Proof: Applying the Euclidean algorithm to calculate \( \gcd(a, b) \), with \( a > b \), \( (a = r_0, b = r_1) \), we obtain the sequence of equations:

\[
\begin{align*}
  r_0 &= r_1q_1 + r_2, & 0 \leq r_2 < r_1 \\
  r_1 &= r_2q_2 + r_3, & 0 \leq r_3 < r_2 \\
  \vdots & \quad \vdots \\
  r_{n-1} &= r_nq_n \\
  r_n = g = \gcd(a, b)
\end{align*}
\]

We use \( n \) divisions to find \( r_n = \gcd(a, b) \).

We must also have \( q_n > 2 \) since \( r_n < r_{n-1} \).

We now find:

\[
\begin{align*}
  r_n &\geq 1 = f_2 \\
  r_{n-1} &\geq 2r_n \geq 2f_2 = f_3 \\
  r_{n-2} &= r_{n-1}q_{n-1} + r_n \geq r_{n-1} + r_n \geq f_3 + f_2 = f_4 \\
  \vdots & \quad \vdots \\
  r_2 &\geq r_3 + r_4 \geq f_{n-1} + f_{n-2} = f_{n+1} \\
  b = r_1 &\geq r_2 + r_3 \geq f_n + f_{n-1} = f_{n+1}
\end{align*}
\]

So, we have \( b \geq f_{n+1} > 2^{n-1} \)

\[
\log_{10} x \geq 0.208 > 1/5
\]

\[
\log_{10} b > (n-1) \log_{10} x > (n-1)/5
\]

\[\Rightarrow \quad n < 5 \cdot \log_{10} b + 1\]

If \( b \) has \( k \) digits, then \( b < 10^k \), so that \( n < 5k + 1 \), or \( n \leq 5k \).

Since \( k \) is an integer.
4. Recursive sets

a. Sets like functions can be built up recursively
Example: Basis step: 3 ∈ S
Recursive step: If x ∈ S and y ∈ S, then x + y ∈ S
0 step: S contains 3
1 step: S contains 3, 6
2 step: S contains 3, 6, 9, 12
We eventually get all positive multiples of 3

b. The set $\Sigma^*$ of strings over an alphabet $\Sigma$ can be defined recursively as
Basis step: $\lambda \in \Sigma^*$ ($\lambda$ = empty string)
Recursive step: $w \in \Sigma^*, \exists x \in \Sigma \Rightarrow wx \in \Sigma^*$
Example: $\Sigma = \{0, 1\}$
0 step: $\Sigma^*$ contains 0, 1, $\lambda$
1 step: $\Sigma^*$ contains $\lambda, 0, 1, 00, 01, 10, 11$

To define the length of a string, we may use
Basis step: $\ell(\lambda) = 0$
Recursive step: $\ell(wx) = \ell(w) + 1$
where $w \in \Sigma^*$ and $x \in \Sigma$

This plays an important role in formal logic

C. Well-formed formulae for propositions
We may start with the following objects
Constants: $\top, \bot$
Propositional variable: $p$
Operators: $\neg, \land, \lor, \rightarrow, \leftrightarrow$
Basis step: \( \top, \bot, p \)

Recursive step: If \( E \) and \( F \) are well-formed formulae, then \( \neg E \), \( E \land F \), \( E \lor F \), \( E \rightarrow F \), and \( E \leftrightarrow F \) are all well-formed formulae

0 Step: \( \top, \bot, p \)

1 step: \( \neg \top \), \( \top \lor p \), \( \top \rightarrow \bot \)
are all examples. There are many more.

2 step: \( \neg p \rightarrow (\top \leftrightarrow p) \), \( p \lor (p \rightarrow \bot) \), are examples that appear at the next step

Informal

d. Rooted trees

Definition: A tree is a graph consisting of vertices and edges. Will be defined by example here.

Trees play a key role in constructing algorithms of many sorts in computer science.

These are trees that contain a distinguished vertex \( \bullet \) called a root and edges connecting all the vertices.

Basis step: A single vertex \( v \) is a rooted tree.

