Lecture 11: Lower bound, Counting sort and Greedy Algorithm

Now, lower bound for sorting. We discuss the issue of whether $O(n \log n)$ sorting is optimal. It is, if we assume only comparison is allowed. See Section 8.1 for more details. The basic idea is as follows. Suppose we perform $d$ comparisons. There are $2^d$ possible combinations of results of these $d$ comparisons. Note that we must determine the proper ordering of the $n$ elements. Since there are totally $n!$ possible orderings, this implies that $2^d \geq n!$. Otherwise, some ordering will not be recognized by the sorting algorithm using only $d$ comparisons, since each comparison result combination can lead to no more than one ordering. Thus, $d \geq \log(n!)$. Note that $n! \geq (n/2)^{n/2}$ (by taking the second half of the elements in $n!$ and then approximate with $n/2$ for each remaining element), we know, $d \geq \Omega(n \log n)$. Thus, the running time will be $\Omega(d) = \Omega(n \log n)$.

Now counting sort. This is explained well in the textbook. One thing to note is that we do need to use the definition of $C[i]$ being equal to the number of array elements smaller or equal to $i$. This is because it is often not enough to output a list of integers (which can be easily done from the first version of definition of C). For example, a database record contains not just its ID, but also other information like name, address, and so on. So we need to be able to place the original record into the result.

The other important thing to note is how to compute $C[i]$ efficiently. Naively $C[i]$ is equal to $\sum_{k=0}^{i} C_0[k]$, where $C_0[i]$ is equal to the number of elements of $A$ equal to $i$. Directly computing like this will not be efficient. Remember: we need to compute each $C[i]$ in constant time. But a simple trick is: $C[i] = C[i-1] + C_0[i]$. So we will compute $C[i]$ from 0 to $k$ sequentially using $C[i] = C[i-1] + C_0[i]$ (since $C[i-1]$ has already been computed at the time we need to compute $C[i]$). This concept is crucial: we will use something similar extensively in the later part of this course.

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 an 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 $O = \{a_1, \ldots, a_k, a_{k+1}, \ldots \}$. 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 $a_{k+1}$ exists. Note: comparing the greedy solution ($A$ in this problem) and some optimal (unknown but must exist) solution ($O$ 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 $f(a_i) \leq f(a_i)$ 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 $f(a_i) \leq f(a_i)$. The key to this is that if this is not true, the greedy algorithm would have selected $a_i$ instead of $a_i$ since $a_i$ is compatible with the previous $a_{i-1}$ (why? Make sure you understand this).

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

Lecture 12: Greedy algorithm

Our next example is the well-known Knapsack problem. Given $n$ items, each with $w_i$ weight and $v_i$ value. We want to get the most value while the total weight is no more than $W$. There are two variations: 0/1
Knapsack (where you either take an item or not) and fractional Knapsack (which allows fractional packing). This fractional version allows an easy greedy solution: prefer items with higher value per unit weight. That is, we sort items by their $v_i/w_i$ ratio and pick items with the highest unit value first and take as much as you can (recall fractional item is OK). This is clearly an $O(n \log n)$ time algorithm. This may not give optimal solution for the 0/1 Knapsack problem. In class, I give the same example as in the textbook: $W = 50$ and we have three items: v60/w10, v100/w20, v120/w30. It is not hard to see the greedy strategy does not give the optimal solution here. In class, I showed that greedy strategy works for the fractional version. The proof uses the exchange argument again. Suppose the greedy algorithm does not give optimal solution. And so we compare solution here. In class, I gave the same example as in the textbook: $W$ we sort items by their $v_i/w_i$. This fractional version allows an easy greedy solution: prefer items with higher value per unit weight. That is, $v_i/w_i$ ratio and pick items with the highest unit value first and take as much as you can.

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 is 10, 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 20M. 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:10, C:111 and D:10. This encoding leads total size of $60 \times 1 + 5 \times 3 + 15 \times 3 + 20 \times 2 = 160$ MBits, which is savings of 20 percent.

Now here is the problem formulation: suppose symbol frequencies are known, $f_1, f_2, ..., f_n$ for the $n$ symbols in the alphabet. Our goal is to design codewords that minimize total encoded length. That is, we want minimize $\sum_i f_i |w_i|$, 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: $\sum_i f_i \cdot \text{depth}(\text{leaf}_{f_i})$, noting leaves correspond to codewords (Why?). An alternative formulation for expressing the total encoded length is to given internal nodes assigned frequencies as the summation of frequencies of the leaves under this node. We claim that total encode length is equal to summation of frequencies of all the nodes, including the internal nodes, in the tree (except the root). Why? Intuitively, we distribute the frequency of a leaf times its depth to each node on the path from root to the leaf.

We now study the properties of the complete binary tree constructed in Huffman encoding. The key observation is that the two lowest frequency symbols must be the two siblings with largest depth. To see this, suppose this is not true and the least frequent symbol is not assigned to the leaf with largest depth. Then we simply swap the symbol with the largest depth with the least frequent symbol, and this swapping will make the text shorter. Also note that there must exist a symbol as the sibling of the deepest symbol.

This implies 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. 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. Try to run some examples yourself to make sure you understand this algorithm.

But why does this work? That is, why the algorithm gives optimal solution? We have all the pieces already but there is no time to wrap it up in class. I will continue next time.