Observe that the function $x \mapsto nx$ 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 $nx \in [i-1, i)$,

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

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 $O(n)$. 
If there are $n_i$ elements in bucket $i$, then the (average) run time of BucketSort is

$$T(n) = \Theta(n) + \sum_{i=1}^{n} \Theta(n_i^2).$$

Our assumption implies that on average, each $n_i = 1$. Thus

$$\sum_{i=1}^{n} \Theta(n_i^2) = \Theta(n),$$

whence

$$T(n) = \Theta(n).$$

(See text for more rigorous treatment.)
Consider the problem of computing the binomial coefficient \( \binom{n}{k} \) using Pascal's identity:

\[
\binom{n}{k} = \begin{cases} 
1 & \text{if } k = 0 \text{ or } k = n \\
\binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\
0 & \text{otherwise}
\end{cases}
\]

The obvious recursive algorithm is:

\[
\text{BinCoeff}(n, k)
\]

1. if \( k = 0 \) or \( k = 1 \)
2. return 1
3. else
4. return \( \text{BinCoeff}(n-1, k-1) + \text{BinCoeff}(n-1, k) \)

Observe that at the bottom level BinCoeff always returns 1, so ultimately it just adds a lot of 1's. Thus BinCoeff runs in time:

\[
\Omega\left(\binom{n}{k}\right)
\]
Exercise

Show that if \( n = 2k \), then

\[
\binom{n}{k} = \Theta \left( \frac{2^n}{\sqrt{n}} \right) = \Theta \left( \frac{4^k}{\sqrt{k}} \right)
\]

Hint: Use Stirling's formula.

The recursive tree shows the problem.

Observe that many of the same values are computed multiple times.

A more efficient approach is to maintain a table of intermediate results.
In fact, it's not necessary to store the whole table, just a single row at a time.

**DynamicSet** $(n, k)$
1. $C[0] \leftarrow 1$
2. for $i \leftarrow 1$ to $k$
3. 

4. for $i \leftarrow 1$ to $n$
5. for $i \leftarrow k$ to $1$
6. $C[i] \leftarrow C[i-1] + C[i]$
7. return $C[k]$.

Note: There is still some efficiency to be gained by defining all zeros, not calculating all of bottom row, etc. This is left as an exercise.
The same problem arises frequently in the divide & conquer approach. The obvious and natural way of dividing a problem into subinstances leads to overlapping (or identical) subinstances which are solved multiple times, leading to an inefficient algorithm.

In dynamic programming we arrange to solve each subinstance only once, saving the result for later use.

We typically store this information as a table (e.g., 2-dimensional array) of known results.

The word "programming" is used here in an archaic sense. "Program" is synonymous with "table", as in linear programming.

Dynamic programming is a technique for solving optimization problems. We seek an optimal solution from among a set of feasible solutions.
Coin Change Problem

Suppose we have coins in \( n \) denominations \( d_1, d_2, \ldots, d_n \), where each \( d_i \) is an integer. We wish to pay an amount \( N \) using the fewest number of coins possible.

Assumption: There is an unlimited supply of coins in each denomination.

There are two questions:

- What is the least number of coins needed to pay \( N \) units.
- Exactly which coins (i.e., which denominations) are to be used.

To answer the first question, we create a table \( C[1..n; 0..N] \), where

\[
C[i,j] = \text{minimum number of coins necessary to pay amount } j \text{ using coins in denominations } \{d_1, \ldots, d_i\}, \quad 1 \leq i \leq n, \quad 0 \leq j \leq N.
\]

Thus we seek \( C[n,N] \).
First observe that \( C[i, 0] = 0 \) for all \( 1 \leq i \leq n \).

Next, notice that to pay the amount \( i \) using denominations \( \{d_1, \ldots, d_i\} \) we have in general two choices:

1. Use no coins values \( d_1 \) (even though this is permitted), we can do this using \( C[i-1, j] \) coins.

2. Use at least 1 coin of value \( d_i \). After handing over one such coin, there are \( j-d_i \) units left to pay from denominations \( \{d_1, \ldots, d_i\} \). We can do this using \( 1 + C[i, j-d_i] \) coins.

\( C[i, j] \) should be whichever alternative uses the fewest coins. Thus,

\[
C[i, j] = \min(C[i-1, j], 1 + C[i, j-d_i])
\]

Note if \( i = 1 \) or \( j < d_i \) then one of the values on right falls outside the table.

It is convenient to think of such values as being +∞.