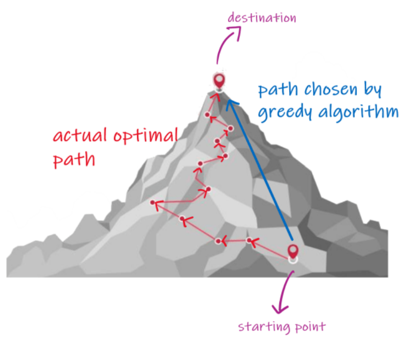
Greedy and DP

Tuesday, December 6, 2022 11:01 AM

Greedy approach -

- to achive optimization we introduced greedy method.
- It is a problem solving paradigm
- It follows principal of "making the choice that seems best at that point of time".
- It is very likely that a greedy algorithm may not give the best solution
- In greedy method once you make a choice you never backtrack it.
- We can not determine whether aur choice is optimal or not
- There will be only one optimal solution
- Analyzing the run time of a greedy algorithm is also easier
- Failing of greedy method,



Greedy method will say that to go on top go straight up on mountain but it is not possible to do.

- Ex. You have 10 hrs and you want to learn max courses in this time
  Physics class - 1hr
  Tennis - 3hrs
  Chemistry - 2hrs
  Cooking - 4hrs
  Cricket - 5hrs

  Using greedy method we will choose those courses having min time,
  Physics + Chemistry + cooking + tennis = 1+3+2+4 = 10

Control abstraction for Greedy algo -

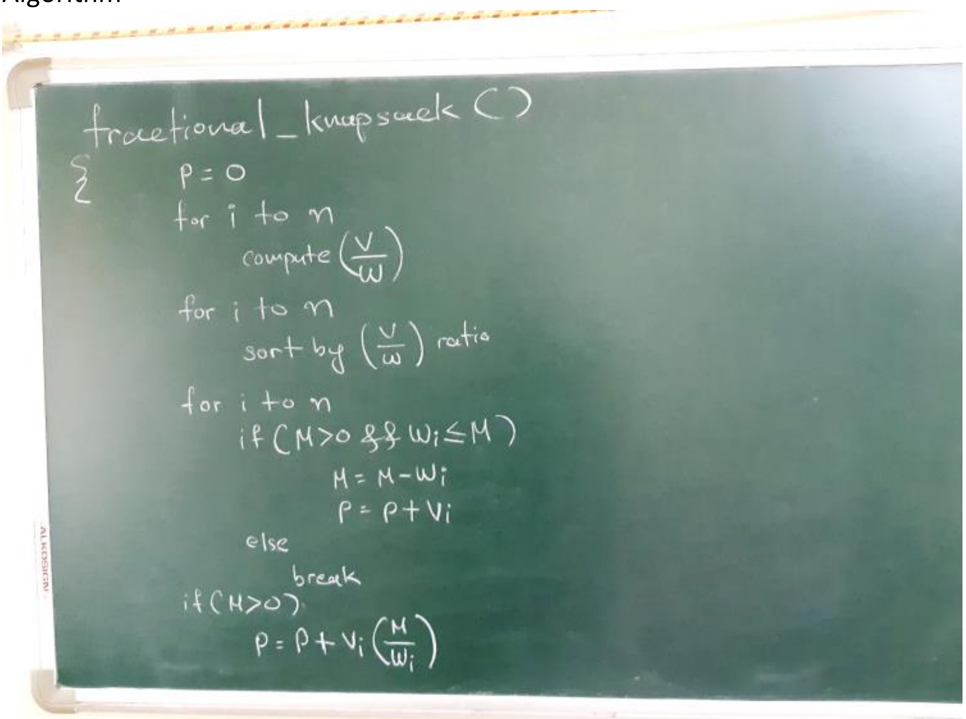
- Algo_greedy(L,n)
- L - List of solutions, n - size of solution
- for i = 1 to n
  choice = select(L)
  if(feasible(choice))
  solution = choice + solution
- end

