UNIT 3

GREEDY AND DYNAMIC PROGRAMMING ALGORITHMIC STRATEGY.

★ GREEDY STRATEGY.

- decision of solution is taken based on the information available.
- straight-forward method.
- builds solution in steps
- at every stage, it selects best choice concerning the local considerations.
- solution is constructed through a sequence of steps, each step contributing towards complete solution of the problem.
- The choice of solution made at each step is feasible, locally optimal & irrevocable.
- a locally optimal partial solution cannot be altered later.
- a global solution is constructed with locally optimal partial solutions gathered at each stage.
- Classic problems solved with Greedy method:

1) Knapsack problem
2) Job sequencing problem
3) Activity selection problem
4) Minimum spanning tree problem
5) Tree vertex splitting problem.

components of Greedy strategy.

1) Feasible solution.
A subset of given inputs that satisfy all specified constraints of a problem.

2) Optimal Solution.
feasible solution that either maximizes or minimizes the objective function specified in a problem

3) Feasibility check.
investigates whether input fulfills all constraints mentioned in the problem.
If yes, then it is added to a set of feasible solutions.

4) ~~Optam~~ Optimality check
~~investigates if whether a selected input fulfills all constraints~~
investigates whether a selected input produces either maximum or minimum value of objective function by fulfilling given constraints

5) Optimal substructure property.
global optimal solution to problem includes the optimal sub solutions

6) Greedy choice property
The globally optimal solution is assembled by selecting locally optimal choices.

Control Abstraction for Greedy Method.

```
procedure GREEDY (A, n)
// A (1:n) contains n inputs //
solution ← ϕ   // initialize solution to empty //
for i ← 1 to n do
  x ← SELECT (A)
  if FEASIBLE (solution; x)
    then solution ← UNION (solution, x)
  endif
repeat
return (solution)
end GREEDY.
```

- SELECT selects an input from a[] & removes it. The selected input's value is assigned to x.
- Feasible is a Boolean value which decides whether x can be included into the solution vector or not.
- Union combines x with the solution & updates objective function.

Types of Greedy Methods.

1) Subset Paradigm
greedy approach that determines a subset of inputs that gives optimal solution
eg- Knapsack problem
2) Ordering Paradigm.
greedy approach that determines some ordering of inputs that gives an optimal solution.
eg. Optimal Storage on Tape.

KNAPSACK PROBLEM.

- follows subset paradigm of greedy approach.
- also known as continuous knapsack problem.
- Consider a knapsack of capacity M.
Suppose n items are given to fill the knapsack.
Each item i ($1 \leq i \leq n$) has a weight w_i & it gives profit $p_i \cdot x_i$, if its fraction x_i ($0 \leq x_i \leq 1$) is kept into the knapsack.
- Fractional knapsack problem is the problem to earn maximum profit by filling knapsack with items considering the constraint of the knapsack capacity.

- Mathematical representation:

Maximize $\sum_{i=1}^{n} p_i x_i$

Subject to $\sum_{i=1}^{n} w_i x_i \leq M$ & $0 \leq x_i \leq 1$, $1 \leq i \leq n$.

- All values of profit & weights are positive numbers.
- To get max profit, greedy approach tries to select objects with maximum profit.
- But, the objects should have lighter weights so that more objects can fit in the knapsack.
- It is achieved by checking the ratio of profit to weight of each object.
- Greedy strategy arranges all items in descending order of ratios.
- Items are kept in the sack according to the order.
- If sufficient space is not available, then fractional part is added to the knapsack to make it full.

* Steps:

1) Sort n items in descending order of their profit to weight ratios
2) Select items as per sorted list & check for the available knapsack capacity.
   If sufficient space available, place the whole item in the knapsack.
3) If the space is insufficient, keep a fractional part of that item equal to the remaining capacity of knapsack.
   Only one item is in fractional part, others all are whole.
4) When knapsack is full, the algorithm terminates.

~~Algorithm~~.

~~Algor~~

O(n) → Time complexity.

Algorithm.

