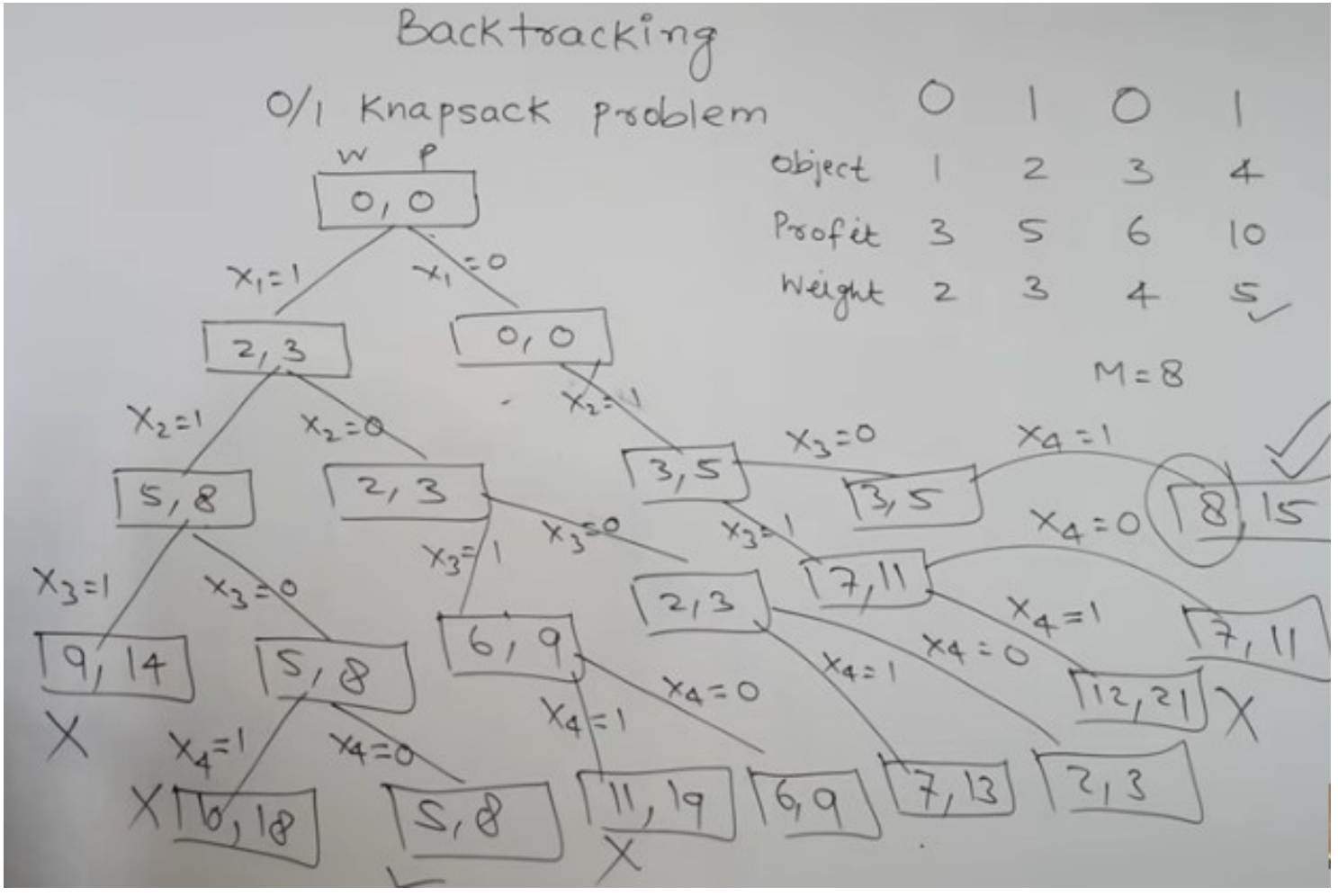
Greedy

Greedy Algorithm	Divide and conquer	Dynamic Programming
Follows Top-down approach	Follows Top-down approach	Follows bottom-up approach
Used to solve optimization problem	Used to solve decision problem	Used to solve optimization problem
The optimal solution is generated without revisiting previously generated solutions; thus, it avoids the re-computation	Solution of subproblem is computed recursively more than once.	The solution of subproblems is computed once and stored in a table for later use.
It may or may not generate an optimal solution.	It is used to obtain a solution to the given problem, it does not aim for the optimal solution	It always generates optimal solution.
Iterative in nature.	Recursive in nature.	Recursive in nature.
efficient and fast than divide and conquer. For instance, single source shortest path finding using Dijkstra's Algo takes O(ElogV) time	less efficient and slower.	**more efficient but slower than greedy. For instance, single source shortest path finding using Bellman Ford Algo takes O(VE) time.**
extra memory is not required.	some memory is required.	more memory is required to store subproblems for later use.
Examples: Fractional Knapsack problem, Activity selection problem, Job sequencing problem.	Examples: Merge sort, Quick sort, Strassen's matrix multiplication.	Examples: 0/1 Knapsack, All pair shortest path, Matrix-chain multiplication.

