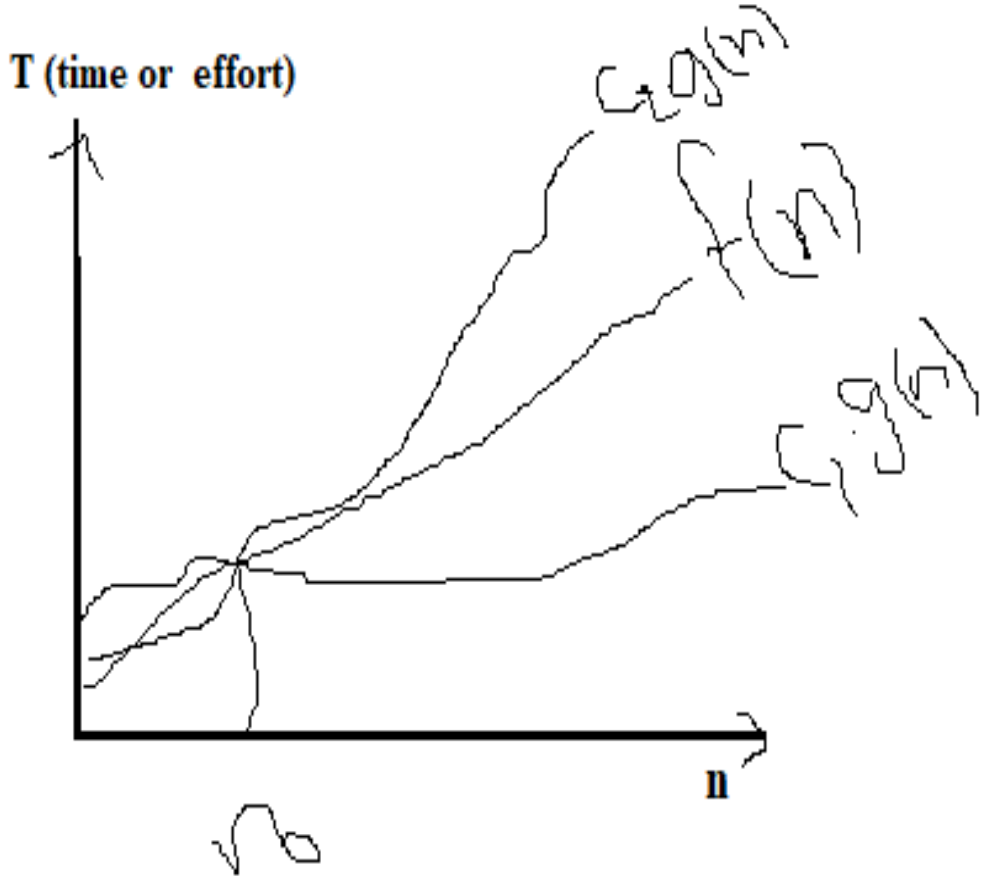
f(n)≥C*g(n) for all n ≥ n0, n0 ≥ 1 and c>0

then it can be written as **f(n)= Ω(g(n))**

(read as “f of n is Omega of g of n”)

Theta notation: θ

- As the amount of resources required to run an algorithm generally varies with the size of the input.
  - The complexity is typically expressed as a function f(n), where n is the size of the input.



If there are **two bounds** above and below the function by some other functions c1*g(n) and c2*g(n) after some input n0,

then **C1*g(n) ≤f(n)≤C2*g(n)** for all n ≥ n0, n0 ≥ 1 and c>0

then it can be written as **f(n)= θ (g(n))**

(read as "f of n is theta of g of n")

Little oh: o

- function f(n)=o(g(n)) (read as “f of n is little oh of g of n”) iff

Lim f(n)/g(n)=0 for all n, n≥1

n→∞

- **Example:**
- $3n+2= o(n^2)$

Bio oh(O): **0<=f(n)<=c.g(n)**:: at least one choice of a constant c>0;

Little oh (o): **0<=f(n)<c.g(n)**:: every choice of constant c>o

Little Omega: ω

- f(n)= ω (g(n)) (read as “f of n is little ohomega of g of n”) iff

Lim g(n)/f(n)=0 for all n, n≥0

n→∞

- ***Best case complexity***: Least or minimum amount of resources that needed over all inputs of size n
- ***Worst-case complexity***: The maximum of the amount of resources that are needed over all inputs of size n
- ***Average-case complexity***: The average of the amount of resources over all inputs of size n.

Time complexity analysis example

Operation: Linear or sequential search

Best Case:

Minimum number of comparisons.

To search element 5 it takes one comparison.

So, here best case (minimum number of comparisons) is 1.

Worst case:

Maximum number of comparison

To search element 15 it takes 10 comparisons.

If there are n elements then n comparisons were required.

Time complexity analysis example

Average case:

Average case is the situation in which it is not either best case or worst case.

Element is found somewhere in the middle of the list.

Probability of successfully search is : P

Probability of unsuccessful search is: 1-P

Consider the given element is found at position ‘i’, then the probability of finding that element at i_{th} position is P/n.

$$C_{Avg} = \underbrace{\left[1 \times \frac{P}{n} + 2 \times \frac{P}{n} + \cdots + n \times \frac{P}{n}\right]}_{\text{Successful search}} + \underbrace{n(1-P)}_{\text{Unsuccessful search}}$$

$$= \frac{P}{n}\left[1+2+\cdots+n\right] + n[1-P]$$

$$= \frac{P}{n}\frac{n(n+1)}{2} + n[1-P]$$

$$= \frac{P[n+1]}{2} + n[1-P]$$

$$= 1 \times \frac{n+1}{2} + \underbrace{n[1-1]}_{0}$$

$$= \frac{n+1}{2}$$

$P=1 \rightarrow$ Successful

$P=0 \rightarrow$ Unsuccessful

$P=0$

$$\frac{0[n-1]}{2} + n[1-0]$$

$$= 0 + n = n$$

$$C_{AVg} = \underbrace{[1*P/n + 2\times P/n + \cdots + n\times P/n]}_{\text{successful search}} + \underbrace{n[1-P]}_{\text{unsuccessful search.}}$$

$$= \frac{P}{n}[1+2+3+\cdots+n] + n(1-P)$$

$$= \frac{P}{n}\left[\frac{n(n+1)}{2}\right] + n[1-P]$$

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

$$C_{AVg}(n) = \frac{1(n+1)}{2} + n(1-1)$$

$$= \underline{\underline{\frac{n+1}{2}}} \rightarrow \text{for successful search.}$$

$$C_{AVg}(n) = \frac{0(n+1)}{2} + n(1-0)$$

$$= \underline{\underline{n}} \rightarrow \text{for unsuccessful search.}$$

Big oh: O	Omega: Ω	Theta: θ
Worst-case complexity The maximum of the amount of resources that are needed over all inputs of size n In any case algorithm time will not exceed this.	Best case complexity Least or minimum amount of resources that needed over all inputs of size n In any case algorithm never achieve better than this.	Average-case complexity The average of the amount of resources over all inputs of size n. When best case and worst case both are same we use this notation.

- *Worst case complexity mostly used in algorithm designs instead of the best and average case complexity*?
- Because
  - Best-case for almost any algorithm is trivial (inconsequential).
  - Average case complexity is often very hard to determine. It's essentially the weighted average of the performance of the algorithm across all inputs—weighted by the probability of encountering the input.
  - The Big Oh (O) notation helps choose the best between two algorithms with different time complexities even though they have the same function (task).

Constant	Logarithmic	Linear time	Polynomial time (q>1	Exponential time
1	2	3	4	5
O(1)	O(log n)	O(n)	$O(n^q)$; If q=2,3	$O(2^n)$

Order of growth

- Suppose you have analyzed two algorithms (let say Algorithm A and B) and expressed their run times (steps to solve a problem) in terms of the size of the input size n:
  - Algorithm A takes 100 n + 1 steps;
  - Algorithm B takes n^2 + n + 1 steps.

Input Size	Run time of Algorithm A	Run time of Algorithm B
1	101	3
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

At n=1 and 10, Algorithm A looks pretty bad; it takes almost 10 times longer than Algorithm B.

But for n=100 they are about the same, and for larger values A is much better.

Order of growth (Cont.)

- An **order of growth** is a set of functions whose asymptotic growth behaviour is considered equivalent.
- For example, $2n$, $100n$ and $n + 1$ belong to the same order of growth, which is written $O(n)$ and often called **linear** because every function in the set grows linearly with n.
- All functions with the leading term n^2 belong to $O(n^2)$; they are **quadratic**, which is a fancy word for functions with the leading term n^2.

Constant time O(1):

- Algorithm requires the same fixed number of steps (regardless of) independent of the size of input.
- Ex:

```
Bool IsFirstElementNull (String [] str)
{
        If ( str[0]==NULL){
        return true;
        }
        else return false;
        }
```

Other real time example:

- ✓ Push and Pop operations of stack containing N elements.
- ✓ Insertion and remove operation from a Queue.

