Observe that the function $x \to nx$ maps

$$[0, 1) \rightarrow [0, n)$$

And hence the subintervals

$$[0, \frac{1}{n}) \rightarrow [0, 1)$$

$$[\frac{1}{n}, \frac{2}{n}) \rightarrow [1, 2)$$

$$\vdots$$

$$[\frac{i-1}{n}, \frac{i}{n}) \rightarrow [i-1, i)$$

$$\vdots$$

$$[\frac{n-1}{n}, 1) \rightarrow [n-1, n)$$

Thus $x \in [\frac{i-1}{n}, \frac{i}{n})$ is mapped to $nx \in [i-1, i)$,

$$\therefore \lfloor nx \rfloor = i-1 \quad \therefore \lfloor nx \rfloor + 1 = i.$$

Thus all elements in the $i^{th}$ subinterval are placed in the $i^{th}$ bucket $B[i]$.

The cost of all operations other than (5) is obviously $\Theta(n!)$.
If there are $n_j$ elements in bucket $i$, then the (average) run time of BucketSort is

$$t(n) = \Theta(n) + \sum_{j=1}^{n} \Theta(n_j^2)$$

Our assumption implies that on average, each $n_j = 1$. Thus $\sum_{j=1}^{n} \Theta(n_j^2) = \Theta(n)$, whence

$$t(n) = \Theta(n)$$

(See text for more rigorous treatment.)
Consider the set of all algorithms (known or unknown) which solve some problem $P$ in all its instances.

Our goals are twofold:

1.) Find an algorithm which solves $P$ in (worst case) time $O(f(n))$ for some function $f(n)$ which we aim to reduce as far as possible.

2.) Prove that any algorithm which solves $P$ must run in (worst case) time $\Omega(g(n))$ for some function $g(n)$ which we aim to increase as far as possible.

Here $n$ denotes the 'size' of an instance of problem $P$.

We are happy when $f(n) = \Theta(g(n))$ for then we know we have the best possible algorithm to solve $P$ (apart from improvements in hidden constants.)
(1) is called **algorithms** by some authors, while (2) is the **theory of computational complexity**. The function $f(n)$ is in (2) is called a **lower bound on the complexity** of problem P.

**Decision Trees / Information Theoretic Lower Bounds**

Ex. Let $m$ be an integer in the range $1 \leq m \leq 6$. **Problem**: Determine the value of $m$ by asking a sequence of **yes/no** questions.

This problem is similar to **binary search**!
Apparently the answer can be obtained by asking no more than 3 questions. (Will 2 suffice?)


A **graph** \( G = (V, E) \) is a pair of sets called **vertices** \( (V) \) and **edges** \( (E) \). Each edge joins a (unique) pair of (distinct) vertices. Two vertices which are joined by an edge are called **adjacent**.

\[
\begin{align*}
&\text{A vertex} \quad e \\
&\text{adjacent to} \quad u \quad \text{and} \quad v
\end{align*}
\]

A **path** in \( G \) is a sequence of consecutively adjacent vertices. \( G \) is called **connected** if every pair of vertices in \( G \) are joined by a path. A **cycle** is a closed path, i.e., a path in which the initial and terminal vertices are identical. \( G \) is called **acyclic** if it contains no cycle. A **tree** is a connected acyclic graph.
A rooted tree is a tree in which one vertex has been distinguished as the root. The vertices in such a tree are often called nodes. The depth of a node is its distance from the root. (Distance means shortest path length.)

![Tree Diagram]

The children of a node $x$ are those nodes adjacent to $x$ whose depth is one more than that of $x$. The parent of $x$ is the (unique) node which is adjacent to $x$ and has depth one less than that of $x$.

The root is the only node which has no parent (since the root has depth 0). If a node $y$ has no children, it is called a leaf. A non-leaf node is called an internal node.
The height of a rooted tree is the maximum node depth, i.e., the length of a longest downward path from the root to a leaf. The height of a node is the height of the subtree rooted at that node.

A binary tree is a rooted tree in which each node has at most 2 children. More generally, a k-ary tree is a rooted tree in which each node has at most k children.

A complete binary tree (CBT) is a binary tree in which all leaves are at the same depth, and each internal node has exactly 2 children.

\[
h = 3
\]

<table>
<thead>
<tr>
<th>Depth</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

\# nodes at depth \(d\) = \(2^d\)
\# leaves = \# nodes at depth \(h\) = \(2^h\)

\[
\text{The height of a CBT with } n \text{ leaves is } h = \log_2(n).
\]
An almost complete binary tree (ACBT) is a binary tree which has the maximum possible number of nodes at each depth, except possibly the last, which is filled from left to right.

\[ h = 3 \]

(An ACBT is the basis of the heap data structure.)

Exercise:
Prove that an ACBT with \( n \) leaves and height \( h \) satisfies \( h = \lceil \log n \rceil \).

Theorem:
The height \( h \) of any binary tree with \( n \) leaves satisfies

\[ h \geq \lceil \log n \rceil \]

Notation:
Let \( L(T) \) and \( H(T) \) denote the number of leaves and the height of a