Applications of Greedy -

- Knapsack
- Minimum Spanning Tree
- Shortest path
- Job scheduling
- Huffman

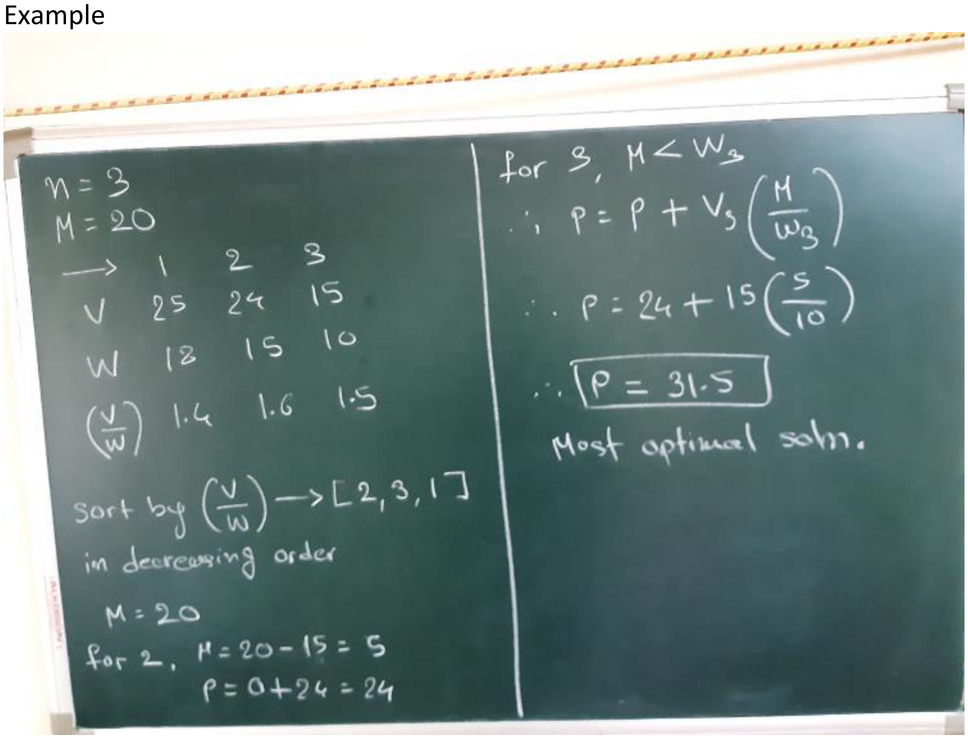
Fractional Knapsack -

Algorithm



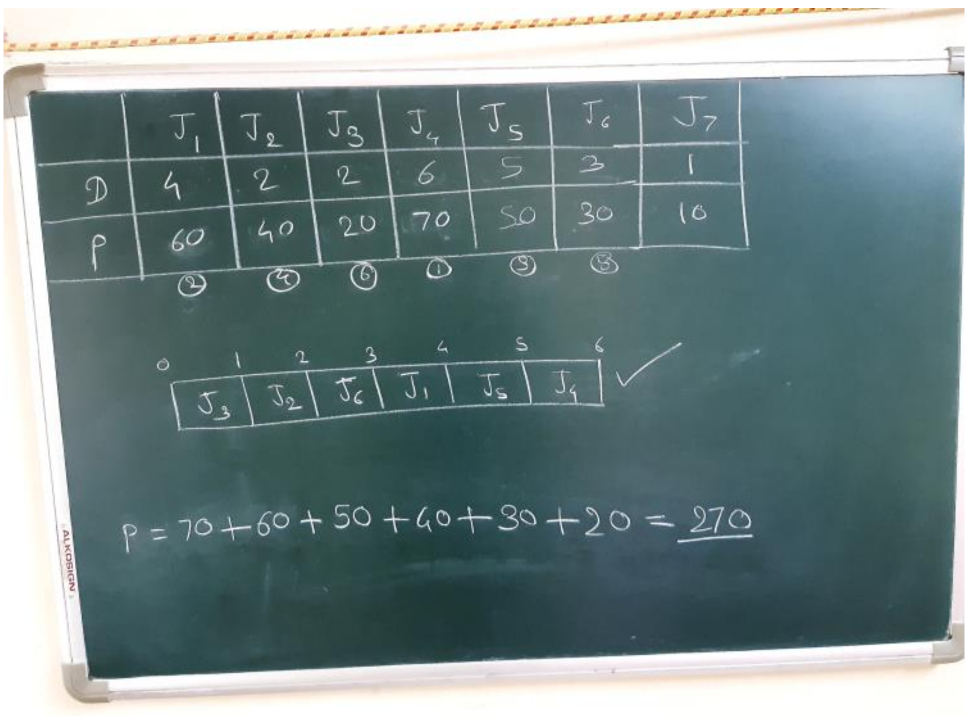
Time Complexity: O(N * log N)
Auxiliary Space: O(N)