Recursive step: Suppose that \( T_1, T_2, \ldots, T_n \), \( n \in \mathbb{Z}^+ \), are rooted trees with roots \( r_1, r_2, \ldots, r_n \).
Then, the graph that is formed by connecting a root \( r \) that is not in any of the trees to \( r_1, r_2, \ldots, r_n \) is also a rooted tree.
e. Extended binary trees

Basis step: The empty set is an extended binary tree.

Recursive step: If $T_1$ and $T_2$ are extended binary trees, there is an extended binary tree consisting of a root $r$ and edges to the roots of $T_1$ and $T_2$ when they are non-empty.

Basis step: $\emptyset$

1 Step: $\emptyset$

(T1 and T2 are both empty, so there are no edges.)

2 Step:
3. Step: This has many objects. Some of them are

```
/ \   \   \   \   \ ...
```

f. Full binary trees

**Basis step:** A single vertex \( r \) is a full binary tree (and is its root).

**Recursive step:** If \( T_1 \) and \( T_2 \) are full binary trees, then there is a full binary tree \( T \), consisting of a root \( r \) with edges connecting the roots of both \( T_1 \) and \( T_2 \).

0 step

```
```

1 step

```
```

2 step

```
```

5. Structural induction

a. This is an approach used to prove results about recursively defined structures.

**Basis step:** Show that the result holds for all elements in the basis step.

**Recursive step:** Show that if the result holds for all elements of the \( n \)-th step, then
it holds for all elements of the
n+1-th step. \( P(n) \rightarrow P(n+1) \)

b. Example: Every well-formed expression as defined on pages 427-8 has an equal number of right and left parentheses. Proof:

Basis step: \( \top, \bot, \text{ and } p \) have zero right and left parentheses.

Recursive step: Assume \( p \) and \( q \) are well-formed formulae, each containing \( L_p \) and \( L_q \) left parentheses and \( R_p \) and \( R_q \) right parentheses. We assume \( L_p = R_p \) and \( L_q = R_q \).

All possible combinations of \( p \) and \( q \) are \( \neg p, \neg q, (p \lor q), (q \lor p) \), \( (p \land q), (p \lor q), (p \rightarrow q), (q \rightarrow p), (p \leftrightarrow q), (q \leftrightarrow p) \).

In all cases, \( l = L_p + L_q + 1 = R = R_p + R_q + 1 \).

c. Example: If \( x \) and \( y \) are strings in \( \Sigma^* \), then \( l(xy) = l(x) + l(y) \). Proof:

Basis step: \( l(x\lambda) = l(x) \) since \( x\lambda = x \)

We also have \( l(\lambda) = 0 \), so that \( l(x\lambda) = l(x) + l(\lambda) \).

Recursive step: Assume \( P(y) \) is true. We will show \( P(ya) \) is true for all \( a \in \Sigma \) (the alphabet).

By definition: \( l(xya) = l(xy) + 1 \), \( l(ya) = l(y) + 1 \).
By inductive hypothesis, \( L(xy) = L(x) + L(y) \).
From algebra, we conclude
\[
L(xya) = L(x) + L(ya)
\]

\[d.\] Definition of the height \( h(T) \) of a full binary tree

Basis step: The height \( h(T) \) of a tree consisting of the root \( R \) is zero.

Recursive step: If \( T_1 \) and \( T_2 \) are full binary trees, then
\[
h(T_1, T_2) = 1 + \max(h(T_1), h(T_2))
\]

Why do we need to define this when we can just count from the picture.

Definition: of the number of vertices \( n(T) \) in a full binary tree \( T \)

Basis step: \( n(T) = 1 \) when the tree only consists of the root \( R \).

Recursive step: If \( T_1 \) and \( T_2 \) are full binary trees, then
\[
h(T) = 1 + h(T_1) + h(T_2)
\]

\[e.\] Theorem: If \( T \) is a full binary tree, \( n(T) \leq 2^{h(T)+1} - 1 \). Proof:

Basis step: When \( T \) consists of the root \( R \),
\[ n(T) = 1 \quad \text{and} \quad 2^{h(T)+1} - 1 = 2 - 1 = 1 \]

