Greedy Algorithms

Greedy algorithms apply to problems that exhibit the following properties :

- greedy choice property,
- optimal substructure.

We have seen that optimal substructure means that optimal solutions are composed of optimal subsolutions.

Sometime there are too many subsolutions to check.

The ”greedy choice property” means that an optimal solution can be obtained by making the “greedy” choice at every step.

Don’t need to know the solutions of all the subproblems in order to make a choice, always make the choice that looks best at the moment.

Greedy Algorithms

Greedy algorithms are usually very simple. In most common situation, we distinguishes the following of an greedy algorithm :

- A candidate set (e.g. the nodes of a graph)
- A set of chosen candidates that have been selected to be part of the solution
- A selection function (greedy criterion) that indicates at any time which is the most promising of the candidates not yet used
- A function that checks whether a particular set of candidates *is a solution* to our problem
- A function that checks whether a set of candidates is *feasible*
- A function to optimize, the objective function (greedy algorithms are applied to optimization problems).

Example : the 0-1 knapsack problem

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

In this example, we have a set of 4 objects each with weight $w[i]$ and value $v[i]$, knapsack capacity $W = 5$ and decision variable x_i, $x_i = 1$ if object i is in the solution, otherwise $x_i = 0$.

To solve the knapsack problem one has to select a subset of objects such to $\max \sum_{i=1}^{4} x_i v[i]$ while $\sum_{i=1}^{4} x_i w[i] \leq W$

The way dynamic programming solves this problem is not very intuitive, it finds the optimal subset of objects for each capacity value

Greedy on the other hand select one object, put it in the knapsack and then select another one, until the knapsack is full

Example : the 0-1 knapsack problem

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

- The set of 4 objects is the initial candidate set
- The selection function (greedy criterion) could be the object in the candidate set that has the largest value (the most promising candidate)
- Adding object i such that $\sum_{i=1}^{4} x_i w[i] > W$ could be the function that checks whether a particular set of candidates *provides a solution*
- $\sum_{i=1}^{4} x_i w[i] \leq W$ is the function that checks whether a set of candidates is *feasible*
- $\max \sum_{i=1}^{4} x_i v[i]$ is the *objective function*

Steps of a greedy Algorithm

- Initially, the set of chosen candidates is empty
- At each step, try to add to this set the best remaining candidate according to greedy criterion
- If the increased set of chosen candidates not feasible, remove the candidate just added, it is never considered again. Otherwise, candidate just added stays in the set of chosen candidates
- Each time the set of chosen candidates is increased, check whether the set now constitutes a solution to the problem
- When a greedy algorithm works correctly, the first solution found in this way is always optimal !

Note, greedy algorithms do not work for the 0-1 knapsack, it provides a feasible solution but not necessary optimal.

The making change problem : a greedy algorithm

Design an algorithm for paying back an customer a certain amount using the smallest possible number of coins.

For example, the smallest amount of coins to pay back \$2.89 (289 cents) is 10 : 2 one dollars, 3 quarters, 1 dime and 4 pennies.

Greedy algorithm :

- Choose the largest coin that is no larger than n (here n is the number of cents). Suppose that coin is the c cent coin.
- Give out $\lfloor n/c \rfloor$ of the c cent coins.
- Make change recursively for $n - \lfloor n/c \rfloor c$ cents.

Example of making change

Assume the coin denominations are 25, 10, 5 and 1 cents and we make change for 97 cents using the above greedy algorithm.

- Choose the largest coin that is no larger than n (here 25 cents is the largest coin no larger than 97 cents)
- Give out $\lfloor n/c \rfloor$ of the c cent coins ($\lfloor 97/25 \rfloor$ = three quarters)
- Make change recursively for $n - \lfloor n/c \rfloor c$ cents ($97 - 3 \times 25 = 22$ cents)
  - 10 cents is the largest coin no larger than 22 cents, so we hand out two dimes and make change for $22 - 20 = 2$ cents
    - a penny is the largest coin no larger than 2 cents, so we hand out two pennies and we're done !

Altogether we hand out 3 quarters, 2 dimes and 2 pennies, for a total of 7 coins

Greedy characteristics of the making change problem

The elements of the problem are :

- Candidate set : a finite set of coins, representing for instance 1, 5, 10, 25 units, and containing enough coins of each type
- Solution : the total value of the chosen set of coins is exactly the amount we have to pay back
- Feasible set : the total value of the chosen set *does not exceed* the amount to be returned
- Greedy criterion : chose the highest-valued coin remaining in the set of candidates ; and
- Objective function : minimize the number of coins used in the solution.

Does greedy algorithms always work ?

Depends. The greedy algorithm always gives out the fewest number of coins in change, when the coins are quarters, dimes, nickels and pennies.

It doesn't give out the fewest number of coins if the coins are 15 cents, 7 cents, 2 cents and 1 cent.

Example : For 21 cents the greedy algorithm would

- give out one 15-cent coin, remainder $21 - 15 = 6$ and
- give out three 2-cent coins,
- for a total of four coins.

Better : give out three 7-cent coins for a total of three coins !

Greedy problems

We can find counter examples for 0-1 knapsack and making change problems where greedy algorithms fail to find an optimal solution

However, both problems exhibit the optimal substructure, the optimal solution to a problem instance contains in it the optimal solution of subproblems

The difference is in the way greedy algorithms solve problems compared to dynamic programming algorithms

Greedy problems

Typically dynamic programming algorithms solve problems bottom-up

- finding the optimal solution to the smallest instances
- then building optimal solutions to largest instances from the optimal solutions of the smallest.

Greedy algorithms solve in top-down fashion making the best choice at the moment and then solving recursively the subproblem that remains

A problem that can be solved in this way is said to exhibit the greedy property

To use a greedy algorithm one must prove that the problem exhibit the greedy property, the prove is often by induction