4.2 KNAPSACK PROBLEM

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \tag{4.2}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

Lemma 4.1 In case the sum of all the weights is $\le m$, then $x_i = 1$, $1 \le i \le n$ is an optimal solution. □

So let us assume the sum of weights exceeds m. Now all the x_i's cannot be 1. Another observation to make is:

Lemma 4.2 All optimal solutions will fill the knapsack exactly. □

Lemma 4.2 is true because we can always increase the contribution of some object i by a fractional amount until the total weight is exactly m.

Note that the knapsack problem calls for selecting a subset of the objects and hence fits the subset paradigm. In addition to selecting a subset, the knapsack problem also involves the selection of an x_i for each object. Several simple greedy strategies to obtain feasible solutions whose sums are identically m suggest themselves. First, we can try to fill the knapsack by including next the object with largest profit. If an object under consideration doesn't fit, then a fraction of it is included to fill the knapsack. Thus each time an object is included (except possibly when the last object is included) into the knapsack, we obtain the largest possible increase in profit value. Note that if only a fraction of the last object is included, then it may be possible to get a bigger increase by using a different object. For example, if we have two units of space left and two objects with ($p_i = 4$, $w_i = 4$) and ($p_j = 3, w_j = 2$) remaining, then using j is better than using half of i. Let us use this selection strategy on the data of Example 4.1.

Object one has the largest profit value ($p_1 = 25$). So it is placed into the knapsack first. Then $x_1 = 1$ and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Object two has the next largest profit ($p_2 = 24$). However, $w_2 = 15$ and it doesn't fit into the knapsack. Using $x_2 = 2/15$ fills the knapsack exactly with part of object 2 and the value of the resulting solution is 28.2. This is solution 2 and it is readily seen to be suboptimal. The method used to obtain this solution is termed a greedy method because at each step (except possibly the last one) we chose to introduce that object which would increase the objective function value the most. However, this greedy method did not yield an optimal solution. Note that even if we change the above strategy so that in the last step the objective function increases by as much as possible, an optimal solution is not obtained for Example 4.1.

We can formulate at least two other greedy approaches attempting to obtain optimal solutions. From the preceding example, we note that considering objects in order of nonincreasing profit values does not yield an optimal solution because even though the objective function value takes on large increases at each step, the number of steps is few as the knapsack capacity is used up at a rapid rate. So, let us try to be greedy with capacity and use it up as slowly as possible. This requires us to consider the objects in order of nondecreasing weights w_i. Using Example 4.1, solution 3 results. This too is suboptimal. This time, even though capacity is used slowly, profits aren't coming in rapidly enough.

Thus, our next attempt is an algorithm that strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit of capacity used. This means that objects are considered in order of the ratio p_i/w_i. Solution 4 of Example 4.1 is produced by this strategy. If the objects have already been sorted into nonincreasing order of p_i/w_i, then function GreedyKnapsack (Algorithm 4.2) obtains solutions corresponding to this strategy. Note that solutions corresponding to the first two strategies can be obtained using this algorithm if the objects are initially in the appropriate order. Disregarding the time to initially sort the objects, each of the three strategies outlined above requires only $O(n)$ time.

We have seen that when one applies the greedy method to the solution of the knapsack problem, there are at least three different measures one can attempt to optimize when determining which object to include next. These measures are total profit, capacity used, and the ratio of accumulated profit to capacity used. Once an optimization measure has been chosen, the greedy

```
Algorithm GreedyKnapsack(m, n)
// p[1 : n] and w[1 : n] contain the profits and weights respectively
// of the n objects ordered such 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];
}
```

4.4 JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example 4.2 Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and (d_1, d_2, d_3, d_4) $(2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

To formulate a greedy algorithm to obtain an optimal solution, we must formulate an optimization measure to determine how the next job is chosen. As a first attempt we can choose the objective function $\sum_{i \in J} p_i$ as our optimization measure. Using this measure, the next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires us to consider jobs in nonincreasing order of the p_i's. Let us apply this criterion to the data of Example 4.2. We begin with $J = \emptyset$ and $\sum_{i \in J} p_i = 0$. Job 1 is added to J as it has the largest profit and $J = \{1\}$ is a feasible solution. Next, job 4 is considered. The solution $J = \{1, 4\}$ is also feasible. Next, job 3 is considered and discarded as $J = \{1, 3, 4\}$ is not feasible. Finally, job 2 is considered for inclusion into J. It is discarded as $J = \{1, 2, 4\}$ is not feasible. Hence, we are left with the solution $J = \{1, 4\}$ with value 127. This is the optimal solution for the given problem instance. Theorem 4.4 proves that the greedy algorithm just described always obtains an optimal solution to this sequencing problem.

Before attempting the proof, let us see how we can determine whether a given J is a feasible solution. One obvious way is to try out all possible permutations of the jobs in J and check whether the jobs in J can be processed in any one of these permutations (sequences) without violating the deadlines. For a given permutation $\sigma = i_1, i_2, i_3, \ldots, i_k$, this is easy to do, since the earliest time job $i_q, 1 \le q \le k$, will be completed is q. If $q > d_{i_q}$, then using σ, at least job i_q will not be completed by its deadline. However, if $|J| = i$, this requires checking $i!$ permutations. Actually, the feasibility of a set J can be determined by checking only one permutation of the jobs in J. This permutation is any one of the permutations in which jobs are ordered in nondecreasing order of deadlines.

Theorem 4.3 Let J be a set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ a permutation of jobs in J such that $d_{i_1} \le d_{i_2} \le \cdots \le d_{i_k}$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline.

Proof: Clearly, if the jobs in J can be processed in the order σ without violating any deadline, then J is a feasible solution. So, we have only to show that if J is feasible, then σ represents a possible order in which the jobs can be processed. If J is feasible, then there exists $\sigma' = r_1, r_2, \ldots, r_k$ such that $d_{r_q} \ge q$, $1 \le q \le k$. Assume $\sigma' \ne \sigma$. Then let a be the least index such that $r_a \ne i_a$. Let $r_b = i_a$. Clearly, $b > a$. In σ' we can interchange r_a and r_b. Since $d_{r_a} \ge d_{r_b}$, the resulting permutation $\sigma'' = s_1, s_2, \ldots, s_k$ represents an order in which the jobs can be processed without violating a deadline. Continuing in this way, σ' can be transformed into σ without violating any deadline. Hence, the theorem is proved. □

Theorem 4.3 is true even if the jobs have different processing times $t_i \ge 0$ (see the exercises).

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1) step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```