Algorithm Fr-Knapsack (M, n)
Input : Knapsack capacity = M.
n is no. of available items with profits P[1:n] & weights W[1:n]
These items are arranged such that P[i]/W[i] ≥ P[i+1]/W[i+1]
Output
S[1:n] is the fixed size of solution vector. S[i] gives the fractional part xi of item i placed in the knapsack $0 \leq x_i < 1$
$1 \leq i \leq n$.

```
{
  for (i:=1; i≤n; i++)
  {
    S[i] = 0.0                    Initialize solution vector.
  }
  balance:=M;                     Balance - remaining capacity of knapsack
                                  initially it is equal to capacity M.

  for (i:=1; i≤n; i++)
  {
     if (W[i] > balance)          Insufficient capacity of a knapsack
     break;
     S[i] = 1.0;                  add whole item i.
     balance = balance - W[i];    Update the remaining capacity.
  }
  if (i≤n)                        add fractional part of item i in knapsack
    S[i] = (balance / W[i])
    return S;
}
```

* Sorting algo → O(n log n) to sort n items in descending order But here the items are already sorted in decreasing order of Pi/wi then ignoring O(n log n) we get O(n).

Q. How does fractional greedy algorithm solve the following knapsack problem with capacity 20, P = (25, 24, 15) & W = (18, 15, 10).

→ M = 20
P = (25, 24, 15)
W = (18, 15, 10)

P	W	Pi/Wi	
25	18	1.389	i3
24	15	1.6	i1
15	10	1.5	i2

First add i1.
∴ Capacity = 20 − 15 = 5.
Due to insufficient capacity, i2 cannot be added as a whole.
∴ fractional part of i2

$$\frac{5}{10} = 0.5 = \frac{1}{2} \times 15 = \underline{\underline{7.5}} \text{ units.}$$

∴ Total profit earned = 24 + 7.5 (P1) (P2)
= 31.5.

Solution vector S = {0, 1, 1/2).

★ JOB SCHEDULING.

- follows subset paradigm
- involves scheduling of jobs such that they get completed before their deadlines giving the profit.
- objective is to maximize total profit gained by completion of jobs.
- Consider there are n jobs that are to be executed.
- At any time only one job is to be executed.
- Each job takes a unit of time

- The profits pi are given.
- Profits are gained by corresponding jobs.
- To gain profits, job needs to be finished within the deadline
- Goal is to schedule jobs to maximize profit.
- Feasible solutions are obtained by various permutation & combi. of jobs.
- Feasible sequence is a sequence that allows all jobs in a sequence to be executed within their deadlines & highest profit can be gained.
- Optimal solution is feasible solution with max. profit.

Q. Solve the following instance of "job scheduling with deadlines" problem: $n=7$, profits = (3, 5, 20, 18, 1, 6, 30) & deadlines (1, 3, 4, 3, 2, 1, 2).

→ D.O. arrangement

Profit	30	20	18	6	5	3	1
Job	P7	P3	P4	P6	P2	P1	P5
Deadline	2	4	3	1	3	1	2

Create an empty array J[].

1	2	3	4	5	6	7
0	0	0	0	0	0	0

Insert first job in the array based on the deadline (index represent deadline)

I1 :

1	2	3	4	5	6	7
	P7					

I2 :

1	2	3	4	5	6	7
	P7		P3			

I3 :

1	2	3	4	5	6	7
	P7	P4	P3			

I4 :

1	2	3	4	5	6	7
P6	P7	P4	P3			

Next job is P_2 ~~but~~ with deadline 3, but index 3 is already occupied ∴ P_2 is discarded.
Same is done with P_1 & P_5.
∴ Optimal sequence → 6-7-4-3.
Max. profit → 30 + 20 + 18 + 6.
= 74.

* Activity Selection Problem.

- follows subset paradigm of greedy algorithm

Dynamic Programming

- Method for solving problems by breaking them down into smaller subproblems & storing the solutions to these subproblem ~~in order to avoid~~
- This is done to avoid recalculate them every time they are needed.
- It is often used for optimization problems, where the goal is to find best possible solution among a set of possible solutions.
- To use dynamic programming, the problem must satisfy two conditions:

i) It must have ~~sub~~ optimal substructure

ii) It must have overlapping sub problems.

