**CSC373S Lecture 11**

- We will now spend a week discussing "divide and conquer", a paradigm that 3rd year CS students have seen before. We hinted at a precise general model for greedy algorithms (but didnt say how we are allowed to order input items). We never did try to provide a precise definition for what is a DP algorithm. We also won't try to precisely define "divide and conquer" (DC) but again will hopefully give enough examples that we will intuitively know what we mean. And although we do not have definitions for DP and DC algorithms, it is worth trying to understand how they differ.

  The greedy paradigm seems te imply that we are considering an optimization problem. There are other (e.g. search) applications but it is true that we usually are considering optimization (or search) in discussing greedy algorithms. Historically, DP was developed (in the mid 50s) as an optimization technique. It has evolved as an algorithmic concept that goes beyond optimziation and search but still is mainly thought of as an optimization technique.

  Divide and conquer DC is a paradigm is probably most often used for function computation but is also used in optimization. Most of the DC applications in the KT text are about function computation. It is also the case that the text applications of divide and conquer allow us to improve over a somewhat obvious polynomial time algorithm so as to produce a faster polynomial algorithm. Our greedy and DP algorithms often allowed us to improve upon naive exponential time algorithms to produce substantially more efficient algorithms.

  Why do DC in this course? As I said, we want to contrast with dynamic programming. But maybe the main reason is the fact that many of these DC algorthms are surprising and DC is one of the best known algorithmic design techiques. Moreover, I will give an application of divide and conquer that provides an efficient approximation algorithm for a problem that is NP hard and hence (we believe) does not have an efficient optimal algorithm. Since DC is an idea that is pretty well understood and studied in previous courses, we will not spend too long in lectures on this topic but I do expect everyone to read chapter 5 of the text.

- Perhaps the first DC algorithm (and also the first expplictly stated recursive algorithm) is merge sort, invented by von Neumann in 1945. As we all know this is an $O(nlogn)$ sorting algorithm whereas the obvious sorting requires $O(n^2)$.
  This is a very typical example of a balanced divide and conquer algorithm:

  1. Divide the problem instance $\mathcal{I}$ (say of size $n$) into $a$ smaller subinstances $\mathcal{I}_1, ..., \mathcal{I}_a$) (of the same problem). Often these subinstances are roughly of the same size $n/b$. In mergesort (and a number of other well know DC algorithms) $a = 2$ and $b = 2$.

2. Recursively solve each subinstance
3. Combine the solutions for the subproblems into a solution for the initial problem.

If the dividing and combining steps take $O(n^d)$ time, then the time complexity is described by the recurrence:

$T(n) = aT(n/b) + O(n^d)$.
$T(n') = O(1)$ for some base case(s) of size $n' \leq n_0$

The resulting complexity $T(n)$ to solve problem of size $n$ depends on $a, b, d$ and how they relate.

There is a so-called master theorem for the asymptotic behavior of this class of recurrences (as in Jepson's notes and many texts); namely;

1. when $a > b^d$, then $T(n) = \Theta(n^{\log_b a})$; i.e. the recursion dominates the "overhead" of dividing and combining
2. when $a = b^d$, then $T(n) = \Theta(n^d \log n)$
3. when $a < b$, then $T(n) = \Theta(n^d)$

In the special case that $a = b = 2$ and $d = 1$ as in merge sort, it is easy to see why $T(n) = O(nlogn)$. The text talks about two ways to determine the complexity induced by these recurrences. The first and most intuitive is called "unrolling the recurrence" and is what I tend to do. The second is to "guess" (usually an educated guess using other known recurrences) a solution and then verify by induction. In fact, if one unrolls the recurrence and determines a solution, it should be verified by induction. So lets consider the computation tree of calls. Each level has a $O(n)$ cost and there are $logn$ levels. To simplify the analysis one often assumes $n = 2^r$ for some $r$. (For most applications this is not an important assumption but later when we discuss the FFT this really is an assumption for many uses of the FFT.)

N.B. Looking at that tree, note that all the problem instances are distinct instances!

- Lets consider two examples where $b = 2$ but $a > 2$. We begin with polynomial multiplication (and similarly integer multiplication where we can do the same development but just have to deal with "carries")

Let $P$ and $Q$ be two degree $n-1$ polynomials over some ring or field $F$. We are considering degree $n-1$ polynomials as they have $n$ coefficients and this will simplify notation as we tend to use $n$ as an input size parameter.

We want to compute $R(x) = P(x) * Q(x)$ . The $j^{th}$ coefficient of $R$ is $r_j = \sum_{0 \leq i \leq j} p_i \cdot q_{j-i}$. Computing any $r_j$ takes $j+1$ scalar mults and $j$ scaler additions or $O(j)$ scalar operations (in $F$). Hence the total complexity (for computing all coefficients) is $O(n^2)$.

Now consider Karatsuba's algorithm:

For simplicity let $n = 2^r$ for some $r$. We observe that $P$ and $Q$ can be both written as the sum of two smaller polynomials, namely $P(x) = \sum_{0 \le i < n/2} p_i x^i + \sum_{n/2 \le i \le n-1} p_i x^i = \sum_{0 \le i < n/2} p_i x^i + [\sum_{0 \le i < n/2} p_{i+n/2} x^i] * x^{n/2} = P'_0(x) + P'_1(x) * x^{n/2}$

Similarly $Q(x) = Q'_0(x) + Q'_1(x) * x^{n/2}$

Now the degree of $P'_0, P'_1, Q'_0, Q'_1$ is $(n-1)/2$.

