Greedy Algorithms

1. Some optimization question.
2. At each step, use a greedy heuristic.
3. Chapter 7 of the book

Coin Changing Problem

1. Some coin denominations say, $1, 5, 10, 20, 50$
2. Want to make change for amount S using smallest number of coins.

Example: Want change for 37 cents.
Optimal way is: $1 \times 20, 1 \times 10, 1 \times 5, 2 \times 1$.

In some cases, there may be more than one optimal solution:
Coins: 1, 5, 10, 20, 25, 50,
$S = 30$
Then, $1 \times 20, 1 \times 10$ and $1 \times 25, 1 \times 5$ are both optimal solutions.

Greedy Solution:

Repeat until $S = 0$.

1. Find the largest coin denomination (say x), which is $\leq S$.
2. Use $\lfloor \frac{S}{x} \rfloor$ coins of denomination x.
3. Let $S = S \bmod x$.

Input: Coin denominations $d[1] > d[2] > \ldots > d[m] = 1$.

Input: Value S

Output: Number of coins used for getting a total of S.

num$[i]$ denotes the number of coins of denomination $d[i]$.

For $i = 1$ to m do $\{$

$\quad$ num$[i] = \lfloor \frac{S}{d[i]} \rfloor$.

$\quad S = S \bmod d[i]$.

$\}$

1. Each step reduces the problem to a smaller problem.

2. Each step is Greedy: tries local optimization.

3. Short-sighted.

4. Easy to construct, simple to implement

5. May or may not give optimal (best) solution.

For coin denominations $1, 5, 10$, the algorithm does give optimal solution.

Proof: Consider any optimal solution. That solution cannot have
– more than four 1's (as five 1's could be replaced by one 5), or
– more than one 5 (as two 5's could be replaced by one 10).

Thus, number of 10s in both greedy algorithm and optimal algorithm must be same, and the value of rest of the coins adds up to at most 9.

So we only need to prove for the cases when amount ≤ 9.

Any amount < 5 can be covered only using 1's and both optimal and greedy algorithm use same number of 1s.

If the amount is between 5 and 9 (both inclusive), then both optimal and greedy solution have exactly one 5 and rest of the value is covered using 1s.

It follows that the greedy algorithm has the same coins as the optimal solution.

For coin denominations $1, 5, 10, 20, 25$, it need not give optimal solution.
Counterexample: S=40.
Greedy algorithm gives, 1×25, 1×10, 1×5.
Optimal is 2×20.

Fractional Knapsack

– A knapsack of certain capacity (S Kg)

– Various items available to carry in the knapsack.

– There are w_i Kg of item i worth total of c_i.

– You may pick fraction of some item i (that is, you don't have to pick all w_i Kg of item i). This will give prorated value for the item picked.

Example:
item 1: weight = 5 Kg, value = \$30

item 2: weight = 3 Kg, value = \$20

item 3: weight = 3 Kg, value = \$10

Capacity = 7 Kg

Pick: item 2, and 4 Kg of item 1.

Total value: \$20+\$30*(4/5)=\$44

Greedy algorithm:

Suppose the items are $1, 2, \ldots, n$.

Weight of item i is w_i and value is c_i.

Rename the items such that they are sorted in non-increasing order of c_i/w_i.

Thus, we assume below that $c_1/w_1 \geq c_2/w_2 \geq \ldots$.

$CapLeft = S$.

$TotalValuePicked = 0$.

For $i = 1$ to n do {

2. Pick $min(CapLeft, w_i)$ of item i.
3. $TotalValuePicked = TotalValuePicked + (c_i/w_i) * min(CapLeft, w_i)$.
4. $CapLeft = CapLeft - min(CapLeft, w_i)$.

}

Optimality: Suppose the optimal took o_i amount of item i, and the algorithm took a_i amout of item i.
For optimal algorithm to be better, there must be least i such that $o_i > a_i$.
Then, at the time the above algorithm was considering item i, it had capacity left $< o_i \leq w_i$.
Thus, Algorithm's value - optimal value is

$$\begin{aligned}
\sum_{j=1}^{n}(a_j - o_j) * \frac{c_j}{w_j} &= \\
\sum_{j=1}^{i-1}(a_j - o_j) * \frac{c_j}{w_j} + (a_i - o_i) * \frac{c_i}{w_i} + \sum_{j=i+1}^{n}(a_j - o_j)\frac{c_j}{w_j} &\geq \\
\sum_{j=1}^{i-1}(a_j - o_j) * \frac{c_i}{w_i} + (a_i - o_i) * \frac{c_i}{w_i} + \sum_{j=i+1}^{n}(a_j - o_j)\frac{c_i}{w_i} &\geq \\
\frac{c_i}{w_i}\sum_{j=1}^{n}(a_j - o_j) &\geq \\
\frac{c_i}{w_i}[\sum_{j=1}^{n} a_j - \sum_{j=1}^{n} o_j] &\geq 0
\end{aligned}$$

Complexity:
$O(n \log n)$ for sorting.
$O(n)$ for the main part.

