Divide and Conquer Algorithms (Continued)

Example 1 (Integer multiplication). *Multiply two large integers x, y, given as sequences of bits $x_0, x_1, ..., x_{n-1}$ and $y_0, y_1, ..., y_{n-1}$ (low-order bit first, i.e., $x = x_{n-1}...x_1x_0$ in binary, and similarly for y).*

Ans: Before we describe a divide and conquer algorithm to do integer multiplication, let's see how a conventional iterative algorithm solves this problem.

Iterative algorithm: Multiply x by each bit of y, shift appropriately, then add the n results to each other.
The running time of this algorithm is $\Theta(n^2)$ (n additions of up to $2n$ bits each). Let's see if we can do better with a divide and conquer approach.

Idea 1: For simplicity, assume n is a power of 2. Let $X_0 = x_{\frac{n}{2}-1} \cdots x_1x_0$ and $X_1 = x_{n-1} \cdots x_{\frac{n}{2}}$ denote the least significant half and most significant half of binary representation of x, respectively. Define Y_0 and Y_1 similarly. Then, $x = 2^{\frac{n}{2}}X_1 + X_0$ and $y = 2^{\frac{n}{2}}Y_1 + Y_0$. We can write

$$xy = 2^n X_1Y_1 + 2^{\frac{n}{2}}X_1Y_0 + 2^{\frac{n}{2}}X_0Y_1 + X_0Y_0$$

Do you see any pattern? The original problem (compute xy) is reduced to four subproblems of half size (compute X_1Y_1, X_1Y_0, X_0Y_1, X_0Y_0), together with some "shift" operations (multiplication by power of 2) and binary additions. Shift operations and binary additions can be done in linear time– i.e., $O(n)$ time. This yields the following recursive algorithm.

```
function MULTIPLY1(x, y, n)                         ▷ x, y are lists of size n
    if n = 1 then
        return x × y                                ▷ multiplication of 1-bit numbers
    else
        Define lists X1, X0, Y1, Y0 as explained above
        p1 = MULTIPLY1(X1, Y1, n/2)
        p2 = MULTIPLY1(X1, Y0, n/2)
        p3 = MULTIPLY1(X0, Y1, n/2)
        p4 = MULTIPLY1(X0, Y0, n/2)
        return 2^n p1 + 2^(n/2) p2 + 2^(n/2) p3 + p4
    end if
end function
```

Now, let's look at the running time of this algorithm? The recurrence relation for worst-case runtime $T(n)$ is:

$$T(n) = \begin{cases} c & n = 1 \\ 4T(\frac{n}{2}) + \Theta(n) & \text{for } n > 1 \end{cases}$$

where $4T(\frac{n}{2})$ is the time that is required to execute the four recursive calls, and $\Theta(n)$ is the time that algorithm spends on performing shifts and binary additions (in addition to initial splitting of input into sublists).

We can apply the Master Theorem to $T(n)$, with $a = 4$, $b = 2$, $d = 1$. Because $a = 4 > 2 = b^d$, we have $T(n) = \Theta(n^{log_2 4}) = \Theta(n^2)$– by third case of Master Theorem. But this is No better than simple iterative algorithm! Should we give up?

Idea 2: If we can improve a such that $\log_b a$ becomes smaller, we can do better than $\Theta(n^2)$– even if $a < b^d$. To decrease a, we need fewer recursive calls– i.e., fewer multiplications. Notice that

$$(X_1 + X_0)(Y_1 + Y_0) = X_1Y_1 + X_1Y_0 + X_0Y_1 + X_0Y_0.$$

This is almost correct expression, except for shifts, and it involves only 1 multiplication instead of 4. Because terms X_1Y_0 and X_0Y_1 shift by same amount, we can use this to save one recursive call:

$$xy = 2^n X_1Y_1 + X_0Y_0 + 2^{\frac{n}{2}}((X_1 + X_0)(Y_1 + Y_0) - X_1Y_1 - X_0Y_0)$$

This yields following recursive algorithm:

function MULTIPLY2(x, y, n)
    if $n = 1$ **then**
        return $x \times y$
    else
        set lists X_1, X_0, Y_1, Y_0 as explained above
        $p_1 = \text{MULTIPLY2}(X_1, Y_1, \frac{n}{2})$
        $p_2 = \text{MULTIPLY2}(X_1 + X_0, Y_1 + Y_0, \frac{n}{2} + 1)$
        $p_3 = \text{MULTIPLY2}(X_0, Y_0, \frac{n}{2})$
        return $2^n p_1 + 2^{\frac{n}{2}}(p_2 - p_1 - p_3) + p_3$
    end if
end function

The worst-case running time $T'(n)$ of this algorithm satisfies:

$$T'(n) = \begin{cases} c & n = 1 \\ 3T'(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

Remark. *This recursive relation is not exact because, depending on the value of n, the recursive calls can be on input sizes of $\lfloor \frac{n}{2} \rfloor$ or $\lceil \frac{n}{2} \rceil$. Moreover, the second recursive call for p2 is on input size $\frac{n}{2} + 1$. But as we saw in the proof of Master Theorem, these issues does not affect the final answer.*

Remark. *The constant hidden by the term $\Theta(n)$ is larger than for the first recursive algorithm (we perform more binary additions)*

What is the running time of the new algorithm? The Master Theorem still applies but with $a = 3$, $b = 2$, $d = 1$. Since $a > b^d$, the third case of Master Theorem yields $T'(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\cdots})$, which is strictly better than $\Theta(n^2)$. □

Algorithm Correctness

We say a program is correct if it produces a correct output on every acceptable input. In order to specify what are the acceptable inputs of a program and what are the correct outputs for each acceptable inputs, we use preconditions and postconditions.

Precondition: Statement specifying what conditions must hold *before* an algorithm is executed (i.e., describes valid inputs).

Postcondition: Statement specifying what conditions hold *after* an algorithm executes (i.e., describes expected output).

Remark. *In general, we want the weakest reasonable precondition (i.e., put as few constraints as possible, only specify what is strictly necessary) and strongest reasonable postcondition (i.e., specify as much as possible).*

Algorithm correctness with respect to specific preconditions and postconditions is usually broken down into two components:

1. **Termination:** If preconditions hold before execution, then algorithm eventually finishes executing
2. **Partial Correctness:** If preconditions hold before execution, then postconditions hold after execution

Recursive Algorithms:

We usually prove termination and partial correctness of recursive algorithms, by induction on size of input. The induction proof techinque matches the recursive structure of algorithms.

Example 2 (Binary search algorithm)**.** *Consider the following recursive implementation of binary search algorithm:*

```
function RecBSearch(x, A, s, f)
    if s == f then
        if x == A[s] then
            return s
        else
            return −1
        end if
    else
        m = (s + f) / 2                                  ▷ Integer Division
        if x ≤ A[m] then
            return RecBSearch(x, A, s, m)
        else
            return RecBSearch(x, A, m + 1, f)
        end if
    end if
end function
```

Precondition:

1. Elements of A comparable with each other and with x
2. Assume array indices start at 0 and hence $0 \le s \le f < length(A)$
3. Array A issorted in nondecreasing order $(A[s] \le \cdots \le A[f])$

Postcondition: RecBSearch(x,A,s,f) terminates and returns index p such that:

1. $s \le p \le f$ or $p = -1$
2. If $s < p$, then $A[p-1] < x$
3. If $s \le p \le f$, then $x = A[p]$