Greedy vs Backtracking

• Method of obtaining optimum solution, without revising previously generated solutions	• Method of obtaining optimum solution, may require revision of previously generated solutions
• State space tree not created	• State space tree created
• Less efficient	• Efficient to obtain optimum solution
• Eg. Job sequencing with deadlines, Optimal storage on tapes	• Eg. 8 Queen problem, Sum of subset problem

Greedy algorithm	Very large Number of feasible solutions	Does not always gives the optimal solution.
Divide and conquer	Algorithm efficiency	Recursion overhead
Dynamic Programming	Recalls performed calculation	Only for overlapping sub problems
Backtracking Algorithm	1.quick test 2.Pair matching 3. Following real life concept	1.Not widely implemented. 2. Cannot express left-recursive rules 3. More time & complexity
Branch and bound	Very large Number of feasible solutions	Finding pruning strategies require clever thinking technologies


Backtracking
0/1 Knapsack problem
0 1 0 1
object 1 2 3 4
Profit 3 5 6 10
Weight 2 3 4 5
M=8
W P
0,0
x1=1
x1=0
2,3
0,0
x2=1
x2=0
x2=1
5,8
2,3
3,5
x3=0
x4=1
3,5
8,15
x3=1
x3=0
x3=1
x3=0
x4=0
9,14
5,8
6,9
2,3
7,11
x4=1
7,11
x3=1
x3=0
x4=1
x4=0
x4=0
x4=1
12,21
X
X
x4=1
x4=0
x4=0
x4=1
X
6,18
5,8
11,19
6,9
7,13
2,3
X

