Binary Multiplication

Q: How do we *multiply* two numbers?

eg.

$$
\begin{array}{lr}
 & 12,345 \\
\times & 6,789 \\
\hline
 & 0 \\
 & 00 \\
+ & 000 \\
\hline
 & 83,810,205
\end{array}
\qquad\qquad
\begin{array}{lr}
 & 10111 \\
\times & 10101 \\
\hline
 & 0 \\
 & 00 \\
 & 000 \\
+ & 0000 \\
\hline
\end{array}
$$

Consider the following algorithm to multiply two *binary* numbers.

```
PRECONDITION:
  x and y are binary bit arrays.
POSTCONDITION:
  Returns result a binary bit array equal to
  the product of x and y.

def MULTIPLY(x, y):
  result = [0];
  for i in range(len(y)-1, -1, -1):
    if y[i] == 1:
      result = BINARY_ADD(result, x)
    x.append(0) #add a 0 to the end of x
  return result
```

Q: If we *measure complexity* by the number of *bit operations*, what is the *worst case complexity* of MULTIPLY?

Q: Is there a *more efficient* way to implement the multiplication?

Divide and Conquer Multiplication

Notice that

$$010111 = 010000 + 111 = 010 \cdot 2^{n/2} + 111$$

$$011101 = 011000 + 101 = 011 \cdot 2^{n/2} + 101$$

i.e., we can *split* a binary number into $n/2$ *high bits* and $n/2$ *low bits*.

Q: What is 010111×011101 written in terms of *high bits* and *low bits*?

Q: What is the complexity of multiplying a number x by $2^{n/2}$?

In general, $x \times y$ in terms of *low bits* $\{x_l, y_l\}$ and *high bits* $\{x_h, y_h\}$ is:

$$x \times y =$$

So we can define a *recursive, divide and conquer, multiplication* algorithm.

```
PRECONDITION:
  x and y are both binary bit vectors of length n
(n a power of 2).
POSTCONDITION:
  Returns a binary bit vector equal to
  the product of x and y.

REC_MULTIPLY( x,  y):
  if (len(x) == 1):
      return ([x[0]*y[0]])

   xh = x[n/2:n]
   xl = x[0:n/2]
   yh = x[n/2:n]
   yl = x[0:n/2]

   a = REC_MULTIPLY(xh, yh)
   b = REC_MULTIPLY(xh, yl)
   c = REC_MULTIPLY(xl, yh)
   d = REC_MULTIPLY(xl, yl)

  b = BINARY_ADD(b,c)
  a = SHIFT(a,n)
  b = SHIFT(b,n/2)
 return BINARY_ADD(a,b,d)
end REC_MULTIPLY
```

Q: What is the *recurrence relation* for the *complexity* of REC_MULTIPLY?

$T(n) =$

Q: What is the *worst case complexity* of REC_MULTIPLY?

This is a bit disappointing...

A Better Divide and Conquer Multiplication Algorithm

Recall we want to compute:

$$x_h y_h \cdot 2^n + (x_l y_h + x_h y_l) \cdot 2^{n/2} + x_l y_l$$

Observation [Gauss]

$$x_l y_h + x_h y_l = (x_h + x_l)(y_h + y_l) - x_h y_h - x_l y_l$$

Q: *Why is this true?*

$$(x_h + x_l)(y_h + y_l) =$$

Q: How does this help us?

1.

2.

3.

Therefore,

$$xy =$$

leading to a new *divided and conquer multiplication* algorithm:

Recursive Multiply – Take 2

```
PRECONDITION:
  x and y are both binary bit arrays of length n,
n a power of 2.
POSTCONDITION:
  Returns a binary bit array equal to
  the product of x and y.

 REC_MULTIPLY_2( x, y):
  if (len(x) == 1):
     return (x[0]*y[0])

  xh = x[n/2:n]
  xl = x[0:n/2]
  yh = x[n/2:n]
  yl = x[0:n/2]

   p1 = REC_MULTIPLY_2(xh, yh)
   p2 = REC_MULTIPLY_2(xh+xl, yh+yl)
   p3 = REC_MULTIPLY_2(xl, yl)

  p2 = BINARY_ADD(p2,-p1, -p3)
  p2 = SHIFT(p2, n/2)
  p1 = SHIFT(p1,n)
  return BINARY_ADD(p1,p2,p3)
```

Q: What is the *recurrence relation* for **REC_MULTIPLY_2**?