- Overlapping subproblems means that program can be broken down into smaller programs that are reused multiple times, rather than being solved ~~one at a time~~ independently each time.
- An optimal substructure means optimal solution can be constructed from the ~~problem~~ optimal solutions to subproblems
- Dynamic programming involve building a table or array to store the solutions to the ~~target~~ sub problem, & then using the table to calculate solution to larger problem.
- The process of filling up the table is done in a specific order, beginning from the smallest subsolution/problem & working up to the larger ones.
- Once table is complete, the solution to the larger problem, can be retrieved easily.
- Eg- Knapsack problem, TSP etc.

Principle of Optimality

- dynamic programming obtains solutions using principle of optimality.
- "in an optimal sequence of decisions or choices, the subsequence must also be optimal.
- when it is impossible to apply principle of optimality, it is impossible to obtain solution using dynamic programming.

Control Abstraction.

- Control abstraction is a technique used in dynamic programming to solve optimization problems.
- It consists of breaking down the problem into small problems & solve them recursively.
- The subproblems are usually solved using recursive function
- It takes input from the current state of the system & returns the optimal solution for that state.
- Solution to original problem is obtained by combining the solutions to subproblems.
- Control abstraction is often used to solve problems that involve making a sequence of ~~time~~ decisions over time.
- Eg. It can be used to determine optimal investment strategy for portfolio.
- To use control abstraction, it is important to ~~the~~ know the states of the system & the decisions that can be made at each stage.
- The recursive function is then defined to take these states as an input & return optimal solution for that state
- The function is implemented bottom-up approach, starting with base class, working up to final solution.
- computationally expensive.

Time Analysis of Control Abstraction.

- Time complexity of control abstraction depends upon the problem being solved & the ~~no. of states~~ recursive function used to solve it.
- In general, time complexity of control abstraction is determined by the no. of states & time required to solve the subproblems at each ~~stat~~ state.
- If there are S states in the optimization problem & it takes T(S) time to solve the problem, then time complexity is calculated as $O(S * T(s))$
- ~~In~~ The time reqd. to solve the optimization problem is proportional to the number of states & time required to solve the subproblem at each stage.
- T.C. of ~~optimization~~ control abstraction can be further refined based on the specific characteristic of the ~~system~~. problem & recursive function used to solve it.
- If a ~~problem uses~~ recursive func. ~~uses~~ top-down approach rather than a bottom-up approach, time complexity may be different.

OBST

- OBST stands for Optimal Binary Search Tree.
- It is type of data structure that is used to store a sorted list of elements in such a way that it is efficient to search & insert.
- ~~In a search tree,~~ A binary search tree is a tree structure that has 2 nodes which has only 2 children.
- In OBST, the keys are stored in the nodes of tree & structure of tree is optimized to minimize the average no. of comparisons required to search for a given tree.

- keys are sorted in A.O.
- Tree is constructed by adding the keys to the tree one at a time, starting with the middle key and working outwards.
- Left child of node → smaller key.
- right child → larger key.
- This process is repeated until all keys are added to the tree.

0/1 Knapsack using Dynamic Programming.

- well known combinatorial problem
- involves selecting a set of items from a given list, with the goal of maximizing the total value of the items while keeping the total weight below capacity.
- each item is either included or not included in the final solution.
- There are no fractional terms.
- makes the problem more difficult to solve than fractional knapsack
- define a recursive function that takes the items as input, their values & weights, & remaining weight that can be carried in the knapsack.
- for recursive case,
  ~~if~~ we either include the item in the knapsack or exclude it.
  ~~if we include it, max value is value of item~~
  if we include it, max val = value of item + max val obtained by solving subproblems with remaining wts & items.
- if we exclude item, max value is obtained by solving subproblem with remaining weight & remaining items
- We take maximum of these two options as solution for the current state
- Soln is then obtained by calling the recursive function with full set of items & full weight capacity of knapsack

Unit 4.

