Divide and Conquer Algorithms (Continued)

**Example 1** (Integer multiplication). Multiply two large integers \(x, y\), given as sequences of bits \(x_0, x_1, \ldots, x_{n-1}\) and \(y_0, y_1, \ldots, y_{n-1}\) (low-order bit first, i.e., \(x = x_{n-1} \cdots x_1 x_0\) 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 2\(n\) 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 \(X_0 = x_{n-1} \cdots x_1 x_0\) and \(X_1 = x_{n-1} \cdots x_2\) 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

\[
xy = 2^n X_1 Y_1 + 2^\frac{n}{2} X_1 Y_0 + 2^\frac{n}{2} X_0 Y_1 + X_0 Y_0
\]

Do you see any pattern? The original problem (compute \(xy\)) is reduced to four subproblems of half size (compute \(X_1 Y_1\), \(X_1 Y_0\), \(X_0 Y_1\), \(X_0 Y_0\)), 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.

```
1: function MULTIPLY1(x, y, n)
2:     if n = 1 then
3:         return x × y
4:     else
5:         Define lists \(X_1, X_0, Y_1, Y_0\) as explained above
6:         \(p_1 = MULTIPLY1(X_1, Y_1, \frac{n}{2})\)
7:         \(p_2 = MULTIPLY1(X_1, Y_0, \frac{n}{2})\)
8:         \(p_3 = MULTIPLY1(X_0, Y_1, \frac{n}{2})\)
9:         \(p_4 = MULTIPLY1(X_0, Y_0, \frac{n}{2})\)
10:        return \(2^n p_1 + 2^\frac{n}{2} p_2 + 2^\frac{n}{2} p_3 + p_4\)
11:     end if
12: end function
```

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

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

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

\[
(X_1 + X_0)(Y_1 + Y_0) = X_1 Y_1 + X_1 Y_0 + X_0 Y_1 + X_0 Y_0.
\]

This is almost correct expression, except for shifts, and it involves only 1 multiplication instead of 4. Because terms \(X_1 Y_0\) and \(X_0 Y_1\) shift by same amount, we can use this to save one recursive call:

\[
xy = 2^n X_1 Y_1 + X_0 Y_0 + 2^\frac{n}{2} ((X_1 + X_0)(Y_1 + Y_0) - X_1 Y_1 - X_0 Y_0)
\]
This yields the following recursive algorithm:

1:  function Multiply2(x, y, n)
2:      if n = 1 then
3:          return x \times y
4:      else
5:          set lists X_1, X_0, Y_1, Y_0 as explained above
6:          p_1 = Multiply2(X_1, Y_1, \frac{n}{2})
7:          p_2 = Multiply2(X_1 + X_0, Y_1 + Y_0, \frac{n}{2} + 1)
8:          p_3 = Multiply2(X_0, Y_0, \frac{n}{2})
9:          return 2^n p_1 + 2^{\frac{n}{2}} (p_2 - p_1 - p_3) + p_3
10:     end if
11:  end function

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

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

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 $p_2$ 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 $T'(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58...})$, 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 technique matches the recursive structure of algorithms.

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

```java
1: function RecBSearch(x, A, s, f)
2:     if s == f then
3:         if x == A[s] then
4:             return s
5:         else
6:             return -1
7:     end if
8:     else
9:         m = (s + f) / 2 ▶ Integer Division
10:        if x ≤ A[m] then
11:            return RecBSearch(x, A, s, m)
12:        else
13:            return RecBSearch(x, A, m + 1, f)
14:        end if
15:     end if
16: end function
```

Precondition:

1. Elements of A comparable with each other and with x
2. Assume array indices start at 0 and hence 0 ≤ s ≤ f < length(A)
3. Array A is sorted in nondecreasing order (A[s] ≤ ⋯ ≤ A[f])

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

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