$T(n) =$

$T(1) =$

Q: Is this really any better than $T(n) = 4T(n/2) + \mathcal{O}(n)$?

A: See Assignment 1!

Program Correctness – Chapter 2

Proving *program correctness* really means proving

> If some *condition P* holds at the *start* of the execution of a program, then
>
> - the program will *terminate*
> - some *condition Q* will hold at the end.

Condition P is called a *precondition*.

Condition Q is called a *postcondition*.

Think of this as a contract, if the *precondition is satisfied* then the progam is required to *meet the postcondition*.

Note: we are not concerned with *runtime errors* (e.g. overflow, division by zero). They are easier to spot.

Two cases we will consider:

- recursive programs (programs with recursive methods)
- iterative programs (programs with loops)

The Correctness of Recursive Programs

Read the book, pages 47–53.

In this section, we consider how to prove correct programs that contain *recursive methods*.

We do this by using

> *simple or complete induction* over the *arguments* to the recursive method.

How to do the proof

To prove a recursive program correct (for a given precondition and a postcondition) we typically

1. prove the recursive *method* totally correct

2. prove the main *program* totally correct

A first example

```
public class EXP
{
  int EXPO(u,v){
    if v == 0 return 1;
    else if v is even
        return SQUARE(EXPO(u,v DIV 2));
    else
        return u*(SQUARE(EXPO(u,v DIV 2)));
  }
  int SQUARE(x){
    return x*x;
  }
  void main(){
    z = EXPO(x,y);
  }
}
```

*Note DIV truncates the decimals

Note: the *main program* here does nothing but call the method on x and y

Lemma: For all $m, n \in \mathbb{N}$, the method **EXPO**(m, n) terminates and returns the value m^n.

Proof: *next page*

Theorem: The program *EXP* is (totally) correct for *precondition* $x, y \in \mathbb{N}$ and *postcondition* $z = x^y$.

Proof: immediate from the lemma.

Lemma: For all $m, n \in \mathbb{N}$, the method **EXPO**(m, n) terminates and returns the value m^n.

Proof: We prove by *complete induction* that $P(n)$ holds for all $n \in \mathbb{N}$, where $P(n)$ is

Assume that $n \in \mathbb{N}$, and that $P(i)$ holds for all $i \in \mathbb{N}$, $0 \leq i < n$.

So, we have that

(*)

To prove $P(n)$, there are three cases to consider:

Case 1: $n = 0$.

For any m, **EXPO**$(m, 0)$ terminates and returns $1 = m^0$.

Case 2: $n > 0$ n *is odd*.

From the code, for any m, when $n > 0$ and n is *odd*,

- **EXPO**(m, n) works by first calling **EXPO**$(m, n$ *DIV* $2)$,
- then calling **SQUARE** on the result,
- and finally *multiplying* that result by m.

Q. *Why is this correct?*

Case 2 cont. Since n is odd,

So we can apply (*),

The method **SQUARE** always terminates, and *returns*

Therefore, **EXPO**(m, n) terminates and *returns*

Case 3: $n > 0$ n is even.

similar to previous case

We conclude that the lemma holds for all n and m. ■

Recall our recursive multiply algorithm:

```
PRECONDITION:
  x and y are both binary bit vectors of length n
(n power of 2).
POSTCONDITION:
  Returns a binary bit vector equal to
  the product of x and y.

REC_MULTIPLY(x, y):
  if (len(x) == 1):
      return ([x[0]*y[0]])

   xh = x[n/2:n]
   xl = x[0:n/2]
   yh = x[n/2:n]
   yl = x[0:n/2]

   a  = REC_MULTIPLY(xh, yh)
   b  = REC_MULTIPLY(xh, yl)
   c  = REC_MULTIPLY(xl, yh)
   d  = REC_MULTIPLY(xl, yl)

  b  = BINARY_ADD(b,c)
  a  = SHIFT(a,n)
  b  = SHIFT(b,n/2)
  return BINARY_ADD(a,b,d)
```

Let's prove that `REC_MULTIPLY_2(x,y)` does indeed return the *product* of x and y.

Proof by complete induction.

1. **Define $P(n)$:**

   RTP: $P(n)$ is true for all even $n \in \mathbb{N}$ and $n = 1$.

2. **Base Case:**

3. **Inductive Hypothesis:**

4. **Inductive Step:**