1) Backtracking.

- general algorithmic technique that considers searching every possible combination in order to solve a computational problem.
- involves trying out all possible solutions & then undoes the changes made to the system in case the solution turns out to be incorrect.
- This process is repeated until the correct solution is found or all possible solutions have been exhausted.
- It recursively tries to build a solution incrementally.
- More detailed description of the backtracking process:

1) Begin by selecting a starting point & initial decision.
2) Consider all possible choices that can be made based on the current decision
3) For each choice, do the foll.
   a) Make the choice & apply it to the current problem.
   b) Check to see if the choice leads to a solution. If it does, return the solution
   c) If the choice does not lead to a solution, undo the choice & try the next one.
4) If none of the choices lead to a solution, ~~undo the~~ go back to the previous decision point & make a different decision/choice.

Repeat this process until a solution is found or all possible choices have been exhausted.

- Backtracking is computationally expensive.
- It is therefore often used as last resort when other methods have failed.

Backtracking principle

- principle of backtracking involves incrementally building a solution to a problem, one piece at a time, & undoing the changes made to a system when it determines particular solution cannot be completed.
- This process is repeated until correct solution is found or all possible solutions have been exhausted.
- Key to using backtracking is to identify the subproblems that make up the large problem & design an algorithm to solve the subproblems incrementally.

* Backtracking control abstraction.

- Backtracking control abstraction is a technique used to break down a larger problem into smaller subproblems that can be ~~used~~ solved individually using a backtracking algorithm.
- It is a way of organizing the solution process to make it more easier & efficient to understand.
- Process:

⇒ ~~Divide the problems into smaller subproblems~~

1) Identify the subproblems that make up the main problem. Each subproblem should be small enough to be solved ~~indi~~ individually, but together they should cover the whole problem.
2) Design a backtracking algorithm to solve each subproblem individually.
The algo. should make a series of decisions & check to see if each decision leads to solution.
If it does, the ~~do~~ algorithm should return a solution.
If it does not, algo. should undo the decision & try the next one

3. Solve each subproblem using backtracking algorithm. As each subproblem is solved, the solution can be used to solve a larger problem.
4. Combine the solutions to the subproblems to form solution to the larger problem.

- By using backtracking control abstraction, the problem can be divided into smaller & more manageable pieces, making it easier to find a solution.
- It allows the algo. to focus on one subproblem at a time rather than trying to solve the entire problem at once.

* Time analysis of control abstraction.

- time complexity depends on the size of the problem & the efficiency of backtracking algo. to solve the subproblems.
- In general it is $O(b^d)$ where b is the branching factor & d is the depth.
- Time complexity increases exponentially with the size of the problem.
- Eg. if branching factor = 10 & depth = 5, than complexity is $10^5 = 100,000$
- Time complexity of backtracking control abstraction can be reduced by using techniques such as pruning, which eliminates branches of search tree that do not lead to a solution.

* 8 Queen's

- 8 queen's problem is a classic computational problem that involves placing 8 queens on the chessboard such that no queen attacks the other.

- A queen can attack any piece in the same row, column or diagonal as the queen.
- Goal is to find a way to place 8 queens such that no two queens are attacking each other.
- One way to solve 8-queens is by backtracking algorithm.
- The algorithm works by making a series of decisions to see & checking to see if each decision is valid & leads to a solution.
- Here the decisions is placement of queens on the chessboard.
- Process.

1) Place the first queen on the chessboard.
2) Check if other queens are attacking the first queen. If there are no attacking queens go to step 3. or else If there are attacking queens, go to step 5.
3) Place the second queen on the chessboard.
4) Check to see if other queens (including 2nd) are attacking each other. If no, go to step 5. If there are, go to step 6.
5) If all 8 queens are placed on the chessboard, without attacking each other, return solution
6) If the queens are attacking each other, remove the last queen placed & try a different placement for it. Repeat until solution is found or all possible placements have been tried.

