Class Notes, CS 3137

1 Measuring the Running Time of Programs

- We can define a function $T(N)$ to represent the number of units of time that an algorithm takes for an input of size N
- We can understand an algorithm's cost by finding its complexity class:
  - If $T(N) = k$, where k is some constant, then we can say $T(N)$ is a *constant* time algorithm. This is a $O(1)$ algorithm.
  - If $T(N) = kN$, where k is some constant, then we can say $T(N)$ is a *linear* time algorithm. This is a $O(N)$ algorithm.
  - If $T(N) = logN$, then we can say $T(N)$ is a *logarithmic* time algorithm. This is a $O(logN)$ algorithm.
  - If $T(N) = kN^2$, then we can say $T(N)$ is a *quadratic* time algorithm. This is a $O(N^2)$ algorithm.
  - If $T(N) = kN^3$, then we can say $T(N)$ is a *cubic* time algorithm. This is a $O(N^3)$ algorithm.
  - If $T(N) = kX^N$, where X is some constant base, then we can say $T(N)$ is an *exponential* time algorithm. This is a $O(X^N)$ algorithm.
- To be more precise, we say $T(N) = O(g(N))$ if there are positive constants k and n_0 such that $T(N) \leq kg(N),\ \forall N \geq n_0$
- Its important to remember that we are looking at *relative* rates when comparing our own algorithm to a known function. We need to see at what rate an algorithm's running time increases as the input size N increases
- When we say $T(N) = O(g(N))$, we are saying that $T(N)$ grows **no faster** that $g(N)$. We can say that $g(n)$ is an *upper bound* on $T(N)$.
- It is instructive to check out figure 2 below to see how the running time of a program changes with each complexity class.

2 Θ and Ω Notation

Most of the time we use Big O notation to understand the running time of programs, which gives us an *upper bound* on the running time. We can define a *lower bound* on the growth of a function using the notation $\Omega(g(n))$. This is the same idea as O notation, except we define a function $g(n)$ such that for $n > n_0$ and some constant $k > 0$, $T(N) \geq kg(N),\ \forall N \geq n_0$. In figure 1, we show this relationship.

Using the notation $\Theta(g(n))$ we can define a lower and upper bound on $f(n)$ by saying $f(n) = \Theta(g(n))$ if for all $n > n_0$, and constants c_1, c_2: $c_2g(n) \geq f(n) \geq c_1g(n)$.

3 Rules for Using Big O Analysis

- Constants don't matter. An algorithm that is $O(kN^2)$ is referred to as $O(N^2)$. It is clear that as N gets larger, the effect of the constant becomes less important.
- Lower order terms can be dismissed since they tend to be dominated by higher order times as N increases. An algorithm that is $O(N^2 + N)$ is referred to as $O(N^2)$. This is true for all polynomial functions: the order of the algorithm is $O(N^k)$ where k is the largest exponent.
- We try to pick a function with the tightest upper bound. So while an algorithm may be $T(N) = O(N)$, we can also say that $T(N) = O(N^2)$, $T(N) = O(N^3)$ and so on, since these are upper bounds. But we want to pick the function with the smallest upper bound possible. Once again, we ignore constants.

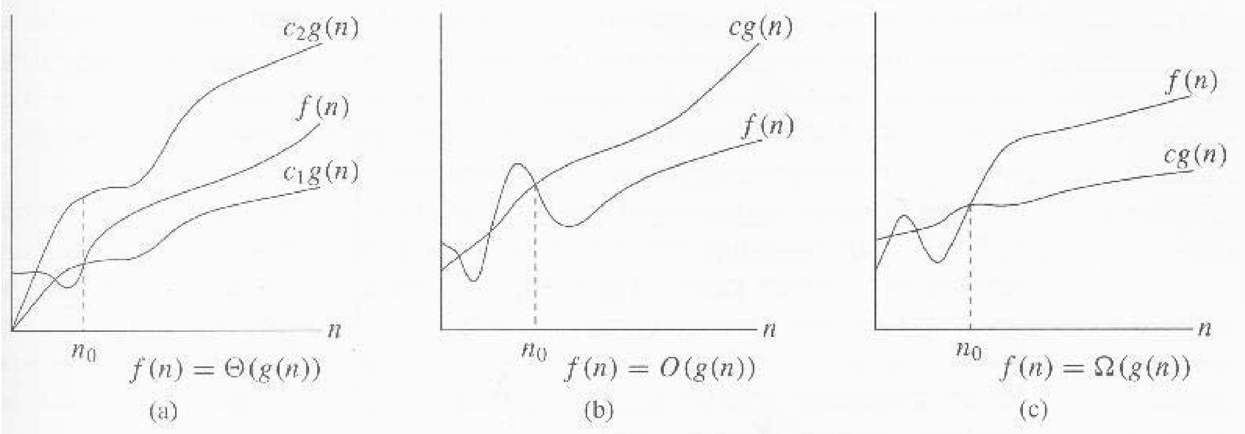


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

Figure 1: Example of Θ, O, and Ω notations (from Intro. to Algorithms, Corman, Leiserson,Rivest, Stein

- Even though we will use Base 2 logarithms in most of our analysis, the base doesn't really matter. We can always convert a logarithm in one base to any other base using a constant. For example to convert Base 2 logarithms to Base 10 logarithms we use:

$$log_{10} N \ = \ \frac{log_2 N}{log_2 10} \tag{1}$$

  Proof:

$$Z \ = \ log_{10} N \ , \ X \ = \ log_2 N \ , \ Y \ = \ log_2 10 \tag{2}$$

  So we have to prove that $Z \ = \ \frac{X}{Y}$ or $X \ = \ Y Z$.

  We know that:

$$10^Z \ = \ N \ , \ 2^X \ = \ N \ , \ , 2^Y \ = \ 10 \tag{3}$$

  Rewriting:

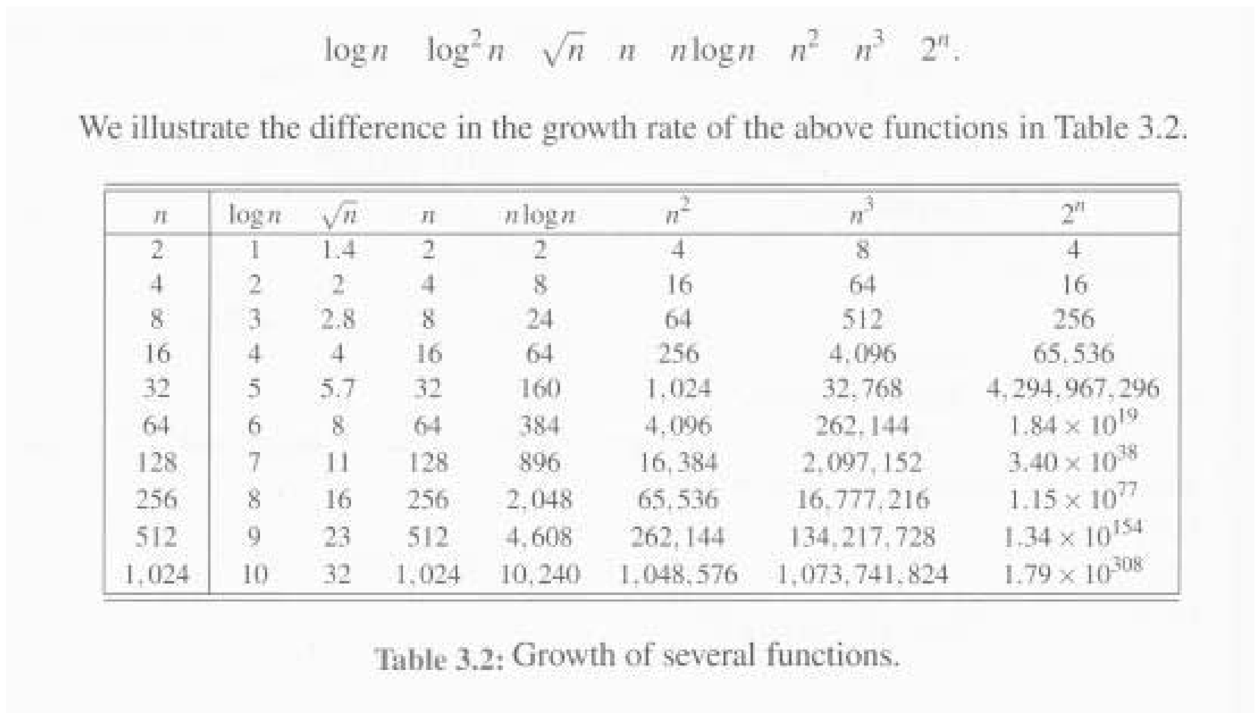
$$2^X \ = \ N \ = \ 10^Z \ = \ (2^Y)^Z \ , \ 2^X \ = \ 2^{YZ} \ , \ X \ = \ Y Z \tag{4}$$

- If $T_1(N) \ = \ O(f(N))$ and $T_2(N) \ = \ O(g(n))$ then:

  a) $T_1(N) + T_2(N) \ = \ max(O(f(N)) \, , O(g(n))$

  b) $T_1(N) * T_2(N) \ = \ (O(f(N)) * O(g(n))$

- $log^k N \ = \ O(N)$. This implies that logarithms grow slowly. Powers of $logN$ still are $O(N)$.



$$\log n \quad \log^2 n \quad \sqrt{n} \quad n \quad n\log n \quad n^2 \quad n^3 \quad 2^n.$$

We illustrate the difference in the growth rate of the above functions in Table 3.2.

n	$\log n$	$\sqrt{n}$	n	$n\log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4,096	65,536
32	5	5.7	32	160	1,024	32,768	4,294,967,296
64	6	8	64	384	4,096	262,144	1.84×10^{19}
128	7	11	128	896	16,384	2,097,152	3.40×10^{38}
256	8	16	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	23	512	4,608	262,144	134,217,728	1.34×10^{154}
1,024	10	32	1,024	10,240	1,048,576	1,073,741,824	1.79×10^{308}

Table 3.2: Growth of several functions.

Figure 2: Growth of functions of N

4 A Proof Using Big O Notation

Suppose we want to show that $5n^3 + 3n^2 + 4n + 8$ is $O(n^3)$. From the definition of the Big O-notation, we need to find K and n_0 such that: $5n^3 + 3n^2 + 4n + 8 \leq Kn^3$,$\forall n \geq n_0$. Letting $n_0 = 1$, $n \geq n_0$ is the same as $n \geq 1$, which is the same as $1 \leq n$. Now, multiplying each side by n successively:

$$1 \leq n$$

$$n \leq n^2$$

$$n^2 \leq n^3$$

using transitivity, we can also say:

$$a : n^2 \leq n^3$$

$$b : n \leq n^3$$

$$c : 1 \leq n^3$$

$$d : n^3 \leq n^3$$

Multiplying inequality (a) by 3 gives $3n^2 \leq 3n^3$
Multiplying inequality (b) by 4 gives $4n \leq 4n^3$
Multiplying inequality (c) by 8 gives $8 \leq 8n^3$
multiplying inequality (d) by 5 gives $5n^3 \leq 5n^3$
Adding up these inequalities gives:

$$5n^3 + 3n^2 + 4n + 8 \leq 20n^3$$

Hence, if we choose K = 20, then for all $n \geq 1$ we have proven that:

$$5n^3 + 3n^2 + 4n + 8 \leq Kn^3$$

which formally establishes that it is $O(n^3)$.

5 Creating a Model and Rules for Analyzing Computer Programs

- Each typical computer statement is assumed to take the same unspecified amount of time. This makes it easier to compare programs in a machine independent manner, since some machines execute instuctions faster or slower. Exceptions are *For, While, If* statments.
- *For and While* loop statements are evaluated by multiplying the number of statements inside the loop by the number of iterations of the loop. If the statements are nested, we just keep multiplying from the innermost loop to the outermost loop.
- *If* statements are evaluated by taking the maximum time of all the possible branches.
- I/O statments (*System.out.println, Reader, GUI commands*) usually take much longer than typical assignment statements etc. However, we still consider them a primitive, 1 time step operation
- If there are consecutive blocks of statements, we simply take the maximum time of the blocks, since this will dominate the other blocks.
- We have to be careful about whether the time we are speaking about is the average case time or what is known as a *worst case* time. This will become clear when we look at certain algorithms that may finish very quickly on one input set and work much longer on a different set. Sorting is an example, as the items to sort may be in order already, and the sort will do little work, or it may do a lot of work. Average cases are sometimes hard to analyze, so we often use worst case analysis.

6 Running Time Analysis of Selection Sort

Selection sort works by finding the smallest (or largest) element in an array, and placing that element in the first (or last) position of the array. It then repeats this procedure on the remaining elements - you can think of it doing the same thing on an array that is 1 element smaller.

```
//This program  generates a set of random integers and inserts
//them into an array and then sorts them using selection sort
import java.util.Random;
public class SelectionSort
{
public static void OrderArray(int[] item, int start,int end)
{
   int i,maxpos;
   maxpos=end;
   for(i=start;i<end;i++){
     if(item[i] > item[maxpos])
         maxpos=i;
   }
   int tmp=item[end];
   item[end]=item[maxpos];
   item[maxpos]=tmp;
}
public static void main( String [ ] args )
{
   int ARRAY_SIZE = 6;
   int MAXINTEGER = 100;
   int i,minpos;
         System.out.println("Random ");
   int[] item= new int[ARRAY_SIZE];
   Random generator= new Random();
   for( i = 0 ; i < ARRAY_SIZE; i ++) {
      int x= generator.nextInt(MAXINTEGER);
      System.out.println("Random number inserted in array is " + x);
      item[i]=x;
   }
   for( i = ARRAY_SIZE-1; i >0; i-- ) {
      OrderArray(item,0,i);
   }
   System.out.println("Sorted Array is:");
   for( i = 0 ; i < ARRAY_SIZE; i ++) {
      System.out.println(item[i]);
   }
}
}
```

6.1 Finding the Running Time of Selection Sort by Induction

We can see that the selection sort for an array of size N takes $N + (N-1) + (N-2) + ... + 1$ iterations. Can we sum these up? This is just the sum of the first N positive integers starting at 1, and the formula for this is:

$$\sum_{i=1}^{N} i = N \cdot (N+1)/2 \tag{5}$$

We can prove this is the formula by the method of *induction*. First, we need to establish a base case for $i = 1$:

Base Case: If $i = 1$, then

$$\sum_{i=1}^{1} i = 1 \cdot (1+1)/2 = 1 \tag{6}$$

So, the formula works for the case of $i = 1$ since 1 is the sum of the first integer in our sequence, 1. To show it works for all cases, we *assume* that the formula is true for $i = N$, and prove it must then also be true for $i = N + 1$. This is called the *induction step*. Once we do this, we can be sure it will always be true by just extending the base case one or more times using the induction step.

Induction Step: Assume the formula is true for $i \; = \; 1 \, to \, N$:

$$\sum_{i=1}^{N} i = N \cdot (N+1)/2 \tag{7}$$

Now, show it also must be true for the case of $i = 1 \, to \, N + 1$:

$$\sum_{i=1}^{N+1} i = (N+1) \cdot (N+2)/2 \;\; = \sum_{i=1}^{N} i + \; (N+1) \tag{8}$$

By the induction step assumption, the term above is simply the sum of the first N integers which we know to be equal to $N \cdot (N+1)/2$, and

$$\sum_{i=1}^{N+1} i = (N \cdot (N+1))/2 + N + 1 = (N^2 + N + 2N + 2)/2 = ((N+1)(N+2))/2 \tag{9}$$

which shows that the formula holds for input of size $N + 1$

What this tells us is that selection sort is $O(N^2)$, since it takes about $N \cdot (N+1)/2$ iterations. Once again, we drop lower order terms and constants to get this result.

Formally we can write this as a *recurrence relation*: $T(N) \; = \; T(N-1) \, + \, N$. A *recurrence relation*is often used to define a computation in terms of itself.

A common way to solve a recurrence relaton is by simply "telescoping" the recurrence, remembering that $T(1) = 1$:

$$T(N) \; = \; T(N-1) \, + \, N \tag{10}$$

$$T(N) \; = \; T(N-2) \, + \, (N-1) \, + \, N \tag{11}$$

$$T(N) \; = \; T(N-3) \, + \, (N-2) \, + \, (N-1) \, + \, N \tag{12}$$

$$T(N) \; = \; T(1) \, + \, 2 \, + \, 3 \, + \, \ldots \, + \, (N-1) \, + \, N \tag{13}$$

$$T(N) \; = \; \sum_{i=1}^{N} i \; = \; \frac{N \, \cdot \, (N+1)}{2} \tag{14}$$

$$T(N) \; \approx \; O(N^2) \tag{15}$$

7 Running Time Analysis of Binary Search

A common type of algorithm is known as *divide-and conquer*. By splitting up a large program into 2 smaller ones, we may make progress towards the solution. Binary Search is an example of this. If we are trying to find a certain element in a sorted array, we can test the middle element, and depending on whether the element we are seeking is in the first half of the array or the second, we can just search a continually decreasing array (by a factor of 2) until we are left with an array of size 1 to search. We can show that any algorithm that can cut its input in half (or by any other fraction) is $O(logN)$.

```
public class BinarySearch {

public static int BinSearch( int A[ ], int X, int N )
{
int Low, Mid, High;
     Low = 0; High = N - 1;
     while( Low <= High )
         {
           Mid = ( Low + High ) / 2;
           if( A[ Mid ] < X )
              Low = Mid + 1;
           else
           if( A[ Mid ] > X )
              High = Mid - 1;
           else
              return Mid;  /* Found */
     }
     return -1;     /* NotFound is defined as -1 */
}

public static void main( String[] args)  {
    int[] A = { 1, 3, 5, 7, 9, 13, 15 };
    int count=A.length;
    int i;
    for(i=0; i<count;i++)
       System.out.println("Array[" + i +"]" + " = " + A[i]);

    for( i = 0; i <=15; i++ ) {
        int found=BinSearch(A,i,count);
        if(found>=0)
          System.out.println("BinarySearch for " + i +
                        " found in Array[" + found+"]");
          else
             System.out.println("BinarySearch for " + i + " not found!");
   }
 }
}
```

We can analyze this program as follows: The running time of the program is

$$T(N) = T(\frac{N}{2}) + 1 \tag{16}$$

At each step we do some constant amount of work (signified by the 1 term) and then work on a problem of half size. Eventually, the problem size will be small enough where we can analyze it. This usually occurs when $T(N) = T(1) = O(1)$ - an input size of 1 is solved in a constant time. In binary search, that just means we can find the element in an array of size 1 in constant time.

To compute the running time for an input of size $N = 2^k$ (N is a power of 2, which simplifies the analysis), we can do the following:

$$\begin{aligned} T(4) &= T(2) + 1 && (17)\\ &= T(1) + 1 + 1 && (18)\\ &= 1 + 1 + 1 = 3 && (19) \end{aligned}$$

$$\begin{aligned} T(8) &= T(4) + 1 && (20)\\ &= T(2) + 1 + 1 && (21)\\ &= T(1) + 1 + 1 + 1 && (22)\\ &= 1 + 1 + 1 + 1 = 4 && (23) \end{aligned}$$

$$\begin{aligned} T(16) &= T(8) + 1 && (24)\\ &= T(4) + 1 + 1 && (25)\\ &= T(2) + 1 + 1 + 1 && (26)\\ &= T(1) + 1 + 1 + 1 + 1 && (27)\\ &= 1 + 1 + 1 + 1 + 1 = 5 && (28) \end{aligned}$$

Its clear that for an input of $N = 2^k$, the running time is a function of $k : T(N) = k + 1$. And since $k = logN$, we can see that

$$T(N) = k + 1 = logN + 1 = O(logN) \qquad (29)$$

8 Mergesort

Mergesort is an interesting algorithm that reflects a typical Computer Science strategy: *divide and conquer*. The idea is very simple:

To sort N items do the following: Split the array in half, forming two arrays of $\frac{N}{2}$ items each. Sort these arrays separately, and then merge the two arrays back together again.

The algorithm is naturally recursive, as we can keep splitting and merging the arrays of size N, $\frac{N}{2}$, $\frac{N}{4}$, , ...

```
public static void mergeSort( Comparable [ ] a )
{
   Comparable [ ] tmpArray = new Comparable[ a.length ];
   mergesort( a, tmpArray, 0, a.length - 1 );
}


public static void mergesort( Comparable [ ] a, Comparable [ ] tmpArray,
                   int left, int right )
{
    if( left < right ) {
       int center = ( left + right ) / 2;
       mergesort( a, tmpArray, left, center );
       mergesort( a, tmpArray, center + 1, right );
       merge( a, tmpArray, left, center + 1, right );
    }
}

public static void merge( Comparable [ ] a, Comparable [ ] tmpArray,
               int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while( leftPos <= leftEnd && rightPos <= rightEnd )
       if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
          tmpArray[ tmpPos++ ] = a[ leftPos++ ];
       else
          tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    while( leftPos <= leftEnd )    // Copy rest of first half
       tmpArray[ tmpPos++ ] = a[ leftPos++ ];

    while( rightPos <= rightEnd )  // Copy rest of right half
       tmpArray[ tmpPos++ ] = a[ rightPos++ ];
    for( int i = 0; i < numElements; i++, rightEnd-- )
       a[ rightEnd ] = tmpArray[ rightEnd ]; // Copy tmpArray back
}
```

To sort N items, we need an additional array to store the merged data, so there is an additional memory cost for mergersort.

