ESO211 (Data Structures and Algorithms): Lecture Notes Set 3

Shashank K Mehta

1 Divide and Conquer Paradigm

This is a very general approach to solve a computing problem. Given a problem instance with size parameter n. The devide and conquer approach involves three steps: (i) Decompose: identify one or more instances of the same problem with smaller parameters; (ii) Solve: solve the smaller problems using the same approach (i.e., make recursive calls); (iii) Assemble: construct the solution of the given problem from the solutions of the smaller problems. This approach works for a large variety of problems. Here we will give two examples.

1.1 Merge Sort

Given a sequence of integers in an array $A[0:n-1]$ sort them in non-decreasing order. Suppose our general problem is to sort the subsequence $A[i:j]$. We first sort the subsequence i to r and also sort the subsequence $r+1$ to j. Then we combine the two sorted sequences into the sorted sequence of range i to j. Here we select r to be the mid point between i and j.

```
$r := \lceil (i+j)/2 \rceil - 1$;
$Mergesort(A[i:r])$;
$Mergesort(A[r+1:j])$;
$Merge(A[i:r], A[r+1:j])$;
```

Algorithm 1: $Mergesort(A[i:j])$

The last step of the above algorithm takes two distinct sorted sequences and *merges* them into a single sorted sequence. Let the sorted sequences be in the ranges $a:b$ and $b+1:c$. The merge process is described below.

First let us analyse the $Merge$ algorithm. The while-statement has a structure in which there are four sets of three statments. In each iteration exactly one of these is executed. And in each case s is incremented once. Hence total number of assignment and comparasion steps executed is at most a constant times of $c-b+1$. Hence the cost of $Merge(A[a:b], A[b+1:c])$ is $O(c-a+1)$.

To analyse the time complexity of $Mergesort$, first assume that n is a power of 2. Then the time complexity is given by the $T(n) \leq 2T(n/2) + cn$. The solution of this recurrence relation gives the time complexity to be $O(n \cdot \log_2 n)$. In case n is not a power of 2. Let n' be the nearest power of 2 greater than n. So $n' \leq 2n$. As $T()$ denote the worst cost, it is a non-decreasing function. So $T(n)$ must be bounded above by $T(n') \leq c.n' \cdot \log_2 n'$. So $T(n)$ is at most $c.(2n)(\log_2 n + 1)$. Clearly $n \cdot \log_2 n \geq n$, so $T(n)$ is at most $c'.n \cdot \log n)$. Hence $T(n)$ is $O(n \cdot \log_2 n)$ for all n.

```
$p := a$;
$q := b + 1$;
$s := a$;
while $p \leq b$ or $q \leq c$ do
    if $q > c$ then
        while $p \leq b$ do
            $B[s] := A[p]$;
            $s := s + 1$;
            $p := p + 1$;
        end
    else
        if $p > b$ then
            while $q \leq c$ do
                $B[s] := A[q]$;
                $s := s + 1$;
                $q := q + 1$;
            end
        else
            if $A[p] \leq A[q]$ then
                $B[s] := A[p]$;
                $s := s + 1$;
                $p := p + 1$;
            else
                $B[s] := A[q]$;
                $s := s + 1$;
                $q := q + 1$;
            end
        end
    end
end
for $s := a$ to $c$ do
    $A[s] := B[s]$;
end
```

Algorithm 2: $Merge(A[a : b], A[b + 1 : c])$

1.2 Multiplication of two n-bit integers

Assume that $n = 2^k$. Let a and b be n-bit integers. Let $a = a_0 + 2^{n/2}a_1$ and $b = b_0 + 2^{n/2}b_1$. Then $a.b = a_0b_0 + 2^{n/2}(a_0b_1 + a_1b_0) + 2^n a_1b_1$.

Let $s_0 = a_0 + b_0$ and $s_1 = a_1 + b_1$. Then $a_0b_1 + a_1b_0 = s_0.s_1 - a_0b_0 - a_1b_1$. so $a.b = a_0b_0 + (s_0s_1 - a_0b_0 - a_1b_1)2^{n/2} + a_1b_1.2^n$.

So the multiplication of two n-bit number has been reduced two multiplications of $n/2$-bit numbers and one $n/2 + 1$-bit number multiplication and four $n/2$-bit additions. An $n/2 + 1$ bit multiplication can be reduced to one $n/2$-bit multiplication and two $n/2$-bit additions.

1.3 Matrix mutiplication

Product of two $n \times n$ matrices requires $O(n^3)$ time. Here we will describe an algorithm by Strassen with improved performance.

1.3.1 Matrix Size

In general an $x \times y$ matrix can be multiplied to an $y \times z$ matrix in $O(xyz)$ time. But using three parameters is not desirable. Since we can always embed any $x \times y$ matrix into a square matrix $n \times n$, with $x \leq n$ and $y \leq n$, by adding $n - y$ zero-columns at the right-end and adding $n - x$ zero-rows at the bottom. Observe that by embedding the two matrices in square matrices and then performing the matrix multiplication also results in the product matrix embedded in the square matrix. Hence there is no loss of generality in resticting our attention to only square matrices.

In our discussion we will reduce the problem of multiplying two $n \times n$ matrices into certain number of instances of multiplication of $n/2 \times n/2$ matrix with another $n/2 \times n/2$. Hence it is desirable to have $n = 2^k$. If n is not a power of 2, then we can find the nearest power of 2 greater than or equal to n, say n', and embed the $n \times n$ matrix into $n' \times n'$. Note that $n' < 2n$.

