Chapter 1

Recursion

1.1 Operations on Integers

Raising an integer to a power b^N, multiplying $x \times y$, and matrix multiplication each have surprising divide and conquer algorithms.

Example 1.1.1 b^N: Suppose that you are given two integers b and N and want to compute b^N.

The Iterative Algorithm: The obvious iterative algorithm simply multiplies b together N times. The obvious recursive algorithm recurses with $Power(b, N) = b \times Power(b, N-1)$. This requires the same N multiplications.

The Straightforward Divide and Conquer Algorithm: The obvious divide and conquer technique cuts the problem into two halves using the property that $b^{\lceil \frac{N}{2} \rceil} \times b^{\lfloor \frac{N}{2} \rfloor} = b^{\lceil \frac{N}{2} \rceil + \lfloor \frac{N}{2} \rfloor} = b^N$. This leads to the recursive algorithm $Power(b, N) = Power(b, \lceil \frac{N}{2} \rceil) \times Power(b, \lfloor \frac{N}{2} \rfloor)$. Its recurrence relation gives $T(N) = 2T(\frac{N}{2}) + 1$ multiplications. The technique in Section **??** notes that $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$ and $f(N) = \Theta(N^0)$ so $c = 0$. Because $\frac{\log a}{\log b} > c$, the technique concludes that time is dominated by the base cases and $T(N) = \Theta(N^{\frac{\log a}{\log b}}) = \Theta(N)$. This is no faster than the standard iterative algorithm.

Reducing the Number of Recursions: This algorithm can be improved by noting that the two recursive calls are almost the same and hence need only to be called once. The new recurrence relation gives $T(N) = 1T(\frac{N}{2}) + 1$ multiplications. Here $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$ and $f(N) = \Theta(N^0)$ so $c = 0$. Because $\frac{\log a}{\log b} = c$, we conclude that time is dominated by all levels and $T(N) = \Theta(f(N) \log N) = \Theta(\log N)$ multiplications.

Code:

```
algorithm $Power(b, N)$
⟨pre−cond⟩: $N \geq 0$ ($N$ and $b$ not both 0)
⟨post−cond⟩: Outputs $b^n$.

begin
    if( $N = 0$ ) then
        return(1)
    else
        $half = \lfloor \frac{N}{2} \rfloor$
        $p = Power(b, half)$
        if( $2 \cdot half = N$ ) then
            return( $p \cdot p$ )    % if $N$ is even, $b^N = b^{\lfloor N/2 \rfloor} \cdot b^{\lfloor N/2 \rfloor}$
        else
            return( $p \cdot p \cdot b$ )% if $N$ is odd, $b^N = b \cdot b^{\lfloor N/2 \rfloor} \cdot b^{\lfloor N/2 \rfloor}$
        end if
    end if
end algorithm
```

Tree of Stack Frames :

```
    <-| return value
      | 32 = 4x4x2
 -----
| b=2 |   <-| return value
|_N=5_|     | 4 = 2x2
       \-----
       | b=2 |  <-| return value
       |_N=2_|    | 2 = 1x1x2
             \-----
             | b=2 |   <-| return value
             |_N=1_|     | 1
                    \-----
                    | b=2 |
                    |_N=0_|
```

Running Time:

Input Size: One is tempted to say that the first two $\Theta(N)$ algorithms require a linear number of multiplications and that the last $\Theta(\log N)$ one requires a logarithmic number. However, in fact the first two require exponential $\Theta(2^n)$ number and the last a linear $\Theta(n)$ number in the "size" of the input, which is typically the number of bits $n = \log N$ to represent the number.

Operation: Is it fair to count the number of multiplications and not bit operations in this case? I say not. The output b^N contains $\Theta(N \log b) = 2^{\Theta(n)}$ bits and hence it will take this many bit operations to simply output the answer. Given this, it is not really fair to say that the time complexity is only of $\Theta(n)$.

Example 1.1.2 $x \times y$: The time complexity of Example 1.1.1 was measured in terms of the number of multiplications. This ignores the question of how quickly one can multiply.

The input for the next problem consists of two strings of n digits each. These are viewed as two integers x and y either in binary or in decimal notation. The problem is to multiply them.