0/1 Knapsack Dynamic Programming

	0	1	2	3	4
P	0	1	2	5	6
wt	0	2	3	4	5

$n = 4$

$m = 8$

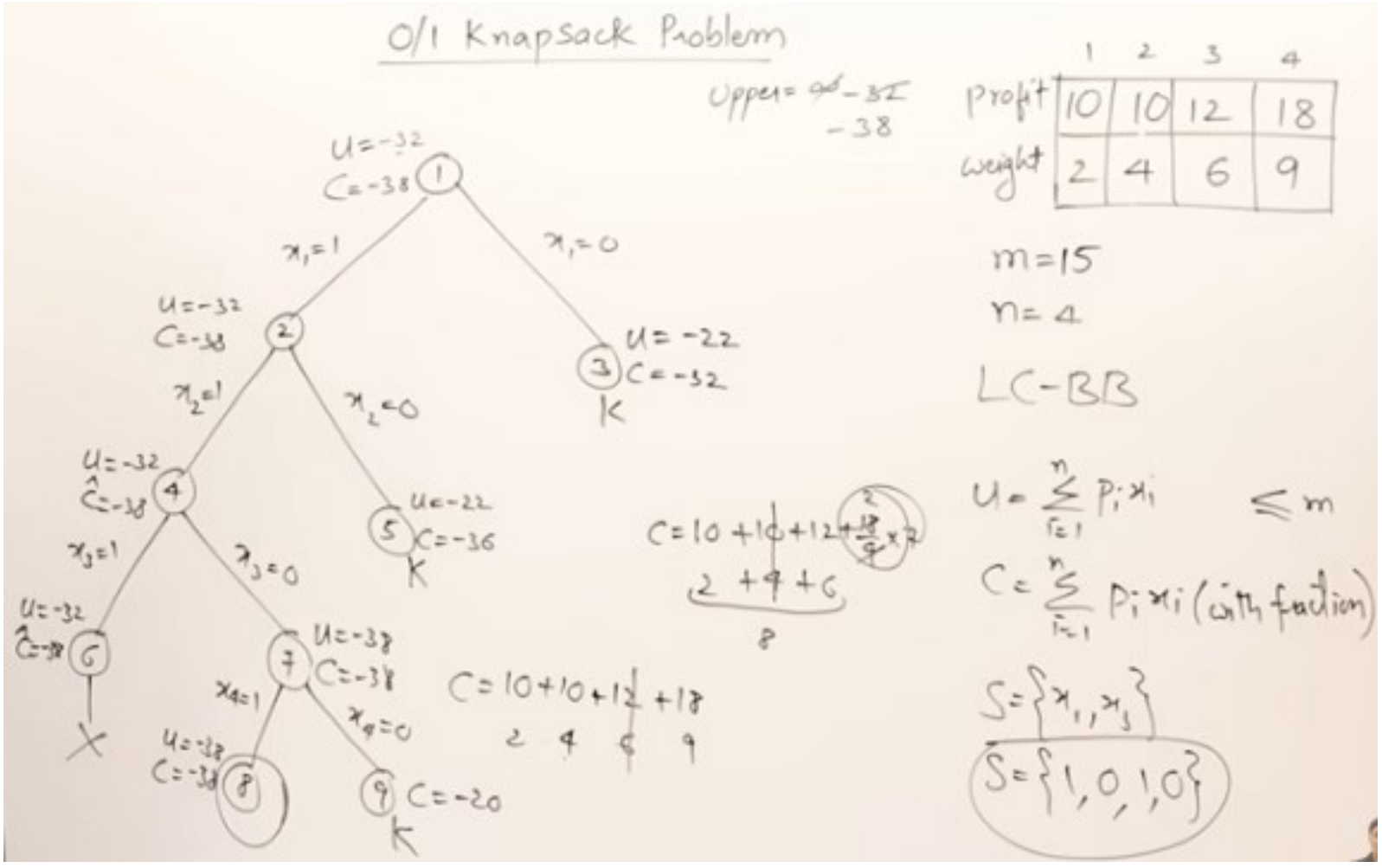
```
i = n; j = m;
while(i>0 && j>0)
{ if (k[i][j] == k[i-1][j])
  { cout << i << "=0" << endl; i--; }
  else { cout << i << "=1" << endl; i--; j=j-wt[i]; }
}
```

i \ w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (P=1, w=2)	0	0	1	1	1	1	1	1	1
2 (P=2, w=3)	0	0	1	2	2	3	3	3	3
3 (P=5, w=4)	0	0	1	2	5	5	6	7	7
4 (P=6, w=5)	0	0	1	2	5	6	6	7	8

```
main()
{
    int P[5]={0,1,2,5,6};
    int wt[5]={0,2,3,4,5};
    int m=8, n=4;
    int k[5][9];

    for(int i=0; i<=n; i++)
    {
        for(int w=0; w<=m; w++)
        {
            if(i==0 || w==0)
                k[i][w]=0;
            else if(wt[i] <= w)
                k[i][w]=max(P[i]+k[i-1][w-wt[i]],
                            k[i-1][w]);
            else k[i][w]=k[i-1][w];
        }
    }
    cout << k[n][w];
}
```

using Branch and Bound



Dijkstra's Algorithm