[Overall Information]

In this class, we studied an algorithm for integer multiplication, which improved the running time from $\Theta(n^2)$ to $\Theta(n^{1.59})$.

We then used some of the insights gained in analyzing Strassen’s algorithm for matrix multiplication.

Finally, we briefly discussed the Quick-Sort algorithm.

[Announcements]

1. Homework#3 was assigned. It covers divide and conquer algorithms and techniques for solving recurrence relations.

2. Quiz#4 is scheduled on 03/08/2006 (next Wednesday). It is the last quiz before the midterm exam and will cover divide&conquer algorithms and their analysis.

3. The results of quiz#3 were announced, and solutions were distributed.
1. At the beginning of today’s lecture, we continued our study of the integer multiplication problem.

First, the class reviewed the divide and conquer approach we studied last time, which used the formula $xy = 2^n x_1 y_1 + 2^{(n/2)}(x_1 y_0 + x_0 y_1) + x_0 y_0$ after dividing the original inputs $x$ and $y$ into $x=2^{(n/2)} x_1+x_0$ and $y = 2^{(n/2)} y_1y_0$. Based on the Master Theorem, it turns out that no improvement has been achieved compared with the classical method (the elementary school method).
To actually improve the running time, we want to reduce the number of recursive calls in each iteration. In other words, we’d like to reduce the number of products we compute from four to some smaller number.

Since the only three quantities we really need are, according to the formula, \(x_1y_1\), \(x_1y_0 + x_0y_1\) and \(x_0y_0\), the trick we are going to use is to only compute \(b = (x_1 + x_0)(y_0 + y_1)\), \(a = x_0y_0\) and \(c = x_0y_0\). Notice that \(x_1y_0 + x_0y_1\) can be derived from these as \((b-a-c)\). So after \(a\), \(b\) and \(c\) are computed recursively, we can calculate the integer multiplication results as \(xy = c2^n + (b-a-c)2^{n/2} + a\).

The three-line pseudocode of this new version integer multiplication algorithm was written onto the white board.
The first and last line both take $\Theta(n)$, because they simply involve a constant number of additions and shifts of numbers of at most $n$ bits. Because $(x_1+x_0)$ and $(y_0 + y_1)$ have at most $(n/2)+1$ bits, the recursive multiplication steps take at most $T((n/2)+1)$. The recurrence equation is thus $T(n) = \Theta(n) + 3T((n/2)+1)$. For large $n$, the added 1 actually makes no difference in the solution (this is not too difficult to prove), so we instead study the simpler recurrence $T(n) = \Theta(n) + 3T((n/2)+1)$.

We need to compare $f(n) = cn$ with $n^{(\log_2 3)}$. Since $n^{(\log_2 3)} = n^{1.59} > n^{1.1}$, we can choose $\epsilon = 0.1$, and see that we are in case 1 of the Master Theorem. So $T(n) = \Theta(n^{1.59})$. Compared with the classical method’s $\Theta(n^2)$, this is clearly an improvement. Intuitively, it says that to multiply two numbers, one does not really need to compute all bit products.
Next, the class moved to the matrix multiplication problem. The classical method was briefly reviewed. In the case of multiplying two n×n matrices, because there are $n^2$ elements to be calculated and each element needs $\Theta(n)$ scalar multiplications, $\Theta(n^3)$ is the total running time.

Merely writing down the $n^2$ elements will take $\Theta(n^2)$ time, so it is impossible to beat $\Theta(n^2)$ for matrix multiplication. However, between $\Theta(n^3)$ and $\Theta(n^2)$, there is still room for improvement. We use divide and conquer to come up with some improvement. In order to do so, we divide both the input and output matrices into four sub-matrices.
For those blocks, one can easily see that the same multiplication rules apply as for individual elements of a 2x2 matrix, so

\[ R = AE + BF; \]
\[ S = AG + BH; \]
\[ T = CE + DF; \]
\[ U = CG + DH. \]

We can compute \( AE, BF, AG \ldots \) recursively and then add them to obtain \( R, S, T, U \). The addition step takes \( \Theta(n^2) \), so the total running time satisfies \( T(n) = 8T(n/2) + \Theta(n^2) \). Unfortunately, but not surprisingly (since this algorithm is just a re-distribution of the original additions and scalar multiplications), the Master Theorem tells us that \( T(n) = \Theta(n^3) \), so we have not improved the running time yet.
6. An actual improvement will thus require us to solve the problem by computing at most 7 smaller products, and then deducing R, S, T, U from them. The basic idea of deriving such products is akin to what we did for integer multiplication, but the actual choice of products, due to Strassen, is not really obvious, nor is there a good explanation why these particular choices should work, beyond the fact that one can prove they do. The products $P_1, P_2, P_3, P_4, P_5, P_6, P_7$ are given in the picture below, and from them, we can derive the desired matrix entries as follows:

