Batch: A3	**Roll No.: 16010121045**
Experiment No. 5	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of Knapsack Problem using Greedy strategy

Objective: To learn the Greedy strategy of solving the problems for different types of problems

CO to be achieved:

CO 2 Describe various algorithm design strategies to solve different problems and analyse Complexity.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://lcm.csa.iisc.ernet.in/dsa/node184.htm**
4. **http://students.ceid.upatras.gr/~papagel/project/kruskal.htm**
5. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/kruskalAlgor.html**
6. **http://lcm.csa.iisc.ernet.in/dsa/node183.html**
7. **http://students.ceid.upatras.gr/~papagel/project/prim.htm**
8. **http://www.cse.ust.hk/~dekai/271/notes/L07/L07.pdf**

Pre Lab/ Prior Concepts:

Data structures, Concepts of algorithm analysis

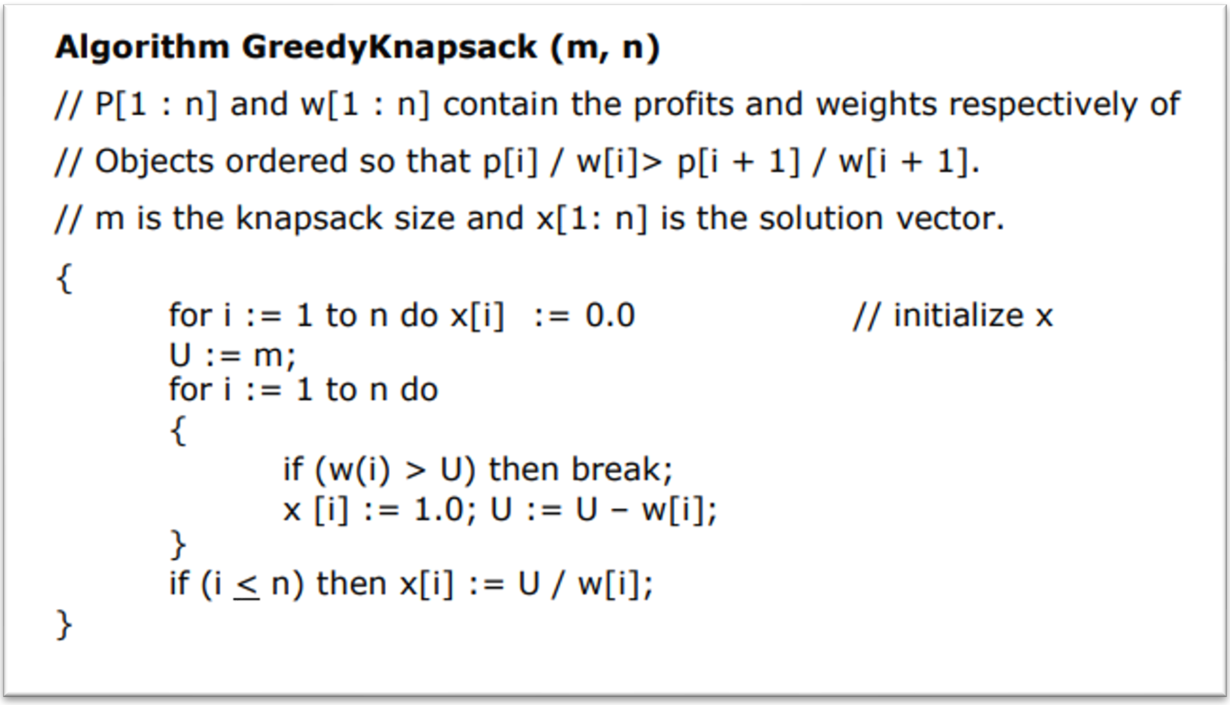
Historical Profile:

The knapsack problem represents constraint satisfaction optimization problems' family. Based on nature of constraints, the knapsack problem can be solved with various problem saolving strategies. Typically, these problems represent resource optimization solution.

Given a set of n inputs. · Find a subset, called feasible solution, of the n inputs subject to some constraints, and satisfying a given objective function. · If the objective function is maximized or minimized, the feasible solution is optimal. · It is a locally optimal method.

