[Overall Information]

We started the study of dynamic programming in today’s class. An introductory problem of Number Triangle was provided to illustrate the basic idea. After analyzing Number Triangle problem in details, we also studied a more complex problem, Matrix-chain Multiplication.

[Announcements]

1. There will be a quiz (NO.4) the coming Wednesday (03/08/2006). It will cover divide and conquer algorithms and related analysis. Students need to know the Master Theorem, but the memorization Strassen’s algorithm is not required.

2. The midterm is scheduled for 03/22/2006.

To help students prepare for the exam, there will probably be one review session on the afternoon of 03/20/2006 (we will have regular lecture in that day’s morning class). The time and location of the review session had not been decided at lecture time, but it will be 3:30-4:50 in THH 116.

Besides the regular office hours on Monday (03/20/2006), the TA will have one-day-long office hours on Tuesday (03/21/2006) from 9:30am to 1:00pm and from 2:00pm to 4:40pm at SAL311 to answer midterm
related questions. Extra office hours can also be set up through email.

3. There will be another interesting guest talk given by professor Eva Tardos of Cornell University this Thursday (03/09/2006). The topic is “Solution Quality in Routing and Network Formation Games”. Related information can be found at the following link. Interested students are encouraged to attend.


[During the Lecture]

1. In today’s lecture, we started the study of dynamic programming. The general idea of dynamic programming is surprisingly simple. It is kind of similar to exhaustive search at the beginning. But instead of doing the same task again and again, it utilizes the result of former calculations to
achieve a better running time.

The introductory example was the Number Triangle problem. Starting from a group of numbers that is in a triangle formation, our goal is to find a maximum weight path (weight is defined as the sum of numbers along the path) from the starting point to the bottom of the triangle.

It is easy to see that greedily picking the highest available number in each step won't work. At the other extreme is exhaustive search, which tries every possible path, and picks the best. It will certainly work, but be very slow. In each step, there is a left and right choice available. So if the triangle has height \( n \), there are \( 2^n \) paths to consider.
2. Pseudocode of exhaustive search was written onto the whiteboard. As we can see, it is a correct solution but very inefficient. The reason why it is inefficient is that we are actually re-computing the maximum value of many small triangles over and over again with the same result for each time of computation. A natural solution is to store the values somewhere the first time they are computed, and then use those computed values in the future.

3. The pseudocode of this improved search algorithm was given afterwards. $a[i,j]$ represents original triangle values while $b[i,j]$ represents the calculated maximum value from a given point down. We notice that in this algorithm, the maximum value at each point is only calculated once. Since each $b[i,j]$ takes constant time to compute,
the total work takes $\Theta(n^2)$, which is much better than $\Theta(2^n)$.

4. For this simple Number Triangle problem, the reason why dynamic programming works is that the optimal solution to the larger problem always contains the optimal solution to the smaller sub-problems. This “optimality of sub-problems” is the key to dynamic programming algorithms in general.
Following the pseudocode of search(i, j) step by step, we were able to see how the algorithm memorizes the computed values while calculating the smaller sub-problems in a bottom-up manner. Based on the observation that values in lower levels are always computed and stored before the higher levels anyway, a further-improved bottom-up algorithm was given. This algorithm is nearly the same as the previous one except that there is no recursion involved. The total running time (\(\sum \Theta(i)\)) is also \(\Theta(n^2)\).

To prove the correctness, we need to show that \(b[i,j]\) stores the maximum path weight from \([i,j]\) to the bottom. The statement is proved by backward induction on \(i\).

The base case when \(i = n\) is trivial.

For the induction step from \(i\) to \(i-1\), for any \(j\), what the algorithm does is \(b[i, j] = \max\{b[i+1, j], b[i+1, j+1]\} + a[i, j]\). By induction hypothesis, we
have $b[i, j] = \max \{\text{OPT}(i+1, j), \text{OPT}(i+1, j+1)\} + a[i, j]$. Because the optimum gets $a[i, j]$ and then does the better of going left/right, picking up the optimal value there, we have $b[i,j]=\text{OPT}(i, j)$. It should be stressed again that the reasoning for the last step is our insight into the optimal substructure property. This relies implicitly (and crucially) on the fact that it does not matter from where a path arrives – once it is at a certain point, it will always take the best solution from there. This is notably different from what would happen, for instance, if the path was not allowed to repeat any number. Then, which path can be used later would depend heavily on the numbers visited earlier.
Our next problem of today’s class is Matrix-chain Multiplication. Suppose the matrix-chain contains k matrixes $A_1, A_2, \ldots, A_k$. The task here is to compute the product $A_1A_2 \ldots A_k$ efficiently using associativity of matrix multiplication.

To illustrate the importance of choosing the right ordering, a simple example of $X(1*n)$, $Y(n*1)$ and $Z(1*n)$ matrix multiplication was given. It turns out that $(XY)Z$ uses a total of $2n$ multiplications, while $X(YZ)$ takes $2n^2$ multiplications. The order of evaluation is thus very important. Generally, to multiply an $m*k$ matrix with a $k*n$ matrix (not using Strassen's algorithm), the number of scalar multiplication we need to perform is $mkn$. This conclusion was used later in today’s class.
9. How can we find the optimal ordering of Matrix-chain Multiplication?

The first observation is: The optimum must to some multiplication last. Let’s suppose it is \((A_1\ldots A_i)(A_{i+1}\ldots A_k)\), where \(1 \leq i < k\). Then, if the optimal solution needs the product \((A_1\ldots A_i)\), it will compute it optimally.
Let $OPT(l, r)$ represent the number of multiplications the optimum ordering needs in order to compute $A_l \ldots A_r$. Suppose the first associative partition happens at index $i$. Then we have $OPT(l, r) = OPT(l, i) + OPT(i+1, r) + m_i n_i n_r$.

Although we don’t know exactly this index $i$, what we do know is $i$ must be chosen in a way that minimizes $OPT(l, r)$. We will continue to talk about Matrix-chain Multiplication problem in next lecture.