1.3.2 Reduction

Let us have two $n \times n$ matrices. Suppose their respective sub-matrices be as follows

$$\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline\end{array} \qquad \text{and} \qquad \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline\end{array}$$

Then the submatrices of the product will be $AE + BG$, $AF + BH$, $CE + DG$, and $CF + DH$. If we simply use this reduction then we get following recurring relation: $T(n) = 8T(n/2) + 4(n/2)^2$. Taking $n = 2^k$, we get $T(n) = n^2(2n - 1)$. This does not give any improvement.

1.3.3 Strassen's approach

Strassen observed the following identity. Let

$$P_1 = A(F - H),$$

$$P_2 = (A + B)H,$$

$$P_3 = D(-E+G),$$
$$P_4 = (C+D)E,$$
$$P_5 = (A+D)(E+H),$$
$$P_6 = (B-D)(G+H),$$
$$P_7 = (-A+C)(E+F).$$

Then we have

$$AE+BG = P_3+P_5+P_6-P_2,$$
$$AF+BH = P_1+P_2,$$
$$CE+DG = P_3+P_4, \texttt{ and}$$
$$CF+DH = P_1+P_5+P_7-P_4.$$

This reduction leads to an efficient algorithm because it has only seven matrix multiplication problems of $n/2 \times n/2$ size.

1.3.4 Time Complexity

This reduction leads to the recurrence relation for time complexity $T(n) = 7T(n/2) + 18(n/2)^2$. This solves to $T(n) = O(n^{2.8075})$ or $O(n^{log_2 7})$.

1.4 Master's Method for Solving Recurrence Relations

Theorem 1 (Master's) *Let $a \geq 1$ and $b > 1$ be real numbers and let $f()$ be a non-decreasing function. Consider the recurrence relation $T(n) = aT(n/b) + f(n)$, where $n > 0$ is a power of b, i.e., $n = b^k$ for some positive integer k. Then*

(i) If $f(n) = \Theta(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(ii) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.

(iii) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $a.f(n/b) \leq c \cdot f(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.

Proof We are given that $n = b^k$ for some integer k. In this case repeated application of the relation gives $T(n) = a^k T(1) + f(n) + af(n/b) + a^2 f(n/b^2) + \ldots a^{\log_b n} f(1)$. But $a^k = n^{\log_b a}$. So $T(n) = T(1)n^{\log_n a} + f(n) + af(n/b) + a^2 f(n/b^2) + \ldots a^{\log_b n} f(1)$. For convenience let $h(n)$ denote $f(n) + af(n/b) + a^2 f(n/b^2) + \ldots a^{\log_b n} f(1)$. Now we will consider each of the three cases separately.

(i) Let $f(x) = c.x^{\log_b a - \epsilon}$. In that case $h(n) = c.n^{\log_b a - \epsilon}(1 + a/b^{\log_b a - \epsilon} + (a/b^{\log_b a - \epsilon})^2 + \cdots + (a/b^{\log_b a - \epsilon})^{\log_b n}$. Since $a/b^{\log_b a - \epsilon} = b^\epsilon$, $h(n) \leq c.n^{\log_b a - \epsilon}(b^{(k+1)\epsilon} - 1)/(b^\epsilon - 1)$. So $h(n) = c.n^{\log_b a} n^{-\epsilon}(b^{\epsilon(k+1)} - 1)/(b^\epsilon - 1)$.

This simplifies to $h(n) = c.n^{\log_b a}(b^\epsilon - 1/n^\epsilon)/(b^\epsilon - 1)$. Thus $c.n^{\log_b a} \leq h(n) \leq (c/(b^\epsilon - 1)).n^{\log_b a}$, that is, $h(n) = \Theta(n^{\log_b a})$. So $T(n) = T(1)n^{\log_n a} + h(n) = \Theta(n^{\log_b a})$.

(ii) Let $f(x)\Theta(c.x^{\log_b a})$, which is equivalent to $c_1.n^{\log_b a} \leq T(n) \leq c_2.n^{\log_b a}$. In that case $h(n) = \Theta(n^{\log_b a})(1 + a/b^{\log_b a} + (a/b^{\log_b a})^2 + \cdots + (a/b^{\log_b a})^{\log_b n}$. So $h(n) = \Theta(n^{\log_b a})(1 + 1 \cdots + 1) = \Theta(n^{\log_b a}).\log_b n = \Theta(n^{\log_b a} \log_b n)$. So $T(n) = n^{\log_b a} T(1) + f(n) = \Theta(n^{\log_b a} \log_b n)$.

(iii) We have $f(x) \geq c.x^{\log_b a + \epsilon}$ and $af(n/b) \leq c.f(n)$ for some constant $c < 1$. In this case $h(n) = f(n) + a.f(n/b) + a^2.f(n/b^2) + \cdots \leq f(n) + c.f(n) + c^2 f(n) + \cdots \leq (1/(1-c))f(n)$. Hence we have $f(n) \leq T(n) \leq n^{\log_b a} T(1) + c'.f(n)$. We are given that $f(n)\Omega(n^{\log_b a + \epsilon}$ so $T(n) = \Theta(f(n))$.
■

Next we will consider general n.

1.5 Master's Theorem for general n

Theorem 2 (Master's) *Let $a \geq 1$ and $b > 1$ be real numbers and let $f()$ be a non-decreasing function. Consider the recurrence relation $T(n) = aT(n/b) + f(n)$, where n/b may either denote $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. Then*

(i) If $f(n) = \Theta(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(ii) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.

(iii) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $a.f(n/b) \leq c \cdot f(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.

Hint: To prove this theorem, use the fact that the nearest power of b greater than or equal to n is less than $b.n$. Similarly the nearest power of b less than or equal to n is greater than n/b.