

Design and Analysis of Algorithms

UNIT NO. II

Analysis of Algorithms and Complexity Theory

Analysis: Input size, best case, worst case, average case

- We can have three cases to analyze an algorithm:
 - 1) The Worst Case
 - 2) Average Case
 - 3) Best Case
- In computing, all above cases of an algorithm depends on the size of the user input value. To understand these terms, let's go through them one by one.
- Let's take an example of Linear search algorithm for further study.

```
#include <bits/stdc++.h>
using namespace std;
// Linearly search x in arr[]. If x is present then return the index, otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] == x)
            return i;
    }
    return -1;
}

int main()
{
    int arr[] = { 1, 10, 30, 15 };
    int x = 30;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << x << " is present at index "
         << search(arr, n, x);
    getchar();
    return 0;
}
```

1) Worst Case Analysis:

- In the worst-case analysis, we calculate the upper limit of the execution time of an algorithm. It is necessary to know the case which causes the execution of the maximum number of operations.
- For linear search, the worst case occurs when the element to search for is not present in the array. When x is not present, the search () function compares it with all the elements of `arr []` one by one. Therefore, the temporal complexity of the worst case of linear search would be $\Theta(n)$.

2) Average Case Analysis:

- In the average case analysis, we take all possible inputs and calculate the computation time for all inputs. Add up all the calculated values and divide the sum by the total number of entries.
- We need to predict the distribution of cases. For the linear search problem, assume that all cases are uniformly distributed. So we add all the cases and divide the sum by $(n + 1)$.

3) Best Case Analysis:

- In the best case analysis, we calculate the lower bound of the execution time of an algorithm. It is necessary to know the case which causes the execution of the minimum number of operations. In the linear search problem, the best case occurs when x is present at the first location.
- The number of operations in the best case is constant. The best-case time complexity would therefore be $\Theta(1)$. Most of the time, we perform worst-case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the execution time of an algorithm which is good information.

Conclusion

- The average case analysis is not easy to do in most practical cases and is rarely done. In the average case analysis, we need to predict the mathematical distribution of all possible inputs. The Best Case analysis is wrong. Guaranteeing a lower bound on an algorithm does not provide any information because in the Worst Case scenario an algorithm can take years to run.
- For some algorithms, all cases are asymptotically the same, that is, there is no worst and best case. For example, Sort by merge. Merge sorting performs $\Theta(n \log n)$ operations in all cases. Most of the other sorting algorithms present the worst and best cases.
- For example, in the typical quicksort implementation, the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the table into two halves.
- For insert sorting, the worst case occurs when the array is sorted in reverse order and the best case occurs when the array is sorted in the same order as the output.

Growth Rate of Algorithm

- The rate at which running time increases as a function of input is called Rate of Growth.
- Let us assume that you went to a shop to buy a car and a cycle. If your friend sees you there and asks what you are buying then in general we say buying a car. This is because. cost of a car is too big compared to cost of cycle (approximating the cost of cycle to the cost of a car).

$$\text{Total Cost} = \text{cost_of_car} + \text{cost_of_cycle}$$

$$\text{Total Cost} \approx \text{cost_of_car} \text{ (approximation)}$$

- For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function ignore the low order terms that are relatively insignificant (for large values of input size, n). As an example in the below case, n^4 , $2n^2$, $100n$, and 500 are the individual costs of some function and approximate it to n^4 . Since, n^4 is the highest rate of growth.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

Commonly used Rate of Growth

Below is the list of rate of growths which we will see further.

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$\log n$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in a unsorted array
$n \log n$	Linear Logarithmic	Sorting n items by 'Divide and Conquer'
n^2	Quadratic	Shortest path between 2 nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Upper and Lower Bounds

- Particularly in order theory, an upper bound or majorant of a subset S of some preordered set (K, \leq) is an element of K that is greater than or equal to every element of S . Dually, a lower bound or minorant of S is defined to be an element of K that is less than or equal to every element of S . A set with an upper (respectively, lower) bound is said to be bounded from above or majorized (respectively bounded from below or minorized) by that bound. The terms bounded above (bounded below) are also used in the mathematical literature for sets that have upper (respectively lower) bounds.
- Example: For example, 5 is a lower bound for the set $S = \{5, 8, 42, 34, 13934\}$ (as a subset of the integers or of the real numbers, etc.), and so is 4. On the other hand, 6 is not a lower bound for S since it is not smaller than every element in S .
- The set $S = \{42\}$ has 42 as both an upper bound and a lower bound; all other numbers are either an upper bound or a lower bound for that S .