Inductive step: We assume \( n(T_1) \leq 2^{h(T_1)+1} - 1 \) and \( n(T_2) \leq 2^{h(T_2)+1} - 1 \), where \( T_1 \) and \( T_2 \) are full binary trees,
\[
h(T) = 1 + n(T_1) + n(T_2) \quad [\text{by definition}]
\leq 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1) \quad [\text{by hypothesis}]
\leq 2 \max(2^{h(T_1)}, 2^{h(T_2)}+1) - 1
\]
[Since \( a + b \leq 2 \max(a, b) \)]
\[
= 2 \cdot 2^{\max\{h(T_1), h(T_2)\} + 1} - 1 \quad \text{[algebra]}
\]

\[
= 2 \cdot 2^h(T) - 1 \quad \text{[definition]}
\]

\[
= 2^{h(T) + 1} - 1 \quad \text{[algebra]}
\]

6. Generalized induction

a. We can extend the concepts of well-ordering and induction beyond integers. One example is to specify \((x_1, y_1)\) is less than \((x_2, y_2)\) if either \(x_1 < x_2\) or \(x_1 = x_2\) and \(y_1 < y_2\).

Note: This corresponds to usual condition for two-digit integers to be less than another if \(x\) refers to the tens-digit and \(y\) refers to the ones-digit.
D. Recursive Algorithms and Algorithm Correctness

1. An algorithm is called recursive (particularly when acting on elements of \( \mathbb{N} \)) when it solves a problem by reducing it to an instance of the same problem with a smaller input and the possible inputs have a well-defined lower bound.

2. Example: We may calculate \( a^n \) recursively, using \( a^{n+1} = a \cdot a^n \) and \( a^0 = 1 \).

   Algorithm (in pseudo-code)
   
   ```
   procedure power (a: nonzero real number, n: non-negative integer)
   
   if n > 0 then power (a, n) := 1
   else power (a, n) := a * power(a, n-1)
   ```

3. Example: Recursive modular exponential

   Goal: Calculate \( b^n \mod m \), where \( b, n, m \in \mathbb{Z} \) with \( m \geq 2 \), \( n \geq 0 \) and \( 1 \leq b < m \).

   Algorithm (in pseudo-code)
   
   ```
   procedure mpower (b, n, m: integers with \( m \geq 2 \), \( n \geq 0 \), \( 1 \leq b < m \))
   
   if n > 0 then
     mpower (b, n, m) = 1
   else if n is even then
     mpower (b, n, m) = mpower (b, n/2, m)² mod m
   else
     mpower (b, n, m) = (mpower (b, (n-1)/2, m)² mod m * b mod m) mod m
   ```

   Note that the routine is calling itself recursively.

\( \{ \text{mpower (b, n, m) } = b^n \mod m \} \)
Proof of algorithm correctness using strong induction.

Basis step: When $n=0$, \( \text{mpower}(b, n, m) = 1 \mod m = 1 \) for any integers \( b \) and \( m \) that satisfy \( m \geq 2 \), \( 1 \leq b < m \).

Inductive step: Our hypothesis is that \( \text{mpower}(b, j, m) = b^j \mod m \), for all \( j \), \( 0 \leq j < n \).

(i) \( k \) is even:

\[
\text{mpower}(b, k, m) = \text{mpower}(b, k/2, m)^2 \mod m
\]

inductive hypothesis

\[
= (b^{k/2} \mod m)^2 \mod m
\]

\[
= b^k \mod m
\]

Since, in general, writing

\[
p = q \cdot m + r, \quad (\text{division algorithm})
\]

where \( 0 \leq r < m \), \( p \in \mathbb{N} \), \( q \in \mathbb{Z} \),

\[
(p \mod m)^2 = (p - q \cdot m)^2 = r^2
\]

\[
(p \mod m)^2 \mod m
\]

\[
= (p^2 - 2q \cdot m + q^2 \cdot m^2) \mod m
\]

\[
= p^2 \mod m
\]

(ii) \( k \) is odd:

\[
(\text{mpower}(b, k/2, m)^2 \mod m \cdot b \mod m) \mod m
\]

\[
= (b^{k-1} \mod m \cdot b \mod m) \mod m
\]

Write \( k = q \cdot m + r \)

\[
b^{k-1} \mod m = (b^{k/2})^q \mod m = q^{k-1} \mod m
\]

\[
b \mod m = q,
\]

Writing \( q \cdot k \mod m = 5m + t, \)

\[
q \cdot q^{k-1} \mod m = q \cdot t = q (q^{k-1} - 5m)
\]

So, \( (q \cdot q^{k-1} \mod m) \mod m = q^k \mod m \)

\[
= b^k \mod m
\]

Hence, \( (\text{mpower}(b, k, m)^2 \mod m \cdot b \mod m) \mod m \)

\[
= b^k \mod m
\]
4. **Linear search**

To search for $x$ in the sequence $a_1, a_2, \ldots, a_n$, at the $i$-th step of the algorithm, we compare $x$ and $a_i$. If $a_i = x$, then $i$ is the location of $x$. Otherwise, we search in the smaller sequence $a_1, a_2, \ldots, a_{i-1}$.