New Concepts to be learned:
Application of algorithmic design strategy to any problem, Greedy method of problem solving Vs other methods of problem solving, optimality of the solution, knapsack problem and their applications

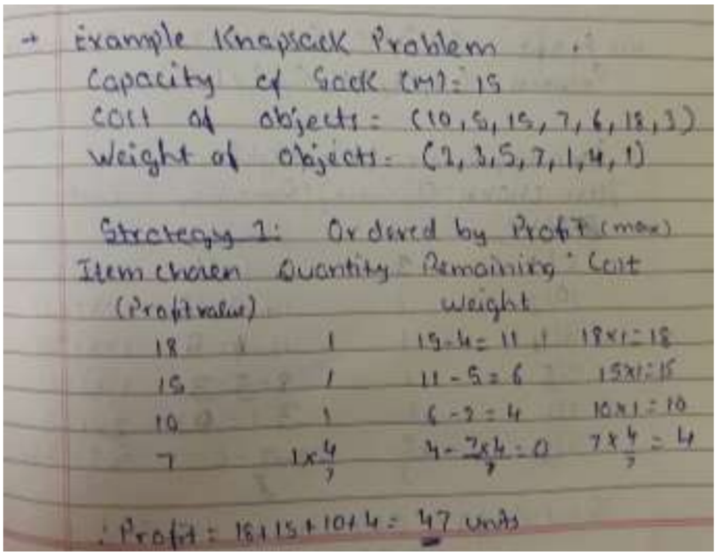
Knapsack Problem Algorithm



Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i]> p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
          for i := 1 to n do x[i]  := 0.0                    // initialize x
          U := m;
          for i := 1 to n do
          {
                    if (w(i) > U) then break;
                    x [i] := 1.0; U := U – w[i];
          }
          if (i ≤ n) then x[i] := U / w[i];
}
```

Example: Knapsack Problem



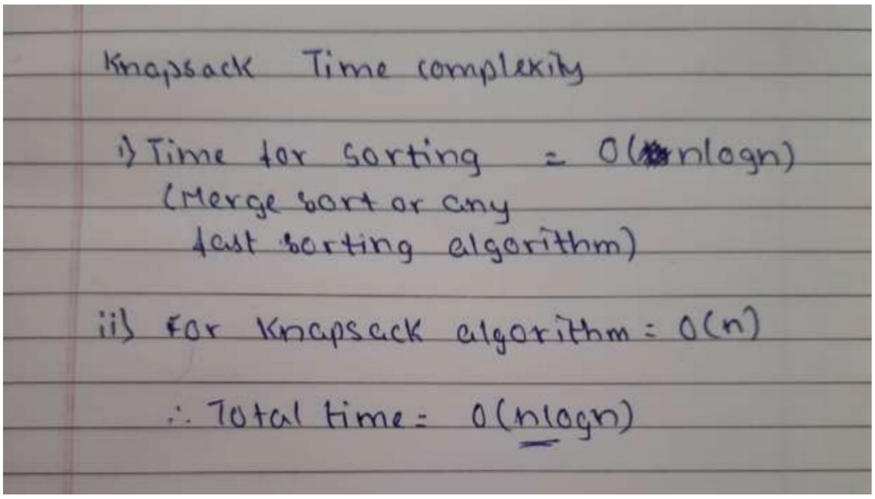
