Lecture 13: Greedy algorithm and dynamic programming

We continue our discussion on huffman coding. Recall that we can choose two lowest frequency symbols and create a subtree. From the second formulation, we know we can assign the frequency of this newly created internal node (called \( w \), with two least frequent symbols \( w_1 \) and \( w_2 \) as children) as the summation of the frequency of \( w_1 \) and \( w_2 \). This allows us to shrink in this problem: get rid of \( w_1 \) and \( w_2 \), and add a new symbol \( w \) into the list of symbols to process. Then we repeat the process. But why does this work?

It is useful to think of picking formulation 2: our goal is to find a tree where each node has a frequency (for internal node, the frequency must equal to the total frequencies of the leaves under this node). Recall that we now have \( n-1 \) symbols in the smaller problem. After getting rid of \( w_1 \) and \( w_2 \) but keep their parent node \( w \), the tree is still a valid Huffman encoding tree (verify this yourself). The only difference is that we have a new leaf \( w \) with frequency \( f(w_1) + f(w_2) \). To minimize the total encoding length of the original tree, of course we should also minimize the total length of this new tree. Due to the equivalence of the first and second formulations, we are really seeking the optimal Huffman encoding tree for the \( n-1 \) symbols and continue.

This leads to a simple algorithm: at each iteration, pick the two symbols with lowest frequencies and form a subtree with these two symbols: remove these two symbols and create a new symbol with the summation of the these removed symbols as its frequency. Continue until there is only one symbol left. This algorithm needs frequent operations to obtain (and extract) the symbols with lowest frequency. This can be done by a heap, which makes each iteration to take \( O(\log n) \) time for \( n \) symbols. So the total running time is \( O(n \log n) \). Try to run some examples yourself to make sure you understand this algorithm.

Now let us think about what properties of a good greedy strategy has. For greedy algorithm to work, we should be able to make locally optimal choices at the moment, and after making the choice, we need to prove we still always get optimal solution. Second, we need the optimal substructure property: after making a choice, we now have a new (and smaller) problem; the key to greedy algorithm to work is that we can solve the smaller problem and then combine this solution with the decision made at the greedy choice to obtain the final solution. Read our textbook for more detailed explanation.

Now we start to discuss what is dynamic programming. Dynamic programming is one of the most useful algorithmic techniques. We use an example problem, longest increasing subsequence (LIS), to illustrate the idea. Given a list of \( n \) numbers (i.e. sequence, denoted as \( A \)), we define subsequence as a sub-list of the numbers (with the original order). For example, let the sequence being 4; 2; 7; 5; 3; 5; 9; 6. A subsequence is 2, 5, 4, 6. We call a subsequence an increasing subsequence if this subsequence consists of numbers in the increasing order. Example of increasing subsequence of the above example is 4, 7, 9. The LIS problem is to find the longest increasing subsequence (LIS). We first study how to compute the length of LIS. In this example, the length of LIS is 4.

Let us define \( L(i) \) as the length of a LIS that ends at position \( i \) (position means index of the array). Then, the length of the LIS of the whole list is simply \( \max_{i=1}^{n} L(i) \) (Why? Make sure you understand this). To see how to compute \( L(i) \), we consider a LIS ending at \( i \). Its previous number should be some at some \( j < i \), where \( A[j] < A[i] \). But which one is \( j \) if there are multiple such \( j \)? Let \( j \) be the position immediately preceding \( i \) in the LIS ending at \( i \). Then we know \( L(i) = L(j) + 1 \). This is because since the LIS uses \( j \)-th number, the it must take the LIS ending at \( j \) (otherwise it contradicts the assumption of the LIS ending at \( i \)). Since we do not know which \( j \) to use, we take the best: \( L(i) = \max_{j<i, A[j]<A[i]} L(j) + 1 \).