So $P * Q = P'_0 * Q'_0 + (P'_0 * Q'_1 + P'_1 * Q'_0) * x^{n/2} + +(P'_1 * Q'_1) * x^n$

Hence if we could compute the 3 terms $P_0 * Q_0, P_0 * Q_1 + P_1 * Q_0$ and $P_1 * Q_1$ in $a$ polynomial multiplications (of degree $n - 1/2$) and some number (say $c$) polynomial additions then the recurrence would be $T(n) = a \cdot T(n/2) + O(n)$.

Obviously we can have $a = 4$, but then $T(n) = O(n^2)$ and we haven't gained anything. But it turns out that $a = 3$ is achievable (but $a = 2$ is not achievable) and this leads to a solution with $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

Note well: The number of poly additions will only effect the constant hidden in the big $O$ notation but not the asymptotic complexity.

How to compute these three terms in 3 poly mults (of degree $(n-1)/2$)?

multiply $(P_0 + P_1) * (Q_0 + Q_1) = P_0 * Q_0 + P_1 * Q_0 + P_0 * Q_1 + P_1 * Q_1$

multiply $P_0 * Q_0$

multiply $P_1 * Q_1$

subtract to get $P_1 * Q_0 + P_0 * Q_1$

Note well: although poly multiplication is defined without subtraction, it can be sped up using subtraction!

- Strassen's matrix multiplication (over a ring)

  Again assume $n = 2^r$

  Write matrices $A, B$ as $2 \times 2$ matrices where each entry is an $n/2 \times n/2$ matrix.

  Then A * B can be viewed as

  $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$
  $C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$
  $C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$
  $C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

  So now the question is how many $n/2 \times n/2$ matrix multiplications are needed to compute these four terms.

  Obviously it can be done in 8 matrix multiplications.

  Note this looks just like the question as to how many scalar multiplications are needed to compute the multiplication of two $2 \times 2$ matrices. But in fact there is an important difference in these questions. Whereas scalar multiplication (say over integers or reals) is commutative, matrix multiplication is not!

So the question is how many multiplications are needed to compute the multiplication of two $2 \times 2$ matrices without using commutativity of multiplication.

We get the recurrence $T(n) = aT(n/2) + O(n^2)$. We can see that for $a \geq 4$, the number of matrix additions will only impact the constant in the big $O$.

For $a = 4, T(n) = O(n^2 \log n)$

For $a > 4, T(n) = O(n^{\log_2 a})$

To improve upon the classical $O(n^3)$ time bound, we need $a < 8$. It turns out (and this is NOT at all easy to see), that $a = 7$ is possible (but $s < 7$ is not possible for non-commutative $2 \times 2$ matrix multiplication) which results in $n \times n$ matrix multiplication being computable in $O(n^{\log_2 7})$ which is approximately $O(n^{2.807})$. The current best algorithm due to Coppersmith and Winograd (in terms of the exponent of $n$) has complexity approximately $O(n^{2.38})$.

Here (from Wikipedia) are the desired non-commutative multiplications and how they are combined:

$\mathbf{M_1} := (\mathbf{A_{1,1}} + \mathbf{A_{2,2}})(\mathbf{B_{1,1}} + \mathbf{B_{2,2}})$

$\mathbf{M_2} := (\mathbf{A_{2,1}} + \mathbf{A_{2,2}})\mathbf{B_{1,1}}$

$\mathbf{M_3} := \mathbf{A_{1,1}}(\mathbf{B_{1,2}} - \mathbf{B_{2,2}})$

$\mathbf{M_4} := \mathbf{A_{2,2}}(\mathbf{B_{2,1}} - \mathbf{B_{1,1}})$

$\mathbf{M_5} := (\mathbf{A_{1,1}} + \mathbf{A_{1,2}})\mathbf{B_{2,2}}$

$\mathbf{M_6} := (\mathbf{A_{2,1}} - \mathbf{A_{1,1}})(\mathbf{B_{1,1}} + \mathbf{B_{1,2}})$

$\mathbf{M_7} := (\mathbf{A_{1,2}} - \mathbf{A_{2,2}})(\mathbf{B_{2,1}} + \mathbf{B_{2,2}})$

$\mathbf{C_{1,1}} = \mathbf{M_1} + \mathbf{M_4} - \mathbf{M_5} + \mathbf{M_7}$

$\mathbf{C_{1,2}} = \mathbf{M_3} + \mathbf{M_5}$

$\mathbf{C_{2,1}} = \mathbf{M_2} + \mathbf{M_4}$

$\mathbf{C_{2,2}} = \mathbf{M_1} - \mathbf{M_2} + \mathbf{M_3} + \mathbf{M_6}$

Note : Of course, we have not given any insight here into how this algorithm was created. Before Strassen's algorithm it was widely believed that $\Omega(n^3)$ was necessary for matrix multiplication and the closely related problem of inverting a matrix or solving a system of equations.

Strassen's matrix multiplication algorithm had a big impact in computer science and especially theoretical CS. It was not the first example of a big surprise but it was such an influential example that it had a big impact in trying to really understand the intrinsic complexity of computing various problems.