Asymtotic Growth

- The rate at which function grows.
- It means complexity of the fuction or the amount of resource(Time & Space) it takes up to complete the execution.
- Classification of growth rate:
 1. Growing with the same rate
 2. Growing with the slower rate
 3. Growing with the faster rate
- Three notations are used to represent complexity of algorithm O, Ω, Θ .

Asymptotic Analysis

- Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.
- For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and the other way is Binary Search (order of growth is logarithmic).

- To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer A and Binary Search on a slow computer B and we pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds. Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.
- For small values of input array size n , the fast computer may take less time. But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after a certain value of input size.
- Here are some running times for this example:
 - Linear Search running time in seconds on A: $0.2 * n$
 - Binary Search running time in seconds on B: $1000 * \log(n)$

- The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

O, Ω, Θ

1) Θ Notation

- 1) The theta (Θ) Notation: The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.
- A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, consider the following expression.

$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$

- Dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.

- For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.
 $\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0\}$
- The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1 * g(n)$ and $c_2 * g(n)$ for large values of n ($n \geq n_0$). The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0 .

2) Big O Notation:

- Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
- For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.
- If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
 - 1. The worst-case time complexity of Insertion Sort is $\Theta(n^2)$.
 - 2. The best case time complexity of Insertion Sort is $\Theta(n)$.

- The Big O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.
- $O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \}$

3) Ω Notation:

- Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- Ω Notation can be useful when we have a lower bound on the time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions.

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

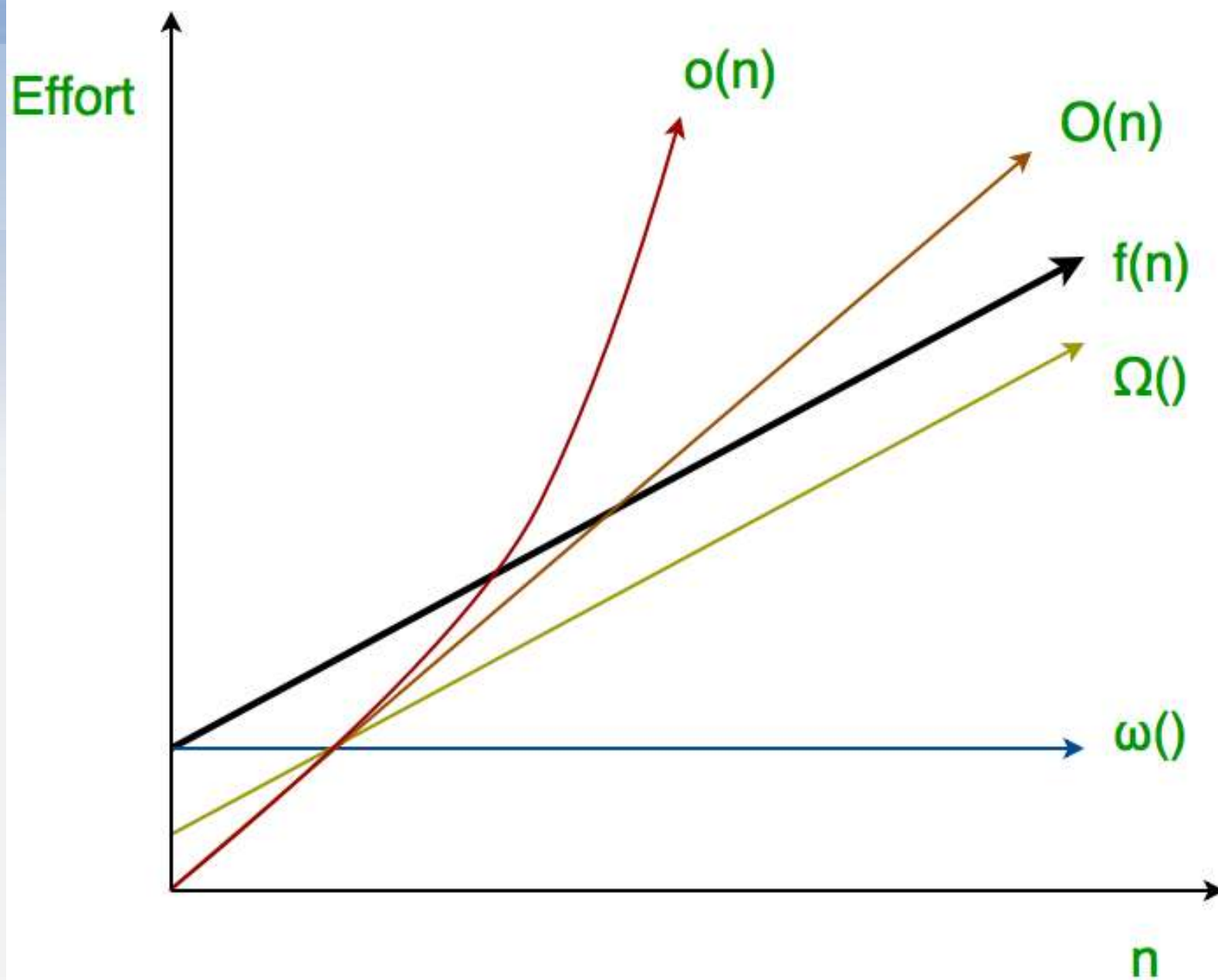
- Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as $\Omega(n)$, but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

