SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 OF THE UGC ACT, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

Course

CSE318 - Algorithm Design Strategies & Analysis

Topic

Unit-4

Shri B. Srinivasan

Assistant Professor-III, School of Computing

SASTRA Deemed To Be University

Topics

- ✓ Non-deterministic Algorithms
  - ✓ Classes of NP-Hard & NP-Complete
  - ✓ Cook's Theorem
  - ✓ Clique Decision Problem
  - ✓ Node Cover Decision Problem
  - ✓ Travelling Salesperson Problem
- ✓ Approximation Algorithms
  - ✓ Scheduling Independent Tasks
  - ✓ Bin-Packing Problem
  - ✓ Interval Partitioning
- ✓ Randomized Algorithms
- ✓ Classes beyond NP – P SPACE
- ✓ Introduction to Quantum Algorithms

Classes of Algorithms

Topics

- ✓ Polynomial vs Exponential
- ✓ Deterministic vs Non-Deterministic
- ✓ P
- ✓ NP
- ✓ NP Hard
- ✓ NP Complete
- ✓ P vs NP
- ✓ NP-Hard vs NP-Complete
- ✓ P, NP, Np-Hard & NP-Complete - Relations

Polynomial vs Exponential

- ✓ Polynomial Time Algorithms
- ✓ Solution can be determined in polynomial time.
  - Linear Search – n
  - Binary Search – log n
  - Selection Sort – n^2
  - Matrix multiplication – n^3
  - Quick sort – n log n

- ✓ Exponential Time Algorithms
- ✓ Solution can be determined in exponential time.
  - 0 / 1 Knapsack – 2^n
  - TSP – 2^n
  - Sum of subset – 2^n
  - Graph coloring – 2^n
  - Hamiltonian Cycle – n^n

Deterministic vs Non-deterministic

- ✓ Usually if we write an algorithm, it is deterministic. i.e. we know clearly that each and every statements in the algorithm, how it works?
- ✓ If we know clearly how the algorithm works, it is deterministic
- ✓ Non-deterministic algorithms are the algorithms, those working behavior is not known to us.
- ✓ A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm.

Deterministic vs Non-deterministic

- ✓ Non-deterministic algorithms are useful for finding approximate solutions, when an exact solution is difficult or expensive to derive using a deterministic algorithm

Deterministic vs Non-deterministic

- ✓ In deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states.
- ✓ But in case of non-deterministic algorithm, for the same input, the compiler may produce different output in different runs. *i.e. Stochastic in nature*. "Stochastic" is a description that refers to *outcomes based upon random probability*.
- ✓ In fact non-deterministic algorithms can solve the problem in polynomial time and can't determine what is the next step.
- ✓ The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a *degree of randomness to it*.

Non-deterministic Algorithms - Examples

✓ Some of the terms related to the non-deterministic algorithm are:

- choice(X) : chooses any value randomly from the set X.
- failure() : denotes the unsuccessful solution.
- success() : Solution is successful and current thread terminates.

Example for Non-deterministic

Example Consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is Algorithm 11.1.

```
1   j := Choice(1, n);
2   if A[j] = x then {write (j); Success();}
3   write (0); Failure();
```

Algorithm Nondeterministic search

Example for Non-deterministic

```
1   Algorithm NSort(A, n)
2   // Sort n positive integers.
3   {
4       for i := 1 to n do B[i] := 0; // Initialize B[ ].
5       for i := 1 to n do
6       {
7           j := Choice(1, n);
8           if B[j] ≠ 0 then Failure();
9           B[j] := A[i];
10      }
11      for i := 1 to n − 1 do  // Verify order.
12          if B[i] > B[i + 1] then Failure();
13      write (B[1 : n]);
14      Success();
15  }
```

Algorithm Nondeterministic sorting

Example for Non-deterministic

Example [Satisfiability] Let $x_1, x_2, \ldots$ denote boolean variables (their value is either true or false). Let $\bar{x}_i$ denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations **and** and **or**. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ and $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. The symbol $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) if and only if it is represented as $\wedge_{i=1}^{k} c_i$, where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in *disjunctive normal form* (DNF) if and only if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF whereas $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The **satisfiability** problem is to determine whether a formula is true for some assignment of truth values to the variables. *CNF-satisfiability* is the satisfiability problem for CNF formulas.

Example for Non-deterministic

```
1    Algorithm Eval(E, n)
2    // Determine whether the propositional formula E is
3    // satisfiable. The variables are x_1, x_2, ..., x_n.
4    {
5        for i := 1 to n do  // Choose a truth value assignment.
6            x_i := Choice(false, true);
7        if E(x_1, ..., x_n) then Success();
8        else Failure();
9    }
```

Algorithm 11.6 Nondeterministic satisfiability

Decision and Optimization Problems

- ✓ **Decision Problem**: computational problem with intended output of "yes" or "no", 1 or 0
- ✓ **Optimization Problem**: computational problem where we try to maximize or minimize some value

Decision and Optimization Problems

- ✓ Most optimization problems can be phrased as decision problems
- ✓ Introduce parameter k and ask if the optimal value for the problem is at most or at least k. Turn optimization into decision.

Example :

Assume we have a decision algorithm X for 0/1 Knapsack problem with capacity M,

i.e. Algorithm X returns "Yes" or "No" to the question

"Is there a solution with profit $\geq$ P subject to knapsack capacity $\leq$ M?"

Class of "P" Problems

- ✓ Class P consists of (decision) problems that are solvable in polynomial time
- ✓ Deterministic in nature
- ✓ Polynomial-time algorithms
  - Worst-case running time is $O(n^k)$, for some constant k
- ✓ Examples of polynomial time:
  - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- ✓ Examples of non-polynomial time:
  - $O(2^n)$, $O(n^n)$, $O(n!)$

Tractable/Intractable Problems

- ✓ Problems in P are also called tractable
- ✓ Problems not in P are intractable or unsolvable
  - Can be solved in reasonable time only for small inputs
  - Or, can not be solved at all
- ✓ Are non-polynomial algorithms always worst than polynomial algorithms?
  - $n^{1,000,000}$ is technically tractable, but really impossible
  - $n^{\log \log \log n}$ is technically intractable, but easy

Intractable Problems

- ✓ Can be classified in various categories based on their degree of difficulty, e.g.,
  - NP
  - NP-complete
  - NP-hard

- Let's define NP algorithms and NP problems …

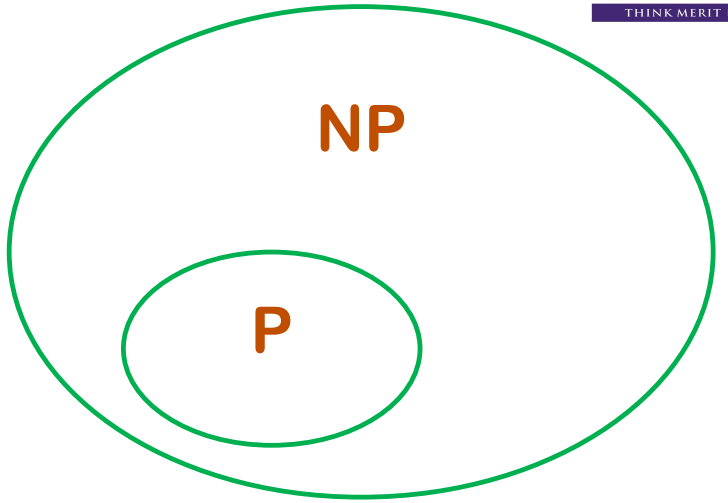
Class of "NP" Problems

- NP stands for non-deterministic polynomial time
- NP is the set of all decision problems solvable by non-deterministic algorithms in polynomial time.
- choose(b): choose a bit in a non-deterministic way and assign to b
- If someone tells us the solution to a problem, we can verify it in polynomial time
- Two Properties:
  1. *non-deterministic* method to *generate possible solutions*,
  2. *deterministic* method to *verify that the solution is correct* in polynomial time.

Note: NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.

P vs NP

- ✓ P is the subset of NP
- ✓ It means, the problems which are known as "Deterministic" was a part of "Non-Deterministic".
- ✓ Example: Linear search now it is known as "Deterministic". But earlier it was "Non-Deterministic", it means, by that time the "Choice()" was not-deterministic. Somebody found the deterministic solution for "Choice()" in linear search, so it become "Deterministic"



- ✓ Today, whichever known as "Deterministic, Yesterday, it was "Non-deterministic".
- ✓ Today, whichever known as "Non-deterministic", Tomorrow, it may become "Deterministic"

NP-Hard vs NP-Complete

To understand NP-Hard problem, you must know about *Reduction*?
What is Reduction?

Suppose we have 2 decision problems, P1 and P2.

Problem : P1 Input: I1 Algorithm: A??? (Not exist)	Problem : P2 Input: I2 Algorithm: B (Exist)

Suppose if we are able to solve the problem P1 by using the algorithm of P2, then no need to write algorithm for P1. i.e., A is not required.

So, *P1 is said to be REDUCIBLE to P2* as we can solve P1 problem with the help of the algorithm of P2.

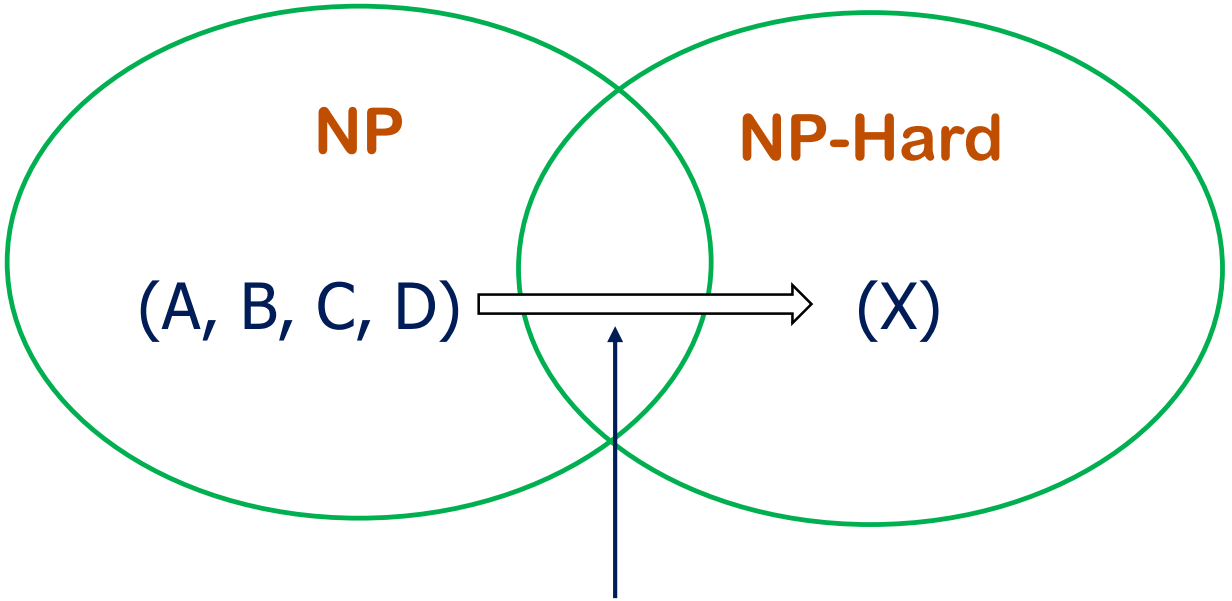
It is denoted as, $P1 \propto P2$

(But this conversion must be in polynomial)

NP-Hard vs NP-Complete

NP-Hard:

A problem (X) is said to be NP-Hard, *if all problems in NP (A, B, C, D) are polynomial time reducible to it*, even though it may not be in NP itself.



If
A ∞ X
AND
B ∞ X
AND
C ∞ X
AND
D ∞ X
Then
X is said to be NP-Hard.
It need not to be NP

NP-Hard vs NP-Complete

Example for NP-Hard

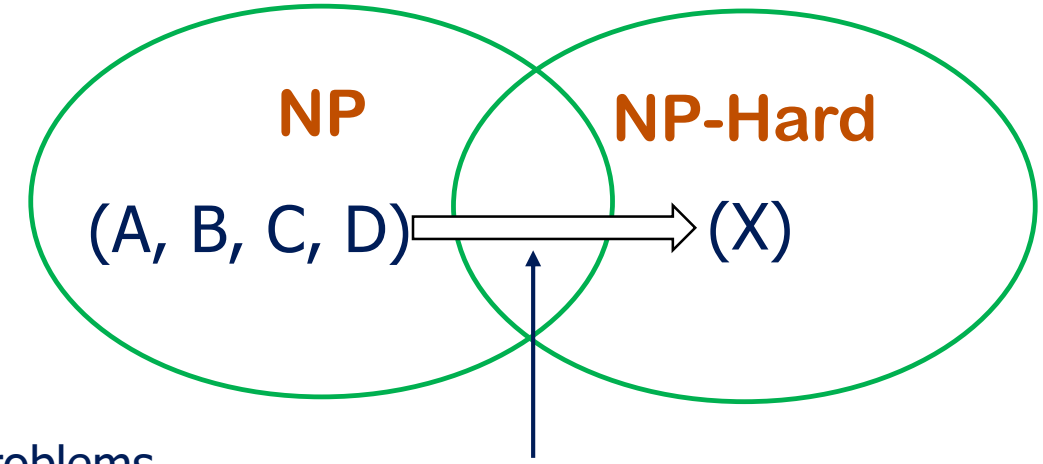
Consider the following problems

1. Satisfiability
2. TSP
3. 0 / 1 Knapsack
4. Sum of subset
5. Graph coloring
6. Hamiltonian Cycle

Among these, "Satisfiability" problem is the base for all other problems. It means, other problems can be solved with the help of the algorithm designed for "Satisfiability" problem.