* Graph coloring problem.

- classic problem that involves coloring of vertices in a graph such that no to two adjacent vertices have the same colour.
- A vertex is adjacent to another vertex if there is an edge connecting the two vertices.
- goal of the problem is to find minimum no. of colors needed to colour the graph such that no two adjacent vertices have the same color.

- One way to solve this problem is by backtracking algorithm
- It involves making a series of decisions & checking to see if each decision leads to a valid solution.
- Decisions in this case are colors assigned to the vertices of the graph.
- Process.

1) Begin by selecting a starting vertex & assigning it a color.
2) Consider the vertices adjacent to the starting ~~vertices~~ vertex
3) For each adjacent vertex:
   a. Assign a color to the vertex
   b. Check if the color ~~not~~ conflicts with any colors assigned to adjacent vertices. If they do, go to step 4. If not, go to step 5
4) If there is conflict, undo the color assignment & try a different color. Repeat this process until a valid color is found or all possible colors have been tried.
5) If all the adjacent vertices are colored without any conflicts go to the next uncolored vertex & repeat the process until all vertices have been colored.

* Sum of subsets problem.

- Sum of subsets is a problem in which a set of numbers is given & goal is to find all the possible subsets of the set that add up to a particular sum.
- Eg. given set {1, 2, 3, 4} & the sum 6 the possible subsets ~~are~~ that add up to 6 are {2, 4}, {1, 2, 3} {1, 5}.
- One way to solve the problem using backtracking algorithm.
- The algorithm works by making a series of decisions & checking to see if each decision leads to a valid solution.
- The decision in this case is elements of set to be

involved in the subset.

- Process

1) Begin by selecting first element of set & adding it to subset.
2) Check if the elements sum up to be the target sum. If yes, add the subset to list of solutions & go to step 4. If not, go to step 3.
3) If sum of subset is not equal to the target sum, add next element to the subset & repeat the process until a solution is found or all elements in the set have been considered.
4) If all elements in the set are considered & no solutions has been found, remove the last element added to the subset & try a different element.

Repeat the process until all possible subsets have been tried.

★ BRANCH AND BOUND.

- Branch & bound is a technique used to solve optimization problems.
- It involves choosing optimal solution from a set of potential solutions.
- It involves dividing the set of potential solutions into smaller subsets & then (branches) & searching for the optimal solution within each subset (bound)
- It is similar to backtracking as it makes a series of decisions & then undoing the decisions if they do not lead to the solution
- The only difference is B&B involves in pruning of branches which do not lead to the solution.
- It uses bounding function which eliminates branches that do not lead to optimal solution.

- allows the algorithm to focus on the most promising branches & reduces the time & resources to find optimal solutions.
- It is good for optimization problems as it focuses on the branch that provide optimal sol" & prune off the others.
- It can be time consuming & computationally intensive.
- often used as last resort when other methods fail.

Principle of Branch & Bound.

- The principle of B & B is to divide the set of potential sol" into ~~a set of~~ smaller subsets (branches) & find the optimal solution from these subsets (bound).
- It involves using a bounding function to estimate the optimal solution in the subset & pruning off the branches which do not lead to a solution.
- This allows algorithm to focus on more promising branches & reduces time & resources needed to find the optimal solution.
- Process

1) Divide the set of potential solutions into smaller subsets (branches)
2) Using a bounding function to estimate the optimal solution in each branch
3) Prune the branches which do not lead to solutions.
4) Search for optimal solution within remaining branches using a search algorithm.
5) Repeat the process until optimal sol" is found or all branches have been explored.

Control Abstraction.

- control abstraction in B & B is breaking down a larger

optimization problem into smaller subproblems that can be solved individually using branch & bound.