**Algorithm**

```plaintext
procedure search (i, j, x)
    if $a_i = x$ then
        location := i
    else if $i = j$ then
        location := 0  \{ failed search \}
    else
        search (i+1, j, x)  \{ call the algorithm again \}
```

5. **Recursion and iteration**

a. Recursive algorithms and definitions advance from larger to smaller integers. Iterative algorithms and definitions advance from smaller to larger integers.

b. Iterative routines are typically not self-referential in the same way as recursive routines and are thus often more efficient.

**Example:** A recursive Fibonacci algorithm

```plaintext
procedure fibonacci (n: non-negative integer)
    if $n = 0$ then fibonacci (0) := 0
    else if $n = 1$ then fibonacci (1) := 1
    else fibonacci (n) := fibonacci (n-1) + fibonacci (n-2)
```
The way this works is we write
\[ f_n = f_{n-1} + f_{n-2} \]
\[ = (f_{n-2} + f_{n-3}) + (f_{n-3} + f_{n-4}) \]
\[ = \ldots \]

(1) So, we are doubling the number of evaluations on each step until we reach \( f_0 \) and \( f_1 \), each of which we evaluate many times, as shown to the left with \( f_4 \).

This algorithm requires \( 2^n - 1 \) additions to find \( f_n \).

(2) An iterative Fibonacci algorithm procedure iterative_fibonacci (n: non-negative integer)

\[ \text{if } n = 0 \text{ then } y := 0 \text{ else} \]
\[ \text{begin} \]
\[ \quad x := 0 \]
\[ \quad y := 1 \]
\[ \quad \text{for } i := 1 \text{ to } n-1 \text{ begin} \]
\[ \quad \text{begin} \]
\[ \quad \quad z := x + y \]
\[ \quad \quad x := y \]
\[ \quad \quad y := z \]
\[ \quad \end{\text{begin}} \]
\[ \end{\text{for}} \]
\[ \end{\text{begin}} \]
\[ \end{\text{if}} \]

\{ y is the n-th Fibonacci number \}

This approach only requires \( n-1 \) additions to find \( f_n \) when \( n > 1 \).
6. Merge sort

In a merge sort, we iteratively split a list into two lists of equal size or sizes that differ by one.

Once we have broken down the list to individual elements, we then build it back up again.

We give a recursive algorithm that requires a separate merging routine, called merge.

a. Recursive merge sort

```
 procedure mergesort (L = a₁, ..., aₙ)
  if n > 1 then
    m := L ⌊n/2⌋
    L₁ := a₁, a₂, ..., aₘ
    L₂ := aₘ₊₁, aₘ₊₂, ..., aₙ
    L := merge (mergesort(L₁), mergesort(L₂))
```
b. merging algorithm

Suppose we want to merge

\[ L_1 = \{ a_{11}, a_{12}, \ldots, a_{1m_1} \} \quad \text{and} \quad L_2 = \{ a_{21}, a_{22}, \ldots, a_{2m_2} \} \]

into a list \( L \), where \( L_1 \) and \( L_2 \) are already sorted; we may proceed iteratively, starting with \( L = \lambda \).

Generally, we write \( L = \{ a_0, a_1, \ldots, a_{m_0} \} \).

So, at \( j = 0 \) (initial iteration), we have \( m_0 = 0 \).

On the \( j \)-th iteration:

If \( L_1 = \lambda \), \( L \rightarrow L \circ L_2 \), \( m_0 \rightarrow m_0 + m_2 \)

\[ [a_0, m_0 + 1 = a_{21}, a_{0, m_0 + 2} = a_{22}, \ldots, a_{0, m_0 + m_2} = a_{2, m_2}] \]. And we end.

