1. Each of these code fragments takes $O(n \log n)$ time, but not necessarily $\Theta(n \log n)$. Give the asymptotic complexity of each in terms of $n$, using $\Theta$ in each case.

(a) for(int i = 1; i < n; i++)
for(int j = 1; j < i; j = 2*j);
cout << "Hello" << endl;
\[ \int_{x=1}^{n} (\ln x) dx = x \ln x - x \bigg|_{x=1}^{n} = \Theta(n \log n) \]

(b) for(int i = 1; i < n; i++)
for(int j = i; j < n; j = 2*j);
cout << "Hello" << endl;
\[ \int_{x=1}^{n} (\ln n - \ln x) dx = x \ln x - x \ln x + x \bigg|_{x=1}^{n} = \Theta(n) \]

(c) for(int i = 1; i < n; i=2*i)
for(int j = 1; j < i; j++);
cout << "Hello" << endl;
Let k = log$_2$i; then $2^k = i$.
for(int k = 0; i < log$_2$ n; k++)
for(int j = 2^k; j < n; j++);
cout << "Hello" << endl;
Let x be the continuous analog of k and y the continuous analog of j.
\[ \int_{x=0}^{\log_2 n} \int_{y=1}^{2^x} dy dx = \int_{x=0}^{\log_2 n} \int_{y=0}^{2^x - 1} dy dx = \frac{2^x - x}{\ln 2} \bigg|_{x=0}^{\log_2 n} = \frac{2^{\log_2 n} - 1}{\ln 2} = \frac{n - 1}{\ln 2} = \Theta(n) \]

(d) for(int i = 1; i < n; i=2*i)
for(int j = i; j < n; j++);
cout << "Hello" << endl; cd /home/larmore/Dropbox/Courses/CS477/S21
Let k = log$_2$i; then $2^k = i$.
for(int k = 0; i < log$_2$ n; k++)
for(int j = 2^k; j < n; j++);
cout << "Hello" << endl;
Let x be the continuous analog of k and y the continuous analog of j.
\[ \int_{x=0}^{\log_2 n} \int_{y=2^x}^{n} dy dx = \int_{x=0}^{\log_2 n} (n - 2^x) dx = \left( nx - \frac{2^x}{\ln 2} \right) \bigg|_{x=0}^{\log_2 n} = n \log_2 n - \frac{\log_2 n - 1}{\ln 2} = n \log_2 n - \frac{n - 1}{\ln 2} = \Theta(n \log n) \]
(e) for(int i = n; i > 1; i=i/2)
   for(int j = i; j > 1; j--);
   cout << "Hello" << endl;
Same as (c). $\Theta(n)$

(f) for(int i = n; i > 1; i=i/2)
   for(int j = n; j > i; j--);
   cout << "Hello" << endl;
Same as (d). $\Theta(n \log n)$

2. These problems are harder than the ones above. Given the asymptotic complexity of each fragment in terms of $n$, using $\Theta$.

(g) for(int i = 1; i < n; i=2*i)
   for(int j = 1; j < i; j=2*j);
   cout << "Hello" << endl;
Hint: Use substitution. Let $m = \log n$, $k = \log i$, $l = \log j$.

for(int k = 0; k < m; k++)
   for(int l = 0; i < k; l++)
   cout << "Hello" << endl;
$\Theta(m^2) = \Theta(\log^2 n)$

(h) for(int i = 2; i < n; i=i*i)
   cout << "Hello" << endl;
Hint: Use substitution. Let $m = \log n$, $k = \log i$.
Use the fact that $\log(x^y) = y \log x$

for(int k = 1; k < m; k=2*k)
   cout << "Hello" << endl;
$\Theta(\log m) = \Theta(\log \log n)$

(i) for(int i = 2; i < n; i=i*i)
   for(int j = 1; j < i; j = 2*j)
   cout << "Hello" << endl;
Hint: Use substitution. Let $m = \log n$, $k = \log i$, $l = \log j$.

for(int k = 1; k < m; k=2*k)
   for(int l = 0; l < k; l++)
$\Theta(m) = \Theta(\log n)$
(j) for(int i = n; i > 1; i = log i)
    cout << "Hello" << endl;
Use the substitution \( m = \log^* n, k = \log^* i \)
for(int k = m; k > 0; k--)

**Added on February 5:**

The recursive definition of \( \log^* x \) for any real number \( x \) is:
- \( \log^* x = 0 \) if \( x \leq 1 \)
- \( \log^* x = 1 + \log^* (\log x) \) if \( x > 1 \)

Let \( i \) be the “old” value of \( i \) in the code, and \( \tilde{i} \) the “new” value of \( i \), namely \( \log i \). Let \( k \) be the old value of \( k \) and \( \tilde{k} \) the new value of \( k \). Thus

\[
    m = \log^* n \\
    \tilde{i} = \log i \\
    k = \log^* i \\
    \tilde{k} = \log^* \tilde{i}
\]

From the definition of \( \log^* \) we have:

\[
    k = \log^* i + \log^* (\log i) = 1 + \log^* \tilde{i} = 1 + \tilde{k}. 
\]

Thus \( \tilde{k} = k - 1 \), and the last parameter of the for statement is \( \tilde{k} - 1 \).

