(DAA)

Unit - 3

Greedy & Dynamic Programming Algorithmic Strategy

* Greedy Technique (Algo) -

- best approach according to that moment.
- no future prospect seen.
- follows local optimal choice of each stage with intent of finding local optim

→ feasible solⁿ (selection) → choice satisfies problem constraints
→ optimal solⁿ → best soln of current
→ Irrevocable (can't change later)

Applications → 1) Knapsack Problem (Min. Cost) 2) Job Scheduling (max Profit)
3) Minimum Spanning Tree 4) Optimal Merge Pattern
5) Huffman Coding (Min. Risk) 6) Dijkstra's Algo.

Characteristics General of G.S.

→ 1) Greedy Choice Property - finding solⁿ → we solve subproblems, whichever choice is best considered on current solⁿ & no future prospect

→ 2) optimal subtree - If an optimal solⁿ to problem contains solⁿ to sub problems.

Knapsack Problem.

- Binary or 0/1 knapsack.
- Item can't be broken down into parts.
- n objects from $i = 1, 2, \dots n$.

weight $\rightarrow w_i$.

profit associated with each object

~~from i = 1.~~

knapsack $\rightarrow$ carry most weight W

1) choose object that give max. profit.
2) Total weight of object $\leq W$.

Time complexity $\rightarrow$ items are presorted $\rightarrow O(n)$

items needs to be sorted $\rightarrow$

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

Fractional Knapsack

↳ breaking of items is allows

ie items can be taken in fractions.

↳ same representation as binary knapsack

Time complexity $\rightarrow$ exhaustive approach - $O(2^n)$

sorted approach - $O(n \log n)$

DOMS | Page No. | Date / /

* Scheduling Algo -

schedule n jobs out of a set of N jobs on a single processor which maximizes profit as much as possible.

- schedule S in an array of slots(s).
- $S(t) = i$ then $t \le d_i$.
- each job → max once.

· goal → feasible solution S while maximizes profit of scheduled jobs

Time complexity → worst case N job may search N slots hence $TC = O(n^2)$ → ds used → $TC = O(n)$.

* Activity Selection Problem -

Schedule max no. of activites that need exclusive access to resource.

span of activity → starting & finishing time → n activities.

Time complexity → each activity at worst there will be (n-1) comparisions hence $O(n^2)$.

sorting single scan → $TC = O(n \log n)$.

Dynamic Programming -

- US Mathematician Richard Bellman → 1950.
- technique → problem with overlapping subproblem
- sub problem → solve one by one
- sub problem → not independent
- bottom up approach.
-

features/elements → 1) optimal subtree → principle of optimality → optimal s
2) overlapping subproblems.
→ problem → subproblems.
soln subproblems → final soln
avoid repition of work & efficient soln

* Applications
- → Multistage Graph
- → Travelling Sales Man Problem
- → Matrix Mutiplication
- → Longest Subsequence Prob.
- → Max. Flow Prob.
- → All Pair Shortes Path Problem.

* Principle of Optimality -

- dynamic programming algo → soln from principle of optimality.
- principle → in an optimal sequence of decisions for choices, each subsequence also be optimal.

- not possible → principle → impossible to obtain soln using dp.

- eg → finding shortest path in given graph uses principle of optimality

DOMS Page No.
Date / /

Greedy method | dynamic programming

Greedy method
- obtain → optimum sol^n
- set of feasible sol^n & pick optimum sol^n
- optimum sol^n → no revising sol^n
- no guarantee of optimum sol^n

dynamic programming
- obtain optimum sol^n
- no set → feasible sol^n.
- consider all possible sequen
- guaranteed sol^n by principle of optimality

Divide & Conquer | Greedy Algorithm.

Divide & Conquer
- obtain sol^n for problem.
- subproblem → independently solved.
  ↳ all subproblem → sol^n
- Duplication in subsol^n → neglected.
- Less efficient → rework on sol^n.
- eg → Quick sort binary search.

Greedy Algorithm.
- optimum sol^n.
- set feasible sol^n → optimum selected
- optimum selection → without revising
- no guarantee of optimum sol^n
- eg → Knapsack, finding minimum spanning tree.

Binomial Coefficients

- problems → overlapping subproblems property → find sol^n → same problem again.
- store sol^n → subproblem → table → refer it needed.

optimal substructure
↳ $$C[i,j] = \begin{cases} 1, & \text{if } i=j \text{ or } j=0 \\ C[i-1, j-1] + C[i-1]j \end{cases}$$

Time complexity → $O(nk)$.

* Optimal Binary Search Tree (OBST) –

- similar to BST only catch → arrangenode acc to prob of searching their frequently i.e. node of high probablity → being searched → closest to root

eg:- word from dictionary searching

$$C(i,j) = \min_{i<k\leq j} \{ C(i,k-1) + C(k,j) \} + W(i,j)$$

$$W(i,j) = W[i, j-1]$$

- minimizes expected search cost.

complexity →

$$T(n) = \sum_{m=1}^{n} \sum_{i=1}^{n-1+1} \sum_{j=1}^{n-1+1} O(1)$$

$$= O(n^3).$$

0/1 Knapsack Problem -

- Greedy algo
- items are either completely or no times are filled in knapsack.
- Time complexity → $O(nM)$ → $n \rightarrow$ Items
  $m \rightarrow$ capacity of knapsack.
- Also solved using DP.

Chain Matrix Multiplication.

what order, n Matrices A_1, A_2, A_3 An should be multiplied to minimize computations to derive result.

$A_1 = 5 \times 4$ $A_2 = 4 \times 6$ $A_3 = 6 \times 2$.

Now, → $(A_1 \cdot A_2) \cdot A_3 = 180$.

$A \cdot (A_2 A_3) = 88$.

simply parenthesis correctly in order to archieve min. computation

Time Complexity -

Iterative without memo → $O(n^3)$

Recursive without memo → $O(2^{n-1})$.