Logarithmic time O(log n):

- Binary Search in a sorted list of n elements.

Linear time O(n):

- The algorithm required number of steps are proportional to the size of the task.
- Example:
  - Finding max and minimum number in given unsorted order list.
  - Sequential or linear search in an unordered list of n elements.
  - Calculating the n factorial in an iterative way.
  - Calculating the Fibonacci series in an iterative way.

Linear logarithmic time: **n(log n)**

- Means algorithms takes O(log n) operations for each item in your input.
- Example: Sorting algorithms (Quick sort and Merge Sort): **n(log n)**

Quadratic time: $O(n^2)$

- Number of operations is proportional to the size of the task square.
- Example: Some simple sorting algorithms (selection sorting)

Bubble Sort

```
BubbleSort (Arr, N) // Arr is an array of size N.
{
    For ( I:= 1 to (N-1) ) // N elements => (N-1) pass
    {
        For ( J:= 1 to  (N-I) ) // Execute the pass
        {
            If ( Arr [J] > Arr[J+1] )
                Swap( Arr[j], Arr[J+1] );
        }
    }
}
```

Selection Sort

```
For ( I:= 1 to (N-1) ) // N elements => (N-1) pass
    {
        min_index = I;
        For ( J:= I+1 to N ) // Search Unsorted Subarray (Right half)
        {
            If ( Arr [J] < Arr[min_index] )
                min_index = J; // Current minimum
        }
        // Swap I-th smallest element with current I-th place element
        If (min_Index != I)
            Swap ( Arr[I], Arr[min_index] );
    }
```

Insertion Sort

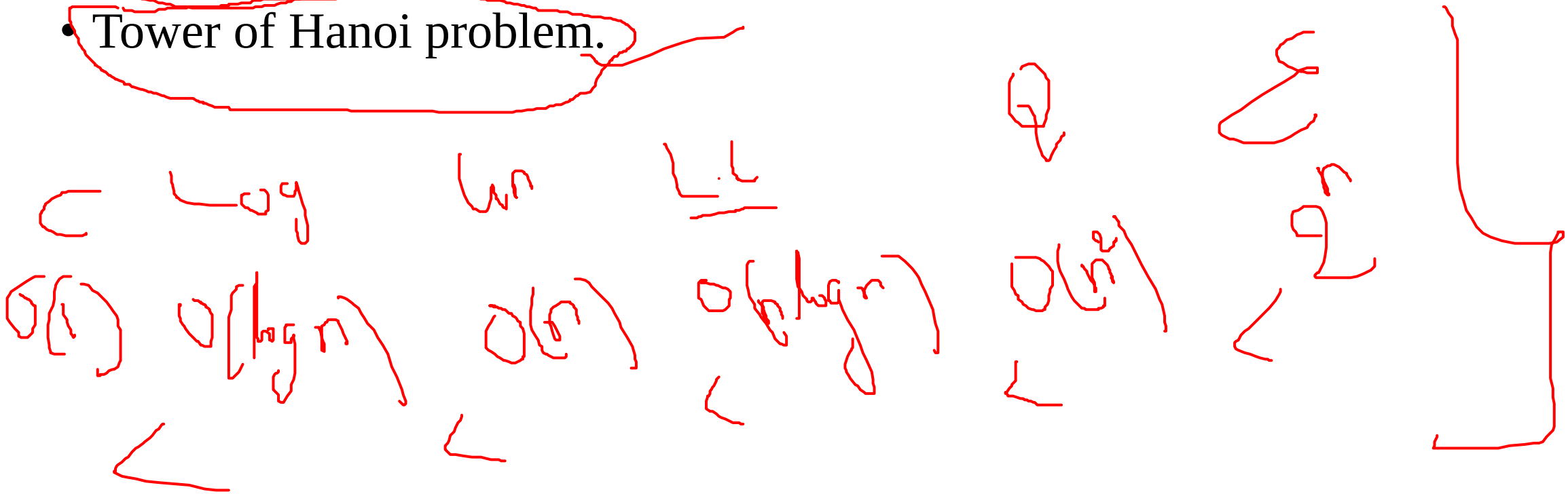
```
For ( I:= 2 to N ) // N elements => (N-1) pass
    {
        insert_at = I; // Find suitable position insert_at, for Arr[I]

        item = Arr[I]; J=I-1;
        While ( J ? 1 && item < Arr[J] )
        {
            Arr[J+1] = Arr[J]; // Move to right
            // insert_at = J;
            J--;
        }
        insert_at = J+1; // Insert at proper position
        Arr[insert_at] = item; // Arr[J+1] = item;
    }
```

Best Case (Sorted array as input)	**O(N); O(1) swaps.**	**[$O(N^2)$]. And O(1) swaps.**	**[O(N)]. And O(1) swaps**
Worst Case	[$O(N^2)$]. $O(N^2)$ swaps	[$O(N^2)$] . O(N) swaps.	[$O(N^2)$] . $O(N^2)$ swaps.

Exponential time: 2^n

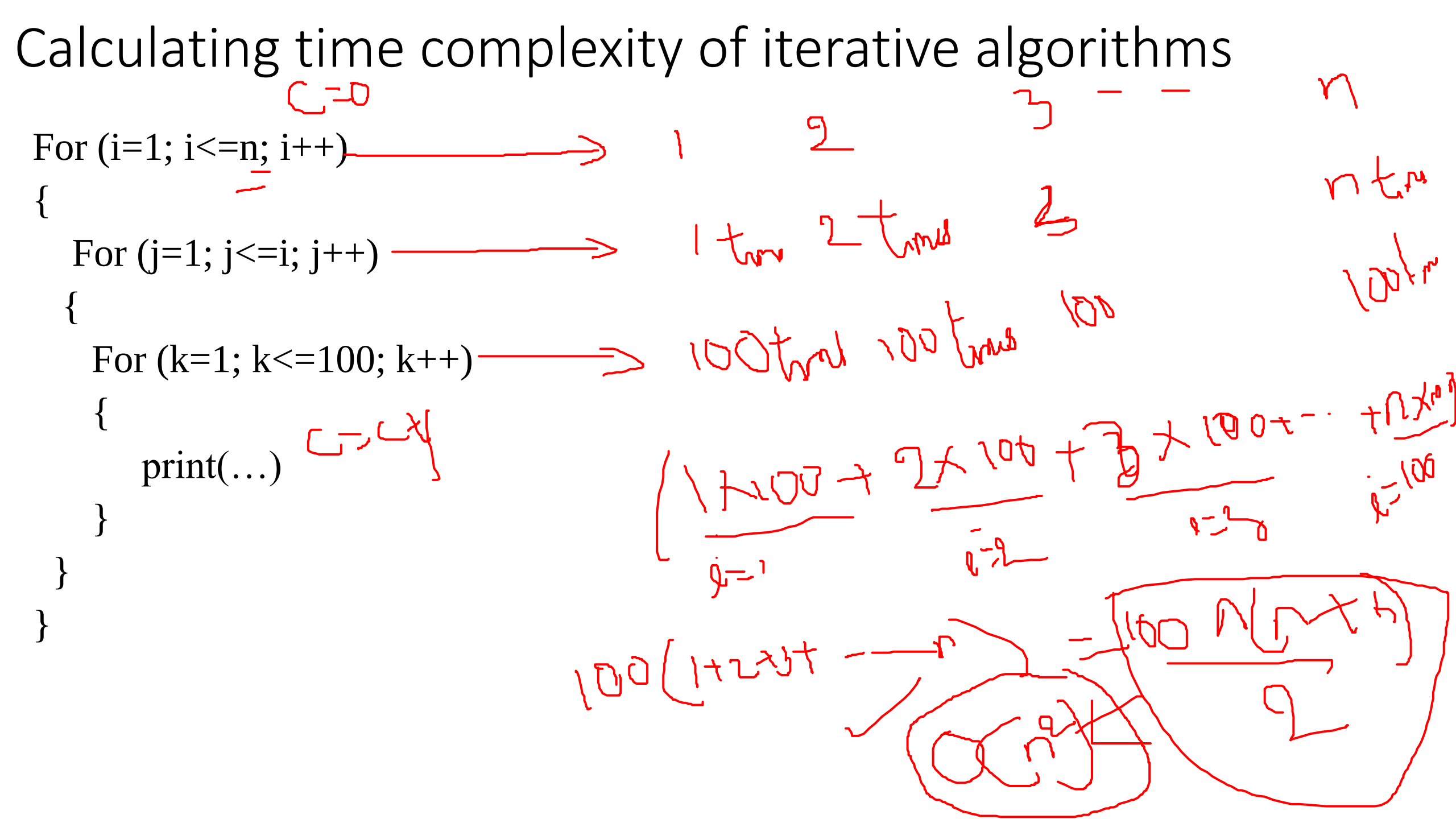
- Example:
  - Recursive Fibonacci implementation
  - Tower of Hanoi problem.