4) Little (o) Notation

- Big-O is used as a tight upper bound on the growth of an algorithm's effort (this effort is described by the function $f(n)$), even though, as written, it can also be a loose upper bound. "Little-o" ($o()$) notation is used to describe an upper bound that cannot be tight.
- little $o()$ means loose upper-bound of $f(n)$. Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

5) Little ω Notation

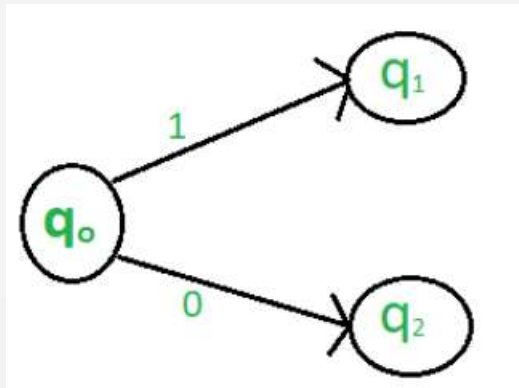
- The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-O and Little o except that now we are looking at the lower bounds. Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth.
- We use ω notation to denote a lower bound that is not asymptotically tight. And, $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.



Relationship between all asymptotic notations

Deterministic Algorithm

- A deterministic algorithm is an algorithm that, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. Deterministic algorithms can be run on real machines efficiently.
- Formally, a deterministic algorithm computes a mathematical function; a function has a unique value for any input in its domain, and the algorithm is a process that produces this particular value as output.



Non Deterministic Algorithm

- A non-deterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm which produces only a single output for the same input even on different runs, a non-deterministic algorithm travels in various routes to arrive at the different outcomes.
- Non-deterministic algorithms are useful for finding approximate solutions, when an exact solution is difficult or expensive to derive using a deterministic algorithm.
- One example of a non-deterministic algorithm is the execution of concurrent algorithms with race conditions, which can exhibit different outputs on different runs.
- One example of a non-deterministic algorithm is the execution of concurrent algorithms with race conditions, which can exhibit different outputs on different runs.

Non Deterministic Algorithm

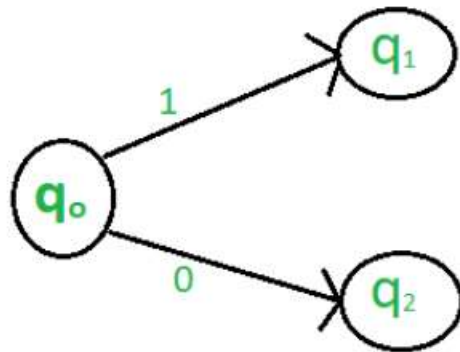
- A variety of factors can cause an algorithm to behave in a way which is not deterministic, or non-deterministic:
 - ✓ If it uses an external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.
 - ✓ If it operates in a way that is timing-sensitive, for example, if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.
 - ✓ If a hardware error causes its state to change in an unexpected way.

Deterministic Algorithm

For a particular input the computer will give always same output.

