Lecture 10: Counting Sort, Selection and Start of Greedy Algorithms

Now counting sort. This is explained well in the textbook. One thing to note is that we do not 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.

Our next subject is the selection problem. A frequently used statistic is the median, which is sometimes better than average (mean). One example is that there are 1000 numbers, where 999 of them are 1 but one is 10000000. Here, median is more representative than mean. Now we consider the more general problem: finding the \( k \)-th smallest element among \( n \) elements. Here is the problem formulation: given a list of \( n \) elements, and an integer \( k \). Our goal is finding the \( k \)-th element in the list. A naive solution is to first sort then output the \( k \)-th item. This can be done in \( O(n\log n) \) time. But it seems to waste a lot of time: we only want to find the \( k \)-th item and we do not need to know the full order. Here, we can apply a randomized divide and conquer method. Just like Quicksort, we partition the input list by picking a pivot \( v \). For this pivot \( v \), we divide the given list into three parts, those smaller than (denoted as \( S_l \)), equal to (\( S_m \)), or larger than \( v \) (\( S_r \)). Now, we define Selection(\( S, k \)) as the function that will return the \( k \)-th element in list \( S \). Then, Selection(\( S, k \)) = Selection(\( S_l, k \)) if \( k \geq |S_l| \), and return \( v \) if \( k > |S_l| \) and \( k \leq |S_l| + |S_m| \), and Selection(\( S_r, k - |S_l| - |S_m| \)) otherwise. Just like Quicksort, the running time depends on how evenly the two lists are partitioned. The worst case running time is \( O(n^2) \). We call lucky case as the situation where the two subproblems are relatively balanced: if the selected \( v \) is within \([25\%, 75\%]\). In this case, the two subproblems can not be very unbalanced (at most 0.75 of original size). If you pick randomly, 50% to be lucky. The expected number of partitions needed to achieve a lucky partition is 2. Thus, after two rounds, the size is no more than 0.75. Let \( T(n) \) be expected run time. We have: \( T(n) \leq T(0.75n) + cn \), and the expected running time is \( T(n) = O(n) \).

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.

Lecture 11: Greedy algorithm

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 (\(A\) 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 \(f(a_i) \leq f(o_i)\) 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 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 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 to minimize \(\sum_i f_i \times |w_i|\), 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: \(\sum_i f_i \times \text{depth}(\text{leaf}_i)\), noting leaves correspond to codewords (Why?). To be continued.