→ Example Knapsack Problem
Capacity of Sack (M) = 15
Cost of objects = (10, 5, 15, 7, 6, 18, 3)
Weight of objects = (2, 3, 5, 7, 1, 4, 1)

Strategy 1: Ordered by Profit (max)

Item chosen (Profit value)	Quantity	Remaining weight	Cost
18	1	$15-4=11$	$18\times1=18$
15	1	$11-5=6$	$15\times1=15$
10	1	$6-2=4$	$10\times1=10$
7	$1\times\frac{4}{7}$	$4-\frac{7\times4}{7}=0$	$7\times\frac{4}{7}=4$

∴ Profit = 18+15+10+4 = 47 units

Analysis of Knapsack Problem algorithm:

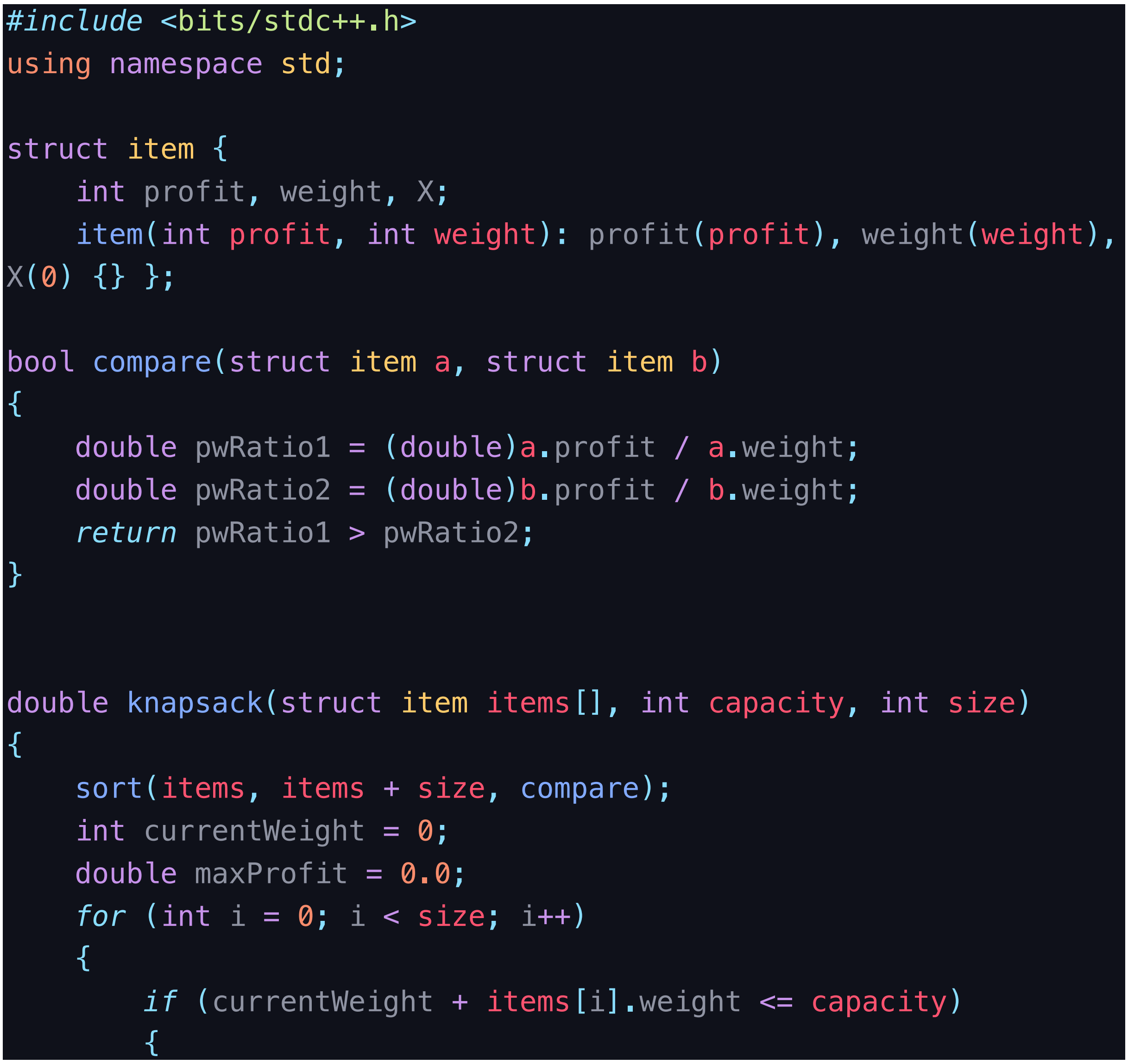


Knapsack Time complexity

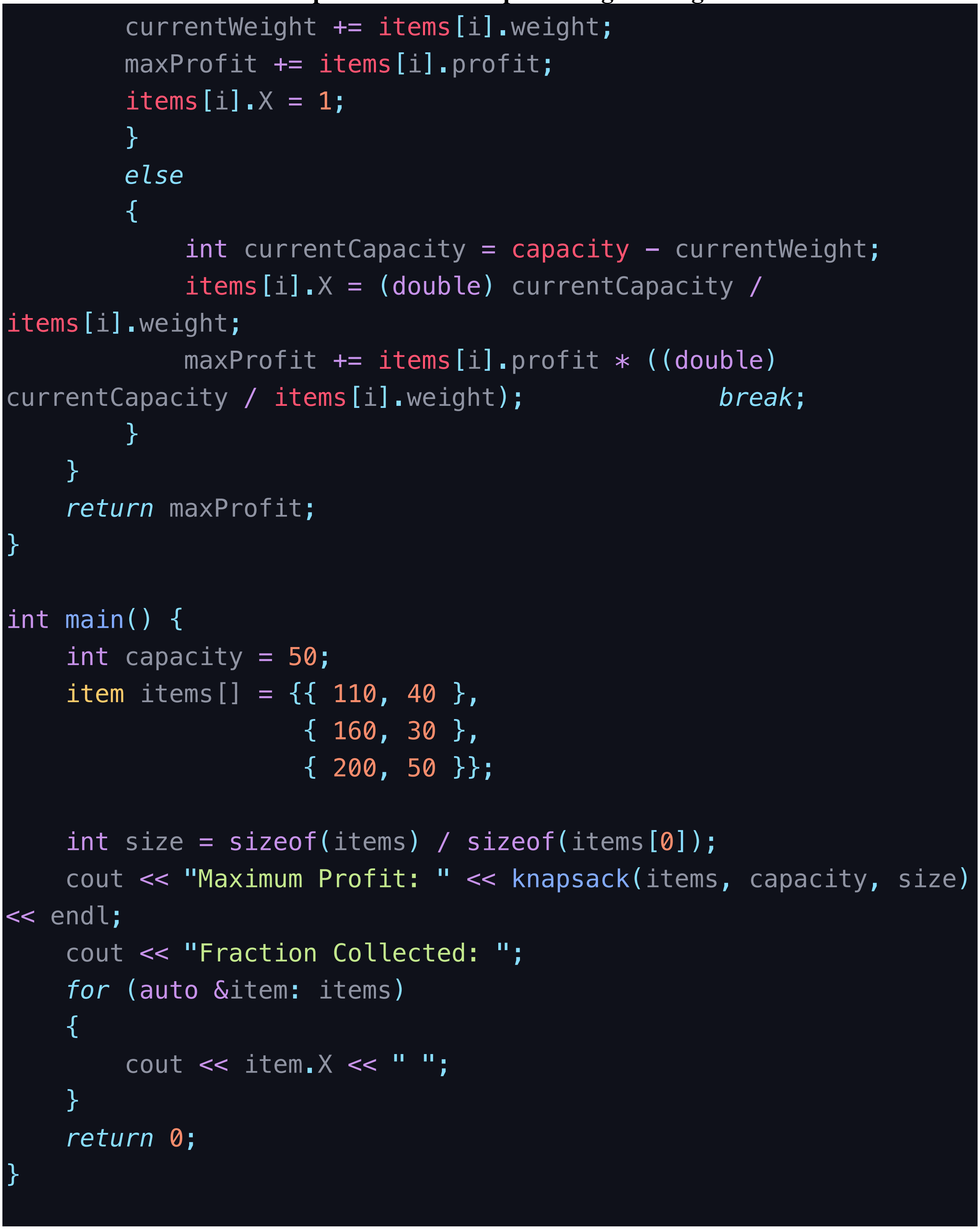
i) Time for sorting = O(nlogn)
(Merge sort or any fast sorting algorithm)

ii) For Knapsack algorithm : O(n)

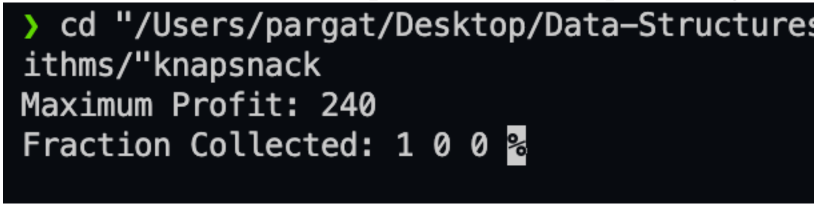
∴ Total time : O(nlogn)