Time complexity calculation for iteration methods

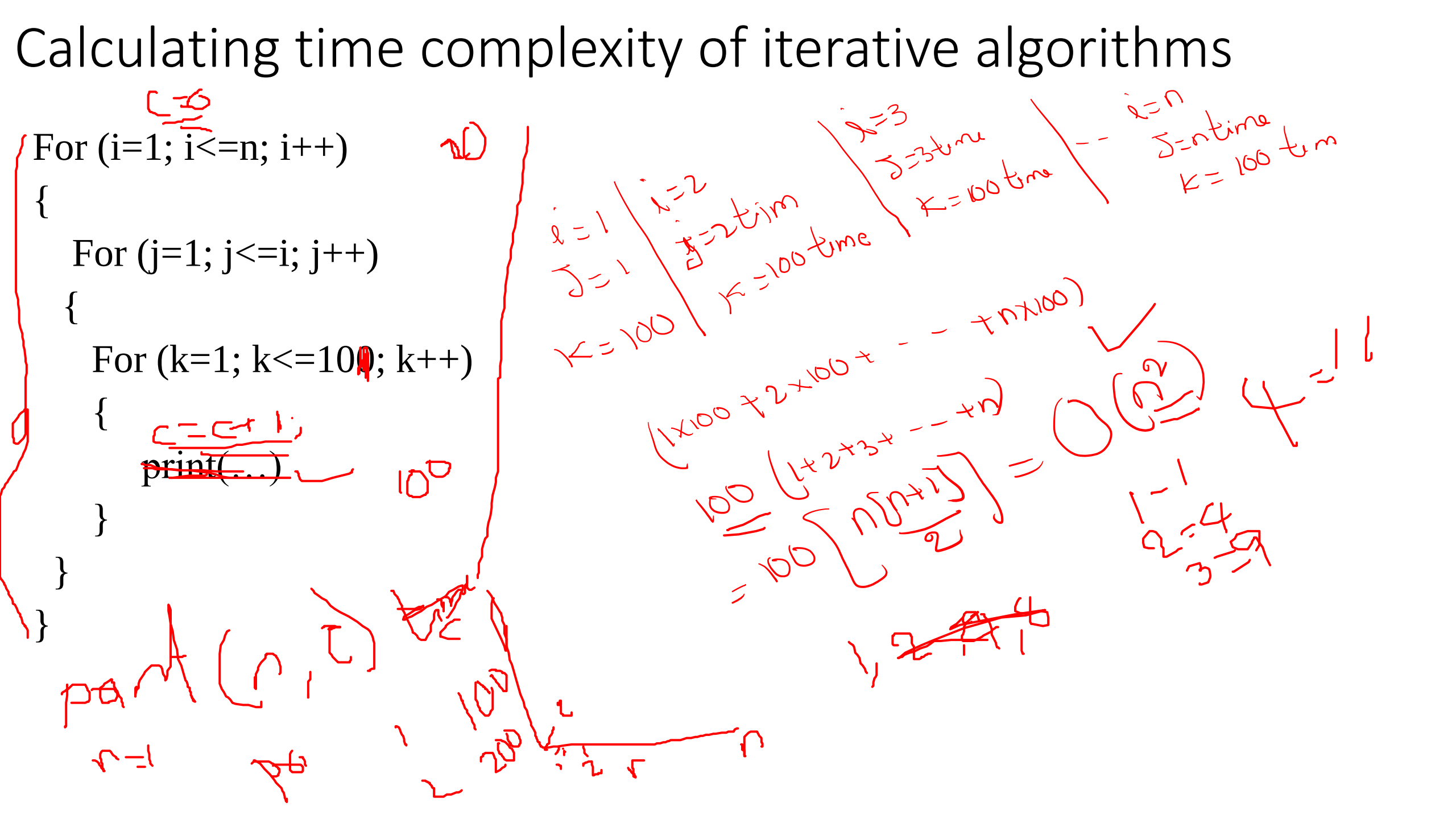
Calculating time complexity of iterative algorithms

```
For (i=1; i<=n; i++)
{
   For (j=1; j<=i; j++)
   {
     For (k=1; k<=100; k++)
     {
        print(…)
     }
   }
}
```



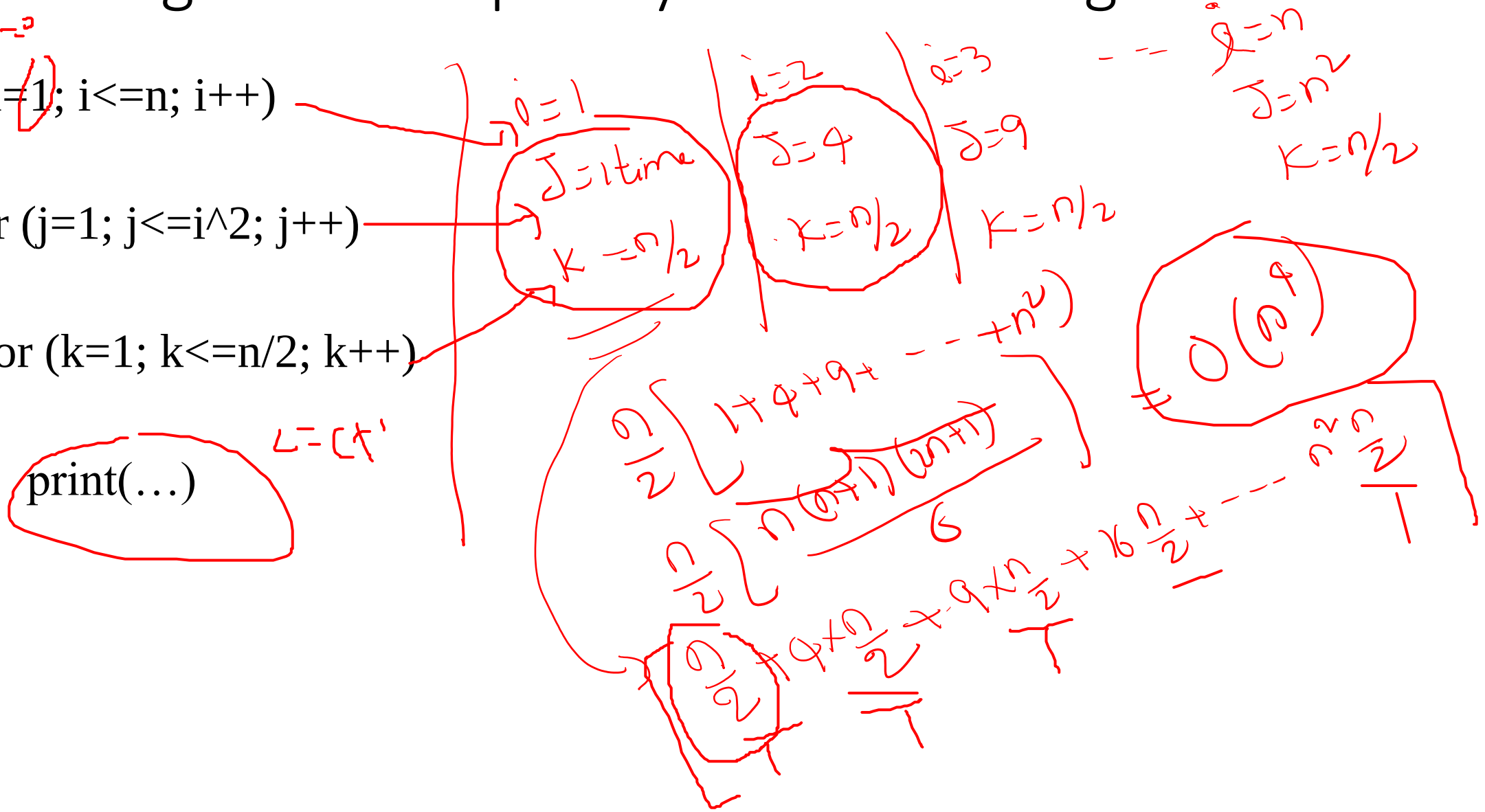
Calculating time complexity of iterative algorithms

```
For (i=1; i<=n; i++)
{
   For (j=1; j<=i; j++)
   {
     For (k=1; k<=100; k++)
     {
        print(...)
     }
   }
}
```