If \( L_2 = \lambda \), we proceed similarly.

Otherwise,

If \( a_{11} < a_{21} \)

\[ a_0, m_0 + 1 = a_{11} \]

\[ a_{11} \rightarrow a_{12}, a_{12} \rightarrow a_{11}, \ldots, a_{1, m_1 - 1} \rightarrow a_{1, m_1} \]

\[ m_0 \rightarrow m_0 + 1 \] and \( m_1 \rightarrow m_1 - 1 \)

If \( a_{21} < a_{11} \)

\[ a_0, m_0 + 1 = a_{21} \]

and the rest is similar.

Example:

\[ L_1 = \{ 2, 3, 5, 6, 7 \}, \quad L_2 = \{ 1, 4, 5 \} \]

\( j = 0 \):

\[ L = \lambda, \quad a_{11} = 1 < a_{11} = 2 \]

\[ L = \{ 1 \}, \quad L_1 = \{ 2, 3, 5, 6, 7 \}, \quad L_2 = \{ 4, 5 \} \]

\( j = 1 \):

\[ a_{11} = 2 < a_{21} = 4 \]

\[ L = \{ 1, 2 \}, \quad L_1 = \{ 3, 5, 6, 7 \}, \quad L_2 = \{ 4, 5 \} \]

\( j = 2 \):

\[ a_{11} = 3 < a_{21} = 4 \]

\[ L = \{ 1, 2, 3 \}, \quad L_1 = \{ 5, 6, 7 \}, \quad L_2 = \{ 4, 5 \} \]

\( j = 3 \):

\[ a_{21} = 4 < a_{11} = 5 \]

\[ L = \{ 1, 2, 3, 4, 5 \}, \quad L_1 = \{ 5, 6, 7 \}, \quad L_2 = \lambda \]

\( j = 4 \):

\[ L = \{ 1, 2, 3, 4, 5, 6, 7 \}, \quad \text{end} \]
7. **Big-O Notation**

We will want to estimate the efficiency of the merge sort algorithm by comparing it to a logarithmic function. This sort of comparison occurs often and has led to the introduction of big-O notation.

**(a) Definition:** Let $f$ and $g$ be functions from $\mathbb{R} \rightarrow \mathbb{R}$ or $\mathbb{R}_+ \rightarrow \mathbb{R}_+$. We say that $f(x)$ is $O(g(x))$ if there is a positive real constant $C$ and some constant $k$, such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

We would say "$f(x)$ is big-oh of $g(x)$".

At least Rosen refers to $C$ and $k$ as witnesses. I am unfamiliar with this nomenclature. **Definition:** $C$ and $k$ are referred to as witnesses.

**(b) Example:** $f(x) = x^2 + 2x + 1 = O(x^2)$

Suppose $x > 1$, then we find $x < x^2$ and $1 < x^2$, so that $f(x) < 4x^2 = O(x^2)$

Note: The use of the equals sign does not indicate a real mathematical equality in this context. It indicates the existence of the inequality given in the definition.
c. \( f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 = O(x^n) \)
where \( a_n \in \mathbb{R}^{+} \) and all other \( a_j \in \mathbb{R} \).

If \( x > 1 \),

\[ \left| f(x) \right| = \left| a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 \right| \]
\[ \leq \left| a_n x^n \right| + \left| a_{n-1} x^{n-1} \right| + \ldots + \left| a_0 \right| \]
\[ = x^n \left( |a_n| + |a_{n-1}|/x + \ldots + |a_0|/x^n \right) \]
\[ \leq x^n \left( |a_n| + |a_{n-1}| + \ldots + |a_0| \right) \]

Letting \( c = |a_n| + |a_{n-1}| + \ldots + |a_0| \),
the result is proved.

d. \( n! = O(n^n) \); \( \log n! = O(n \log n) \)

\[ n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq n \cdot n \cdot n \cdot n \cdots n = n^n \]
Taking \( c = 1 \) and \( k = 1 \), \( n! = O(n^n) \).
Hence, we find
\[ \log n! \leq \log n^n = n \log n \]

