CSC373S Lecture 12

- We continue with our discussion of divide and conquer (DC). In doing so, we will start with an application that might not seem like DC since we may intuitively assume DC entails breaking the problem into at least 2 subproblems but I do not think this is a necessary aspect of DC.

  "Decrease and conquer" (Wikipedia definition) algorithms are divide and conquer algorithms where after splitting the problem we only have to solve one subproblem. A canonical example is binary search. That is, consider the following search problem: given y and a sorted list $x_1 < x_2 < ... < x_n$, determine if $y \in S = \{x_1, \ldots, x_n\}$ or more generally find the closest $x_j \in S$ to y. The binary search recurrence is $T(n) = T(n/2) + O(1)$ which implies $T(n) = O(\log n)$.

- In our next three applications of DC, I am going to discuss DC applications which also use randomization. The KT text as well as Jepson's notes leave randomized algorithms to the end of the course just as approximation algorithms are considered later in the course. (I would strongly suggest looking at the Jepson notes or KT section 13.3 as a quick way to review/learn the few probability concepts needed in what follows. See also 13.12))

  Just like any of our algorithmic technqiues can be and often are used for approximation, randomization can be utilized with any of our algorithmic techniques (and almost as a technique in itself, say as in random sampling). So rather than wait for the end of the term, I want to start discussing randomized algorithms now. You may or may not already be familiar with quicksort as a randomized algorithm (as well as a deterministic algorithm that is fast "on average" for sorting).

  Here then is quicksort Quicksort(S) as a randomized algorithm. (See section 13.5 of the text.) Let $S = \{a_1, \ldots, a_n\}$ be the set of elements to be sorted. For simplicity lets assume all elements are distinct.

  IF $|S| \leq 3$, then brute force sort S and Output sorted list
  Else Choose a splitter pivot element $a_i \in S$ uniformly at random
  For each $a_j \in S - \{a_i\}$
  Put a_j in S^- if $a_j < a_i$ and put a_j in S^+ if $a_j > a_i$
  End For
  Output Quicksort(S^-)
  Output a_i
  Output Quicksort(S^+)

  I will delay the analysis for now (see the analysis for the randomized "Select k^{th} smallest element" algorithm below) but it is not hard to show that Quicksort has *expected run time* $O(nlogn)$ for every input set S. What is this expectation over? We are choosing pivots in S randomly and hence the run time is a random variable. In the worst case (i.e. when pivots always turn out to be say the maximum or

minimum values in the set being sorted), the run time would be $\Omega(n^2)$. But with sufficiently high probability, the pivot elements give a reasonably balanced partition.

- Lets consider a related application. Here we are also using randomization in a divide and conquer (or more specifically, a decrease and conquer algorithm) and well generalizing a problem to lend itself to a DC approach. (See again section 13.5 of KT.)

  The randomized median algorithm. Suppose we want to find the median (i.e. the $\lceil (n/2) \rceil$ th smallest/largest number in a (unsorted) list of n (say real) distinct numbers $S = \{x_1, ..., x_n\}$. We will be led to consider more generally, the *selection problem*: given S and k, we want to compute the k^{th} smallest/largest element. For k small (or close to n) we can clearly do this in $O(\max\{k, n-k\} \cdot n)$. Of course we can also sort the numbers with complexity $O(n \log n)$.

  In some settings (comparison based algorithms) we know that we cannot sort in better than $(nlogn)$ comparisons but can we compute the median (the seemingly hardest case for element selection) in $O(n)$ time? Or must we essentially sort the elements?

  This was an open problem for some time (until the 1970s) when a deterministic $O(n)$ algorithm was found (while trying to prove the impossibility of such an algorithm!) but prior to this it was known how to find the median (or k^{th} smallest/largest) by a randomized algorithm with *expected time* $O(n)$.

  We emphasize again that for many divide and conquer algorithms, as in dynamic programming, it often pays to generalize the problem! For another DC example, see the "counting inversions" problem in section 5.3 of KT. In this case instead of just counting inversions, we expand the problem to counting inversions and sorting the list at the same time. Also the 4th question of assignment 2 seems to call for a generalization of the problem as stated.

  So as already stated, instead of just trying to compute the median it is better to consider computing the k^{th} smallest for any given k. (I think the KT text says k^{th} largest but the algorithm seems to be for the k^{th} smallest.)

  Suppose we randomly choose $i \in \{1, 2 \ldots, n\}$, and let $S^- = \{x_j \in S | x_j < x_i\}$ and $S^+ = \{x_j \in S | x_j > x_i\}$

  Then if we were really lucky, $|S^-| = k-1$ and hence x_i is the k^{th} smallest. If $|S^-| > k-1$ then the k^{th} smallest in S is the k^{th} smallest in S^-. If $|S^-| = \ell < k-1$ then the k^{th} smallest in S is the $[k-(\ell+1)]^{th}$ smallest element in S^+.

  If we are very lucky and x_i is the $(k-1)^{th}$ smallest, then we computed the desired k^{th} smallest in the $O(n)$ steps used to partition S according to the location of x_i. Of course, it would be very lucky (prob $1/n$) to choose i such that x_i is the $(k-1)^{th}$ smallest. If we are very unlucky and always choose the index of the largest element then the complexity would be $O(n + (n-1) + ...) = O(n^2)$ But there is actually a good chance (i.e. good probability) that we will be making substantial progress on many steps if we randomly choose i. If say we are always getting rid of $(1/4)^{th}$ of the elements then $T(n) = T(3n/4) + O(n)$ so that $T(n) = O(n + 3n/4 + 3^2 n/4^2 + \ldots)$ which implies $T(n) = O(n)$. Although we cannot guarantee always getting rid of

$(1/4)^{th}$ of the elements in each recursive call, the likelihood is that it won't take many iterations to do so.