Can solve the problem in polynomial time.

Can determine the next step of execution.



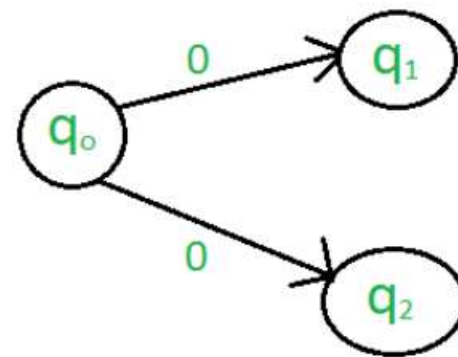
Deterministic Algorithm

Non-deterministic Algorithm

For a particular input the computer will give different output on different execution.

Can't solve the problem in polynomial time.

Cannot determine the next step of execution due to more than one path the algorithm can take.



Non-Deterministic Algorithm

Complexity Classes

- In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as Complexity Classes.
- In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.
- The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

P-Problems

- The P in the P class stands for Polynomial Time. It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.
- The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where k is constant.
- These problems are called tractable, while others are called intractable or superpolynomial.
- Features:
 - ✓ The solution to P problems is easy to find.
 - ✓ P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

P-Problems

- This class contains many natural problems like:
 - ✓ Calculating the greatest common divisor.
 - ✓ Finding a maximum matching.
 - ✓ Decision versions of linear programming.

NP -Problems

- The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- Features:
 - ✓ The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
 - ✓ Problems of NP can be verified by a Turing machine in polynomial time.
- Example:
 - ✓ Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons.
 - ✓ This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

NP -Problems

- It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.
- This class contains many problems that one would like to be able to solve effectively:
 - ✓ Boolean Satisfiability Problem (SAT)
 - ✓ Hamiltonian Path Problem
 - ✓ Graph coloring

NP-hard Problem

- An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.
- Features:
 - ✓ All NP-hard problems are not in NP.
 - ✓ It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
 - ✓ A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

NP-Hard Problem

- Some of the examples of problems in Np-hard are:
 - ✓ Halting Problem.
 - ✓ Qualified Boolean formulas.
 - ✓ No Hamiltonian cycle.

NP-Complete

- A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.
- Features:
 - ✓ NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
 - ✓ If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.
- Some example problems include:
 - ✓ Decision version of 0/1 Knapsack.
 - ✓ Hamiltonian Cycle.
 - ✓ Satisfiability.
 - ✓ Vertex cover.

NP-Hard and NP-Complete

Polynomial Time

Linear Search — n

Binary Search — $\log n$

Insertion Sort — n^2

Merge Sort — $n \log n$

Matrix Multiplication — n^3

Exponential Time

0/1 Knapsack — 2^n

Traveling SP — 2^n

Sum of Subsets — 2^n

Graph Coloring — 2^n

Hamiltonian Cycle — 2^n

NP-Hard and NP-Complete

Nondeterministic

```
Algorithm NSearch(A, n, key)
{
    j = choice();
    if (key = A[j])
    {
        write(j);
        Success();
    }
    write(0);
    Failure();
}
```