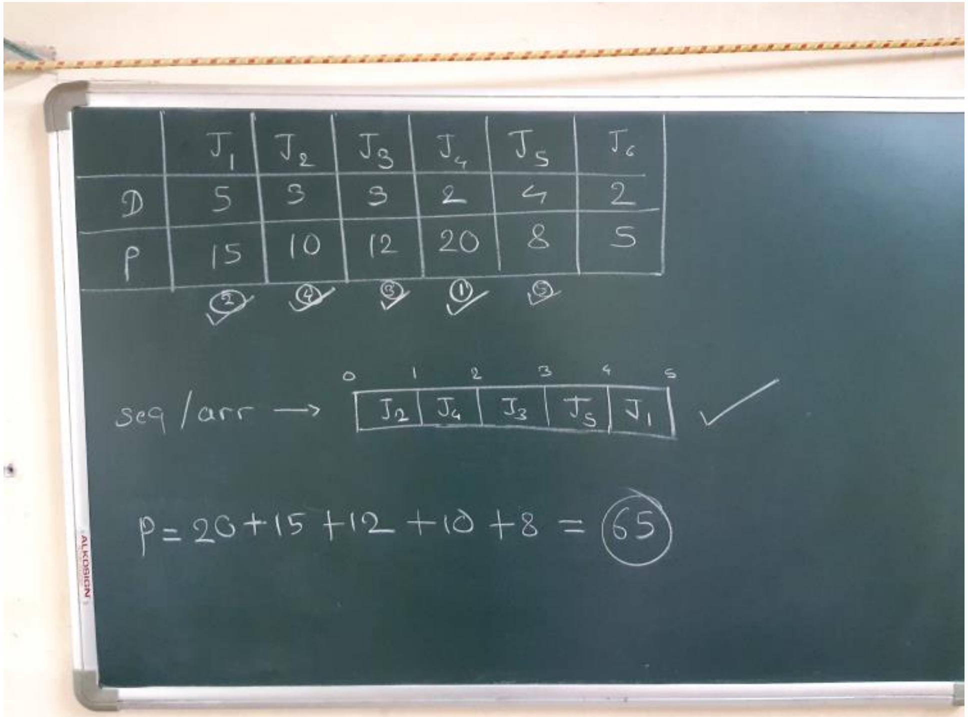
Example



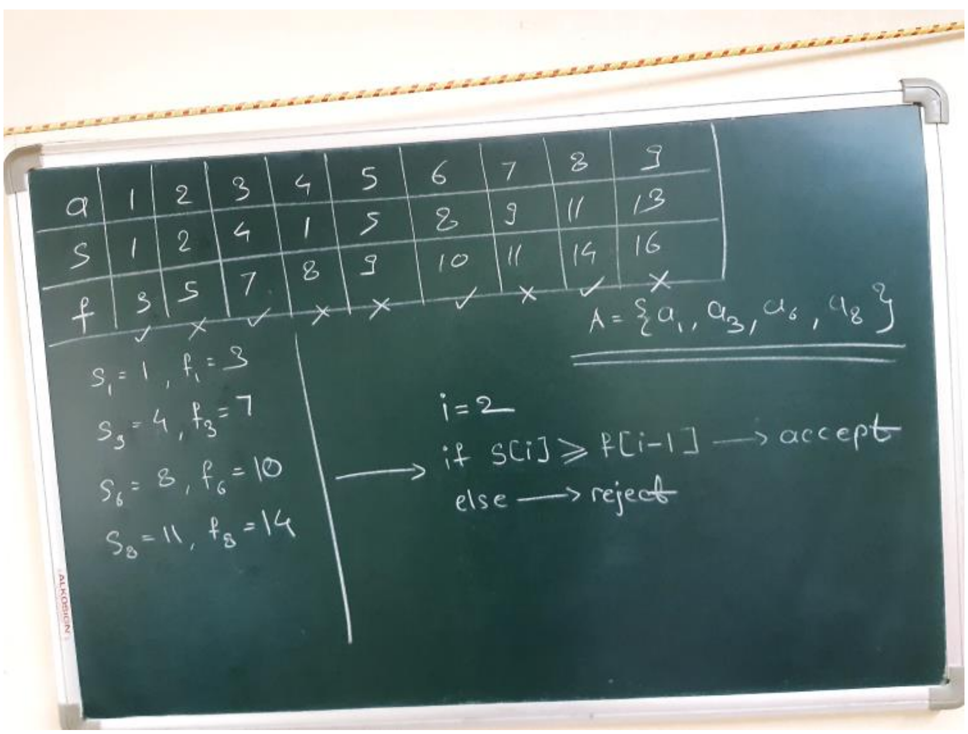
Job Scheduling -

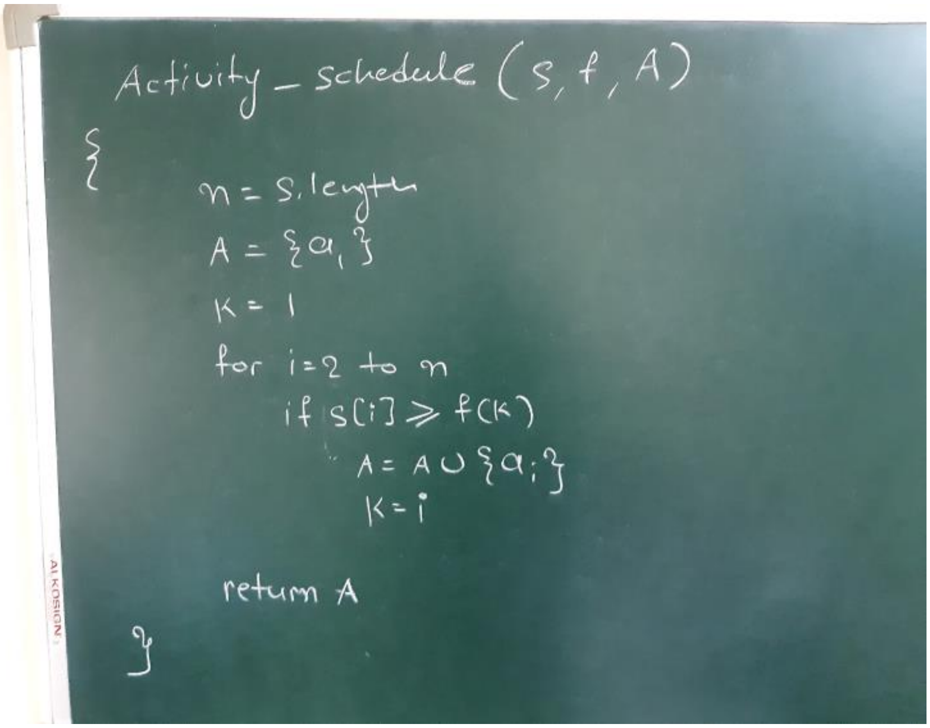
- deadlines to perform the jobs
- profits associated with this jobs
- take a sequence/array with the size of max deadline given
- start with max profit job and so on
- bounds for placing the job is deadline and initial bound that starting point
- keep the job farthest from initial bound and within deadline
- Time complexity - O(n^2)
- Space complexity - Extra space used by sequence array





