Greedy Algorithms

Main Focus: Huffman Coding Algorithm

Keeping in mind that the Greedy technique is best suited for finding a solution for the immediate situation, we agree that the Basic Greedy Design comprises of-

1. Greedy Choice Property
2. Optimum Substructure Property

Greedy Choice Property: This property says that the globally optimal solution can be obtained by making a locally optimal solution which gives us the greedy approach. It depends and is influenced by previous choices and not by the future choices.

Optimal Substructure: The main goal here is to solve subproblems and build up solutions to solve larger problems.
The ideal structure:

> Optimal solution to problem += Optimal solution to subproblems

---

Activity Selection:

Set of activites = $\{a_1, a_2, \ldots, a_n\}$ where, a_i needs resource time a_i during period $[s_i, f_i)$ where s_i and f_i are 'start time' and 'finish time' respectively.

GOAL: Select the largest possible set of non-overlapping activities. They should be mutually compatible.

The Greedy Choice: The activity with the least finish time (say a_0 belongs to some optimum solution).

Proof: Let A be a maximum size subset of mutually compatible activities from S. We then order thr activites in monotonicall increasing order of the finish times. Let a_k be the first activity in A.

> If $a_k = a_0$: done. Else construct $B = A - ([a_k] \cup [a_0])$

The Optimum Substructure: Activites that start after a_0 finishes.

> Assume that $f_0 \leq f_1 \leq \ldots \leq f_n$

> $Greedy - Activity - Selector(s, f, n)$\
> $A \leftarrow [a_1]$\
> $i \leftarrow 1$
>
> for $m \leftarrow 2$ to n
>
> do if $s_m \geq f_i$
>
> then $A \leftarrow A \cup [a_m]$
>
> $i \leftarrow m$
> return A

Huffman Codes:

The basic idea behind the Huffman Coding Algorithm is to use fewer bits for more frequently occuring characters. It compresses the storage of data using variable length codes.

When reading a file, the system generally reads 8 bits at a time to read a single character. This is highly ineffecient. Major reason for this being is that some characters are more frequently used than the other characters. The simple soltuion to that is to give longer binary codes for less frequency characters and groups of characters.

IMPORTANT: NO TWO CHARACTER CODES ARE PREFIXES OF EACH OTHER.

The Precise Problem: Given the frequencies $f_1, f_2, \ldots, f_n$ of n symbols, we want a tree whose

The Greedy Choice approach: The two symbols with the smallest frequencies must be at the bottom of the optimal tree. They behave as children of the lowest internal node. We support this claim by saying that we replace them the two symbols with whatever is the lowest then and clearly, the encoding just improves.

The Optimum Substructure property: The cost of the tree is sum of the frequencies of all leaves and internal nodes apart from the root of the tree.

EXAMPLE:

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

So we create a **binary tree** for each character that also stores the frequency with which it occurs. In the list, find the two binary trees that store minimum frequencies at their nodes. When u build the tree, each leaf node corresponds to a letter with the code. To determine the code, we traverse from root to leaf node.

1. For each move to the left, append 0 to the code.
2. For each move to the right, append 1 to the code.

Thus, we get following codes:

Character	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

Now, we try to see and understand how many *bits* we saved to appreciate the algorithm.
All we do is, calculate the number of bits originally used to store the data nd substract from that the number of bits used to store data using the algorithm.

1. Since we have 6 characters assume each character to be stored using 3-bit code. Since, there are 133 characters, we multiply total frequencies by 3 and get total bits $3 * 133 = 399$.
2. Using the Huffmans Algorithm, we get:

Character	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85

TOTAL: 238

Thus, we have saved $399 - 238 = 161$ bits which is about 40 of the storage space.

Algorithm for Huffman Coding:

> procedure Huffman(f)
>
> Input: An array $f[1...n]$ of frequencies
>
> Output: An encoding tree with n leaves
>
> Let H be a priority queue of integers, ordered by f
>
> for $i = 1$ to n : insert (H, i)
>
> for $k = n + 1$ to $2n - 1$:
>
> i = deletemin (H) , j = deletemin (H)

> create a node numbered k with children i, j
>
> $f[k] = f[i] + f[j]$
>
> insert (H, k)

Entropy:

It relates to the information in the system and how to optimum encode it. We realise the relation:

$morecompressible = lessrandom = morepredictable.$

Suppose there are n possible outcomes, with probabilities $p_1, p_2, \ldots, p_n$. Assume that $p_i's$ are all of the form $(1/2)^k$ and are observed frequencies.

Thus the average number of bits needed to encode a single draw from the distribution is:

> $\sum_{i=1}^{n} m * p_i * log(1/p_i)$, where $m = 1$

This is the *entropy* of distribution, a measure of how much randomness it contains.

Time Complexity: $O(n * logn)$, since there will be one build_heap, $2n - 2$ delete_mins, and $n - 2$ inserts, on a *priority queue* that never has more than n elements.