Here then is a more mathematical analysis:

Let the j^{th} phase ($j = 0, ..., log_{4/3}n$) be when the number n_j of elements in the current set under consideration (call it S_j) satisifies $n \cdot (3/4)^{j+1} < n_j = |S_j| \leq n \cdot (3/4)^j$. Let X_j be the random variable (r.v.) representing the number of iterations taken while the algorithm is in the j^{th} phase. (Note: 3/4 is rather arbitrary as we can see from the analysis.) If Y_j = the number of steps (i.e. the time) while in the j^{th} phase, then $Y_j = O(X_j \cdot n3/4)^j)$

The total complexity = $\sum_{j=0,...} Y_j$ and hence E[total complexity] = $\sum_j E[Y_j]$ by the linearity of expectation.

Call a choice of i and hence x_i to be a central element if $|S^-|$ and $|S^+|$ are both $\geq n_j/4$. This is equivalent to saying that the *rank of* x_i in S is in the range $[1/4, 3/4]$. If x_i is a central element then phase j ends. It can end by choosing a number of non central elements but for sure the number of recursive calls while in phase j is at most the time to pick a central element.

The probability of choosing a central element is at least $1/2$ so that the expected time to choose a central element is at most 2. (See section 13.3 of text.) Hence $E[X_j] \leq 2$ and $E[Y_j] = O(2n(3/4)^j$ and hence E[total number of steps] = $\sum_j E[Y_j] = O(n)$.

NOTE: Looking at the expected time for a randomized algorithm that is computing a function is one way to measure its complexity. Sometimes a more detailed analysis will state a result of the form "with high probability" the time is at most T.

- For a final randomized DC application (very related to Quicksort) we present an approximation algorithm for what is called the (weighted) *feedback arc set problem on tournaments* (FAS-Tournaments) and its application to agggregate (aka universal) ranking and aggregate clustering. Of course, applying FAS-Tournaments to these latter problems is just another example of *reduction*. One final idea we will encounter is that sometimes it is beneficial to consider 2 or more algorithms for a given problem and take the best answer (for a given input) provided by the different algorithms. This can be a useful "meta idea" (like reduction) when the different algorithms tend to work well on different input sets. As another example of this meta idea, we can show that the best of two greedy algorithms for the knapsack problem provides a 2-approximation whereas no single greedy algorithm (for the template we have suggested) can provide a constant. approximation.

- A weighted tournament is a complete directed graph $G = (V.E)$ with a weight function $w(e) = w_{ij}$. Here complete means that for every $i \neq j$, the directed edge (i, j) exists. We will only be interested in weighted tournaments where $w_{ij} + w_{ji} = 1$ in which case we can think of the weights as probabilities. An unweighted tournament is such a weighted tournament where all edge weights are in $\{0, 1\}$; that is, we can think of an unweighted tournament as one where for every $i \neq j$, exactly one of (i, j) or (j, i) is in E. The weighted feedback arc set problem in a weighted graph

is the problem of choosing a total ordering $<_\pi$ on the vertices (i.e. choosing a permutation π of the vertices) so as to minimize $\sum_{(i,j):i<_\pi j} w_{ji}$. That is, we are trying to minimize the combined weight of all edges that are inconsistent with the total ordering given. In the unweighted case this is simply $|(i,j) : (j,i) \in E|$.

The weighted FAS on tournaments problem was known to be NP-hard and recently the unweighted problem has been shown to be "close to NP-hard" in the sense that a poly time optimal (or even PTAS) algorithm will show that every problem in NP can be solved in randomized polynomial time.

As a direct application of the FAS on tournaments, consider the following *tournament ranking problem*. Lets say that n players have played against each other at least once. This induces a weighted (tournament) graph which is a complete directed graph with edge weights w_{ij} for each directed edge (i,j) where in this case w_{ij} = fraction of times player i beat player j. Note here that $w_{ij} + w_{ji} = 1$. The tournament ranking problem (think of this as seeding players) is to produce a total ordering π so as to minimize the sum $\sum_{(i,j):i<_\pi j} w_{ji}$; that is, to minimize the (weighted) number of pairs which are improperly ranked which is precisely the weighted FAS-Tournament problem.

Let me mention another application of a weighted tournament; namely, aggregate ranking. Consider a set of say k search engines each rankina the same n documents (or k different voters voting on a slate of candidates). Then we can set up a weighted tournament with n nodes where w_{ij} = fraction of search engines ranking i above j. In this case we have an additional triangle inequality property in that $w_{ij} \leq w_{ik} + w_{kj}$. The goal is to find an aggregate ranking that is in some sense "most compatible" with the given k rankings. To make this question precise, we need a definition for the distance between two rankings.

The Kendall-Tau distance $d(\pi_1, \pi_2)$ between two different rankings (permutations) is the number of pairs ranked oppositely.

Kemeny criteria (cost) for meta ranking (Kemeny aggregation): Find an aggregate π so as to minimize $\sum_{1\leq i\leq k} d(\pi, \pi_i)$. This then becomes the weighted FAS on tournaments problem with weights satisfying triangle inequality (as well as opposite edge weights summing to 1). yields $E[cost(Best)] \leq (11/7) \cdot cost(OPT)$.