To analyze the cost of mergesort, we note that the merge step is a linear operation ($O(N)$) in the number of elements to sort. For $N = 1$ the time for mergesort is constant, and we can represent this as $T(1)$. Otherwise, the time to mergesort N items is the time to do 2 recursive mergesorts of size $\frac{N}{2}$, plus the merge step (which is linear). We can express these facts as:

$$T(1) = 1 \tag{30}$$

$$T(N) = 2T(\frac{N}{2}) + N \tag{31}$$

The above is a recurrence relation, expressing the recursive timing nature of this problem. To solve such a recurrence relation we do the following (we will assume N is a power of 2 so that our splits of the array are always into evenly balanced halves):

Divide both sides by N:

$$\frac{T(N)}{N} = \frac{T(\frac{N}{2})}{\frac{N}{2}} + 1 \tag{32}$$

Since the expression above is valid for any power of 2, we can express the time to sort an array of $\frac{N}{2}$ elements as:

$$\frac{T(\frac{N}{2})}{\frac{N}{2}} = \frac{T(\frac{N}{4})}{\frac{N}{4}} + 1 \tag{33}$$

and continuing with successive powers of 2...

$$\frac{T(\frac{N}{4})}{\frac{N}{4}} = \frac{T(\frac{N}{8})}{\frac{N}{8}} + 1 \tag{34}$$

$$... = ... \tag{35}$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1 \tag{36}$$

Now, add up all these equations. Notice that almost all the terms on the left hand sides of the equations also appear on the right hand sides of the equations. Cancelling these terms, we are left with the following:

$$\frac{T(N)}{N} = \frac{T(1)}{1} + log\,N \tag{37}$$

The $log\,N$ term is simply the sum of all the $1's$ on the right hand sides of the equations. Because we are splitting by powers of 2, there are $log\,N$ equations total. Hence the sum of the $1's$ is $logN$.

Now, multiply through by N:

$$T(N) = N\,T(1) + N\,log\,N = N + N\,log\,N = O(Nlog\,N) \tag{38}$$

9 Recursive Power Function

To compute the cost of the recursive power function, remember it is another divide-and-conquer method. We can analyze the problem as:

$$5^{16} = (5^2)^8 = (5^4)^4 = (5^8)^2 = (5^{16})^1 \tag{39}$$

So by continuing to half the exponent and raise the squared base, we eventually can get to a case where the exponent is 1. The number of steps required for this is equal to the number of times it takes to divide the exponent in half until its 1.

In Java, we can write this as follows:

```
public static long pow( long x, int n )  {
    if( n == 0 )
       return 1;
    if( n == 1 )
       return x;
    if( isEven( n ) )
        return pow( x * x, n / 2 );
    else
        return pow( x * x, n / 2 ) * x;
}
```

This is again an $O(logN)$ algorithm of the type:

$$T(N) \;=\; T(\frac{N}{2}) + 1 \tag{40}$$

which is the same cost as binary search.
Example:
if we call Pow(2,4), this gets broken down in to the recursive calls:

```
Pow(2,4)
   Pow(2*2,2)
      Pow(4*4,1) ---> returns 16 when exponent reaches 1
```

Now, if we call Pow(2,8) (double the exponent) this gets broken down into:

```
Pow(2,8)
   Pow(2*2,4)
      Pow(4*4,2)
         Pow(16*16,1) ---> returns 256 when exponent reaches 1
```

So doubling the exponent resulted in only 1 extra recrusive call - this is exactly what it means for an algortihm to grow at a rate of $O(logN)$

9.1 A Method with no speed up!

Note that if we think we can speed up the operation by doing something like this:

$$Pow(X,N) \;=\; Pow(X,N/2) * \; Pow(X,N/2) \tag{41}$$

If we do the recursive calls here, we can see:

```
                  Pow(2,4)

         Pow(2,2)    *         Pow (2,2)

 Pow(2,1) * Pow(2,1)   Pow(2,1) * Pow(2,1)
```

We see that for N=4, there are 6 recursive calls. If we double N to 8, then we can see that there are exactly twice as many recursive calls, since we are effectively calling Pow(2,4) twice.

```
                  Pow(2,8)

         Pow(2,4)    *         Pow (2,4)
```

we no longer have an $O(logN)$ algorithm; the cost is now $O(N)$. To prove this, assume that N is a power of 2 (i.e. $N = 2^k$):

$$
\begin{aligned}
T(N) &= T(\frac{N}{2}) + T(\frac{N}{2}) + 1 && (42)\\
&= 2 \cdot T(\frac{N}{2}) + 1 && (43)\\
& && (44)
\end{aligned}
$$

now divide by N:

$$
\begin{aligned}
\frac{T(N)}{N} &= \frac{T(\frac{N}{2})}{\frac{N}{2}} + \frac{1}{N} && (45)\\
\frac{T(\frac{N}{2})}{\frac{N}{2}} &= \frac{T(\frac{N}{4})}{\frac{N}{4}} + \frac{1}{\frac{N}{2}} && (46)\\
\frac{T(\frac{N}{4})}{\frac{N}{4}} &= \frac{T(\frac{N}{8})}{\frac{N}{8}} + \frac{1}{\frac{N}{4}} && (47)\\
&\cdots && (48)\\
\frac{T(2)}{2} &= \frac{T(1)}{1} + \frac{1}{2} && (49)\\
& && (50)
\end{aligned}
$$