Rosen 144-146 & 123-124
There is also space complexity

**Complexity (focusing on time complexity)**

a. An analysis of the computer time required to solve a problem is called time complexity. An analysis of the memory involved is called space complexity. When one says that an algorithm is \( \Theta(n^2) \), that means that
\[ C_1 n^2 \leq \text{time}(n) \leq C_2 n^2, \]
where \( C_1 \) and \( C_2 \) are positive constants. We say that the algorithm is "order-\( n^2 \)."

b. This \( \Theta \)-notation is an extension of big-\( O \) notation. More generally, we need to show both sides of the equality.
Example: \[ 1 + 2 + \cdots + n = \Theta \left( n^2 \right) \]

Proof:
\[ S = 1 + 2 + \cdots + n = n + n + \cdots + n = n \cdot n = n^2 \]

Hence, \( S(n) = \Theta(n^2) \)

Also,
\[
S(n) = 1 + 2 + \cdots + n \\
\geq \left( \frac{n^2}{2} + \frac{n}{2} \right) + 1 + \cdots + n \\
\geq \left( \frac{n^2}{2} + \frac{n}{2} \right) + \cdots + \frac{n}{2} \\
= \left( n - \frac{n}{2} + 1 \right) \cdot \frac{n}{2} \\
\geq \left( \frac{n}{2} \right)^2 = \frac{n^2}{4}
\]

Hence, \( S(n) \geq C_1 n^2 \), where \( C_1 = \frac{1}{4} \), which we write \( S(n) = \Omega(n^2) \).

Since \( S(n) = \Theta(n^2) \) and \( S(n) = \Omega(n^2) \), then \( S(n) = \Theta(n^2) \).

C. Consider the linear search algorithm

(1) Linear search: We compare
\( x \) to \( a_1, a_2, \ldots, a_n \) sequentially. If \( x = a_j \), the algorithm halts and we set location to \( j \). If we go through the list, we set location = 0.

(2) Algorithm:

\begin{verbatim}
procedure linear_search (x:
integer, a_1, a_2, ..., a_n: distinct
integers)

i := 1

while (i \leq n and x \neq a_i)
  i := i + 1

if i \leq n then location := i
else location := 0

If x = a_i, we need 2n comparisons. At each
for i, we determine if i \leq n and if x = a_i.
Outside the loop, we make one comparison.
\end{verbatim}
When $x$ is not in the loop, we make $2n+2$ comparisons. Hence the algorithm is $O(n)$. This is the worst-case complexity.

(3) We can also average over all possible inputs to find the average-case complexity.

One average, the number of comparisons if $x$ is in the list is:

$$\frac{3+5+7+\ldots+(2n+1)}{n}$$

$$= \frac{2}{n} \frac{(1+2+3+\ldots+n)+1}{n} = \frac{2}{n} \frac{n(n+1)}{2} + 1$$

$$= n + 2 \in \Theta(n)$$

which is the average-case complexity.

9. Complexity of the merge sort algorithm

a. Lemma: Two sorted lists with $m$ elements and $n$ elements can be merged into a sorted list with $m+n-1$ operations.

Proof: In the worst case, $m+n-2$ comparisons will be carried out, leaving one element in both $L_1$ and $L_2$. After the next comparison, either $L_1$ or $L_2$ will be empty, leaving a total of $m+n-1$ comparisons.

6. We now consider a list of length $n$ and determine the complexity of the merge sort algorithm.
Note that this has nothing to do with $m$ and $n$ in the previous lemma.

We consider the special case when $n = 2^m$. At the end of the splitting, we have $2^m$ lists, each with one element. We now combine lists, starting at $k = m$, and going through $k = m-1, k = m-2, \ldots, k = 1$.

At level $k$, $2^k$ lists are merged into $2^{k-1}$ lists. Each of these $2^k$ lists has $2^{m-k}$ elements. From our lemma, each merger has at most $2^{m-k} (2^{m-k+1} - 1)$ comparisons. Hence, $2^{k-1} (2^{m-k+1} - 1)$ comparisons are needed at most to go from level $k$ to level $k-1$. The total number of comparisons is at most:

$$\sum_{k=1}^{m} 2^{k-1} (2^{m-k+1} - 1) = \sum_{k=1}^{m} 2^{m} - \sum_{k=1}^{m} 2^{k-1}$$

$$= m 2^{m} - (2^{m-1}) = n \log n - n + 1$$

$$= O(n \log n)$$

Hence, the complexity is $n \log n$ in the worst case.