Lecture 11: Greedy algorithm

Now, we start Chapter 16, the Greedy algorithms. Greedy algorithms make locally optimal decisions, which sometimes lead to globally optimal solution. Our first example is the coin change problem. You are given unlimited supplies of several coins (say quarters, dimes, nickels and pennies), you want to give out \( n \) cents using the fewest number of coins. The common greedy strategy of using largest coins is sometimes globally optimal, as in the US coins. However, if we slightly change the coin types, the situation will change. Suppose we have quarters, dimes and pennies but no nickels, the greedy strategy does not work. This can be seen by \( n = 30 \) example.

Our next example is the activity selection problem in the textbook. In class, several strategies were proposed:

1. First come, first serve.
2. Picking shortest requests.

We showed counter-examples that these two strategies are not globally optimal.

I claim that choosing the requests with earliest finish time is optimal. In class, I showed how to prove this strategy is optimal. This is done in two steps. This needs considering an optimal solution \( O \), where \( O = \{ o_1, \ldots, o_k, o_{k+1}, \ldots \} \). We denote the solution found by greedy algorithm is \( A = \{ a_1, \ldots a_k \} \). Note that if \( O \) and \( A \) contain the same number of requests, we are done. Otherwise, we assume \( o_{k+1} \) exists. Note: comparing the greedy solution \( (A \text{ in this problem}) \) and some optimal (unknown but must exist) solution \( (O \text{ in this case}) \) is a frequently used scheme in proving the correctness of the greedy algorithm. Please carefully review the proof process. The outline is given below.

First, I show that \( f(a_i) \leq f(o_i) \) for each \( 1 \leq i \leq k \). This is proved by induction. When \( i = 1 \), it is true because the greedy algorithm picks the one with smallest finish time. Then we assume the claim holds for all indices less than \( i \). We now show \( f(a_i) \leq f(o_i) \). The key to this is that if this is not true, the greedy algorithm would have selected \( o_i \) instead of \( a_i \) since \( o_i \) is compatible with the previous \( a_{i-1} \) (why? Make sure you understand this).

Then, I show that \( o_{k+1} \) can not exist. Otherwise, \( f(a_k) \leq f(o_k) \), which will suggest the greedy algorithm will not stop picking after \( a_k \).

We now switch our attention to the Huffman encoding, which is a widely used data compression method. Suppose we have a text from some alphabet with A, B, C, D. We need to convert the text to a binary encoded string. The straightforward approach is to use 2 bits per symbol: A is encoded as 00, B is 01, C is10, and D is 11. Suppose the text is 100M long, and we need 100M*2=200 MBits. One observation is that symbol frequency is quite different. Thus, to reduce the encoded string length, we want to encode more frequent words with shorter codes. For example, let us suppose the number of As in text is 60M, Bs of 5M, Cs of 15M, and Ds of 25M. We will use variable length codewords. To avoid ambiguity for decoding, we require that no code word is a prefix of other code words (such codewords are called prefix code). One possible prefix code for our example is: A:0, B:110, C:111 and D:10. This encoding leads total size of 60\( \times \)1 + 5\( \times \)3 + 15\( \times \)3 + 25\( \times \)2 = 170 MBits, which is savings of 15 percent.

Now here is the problem formulation: suppose symbol frequencies are known, \( f_1, f_2, \ldots, f_n \) for the \( n \) symbols in the alphabet. Our goal is to design codewords that minimize total encoded length. That is, we want minimize \( \sum_i f_i \cdot |w_i| \), where \( w_i \) is the code word for the \( i \)-th symbol. A nice property of prefix code is that prefix code can be represented as a binary tree. In this binary tree, a node is either a leaf or has two children. That is, a node with a single child is not allowed (Why do we have this full binary tree property?). Each branch of the tree is labeled by a single digit, 0 or 1. Look at Figure 16.4 for an example of this tree. We call the depth of a node as the number of bits traversed from root to that node. Thus, the total length of the encoded string is: \( \sum_i f_i \cdot depth(leaf_i) \), noting leaves correspond to codewords (Why?). To be continued.