Now to compute a \( L(i) \), we can use the previous recursion: it will then recursively compute \( L(j) \). This works, but with a great cost. The problem is, many subproblems \( L(j) \) are computed multiple times. To see how serious the problem is, we consider the worst case for computing \( L(n) \). Let \( S(i) \) be the number of recursion calls (i.e. nodes in recursion tree) for computing \( L(i) \). Then, in the worst case,

\[
S(n) = 1 + \sum_{i=1}^{n-1} S(i)
\]

We observe \( S(1) = 1, S(2) = 2, S(3) = 4 \). Then we guess: \( S(i) = 2^{i-1} \), and this is easily provable using induction. So, \( S(n) = 2^{n-1} \), which implies that the recursion based algorithm runs in \( \text{exponential} \) time. Now
we apply the dynamic programming that will lead to a polynomial time algorithm. The key idea is to memorize the subproblem solutions so that we do not re-compute them, using a bottom-up approach.

1: for \( i = 1 \) to \( n \) do
2: \( L[i] = 1 \)
3: for \( j = 1 \) to \( i - 1 \) do
4: \( \text{if } A[j] < A[i] \text{ and } 1 + L[j] > L[i] \text{ then} \)
5: \( L[i] = 1 + L[j] \)
6: end if
7: end for
8: end for
9: return \( \text{MAX}_{i=1}^{n}(L[i]) \).

Note: when computing \( L(i), L(j) \) has already been computed. And for each \( L(i) \), it gets updated just once. Running time: there are \( n \) iterations, each iteration taking at most \( O(n) \) time. So the total time is \( O(n^2) \).

The only remaining issue is that we only know the LIS’s length, but what is LIS itself? That is not so hard. We suppose the LIS ending at node \( p \) (i.e. we know the last letter of the LIS, \( A[p] \)). To find the one before \( A[p] \), we perform trace back. In other words, we check which position \( j \) that leading to the current value at \( L[p] \); then we move to this \( j \). We do this recursively (each time figure out its immediate previous node). I suggest you to work out a few examples to better understand this.

**Lecture 14: Dynamic Programming**

The main subject of this lecture is the longest common subsequence (LCS) problem. Our textbook covers this topic very well (read Section 15.4). So I will only mention a few things. We use a two dimensional array: \( LCS[i, j] \), which represents the length of LCS of two prefixes \( X[1..i] \) and \( Y[1..j] \). By this definition, the length of LCS of \( X \) and \( Y \) is equal to \( LCS[n, m] \), where \( n \) and \( m \) are the length of \( X \) and \( Y \) respectively. In class, someone suggested the solution can be found by finding the largest value with \( LCS(i, j) \). This is correct, but checking the entire LCS array is not necessary because \( LCS[n, m] \) is the largest among \( LCS[i, j] \) (why?).

A key to dynamic programming is how to come up with recurrences of the subproblems. The recurrence allows you to compute the value of a subproblem using the pre-computed values of smaller subproblems. This often involves analyzing the situation to see what you can do to some specific parts of the subproblems. Often you have multiple choices, and you will just take the most desirable one. In the LCS example, we consider what happens to the last two characters. If these two characters are the same, then they must be in the LCS (Why? Make sure you understand this).

For the subproblem \( LCS(i, j) \), we look at the two cases: \( X[i] = Y[j] \) and \( X[i] \neq Y[j] \). When \( X[i] = Y[j] \), I explained in class that it is always safe to include \( X[i] \) in the LCS of \( X[1..i] \) and \( Y[1..j] \). Make sure you understand why. When \( X[i] \neq Y[j] \), clearly either \( X[i] \) is not included or \( Y[j] \) is not included in the LCS. And one of these two must happen. We then take the better of the two choices.

During the lecture, I described how the LCS dynamic programming table is filled in, and how to find the LCS by trace-back. Read the textbook if you have doubts. You should be familiar with the track back after working on the two homework problems.

Finally, here are some tips from my personal experience about how to apply dynamic programming. The usual procedure I take for designing a dynamic programming algorithm.

- **Step 0.** First find a good subproblem definition. This is sometimes tricky but is really the key for a good dynamic programming algorithm.
- **Step 1.** Make sure you know how to find answer for the original problem from the subproblems you have defined. This usually helps you to understand the subproblems better.
- **Step 2.** Describe how to get started: how are you going to determine values for the smallest subproblems.
- **Step 3.** Describe how you can obtain the value for the larger subproblems from known smaller subproblems (this is the bottom-up step).
- **Step 4.** Write down the algorithm.
- **Step 5.** Analyze the running time.