Data Compression

– Very important to save space. MP3, mpeg, ...
– Example: We want to code the string $AAABBBCDAA$,

If we code using 2 bits each ($A = 00, B = 01, C = 10, D = 11$), then it takes 20 bits.

(Code=00000001010110110000)

If we code $A = 0$, $B = 10$, $D = 110$ and $C = 111$, then we take total of $1 * 5 + 2 * 3 + 1 * 3 + 1 * 3$ bits, that is 17 bits

(Code=00010101011111000)

– What happens if we use $A = 0, B = 1, C = 00, D = 10$?

– Code 00 stands for both AA and C.

– Prefix Code: code for no character is a prefix of a code for another character.

Huffman Code

One can represent any prefix code as a Huffman coding tree (or Huffman tree).

– Leaves are the characters which are being coded.

– The left branch represents 0, and right branch represents 1.

– Path from the root to the leaf represents the code for the leaf.

Can recover the string from the code:
– start from the root of the tree, and follow the code to find the first character coded.
– repeat until code is finished.

Huffman Code: Finding an optimal code

Let f_i denote the frequency of the character i.
The aim is to place the characters with least frequency at the bottom of the tree.

Repeat until only one object is left.

(a) Find two least frequent objects x and y.

(b) Delete these two objects x and y.
Insert a new object (called xy) with frequency $f_{xy} = f_x + f_y$.
(* Intuitively, in the construction of the Huffman tree, object xy would have two children x and y, which we collectively think of as one object from now on. *)

– There are n iterations of the loop.

– Each loop iteration may take $O(n)$ time, if not implemented properly.

– Use binary minheap. Finding minimal element, deleting and insertion then takes $O(\log n)$ time. Thus total time taken will be $O(n \log n)$.

– See textbook if you are not familiar with binary minheap.

How good is our algorithm?

Measure of goodness:
If $m_1, m_2, \ldots, m_k$ are the bits needed for the objects, and their frequencies are $f_1, f_2, \ldots, f_k$, then the goal is to minimize the cost:

$$\Sigma_{i=1}^{k} m_i f_i$$

A Huffman tree is optimal if it minimizes the cost.

NOTE: There may be several optimal trees. The method we used only finds one of them.

Optimality of our method

Lemma 1: Given a set of objects and their frequencies, there exists an optimal Huffman coding tree where the two smallest frequencies have the same parent.

Proof: Consider the optimal tree. If the smallest frequency object is not at the bottom of the tree, then swap it with the object at the bottom of the tree. This does not increase the cost.
Then, if the sibling of the smallest frequency object is not the 2nd smallest frequency object, then again swap the sibling with the second smallest frequency object. This again does not increase the cost.
QED

Induction Step

Lemma 2: Suppose T is an optimal Huffman tree for frequency counts $f_1, f_2, \ldots, f_k$ (in non-decreasing order, that is $f_1 \leq f_2 \leq \ldots \leq f_k$) and suppose that f_1 and f_2 have same parent in T.

Let tree T' be obtained by deleting the leaves corresponding to f_1 and f_2 in T and using frequency $f_1 + f_2$ for the parent of f_1, f_2 in T.

Then T' is optimal for frequency counts $f_1 + f_2$, $f_3, f_4, \ldots$.

Proof: Suppose otherwise. Let S' be optimal for frequency count $f_1 + f_2$, $f_3, f_4, \ldots$. Then obtain S from S' by adding two children with frequency f_1 and f_2 to the node with frequency $f_1 + f_2$ in S'. Then, S has lower cost than T, a contradiction.

Theorem: The algorithm generates an optimal tree.

Proof: By induction. For $k = 1$, it is trivial.
For $k = n$, suppose the algorithm generates a tree T.
Also, there exists an optimal tree S where the two leaves with smallest frequencies (say f_1 and f_2) have the same parent (by Lemma 1).
Then, combine the two leaves in T with least frequency, to form a tree T'.
Similarly, combine the two leaves in S with least frequency, to form a tree is S'.
By induction T' is optimal. By Lemma 2, S' is optimal too.
Thus, $cost(T) = cost(T') + f_1 + f_2 = cost(S') + f_1 + f_2 = cost(S)$.
Thus, T is optimal.