Now, add up all these equations. Notice that almost all the terms on the left hand sides of the equations also appear on the right hand sides of the equations. Cancelling these terms, we are left with the following:

$$
\begin{aligned}
\frac{T(N)}{N} &= \frac{T(1)}{1} + \frac{1}{N}\sum_{i=0}^{k-1} 2^i && (51)\\
T(N) &= N\ T(1) + \sum_{i=0}^{k-1} 2^i && (52)\\
T(N) &= N + 2^k - 1 && (53)\\
T(N) &= N + N - 1 && (54)\\
T(N) &= 2N - 1\ \approx O(N) && (55)
\end{aligned}
$$

10 Fibonacci Number Computation

A common recurrence equation we run into is generating the Fibonacci Number sequence. The Fibonacci sequence is defined as:

Fib(0)=1; Fib(1)=1; Fib(n) = Fib(n-1) + Fib(n-2)

We can write a very simple recursive program to compute the Fibonaci sequence:

```
// Computes Fibonacci number n, where n is command line argument

public class fibonacci{

public static double fib(int n)
{
  if ((n==0) || (n==1)) return 1;
      else return fib(n-1) + fib(n-2);
}

public static void main(String[] args)
{
  int num;
  num= Integer.parseInt(args[0]);
  System.out.println("Fib(" +  num  +")=" + fib(num));
}
}
```

While this is a simple recursive program that mimics the recurrence equation, it is quite inefficient. The problem is that at each step of the recursion, we recompute a Fibonacci value that has already been computed. Below we show the recursive calls for fib(5), which calls fib(4) and fib(3) and so on:

```
                           fib(5)
                     /               \
             fib(4)                       fib(3)
            /       \                    /       \
     fib(3)         fib(2)           fib(2)     fib(1)
     /    \         /    \         /     \
  fib(2)  fib(1)  fib(1) fib(0)  fib(1)  fib(0)
   /   \
fib(1)  fib(0)
```

We can see how we are recomputing fib() values many times, greatly increasing the running time of the program. fib(3) is computed 2 times, fib(2) 3 times, fib(1) 5 times and fib(0) 3 times.

Section 1.25 of Weiss gives a proof that shows that Fibonacci computation is bounded by: $fib(k) \leq (\frac{5}{3})^k$. Thus, the Fibonacci sequence is an *exponential* time algorithm.

However, we can use a simple technque of storing the Fibonacci sequence in a table, computing only those members of the sequence we need to each time. Once we compute a value of Fib(N), we remember it and reuse it as needed. This technique is sometimes called *Dynamic Programming* as we create the values we need on the fly and memorize them in a table for reuse.

```
import java.io.*;
public class Fibdynamic
{
static final int maxN = 47;  /* Fib(47) is maximum 32 bit integer */
static int knownF[] = new int [maxN];

static int F(int i)
{
  if (i < 0) return 0;
  if (i == 1) return 1;
  if (knownF[i] != 0) return knownF[i];
  int t = i;
  if(i>1) t = F(i-1) + F(i-2);
  return knownF[i] = t;
}
  public static void  main(String[] args) {
  int n=Integer.parseInt(args[0]);
  System.out.println("Fib(" +n + ")" + "= "  +F(n));
  }
}
```

