GREEDY ALGORITHMS

Greedy algorithms solve problems by making the choice that seems best at the particular moment. Many optimization problems can be solved using a greedy algorithm. Some problems have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal.

A greedy algorithm works if a problem exhibits the following **two properties**:

1. **Greedy choice property**: A globally optimal solution can be arrived at by making a locally optimal solution. In other words, an optimal solution can be obtained by making "greedy" choices.

2. **optimal substructure**. Optimal solutions contain optimal sub solutions. In other Words, solutions to sub problems of an optimal solution are optimal.

Difference Between Greedy and Dynamic Programming

- The most difference between greedy algorithms and dynamic programming is that we don't solve every optimal sub-problem with greedy algorithms. In some cases, greedy algorithms can be used to produce sub-optimal solutions. That is, solutions which aren't necessarily optimal, but are perhaps very dose.
- In dynamic programming, we make a choice at each step, but the choice may depend on the solutions to subproblems.
- In a greedy algorithm, we make whatever choice seems best at 'the moment and then solve the sub-problems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems.
- Thus, unlike dynamic programming, which solves the sub-problems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, interactively reducing each given problem instance to a smaller one.

22.3 KNAPSACK PROBLEMS

We want to pack n items in your luggage

- The ith item is worth v_i dollars and weighs w_i pounds
- Take as valuable a load as possible, but cannot exceed W pounds
- v_i, w_i, W are integers

0-1 Knapsack Problem

- each item is taken or not taken.
- cannot take a fractional amount of an item or take an item more than once

Fractional Knapsack Problem

- fractions of items can be taken rather than having to make a binary (0-1) choice for each item

Both exhibit the optimal-substructure property

0-1 knapsack problem. Consider a optimal solution. If item j is removed from the load, the remaining load must be the most valuable load weighing at most $W - w_j$.

Fractional knapsack problem. If w of item j is removed from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from other $n-1$ items plus $w_j - w$ of item j.

The ***0-1 knapsack problem*** is posed as follows.

A thief robbing a store finds n items; the i^{th} item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once.

In the ***fractional knapsack problem***,

the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. **Fractional** knapsack problem can be solvable by the greedy strategy whereas the ***0-1 knapsack*** problem is not.

To solve the fractional problem:

- Compute the value per pound v_i / w_i for each item
- Obeying a greedy strategy, we take as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and we can still carry more, we take as much as possible of the item with the next value per pound, and so forth until we cannot tarn any more.
- Sorting the items by value per pound, the greedy algorithm runs in 0(n lgn) time .

0-1 knapsack problem cannot be solved by the greedy strategy because

- it is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load
- we must compare the solution to the sub-problem in which
  - the item is included with the solution to the sub-problem
  - the item is excluded

before we can make the choice.

```
Fractional Knapsack (Array v, Array w, int W)
1.  for i = 1 to Size(v)
2.      do p[i] = v[i] / w[i]
3.  Sort-Descending(p)
4.  i ← 1
5.  while (W > 0)
6.      do amount = min(W, w[i])
7.      solution[i] = amount
8.      W = W-amount
9.      i ← i + 1
10. return solution
```

Example:

Consider 5 items along their respective weights and values

$$I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$$
$$w = \langle 5, 10, 20, 30, 40 \rangle$$
$$v = \langle 30, 20, 100, 90, 160 \rangle$$

e capacity of knapsack $W = 60$. *Find the solution to the fractional knapsack problem.*

Solution. Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio *i.e.*, $p_i = v_i / w_i$

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

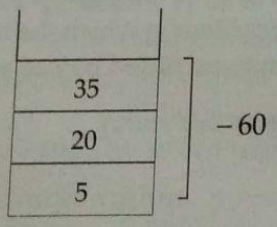
Now, arrange the value of p_i in decreasing order.

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of p_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is $5+20=25$.

Now, the next item is I_5 and its weight is 40, but we want only 35. So we choose fractional part of if *i.e.*,



The value of fractional part of I_5 is

$$\frac{160}{40} \times 35 = 140$$

Thus the maximum value is

$$= 30 + 100 + 140 = 270$$