Greedy Algorithms

1. Some optimization question.
2. At each step, use a greedy heuristic.

Coin Changing Problem

- Some coin denominations say, $1, 5, 10, 20, 50$
- 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.
- Make a locally optimal choice: choose the largest coin possible at each step.

Coin Changing Problem

Greedy Solution:

Repeat
    Find the largest coin denomination (say x), which is $\leq S$.
    Use $\lfloor \frac{S}{x} \rfloor$ coins of denomination x.
    Let $S = S$ mod x.
until $S = 0$.

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

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

$S = S$ mod $d[i]$.

EndFor

Greedy Strategy

The greedy algorithm makes a locally optimal solution. In some cases this is enough to get a globally optimal solution.

- The choice at each step must be feasible: satisfy the requirements.
- Each step reduces the problem to a smaller problem.
- Locally optimal: Each step is Greedy and tries local optimization, that is, chooses best among local choices based on some metric.
- Short-sighted.
- Easy to construct, simple to implement
- Irrevocable: once choice is made, there is no going back.
- 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.

Greedy Strategy

- Optimal Solutions:
  - Minimum Spanning Tree
  - Single Source Shortest Paths
  - Huffman codes
- Approximation Algorithms
  - Traveling Salesman Problem
  - Knapsack problem

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
- 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. $CapLeft = CapLeft - min(CapLeft, w_i)$.
EndFor

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

$$\sum_{j=1}^{n}(a_j - o_j) * \frac{c_j}{w_j} =$$

$$\sum_{j=1}^{i-1}(a_j - o_j) * \frac{c_j}{w_j} + (a_i - o_i) * \frac{c_i}{w_i} + \sum_{j=i+1}^{n}(a_j - o_j)\frac{c_j}{w_j} \geq$$

$$\sum_{j=1}^{i-1}(a_j - o_j) * \frac{c_i}{w_i} + (a_i - o_i) * \frac{c_i}{w_i} + \sum_{j=i+1}^{n}(a_j - o_j)\frac{c_i}{w_i} \geq$$

$$\frac{c_i}{w_i}\sum_{j=1}^{n}(a_j - o_j) \geq$$

$$\frac{c_i}{w_i}[\sum^{n} a_j - \sum^{n} o_j] \geq 0$$

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

Single Source Shortest Paths

- Dijkstra's Algorithm for Single Source Shortest Path
- Given a weighted graph, find the shortest path from a given vertex to all other vertices.
- Path (from v_0 to v_k): $v_0 v_1 \ldots v_k$, such that (v_i, v_{i+1}) is an edge in the graph, for $i < k$.
- Simple Path: In the path, if $i \neq j$ then $v_i \neq v_j$.
- Weight of the path: sum of weights of the edges in the path.
- Shortest path: path with minimal weight.

- Intuition behind algorithm:
- Assuming positive integral distances between nodes.
- Walk in “all possible directions” for 1 step. If we encounter a node, then the shortest distance to it must be 1.
  Then walk further for 1 step in all possible directions. If we encounter a node, not yet encountered, then the shortest distance to it must be 2.
- Continue, similarly until all nodes are encountered.

- Graph $G = (V, E)$, with $wt(u, v)$ giving the weight of edge (u, v)
- Graph edges are given as adjacency list.
- s is the source vertex.
- $Dist(u)$ denotes the currently best known distance from s to u.
- Rem denotes the nodes for which the shortest distance is not yet found.
- $V - Rem$ will thus denote the vertices for which shortest distance is already found.
- $prev(v)$ denotes the predecessor node of v in the currently known shortest path from s to v.

Dijkstra's Algorithm

Set $Dist[u] = \infty$, and $Prev[u] = null$ for all $u \in V$.
Set $Dist[s] = 0$.
Set $Rem = V$.
While $Rem \neq \emptyset$ Do
1. Find node $u \in Rem$ with minimal $Dist[u]$.
2. $Rem = Rem - \{u\}$.
3. For each edge (u, v) such that $v \in Rem - \{u\}$ Do
   3.1 Let $Z = min(Dist[v], Dist[u] + wt(u, v))$.
   3.2 If $Dist[v] > Z$, then let $Dist[v] = Z$ and $Prev(v) = u$
   Endif
   EndFor
End While

Correctness:
Let $Short(v)$ denote the shortest distance from s to v.
By induction we will show that in every iteration, when a node u is removed from Rem, then

(a) $Dist(u) = Short(u)$, and $Dist(u)$ is never updated again (note that this implies: for all $w \in V - Rem$, $Dist(w) = Short(w)$).
(b) $Dist(v)$, at the beginning of any iteration, is the shortest distance from s to v, using only the nodes in $V - Rem$ in the path (except for the "last" node v itself).
(c) For all the nodes w in Rem, and w' in $V - Rem$, $Short(w) \geq Short(w')$.

At the beginning the invariants clearly hold.

Now consider any iteration of the loop.
Suppose u is deleted from Rem in this iteration.

Then, we first claim that for all $w \in Rem - \{u\}$, at this iteration, $Short(w) \geq Dist(u)$.
Suppose otherwise, that is $Short(w) < Dist(u)$ for some w in Rem. Choose such a w which minimizes $Short(w)$.
Then, all the nodes in the shortest path from s to w (except for w) are already in $V - Rem$ (as w was the closest to s which is in Rem). Thus, by inductive hypothesis (b), we have $Dist(w) = Short(w) < Dist(u)$, a contradiction to the choice of u.
This implies that invariants (a) and (c) hold.