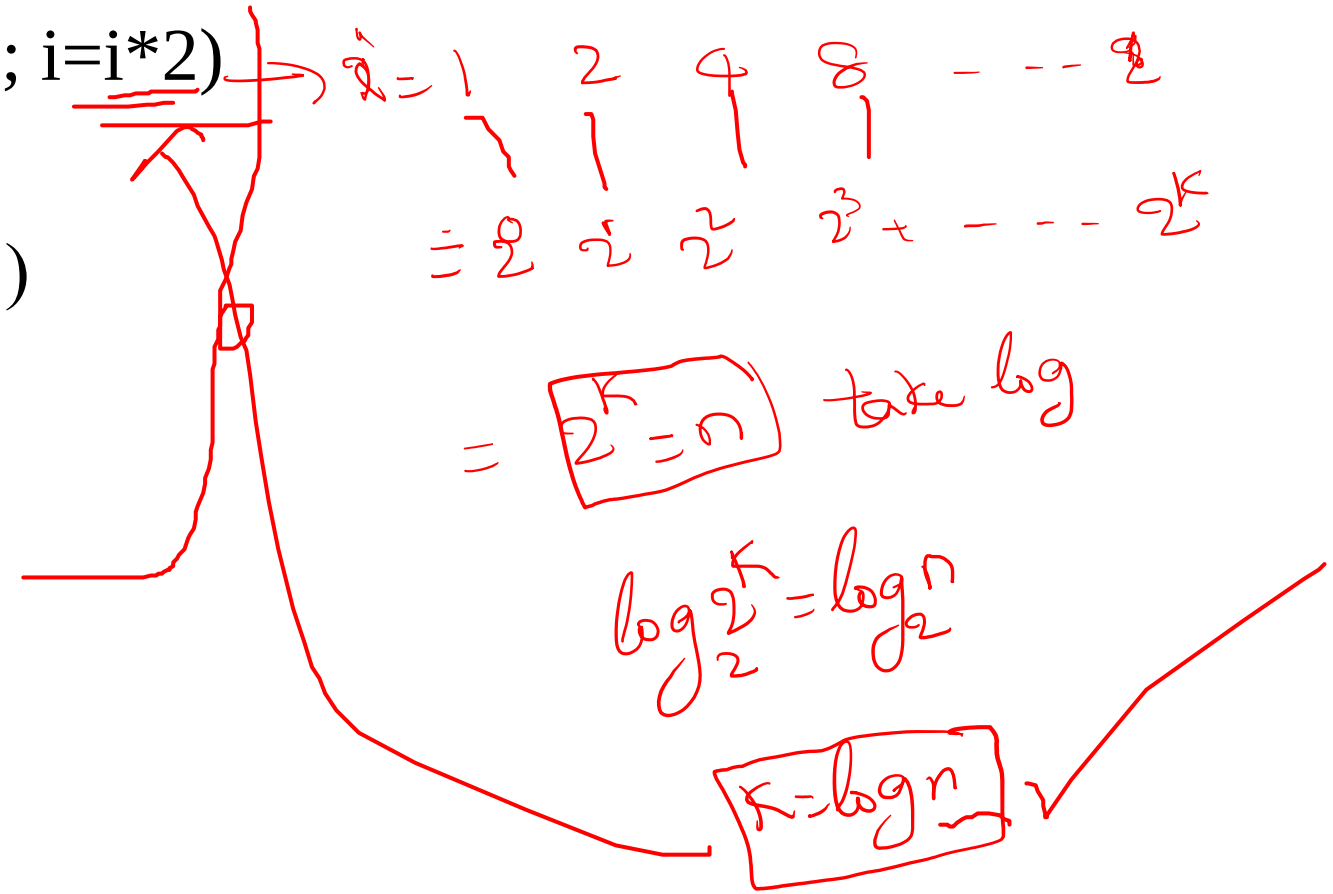
Calculating time complexity of iterative algorithms

```
For (i=1; i<=n; i++)
{
   For (j=1; j<=i^2; j++)
   {
      For (k=1; k<=n/2; k++)
      {
         print(…)
      }
   }
}
```



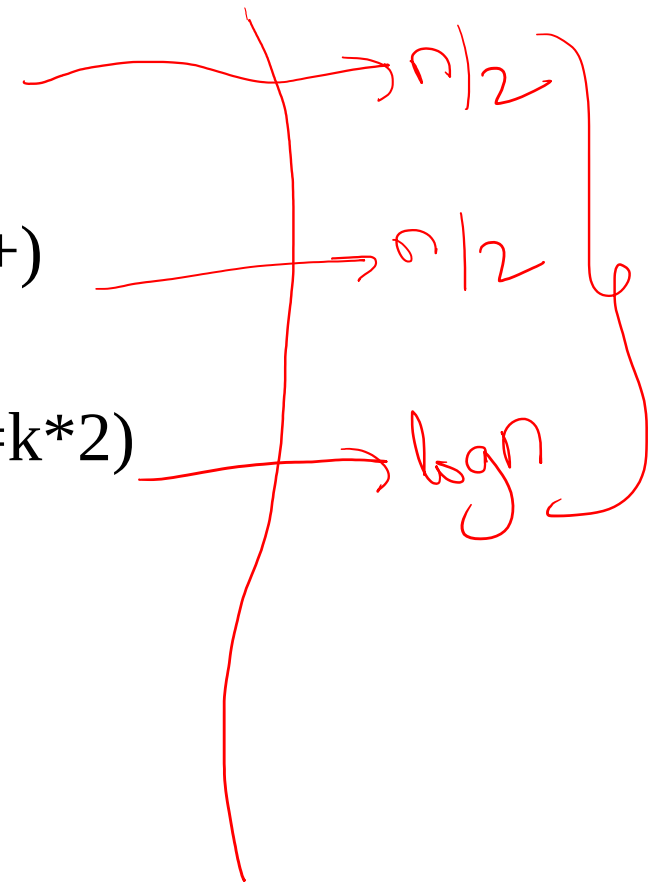
Calculating time complexity of iterative algorithms

```
For (i=1; i<=n; i=i*2)
    {
        print(…)
    }
```



Calculating time complexity of iterative algorithms

```
For (i=n/2; i<=n; i++)
{
   For (j=1; j<=n/2; j++)
  {
     For (k=1; k<=n; k=k*2)
     {
         print(…)
     }
  }
}
```



$= \frac{n}{2} \times \frac{n}{2} \log_2$

$= n^2 \log n$

$O(n^2 \log n)$

Calculating time complexity of iterative algorithms

```
For (i=n/2; i<=n; i++)
{
   For (j=1; j<=n; j=2*j)
   {
     For (k=1; k<=n; k=k*2)
     {
         print(…)
     }
  }
}
```

$n/2$

$\log n$

$\log n$

$\frac{n}{2} \times \log n \times \log n$

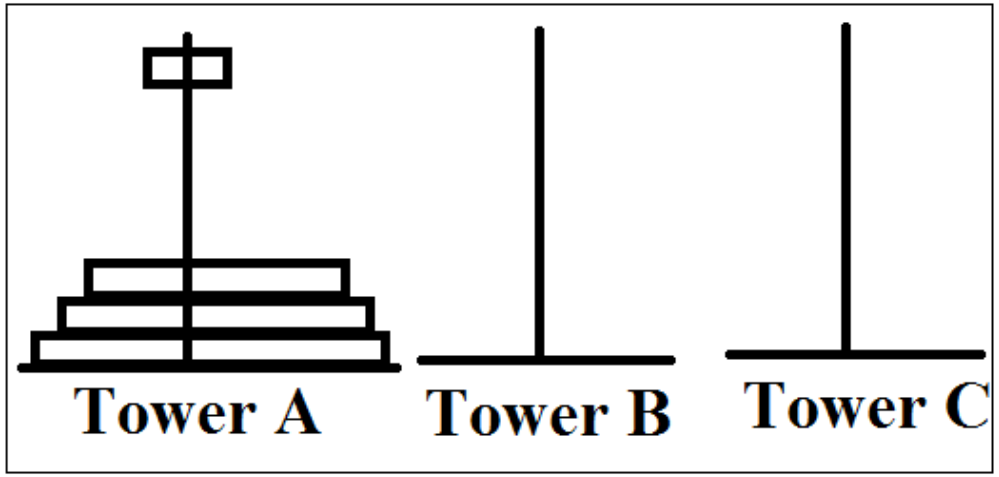
$O(n \log^2 n)$

$\log_2 * \log_2 = (\log_2)^2$

Analysis of Recursive Algorithms

- Analysis of Recursive Algorithms through Recurrence Relations
- Recurrence relation is an equation that recursively defines a sequence or multidimensional array of values.
- There are mainly three ways for solving recurrences.
  - ✓**Substitution Method**: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.
  - ✓**Recurrence Tree Method**: In this method, a recurrence tree is formed where each node represents the cost.
  - ✓**Master Method**: is a direct way to get the solution.

Recursive Algorithm: Example



Towers of Hanoi

Task: Move all disks from tower A to tower C by using intermediate tower B.

Rules/Condition:

Move one disk at a time.

The small disk should be above the big disk and

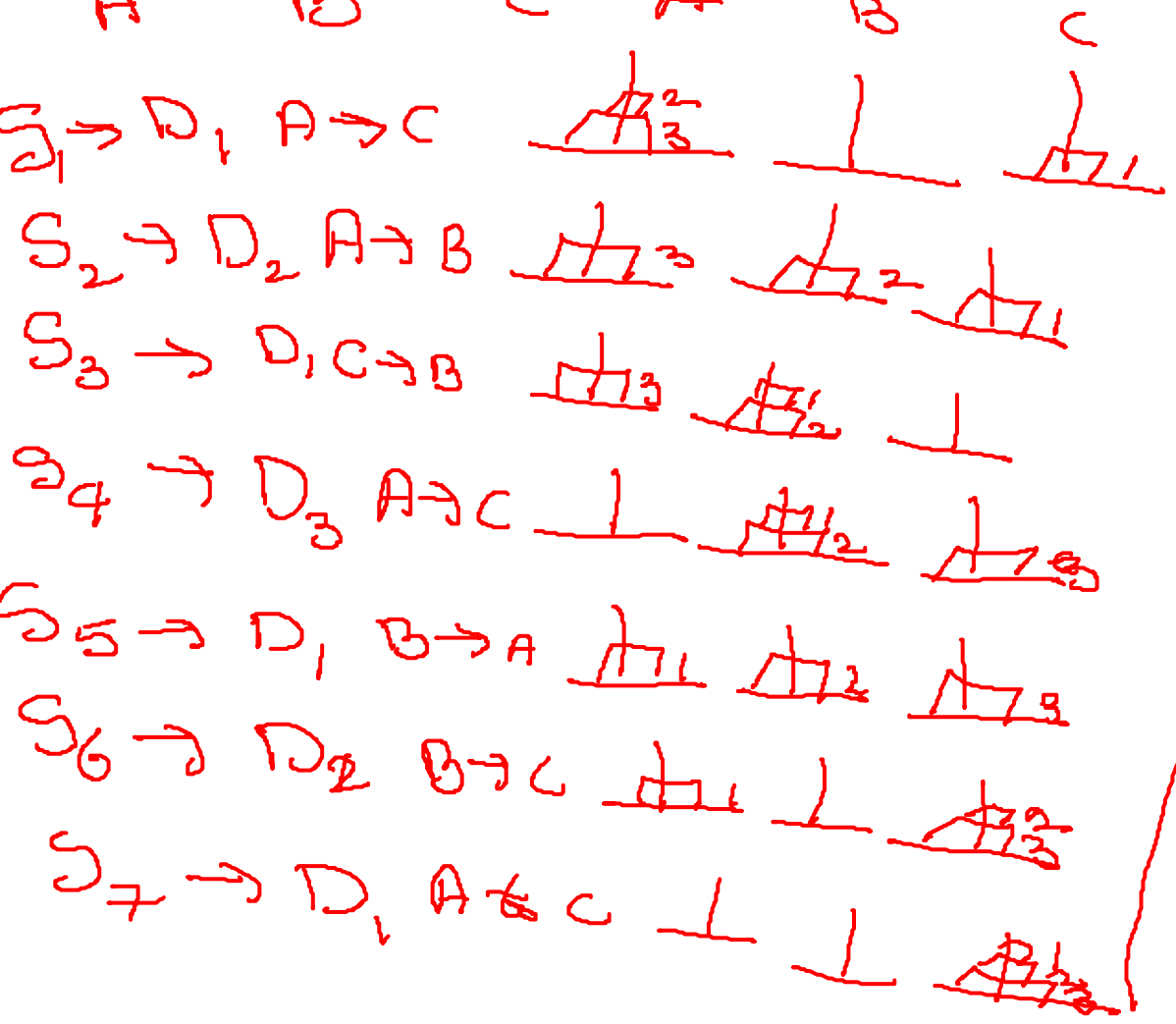
size(disk1)<size(disk2)<size(disk3)....,<size(disk n-1)<size(disk n)

Towers of Hanoi: Solution

Step 1: Move first (n-1) disks from tower A to tower B with the help of tower c.

Step2: Move nth disk from tower A to tower C.

Step3: Move (n-1) disks from tower B to tower C with the help of tower A



$7 \Rightarrow 2^3 - 1$

$4 = 2^4 - 1 = 15$

$5 = 2^5 - 1 = 31$

$2 \rightarrow 3$

$3 \rightarrow 7$

$4 = 15$

$n = 2^n - 1$

$2^n - 1$

SIT

RBT

10%

Analysis of Recursive Algorithms: **Substitution Method**

- Guess the Solution.
- Use the mathematical induction to find the boundary condition and shows that the guess is correct.

Towers Of Hani

```
Algorithm TowersOfHanoi(n, x, y, z)
// Move the top n disks from tower x to tower y.
{
    if (n ≥ 1) then
    {
        TowersOfHanoi(n − 1, x, z, y);
        write ("move top disk from tower", x,
        "to top of tower", y);
        TowersOfHanoi(n − 1, z, y, x);
    }
}
```

T(n)

T(n-1)

1

T(n-1)

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

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

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

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

$= 1 + 2 + 4T(n-2)$

$= 1 + 2 + 4[1 + 2T(n-3)]$

$= 1 + 2 + 4 + 8T(n-3)$

$= 1 + 2 + 4 + 8 + 16T(n-4)$

$= 1 + 2^1 + 2^2 + 2^3 + 2^4 T(n-4)$

$= 1 + 2 + 2^2 + 2^3 + 2^4 + \cdots + 2^{k-1} + 2^k T(n-k)$

$= 1 + 2 + 2^2 + 2^3 + 2^4 \cdots 2^{n-1} + 2^n T(n-n)$

$= 1 + 2^1 + 2^2 + \cdots + 2^{n-1} \Rightarrow O(2^{n-1})$

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

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

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

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

$O(2^{n-1}) = O(2^n)$ ✓

$O(2^{n-1}) \simeq O(2^{n-2})$ ✗

$n \quad a^{(1-x)} \quad 1-x$

$2^{n-1} \quad O(2^n)$ ✓

Towers Of Hani algorithm analysis using substitution method

$T(n) = T(n-1) + 1 + T(n-1)$ [∵ From Solution Steps]

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

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

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

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

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

$= 1 + 2 + 4T(n-2)$

$\therefore T(n) = 1 + 2 + 4(1 + 2T(n-3))$

$= 1 + 2 + 4 + 8T(n-3)$

$= 1 + 2 + 4 + 8(1 + 2T(n-4))$

$= 1 + 2 + 4 + 8 + 16T(n-4)$

$= 1 + 2 + 4 + 8 + 16(1 + 2T(n-5))$

$= 1 + 2 + 4 + 8 + 16 + 32T(n-5)$

2^{5-1} 2^5

$= 1 + 2 + 4 + 8 + 16 + \cdots + 2^{n-1} + 2^n T(n-n)$

$= 1 + 2 + 4 + 8 + 16 + \cdots + 2^{n-1} + 2^n T(0)$

$= 1 + 2 + 4 + 8 + 16 + \cdots + 2^{n-1} + 2^n (0)$ [∵ $T(0) = 0$]

$= 1 + 2 + 4 + 8 + 16 + \cdots + 2^{n-1}$

$\therefore \boxed{T(n) = 2^n - 1}$

```
Algorithm TowersOfHanoi(n, x, y, z)
// Move the top n disks from tower x to tower y.
{
    if (n ≥ 1) then
    {
        TowersOfHanoi(n − 1, x, z, y);
        write ("move top disk from tower", x,
        "to top of tower", y);
        TowersOfHanoi(n − 1, z, y, x);
    }
}
```

Towers Of Hani

Substitution Method: sum of n numbers Example

```
Algorithm RSum(a, n)
{
    count := count + 1; // For the if conditional
    if (n ≤ 0) then
    {
        count := count + 1; // For the return
        return 0.0;
    }
    else
    {
        count := count + 1;  // For the addition, function
                             // invocation and return
        return RSum(a, n − 1) + a[n];
    }
}
```

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

So the step count for Rsum is:2n+2.

Substitution Method: Fibonacci series example

```
Alg, fib(n) {
read n
if n := 1 then return 0
if n > 1 && n ≤ 3 then return 1
else
return (fib(n-1) + fib(n-2))
}
```

T(n)

T(n-1) T(n-2)

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

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

Time complexity :-

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

$T(n) \geq 2T(n-2) + O(1)$

$T(n-2) = 2T(n-4) + O(1)$

$T(n) = 4T(n-4) + 2O(1) + O(1)$

$= 8T(n-8) + 4O(1) + 2(O(1)) + \dots$

$= \vdots$

$= 2^{n/2}$

Time complexity $= O(2^{n/2})$

i.e., exponential time

Substitution Method: Fibonacci series example

Fibonacci Series: Iterative way

```
Alg.  fibI(n)
{
 f := 0 , s := 1
 for c = 0 to n do {
  if c ≤ 1 then nc = c
      return nc
  else
        nc = f+s ,
          f = s
          s = nc
          return nc.
 }
}
```

Time complexity = O(n)

Fibonacci Series: recursive way way

```
Alg.  fib(n) {
read n
if n := 1 then return 0
if n > 1 && n ≤ 3 then return
else
return (fib(n-1) + fib(n-2))
}
```

Time complexity = $O(2^{n/2})$

↓

i.e, exponential time

Substitution Method:Binary Search

- T(n) = time taken to search for an item in an array of size n
- Binary Search **$T(n) = T(n/2) + O(1) \Rightarrow T(n) = T(n/2) + c$**
- **Solving by Substitution Method**
- $T(n) = T(n/2) + c$
- $= T(n/4)+c + c = T(n/4) + 2c$
- $= T(n/8) + 3c$
- …. $T(n) = T(n/2^k) + kc$
- Let $n = 2^k$. => $k = \log_2 n$
- $T(n) = T(2^k/2^k) + \log_2 n.c$
- $T(n) = T(1) + \log_2 n.c$
- T(1): Time take to search for an item in an array of size 1 = const. time = p
- $T(n) = p + \log_2 n.c$ => **$T(n) = O(\log n)$**
- P= O(1)
- $\text{Log}_2 n.c = O(\log n)$
- $P+ \log_2 n.c = O(\max\{\log n, 1\}) = O(\log n)$

Alg

Recurrence Relation.

Tower of Hanoi — $T(n) = 2T(n-1)+1$

fib — $T(n) = T(n-1)+T(n-2)+\Theta(1)$

$= 2T(n-1)+\Theta(1)$ ⓐ $2T(n-2)+\Theta(1)$.

Sum of n — $T(n) = 2+T(n-1)$

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

Binary search — $T(n) = T(n/2)+\Theta(1)$

Question

Avg/B.C = $T(n) = 2T(n/2) + \Theta(n)$

W.C = $T(n) = T(n-1) + \Theta(n)$

Linear Search

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

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

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

Insertion Sort

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

Selection Sort

Analysis of Recursive Algorithms through Recurrence Relations: **Masters' Theorem.**

- Master Method is a direct way to get the solution in a simple and quick way.
- 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 sub-problems in the recursion
  - **n/b** = size of each sub-problem. All sub-problems 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) is an asymptotically positive function.

- $T(n) = aT(n/b) + f(n)$ Here,
  - $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function, where, $T(n)$ has the following asymptotic bounds:

- **Case 1:** If **$f(n) = O(n^{\log_b a - \epsilon})$,** then **$T(n) = \Theta(n^{\log_b a})$.**
  - 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. $n^{\log_b a}$

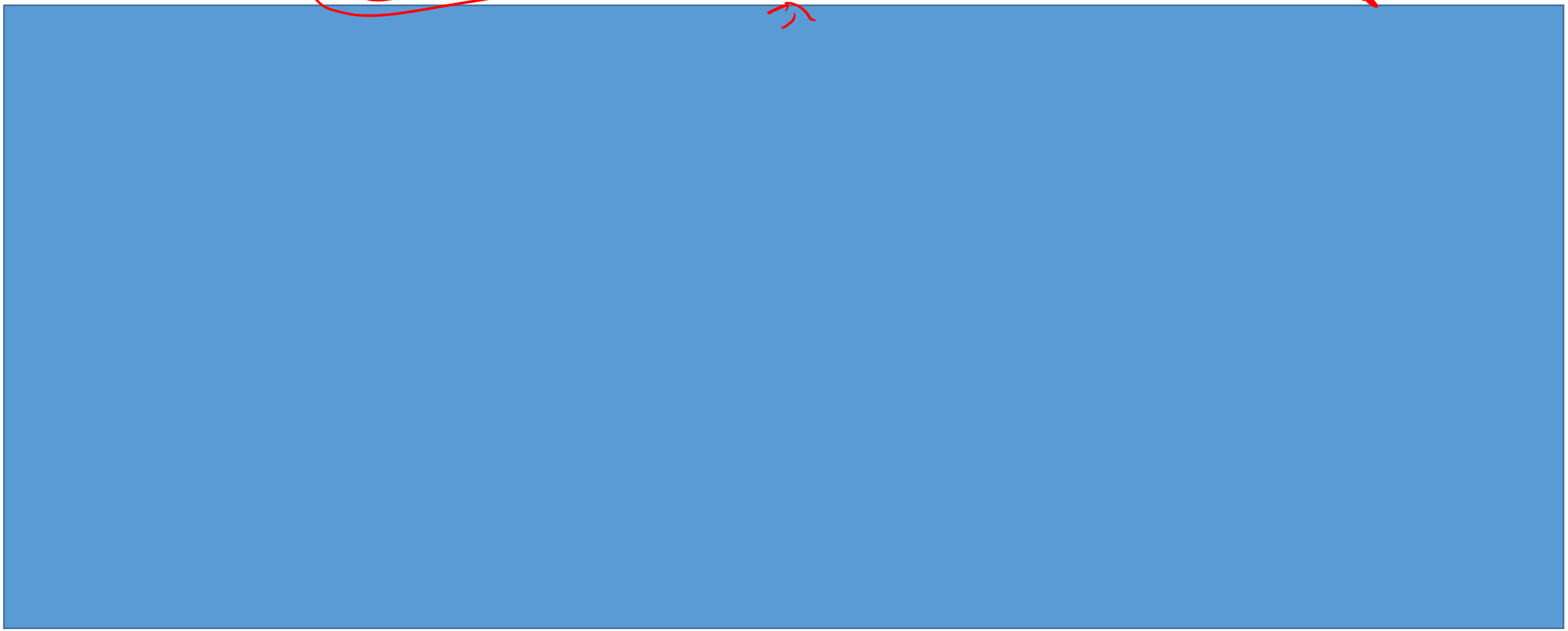
- **Case 2**: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} * \log^{k+1} n)$.
  - If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b a}$.
  Thus, the time complexity will be $f(n)$ times the total number of levels ie. $n^{\log_b a} * \log n$.

- **Case 3**: If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.
  - 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 $f(n)$.

$\epsilon > 0$ is a constant.

Example 1:

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



Example 1:

Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$

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

Solution: $T(n) = \Theta(n^2)$

How:

$F(n)=n^2$

$a=3$ and $b=2$;

in case 3, $\Omega(n^{\log_b a+\epsilon}) = n^{\log_2 3+\epsilon} = n^2$ if $\epsilon = 1$

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

Example 2:

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

Example 2:

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

Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} * \log^{k+1} n)$.

How:

$F(n)=n^2$

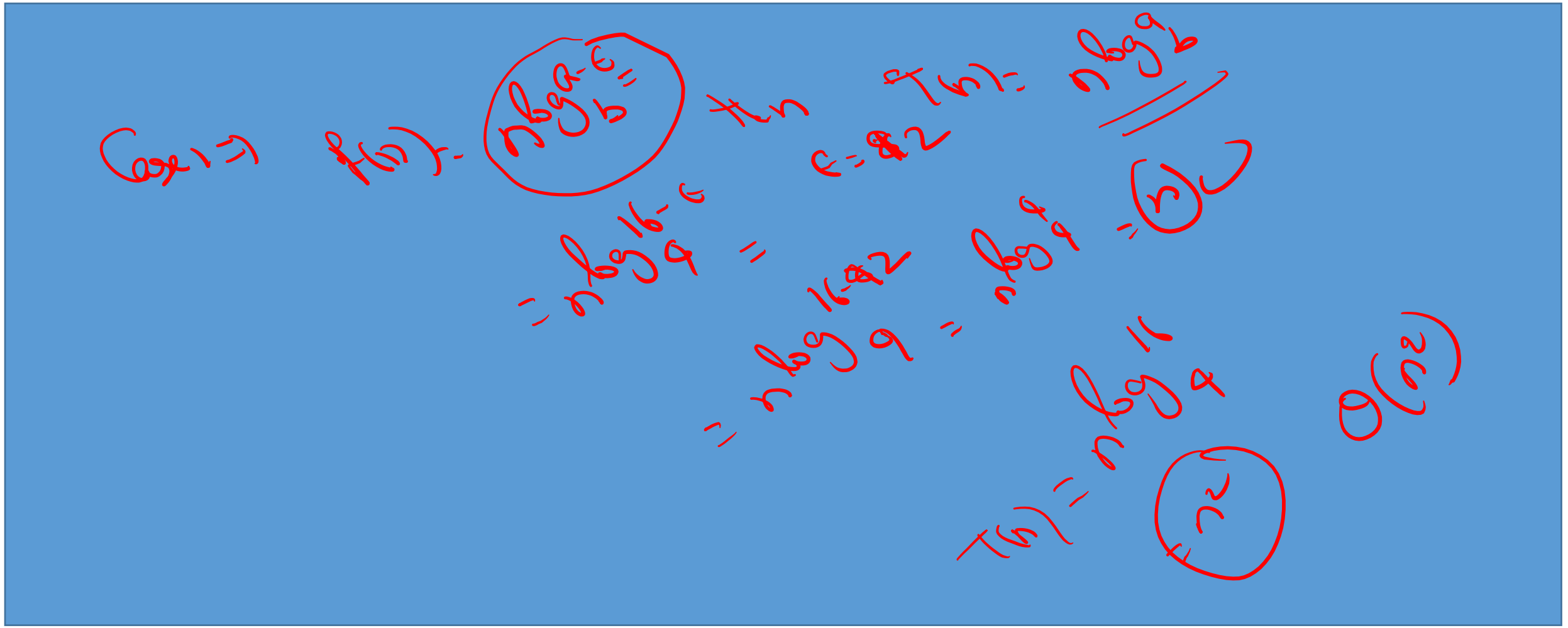
$a=4$ and $b=2$;

in case 2, $= \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$ $n^{\log_2 4} * \log^k n = n^2 * \log n$ if $k = 0$

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

Example 3:

$$T(n) = 16T(n/4) + n:$$



Example 3:

$$T(n) = 16T(n/4) + n$$

Solution: $T(n) = \Theta(n^2)$

How:

F(n)=n

a=16 and b=4;

in case 1, $\Omega(n^{\log_b a-\epsilon}) = n^{\log_2 16-\epsilon} = n$ if $\epsilon = 14$

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

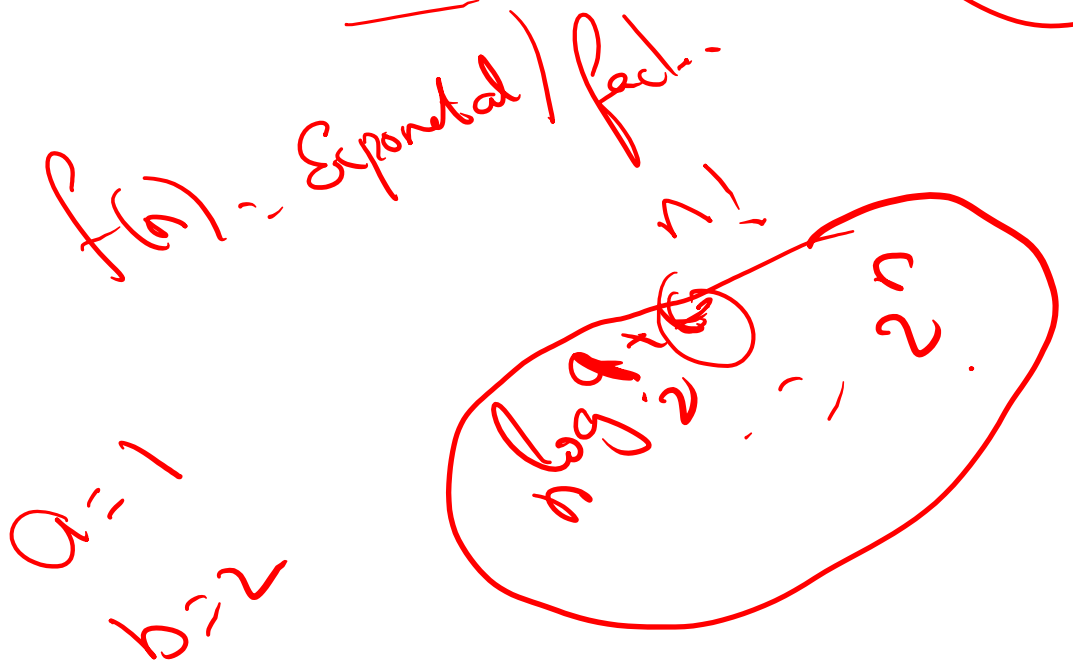
Case 1: If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

Example 4:

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

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

Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$



Example 4:

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

Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$

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

- Merge Sort: T(n) = 2T(n/2) + Θ(n).

solution is Θ(n Logn)

Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} * \log^{k+1} n)$.

Binary Search: T(n) = T(n/2) + Θ(1).

solution is Θ(Logn)

$\log_2(1) = 0$

$2^0 = 1$

- Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$.

solution is $\Theta(n \log n)$

Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} * \log^{k+1} n)$.

Binary Search: $T(n) = T(n/2) + \Theta(1)$.

solution is $\Theta(\log n)$

$\log_2(1) = 0$

$2^0 = 1$

- It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem

$T(n) = 2^n T(n/2) + n^n \implies$ Does not apply (a is not constant)

$T(n) = 2T(n/2) + n/\log n \implies$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)

$T(n) = 64T(n/8) - n^2 \log n \implies$ Does not apply ($f(n)$ is not positive)

$T(n) = 0.5T(n/2) + 1/n \implies$ Does not apply ($a < 1$)

$T(n) = T(n/2) + n(2 - \cos n) \implies$ Does not apply. We are in Case 3, but the regularity condition is violated. (Consider $n = 2\pi k$, where k is odd and arbitrarily large. For any such choice of n, you can show that $c \geq 3/2$, thereby violating the regularity condition.)

Some more examples:

$$T(n) = 2T(n/2) + n\log n \implies T(n) = n\log^2 n \text{ (Case 2)}$$

$$T(n) = 2T(n/4) + n^{0.51} \implies T(n) = \Theta(n^{0.51}) \text{ (Case 3)}$$

$$T(n) = 16T(n/4) + n! \implies T(n) = \Theta(n!) \text{ (Case 3)}$$

$$T(n) = \sqrt{2}T(n/2) + \log n \implies T(n) = \Theta(\sqrt{n}) \text{ (Case 1)}$$

$$T(n) = 3T(n/2) + n \implies T(n) = \Theta(n^{\lg 3}) \text{ (Case 1)}$$

$$T(n) = 3T(n/3) + \sqrt{n} \implies T(n) = \Theta(n) \text{ (Case 1)}$$

$T(n) = 4T(n/2) + cn \Longrightarrow$

$T(n) = 3T(n/4) + n \log n \Longrightarrow$

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

$T(n) = 6T(n/3) + n^2 \log n \Longrightarrow$

$T(n) = 4T(n/2) + n/\log n \Longrightarrow$

$T(n) = 7T(n/3) + n^2 \Longrightarrow$

$T(n) = 4T(n/2) + \log n \Longrightarrow$

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real numbers.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$

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

   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$

   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$

   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

   b) if $p < 0$; then $T(n) = O(n^k)$

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

$a = 3 \quad b = 2 \quad k = 2 \quad p = 0$

a	b^k
3	2^2

$3 < 4$

$T(n) = \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$

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

$a = 4 \quad b = 2 \quad k = 2 \quad p = 0$

$4 = 2^2$

2) a) $T(n) = \Theta(n^{\log_2 4} \log^1 n)$

$= \Theta(n^2 \log n)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real numbers.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$

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

b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$

c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$

a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

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

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

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

$a = 1 \quad b = 2 \quad k = 2 \quad p = 0$

$1 < 2^2$

3)(a) $T(n) = \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

$T(n) = 16T(n/4) + n$.

$a = 16 \quad b = 4 \quad k = 1 \quad p = 0$.

$16 > 4^1$. $T(n) = \Theta(n^{\log_b a})$

$= \Theta(n^{\log_4 16}) \cdot \Theta(n^2)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

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

$a = 2 \quad b = 2 \quad k = 1 \quad p = 1$.

$2 = 2^1$

2)a). $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$.

$= \Theta(n^1 \log^2 n)$.

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

$T(n) = 2T(n/2) + n/\log n \implies$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real numbers.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

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

$= 2T(n/2) + n\log^{-1} n$.

$a = 2, b = 2, k = 1, p = -1$.

$2 = 2^1$

2) b) $T(n) = \Theta(n^{\log_b a} \log\log n)$

$= \Theta(n \log\log n)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$; then $T(n) = O(n^k)$

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

$a = 2 \quad b = 4 \quad k = 0.51 \quad p = 0$

$2 < 4^{0.51}$.

3) a) $T(n) = \Theta(n^k \log^p n)$.

$= \Theta(n^{0.51})$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$; then $T(n) = O(n^k)$

10) $T(n) = 6T(n/3) + n^2 \log n$.

$a = 6 \quad b = 3 \quad k = 2 \quad p = 1$.

$6 < 3^2$

3) a) $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^1 n)$

$= \Theta(n^2 \log n)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real numbers.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

12) $T(n) = 7T(n/3) + n^2$.

$a = 7 \quad b = 3 \quad k = 2 \quad p = 0$.

$7 < 3^2$

3) a) $T(n) = \Theta(n^k \log^p n)$.

$= \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$.

13) $T(n) = 4T(n/2) + \log n$

$a = 4 \quad b = 2 \quad k = 0 \quad p = 1$.

$4 > 2^0$ $T(n) = \Theta(n^{\log_b a})$.

$= \Theta(n^2)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

→ $T(n) = \sqrt{2}\, T(n/2) + \log n$

$a = \sqrt{2}$ $b = 2$ $k = 0$ $p = 1$

$\sqrt{2} > 2^0$

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 \sqrt{2}})$

$= \Theta(\sqrt{n})$

→ $T(n) = 2\, T(n/2) + \sqrt{n}$

$a = 2$ $b = 2$ $k = 1/2$ $p = 0$

$2 > 2^{1/2}$ $T(n) = \Theta(n^{\log_2 2})$

$= \Theta(n)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real numbers

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$; then $T(n) = O(n^k)$

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

$a = 3 \quad b = 2 \quad k = 1 \quad p = 0$

$3 > 2^1 \quad T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$

$T(n) = 3T(n/3) + \sqrt{n}$

$a = 3 \quad b = 3 \quad k = 1/2 \quad p = 0$

$3 > 3^{1/2}$

$T(n) = \Theta(n^{\log_3 3}) = \Theta(n)$.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1, b > 1, k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

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

$a = 4 \quad b = 2 \quad k = 1 \quad p = 0$

$a \quad b^k$

$4 > 2^1 \quad T(n) = \Theta(n^2)$.

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

$a = 3 \quad b = 4 \quad k = 1 \quad (p = 1)$

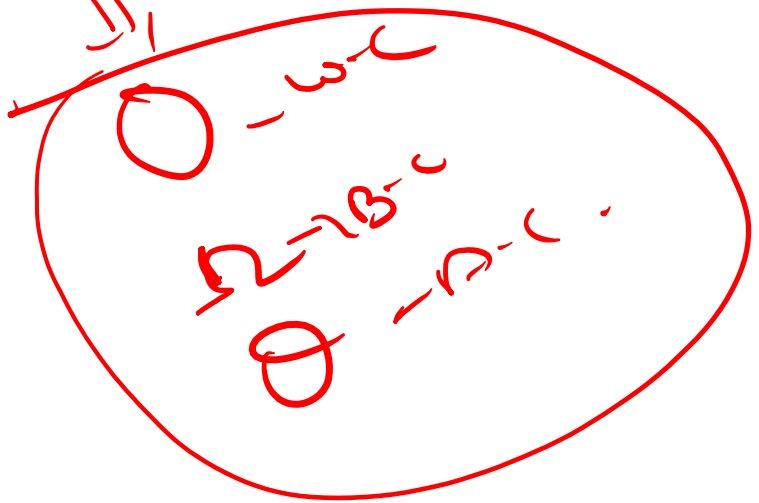
$3 < 4^1 \quad 3)a \quad T(n) = \Theta(n^1 \log^1 n)$.

$= \Theta(n \log n)$

Analysis of the algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of the analysis of algorithms is the required time or performance.

Along with the asymptotic notation analysis there are a few more methodologies there

- Amortized analysis
- Probabilistic analysis



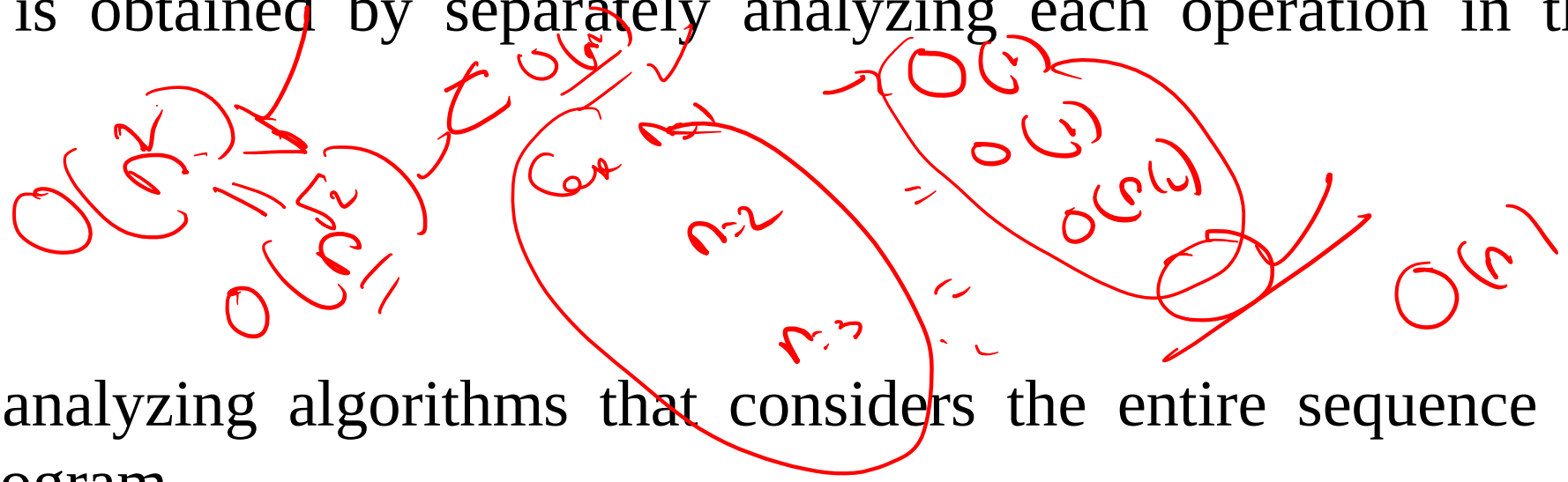
Probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or a computational problem.

- It starts from an assumption about a probabilistic distribution of the set of all possible inputs.
- This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm

Amortized analysis

A sequence of operations applied to the input of size a averaged over time.

Means, to obtain a tighter bound on the overall or average cost per operation in the sequence than is obtained by separately analyzing each operation in the sequence.



OR

It is a method of analyzing algorithms that considers the entire sequence of operations of the program.

It allows for the establishment of a worst-case bound for the performance of an algorithm irrespective of the inputs by looking at all of the operations.

This analysis is most commonly discussed using big O notation.

When Amortized analysis is preferable:

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster.

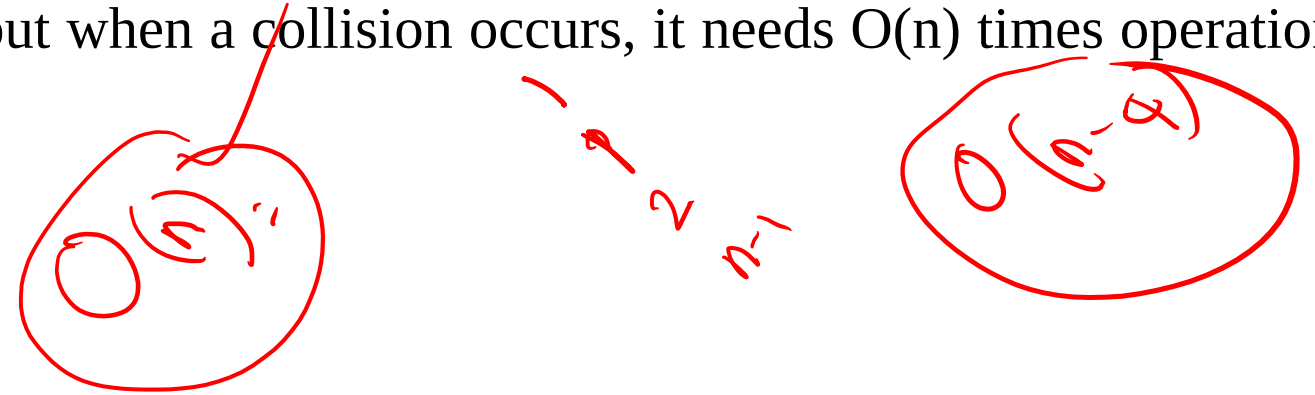
In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.

Situation scenario Explanation:

Means:

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.

In the Hash-table, the most of the time the searching time complexity is O(1), but sometimes it executes O(n) operations. When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task, but when a collision occurs, it needs O(n) times operations for collision resolution.



There are ***three main techniques*** used for amortized analysis:

The *aggregate method*, where the total running time for a sequence of operations is analyzed.

$c_i = i$ if $i-1$ is a power of 2

$c_i = 1$ otherwise

The *accounting* (or banker's) method, where we impose an extra charge on inexpensive operations and use it to pay for expensive operations later on.

The potential (or physicist's) method, in which we derive a potential function characterizing the amount of extra work we can do in each step. This potential either increases or decreases with each successive operation, but cannot be negative.

Any

Question?

Thank You!

FOR YOUR ATTENTION