The Iterative Algorithm: The standard elementary school algorithm considers each pair of digits, one from x and the other from y, and multiplies them together. These n^2 products are shifted appropriately and summed. The total time is $\Theta(n^2)$. It is hard to believe that one could do faster.

```
          8  2  7
          5  9  6
-----------------
             4  2
          1  2
       4  8
          6  3
       1  8
    7  2
       3  5
    1  0
 4  0
-----------------
 4  9  2  8  9  2
```

The Straightforward Divide and Conquer Algorithm: Let us see how well the divide and conquer technique can work. Split each sequence of digits in half and consider each half as an integer. This gives $x = x_1 \cdot 10^{\frac{n}{2}} + x_0$ and $y = y_1 \cdot 10^{\frac{n}{2}} + y_0$. Multiplying these symbolically gives

$$\begin{aligned} x \times y &= \left(x_1 \cdot 10^{\frac{n}{2}} + x_0\right) \times \left(y_1 \cdot 10^{\frac{n}{2}} + y_0\right) \\ &= (x_1 y_1) \cdot 10^n + (x_1 y_0 + x_0 y_1) \cdot 10^{\frac{n}{2}} + (x_0 y_0) \end{aligned}$$

The obvious divide and conquer algorithm would recursively compute the four subproblems x_1y_1, x_1y_0, x_0y_1, and x_0y_0, each of $\frac{n}{2}$ digits. This would take $4T(\frac{n}{2})$ time. Then these four products are shifted appropriately and summed. Note that additions can be done in $\Theta(n)$ time. See Section **??**. Hence, the total time is $T(n) = 4T(\frac{n}{2}) + \Theta(n)$. Here $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2$ and $f(n) = \Theta(n^1)$ so $c = 1$. Because $\frac{\log a}{\log b} > c$, the technique concludes that time is dominated by the base cases and $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$. This is no improvement in time.

Reducing the Number of Recursions: Suppose that we could find a trick so that we only need to recurse three times instead of four. One's intuition might be that this would only provide a linear time savings, but in fact the savings is much more. $T(n) = 3T(\frac{n}{2}) + \Theta(n)$. Now $\frac{\log a}{\log b} = \frac{\log 3}{\log 2} = 1.58..$, which is still bigger than $c =$. Hence, time is still dominated by the base cases, but now this is $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{1.58..})$. This is a significant improvement from $\Theta(n^2)$.

The Trick : The first step is to multiply x_1y_1 and x_0y_0 recursively as required. This leaves us only one more recursive multiplication.

If you review the symbolic expansion of $x \times y$, you will see that we do not actually need to know the value of x_1y_0 and x_0y_1. We only need to know their sum. Symbolically, we can observe the following.

$$\begin{aligned} & x_1y_0 + x_0y_1 \\ &\quad = [x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0] - x_1y_1 - x_0y_0 \\ &\quad = [(x_1 + x_0)(y_1 + y_0)] - x_1y_1 - x_0y_0 \end{aligned}$$

Hence, the sum $x_1y_0 + x_0y_1$ that we need can be computed by adding x_1 to x_0 and y_1 to y_0; multiplying these sums; and subtracting off the values x_1y_1 and x_0y_0 that we know from before. This requires only one additional recursive multiplication. Again we use the fact that additions are fast, requiring only $\Theta(n)$ time.

Code:

algorithm *Multiply*(x, y)

$\langle$***pre−cond***$\rangle$**:** x and y are two integers represented as an array of n digits

***$\langle$post−cond$\rangle$*:** The output consists of their product represented as an array of $n+1$ digits

```
begin
    if(n=1) then
        result( $x \times y$ ) % product of single digits
    else
        $\langle x_1, x_0 \rangle$ = high and low order $\frac{n}{2}$ digits of x
        $\langle y_1, y_0 \rangle$ = high and low order $\frac{n}{2}$ digits of y
        $A = Multiply(x_1, y_1)$
        $C = Multiply(x_0, y_0)$
        $B = Multiply(x_1 + x_0, y_1 + y_0) - A - C$
        result( $A \cdot 10^n + B \cdot 10^{\frac{n}{2}} + C$ )
    end if
end algorithm
```

It is surprising that this trick reduces the time from $\Theta(n^2)$ to $\Theta(n^{1.58})$.

Dividing into More Parts: The next question is whether the same trick can be extended to improve the time even further. Instead of splitting each of x and y into two pieces, let's split them each into d pieces. The straightforward method recursively multiplies each of the d^2 pairs of pieces together, one from x and one from y. The total time is $T(n) = d^2T(\frac{n}{d}) + \Theta(n)$. Here $a = d^2$, $b = d$, $c = 1$, and $\frac{\log d^2}{\log d} = 2 > c$. This gives $T(n) = \Theta(n^2)$. Again, we are back where we began.

Reducing the Number of Recursions: The trick now is to do the same with fewer recursive multiplications. It turns out it can be done with only $2d-1$ of them. This gives time of only $T(n) = (2d-1)T(\frac{n}{d}) + \Theta(n)$. Here $a = 2d-1$, $b = d$, $c = 1$, and $\frac{\log(2d-1)}{\log(d)} \approx \frac{\log(d)+1}{\log(d)} = 1 + \frac{1}{\log(d)} \approx c$. By increasing d, the time for the top stack frame and for the base cases becomes closer and closer to being equal. Recall that when this happens, we must add an extra $\Theta(\log n)$ factor to account for the $\Theta(\log n)$ levels of recursion. This gives $T(n) = \Theta(n \log n)$, which is a surprising running time for multiplication.

Fast Fourier Transformations: We will not describe the trick for reducing the number of recursive multiplications from d^2 to only $2d-1$. Let it suffice that it involves thinking of the problem as the evaluation and interpolation of polynomials. When d becomes large, other complications arise. These are solved by using the $2d$-roots of unity over a finite field. Performing operations over this finite field require $\Theta(\log\log n)$ time. This increases the total time from $\Theta(n \log n)$ to $\Theta(n \log n \log\log n)$. This algorithm is used often for multiplication and many other applications such as signal processing. It is referred to as *Fast Fourier Transformations.*

Example 1.1.3 Strassen's Matrix Multiplication: The next problem is to multiply two $n \times n$ matrices.

The Iterative Algorithm: The obvious iterative algorithm computes the $\langle i, j \rangle$ entry of the product matrix by multiplying the i^{th} row of the first matrix with the j^{th} column of the second. This requires $\Theta(n)$ scalar multiplications. Because there are n^2 such entries, the total time is $\Theta(n^3)$.

The Straightforward Divide and Conquer Algorithm: When designing a divide and conquer algorithm, the first step is to divide these two matrices into four submatrices each. Multiplying these symbolically gives the following.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae+bf & ag+bh \\ ce+df & cg+dh \end{pmatrix}$$

Computing the four $\frac{n}{2} \times \frac{n}{2}$ submatrices in this product in this way requires recursively multiplying eight pairs of $\frac{n}{2} \times \frac{n}{2}$ matrices. The total computation time is given by the recurrence relation $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^{\frac{\log 8}{\log 2}}) = \Theta(n^3)$. This is no faster than the standard iterative algorithm.

Reducing the Number of Recursions: Strassen found a way of computing the four $\frac{n}{2} \times \frac{n}{2}$ submatrices in this product using only seven such recursive calls. This gives $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\frac{\log 7}{\log 2}}) = \Theta(n^{2.8073})$. We will not include the details of the algorithm.

Exercise 1.1.1 *(See solution in Section* **??***) Recursive GCD*

1. *Write a recursive program to find the GCD of two numbers. The program should mirror the iterative algorithm found in Section* **??***.*

2. *Rewrite this recursive algorithm to solve the following more general problem. The input still consists of two integers a and b. The output consists of three integers g, u, and v, such that $u \cdot a + v \cdot b = g = GCD(a,b)$. For example, on $a = 25$ and $b = 15$ the algorithm outputs $\langle 5, 2, -3 \rangle$ because $2 \cdot 25 - 3 \cdot 15 = 50 - 45 = 5 = GCD(25, 15)$. Provide both a paragraph containing the friend's explanation of the algorithm and the recursive code.*

3. *Write an algorithm for the following problem. The input consist of three integers a, b and w. Assume that you live in a country that has two types of coins, one worth a dollars and the other b dollars. Both you and the store keeper have a pocket full of each. You must pay him w dollars. You can give him any number of coins and he may give you change with any number of coins. Your algorithm must determine whether or not this is possible and if so describe some way how (not necessarily the the optimal way). Hint: compute $GCD(a,b)$ and use the three values g, u, and v. Consider the two cases when g divides w and when it does not. (If you want to find the optimal number of coins. Basically, you change a solution by using the fact that $(\frac{b}{g}) \cdot a - (\frac{a}{g}) \cdot b = 0$.)*

4. *Designing an algorithm that when given a prime p and an integer $x \in [1, p-1]$, outputs an* inverse *y such that $x \cdot y \equiv_{mod\ p} 1$. Hint: First show that $GCD(p,x) = 1$. Then compute $GCD(p,x)$ and use the values u, and v. Proving that every x has such an inverse proves that the integers mod a prime forms a* field.

- Answer:

  a) Given the integers a and b, the iterative algorithm creates two numbers $x = b$ and $y = a \ mod\ b$. It notes that $GCD(a,b) = GCD(x,y)$, and hence it can return $GCD(x,y)$ instead of $GCD(a,b)$. This algorithm is even easier when you have a friend. We simply give the subinstance $\langle x, y \rangle$ to the friend and he computes $GCD(x,y)$ for us. For the iterative algorithm, we need to make sure we are "making progress" and for the recursive algorithm, we need to make sure that we give the friend a “smaller” instance. Either way, we make sure that in some way $\langle x, y\rangle$ is smaller than $\langle a, b\rangle$. For the iterative algorithm, we need an exit condition that we are sure to eventually meet and for the recursive algorithm, we need bases cases such that every possible instance is handled. Either way, we consider the case when y or b is zero. The resulting code is

algorithm $GCD(a, b)$
⟨*pre−cond*⟩: a and b are integers.
⟨*post−cond*⟩: Returns $GCD(a, b)$.

```
begin
    if($b = 0$) then
        return( $a$ )
    else
        return( $GCD(b, a\ mod\ b)$ )
    end if
end algorithm
```

b) We will need to understand this relationship $y = a\ mod\ b$ better. Here y is the remainder when you divide a by b. If we let $r = \lfloor \frac{a}{b} \rfloor$, then $a = r \cdot b + y$ or $y = a - r \cdot b$.

When we generalize problem, the friend in addition to g also gives us u_{sub} and v_{sub} such that $u_{sub} \cdot x + v_{sub} \cdot y = g = GCD(x, y) = GCD(a, b)$. Plugging in $x = b$ and $y = a - r \cdot b$ gives $u_{sub} \cdot b + v_{sub} \cdot (a - r \cdot b) = g$ or $v_{sub} \cdot a + (u_{sub} - v_{sub} \cdot r) \cdot b = g$. Hence, if we set $u = v_{sub}$ and $v = u_{sub} - v_{sub} \cdot r$, then we get $u \cdot a + v \cdot b = g = GCD(a, b)$ as required. We simply provide these answers. For the base case with $b = 0$, $g = GCD(a, b) = a$. Hence, $u = 1$ and $v = 0$, gives that $u \cdot a + v \cdot b = g = GCD(a, b)$. The resulting code is

algorithm $GCD(a, b)$
⟨*pre−cond*⟩: a and b are integers.
⟨*post−cond*⟩: Returns integers g, u, and v such that $u \cdot a + v \cdot b = g = GCD(a, b)$.

```
begin
    if($b = 0$) then
        return( $\langle a, 1, 0\rangle$ )
    else
        $x = b$
        $r = \lfloor \frac{a}{b} \rfloor$
        $y = a - r \cdot b$
        $\langle g, u_{sub}, v_{sub}\rangle = GCD(x, y)$
        $u = v_{sub}$
        $v = u_{sub} - v_{sub} \cdot r$
        return( $\langle g, u, v\rangle$ )
    end if
end algorithm
```

c) Our goal is to find two integers U and V such that $U \cdot a + V \cdot t = w$. Then you “give” the store keeper U of the a coins and V of the b coins for a total worth of w dollars. If U or V is negative, this amounts to the store keeper giving you coins as change.

To find U and V, lets start by calling the GCD algorithm on a and b. This returns integers g, u, and v such that $u \cdot a + v \cdot b = g = GCD(a, b)$.

If g divides evenly into w, then multiplying through by $(\frac{w}{g})$ gives $(\frac{uw}{g}) \cdot a + (\frac{vw}{g}) \cdot b = g(\frac{w}{g}) = w$ and we are done.

By the definition of $g = GCD(a, b)$, we know g divides into a and into b and hence, it divides evenly into $U \cdot a + V \cdot b$. It follows that if g does not divide evenly into w, then there is no integer solution to $U \cdot a + V \cdot b = w$.

d) A *Group* is a set of values closed under "plus" and "times". For example, the integers *mod* 6 forms a group. In this group, $4 \times 4 \equiv_{mod\ 6} 16 \equiv_{mod\ 6} 4$ and $4 \times 3 \equiv_{mod\ 6} 12 \equiv_{mod\ 6} 0$. Do not confuse this notation with the function $Mod(x, 6)$ which returns the remainder of $\frac{x}{6}$ which is between 0 and 5. Instead we mean the following. When we are working in the world of integers *mod* 6, we have that $\ldots -11 = -5 = 1 = 7 = 13 \ldots$ are different "names" for the same object. Similarly $-6 = 0 = 6 = 12$. Hence, our world only has 6 objects, those that are equal to 0 mod 6, those equal to 1, ..., and those equal to 5. Then the group defines how to add and multiply these objects together. For example, $-5 \times 8 \equiv_{mod\ 6} -40 \equiv_{mod\ 6} 2$ and $1 \times 2 \equiv_{mod\ 6} 2$ are two ways of saying the same thing.

A *Field* is the same except every value x other than zero has an *inverse* y such that $x \cdot y = 1$. We will see that the integers *mod* 6 is not a field, but the integers *mod* 7 is a field. For example, *mod* 7, the inverse of 2 is 4 because $2 \times 4 = 8 \equiv_{mod\ 7} 1$.

The integers *mod* p forms a field if p is prime. This exercise proves this, namely the exercise is to design an algorithm that when given a prime p and an integer $x \in [1, p-1]$, outputs an inverse y such that $x \cdot y \equiv_{mod\ p} 1$.

First note that because p is prime and p does not divide x, it follows that $GCD(p, x) = 1$. We start by calling the GCD algorithm on p and x. This returns integers $g = 1$, u, and v such that $u \cdot p + v \cdot x = 1$. This is another way of writing $x \cdot v \equiv_{mod\ p} 1$. Hence, v is the inverse of x.

In contrast to the above field, the group consisting of the integers *mod* n is a little odd when n is not prime. For example, suppose $n = a \cdot b$, with a and b neither zero nor one *mod* n. These integers are what we call *zero-divisors* because $a \cdot b \equiv_{mod\ n} 0$. We use this to prove that a does not have an inverse *mod* n. This will prove that the integers *mod* n is not a field if n is not prime.

Suppose by way of contradiction that a has an inverse c such that $c \cdot a \equiv_{mod\ n} 1$. Then if we multiply $a \cdot b \equiv_{mod\ n} 0$ through by c, we get that $c \cdot a \cdot b \equiv_{mod\ n} c \cdot 0$ or that $1 \cdot b = b \equiv_{mod\ n} 0$, which is a contradiction.

1.2 Ackermann's Function

If you are wondering just how slowly a program can run, consider the algorithm below. Assume the input parameters n and k are natural numbers.

Algorithm:

```
algorithm $A(k, n)$
    if( $k = 0$) then
        return( n+1+1 )
    else
        if( $n = 0$) then
            if( $k = 1$) then
```

```
                    return( 0 )
                else
                    return( 1 )
            else
                return( $A(k-1, A(k, n-1))$)
            end if
        end if
    end algorithm
```

Recurrence Relation: Let $T_k(n)$ denote the value returned by $A(k, n)$. This gives $T_0(n) = 2+n$, $T_1(0) = 0$, $T_k(0) = 1$ for $k \geq 2$, and $T_k(n) = T_{k-1}(T_k(n-1))$ for $k > 0$ and $n > 0$.

