CS2223: Algorithms
Greedy Algorithms

1 Greedy Algorithm Fundamentals

Remember that Dynamic programming requires the problem to have optimum substructure property. In that case, it computes the solution for various sub-problems and finds which subproblem will give the overall optimum solution. In other words, dynamic programming works bottom-up.

Greedy algorithms can be used for problems that exhibit the optimal substructure property, and also exhibit the **greedy choice property**. Here, given a problem, you can determine what will be part of the optimum solution, and thus solve only one subproblem. In other words, greedy works in a top-down manner.

As we may expect, greedy algorithms are indeed more efficient (typically) than dynamic programming as we need to solve only one subproblem. Of course, greedy-choice property might not hold for all problems (it does not hold for 0/1 Knapsack problem, edit distance problem).

2 Activity-selection Problem

Consider the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., a_i and a_j are compatible if $s_i \geq f_j$ or $s_j ? f_i$). The activity-selection problem is to select a maximum-size subset of mutually compatible activities.

This problem has the **optimal substructure** property. Suppose we define S_{ij} as the set of activities that start after item i's finish time, and finish before item j's start time. The overall optimal solution is given by optimal solution among the activities in S_{kl}, where a_k has the earliest start time among all activities, and a_l has the latest finish time among all activities.

Suppose activity $a_p \in S_{ij}$ is in optimal solution for S_{ij}. Then the optimal solution for S_{ij} includes optimal solution for S_{ip}, optimal solution for S_{pj} and activity a_p.

But the problem also exhibits the **greedy-choice property**. We can make a choice as to what activity must be in the optimum solution and thus solve only one subproblem.

Suppose activity a_m is the activity with the earliest finish time among activities in S_{ij}. There exists an optimum solution which includes a_m.

Proof: Order the activites in the optimum solution for S_{ij} in the increasing order of their finish times. Suppose the first activity is $a_h \neq a_m$. Replace a_h with a_m, and this is another valid solution that is also optimum. (Because a_h's finish time must be greater than the finish time of a_m, replacing a_h with a_m will not impact other activities in the optimum solution).

Therefore we can get the algorithm for activity-selection problem as follows.

1. Sort the activities in the non-decreasing order of finish times.

2. Scan the above sorted list once, the current item is in the optimum solution if it is compatible with the other activities already chosen.

Overall complexity if $O(n\ log\ n)$.

3 Fractional Knapsack Problem

This problem is similar to the 0/1 knapsack problem discussed earlier, the only difference is that the thief can now take fractions of items as well.

The problem definition is: A thief robbing a store finds n items; the ith 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. He can take any fraction of a given item between $[0, 1]$. Which items should he take and in what quantities?

This problem exhibits the greedy-choice property. Suppose we sort the items in the non-decreasing order of the ratio v_i/w_i. There exists an optimum solution that includes as much as possible of the first item in the above ordered list.

Proof: Suppose item x has the largest value/weight ratio, and the optimum solution does not include as much as possible of item x and took only f_x (fraction). Suppose also, we have taken some amount of item y, say f_y (fraction). Consider $w = \min \{(1 - f_x) * w_x, f_y * w_y\}$ (in other words, w is the minimum of the weight of x not taken, or the weight of y taken. If we take $f_y - w/w_y$ fraction of y and $f_x + w/w_x$ fraction of x, we get a solution at least as good as the optimum solution.

We can therefore do the above till we have taken as much as possible of item x.

The above greedy-choice property yields the following algorithm. Sort the items in the non-increasing order of value/weight. Traverse this sorted list, taking as much as possible of each item.

Complexity is $O(n\ log\ n)$ for sorting and $O(n)$ for the linear search. Therefore $O(n\ log\ n)$ is the overall complexity.

4 Huffman codes

Suppose you want to transmit a string of characters where the alphabet is from a string. The problem is what binary codes to allot to each character so that the message size is minimized.

For example, consider the alphabet to consist of four characters, say a, b, c, d, and your string is composed of these characters – say a string could be aaabccbbbddaabb

A straight forward solution is for 4 characters, use 2 bits to represent each character. Say a - 00, b - 01, c - 10, d - 11. So for a 100,000 long string, we need 200,000 bits.

However, suppose the frequency of appearance of these characters in the string vary – say a appears 60%, b appears 10%, c appears 25% and d appears 5%. Consider a code: a - 1, b - 001, c - 01, d - 000; number of bits needed $= 60000 + 30000 + 50000 + 15000 =$ 155,000 bits.

Let us come up with an algorithm to find out these codes, all frequencies are > 0. We will consider only **prefix-free codes** – no code is a prefix of another code. This helps in easier decoding.

We can depict the prefix codes as a binary tree.

Property: The binary tree must be full – in other words, every non-leaf node will have exactly 2 children.

Proof: Suppose the binary tree is not full, and a non-leaf node has only one child. Consider this non-leaf node. We can decrease the code-length for all the descendants of this node by removing the "edge" between this node and its only child.

Greedy-Choice Property: Consider two characters (say x and y) with the lowest frequency. There exists an optimum solution where x and y are at the maximum depth, and are "siblings" of the same non-leaf node.

Proof: Consider an optimal solution where a and b are siblings of the node and have the maximum depth. Without loss of generality let $f_x \leq f_a$ (f_x denotes frequency of appearance of x), and $f_y \leq f_b$. Swap a with x and b with y and we get another optimum solution.

This gives the algorithm as follows – Consider two characters with the lowest frequency. Construct a tree of height 1 with these 2 characters as leaves. Now replace the two characters with a "character block" whose frequency is the sum of these two frequencies. Repeat the same technique till we have one "character block".

We can use a min-heap data structure for the above. In which case, we can obtain the Huffman codes in $O(n\ log\ n)$.