[Overall Information]

In today’s lecture, we summarized the algorithms we have studied before as greedy algorithms and began the study of a new class of algorithms: divide and conquer algorithms.

The first algorithm we studied using divide and conquer is Merge Sort. We proved the correctness of Merge Sort and began its running time analysis.

[Announcements]

1. The quiz#3 will take place the coming Monday at the beginning of the class. The questions of this quiz, which will be similar to homework questions as usual, will cover Minimum Spanning Tree problem and Shortest Path problem.

2. The solution to homework set 3 was distributed after today’s class. Those who did not get the solution can pick it up either at the professor’s or TA’s office during proper office hours.

3. The midterm exam of CS303 is scheduled on 03/22/2006 in class time.

4. There will be another interesting guest talk this Friday 02/24/2006 starting from 11:00 am at GFS 101.
[During the Lecture]

1. After the class began, we quickly reviewed all the problems and algorithms we studied so far in this course. All of them can be classified as “greedy algorithms”, which deal with a small part of the problem in each step and never undo or modify what has been done before. Such algorithms are frequently simple but unfortunately, also frequently don’t work.

The coming few lectures will talk about “divide and conquer algorithms”. An informal description of the “divide and conquer” approach is: divide the whole problem into several smaller-size problems and run the same algorithm on each small problem. After that, combine the solutions to the smaller problems into a solution for the bigger one.
2. The formal outline was also written onto the whiteboard. The small problems are represented by $I_1, I_2 \ldots I_k$ ($k \geq 2$). They may or may not overlap with each other (though frequently, they don't).
As a specific example of the “divide and conquer” strategy, we studied Merge Sort. Suppose the array to be sorted is \( a[l, \ldots, r] \), where \( l \) and \( r \) are left-most and right-most elements.

If \( l = r \), the algorithm is done with the sorting.

If \( l \neq r \), let \( m = \lfloor (l+r)/2 \rfloor \) be the middle of the array. Then recursively call the same Merge Sort algorithm on both \( a[l, \ldots, m] \) and \( a[m+1, \ldots, r] \). If we get the implementation right for the smaller instances, then the result of these two function calls will be two sorted sub-arrays. Then, combine the two sorted subarrays by calling \( \text{merge}(a, l, m, r) \).

What \( \text{merge}(a, l, m, r) \) does is the following:

We start from two pointers: \( i=0 \) and \( j=r-m \). Each of them will scan a sub-array and compare the two elements they are pointing to in each step. Based on the comparison result, copy the smaller element (or the element still left in an array if the other array has been scanned out) to form a new merged and sorted array.
4. A specific example (array: 4, 2, 1, 5, 3, 8, 6, 7) was provided afterwards in order to illustrate the idea. We were able to see how the algorithm works step by step. For example, when the two sub-arrays are \{2, 4\} and \{1, 5\} respectively, at the beginning, the two pointers are pointing to 2 and 1, respectively. Because 1 is smaller, it is copied into the new array and the second pointer was increased by 1 and consequently points to 5. Then, comparing 2 with 5, 2 is copied and the first pointer is increased by 1 etc.

The merge() continues until the two sorted sub-arrays are merged into one sorted array. Merge() works in this way every time the Merge Sort
algorithm calls it, so finally, the whole array can be sorted.

5. Since the Merge Sort algorithm is recursive, it seems that in order to prove the algorithm sort the whole array, we need to assume that it successfully sort the two sub-arrays. That is exactly the idea of induction, and we will be able to prove correctness by induction.

The correctness claims of Merge Sort are:

(1) The array contains the same elements before and after the Merge Sort;

(2) The array after calling Merge Sort is sorted.
The base case which contains only one element is obviously correct.

Assume Merge Sort is correct for all array sizes $k < n$. We want to prove that the algorithm also works for an array with size $n$. By induction hypothesis (the hypothesis can be used because $(m-1)<n$ and $(r-(m+1))<n$), when merge() is called, the two half-sized arrays are sorted and contain the same elements as before.

6. The Merge function invoked for combining the two subarrays is actually a greedy algorithm (while Merge-Sort is not). The claim that merge() will merge two sorted arrays into one sorted big array and maintain the same elements as before was proved by induction.

For the induction step, we’d like to prove that the copied elements $b[i+j]$ is no smaller than $b[i+j-1]$. Without loss of generality, we can assume
that $b[i+j]$ comes from the left half array.

(1) If $b[i+j-1]$ also comes from the left half, because the left half array is already sorted and $b[i+j]$ is the direct successor of $b[i+j-1]$ in the left half array, we have $b[i+j-1] \leq b[i+j]$.

(2) If $b[i+j-1]$ comes from the right half, then $j$ was increased after writing $b[i+j-1]$. Thus, according to the merge algorithm, before copying $b[i+j-1]$, we must have compared what ended up being $b[i+j]$ with $b[i+j-1]$ and found $b[i+j-1] \leq b[i+j]$ (that is the reason why $b[i+j-1]$ was copied earlier).

The merged array still contains the same elements because the “while” loop copies each element exactly once. So we have proved that the merge function is correct, and because it is, so is Merge Sort.

Proof by induction on $i + j$ that $b[i + j]$ contains the same elements from $a$.

Base case ($i = j = 0$): $b[0] = a[0]$.

Inductive step ($i + j = 0$): Prove that element copied into $b[i+j]$.

Assume that $b[i+j] = a[c + j - 1]$. Then, either $b[i+j-1] = a[c + (j-1)]$ copied from left, or $b[i+j-1] = a[c + j]$ copied from right.

In the former case, $b[i+j-1] \leq b[i+j]$ because left half of $a$ was sorted by induction hypothesis. In the other case, $j$ was increased after writing $b[i+j-1]$, but $i$ was not. So $a[c + j]$ was compared with $a[j + m]$ right border choosing $a[c + j]$ for $b[i+j-1]$, and $a[j + m]$ was found to be smaller.

So sorted.
7. If we try to see how Merge Sort fits into the “divide and conquer” framework, we notice that the “combine” step is to call merge(), which does most of the work. In contrast, the divide step (computing \( m = \lfloor (l+r)/2 \rfloor \)) is rather simple. In the coming lectures, we will see other algorithms which do more work when dividing the problem than combining the result.

8. At the end of today’s lecture, we began the running time analysis of Merge Sort. Let \( T(n) \) represent the running time of Merge Sort on an array of size \( n \). We have:

\[
T(1) = \text{constant};
\]

\[
T(n) = cn + T([n/2]) + T([n/2]);
\]
Thus, $T(n)$ for larger values of $n$ is defined in terms of $T(n')$ for smaller values of $n'$. This is called a recursive definition (or recurrence relation) for $T$. Our goal in the following lectures is to find out a closed-form solution for $T$.

One possible way is to guess a function $T$ first, and then prove that it satisfies the recurrence using induction. However, guessing is not always easy, so we will learn some recipes for guessing well, and a general theorem, in the next lecture.

$T(n)$, many line of MergeSort on an array of size $n > 0$.

$T(1) = 1$.

$T(n) = \left\{ \begin{array}{ll}
\frac{n}{2} & \text{divide} \\
T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) & \text{merge, etc.}
\end{array} \right.$

Recursive definition of $T$, recurrence relation on $T$.

Want to calculate which $T$ solves these equations.

Method 1: Guess $T$; prove it works by induction.