```
#include <bits/stdc++.h>
using namespace std;

struct item {
    int profit, weight, X;
    item(int profit, int weight): profit(profit), weight(weight),
X(0) {} };

bool compare(struct item a, struct item b)
{
    double pwRatio1 = (double)a.profit / a.weight;
    double pwRatio2 = (double)b.profit / b.weight;
    return pwRatio1 > pwRatio2;
}


double knapsack(struct item items[], int capacity, int size)
{
    sort(items, items + size, compare);
    int currentWeight = 0;
    double maxProfit = 0.0;
    for (int i = 0; i < size; i++)
    {
        if (currentWeight + items[i].weight <= capacity)
        {
```



```
        currentWeight += items[i].weight;
        maxProfit += items[i].profit;
        items[i].X = 1;
        }
        else
        {
            int currentCapacity = capacity - currentWeight;
            items[i].X = (double) currentCapacity /
items[i].weight;
            maxProfit += items[i].profit * ((double)
currentCapacity / items[i].weight);             break;
        }
    }
    return maxProfit;
}

int main() {
    int capacity = 50;
    item items[] = {{ 110, 40 },
                    { 160, 30 },
                    { 200, 50 }};

    int size = sizeof(items) / sizeof(items[0]);
    cout << "Maximum Profit: " << knapsack(items, capacity, size)
<< endl;
    cout << "Fraction Collected: ";
    for (auto &item: items)
    {
        cout << item.X << " ";
    }
    return 0;
}
```



```
❯ cd "/Users/pargat/Desktop/Data-Structure
ithms/"knapsnack
Maximum Profit: 240
Fraction Collected: 1 0 0 %
```

Conclusion:

Learnt the greedy approach for solving the knapsack problem, implemented all the strategies for solving the problem and analysed the time complexity of the algorithm.