$R = p_5 + p_4 - p_2 + p_6;$

$S = p_1 + p_2;$
\[ T = p_3 + p_4; \]
\[ U = p_5 + p_1 - p_3 + p_7. \]

Those expressions can be verified quite easily with standard arithmetic (even though coming up with them is likely much harder). In class, the expression of R was verified as an example.
Using those formulas and expressions, Strassen’s Algorithm is rather simple. \( P_1, P_2, \ldots, P_7 \) are computed recursively after the division step. Then compute \( R, S, T, U \) using the above expressions.

The correctness of Strassen algorithm directly follows the mathematics verifications.

Both the “divide” and “combine” steps take time \( \Theta(n^2) \), so the total running time is: \( T(n) = 7T(n/2) + \Theta(n^2) \). This time, the Master Theorem tells us \( T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81}) \), which again shows that not all scalar multiplications that seemed necessary at first in fact are.

Improving the running time of matrix multiplication is still a very active topic in algorithm research, not least because matrix multiplication is a core step for many scientific applications, data mining, machine learning, graphics, and other application domains. Currently, the best algorithm,
due to Coppersmith and Winograd, has a running time of $\Theta(n^{2.39})$.

Unfortunately, the constant is huge, so the algorithm is not practical for any inputs. Strassen's algorithm, on the other hand, is quite practical for large enough $n$ (though its constant is also worse than for the default algorithm); depending on the exact parameters of the computer, the break-even point happens somewhere between $n=20$ and $n=100$. Naturally, one can design a hybrid approach, which starts by using Strassen's algorithm for large $n$, and once the problem has been broken into small enough subproblems, switches to the default algorithm. It should also be noted that, while the default and Strassen's algorithm are good choices for dense matrices, if the matrix is sparse (i.e., has mostly zeroes), specialized algorithms for sparse matrices run much faster.

After presenting and discussing Strassen's algorithm, a small-size matrix multiplication problem was done in class. We were able to see how the algorithm works in detail as well as how the bigger constant came back to haunt us for small sizes.
During the last 10 minutes of the lecture, we had a brief introduction to the Quick Sort algorithm. Recall that in Merge Sort, most of the work is done in the combination step, while the division step is trivial. Quick Sort is the opposite way.

Quick Sort first chooses a pivot for each recurrence. It then puts all elements smaller than the pivot in one subarray (on the left), and all elements larger than the pivot in another (on the right). The subarrays themselves are not sorted yet. Afterwards, Quick Sort is called recursively on the left and right sub-arrays respectively, resulting in a completely sorted array without any further combination step.

If $m$ is the index of the pivot element, then $T(n) = T(m-1) + T(n-m+1) + \Theta(n)$. The ideal $m$ should thus be close to $n/2$, in which case the running
time is: \( T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log(n)) \). On the other hand, a poorly chosen \( m \) (for instance, \( m=1 \) or \( m=n \)) would result in a running time of \( \Theta(n^2) \).

Thus, ideally, Quick Sort would want to choose the median for \( m \) in each iteration. Alternatively, picking a pivot randomly (i.e., using a random index) also works well. A bad choice would be to pick the first or last element of the array, as arrays are frequently already partially sorted, and many pivots would thus be close to the smallest or largest element.

With respect to finding the median, it should be noted that there is a divide and conquer algorithm, which will find the median of an array in
time $\Theta(n)$. This algorithm divides the whole array into $n/5$ small arrays, each containing 5 numbers. The median of those small arrays can be calculated in constant time. Then the algorithm calculates the median of all those “baby medians”. Using this median-of-medians $m$, we can return to the size-5 arrays. If the median $m'$ of such a small array is larger than $m$, then the two largest elements of that array can be pruned, whereas if $m'$ is smaller than $m$, then the two smallest elements can be pruned. The algorithm then recursively finds the median on the remaining $3n/5$ elements.

Please refer to the textbook for the details of Quick Sort and median calculation if you are interested.

In the coming lectures, we will move forward to the study of Dynamic Programming.