CS 3410 – Ch 5 (DS*), 23 (IJP^)

*CS 1301 & 1302 text, Introduction to Java Programming, Liang, 7th ed.
^CS 3410 text, Data Structures and Problem Solving Using Java, Weiss, 4th edition

Sections	Pages	Review Questions
23.1-23.8	742-747	1-6

Sections	Pages
5.1-5.2,5.4-5.8,	187-193, 201-210, 212-214

Introduction

1. An algorithm is

   - A set of steps, that when followed, solve a problem.
   - "...a clearly specified set of instructions the computer will follow to solve a problem." [DS Text]
   - "An algorithm is a set of unambiguous instructions describing how to transform some input into some output." [http://www.cs.uiuc.edu/class/fa07/cs173/notes/algorithms.pdf]

2. We can think of a software system as an algorithm itself, which contains a collection of (sub)algorithms, *etc.* As programmers and software engineers, we *design* algorithms. A completed algorithm should undergo *validation* and *verification*. A *valid* algorithm solves the correct problem (e.g. it is not based on a misunderstanding with the customer). A *verified* (*correct*) algorithm solves the problem correctly (e.g. It doesn't mess up).

3. It is important to understand how efficient a software system is relative to time and space. In other words, *how much time will it take the system to run?* and *how much disk space/RAM will be required*? To understand this, we must understand how efficient our algorithms are. This is called *algorithm analysis*, where we, "...determine the amount of resources, such as time and space, that the algorithm will require." [DS]. Throughout this course we will study the efficiency of standard algorithms and custom algorithms. You will be responsible for learning the algorithms and knowing why they perform the way they do in terms of efficiency of time and sometimes space.

Sections 23.1, 23.2 (IJP) 5.1 (DS) – What is algorithm analysis?

1. The amount of time that it takes to run an algorithm almost always depends on the amount of input. For instance, the time to sort 1000 items takes longer than the time to sort 10 items.

2. The exact run-time depends on many factors such as the speed of the processor, the network bandwidth, the speed of RAM, the quality of the compiler, and the quality of the program, the amount of data. Thus, we can use execution time (clock time) to determine how "fast" an algorithm is on a particular piece of hardware, but we can't use it to compare across platforms.

3. To overcome this shortcoming, we measure time in a relative sense in a way so that it is independent of hardware. We do this by counting the number of times a *dominate operation* is executed in an algorithm. Consider this piece of code:

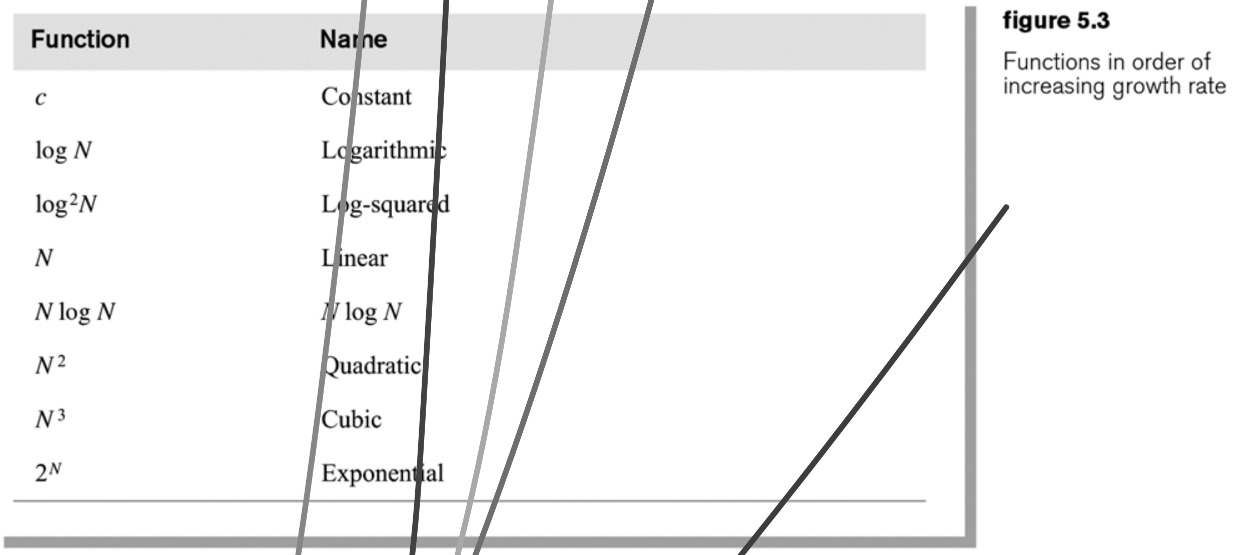
```
1.      int sum = 0;

2.      Scanner input = new Scanner( System.in );
3.      System.out.print( "How big is n? " );
4.      int n = input.nextInt();

5.      for( int i=1; i<=n; i++ )
        {
6.           sum += i;
        }

7.      System.out.println( sum );
```

In line 1 we have an initialization, lines 2-4 we read the value of *n.* The first time line 5 is executed, we have an initialization (i=1) and a comparison (i<n). Each subsequent time, we have an increment (i++) and a comparison. Line 6 is an addition and we can see that it is executed *n* times. Finally, line 7 is a print. Thus, we see that lines 1-4 and 7 each execute once and in the loop, we see that we have 1 initialization, n comparisons, n increments, and n additions. Usually, in such a situation, we will assume that the *dominant operation* is the addition in line 6 and to simplify matters more, we will effectively ignore the statements that are executed as part of the loop itself (initialization, comparison, increment). Now, suppose for a moment that all lines of code execute in exactly the same amount of time (on a given machine), say *x* seconds. Thus, how many statements are executed? Answer: 5 lines are executed once (lines 1-4 and 7) and line 6 is executed *n* times. So: 5+*n* so that the run-time is (5+*n*)*x*. Since *x* depends on the machine, we can ignore it to obtain a measure of the relative time of the algorithm. Thus, we will say that the time of the algorithm is given by $T(n) = n + 5$.

In *algorithm analysis* we are only concerned with the highest order term. In this case, it is *n*, so we ignore the 5 and say that this is a *linear time* algorithm. In other words, we say that the runtime is proportional to *n.* We frequently refer to this as the *complexity* of an algorithm (in this case, linear complexity). This analysis makes sense when we thing of *n* as a "large" value because as *n* becomes large, the 5 makes a negligible contribution to the total. For instance, if *n is 1,000,000,* then there is very little difference between 1,000,000 and 1,000,005.

4. Suppose that the time to download a file is: $T(N) = \frac{N}{1.6} + 2$, where *N* is the size of the file in kilobytes and 2 represents the time to obtain a network connection. Thus, an 80K file will take $T(N) = \frac{80}{1.6} + 2 = 52$ seconds to download. We see that $T(N)$ is a linear function.

5. Common run-time functions:

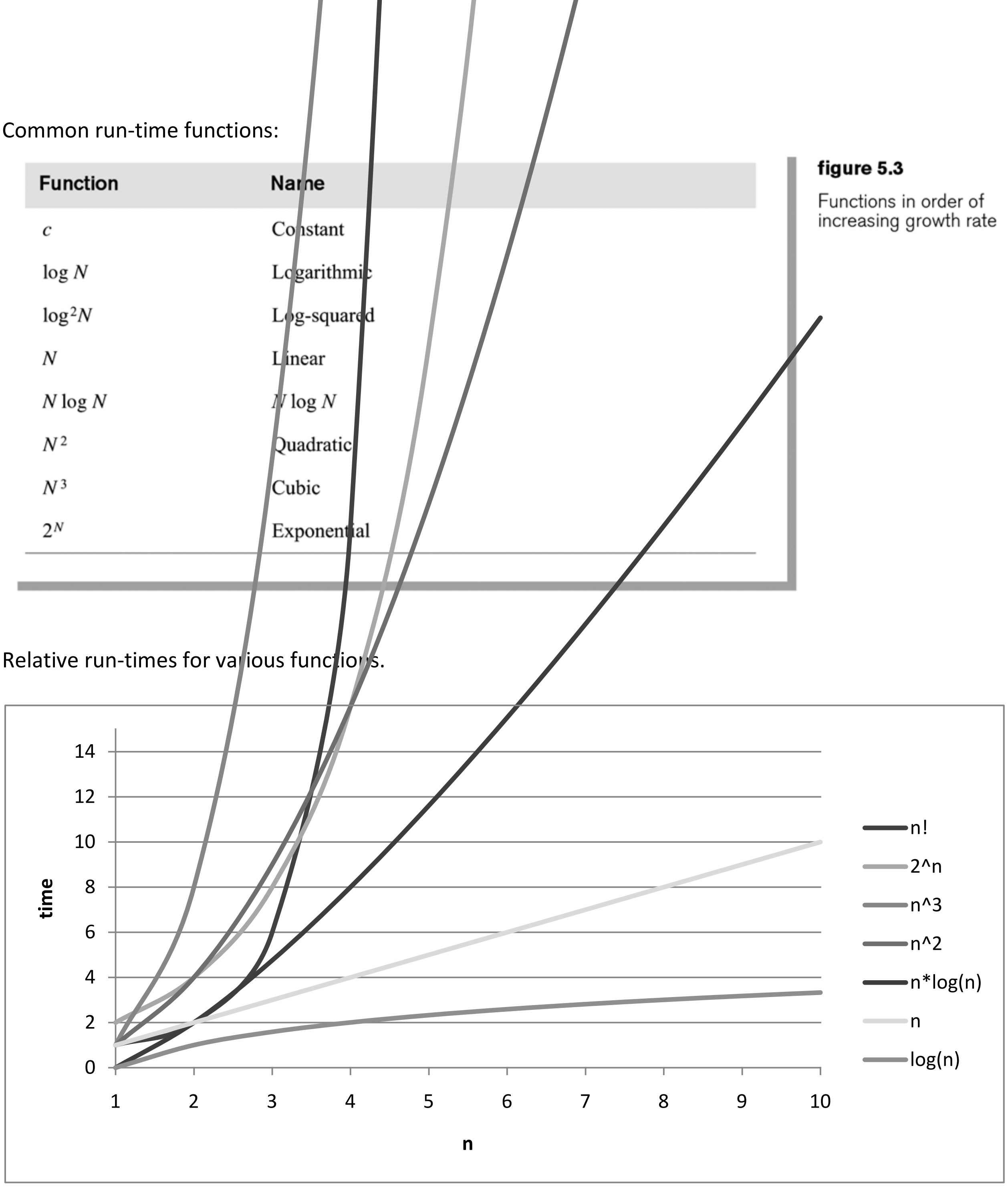


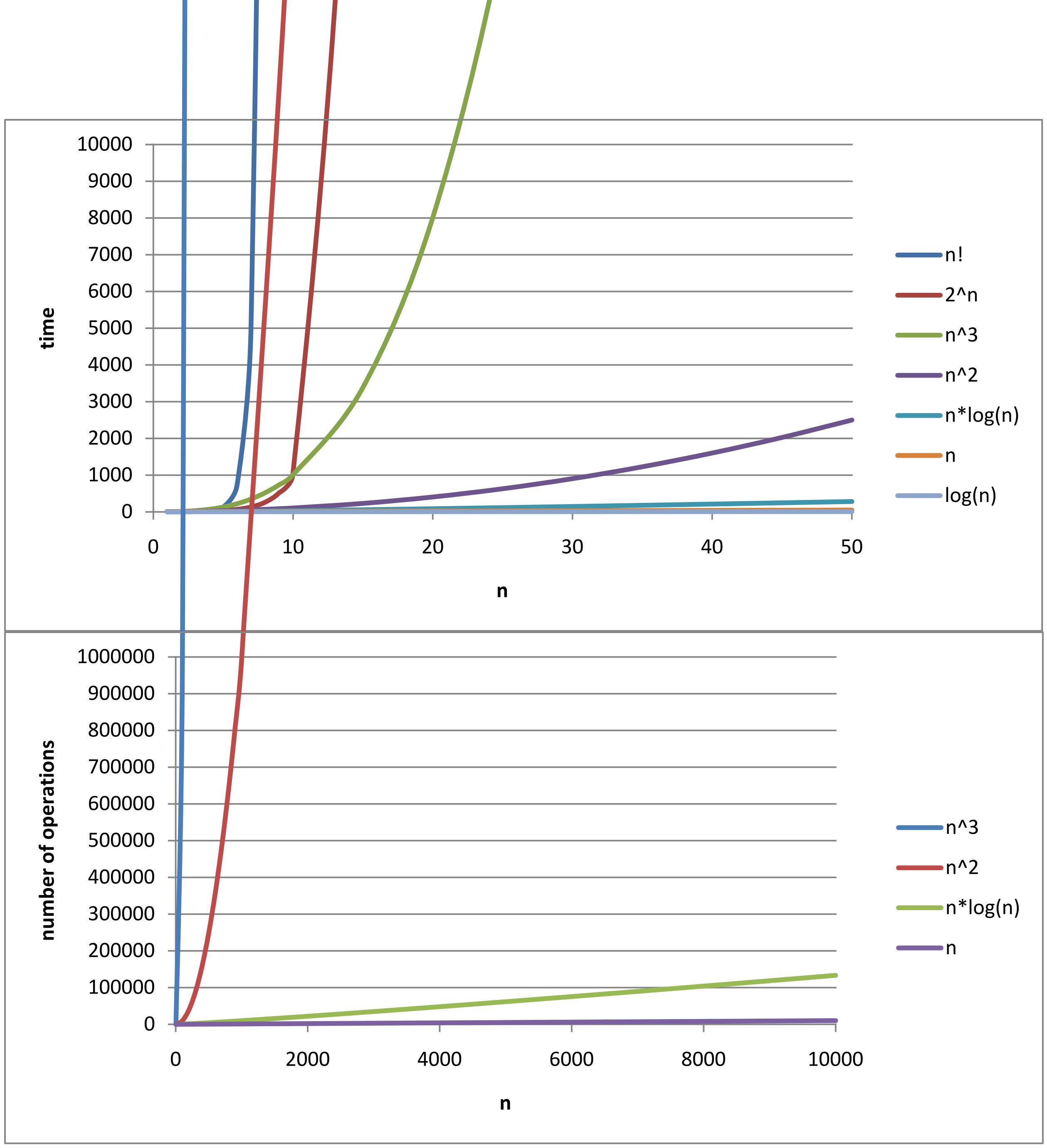
Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

figure 5.3

Functions in order of increasing growth rate

6. Relative run-times for various functions.





7. For large inputs, it is clear that from the figures above that a *log* algorithm would be the most efficient, followed by, linear, log-linear. Quadratic, cubic, exponential and factorial algorithms are progressively not feasible for large inputs.

8. A *constant* algorithm would be even more efficient than a *log* algorithm because the complexity would not depend on the input size. For instance, suppose an algorithm returns the smallest element in a sorted array. The algorithm is: return x[0]. Clearly, this algorithm doesn't depend on how many items there are in the array. Thus, we say that this algorithm has *constant complexity* or $O(c)\ or\ \ O(1)$.

Similarly, suppose we had an algorithm that returned the 5 smallest elements from a sorted array in another array. The algorithm:

```
int[] x = new int[5];

for( int i=0; i<5; i++ ) x[i] = array[i];
```

The dominant operation is the assignment, which occurs exactly 5 times. This too, is a constant time algorithm.

Homework 5.1

1. List the common complexities in order of preference.
2. Draw a graph of the common complexities.
3. Problem 5.4, p.217 (DS text)
4. Problem 5.5ab, p.217 (DS text)
5. Problem 5.19, p.217 (DS text).

Sections 23.1, 23.2 (IJP) 5.1 (DS) – What is algorithm analysis? (continued)

9. Suppose the run-time for an algorithm is: $T(N) = 10N^3 + N^2 + 40N + 80$. We say that such a function is a *cubic function* because the dominant term is a constant (10) times N^3. In algorithm analysis, we are concerned with the *dominant* term. For large enough *N*, the function's value is almost entirely dominated by the cubic term. For instance, when *N=1000*, the function has value 10,001,040,080 for which the cubic term contributed 10,000,000,000. We use the notation $O(N^3)$ to describe the complexity (growth rate) of this function. We read this, "order n-cubed.". This is referred to as *big-O* notation. Meaning the same thing, sometimes, we will say that this algorithm has cubic complexity.

10. Suppose you have a computer that can do 1 billion primitive operations per second. Consider how long it would take to run the following algorithms with the specified number of inputs.

n	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
100	0.000000002 *sec*	0.0000001 *sec*	0.0000002 *sec*	0.00001 *sec*	0.001 *sec*
1000	0.000000003 *sec*	0.000001 *sec*	0.000003 *sec*	0.001 *sec*	1 *sec*
10,000	0.000000004 *sec*	0.00001 *sec*	0.00004 *sec*	0.1 *sec*	17 *min*
100,000	0.000000005 *sec*	0.0001 *sec*	0.0005 *sec*	10 *sec*	12 *days*
1 million	0.000000006 *sec*	0.001 *sec*	0.006 *sec*	17 *min*	32 *yrs*
1 billion	0.000000009 *sec*	1 *sec*	9 *sec*	32 *yrs*	32 *billion yrs*

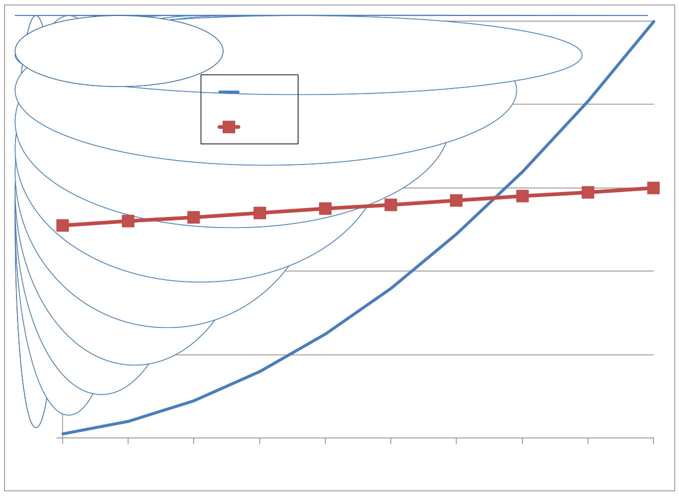
n	$O(2^n)$	$O(n!)$
10	0.000001 *sec*	0.003 *sec*
15	0.00003 *sec*	22 *min*
20	0.001 *sec*	77 *yrs*
25	0.03 *sec*	491 *million yrs*
30	1 *sec*	8411 *trillion yrs*
50	13 *days*	
75	1 *million yrs*	
100	40 *trillion yrs*	
1 million		
1 billion		

11. In this chapter, we address these questions:

- Is it always most important to be on the most efficient curve?
- How much better is one curve than another?
- How do you decide which curve an algorithm lies on?
- How do you design algorithms that avoid being on less-efficient curves?

12. As we saw earlier, as n gets large there is a clear indication of which curves are more efficient. However, when n is small, and time is a real concern, we may need to look more exactly at the time functions and the domain over which we would prefer one algorithm over another. Suppose we have two algorithms to solve the same problem, $F(n) = n + 50$ and $G(n) = n^2$. Clearly, we would prefer $F(n)$ for large *n*. However, suppose that *n* is always 5. From the curves below, we can see that $G(n)$ gives a faster execution. If the algorithm is run just once or so, this may not be a big deal. However, suppose the algorithm had to run 1 billion times then we would probably prefer $G(n)$. Suppose that primitive operations are performed 1 billion per second. Then, with *n=5*, algorithm *F* would require 55 seconds while *G* would require 25 seconds., at any point, either function can have a smaller value than the other. Thus, it usually doesn't make sense to say, $F(N) < G(N)$ for all *N*. For instance, consider these two functions:

n	$F(n) = n + 50$	$G(n) = n^2$
1	51	1
2	52	4
3	53	9
4	54	16
5	55	25
6	56	36
7	57	49
8	58	64
9	59	81
10	60	100



13. In algorithm analysis, we focus on the *growth rate* of the function, how much change there is in runtime when the input size increases. There are three reasons for this:

    a. The dominant term almost entirely dominates the value of the function for sufficiently large *N*.
    b. We usually are not concerned with an algorithm's run-time for small values of *N*.
    c. The leading constant of the dominant term is not meaningful across different computers.

Examples

1. An algorithm takes 3 sec. to run when the input size is 100. How long will it take to run when then input size is 500 when the algorithm is:

    a. linear: $\frac{3}{100} = \frac{x}{500} \quad \Longrightarrow \quad x = \frac{500}{100} * 3 = 5 * 3 = 15$, or 5 times longer

    b. quadratic: $\frac{3}{100^2} = \frac{x}{500^2} \quad \Longrightarrow \quad x = \frac{500^2}{100^2} * 3 = 5^2 * 3 = 75$, or 25 times longer

c. $O(\log n)$: $\frac{3}{\log 100} = \frac{x}{\log 500} \quad \Longrightarrow \quad x = \frac{\log 500}{\log 100} * 3 = 1.349 * 3 = 4.048$, or 1.349 times longer

2. An exponential algorithm takes 3 sec. to run when the input size is 5. How long will it take to run when then input size is 15?

   $\frac{3}{2^5} = \frac{x}{2^{15}} \quad \Longrightarrow \quad x = \frac{2^{15}}{2^5} * 3 = 2^{10} * 3 = 3072$, or $2^{10} = 1024$ times longer

3. How much longer does it take an exponential algorithm to run when you double the input size?

   $\frac{t}{2^n} = \frac{x}{2^{2n}} \quad \Longrightarrow \quad x = \frac{2^{2n}}{2^n} * t = 2^{2n-n} * t = 2^n * t$, or 2^n times longer

4. How much longer does it take an $O(n^3)$ algorithm to run when you double the input size?

   $\frac{t}{n^3} = \frac{x}{(2n)^3} \quad \Longrightarrow \quad x = \frac{(2n)^3}{n^3} * t = 2^3 * t$, or 8 times longer

5. Suppose an algorithm does exactly $T(n) = n^2$ operations and the computer can do one billion instructions per second. Suppose this algorithm can never have a runtime longer than one minute. How large can the input be?

   $\frac{1\ sec}{1{,}000{,}000{,}000\ instr} = \frac{60\ sec}{n^2} \quad \Longrightarrow \quad n^2 = \sqrt{60{,}000{,}000{,}000} = 244{,}948$

Homework 5.2

1. An algorithm takes 2 sec. to run when the input size is 1000.

   a. How long will it take to run when then input size is 50,000 (in hours)?
   b. How much longer will it take to run?

2. Consider doubling the size of the input for a logarithmic algorithm. A useful identity is $\log(xy) = \log(x) + \log(y)$.

   a. How much longer does it take?
   b. What happens to this value as n gets very large?
   c. How much longer when the original size of the problem is 1,000,000?

3. Problem 5.14, p.217 (DS text).
4. Problem 5.15 p.217 (DS text).

Section 5.4 – General big-oh rules

1. Definition: Let $T(n)$ be the exact run-time of an algorithm. We say that $T(n)$ is $O(F(n))$ if there are positive constants c and n_o such that $T(n) \leq cF(n)$ when $n > n_o$. This says that at some point n_o, for every n past this point, the function $T(n)$ is bounded by some multiple of $F(n)$.

2. Example:

   Suppose $T(n) = 5n^2 + 16$ and $F(n) = n^2$. Thus, we can say that $T(n)$ is $O(n^2)$ because $T(n) \leq cF(n)$, for all $n_o \geq 1$ when $c = 21$. Proof:

$$T(n) \leq cF(n) \quad \Longrightarrow \quad c \geq \frac{T(n)}{F(n)} = \frac{5n^2+16}{n^2} = 5 + 16/n^2$$

Thus, when $n = 1$, $c \geq 21$ works.

Homework 5.3

1. Use the definition of Big-oh to show that

   a. $T(n) = 3n^2 + 4n + 6$ is $O(n^2)$

   b. $T(n) = e^{n+4}$ is exponential.

Section 5.4 – General big-oh rules (continued)

2. Using big-oh notation, we do not include constants or lower order terms. Thus, in the example above, the algorithm is $O(n^2)$.

3. The running time of a loop is at most the running time of the statements inside the loop times the number of iterations.

4. Examples: How many total statement executions are there?

Algorithm	**Time**	**Complexity**

for(i=1 to n)
    statement 1
	$T(n) = 1 + 1 + \cdots + 1 = n$	$O(n)$

for(i=1 to n)
    statement 1
    statement 2
	$T(n) = 2 + 2 + \cdots + 2 = 2n$	$O(n)$

for(i=1 to n)
    statement 1
statement 2
	$T(n) = n + 1$	$O(n)$

for(i=1 to n)
    statement 1
for(i=1 to n)
    statement 2
	$T(n) = n + n = 2n$	$O(n)$

for(i=1 to 2*n)
    statement 1
	$T(n) = 2n$	$O(n)$

for(i=1 to n/4)
    statement 1
	$T(n) = n/4 = \frac{1}{4}n$	$O(n)$

for(i=1 to n)
    for(j=1 to 1000)
        statement 2
	$T(n) = 1000 + 1000 + \cdots 1000 = 1000n$	$O(n)$

Algorithm	Time	Complexity

for(i=1 to n)
    for(j=1 to n)
        statement 2
	$T(n) = n + n + \cdots n = n^2$	$O(n^2)$

for(i=1 to n)
    statement 1
    for(j=1 to n)
        statement 2
	$T(n) = (n+1) + (n+1) + \cdots (n+1) = n(n+1) = n^2 + n$	$O(n^2)$

for(i=1 to n*n)
    statement 1
	$T(n) = n^2$	$O(n^2)$

for(i=1 to n*n/4)
    statement 1
	$T(n) = n^2/4$	$O(n^2)$

for(i=1 to 10)
    for(j=1 to n)
        statement 1
for(i=1 to n)
    for(j=1 to n)
        statement 2
for(i=51 to 100)
    statement 1
	$T(n) = [n + \cdots + n] + n^2 + 50 = 10n + n^2 + 50$	$O(n^2)$

for(i=1 to n)
    for(j=1 to n*n)
        statement 2
	$T(n) = n(n^2) = n^3$	$O(n^3)$

Homework 5.4

1. Let *n* represent the size of the input, *x.length* in the following method.

```
public static double stdDev( double[] x )
{
    double sum=0, avg, sumSq=0, var, sd;

    for( int i=0; i<x.length; i++ )
       sum += x[i];

    avg = sum / x.length;

    for( int i=0; i<x.length; i++ )
       sumSq += Math.pow( x[i]-avg, 2 );

    sd = Math.pow( sumSq / (x.length-1),
0.5 );

    return sd;
}
```

   a. Provide an expression for $T(n)$, the number of additions performed by this algorithm.
   b. Provide the complexity of this algorithm and justify your answer.

2. Let *n* represent the size of the input, *x.length* in the following method.

```
public static double stdDev2( double[] x )
{
   double sum=0, avg, sumSq=0, var, sd;

   for( int i=0; i<x.length; i++ )
   {
      sum = 0.0;
      for( int j=0; j<x.length; j++ )
         sum += x[j];

      avg = sum / x.length;
      sumSq += Math.pow( x[i]-avg, 2 );

   }

   sd = Math.sqrt( sumSq / (x.length-1));

   System.out.println( sd );

   return sd;
}
```

   a. Provide an expression for $T(n)$, the number of additions performed by this algorithm.
   b. Provide the complexity of this algorithm and justify your answer.

3. In the following method, *y* is a square matrix so let *n* represent the number of rows (or columns). The method, *stdDev* is defined in Problem 1. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static void analyze( double[][] y )
{
   for( int i=0; i<y.length; i++ )
   {
     double[] x = y[i];

      System.out.println( stdDev( x ) );
   }
}
```

4. In the following method, *x* and *y* are square matrices. Thus, we let *n* represent the number of rows (or columns) in either matrix. The method, *matMultiply* multiplies the two matrices and returns the result. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static double[][] matMultiply(
           double[][] x, double[][] y )
{
    int n = x.length;
    double[][] z = new double[n][n];

    for( int i=0; i<n; i++ )
       for( int j=0; j<n; j++ )
          for( int k=0; k<n; k++ )
              z[i][j] += x[i][k] * y[k][j];
    return z;
}
```

5. Let *n* be the length of the input array, *x* in the method below. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static double skipper( double[] x )
{
   double sum=0;

   int incr = (int)Math.sqrt( x.length );

   for( int i=0; i<x.length; i+=incr )
      sum += x[i];
   return sum;
}
```

6. Let *n* be the length of the input array, *x* in the method below. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static double dopper( double[] x )
{
   double sum=0, sum2;

   int incr = (int)Math.sqrt( x.length );

   for( int i=0; i<x.length; i++ )
   {
      sum += x[i];

      sum2 = 0;
      for( int j=0; j<x.length; j+=incr )
         sum2 += x[j];

      sum -= sum2;
   }
   return sum;
}
```

7. Let *n* be the length of the input array, *x* in the method below. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static double lantern( double[] x )
{
   double sum=0, sum2=0;

   int incr = (int)Math.sqrt( x.length );

   for( int j=0; j<x.length; j+=incr )
      sum2 += x[j];

   for( int i=0; i<x.length; i++ )
      sum += x[i] - sum2;

   return sum;
}
```

8. Let *n* be the length of the input array, *x* in the method below. Provide the complexity of this algorithm, in terms of *n* and justify your answer for the (a) best case, (b) worst case, (c) average case.

```
public static double horn( double[] x )
{
   double sum=0, sum2=0;

   int n = x.length;

   for( int i=0; i<n-1; i++ )
      if( x[i] < x[i+1] )
         for( int j=0; j<n; j++ )
            sum += j;
      else
         for( int j=0; j<n; j++ )
            for( int k=0; k<n; k++ )
               sum += i*j;

   return sum;
}
```

9. Problem 5.7, p.217 (DS text).

Section 23.2 (IJP) – Useful Mathematical Identities

These will be useful as we consider complexity further:

- $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$
- $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + (n-1)^2 + n^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=0}^{n} a^i = a^0 + a^1 + a^2 + \cdots + a^{(n-1)} + a^n = \frac{a^{n+1}-1}{a-1}$

  Special Case: $\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \cdots + 2^{(n-1)} + 2^n = \frac{2^{n+1}-1}{2-1} = 2^{n+1} - 1$
- $\sum_{i=1}^{n} 1/i = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} + \frac{1}{n} \approx O(\log n)$

More Examples

Sometimes, we have to be more careful in our analysis.

1. Example: Consider this algorithm:

```
for(i=1; i<=n; i++)
     for(j=1; j<=i; j++)
          k += i+j;
```

The outer loop executes *n* times. However, the inner loop executes a number of times that depends on *i.*

i	Num. Executions
1	1
2	2
3	3
…	…
n	n

Thus, the complexity is:

$$1 + 2 + 3 + \cdots + (n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

2. Example: Consider this algorithm:

```
for(i=1; i<=n; i++)
     for(j=1; j<i; j++)
          k += i+j;
```

The outer loop executes *n* times and again, the inner loop executes a number of times that depends on *i.*

i	Num. Executions
1	0
2	1
3	2
…	…
n	n-1

Thus, the complexity is:

$$1 + 2 + 3 + \cdots + (n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{n(n-1)}{2} = O(n^2)$$

3. Example: Consider this algorithm. Exactly how many times is statement1 executed?

```
for(i=1; i<=n; i++)
     for(j=1; j<=i*i; j++)
          statement1;
```

Answer: $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + (n-1)^2 + n^2 = \frac{n(n+1)(2n+1)}{6}$ times. Thus, the algorithm has complexity $O(n^3)$. We could also have gotten this complexity result by inspection as we see that the inner loop can executed up to n^2 times and the outer loop executes exactly n times.

4. Example: Consider an algorithm to compute a^n for some value of *n*. Thus, a simple algorithm is to multiply *a* by itself *n* times:

   ```
   res=1;
   for(i=1; i<=n; i++)
         res *= a;
   ```

   Thus, *n* multiplications are carried out and the complexity is $O(n)$. Or, you could:

   ```
   res=a;
   for(i=1; i<n; i++)
         res *= a;
   ```

   Thus, *n-1* multiplications are carried out, but the complexity is still $O(n)$.

   Suppose that $n = 2^k$, for some integer value of *k*. For example, suppose that we want to compute a^{16}. Thus, we see that $a^{16} = a^{2^4}$ and, thus, *k=4*. Now, consider the following multiplications:

k	
1	$a * a = a^2$
2	$a^2 * a^2 = a^4$
3	$a^4 * a^4 = a^8$
4	$a^8 * a^8 = a^{16}$

   This suggests the algorithm:

   ```
   res=a;
   for(i=1; i<=k; i++)
         res *= res;
   ```

   So, what is the complexity of this algorithm?

   k multiplications take place. But, what is *k* in terms of *n*, the input size?

   $$n = 2^k \implies \log n = \log 2^k \implies \log n = k \log 2 \implies k = \log n / \log 2 \implies O(\log n)$$

Homework 5.5

1. Let *n* represent the size of the input, *x.length* in the following method. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static double orb( double[] x )
{
   double sum=0;

   int beg = -x.length + 1;

   for( int i=beg; i<x.length; i++ )
   {
      sum += x[Math.abs(i)] * i;
      System.out.println( i + ", " + sum );

   }
   return sum;
}
```

2. Let *n* represent the size of the input, *x.length* in the following method. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static void heunt( double[] x )
{
    int i=0;
    while( i < x.length )
    {
        x[i] -= i;
        i += 2;
     }
     i=1;
     while( i < x.length )
     {
        x[i] += i;
        i += 2;
     }
}
```

3. Let *n* represent the size of the input, *x.length* in the following method. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static void jerlp( double[] x )
{
   double sum = 0.0;
   int k = (int)Math.sqrt( x.length );

   for( int i=0; i<k; i++ )
      for( int j=0; j<=i; j++ )
         sum += x[i]*x[j];
}
```

4. Problem 5.11, p.217 (DS text).

Section 5.2 – Examples of algorithm running times

1. Find the minimum value in an array of size *n*. To find the minimum, we initialize a variable to hold the first value and then subsequently check each other value to see if it is less than the minimum, updating the minimum as needed.

```
min = x[0];

for( int i=1; i<n; i++)
{
   if( x[i] < min )
      min = x[i];
}
```

In this case we say that *comparisons* are the dominant operation and we can see that $n-1$ comparisons are made so we say that the algorithm is $O(n)$.

2. Given *n* points in 2-d space, find the closest pair.

   Rough Algorithm:

Find distance between first point and the 2nd, 3rd, 4th, 5th, *etc*.	n-1 calculations
Find distance between second point and the 3rd, 4th, 5th, *etc*.	n-2 calculations
Find distance between third point and the 4th, 5th, *etc*.	n-3 calculations
...	
Find distance between next-to-last point and last	1 calculation.

   Refined Algorithm:

```
min = dist(x[1],x[2])
for( i=1 to n-1)
        for( j=i+1 to n)
                d = dist(x[i],x[j])
                if( d < min )
                        store i,j
                        min = d
```

i	j	Num. Executions
1	2,n	n-2+1=n-1
2	3,n	n-3+1=n-2
3	4,n	n-4+1=n-3
...		...
n-2	n-1,n	n-(n-1)+1=2
n-1	n,n	1

Thus, $(n-1)+(n-2)+\cdots+2+1$ distances are calculated. So that the complexity is

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \approx O(n^2)$$

There exist algorithms for this problem that take $O(n \log n)$ and $O(n)$, respectively, but are beyond the scope of this course.

3. Given *n* points in 2-d space, find all sets of three points that are collinear (form a straight line).

```
for( i=1 to n-2 )
   for( j=i+1 to n-1 )
         for( k=j+1 to n )
               if( areCollinear( x[i], x[j], x[k] ) display x[i], x[j], x[k]
```

   It can be shown that $\frac{n(n-1)(n-2)}{6} = \frac{n^3-3n^2+2n}{6}$ calls are made to *areCollinear*. Thus, this algorithm is $O(n^3)$.

4. Consider this algorithm:

```
for( i=1; i<=n; i++ )
   for( j=1; j<=n; j++ )
         if( j%i == 0 )
               for( k=1; k<=n; k++ )
                     statement;
```

What is the complexity of this algorithm? How many times is *statement* executed? First, we see that the inner most loop is $O(n)$. Second, we see that the *if* statement is evaluated exactly n^2 times. This might lead us to say, then, that the complexity is $O(n) * O(n^2) = O(n^3)$. However, this is false because the inner loop is selectively evaluated. For instance, when i=1, the *if* statement is always true, thus the inner most loop occurs n times.

i	j	j%i==0	Num. Executions of inner loop	Total
1	1	True	1	n
	2	True	1	
	3	True	1	
	…			
2	1	False	0	$\lfloor \frac{n}{2} \rfloor$
	2	True	1	
	3	False	0	
	4	True	1	
	…			
3	1	False	0	$\lfloor \frac{n}{3} \rfloor$
	2	False	0	
	3	True	1	
	4	False	0	
	5	False	0	
	6	True	1	
	…			
	…			

Thus, the inner most loop is executed:

$$n + \lfloor \tfrac{n}{2} \rfloor + \lfloor \tfrac{n}{3} \rfloor + \lfloor \tfrac{n}{4} \rfloor + \lfloor \tfrac{n}{n-1} \rfloor + \lfloor \tfrac{n}{n} \rfloor < n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots + \frac{n}{n-1} + \frac{n}{n}$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} + \frac{1}{n}\right) = n \sum_{i=1}^{n} \frac{1}{i} \text{ times.}$$

From Theorem 5.5, we see that $\sum_{i=1}^{n} \frac{1}{i}$ has complexity $O(\log n)$. Thus, the inner most loop is executed $O(n \log n)$ times. Finally, since the inner most loop is $O(n)$, then the complexity of the algorithm is: $O(n) * O(n \log n) = O(n^2 \log n)$.

5. Consider Problem 5.16 from the text:

```
for( i=1; i<=n; i++ )
   for( j=1; j<=i*i; j++ )
         if( j%i == 0 )
               for( k=0; k<=j; k++ )
                     sum++;
```

i	j	j%i==0	Num. Executions of inner loop
1	1	True	1
2	1	False	0
	2	True	1
	3	False	0
	4	True	1
3	1	False	0
	2	False	0
	3	True	1
	4	False	0
	5	False	0
	6	True	1
	7	False	0
	8	False	0
	9	True	1
	...		

By inspection, we might say that the outer loop is $O(n)$, the middle loop is $O(n^2)$, and the inner most loop is $O(n^2)$, for a total complexity of $O(n^5)$. However, this is false because the *if* statement selectively allows the inner most loop to execute.

Careful examination of the *if* statement reveals that it allows access to the inner loop exactly *i* times for each value of *i*, as demonstrated in the table at left. So, the inner loop is executed $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$ times. The inner most loop itself also has complexity $O(n^2)$. Thus, the total complexity is $O(n^2) * O(n^2) = O(n^4)$. So, effectively, the first three statements have a complexity of $O(n^2)$.

Homework 5.6

1. Let *n* be the length of the input array, *x* in the method below. Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static void bleater( double[] x )
{
   double sum=0, avg, lastAvg=0;

   int n = x.length;

   for( int i=0; i<n; i++ )
   {
      sum = 0;
      for( int j=0; j<n/(i+1); j++ )
         sum += j;
      avg = sum / (int)(n/(i+1));
      System.out.println( avg );

   }
}
```

2. Let *n* be the length of the input array, *x* in the method below. (a) Exactly how many times is the line: `x[i] += m;` executed in terms of *n*? (b) Provide the complexity of this algorithm, in terms of *n* and justify your answer.

```
public static void orange( double[] x )
{
   int n = x.length;

   for( int i=0; i<n; i++ )
   {
      long j=0;
      long k = (long)Math.pow(2,i);
      while( j < k )
      {
         long m = ( j%2==0 ? j : -j);
         x[i] += m;
         j++;
      }
   }
   System.out.println( bigCount );
}
```

Section 5.3 – The maximum contiguous subsequence sum problem

1. Maximum contiguous subsequence sum problem (MCSSP): Given (possibly negative) integers $A_1, A_2, \dots, A_n$, find (and indentify the sequence corresponding to) the maximum value of $\sum_{k=i}^{j} A_k$. If all the integers are negative, then the sum is defined to be zero. In other words, find *i* and *j* that maximize the sum.

2. Example – Consider these values for the MCSSP: -2, 11, -4, 13, -5, 2

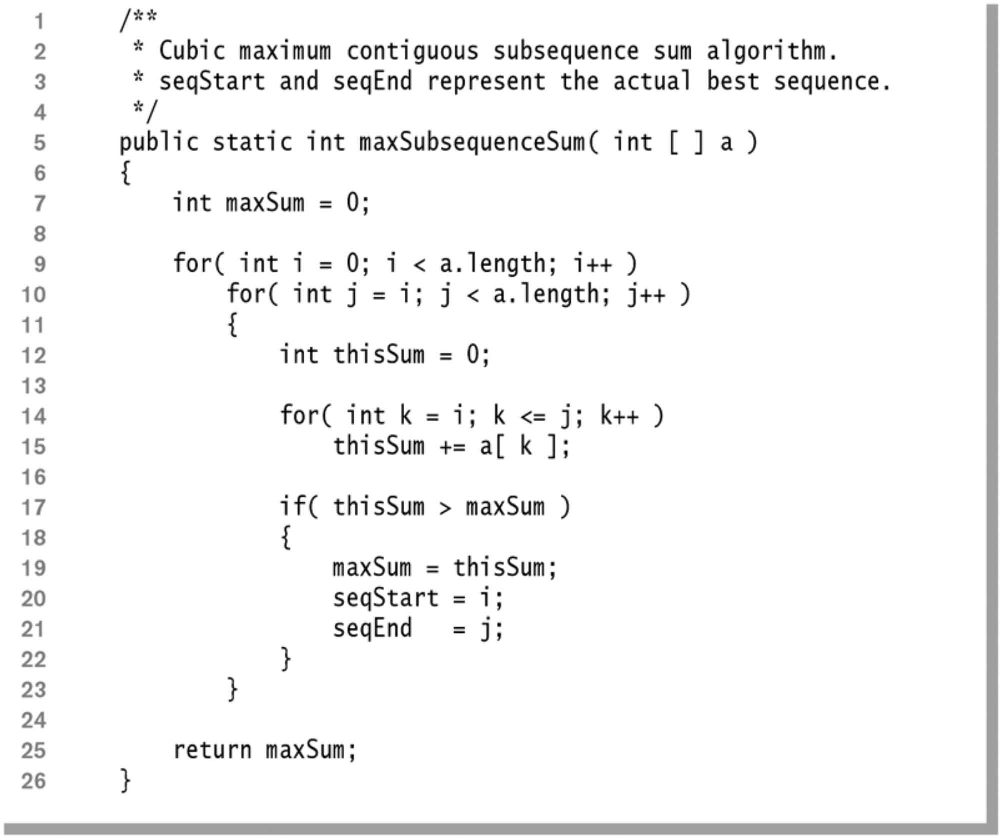
   The simplest algorithm is to do an exhaustive search of all possible combinations of contiguous values. Sometimes we call such an approach a *brute-force algorithm*.

1st	1st, 2nd	1st, 2nd, 3rd	1st, 2nd, 3rd, 4th	*etc.*
	2nd	2nd, 3rd	2nd, 3rd, 4th	*etc.*
		3rd	3rd, 4th	*etc.*
			etc.	*etc.*

   For this example, we find:

-2	-2+11=9	-2+11-4=5	-2+11-4+13=18	-2+11-4+13-5=13	-2+11-4+13-5+2=15
	11	11-4=7	**11-4+13=20**	11-4+13-5=15	11-4+13-5+2=17
		-4	-4+13=9	-4+13-5=4	-4+13-5+2=6
			13	13-5=8	13-5+2=10
				-5	-5+2=-3

3. An $O(n^3)$ algorithm for MCSSP is shown below implements the exhaustive search. The first loop determines the starting index for the sequence, the second loop determines the ending index for the sequence, and the third loop sums the values in the sequence. In big-oh analysis we focus on a dominant computation, one that is executed the most times. In this case, it is the addition at line 15. Also, it is not necessary to determine the exact number of iterations of some code. It is simply necessary to bound it. So, in the algorithm above, we see that the first loop can execute *n* times, the second (nested) loop can execute *n* times, and the inner loop can execute *n* times. Thus, the complexity is $O(n^3)$.



```
 1      /**
 2       * Cubic maximum contiguous subsequence sum algorithm.
 3       * seqStart and seqEnd represent the actual best sequence.
 4       */
 5      public static int maxSubsequenceSum( int [ ] a )
 6      {
 7          int maxSum = 0;
 8
 9          for( int i = 0; i < a.length; i++ )
10              for( int j = i; j < a.length; j++ )
11              {
12                  int thisSum = 0;
13
14                  for( int k = i; k <= j; k++ )
15                      thisSum += a[ k ];
16
17                  if( thisSum > maxSum )
18                  {
19                      maxSum = thisSum;
20                      seqStart = i;
21                      seqEnd   = j;
22                  }
23              }
24
25          return maxSum;
26      }
```

figure 5.4

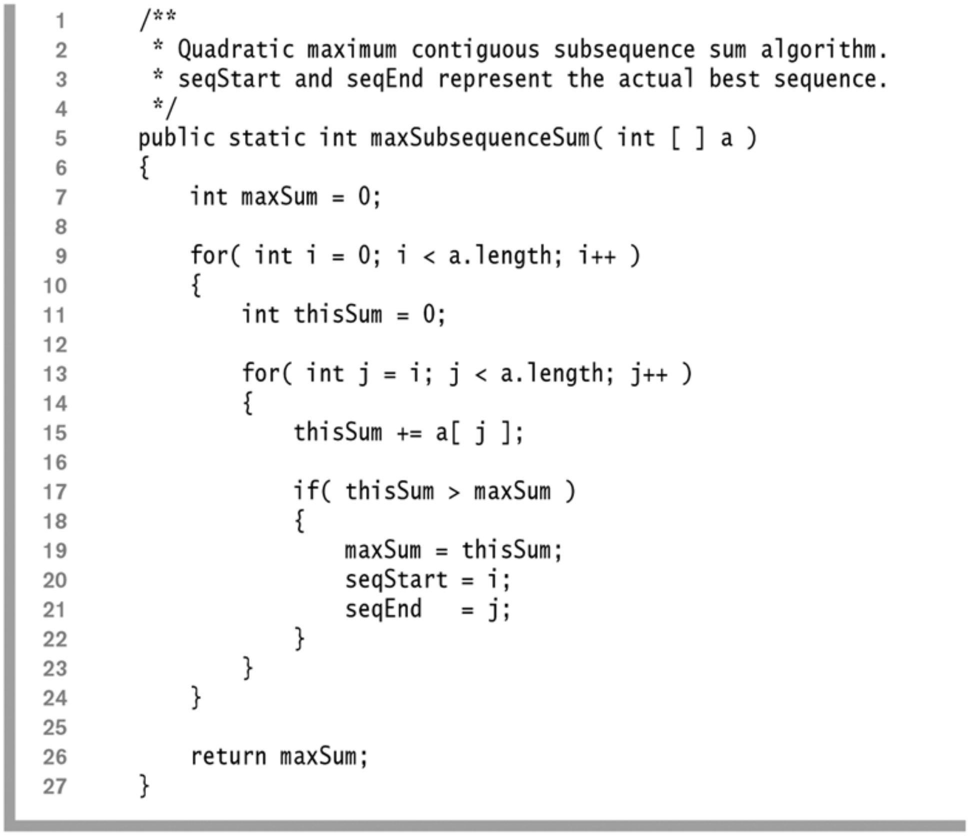
A cubic maximum contiguous subsequence sum algorithm

4. We can improve this algorithm to $O(n^2)$ by simply observing that the sum of a new subsequence is simply the sum of the previous subsequence plus the next piece. In other words, we don't need the inner loop (lines 14 and 15), and simply move line 15 to replace line 12.

-2	-2+11=9	9-4=5	5+13=18	18-5=13	13+2=15
	11	11-4=7	**7+13=20**	20-5=15	15+2=17
		-4	-4+13=9	9-5=4	4+2=6
			13	13-5=8	8+2=10
				-5	-5+2=-3

figure 5.5

A quadratic maximum contiguous subsequence sum algorithm



```java
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    int maxSum = 0;

    for( int i = 0; i < a.length; i++ )
    {
        int thisSum = 0;

        for( int j = i; j < a.length; j++ )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd   = j;
            }
        }
    }

    return maxSum;
}
```

5. Improving the performance of an algorithm often means getting rid of a (nested) loop. This is not always possible, but clever observation and taking into account the structure of the problem can lead to this reduction sometimes.

6. There are a number of observations we can make about this problem. First, if any subsequence has sum < 0, then any larger subsequence that includes the negative subsequence cannot be maximal, because we can always remove the negative subsequence and have a larger value. This tells us that we can break from the inner loop when we find a negative subsequence. However, this does not reduce the complexity.

7. Another observation we can make is that when we encounter a negative subsequence, we only need to continue looking at subsequences that begin after the negative subsequence. This leads to an $O(n)$ algorithm

8. Example – Consider these values for the MCSSP: -2, 11, -4, 13, -5, 2

-2	11	11-4=7	7+13=20	20-5=15	15+2=17

9. Example – Consider these values for the MCSSP: 1, -3, 4, -2, -1, 6

1	1-3=-2	4	4-2=2	2-1=1	1+6=7

10. An $O(n)$ algorithm for the MCSSP.

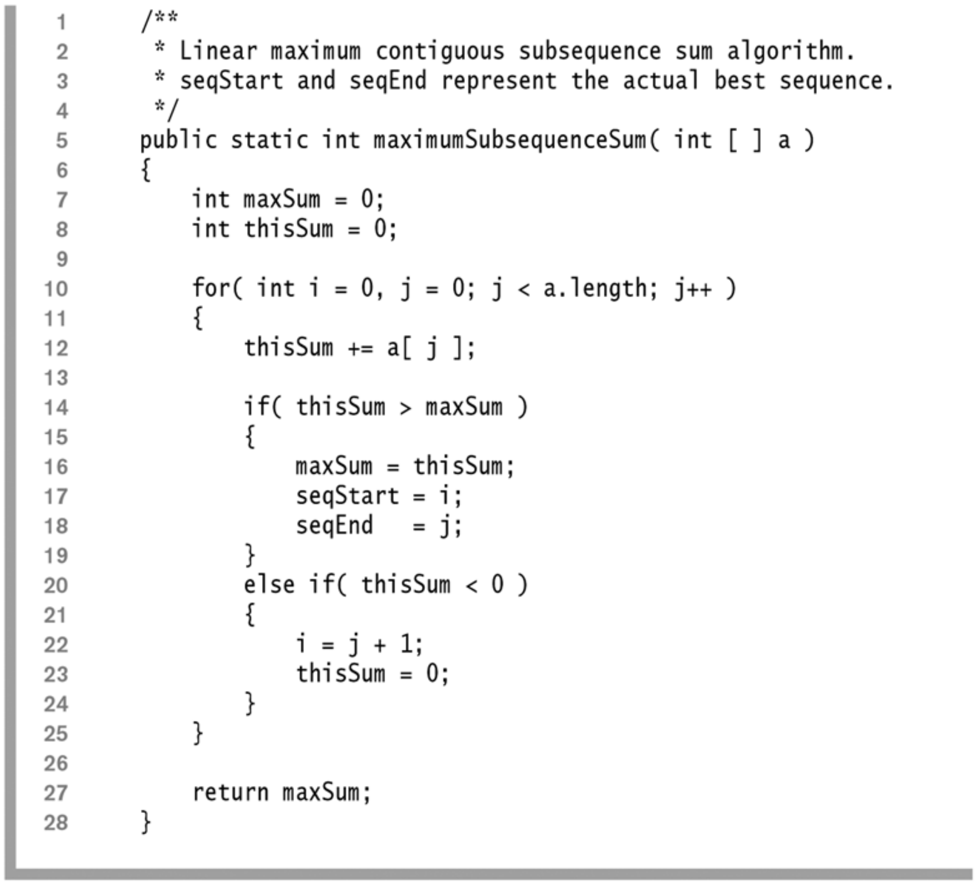


figure 5.8

A linear maximum contiguous subsequence sum algorithm

```java
/**
 * Linear maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maximumSubsequenceSum( int [ ] a )
{
    int maxSum = 0;
    int thisSum = 0;

    for( int i = 0, j = 0; j < a.length; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }

    return maxSum;
}
```

Section 5.5 – The Logarithm

1. Definition of a logarithm: $\log_a n = b$ if $a^b = n$, for $n > 0$, where a is referred to as the *base* of the logarithm. In CS we will assume the base is 2 unless otherwise stated. However, it can be proved that the base is unimportant when dealing with complexity.

2. *Repeated Halving Principle* – How many times can you half *n* until you get to 1 or less? The answer is (roughly) $\log n$.

   Consider $n = 100$, so that $\log 100 = 6.64 \approx 7$

Count	Result
1	100/2=50
2	50/2=25
3	25/2=12.5
4	12.5/2=6.25
5	6.25/2 = 3.125
6	3.125/2 = 1.5625
7	1.5625/2 = 0.78125

   This is especially useful in CS because we encounter algorithms that repeatedly divide a problem in half. The *repeated doubling principle* is has the same answer: start with $x = 1$, how many times can you double x until you reach n or more?

Section 5.6 – Searching & Sorting

1. **Sequential Search** – Search an unordered collection for a key. In worst case you must search all elements. Thus, $O(n)$.

2. **Binary Search** (23.4 IJP, 5.6 DS) – If the items in the collection are ordered, we can do a binary search. At each iteration, we determine whether the key is to the left of the middle, the middle, or to the right of the middle. Thus, each iteration of binary search involves either 1 or 2 comparisons which is $O(c)$. If the item is not in the middle, then we use only the left (or right) sublist in the subsequent iteration. So, how many iterations are performed? In the worst case, we repeatedly half the list until there is only a single item left. By the repeated halving principle, this can occur at most $\log n$ times. Thus, the binary search algorithm is $O(\log n)$.

4. **Selection Sort** – The idea is to find the largest element in the array and swap it with the value in the last position. Next, find the next largest value and swap it with the value in the next-to-last position, etc. In this way, we build a sorted array

   The first iteration involves n-1 comparisons and possibly 1 move, the second takes n-2 comparisons and possibly one move, *etc*. Thus,

   Thus, $(n-1)+(n-2)+\cdots+2+1$ comparisons are made and up to *n* moves are made.

   We remember that: $\sum_{i=1}^{p} i=\frac{p(p+1)}{2}$, so that: $\sum_{i=1}^{n-1} i=\frac{(n-1)(n-1+1)}{2}=\frac{n(n-1)}{2}=\frac{n^2-n}{2}=\frac{1}{2}n^2-\frac{1}{2}n$ comparisons are made. Thus, the complexity is $O(n^2)$. Note that even in the best case when the data is initially sorted, it still takes $\frac{1}{2}n^2-\frac{1}{2}n$ comparisons. Thus, the best, average, and worse case complexity are all the same, $O(n^2)$.

5. **Insertion Sort** (IJP 23.6) – The basic idea is that at each iteration you have a sorted sublist and you put the next (unsorted) value in this sorted sublist. It begins by putting choosing the first element to initialize the sorted sublist and then looks at the second value and determines whether it goes in front of the first value or after. At completion of this iteration, the sublist of size 2 is sorted. Next, we take the third value and determine where it goes in the sorted sublist, *etc*. Thus, we see that at each iteration, we have to do one more comparison than the previous iteration.

   How many *comparisons* are required for Insertion Sort assuming an array with *n* elements?

   The algorithm will make *n-1* passes over the data. The first pass makes 1 comparison, the 2nd item with the 1st. The second pass, in the worst case, requires 2 comparisons, item 3 with the 2 sorted items, *etc*. Thus, $1+2+\cdots+(n-2)+(n-1)=\frac{1}{2}n^2-\frac{1}{2}n$ comparisons are made in the worst case. Using comparisons as the dominant operation, the complexity is $O(n^2)$.

   If we consider the swaps as the dominant operation, the complexity is also $O(n^2)$. This is so because the maximum number of swaps is the same as the maximum number of comparisons.

   However notice that in the best case, when all the numbers are already in order, that *n-1* comparisons are made and no moves. Thus, best-case complexity is $O(n)$. On average, Insertion Sort is somewhere in between best case and worst case. This also suggests that if the data is mostly in order, but needs to be sorted, and we must choose between Selection sort and Insertion sort, we might choose Insertion.

Homework 5.7

1. In the method below, (a) provide the exact number of times *x* is modified, (b) provide and justify he complexity.

```
public static double gorwel( double x )
{
   while( x > 1 ) x /= 3;
   return x;
}
```

2. Suppose a method takes a one-dimensional array of size *n* as input. The method creates a two-dimensional array from the original one-dimensional array. Assume this takes $O(n^2)$. Next, the method loops through all the elements in the two-dimensional array doing a binary search of the original array for each one. Provide and justify the complexity of this method.

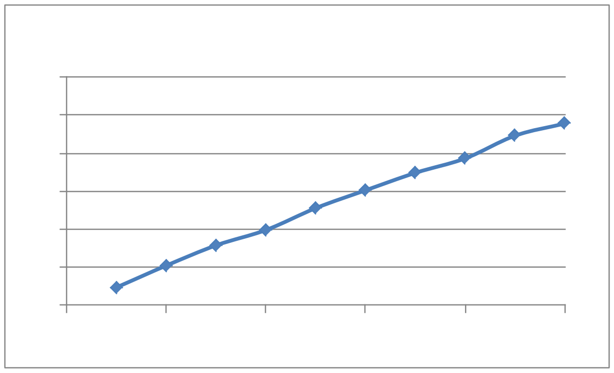
3. Problem 5.8, p.217 (DS text).

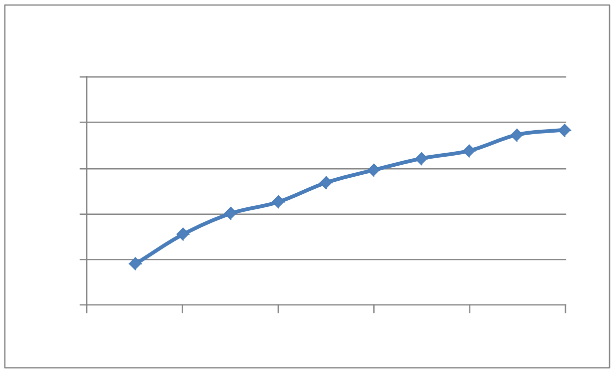
Section 5.7 – Checking an Algorithm Analysis

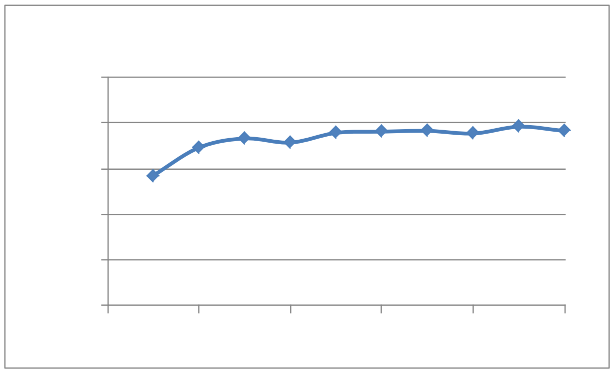
1. Let $T(n)$ be the empirical running time of an algorithm and let $F(n)$ be the proposed complexity, $O(F(n))$. If $F(n)$ is correct, then $T(n)/F(n)$ should converge to a positive constant, as n gets large. If $F(n)$ is an underestimate, then $T(n)/F(n)$ will diverge. If $F(n)$ is an overestimate, then $T(n)/F(n)$ will converge to zero.

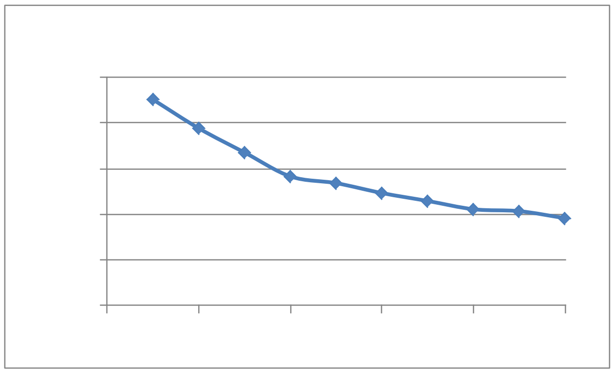
2. The graphs below show empirical results for this algorithm:

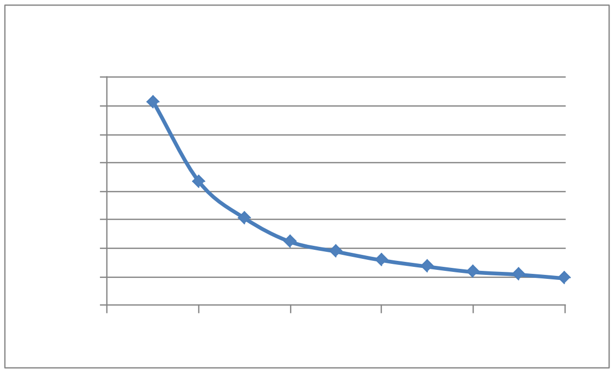
```
for( int j=0; j<n; j++ )
{
   System.out.println( j );
}
```





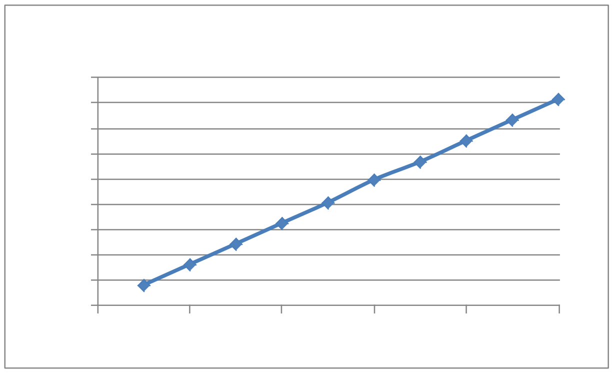


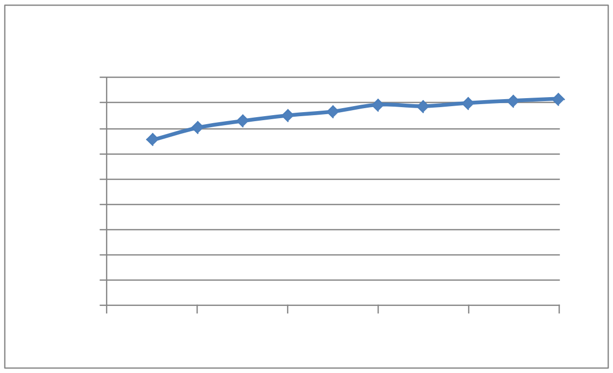


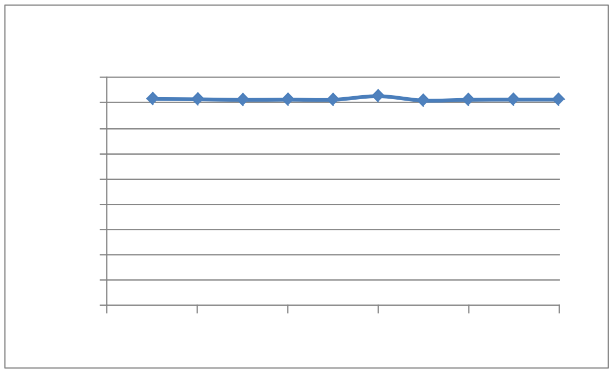


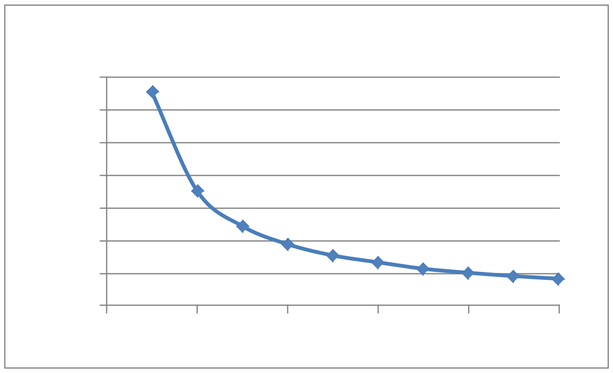
3. The graphs below show empirical results for this algorithm:

```
for( int j=0; j<n; j++ )
{
   double x = j;
   while( x > 1 ) x /= 2;
}
```









4. A brute force algorithm for computing an integer power was shown earlier and it was seen that it was linear. Java has a built-in power function: Math.pow(double, double). Finally, Patricia Shanahan proposed an algorithm closely related to one Knuth originally published, which is very fast, but works for integer exponents only.

   The graphs below show empirical results for these algorithms. The setup for the brute force experiment was a random double was generated between 1 and 100 raised to integer power 0. This was repeated 1,000,000 times to obtain an average power shown along the horizontal axis. This was then repeated for powers 1,...,30. Next, the same setup was used for Shanahan's Algorithm and Math.pow. Finally, Math.pow was also considered with the same setup, except that the exponents were random doubles between p-1 and p+1, where p is the value on the horizontal axis. For instance, when p=20, the exponent was a random double between 19 and 21.

   The second graph shows the results for large, integer exponents.