For (b) note that, for $v \in Rem$, either the shortest path from s to v has last node (just before v) u or not. In either case, steps 3.1 and 3.2 of the algorithm ensure that invariant (b) is maintained.

Complexity:
While Loop is executed n times.

Simple implementation:
Rem as a set: $Rem[u]$ =true, if $u \in Rem$.

Overall time for
Step 1: $n * O(n)$
Step 2: $n * O(1)$
Step 3: $O(m)$
Total: $O(n^2 + m)$.

If one uses binary minheap for weights:
Step 1: $O(n)$
Step 2: $O(n \log n)$
Step 3: $O(m \log n)$
Total: $O(m \log n + n \log n)$

Can be reduced further to $O(n \log n + m)$ using Fibonacci heaps.

Prim's Algorithm for MST

- Spanning tree of a graph $G = (V, E)$ is a tree $T = (V, E')$ such that $E' \subseteq E$.
  (Note: G needs to be a connected graph)
- Minimal spanning tree of a weighted graph is a spanning tree of the graph with minimal weight.
- Note that there may be several minimal spanning trees.

Intuition:

- Start with one vertex (say start) to be in the spanning tree.
- Add an edge with minimum weight that has one endpoint in the tree already constructed, and the other endpoint outside the tree constructed. (Greedy!)
- Repeat until all vertices are in the tree

- Suppose Rem denotes the set of vertices not yet in the spanning tree constructed.
- We will maintain an array $D(v)$ and $parent(v)$:
- For v already in the tree, $parent(v)$ gives the node to which it is connected in the tree (at the time it was added to the tree). $parent(start) = null$.
- For v in Rem, $D(v)$ gives the shortest distance from v to the tree already constructed; $parent(v)$ gives the node in tree to which this shortest distance applies. Note that $D(v)$ may be ∞ in case there is no edge from v to the nodes already in constructed tree. In this case $parent(v) = null$

- In each iteration, we will choose the vertex u in Rem which minimizes $D(u)$. We will add u to the tree. Furthermore, we will update $D(v)$, for v in $Rem - \{u\}$, in case, $wt(u, v) < D(v)$.

Prim's algorithm

1. Pick any node in V as $start$.
2. Let $D(v) = \infty$ for all $v \in V - \{start\}$
3. Let $D(start) = 0$.
2. Let $parent(v) = null$ for all $v \in V$.
4. Let $Rem = V$.
5. While Rem is not empty Do
   Choose a node u in Rem which minimizes $D(u)$.
   Delete u from Rem.
   For each edge (u, v) such that $v \in Rem - \{u\}$,
   If $D(v) > wt(u, v)$,
   then update $D(v) = wt(u, v)$ and
   $parent(v) = u$
   EndFor

End While

- Time complexity: same as Dijkstra's Algorithm (algorithm structure is very similar)
- Correctness:
  By Induction:
  At each iteration, the tree constructed is part of some minimal spanning tree.
  Initially, this clearly holds, as starting vertex must be part of every minimal spanning tree.
  Induction: Suppose T is a minimal spanning tree which contains T', the part of the tree already constructed before the iteration.
  Suppose in the iteration, v is added to the spanning tree, and its parent is $parent(v)$.

- If $(v, parent(v))$ is an edge in T then we are done.
- Otherwise, consider $G' = T \cup \{(v, parent(v))\}$. This graph contains a loop, and n edges, where $|V| = n$.
- Furthermore, the loop must contain the edge $(v, parent(v))$, and some other edge which is not already in the tree T', say (w, w').
- Now, if we delete (w, w') from G', we get a spanning tree T''.
- Weight of this spanning tree is no more than weight of T (as we added $(v, parent(v))$ and deleted (w, w'), the weight of former being no more than the weight of latter).
- T'' is thus a minimal spanning tree, which contains T' and $(v, parent(v))$.

Kruskal's Algorithm

- Edges are initially sorted by increasing weight.
- Start with n isolated vertices (Forest)
- Grow the Forest by adding one edge at a time
- At each stage, add minimal weight edge which can be added without creating a cycle.
- Stop when $n - 1$ edges have been added.

Kruskal's Algorithm

Kruskal; Input: $G = (V, E)$

Sort E in non-decreasing order of the edge weights $w(e_1) \leq w(e_2) \ldots$.

$E_T \leftarrow \emptyset$; $counter \leftarrow 0$, $k \leftarrow 0$

While $counter < |V| - 1$ do

$k \leftarrow k + 1$.

If $E_T \cup \{e_k\}$ is acyclic

Then, $E_T \leftarrow E_T \cup \{e_k\}$; $counter \leftarrow counter + 1$.

Endwhile

End

Correctness

- Suppose O_T is the edges in the optimal spanning tree.
- Suppose k is smallest so that $e_k \in E_T - O_T$.
- Adding e_k to O_T creates a cycle with at least one edge e' in O_T which has at least as large a weight as e_k.
- Dropping e' and adding e_k to O_T gives an MST of at most the same weight.

Time Bounds

- Can consider the checking of acyclic as follows:
- Keep the trees in the forest as sets. If the edge e_k has endpoints in different sets, then add it to the forest and union the two sets; otherwise skip the edge.
- Union-Find algorithm: Takes $O(m \log n)$ time for n unions and m finds.
- Sorting takes $O(m \log m)$ time, which also bounds the overall running time.

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. *)
EndRepeat

- 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 $O(n \log n)$.

- 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.

Take Away Message

- Greedy algorithm makes a locally optimal choice. In some cases this leads to globally optimal solution.
- The choice in each step must be feasible, locally optimal, and irrevocable.