So, "Satisfiability" is known as NP-Hard problem.

Also, this problem is non-deterministic problem. So, it is also known as NP-Complete.



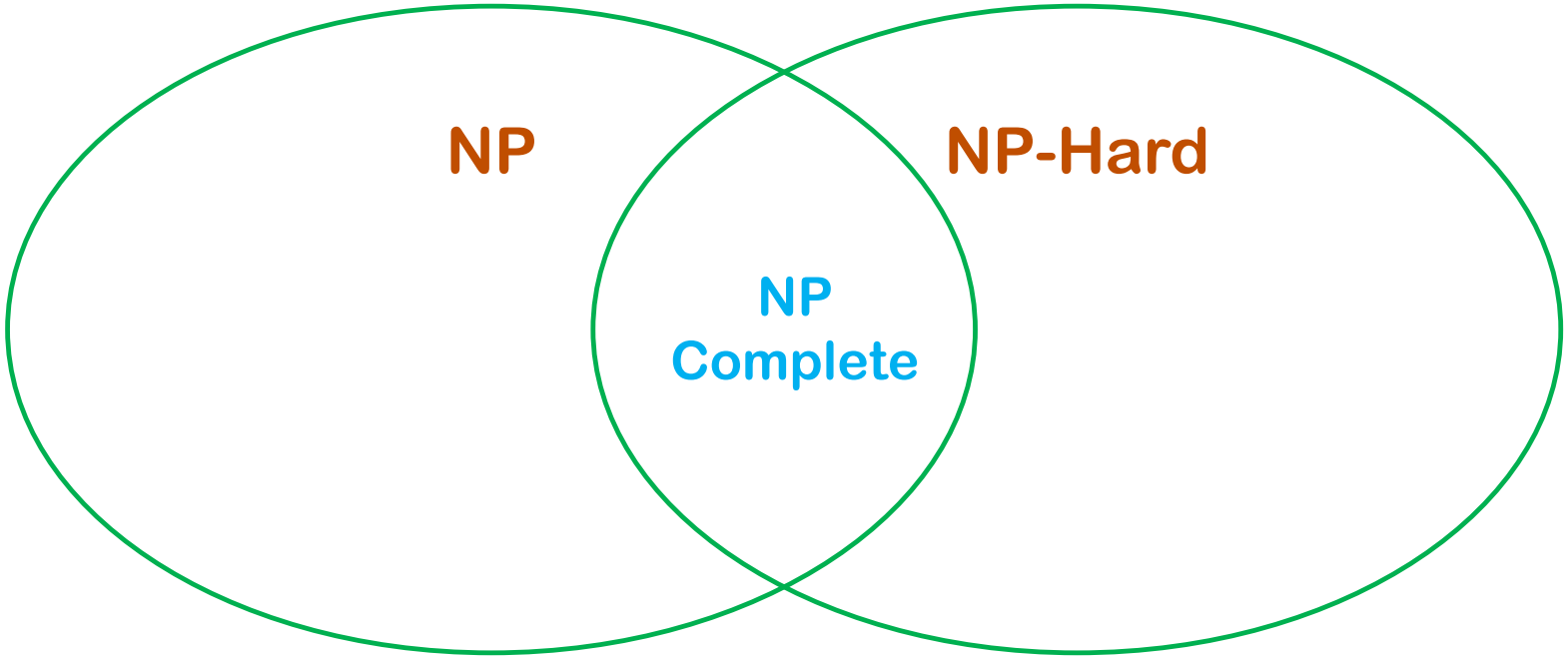
These reductions must be Polynomial

NP-Hard vs NP-Complete

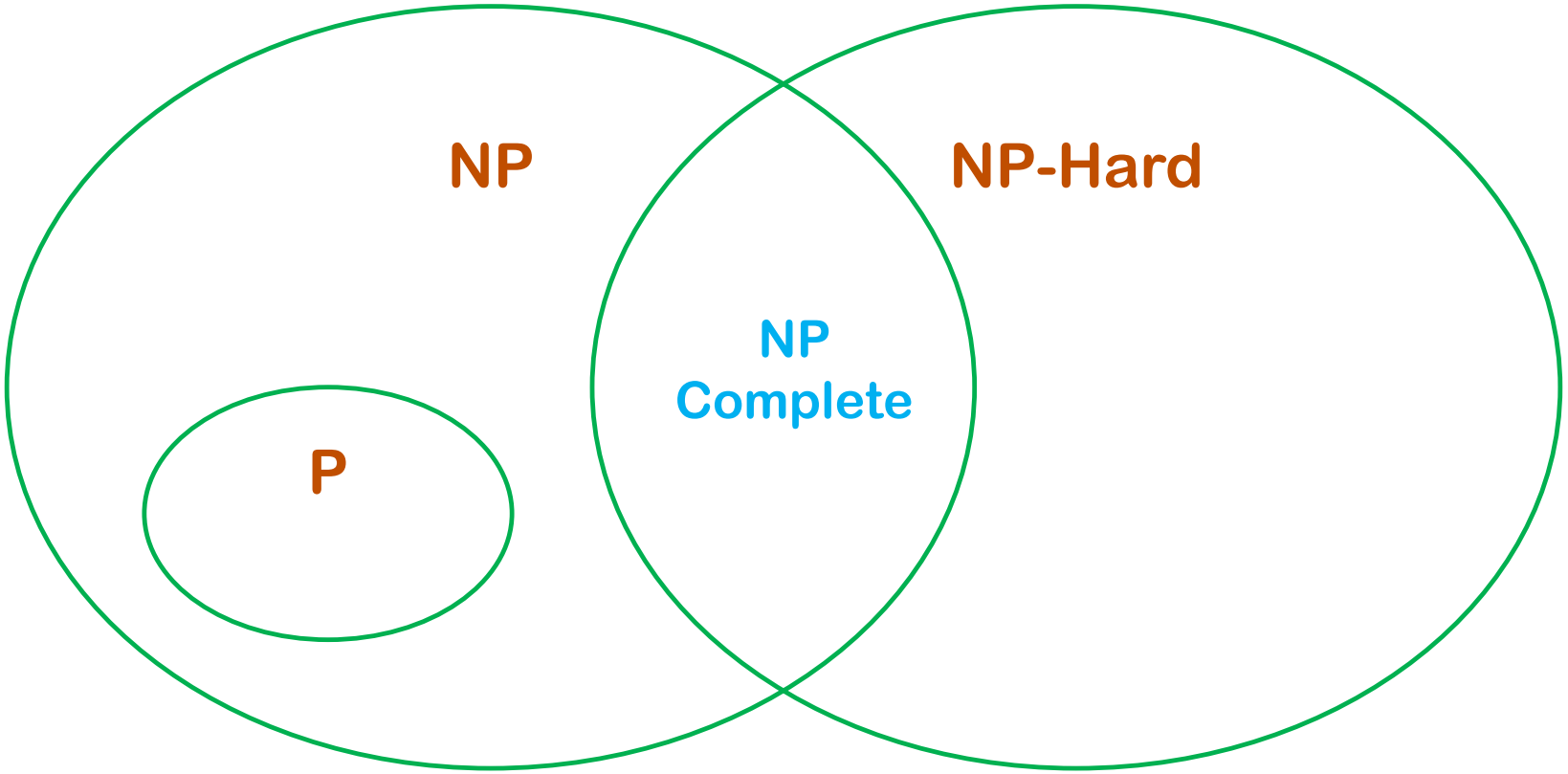
NP-Complete:

A problem is said to be NP-Complete, if it is a part of NP and NP-Hard problems.

i.e., If an algorithm which is known as NP-Hard, is an non-deterministic, then it is also said to be NP-Complete.



P, NP, NP-Hard and NP-Complete

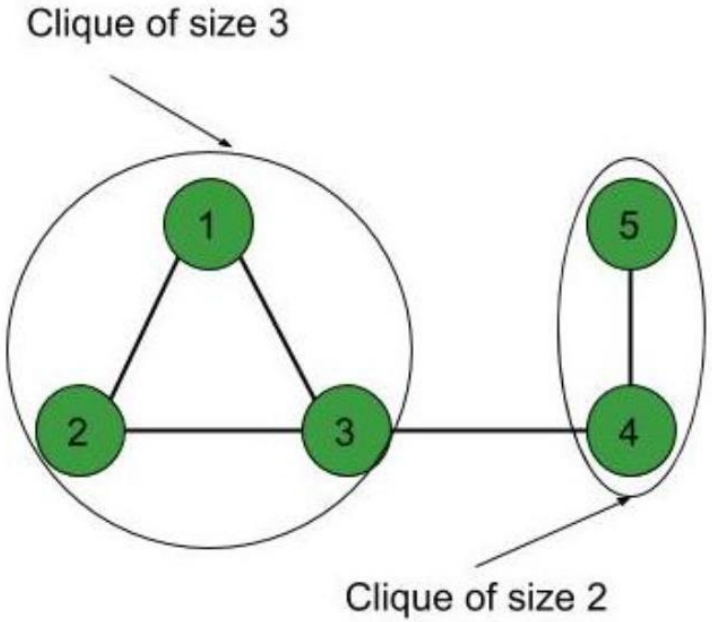


Clique Decision Problem
NP-Complete - Proof

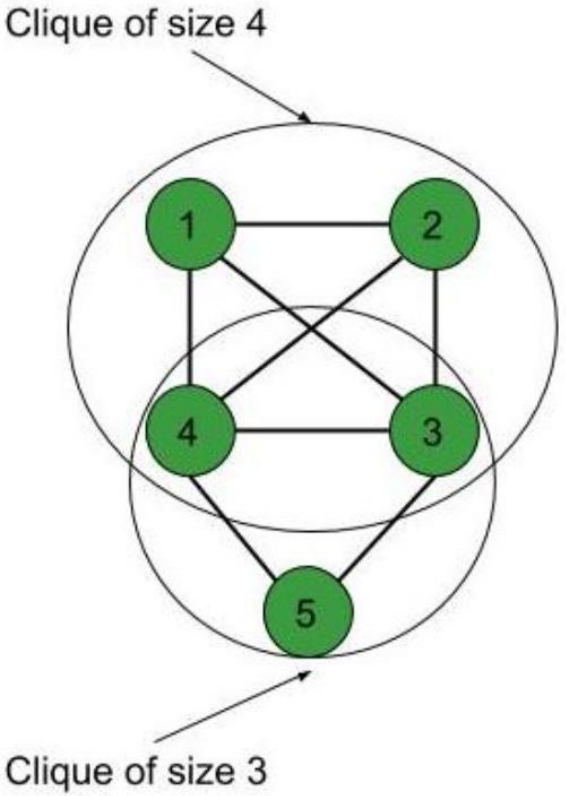
Clique Decision Problem

- ✓ A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph.
- ✓ The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices. This is an optimization problem.
- ✓ Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph or not.

Clique Decision Problem



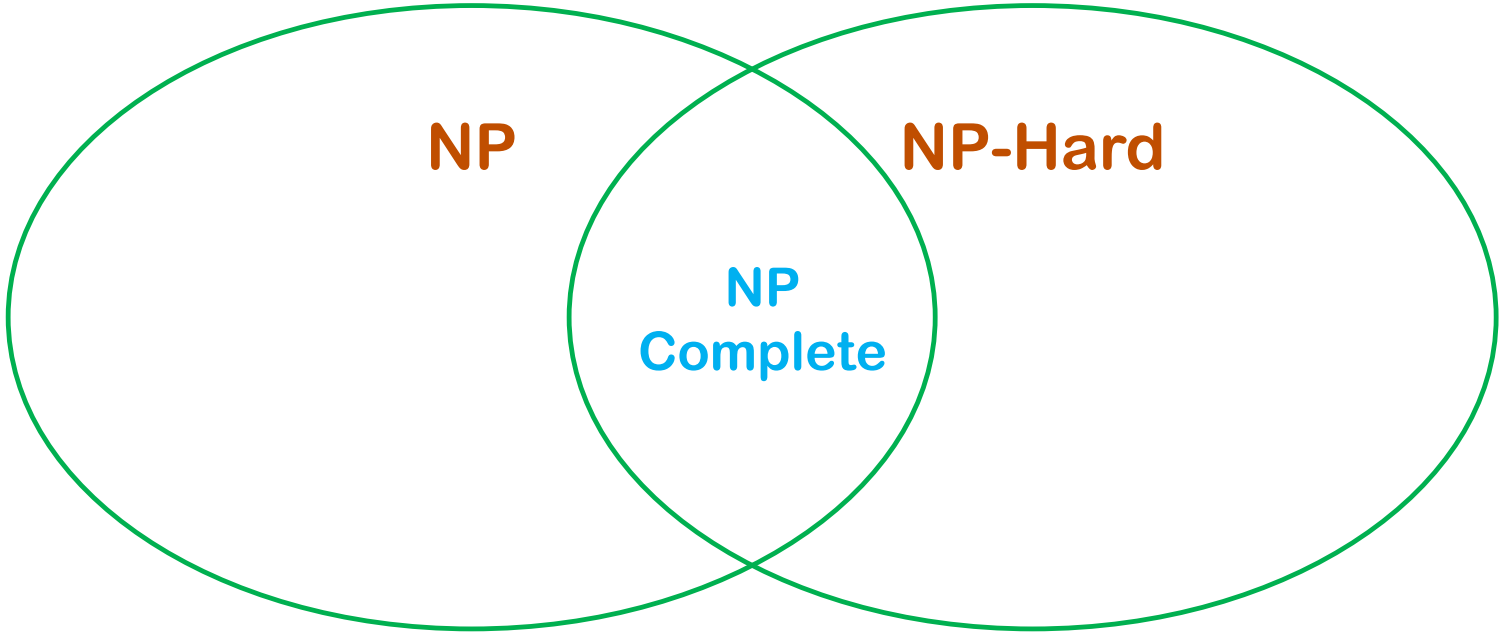
The above graph contains a maximum clique of size 3



The above graph contains a maximum clique of size 4

CDP – NP-Complete - Proof

- ✓ To prove that a problem is NP-Complete, we have to show that it belongs to both NP and NP-Hard Classes.



CDP – NP-Complete - Proof

1. The Clique Decision Problem belongs to NP

If a problem belongs to the NP class, then it should have polynomial-time verifiability, that is given a certificate, we should be able to verify in polynomial time if it is a solution to the problem.

Proof:

Certificate – Let the certificate be a set S consisting of nodes in the clique and S is a subgraph of G.

CDP – NP-Complete - Proof

Verification –

- ✓ We have to check if there exists a clique of size k in the graph.
- ✓ Time to verify if number of nodes in S equals k → O(1)
- ✓ Time to verify whether each vertex has an out-degree of (k-1) → $O(k^2)$ time.

  Therefore, to check if the graph formed by the k nodes in S is complete or not, it takes $O(k^2) = O(n^2)$ time (since k<=n, where n is number of vertices in G).
- ✓ Therefore, the Clique Decision Problem has polynomial time verifiability and hence belongs to the NP Class.

CDP – NP-Complete - Proof

2. The Clique Decision Problem belongs to NP-Hard –

- ✓ A problem L belongs to NP-Hard if every NP problem is reducible to L in polynomial time.
- ✓ Let the Clique Decision Problem by C.
- ✓ To prove that C is NP-Hard, we need to take an already known NP-Hard problem, say S, and reduce it to C for a particular instance.
- ✓ Need to prove, S∞C
- ✓ If this reduction can be done in polynomial time, then C is also an NP-Hard problem.

CDP – NP-Complete - Proof

- ✓ The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by the Cook's theorem.
- ✓ Every problem in NP can be reduced to S in polynomial time.
- ✓ If S is reducible to C in polynomial time, every NP problem can be reduced to C in polynomial time, thereby proving C to be NP-Hard.

CDP – NP-Complete - Proof

Proof: The BSP reduces to the CDP: BSP ∞ CDP

Let the Boolean expression be:

$$F = (x1 \vee x2) \wedge (x1' \vee x2') \wedge (x1 \vee x3)$$

C1 C2 C3

Where,

x1, x2, x3 are the variables,

^ denotes logical 'and',

v denotes logical 'or' and

x' denotes the complement of x.

Three clauses – C1, C2 and C3.

CDP – NP-Complete - Proof

- ✓ Need to draw the corresponding graph with the vertices from BSP
  - G = <V,E>
  - V = { <a,i> | aϵCi }
  - E = { (<a,i>,<b,j>) | i!=j and b!=a' }

The vertices: <x1, 1>; <x2, 1>; <x1', 2>; <x2', 2>; <x1, 3>; <x3, 3>

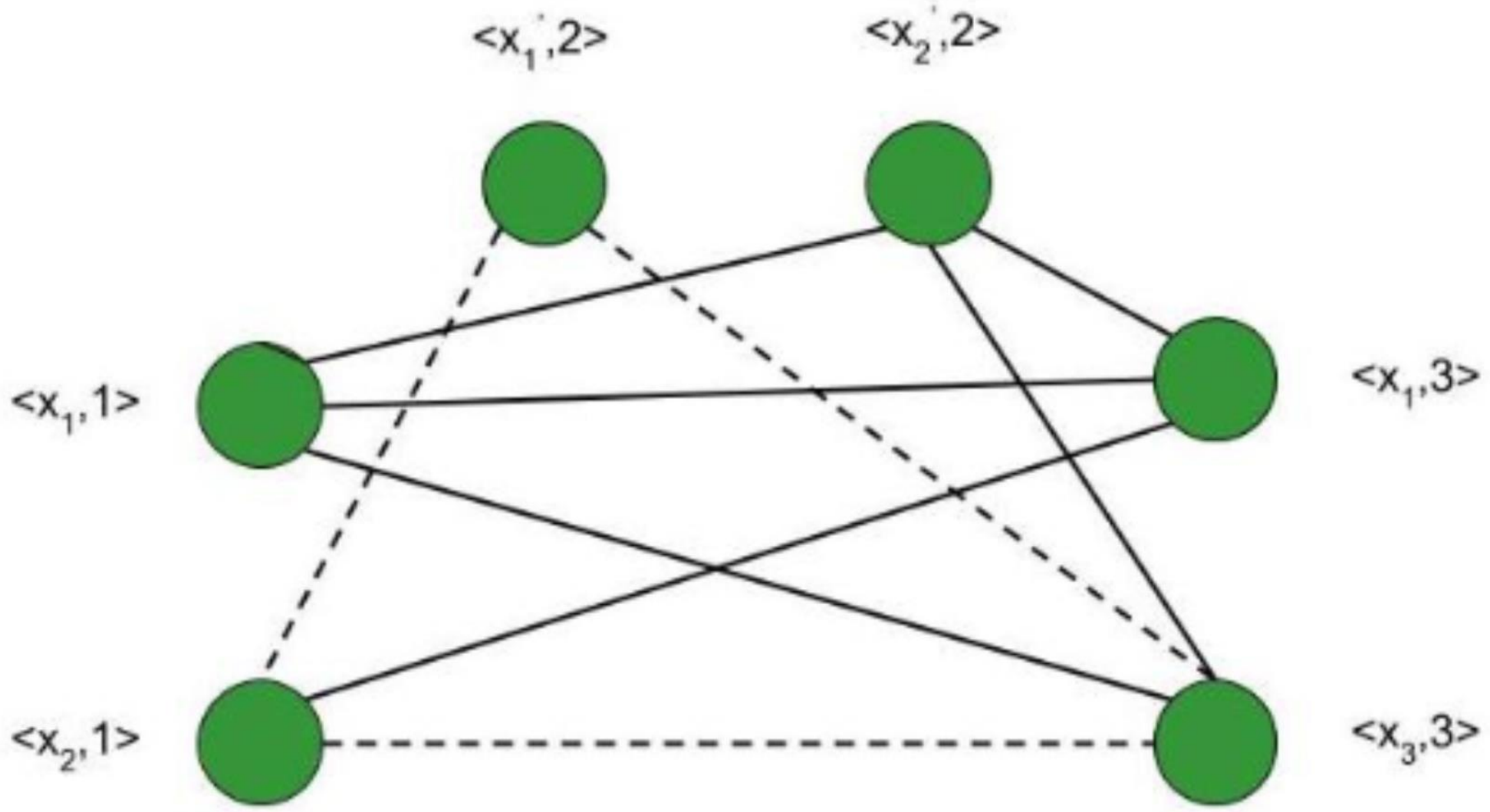
where the second term in each vertex denotes the clause number they belong to.

We connect these vertices such that –

1. No two vertices belonging to the same clause are connected.

2. No variable is connected to its complement.

CDP – NP-Complete - Proof



CDP – NP-Complete - Proof

- ✓ Size of the Clauses in BSP: 3, i.e., k=3
- ✓ So, we need to check for the clique with size k=3 is present in the graph or not.
- ✓ Consider any clique with size k=3:
- ✓ Consider the subgraph of G with the vertices <x2, 1>; <x1', 2>; <x3, 3>. It forms a clique of size 3
- ✓ Corresponding to this, for the assignment: <x1, x2, x3> = <0, 1, 1>
- ✓ Substitute this values in BSP for check it is true or not.

F = (x1 v x2) ^ (x1‘ v x2‘) ^ (x1 v x3) = (0V1)^(1V0)^(0V1) = 1^1^1 = 1 →True

CDP – NP-Complete - Proof

- ✓ Therefore, if we have k clauses in our satisfiability expression, we get a max clique of size k and for the corresponding assignment of values, the satisfiability expression evaluates to true.
- ✓ Hence, for a particular instance, the satisfiability problem is reduced to the clique decision problem.
- ✓ Therefore, the Clique Decision Problem is NP-Hard.

Conclusion

- ✓ The Clique Decision Problem is NP and NP-Hard. Therefore, the Clique decision problem is NP-Complete.

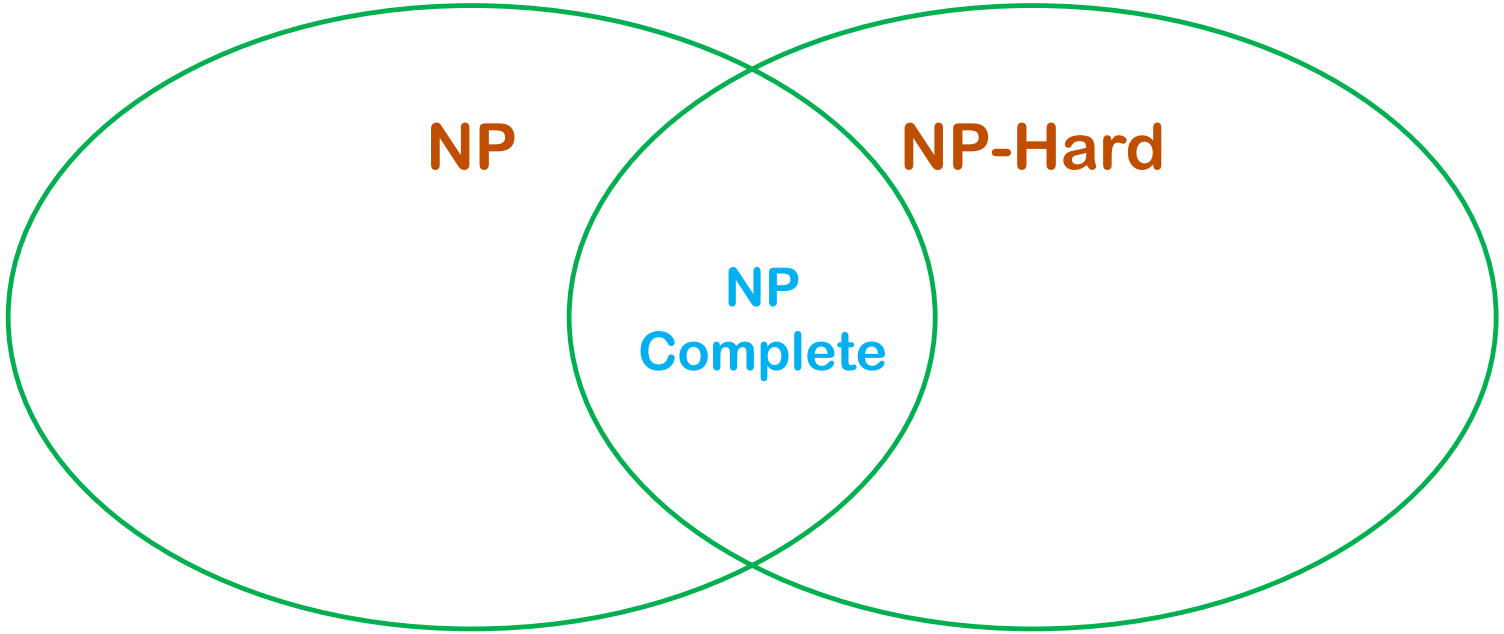
Travelling Salesperson Problem
NP-Complete - Proof

Travelling Salesperson Problem (TSP)

- ✓ Problem of finding shortest trip which cover all the cities.
- ✓ Given a set of cities and the distance between each pair of cities.
- ✓ The travelling salesman problem finds the path between these cities such that it is the shortest path.
- ✓ Traverses every city once, returning back to the starting point.
- ✓ TSP is NP-Complete – Need to prove

CDP – NP-Complete - Proof

- ✓ To prove that a problem is NP-Complete, we have to show that it belongs to both NP and NP-Hard Classes.



TSP – NP-Complete - Proof

1. The TSP to NP

If a problem belongs to the NP class, then it should have polynomial-time verifiability, that is given a certificate, we should be able to verify in polynomial time if it is a solution to the problem.

Proof:

Verification – The verification algorithm checks that this sequence contains each vertex exactly once. Sums up the edge costs. Checks whether the sum is at most k or not.

TSP – NP-Complete - Proof

SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 OF THE UGC ACT, 1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

2. The TSP belongs to NP-Hard –

- ✓ A problem L belongs to NP-Hard if every NP problem is reducible to L in polynomial time.
- ✓ Let the TSP problem by C.
- ✓ To prove that C is NP-Hard, we take an already known NP-Hard problem, say S, and reduce it to C for a particular instance.
- ✓ Need to prove, S $\propto$ C
- ✓ If this reduction can be done in polynomial time, then C is also an NP-Hard problem.

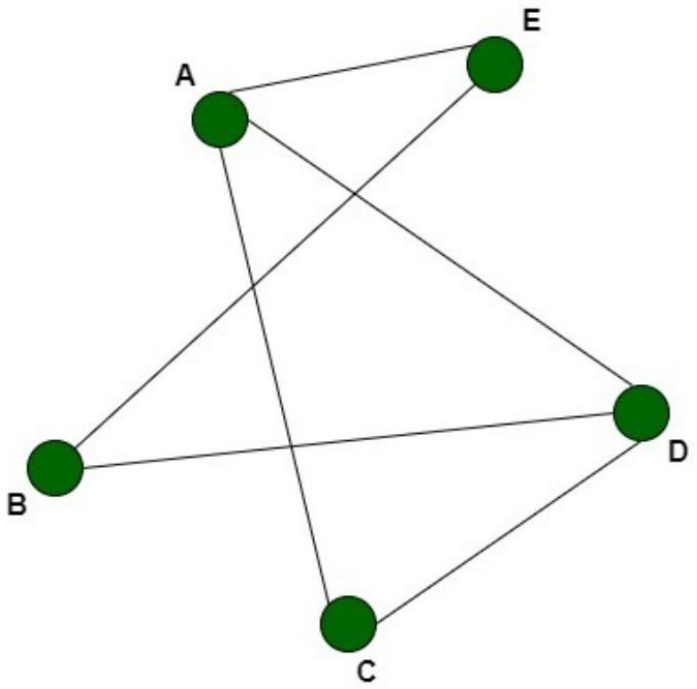
TSP – NP-Complete - Proof

- ✓ In order to prove the Travelling Salesman Problem is NP-Hard, we will have to reduce a known NP-Hard problem to this problem.
- ✓ Consider a NP-Hard problem, called Hamiltonian Cycle Problem (HCP)– to verify a Hamiltonian cycle is present in a graph or not.
- ✓ A Hamiltonian cycle is a closed loop on a graph where every node (vertex) is visited exactly once.
- ✓ If we are able to reduce HCP to TSP, we can say TSP is also NP-Hard.
- ✓ Proof Needed for: HCP reduces to the TSP: $HCP \propto TSP$

TSP – NP-Complete - Proof

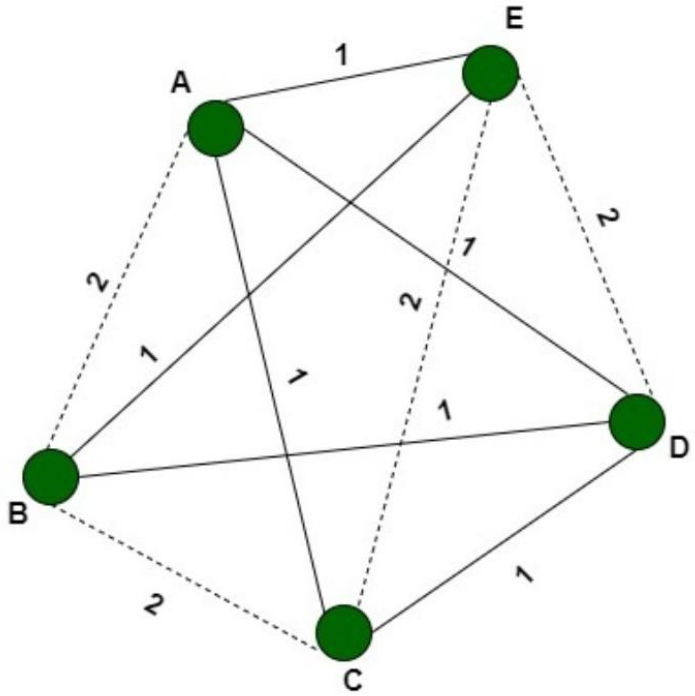
- ✓ Let G=(V,E), an instance of the Hamiltonian Cycle problem.
- ✓ An instance of TSP is constructed as:
- ✓ Form a complete graph G′ = (V, E′)
- ✓ Where, E′ = {(i,j) | i, j ϵ V and i!=j }
- ✓ c(i,j) = 1, if (i,j) ∈ E.
  2, if (i,j) ∉ E
- ✓ Set K = N.
- ✓ The new graph G′ can be constructed in polynomial time by just converting G to a complete graph G′ and adding corresponding costs

CDP – NP-Complete - Proof



G =

Hamiltonian cycle {EACDBE}



G' =

TSP {EACDBE}

Cost = 5 (=n)

Proof of Reduction

Graph G:

- ✓ G contains a Hamiltonian Cycle, traversing all the vertices V of the graph.
- ✓ Now, these vertices form a TSP with cost = N Since it uses all the edges of the original graph having cost c(e)=1. And, since it is a cycle, therefore, it returns back to the original vertex.

Graph G':

- ✓ G' contains a TSP with cost, K = N. The TSP traverses all the vertices of the graph returning to the original vertex.
- ✓ Now since none of the vertices are excluded from the graph and the cost sums to n, therefore, necessarily it uses all the edges of the graph present in E, with cost 1, hence forming a hamiltonian cycle with the graph G

Conclusion

- ✓ Thus we can say that the graph G' contains a TSP if graph G contains Hamiltonian Cycle. Therefore, any instance of the Travelling salesman problem can be reduced to an instance of the hamiltonian cycle problem. Thus, the TSP is NP-Hard.
- ✓ Thus we proved that, the TSP is NP and NP-Hard.
- ✓ Therefore, the TSP is NP-Complete.

Approximation Algorithms
Bin Packing Problem

Approximation Algorithm

- ✓ Used for solving many optimization problems for which there is no polynomial time algorithm for its solution.

- ✓ Approximation algorithm allow for getting a solution close to the solution of an optimization problem in polynomial time.

- ✓ An algorithm is an α – approximation algorithm for optimization problem If:
  - The algorithm runs in polynomial time
  - The algorithm always produces a solution that is within a factor of α of the optimal solution
  - For given problem I, $Approximation\ Ratio = \frac{Algo(I)}{Opt(I)}$

Bin Packing Problem

Input:

- ✓ Number of items: n Items (of different weights)
- ✓ Weight List (weight for each item): Weight[1..n]
- ✓ Various numbers of Bins each of capacity: c

Problem:

- ✓ Assign each item to a bin such that number of total used bins is minimized.

Objective:

- ✓ Minimize the number of BINS

Assumption:

- ✓ All items have weights smaller than bin capacity.

Bin Packing Problem - Example

Input: Weight[] = {4, 8, 1, 4, 2, 1}

Bin Capacity c = 10

Output: 2

Need minimum 2 bins to accommodate all items

Bin Content: {4, 4, 2} and {8, 1, 1}

Bin Packing Problem - Example

Lower Bound (Minimum numbers of bins required)

We can always find a lower bound on minimum number of bins required.

The lower bound can be given as:

$$Min\ no.of\ bins \geq \lceil Total\ Weight\ /\ Bin\ Capacity \rceil$$

lower bound for first example is " ⌈(4 + 8 + 1 + 4 + 2 + 1)/10⌉ " = 2

Bin Packing Problem

Approximation Algorithms

- ✓ First Fit (FF)
- ✓ Best Fit (BF)
- ✓ First Fit Decreasing (FFD)
- ✓ Best Fit Decreasing (BFD)

Bin Packing Problem - Example

Problem:

n = 6 *No. of Objects*

c = 10 *Bin Capacity*

w[1..6] = {5, 6, 3, 7, 5, 4 } *Weights of objects*

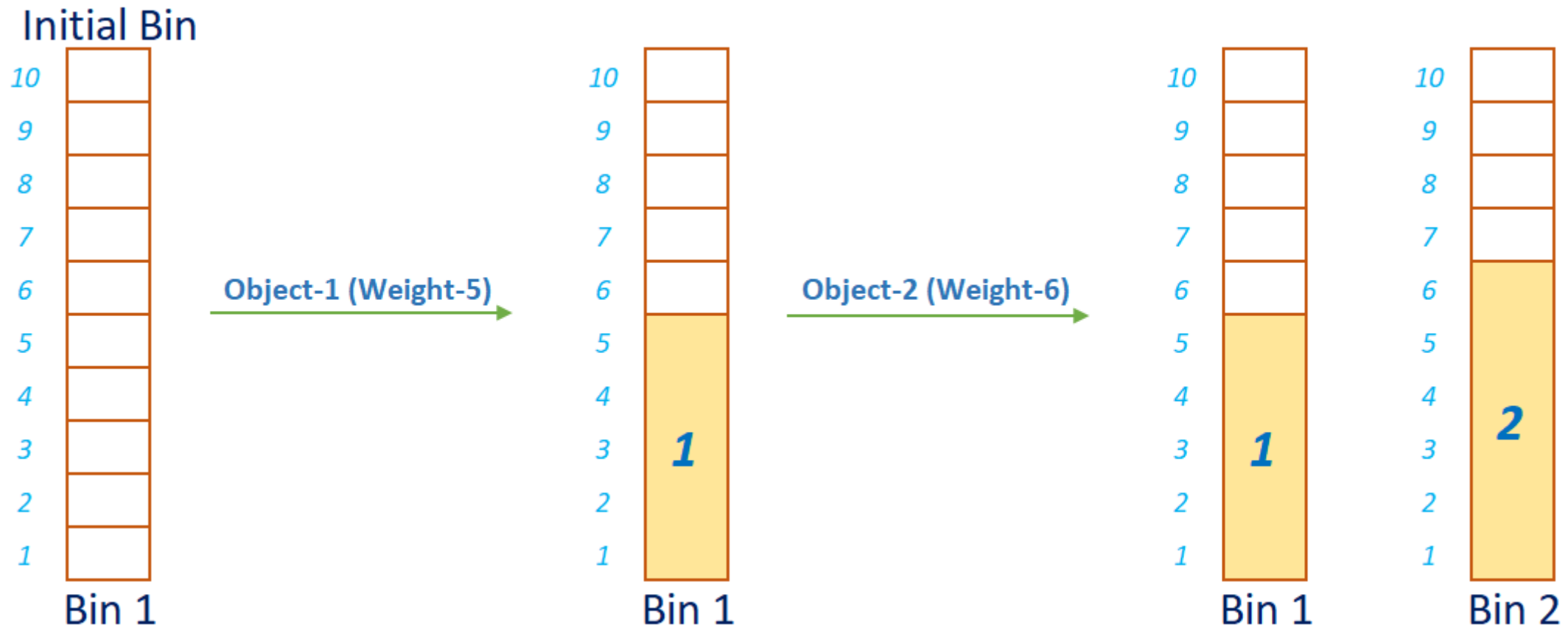
Lower Bound = (5+6+3+7+5+4)/10 = 3

Minimum Numbers of Bins = 3

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*

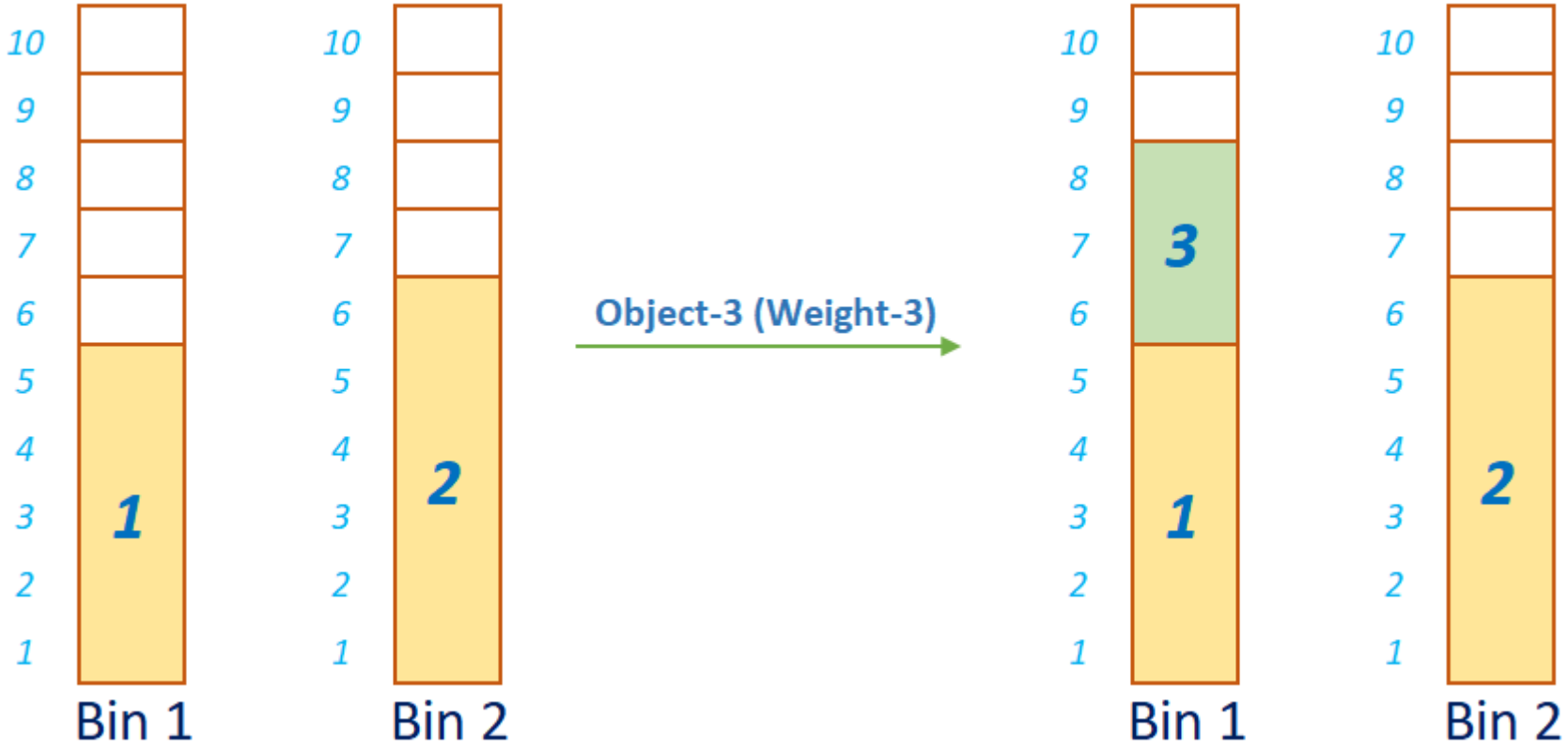
BPP – First Fit Approximation Algorithm

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*



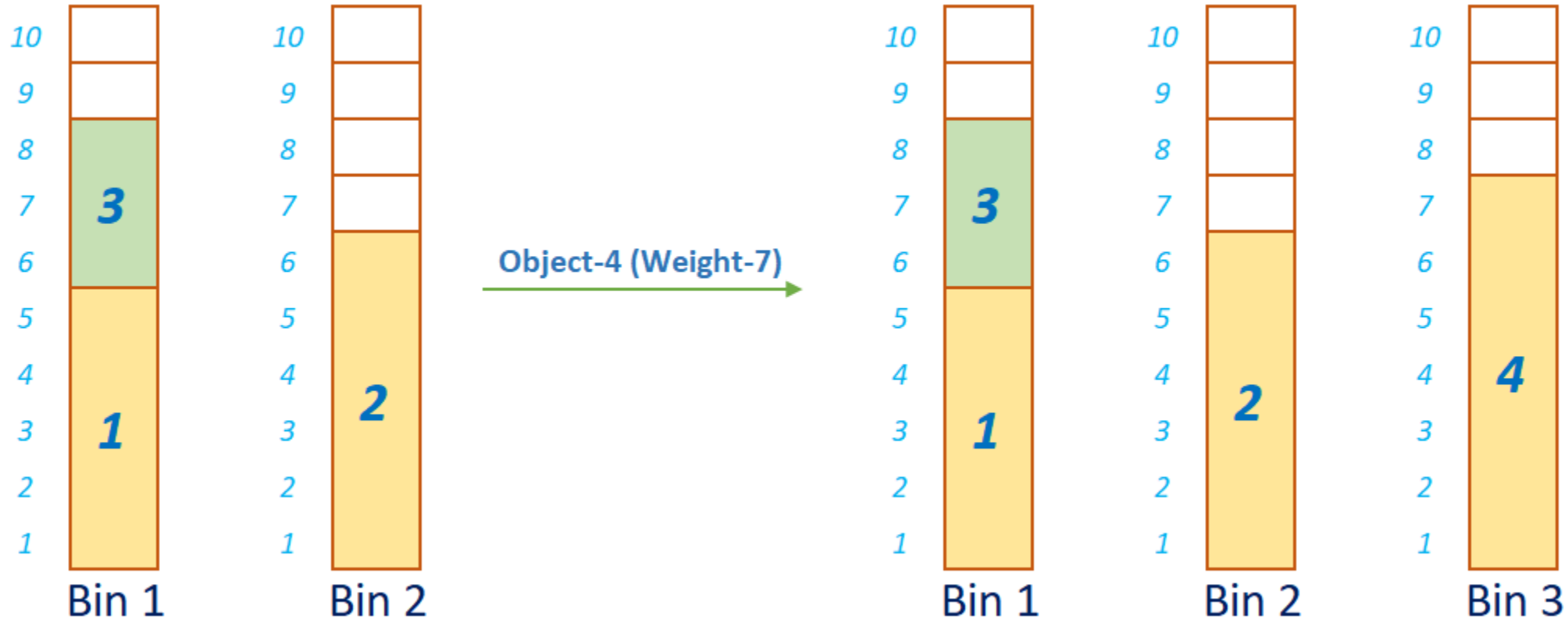
BPP – First Fit Approximation Algorithm

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*



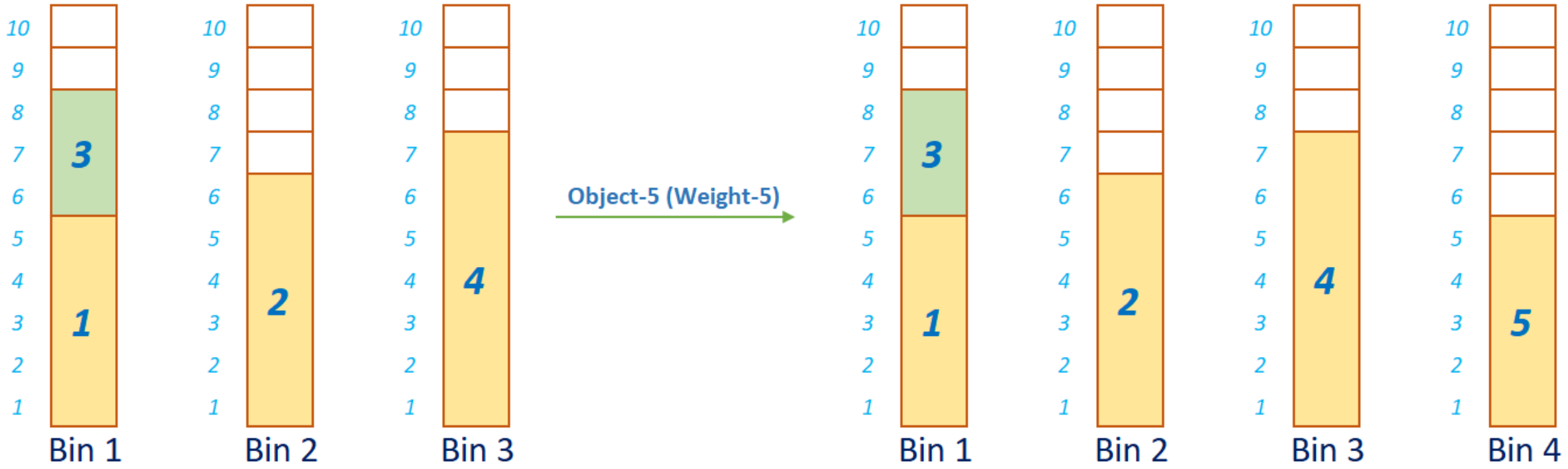
BPP – First Fit Approximation Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight



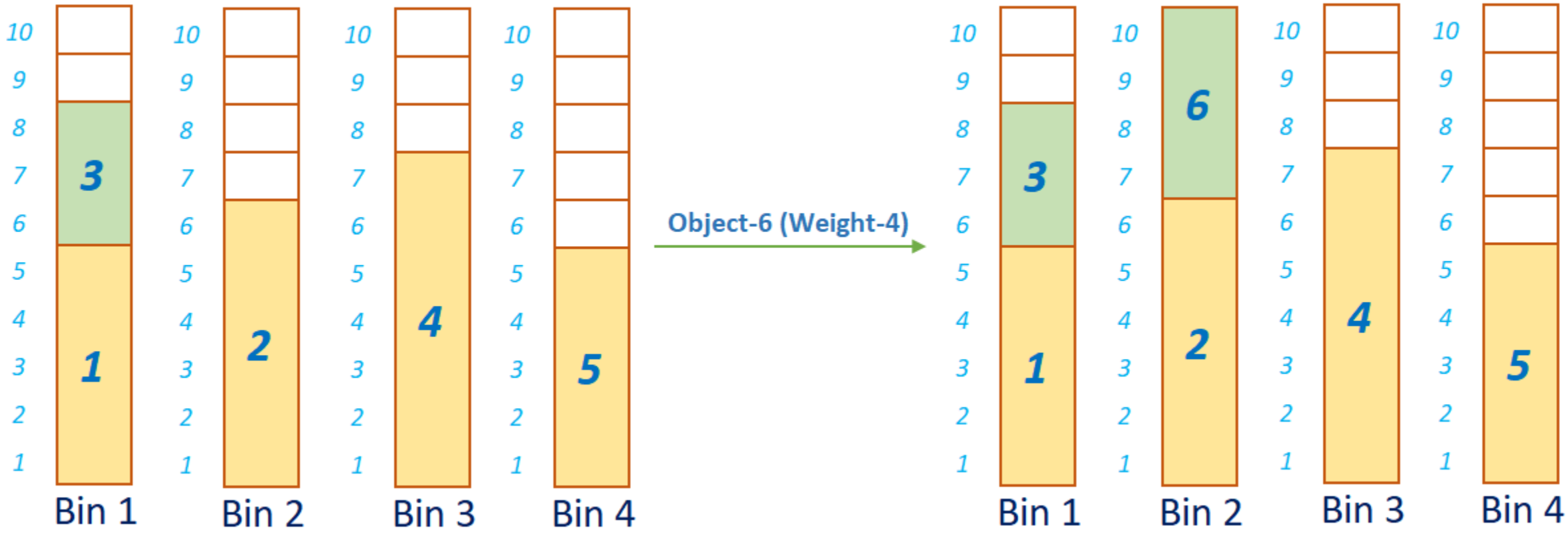
BPP – First Fit Approximation Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight



BPP – First Fit Approximation Algorithm

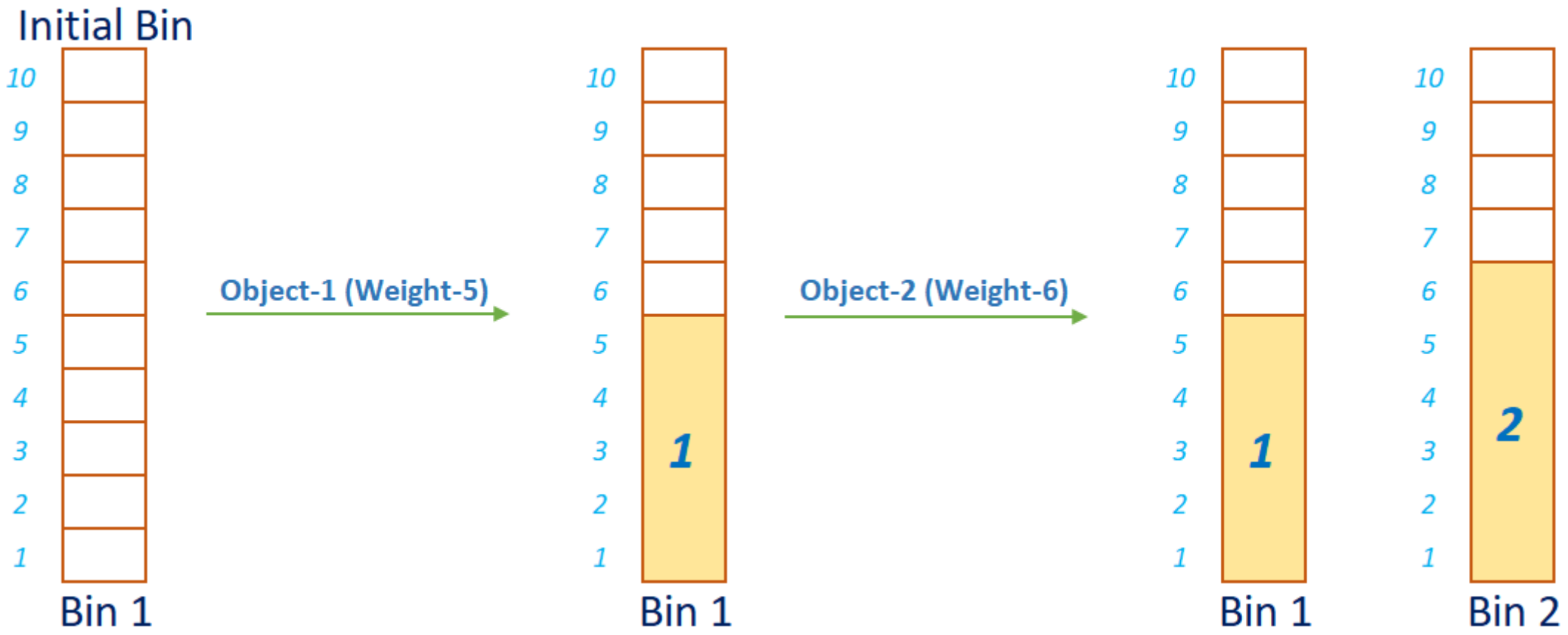
1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight



Solution: Minimum Numbers of Bin Required = 4 with objects {1,3}, {6,2}, {4} and {5}

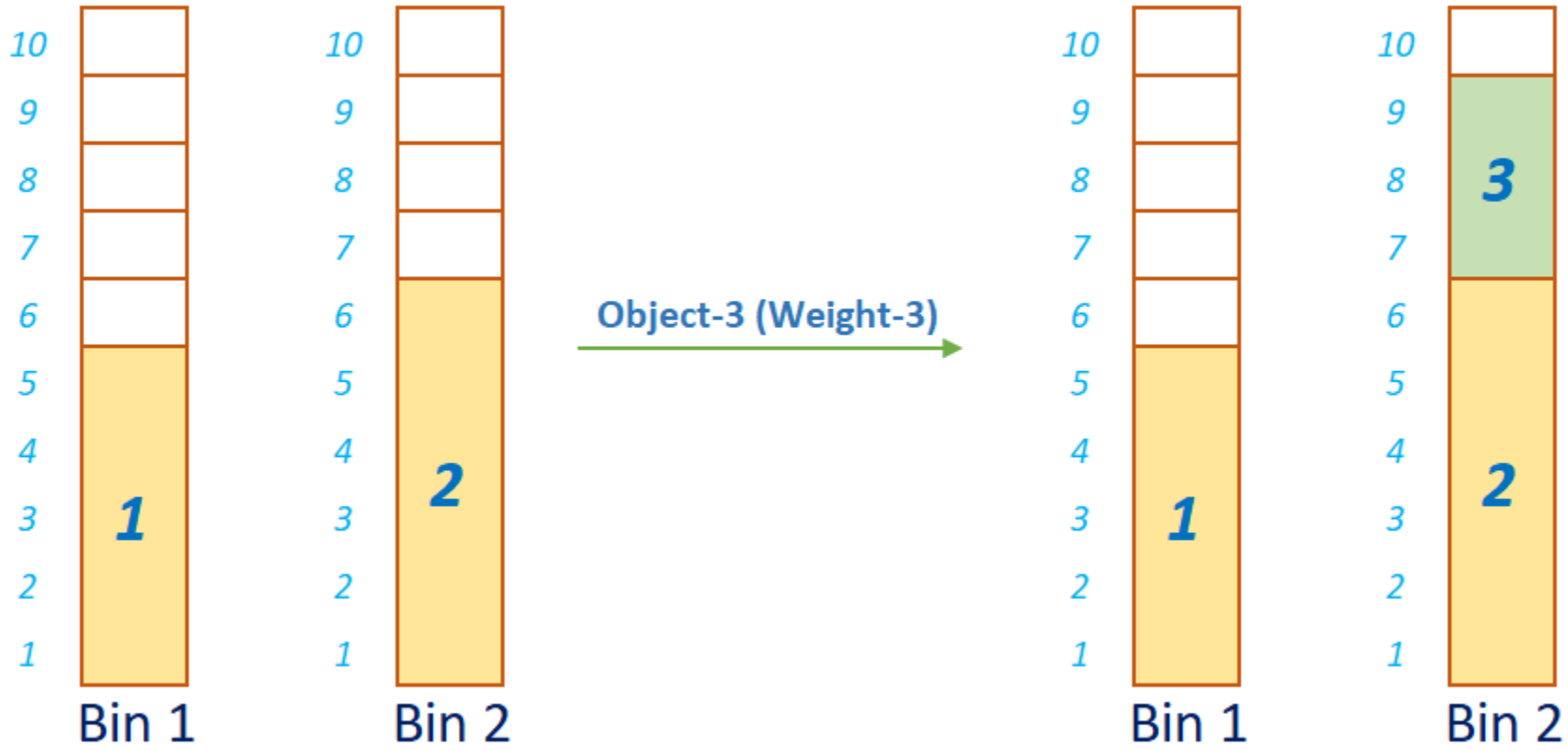
BPP – Best Fit Approximation Algorithm

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*



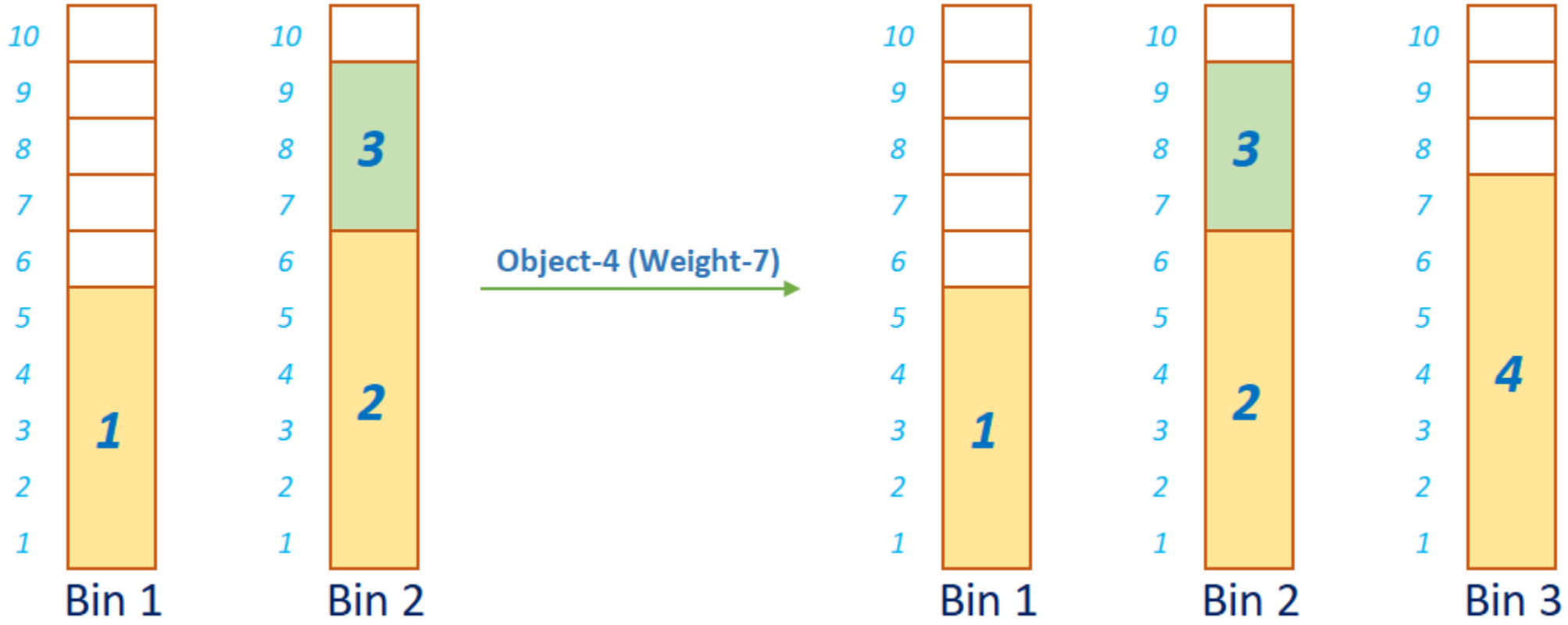
BPP – Best Fit Approximation Algorithm

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*



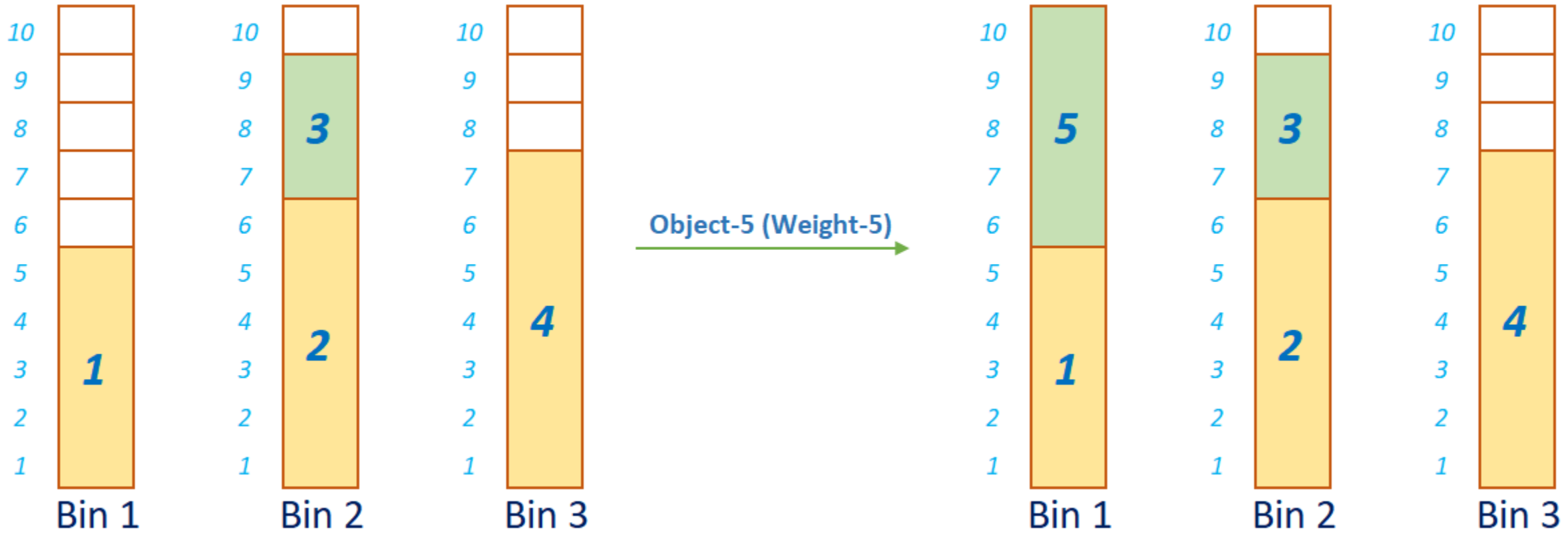
BPP – Best Fit Approximation Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight



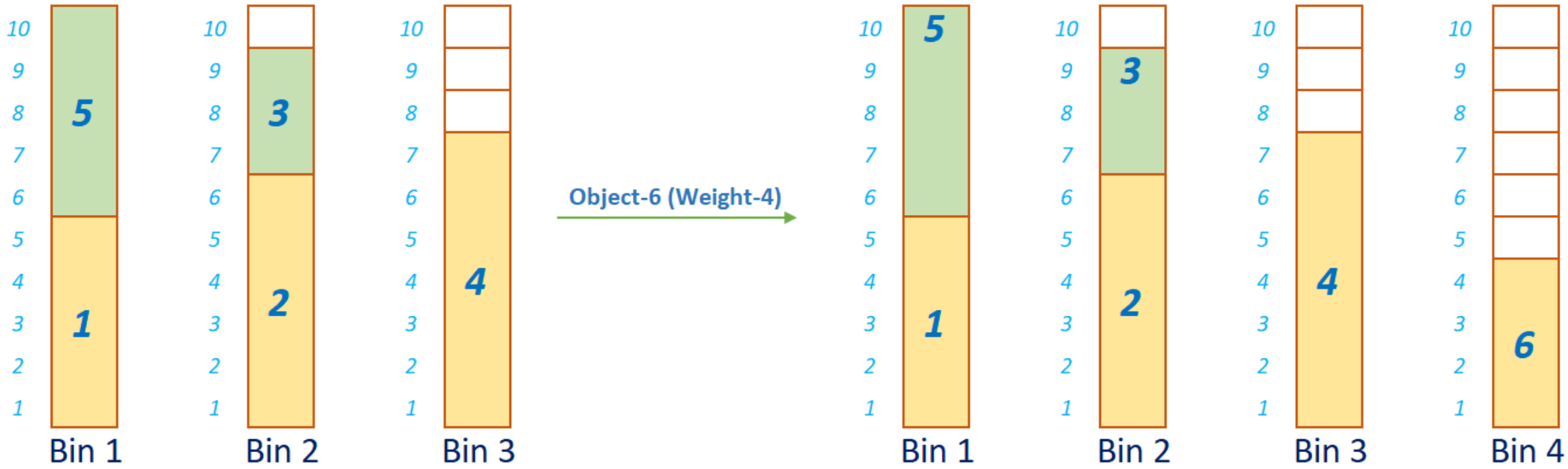
BPP – Best Fit Approximation Algorithm

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*



BPP – Best Fit Approximation Algorithm

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*



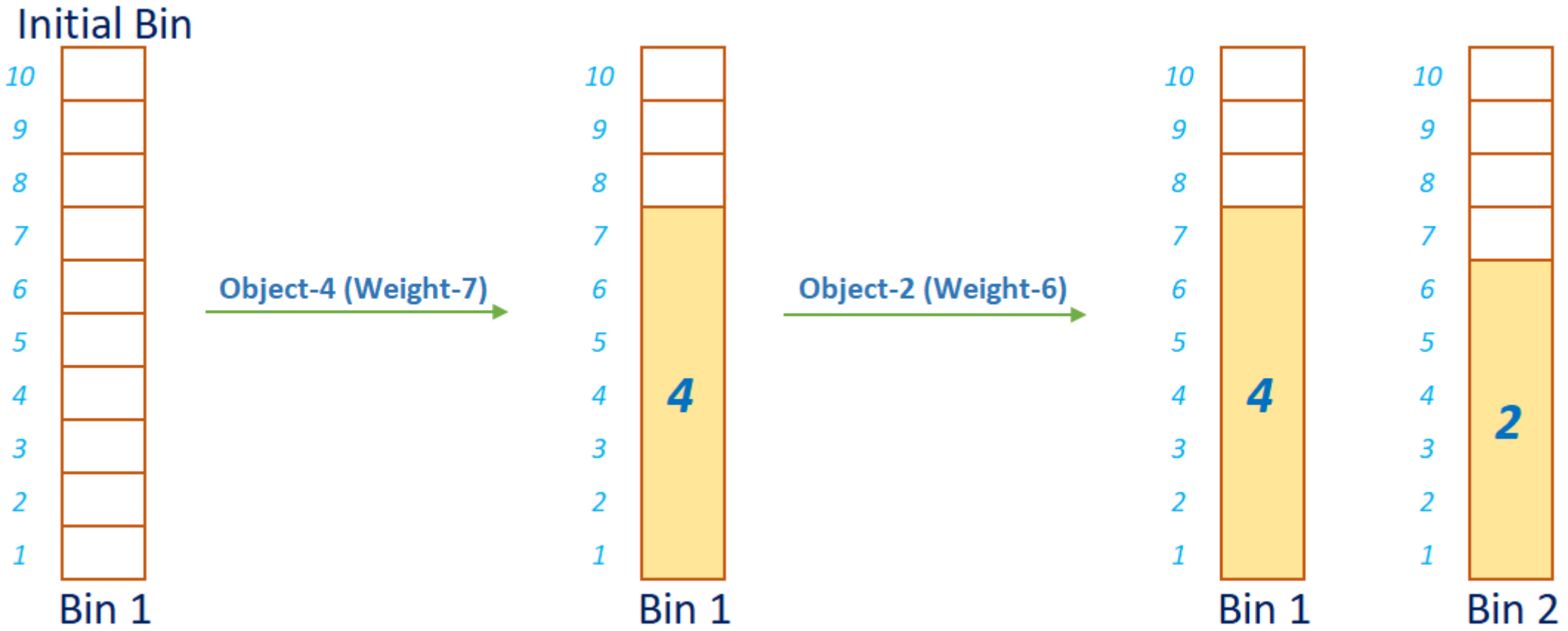
Solution: Minimum Numbers of Bin Required = 4 with objects {1,5}, {2,3}, {4} and {6}

BPP – Next Fit Decreasing (NFD) Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight

Decreasing Order →

4	2	1	5	6	3	Object Number
7	6	5	5	4	3	Object's Weight

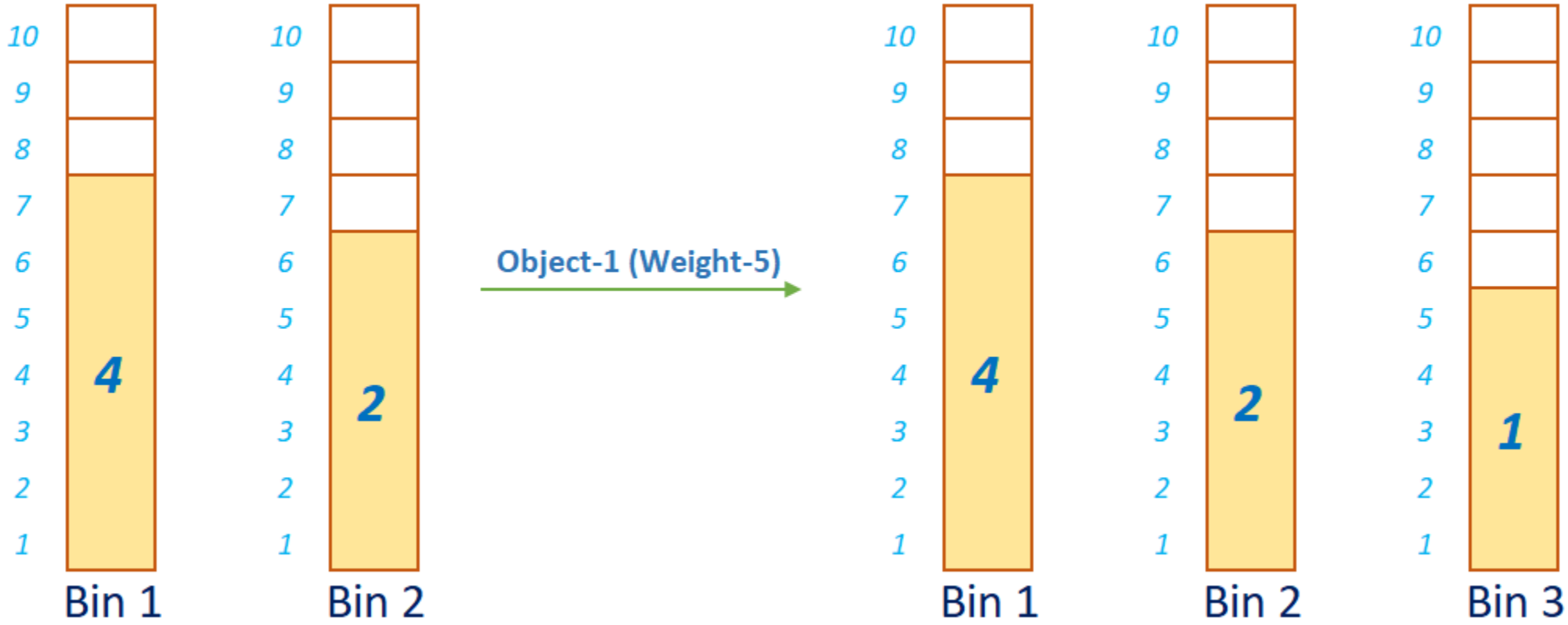


BPP – Next Fit Decreasing (NFD) Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight

Decreasing Order →

4	2	1	5	6	3	Object Number
7	6	5	5	4	3	Object's Weight

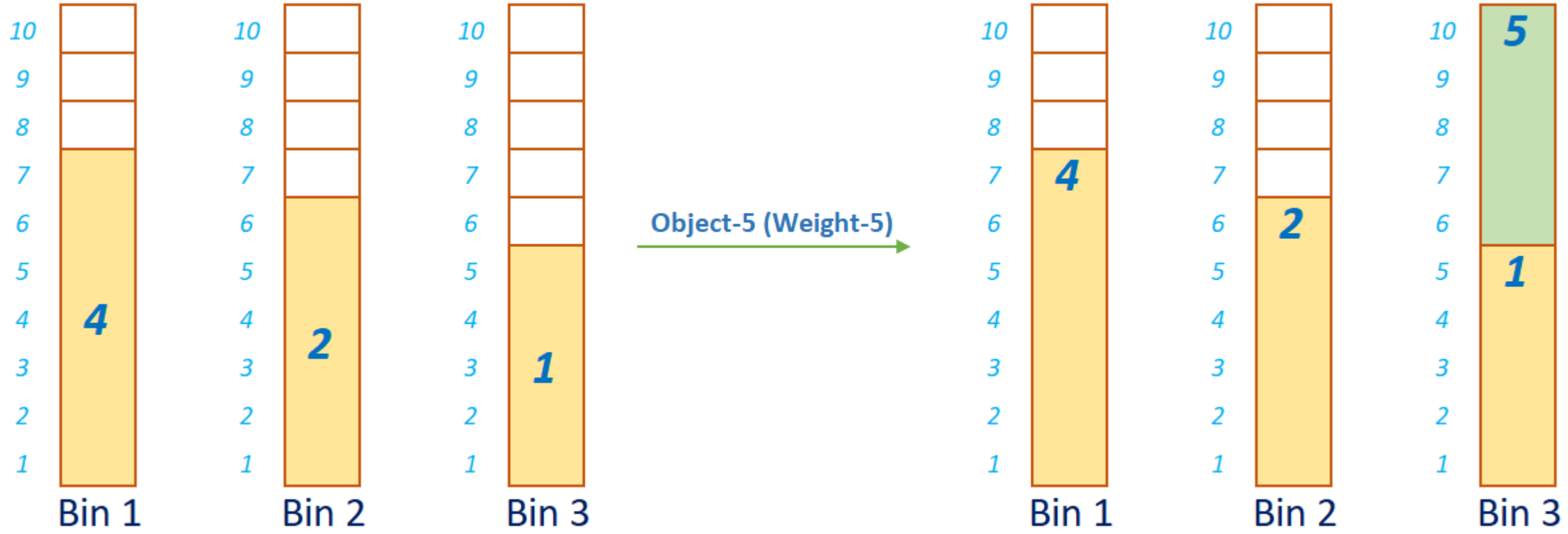


BPP – Next Fit Decreasing (NFD) Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight

Decreasing Order →

4	2	1	5	6	3	Object Number
7	6	5	5	4	3	Object's Weight

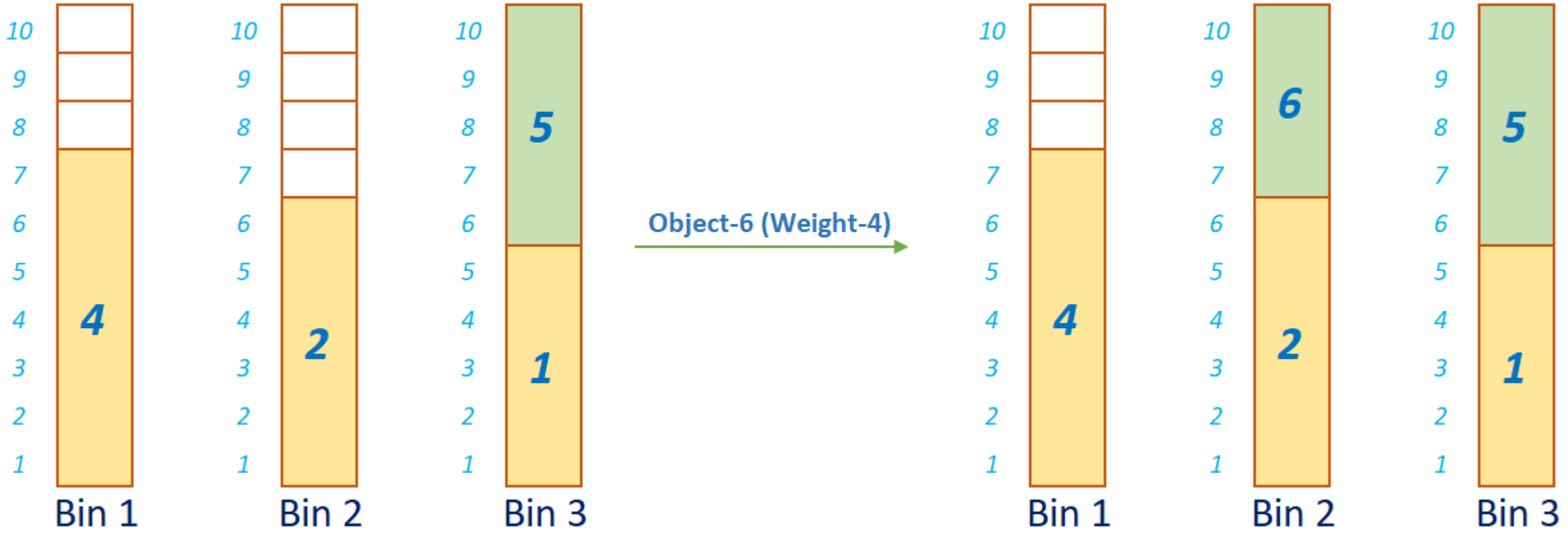


BPP – Next Fit Decreasing (NFD) Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight

Decreasing Order →

4	2	1	5	6	3	Object Number
7	6	5	5	4	3	Object's Weight

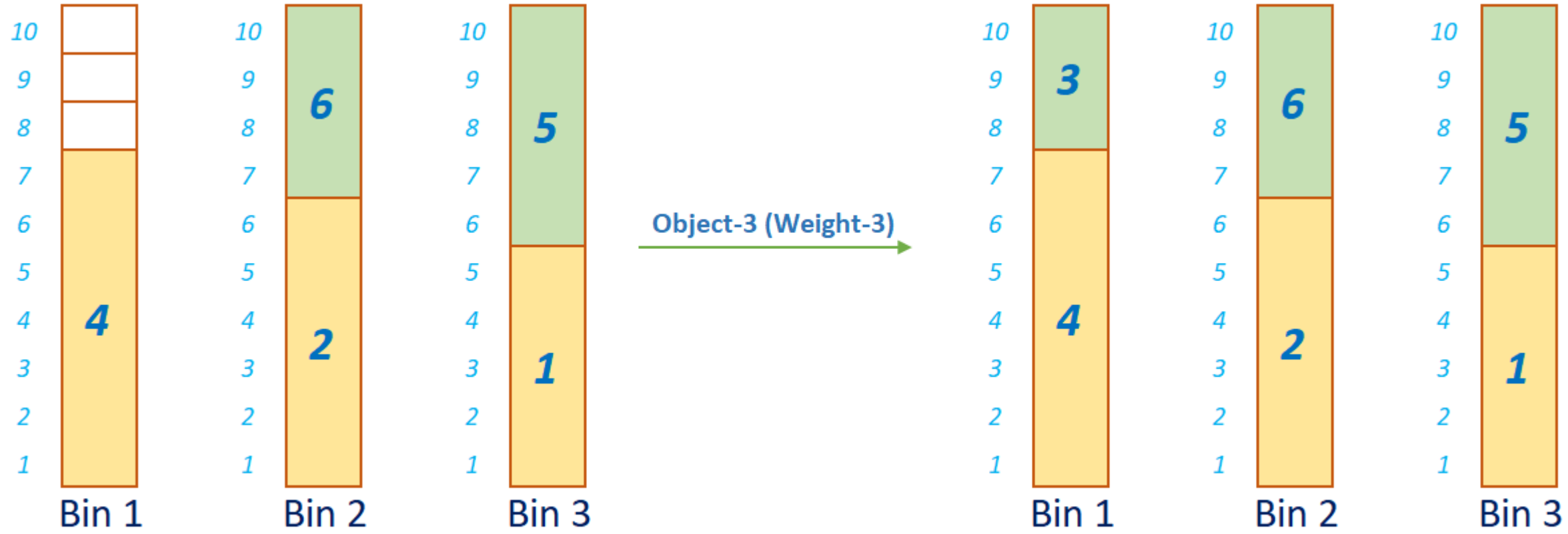


BPP – Next Fit Decreasing (NFD) Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight

Decreasing Order →

4	2	1	5	6	3	Object Number
7	6	5	5	4	3	Object's Weight



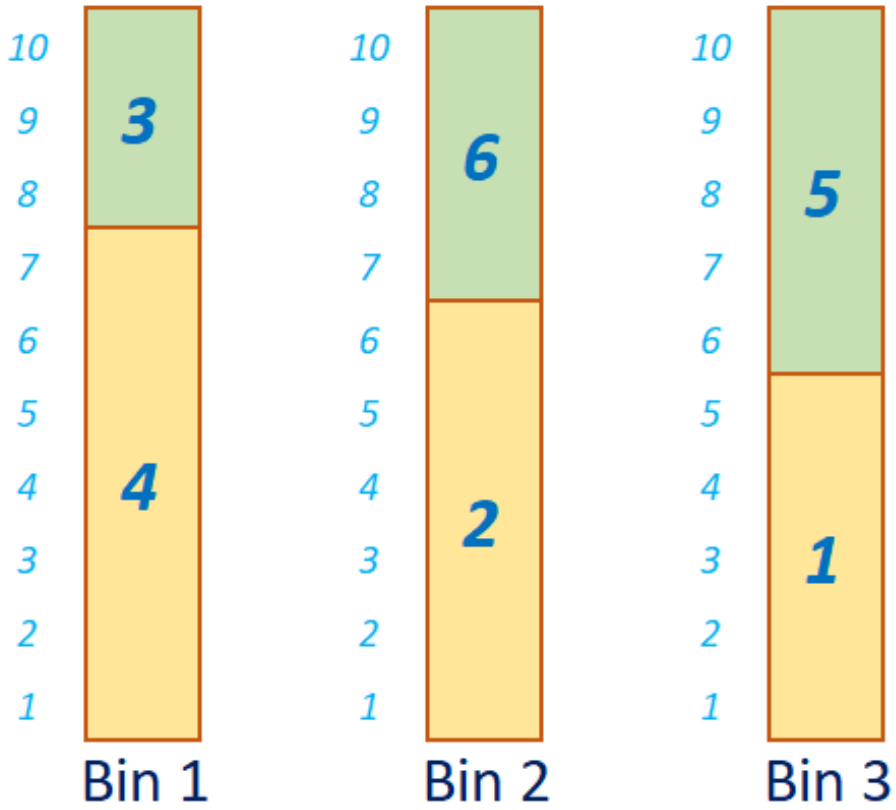
Solution: Minimum Numbers of Bin Required = 3 with objects {4,3}, {2,6} and {1,5}

BPP – Best Fit Decreasing (NFD) Algorithm

1	2	3	4	5	6	Object Number
5	6	3	7	5	4	Object's Weight

Decreasing Order →

4	2	1	5	6	3	Object Number
7	6	5	5	4	3	Object's Weight



Solution: Minimum Numbers of Bin Required = 3

Objects: {4,3}, {2,6} and {1,5}

Conclusion on BPP algorithms

- ✓ All the four algorithms are giving approximate solution
- ✓ Among them, FFD and BFD are giving solution close to the optimal solution.