- This allows the algorithm to focus on one subproblem at a time, rather than trying to solve the whole problem at once.
- Process

1) Identify the subproblems that make up the larger problem. Each problem should be small enough to be solved individually but together should cover the whole problem.
2) Design a branch & bound algorithm to solve each subproblem individually. The algorithm should divide the set of potential ~~problems~~ solutions into smaller subsets (branches). Using bounding function, estimate the optimal solutions from the subsets & prune the branches that do not lead to the optimal solution. It should then search the optimal solution within the remaining branches using a search algorithm.
3) Solve each subproblem using b & b algorithm. As each subproblem is solved, the sol^n can be used to solve larger problem.
4) Combine the sol^n to the subproblems to form the solution to the larger problem.

Time analysis of Control Abstraction

- Time complexity of ~~solving~~ control abstraction in branch and bound ~~is $O(b^{n-d})$~~ depends on size of the problem, efficiency of bounding function & efficiency of ~~sequen~~ search algorithm used to solve the subproblems.
- In general, the time complexity of b & b is $O(b^d)$ where b is branching factor & d is the depth.
- Time complexity increases exponentially with size of problem.
- Eg- b = 10 d = 5
  Time complexity = 10^5 = 1,00,000

- Time complexity of control abstraction in b& b can be reduced by using more efficient bounding function & search algorithm.
- Eg- backtracking >> bruteforce as searching algorithm

★ First In First Out Branch & Bound.

- FIFO is a $ queing mechanism in which first element to enter the queue is also the first element to be removed from the queue.
- In branch & bound it is used to manage the order in which branches are explored.
- Eg. algo. can use FIFO queue to store the branches that have been generated & then explore these branches in the order they were added to the queue.
- This ensures that the algo. does not get stuck in a loop & all branches are explored in systematic manner.
- Process

1) Begin by dividing the set of potential ~~branches~~ solutions into smaller subsets (branches)
2) Using a bounding function, estimate the potential value of solutions in each branch
3) Add the branches to FIFO queue in order they are generated
4) Remove the first branch from the queue & search for the optimal solution using a searching algorithm.
5) Repeat this process until the optimal solution is found or all branches have been explored.

★ Last in First Out Branch & Bound.

- LIFO is a queing mechanism in which last element to enter the queue is also the first one to get out.
- In B&B, LIFO can be used to manage the order in

which the branches are explored.
- For eg, the algo. can use a LIFO stack to store the branches that have been generated, & then explore the branches in reverse order in which they were added to stack.
- Process

1) Begin by dividing the set of potential solutions into smaller subsets.
2) Using a bounding function, estimate the potential value of solutions in each branch.
3) Add the branches to a LIFO stack in the order in which they were generated.
4) Remove the last branch ~~from the~~ added to stack & search for optimal sol^n within the branch using search algorithm.
5) Repeat the process until the optimal solution is found or all branches have been explored.

Ch. 5
Amortized Analysis

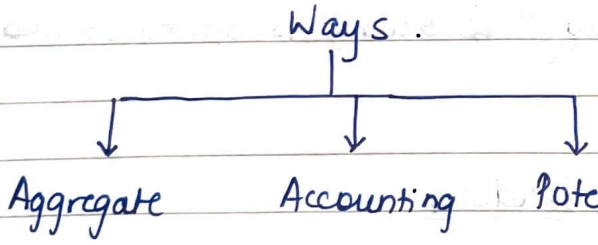
★ Amortized Analysis

- method of analysing the cost of an algorithm.
- done by considering every operation & its cost & determining cost of the algorithm
- used to analyse algorithms that perform a sequence of operations where ~~every operatic~~ cost of every operation varies.
- goal is to find out average cost per operation of the algorithm over a long sequence of operations.
- used for determining efficiency of algorithm
- comparing performance of different algorithms.

★ Difference between amortized analysis & average case analysis?

- average case : we consider avg of all possible set of inputs
  amortized : we consider avg. of sequence of operations.
  ↳ assumes worst case input.