**End Added Text**

The solution is \( \Theta(m) = \Theta(\log^* n) \) where \( \log^* \) is the *iterated logarithm*. For any positive real number \( x \), \( \log^* x \) is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1.

We use the base 2 logarithm. In that case, the iterated algorithm is sometimes written as \( \lg^* \).

i. What is \( \log^* 65536 \)? Answer: 4.

ii. What is \( \log^* 65537 \)? Answer: 5.

iii. Let \( N \) be the number of baryons in the visible universe. (Neutrons and protons are baryons.)
What is \( \log^* N \)? Answer: 5.

iv. It has been seriously conjectured that the radius of the entire universe is \( 10^{100} \) times the radius of the visible universe! If that is true, what is \( \log^* \) of the number of baryons in the universe?
Answer 5.

\( \log^* \) grows very slowly. However, it is not the slowest growing unbounded function that regularly arises in complexity theory. That honor goes to the inverse Ackermann function.

(k) for(int i = 2; i < n; i = i*i)
    for(int j = 0; j < i; j++)
        cout << "Hello" << endl;
In my opinion, this is the hardest problem in this assignment. The time complexity of the code is \( O \) of one function of \( n \) and \( \Omega \) of a different function of \( n \), but is not \( \Theta \) of any of the “usual” functions of \( n \). Give both the \( O \) and the \( \Omega \) answers, both of which are “usual” functions.\(^1\)

Answer: The time complexity both \( O(n) \) and \( \Omega(\sqrt{n}) \).

\(^1\)By *usual functions* I mean the functions we have discussed so far in class, which include polynomials, logarithms, iterated logarithms, powers of logarithms, roots, and even the iterated logarithm \( \log^* \).
The outer loop iterates $O(\log \log n)$ times. For each value of $i$ used during the outer loop, the inner loop iterates $I$ times. Those values of $i$ are numbers of the form $2^{2^k}$ for integers $k \geq 0$. That is,

- $2^{2^0} = 2$,
- $2^{2^1} = 2^2 = 4$,
- $2^{2^2} = 4^2 = 16$,
- $2^{2^3} = 16^2 = 256$,
- $2^{2^4} = 256^2 = 65536$,
- $2^{2^5} = 65536^2 = 4294967296$.

Since $i$ increases rapidly, the time complexity of the code is dominated by the largest value of $i$ generated in the outer loop, which is the largest value of $2^{2^k}$ less than $n$. Let’s call that value $I$. For example, if $4 < n \leq 16$, $I = 4$; if $16 < n \leq 256$, $I = 16$; and if $256 < n \leq 65536$, $I = 256$; and so forth. Note that $I < n \leq I^2$, which implies that $\sqrt{n} \leq I < n$. The time complexity of the code is $\Theta(I)$, and we obtain our result.
3. Solve each of the following recurrences, giving the answer as \( \Theta \) of a function of \( n \).

(l) \( F(n) = F(n/2) + n^2 \)
Master theorem: \( A = 1, B = 2, C = 2 \): Note that \( A < B^C \).
Thus \( F(n) = \Theta(n^C) = \Theta(n^2) \)

(m) \( F(n) = F(n/3) + 1 \)
Master theorem: \( A = 1, B = 3, C = 0 \): Note that \( A = B^C \).
Thus \( F(n) = \Theta(n^C \log n) = \Theta(\log n) \)

(n) \( F(n) = 16F(n/4) + n^2 \)
Master theorem: \( A = 16, B = 4, C = 2 \). Note that \( A = B^C \).
Thus \( F(n) = \Theta(n^C \log n) = \Theta(n^2 \log n) \)

(o) \( F(n) = F(n-1) + n^5 \)
Anti-derivative method: \( \frac{F(n) - F(n-1)}{1} = n^5 \)
\( F'(n) = \Theta(n^5) \)
\( F(n) = \Theta(n^6) \)

(p) \( F(n) = F(n - \log n) + \log n \)
Anti-derivative method: \( \frac{F(n) - F(n - \log n)}{\log n} = \frac{\log n}{\log n} \)
\( F'(n) = \Theta(1) \)
\( F(n) = \Theta(n) \)

(q) \( F(n) = 16F(n/4) + n \)
Master theorem: \( A = 16, B = 4, C = 1 \). Note that \( A > B^C \), and that \( \log_B A = 2 \).
Thus \( F(n) = \Theta(n^{\log_B A}) = \Theta(n^2) \).