Problems

Problem-1: (BPP)

n = 7

w[1..7] = { 11, 2, 15, 5, 6, 17, 7 }

c = 20

Find optimal numbers of bins required.

Apply all the four algorithms to obtain solution.

Problems

Problem-2: (BPP)

n = 9

w[1..9] = { 0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}

c = 1.0

Find optimal numbers of bins required.

Apply all the four algorithms to obtain solution.

Approximation Algorithms
Scheduling Independent Tasks Problem

Scheduling Independent Tasks Problem

Problem Statement

Obtaining minimum finish time schedules on m, $m \geq 2$, identical processors is $\mathcal{NP}$-hard. There exists a very simple scheduling rule that generates schedules with a finish time very close to that of an optimal schedule. An instance I of the scheduling problem is defined by a set of n task times t_i, $1 \leq i \leq n$, and m, the number of processors. The scheduling rule we are about to describe is known as the LPT (longest processing time) rule. An LPT schedule is a schedule that results from this rule.

Scheduling Independent Tasks Problem

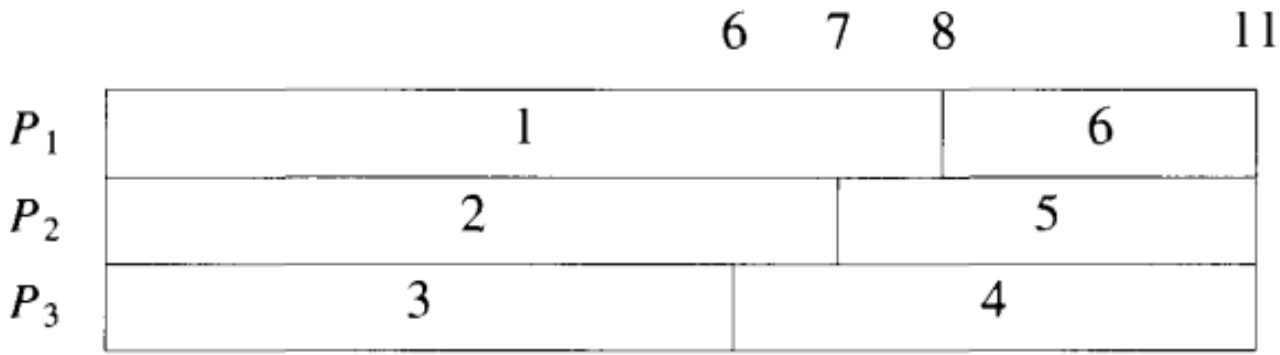
LPT Schedule

Definition 12.7 An *LPT schedule* is one that is the result of an algorithm that, whenever a processor becomes free, assigns to that processor a task whose time is the largest of those tasks not yet assigned. Ties are broken in an arbitrary manner. □

Scheduling Independent Tasks Problem

Example

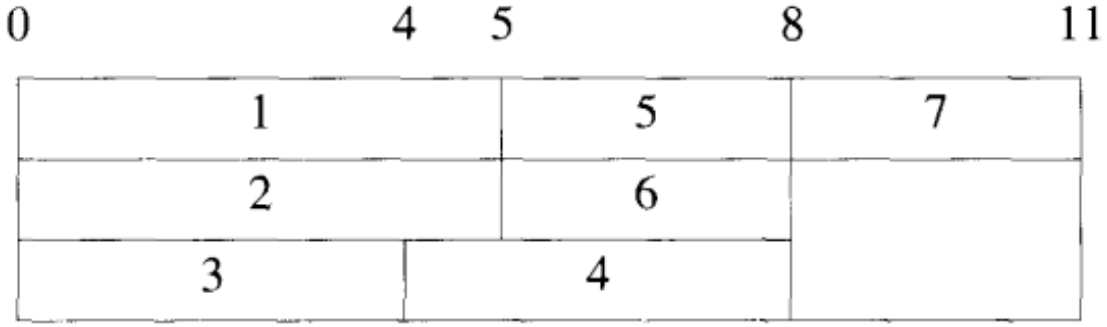
Example 12.6 Let $m = 3, n = 6$, and $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 7, 6, 5, 4, 3)$. In an LPT schedule, tasks 1, 2, and 3 are assigned to processors 1, 2, and 3 respectively. Tasks 4, 5, and 6 are respectively assigned to processors 3, 2, and 1. Figure 12.2 shows this LPT schedule. The finish time is 11. Since $\sum t_i/3 = 11$, the schedule is also optimal. □



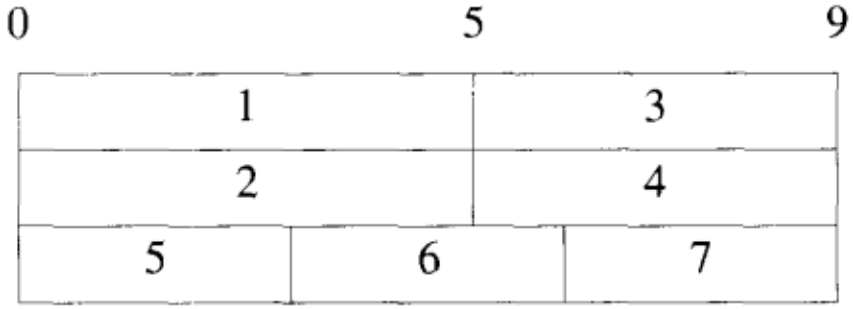
Scheduling Independent Tasks Problem

Example

Example 12.7 Let $m = 3$, $n = 7$, and $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (5, 5, 4, 4, 3, 3, 3)$. Figure 12.3(a) shows the LPT schedule. This has a finish time of 11. Figure 12.3(b) shows an optimal schedule. Its finish time is 9. Hence, for this instance $|F^*(I) - \hat{F}(I)|/F^*(I) = (11 - 9)/9 = 2/9$. □



(a) LPT schedule



(b) Optimal schedule

Scheduling Independent Tasks Problem

LPT vs Optimal Schedule

LPT rule may generate optimal schedules for some problem instances, it does not do so for all instances. How bad can LPT schedules be relative to optimal schedules? This question is answered by the following theorem.

Theorem 12.5 [Graham] Let $F^*(I)$ be the finish time of an optimal m-processor schedule for instance I of the task scheduling problem. Let $\hat{F}(I)$ be the finish time of an LPT schedule for the same instance. Then,

$$\frac{|F^*(I) - \hat{F}(I)|}{|F^*(I)|} \leq \frac{1}{3} - \frac{1}{3m}$$

Scheduling Independent Tasks Problem

Example

1. Consider seven independent tasks with processing times (in hours) given by 1, 4, 5, 7, 8, 9, and 10.

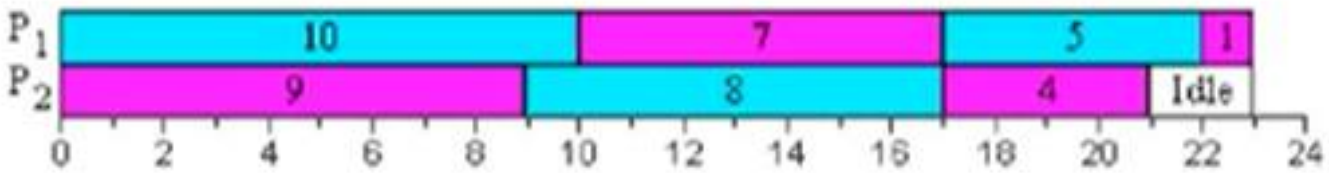
(a) Schedule these tasks with N = 2 processors using the critical-path algorithm. Show the timeline, and give the project finishing time Fin.

(b) Find the optimal finishing time Opt for N = 2 processors.

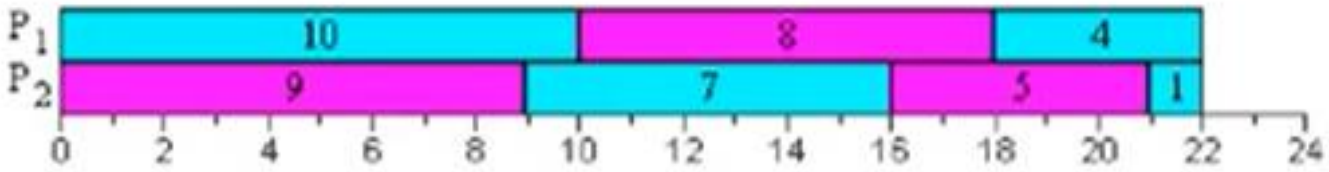
(c) Compute the relative error of the critical-path schedule found in (a) expressed as a percent.

(a) Choose the correct answer below.

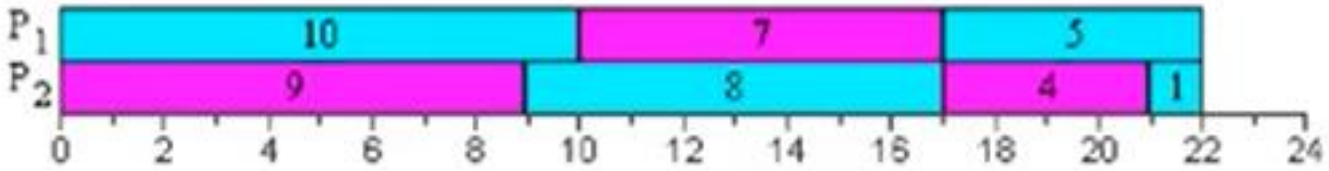
○ **A.**



○ **B.**



○ **C.**



What is the project's finishing time?

Fin = 22 hours

(b) Opt = 22 hours

(c) ε = 0 %

(Type an integer or a decimal.)

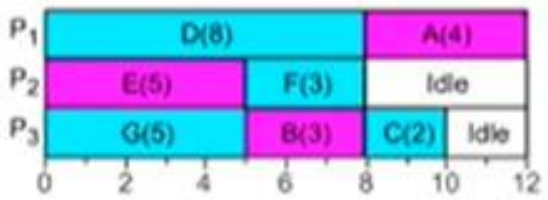
Scheduling Independent Tasks Problem

Example

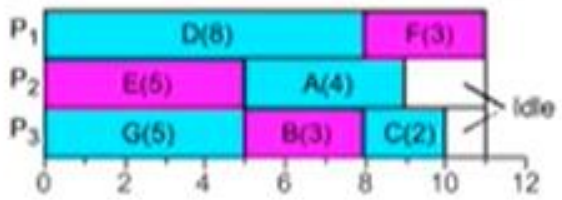
2. Consider the following set of independent tasks: A(4), B(3), C(2), D(8), E(5), F(3), G(5). Complete (a) through (c) below.

(a) Schedule these tasks with N = 3 processors using the critical-path algorithm. Show the timeline, and give the project finishing time Fin. Choose the correct answer below.

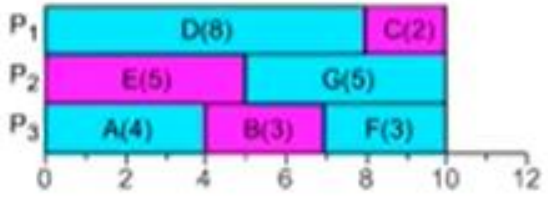
○ **A.**



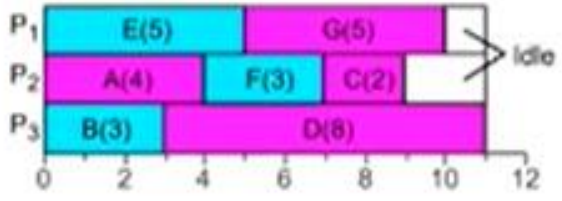
○ **B.**



○ **C.**



○ **D.**



What is the optimal finishing time?

Opt = 10

(c) Compute the relative error of the critical-path schedule found in (a) expressed as a percent.

ε = 10.00 %

(Round to two decimal places as needed.)

$$\frac{11-10}{10} \times 100\% = 10\%$$

Summary

- ✓ Polynomial vs Exponential
- ✓ Deterministic vs Non-Deterministic
- ✓ P vs NP
- ✓ NP-Hard vs NP-Complete
- ✓ P, NP, Np-Hard & NP-Complete – Relations
- ✓ Cook's Theorem
- ✓ Clique Decision Problem
- ✓ Travelling Salesperson Problem
- ✓ Bin-Packing Problem

Thank You