11 Towers of Hanoi Problem

Problem: move N numbered disks from one peg (peg A) to another peg (peg B) using a temporary storge peg (peg C) without ever having a larger disk sit on top of a smaller disk (disk 1 is the smallest disk, disk N is the largest disk). At the start of the game, all disks are on peg A, to be moved to peg B, using peg C ast temporary store, with largest disk on bottom and smallest on top.

```
public class tower
{
  public static void tower(char from, char to, char spare,int disk)
  {
   if (disk>0){
     tower(from, spare, to,disk-1);
     System.out.println("move disk " + disk + " from " +  from + " to " + to);
     tower(spare, to , from, disk-1);
   }
  }
public static void main(String[] args)
{
   int disk;
   disk= Integer.parseInt(args[0]);
   tower('A','B','C',disk);
}
>tower 2
move disk 1 from A to C
move disk 2 from A to B
move disk 1 from C to B
>tower 3
move disk 1 from A to B
move disk 2 from A to C
move disk 1 from B to C
move disk 3 from A to B
move disk 1 from C to A
move disk 2 from C to B
move disk 1 from A to B
>tower 4
move disk 1 from A to C
move disk 2 from A to B
move disk 1 from C to B
move disk 3 from A to C
move disk 1 from B to A
move disk 2 from B to C
move disk 1 from A to C
move disk 4 from A to B
move disk 1 from C to B
move disk 2 from C to A
move disk 1 from B to A
move disk 3 from C to B
move disk 1 from A to C
move disk 2 from A to B
move disk 1 from C to B
```

12 Complexity of Towers of Hanoi

We can analyze the towers of hanoi problem by the following recurrence relation:

$$T(N) \;=\; 1 \,+\, 2\,T(N-1) \tag{56}$$

$$T(N) \;=\; 1 \,+\, 2(1+2\,(T(N-2)) \tag{57}$$

$$T(N) \;=\; 1 \,+\, 2+2^2\,T(N-2) \tag{58}$$

$$T(N) \;=\; 1 \,+\, 2+2^2(1+2\,T(N-3)) \tag{59}$$

$$T(N) \;=\; 1 \,+\, 2+2^2+2^3\,T(N-3)) \tag{60}$$

$$T(N) \;=\; 1 \,+\, 2+2^2+2^{i-1}+2^i\,T(N-i)) \tag{61}$$

$$T(N) \;=\sum_{i=0}^{N-1}\;2^i \tag{62}$$

This is just a standard geometric progression that adds up the powers of 2. The formula for this is:

$$T(N) \;=\sum_{i=0}^{N-1}\;2^i \;=\; 2^N-1 \tag{63}$$

Remember that if the first term of a geometric progression is a, the common ratio is r and if there are N terms, the sum S is:

$$S \;=\; a\left(\frac{r^N-1}{r-1}\right) \tag{64}$$

What we see here is that the complexity is exponential. It grows very fast, as a power of 2. THe complexity is then:

$$T(N) \,\approx\, O(2^N) \tag{65}$$

13 Recap: Common Recurrence Relations

13.1 $T(N) \;=\; T(N-1) \,+\, N$

This recurrence is for an algorithm that loops through the input, eliminating 1 element at at time. An example is selection sort, which finds the smallest item in an array, and then repeats this for an array of size one less. Its time complexity is:

$$T(N) \;\approx\; O(N^2) \tag{66}$$

13.2 $T(N) \;=\; T(\frac{N}{2}) \,+\, 1$

This recurrence is for an algorithm that recrusively breaks a problem down into a problem half the size in one step. An example is binary search, which at each step does a simple constant time test to look at only half of the remaining input, until there is only 1 item left.

$$T(N) \;\approx\; O(\,log(N)) \tag{67}$$

13.3 $T(N) = T(N-1) + 1$

This is just a linear find operation: for N items, look at 1 item and then look at the rest of the input ($(N-1)$ items).

$$T(N) = T(N-1) + 1 \tag{68}$$

$$T(N) = T(N-2) + 1 + 1 \tag{69}$$

$$T(N) = T(N-3) + 1 + 1 + 1 \tag{70}$$

$$T(N) = 1 + 1 + \ldots + 1 \tag{71}$$

$$T(N) = N \approx O(N) \tag{72}$$

13.4 $T(N) = 2T(\frac{N}{2}) + N$

This is a recurrence relation for an algorithm that breaks the problem down into 2 smaller problems of half the size (divide-and-conquer), and it also requires examining every member of the input. An example is Mergesort of N items: we sort each half of the array independently, and then do a linear merge of all N items.

The result is $T(N) \approx O(N\, log(N))$ (see the proof in section 8 of these notes).

13.5 $T(N) = 2T(\frac{N}{2}) + 1$

This is the recurrence for splitting a problem in half, and doing a constant amount of other work each time.

$$T(N) \approx O(N) \tag{73}$$

13.6 $T(N) = 2T(N-1) + 1$

This is a recurrence that shows exponential growth (see proof in section for Towers of Hanoi problem) and its time complexity is:

$$T(N) \approx O(2^N) \tag{74}$$