★



1) Aggregate method
- involves grouping together the seq. of operations & analyzing their total cost.
- The total cost is then divided by the no. of operations to determine avg. cost per operation.
- operations take worst case time $T(n)$.

$$\frac{T(n)}{n}.$$

2) Accounting
- based on adding charges called credits to the operation
- charges are added to each operation
- Amortized cost = Actual cost + credit.
- Eg. To push obj = $2
  we can use $1 to pay for push
  remaining $1 for credit.
- If we go on pushing items, credit will get accumulated over time.
- popping object → credit used to prepay for pop operation
- If we charge push > pop, we don't have to charge anything for pop operation.
- This is the same for multi-pop.
- ensures credits → non-negative

Total amortized cost (push)(pop), multipop = O(n)

Avg cost = $\frac{O(n)}{n} = 1$.
each operation.

3) Potential
- Method is similar to accounting method.
- concept is "prepaid" is used
- potential "energy" ~~is~~ which is stored is used to pay for future operations.

✗ Time-space tradeoff.

- situation where time efficiency is achieved at the cost of space or space-efficiency is achieved at the cost of time.
- consider prog. like compilers where variables & symbols are stored in a symbol table. Now if we ~~consider~~ store the entire table in the program, it will be easier to search the variable & therefore time will be reduced but memory req ↑.

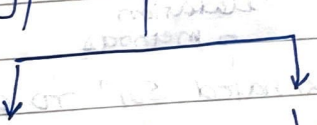
If we do not store symbol table in program, memory will be reduced but processing time will be more.

* Tractable problem
  - The problems which can be solved in polynomial time.
  - It can reasonable amt. of time using algos & methods availab
  - Eg. Finding an element in a list.

* Non-tractable problem
  - cannot be solved in polynomial time
  - difficult to solve
  - Eg. TSP, Knapsack problem.

* Randomized Algorithm.

  - uses randomness as a part of computation
  - takes input random numbers, or use randomness of some sort for computation
  - decision is based on current output.
  - execution time may vary from run to run
  - output may be different for the same inputs every time
  - Types



1) Monte-Carlo
   - type in which, it produces different outcomes for same input on each execution.
2) Las Vegas
   - type which produces same outcome for same input on each execution.

Advantages

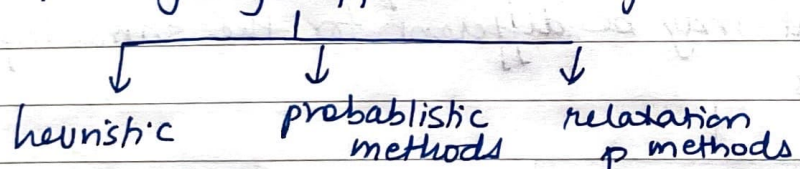
1) easy to implement
2) robust
3) efficient than traditional algos.

Disadvantages

1) small degree of error may be dangerous.
2) ~~predictable~~ non-predictable outputs.

☆ Approximation Algorithm

- used to find the approximate solution to optimization problems
- It is not the exact optimal solution but somewhere close to the optimal solution.
- associated with NP-hard where it is very difficult to find the solution to the problem in polynomial time.
- used to solve problems which can be solved using poly. algo but not efficient due to large datasets.
- philosophy behind it is "find a good solution fast".
- won't get exact solution but will get app. solution fast.
- Ways of designing approximation algos:



- Accuracy ratio: Ratio of approximated soln to actual soln

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

s_a = approximate solution

$r(s_a)$ = ~~approx.~~ accuracy ratio

$r(s_a)$ closer to 1, better approx. soln.

- Performance ratio
  Denoted by R_A.
  by knowing performance ratio, one can judge quality of app. algo.
  Closer to 1 means better approx.

★ Embedded ~~Algo~~ System.
- computer system with ~~embedded~~ dedicated function
- does not have keyboard, mouse, monitor but has processor, software & memory.
- permanent part within big system.
- Eg. Smart card, Mobile.
- Characteristics

★ 1) Single functionality - dedicated to one functionality.
2) Tightly constrained - computed system with tightly coupled hardware & software integration
3) Reactive & Real time - responds to event & triggers.
4) Programmable - contain programmable unit.

★ Embedded System Scheduling.

1) Fixed Priority.
- assigns all priorities to the tasks at design time.
- priorities remain constant throughout the lifetime of task
- Implementation.
  Step 1: