1. Problem 1-1)
In general, this problem requires f(n) = some time period be solved for a value n. This can be done for all cases except n \( \log n \) and n!, which will be evaluated numerically. For example,

\[ F(n) = \log n \text{ microsec} = 1 \text{ sec} \Rightarrow 2^{\log n} = 2^{1E6} \Rightarrow n = 2^{1E6} \]

<table>
<thead>
<tr>
<th></th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
<th>1 day</th>
<th>1 month</th>
<th>1 year</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>( 2^{1E6} )</td>
<td>( 2^{6E7} )</td>
<td>( 2^{1.6E9} )</td>
<td>( 2^{8.64E10} )</td>
<td>( 2^{2.59E12} )</td>
<td>( 2^{3.15E13} )</td>
<td>( 2^{3.15E15} )</td>
</tr>
<tr>
<td>( n^{0.5} )</td>
<td>1.00E12</td>
<td>3.60E15</td>
<td>1.30E19</td>
<td>7.46E21</td>
<td>6.72E24</td>
<td>9.95E26</td>
<td>9.95E30</td>
</tr>
<tr>
<td>( n )</td>
<td>1E6</td>
<td>6E7</td>
<td>3.6E9</td>
<td>8.64E10</td>
<td>2.59E12</td>
<td>3.15E13</td>
<td>3.15E15</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>62746</td>
<td>2.8E6</td>
<td>1.33E8</td>
<td>2.75E9</td>
<td>7.18E10</td>
<td>7.97E11</td>
<td>6.86E13</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>1.00E03</td>
<td>7.75E03</td>
<td>6.00E04</td>
<td>2.94E05</td>
<td>1.61E06</td>
<td>5.62E06</td>
<td>5.62E07</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>100</td>
<td>391</td>
<td>1532</td>
<td>4420</td>
<td>13736</td>
<td>31593</td>
<td>146645</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>19</td>
<td>25</td>
<td>31</td>
<td>36</td>
<td>41</td>
<td>44</td>
<td>51</td>
</tr>
<tr>
<td>( n! )</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

In solving for problems like n! and n \( \log n \), numerically solving in Excel is an acceptable approach. For example, let cell A1 equal n and cell A2 equal “=FACT(A1)” Next, we recognize 1 second equals \( 1 \times 10^6 \) microseconds. So, increase cell A1 until cell A2 equals \( 1 \times 10^6 \). Rather than manually changing cell A1, you could use Excel’s Goal Seek function.

2. Exercise 2.2-2)

\[
\begin{align*}
\text{n} &\leftarrow \text{length}[A] \\
\text{for} \ i &\leftarrow 1 \text{ to } n-1 \\
\text{do} & \quad \text{smallest} \leftarrow i \\
\text{for} \ i &\leftarrow j + 1 \text{ to } n \\
\text{do if} & \quad A[i] < A[\text{smallest}] \\
\text{then} & \quad \text{smallest} \leftarrow i \\
\text{exchange} & \quad A[j] \leftrightarrow A[\text{smallest}] \\
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n ) &amp; \text{c}_1 &amp; 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{for} \ i &amp; \text{c}_2 &amp; n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{do} \ \text{smallest} &amp; \text{c}_3 &amp; n-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{for} \ i &amp; \text{c}_4 &amp; n</td>
<td>\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1</td>
<td></td>
</tr>
<tr>
<td>\text{do if} \ A[i] &amp; \text{c}<em>5 &amp; \sum</em>{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{then} \ \text{smallest} &amp; \text{c}<em>6 &amp; \sum</em>{j=2}^{n} (t_j - 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{exchange} &amp; \text{c}_7 &amp; n-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The algorithm maintains the loop invariant that at the start of each iteration of the outer for loop, the subarray A[1, ..., j-1] consists of the j-1 smallest elements in the array A[1, ..., n], and this subarray is the sorted order. After the first n-1 elements, the subarray A[1, ..., n-1] contains the smallest n-1 elements, sorted, and therefore element A[n] must be the largest element.

Best Case) Best-case will occur when the array is already sorted, thus \( c_6 \) will never be executed. Hence the running time is:

\[
T(n) = c_1(1) + c_2(n) + c_3(n-1) + c_4(n^2+n-2)/2 + c_5(n^2-n)/2 + c_7(n-1)
\]
\[
T(n) = (c_4/2 + c_5/2)n^2 + (c_2 + c_3 + c_4/2 - c_5/2 + c_7)n + (c_1 - c_3 - c_4 - c_7)
\]
\[
T(n) = a n^2 + b n + c
\]
\[
\mathcal{T}(n) = \Theta(n^2)
\]

Worse Case) Worst-case will occur when the array is unsorted in reverse order (i.e. \(<5, 4, 3, 2, 1>\)), thus \( c_6 \) will occur \( n(n-1)/2 \) time. Hence the running time is:

\[
T(n) = c_1(1) + c_2(n) + c_3(n-1) + c_4(n^2+n-2)/2 + c_5(n^2-n)/2 + c_6(n^2-n)/2 + c_7(n-1)
\]
\[
T(n) = (c_4/2 + c_5/2 + c_6/2)n^2 + (c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7)n + (c_1 - c_3 - c_4 - c_7)
\]
\[
T(n) = a n^2 + b n + c
\]
\[
\mathcal{T}(n) = \Theta(n^2)
\]

The running time for the algorithm is \( \Theta(n^2) \) for all cases.

3. Exercise 2.2-4)
Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

4. Exercise 2.3-1)
5. Exercise 2.3-3)
Prove by induction that

\[ T(n) = \begin{cases} 
2 & \text{if } n = 2 \\
2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 
\end{cases} \]

is

\[ T(n) = n \lg n \]

To prove by induction, we must 1) show it is true for some base case, 2) hypothesize for some value of k, and 3) show that it holds for k+1.

Base Case:
when k=1 => n = 2
T(n) = n \lg n = 2 \lg 2 = 2 \times 1 = 2.

Hypothesis Step:
\[ T(n) = (n) \lg(n) \]
\[ T(2^k) = (2^k) \lg(2^k) \]

Inductive Step k+1:
\[ T(2^{k+1}) = 2T(2^{k+1}/2) + 2^{k+1} = 2T(2^k) + 2^{k+1} \]
\[ T(2^{k+1}) = 2^{k+1} \left( \lg 2^k + 1 \right) \]
\[ T(2^{k+1}) = 2^{k+1} \lg 2^{k+1} \]
Q.E.D.

6. Exercise 2.3-5)
The binary search procedures takes a sorted array A, a value v, and a range \([\text{low} \ldots \text{high}]\) of the array A, in which to search for the value v.

### Iterative-Binary-Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
</table>
| **ITERATIVE-BINARY-SEARCH** | A, v, low, high 
 | while low ≤ high 
 | do mid = \((\text{low} + \text{high}) / 2\) 
 | If v = A[mid] 
 | then return mid 
 | if v > A[mid] 
 | then low ← mid + 1 
 | else high ← mid - 1 
 | return NIL |

### Recursive-Binary-Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
</table>
| **RECURSIVE-BINARY-SEARCH** | A, v, low, high 
 | if low > high 
 | then return NIL 
 | mid ← \((\text{low} + \text{high}) / 2\) 
 | if v = A[mid] 
 | then return mid 
 | if v > A[mid] 
 | then return RECURSIVE-BINARY-SEARCH(A, v, mid+1, high) 
 | else return RECURSIVE-BINARY-SEARCH(A, v, low, mid-1) |

Based on the comparison of v to the middle element in the search range, the search continues with the range halved.
The depth of the tree is computed by noting that \( T(1) \) occurs when \( T(n/2^2) = T(1) \), or \( n/2^j = 1 \). This means \( n = 2^j \), or \( j = \log n \).

**7. Exercise 3.1-4**

(a) Is \( 2^{n+1} = O(2^n) \)?

\[
\text{f(n)} = O(g(n))
\]

\[
0 \leq f(n) \leq c g(n) \text{ where } c > 0 \text{ and } n > n_0
\]

Hence,

\[
0 \leq 2^{n+1} \leq c 2^n \text{ where } n > n_0
\]

\[
0 \leq 2^{n+1} / 2^n \leq c \text{ where } n > n_0
\]

\[
0 \leq 2 \leq c \text{ where } n > 0
\]

\[
c \geq 2 \text{ and } n > 0
\]

Therefore,

\[
2^{n+1} = O(2^n) \text{ Q.E.D.}
\]

(b) Is \( 2^{2n} = O(2^n) \)?

\[
\text{f(n)} = O(g(n))
\]

\[
0 \leq f(n) \leq c g(n) \text{ where } n > n_0
\]
Hence,  
\[ 0 \leq 2^{2n} \leq c \cdot 2^n \text{ where } n > n_0 \]  
\[ 0 \leq 2^{2n} / 2^n \leq c \text{ where } n > n_0 \]  
\[ 0 \leq 2^n \leq c \text{ where } n > n_0 \]  
\[ 0 \leq 2^{n_0} \leq c \]

Now,  
\[ 0 \leq 2^{2n} \leq 2^{n_0} \cdot 2^n \text{ where } n > n_0 \]  
\[ 0 \leq 2^{2n} \leq 2^{n+n_0} \text{ where } n > n_0 \]  
\[ 2n \leq n + n_0 \text{ for } n > n_0 \]  
\[ n \leq n_0 \text{ for } n > n_0 \]

Is a contradiction so,  
\[ 2^{2n} \neq O(2^n) \text{ Q.E.D.} \]

8. **Show that** \( n(n-1)/2 \) **is** \( O(n^2) \)

\[ f(n) = O(g(n)) \]

\[ 0 \leq f(n) \leq c \cdot g(n) \text{ where } c > 0 \text{ and } n > n_0 \]

Hence,  
\[ 0 \leq n(n-1)/2 \leq c \cdot n^2 \text{ where } c > 0 \text{ and } n > n_0 \]  
\[ 0 \leq 1/2-1/2n \leq c \text{ where } c > 0 \text{ and } n > n_0 \]  
\[ 0 \leq 1/2-1/4 \leq c \text{ where } n \geq 2 \]

\[ c = 1/4 \]

Hence,  
\[ 0 \leq n(n-1)/2 \leq n^2/4 \text{ where } n \geq 2 \]

Therefore,  
\[ n(n-1)/2 = O(n^2) \]

9. Using "big-O" notation, give the worst case running times of the following procedures as a function of \( n \).

```pascal
procedure mystery (n: integer);
var
    i,j,k: integer;
begin
    for i := 1 to n-1 do
        for j := i + 1 to n do
```
for \( k := 1 \) to \( j \) do

... a statement requiring \( O(1) \) time

end

<table>
<thead>
<tr>
<th>Line</th>
<th>Pseudo code</th>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for ( i := 1 ) to ( n-1 ) do</td>
<td>( c_1 )</td>
<td>( n )</td>
</tr>
<tr>
<td>2</td>
<td>for ( j := i + 1 ) to ( n ) do</td>
<td>( c_2 )</td>
<td>( \sum_{j=2}^{i} j = \frac{n^2 + n - 2}{2} )</td>
</tr>
<tr>
<td>3</td>
<td>for ( k := 1 ) to ( j ) do</td>
<td>( c_3 )</td>
<td>( \sum_{k=1}^{n} \sum_{j=k}^{n+1} j = \frac{n(2n + 5)(n - 1)}{6} )</td>
</tr>
<tr>
<td>4</td>
<td>... a statement requiring ( O(1) ) time</td>
<td>( c_4 )</td>
<td>( \sum_{k=2}^{n} \sum_{j=k}^{n} j = \frac{n(n + 1)(n - 1)}{3} )</td>
</tr>
</tbody>
</table>

\[ T(n) = c_1 n + c_2 (n^2 + n - 2)/2 + c_3 (n)(2n+5)(n-1)/6 + c_4(n)(n+1)(n-1)/3 \]

\[ T(n) = (c_3 + c_4)n^3/3 + (c_2 + c_3)n^2/2 + (n_1 + c_2/2 - 5c_3/6 - c_4/3)n - c_2 \]

\[ T(n) \propto n^3 + bn^2 + cn + d \]

\[ T(n) = O(n^3) \]