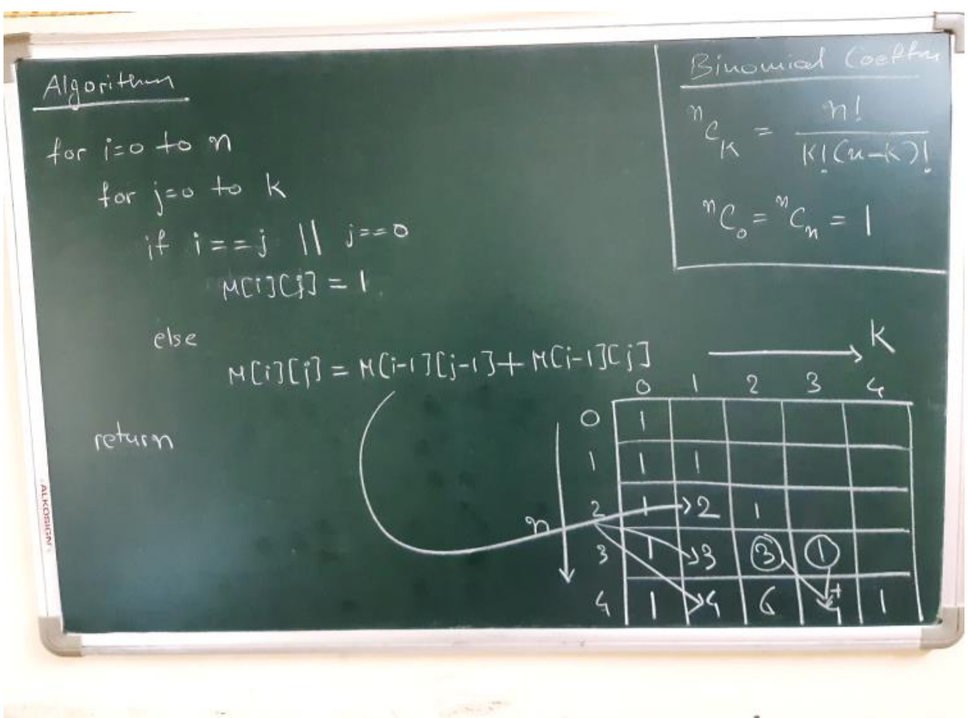
Activity selection problem -





Time Complexity: O(N)
Auxiliary Space: O(1)

Binomial Coefficient using DP -



0/1 Knapsack using DP -

The idea is to use recursion to solve this problem. For each item, there are two possibilities:

1. Include the current item in the knapsack and recur for remaining items with knapsack's decreased capacity. If the capacity becomes negative, do not recur or return -INFINITY.
2. Exclude the current item from the knapsack and recur for the remaining items.
3. Finally, return the maximum value we get by including or excluding the current item.

```
int include = v[n] + knapsack(v, w, n - 1, W - w[n]);
int exclude = knapsack(v, w, n - 1, W);
```

DP -

- DP is a problem solving approach or programing paradigm.
- It solves given problem by dividing it into sub problems using recursion.
- It stores results of subproblems to avoid re-computation of subproblems.

Characteristics Components of DP -

Overlapping Subproblem -

In DP we store results of subproblems to avoid re calculations. But if a problem does not have common subproblem or overlapping subproblem then DP can't be applied to it.
Ex. Binary Search - it does not have any overlapping subproblem

Optimal Substructure -

An optimal solution can be found using optimal solutions of its subproblems.
If node x lies in the shortest path from a source node U to destination node V then the shortest path from U to V is a combination of the shortest path from U to X and the shortest path from X to V.

DP control abstraction -

- Control abstraction for dynamic programming is shown below :

```
Algorithm DYNAMIC_PROGRAMMING (P)
if solved(P) then
    return lookup(P)        If P is already solved, then retrieve stored answer
else
    Ans ← SOLVE(P)
    store (P, Ans)
end

Function SOLVE(P)
if sufficiently small(P) then
    solution(P)             // Find solution for sufficiently small problem
else
    Divide P into smaller sub-problems P1, P2, ..., Pn
    Ans1 ← DYNAMIC_PROGRAMMIN(P1)
    Ans2 ← DYNAMIC_PROGRAMMIN(P2)
    ...
    Ansn ← DYNAMIC_PROGRAMMIN(Pn)
    return(combine(Ans1, Ans2, ..., Ansn))     Combine solutions of smaller sub problems in order to achieve the solution to larger problem
end
```