Exponential Time

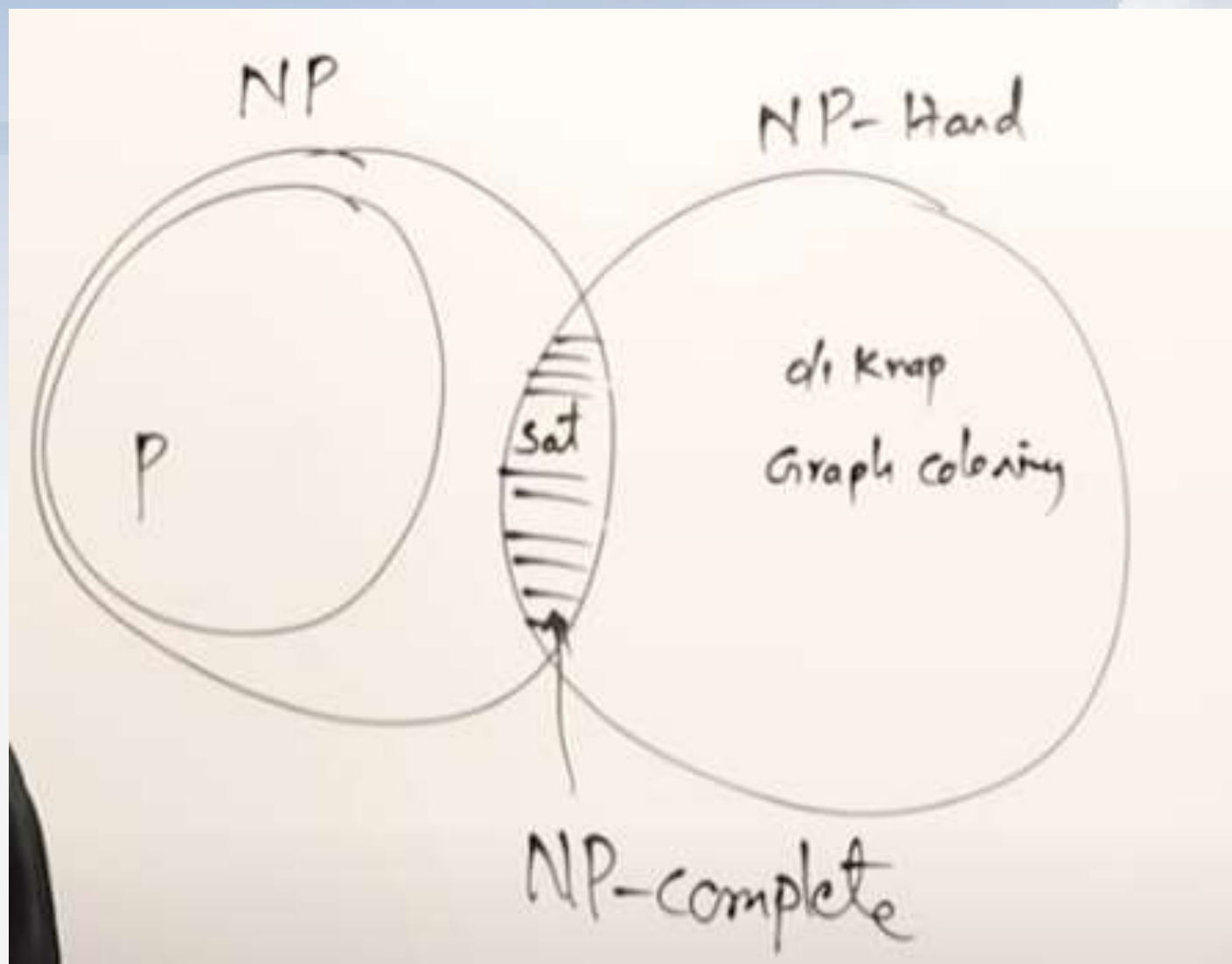
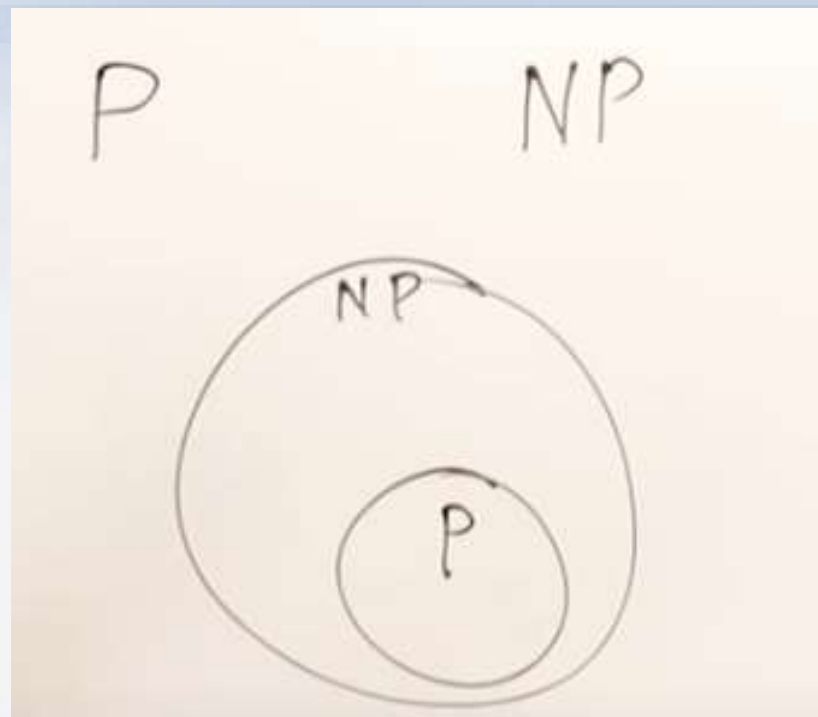
0/1 Knapsack — 2^n

Traveling SP — 2^n

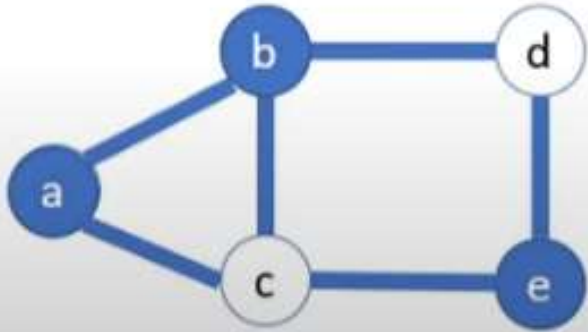
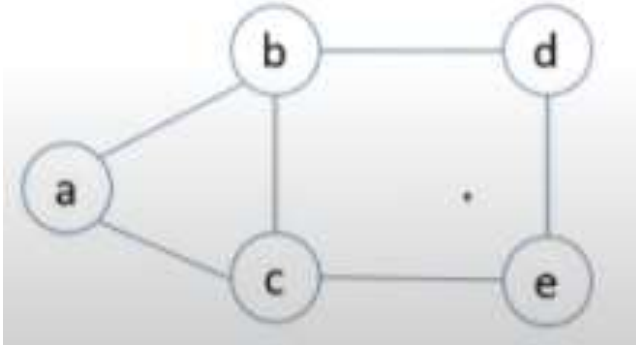
Sum of Subsets — 2^n

Graph Coloring — 2^n

Hamiltonian Cycle — 2^n



Vertex Cover Problem



Vertex cover = {a,b,e} & it's of size 3.

A **vertex cover** of an undirected graph $G=(V,E)$ is a sub set $V' \subseteq V$ such that if $(u,v) \in E$, then $u \in V'$ or $v \in V'$, or both.

That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E .

The **size** of a vertex cover is the number of vertices in it.

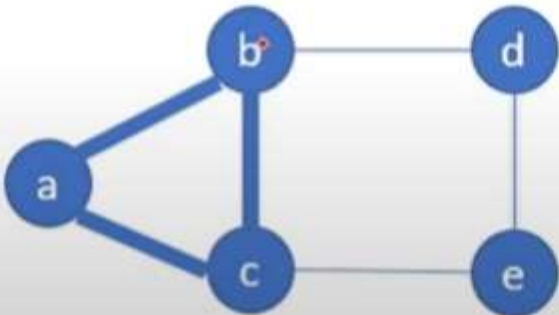
A naive algorithm for determining whether a graph $G=(V,E)$ has a vertex cover of size k is to list all k -subsets of V , and check each one to see whether it covers all edges.

Is the graph contain a vertex cover of size k ?

Clique Problem

Is the graph contain a clique of size k ?

A clique, C , in an undirected graph $G = (V, E)$ is a subset of the vertices, $C \subseteq V$, such that every two distinct vertices are adjacent. This is equivalent to the condition that the induced subgraph of G induced by C is a complete graph.

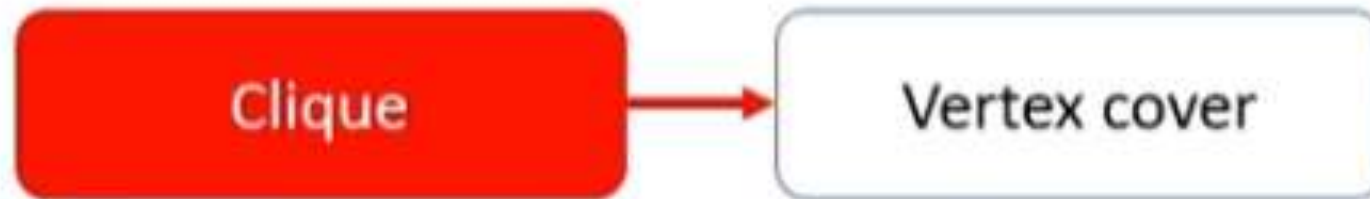


Here this graph contains 2 cliques as $\{a, b, c\}$ and $\{b, c, d, e\}$ so $k = 2$.

Vertex Cover Problem: NP Completeness Proof

Vertex cover \in NP

Suppose we are given a graph $G=(V,E)$ and an integer k . Let $V' \subseteq V$ and $|V'|=k$. Then it checks for each edge $(u,v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify this in polynomial time.



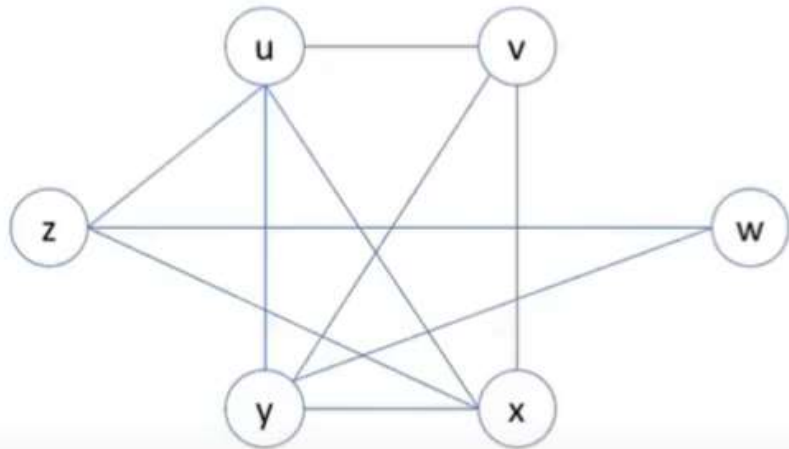
Given a graph $G=(V, E)$ we define complement of G as $\bar{G} = (V, \bar{E})$,
Where $\bar{E}=\{(u, v): u, v \in V, u \neq v \text{ and } (u, v) \notin E\}$

The reduction algorithm takes as input an instance (G, k) of the clique problem.

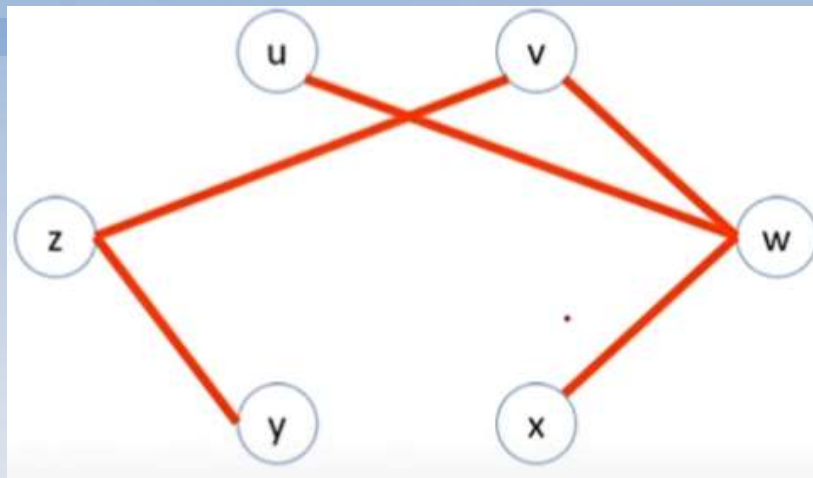
It computes the complement \bar{G} , in polynomial time.

$(\bar{G}, |V| - k)$ is an instance of vertex cover problem.

The graph G has clique of size k if and only if graph G has vertex cover of size $|V| - k$



Graph G is having clique of size $k = 4$



Graph G bar is having vertex cover of size 2.

:Example

But graph G has clique of size k if and only if graph G has vertex cover of size $|V| - k$.

Here, $|V| = 6$, $k = 4$, So $|V| - k = 6 - 4 = 2$

And as per graph, vertex cover is also equal to 2.

So, we have reduced Clique problem to Vertex Cover Problem.

So, vertex cover problem is also NP-hard.

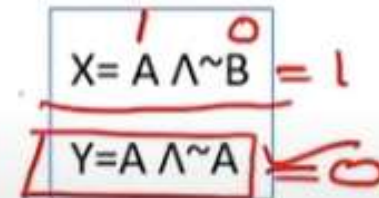
Here we proved that vertex cover problem is Np as well as NP-hard so it is NP-Complete Problem.