Solving:

$$T_0(n) = 2 + n$$

$$T_1(n) = T_0(T_1(n-1)) = 2 + T_1(n-1) = 4 + T_1(n-2) = 2i + T_1(n-i) = 2n + T_1(0) = 2n.$$

$$T_2(n) = T_1(T_2(n-1)) = 2 \cdot T_2(n-1) = 2^2 \cdot T_2(n-2) = 2^i \cdot T_2(n-i) = 2^n \cdot T_2(0) = 2^n$$

$$T_3(n) = T_2(T_3(n-1)) = 2^{T_3(n-1)} = 2^{2^{T_3(n-2)}} = \left[\underbrace{2^{2^{\cdots^{2}}}}_{i}\right]^{T_3(n-i)} = \left[\underbrace{2^{2^{\cdots^{2}}}}_{n}\right]^{T_3(0)} = \underbrace{2^{2^{\cdots^{2}}}}_{n}$$

$$T_4(0) = 1.\ T_4(1) = T_3(T_4(0)) = T_3(1) = \underbrace{2^{2^{\cdots^{2}}}}_{1} = 2.$$

$$T_4(2) = T_3(T_4(1)) = T_3(2) = \underbrace{2^{2^{\cdots^{2}}}}_{2} = 2^2 = 4.$$

$$T_4(3) = T_3(T_4(2)) = T_3(4) = \underbrace{2^{2^{\cdots^{2}}}}_{4} = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65,536.$$

Note $\underbrace{2^{2^{\cdots^{2}}}}_{5} = 2^{65,536} \approx 10^{21,706}$, while the number of atoms in the universe is less than 10^{100}.

$$T_4(4) = T_3(T_4(3)) = T_3(65,536) = \underbrace{2^{2^{\cdots^{2}}}}_{65,536}.$$

Ackermann's function is defined to be $A(n) = T_n(n)$. $A(4)$ is is bigger than any number in the natural world. $A(5)$ is unimaginable.

Running Time: The only way that the program builds up a big number is by continually incrementing it by one. Hence, the number of times one is added is at least as huge as the value $T_k(n)$ returned.

Crashing: Programs can stop at run-time because of: 1) over flow in an integer value; 2) running out of memory; 3) running out of time. Which is likely to happen first? If the machine's integers are 32 bits, then they hold a value that is about 10^{10}. Incrementing up to this value will take a long time. However, much worse than this, each two increments needs another recursive call creating a stack of about this many recursive stack frames. The machine is bound to run out of memory first.

Exercise 1.2.1 *Design the algorithm and compute the running time when $d = 3$.*

1.3 Exercises

Exercise 1.3.1 *Review the problem on Iterative Cake Cutting. You are now to write a recursive algorithm for the same problem. You will, of course, need to make the pre and post conditions more general so that when you recurse, your subinstances meet the preconditions. Similar to moving from insertion sort to merge sort, you need to make the algorithm faster by cutting the problem in half.*

1. *You will need to generalize the problem so that the subinstance you would like your friend to solve is a legal instance according to the preconditions and so that the post conditions states the task you would like him to solve. Make the new problem, however, natural. Do not, for example, pass the number of players n in the original problem or the level of recursion. The input should simply be a set of players and a sub-interval of cake. The post condition should state the requirements on how this subinterval is divided among these players. To make the problem easier, assume that the number of players is $n = 2^i$ for some integer i.*

2. *Give recursive pseudo code for this algorithm. As a big hint, towards designing a recursive algorithm, we will tell you the first things that the algorithm does. Each player specifies where he would cut if he were to cut the cake in half. Then one of these spots is chosen. You need to decide which one and how to create two subinstances from this.*

3. *Prove that if your instance meets the preconditions, then your two subinstances also meet the preconditions.*

4. *Prove that if your friend's solutions meet the postconditions, then your solution meets the postcondition.*

5. *Prove that your solution for the base case meets the postconditions.*

6. *Give and solve the recurrence relation for the Running Time of this algorithm.*

7. *Now suppose that n is not 2^i for some integer i. How would we change the algorithm so that it handles the case when n is odd? I have two solutions. One which modifies the recursive algorithm directly and one that combines the iterative algorithm and the recursive algorithm. You only need to do one of the two (as long as it works and does not increase the bigOh of the running time.)*