SAT Problem

Is there exists an interpretation that satisfies a given Boolean formula?

A Boolean formula is said to be satisfiable if a truth assignment that evaluate the formula to be 1

if the formula has k inputs, then we would have to check up to 2^k possible assignments



A handwritten diagram illustrating a truth table for a SAT problem. It consists of two rows, each enclosed in a box. The first row is labeled $X = A \wedge \sim B$ and has a red '1' written above the 'A' and a red '0' written above the ' $\sim B$ '. To the right of the box, a red '=' is followed by a red '1'. The second row is labeled $Y = A \wedge \sim A$ and has a red '0' written to its right. A red arrow points from the '0' to the right side of the box.

$X = A \wedge \sim B$	1
$Y = A \wedge \sim A$	0

3 SAT Problem

A Boolean formula is in conjunctive normal form, or CNF, if it is expressed as conjunctions (by AND) of clauses, each of which is the disjunction (by OR) of one or more literals

A Boolean formula is in 3-conjunctive normal form, or 3-CNF-SAT, if each clause has exactly three distinct literals. For example, the Boolean formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$ is in 3-CNF-SAT

In 3-CNF-SAT, we are asked whether a given boolean formula ϕ in 3-CNF-SAT is satisfiable

if the formula has k inputs, then we would have to check up to 2^k possible assignments

3-SAT NP-Completeness Proof

3CNF SAT \in NP

3CNF SAT can be verified in polynomial time by simply replacing each variable in the formula with its corresponding value and then evaluates the expression.

SAT

3 CNF SAT

SAT \leq_p 3 CNF SAT

$$F = X + YZ$$

$$= (X + Y) (X + Z)$$

$$= (X + Y + ZZ') (X + YY' + Z)$$

$$= (X + Y + Z) (X + Y + Z') (X + Y + Z) (X + Y' + Z)$$

$$= (X + Y + Z) (X + Y + Z') (X + Y' + Z)$$

Analysis of Iterative and Recursive Algorithm

1. **Time Complexity:** Finding the Time complexity of Recursion is more difficult than that of Iteration.
- ✓ **Recursion:** Time complexity of recursion can be found by finding the value of the nth recursive call in terms of the previous calls. Thus, finding the destination case in terms of the base case, and solving in terms of the base case gives us an idea of the time complexity of recursive equations.
 - ✓ **Iteration:** Time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.

Analysis of Iterative and Recursive Algorithm

- 2. Usage:** Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.
- ✓ **Recursion:** Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.
 - ✓ **Iteration:** Iteration is repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.

Analysis of Iterative and Recursive Algorithm

3.Overhead: Recursion has a large amount of Overhead as compared to Iteration.

- ✓ Recursion: Recursion has the overhead of repeated function calls, that is due to repetitive calling of the same function, the time complexity of the code increases manyfold.
- ✓ Iteration: Iteration does not involve any such overhead.

Analysis of Iterative and Recursive Algorithm

- 4. Infinite Repetition:** Infinite Repetition in recursion can lead to CPU crash but in iteration, it will stop when memory is exhausted.
- ✓ **Recursion:** In Recursion, Infinite recursive calls may occur due to some mistake in specifying the base condition, which on never becoming false, keeps calling the function, which may lead to system CPU crash.
 - ✓ **Iteration:** Infinite iteration due to mistake in iterator assignment or increment, or in the terminating condition, will lead to infinite loops, which may or may not lead to system errors, but will surely stop program execution any further.

Example of factorial using Recursion

```
// ----- Recursion -----  
// method to find factorial of given number  
int factorialUsingRecursion(int n)  
{  
    if (n == 0)  
        return 1;  
  
    // recursion call  
    return n * factorialUsingRecursion(n - 1);  
}
```

Example of factorial using Iteration

```
// ----- Iteration -----  
// Method to find the factorial of a given number  
int factorialUsingIteration(int n)  
{  
    int res = 1, i;  
  
    // using iteration  
    for (i = 2; i <= n; i++)  
        res *= i;  
  
    return res;  
}
```

Difference between Iterative and Recursive Algorithm

Property	Recursion	Iteration
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high (generally exponential) time complexity.	Relatively lower time complexity (generally polynomial-logarithmic).