1. **Computing the Binomial Coefficients:**

A binomial is an expression like \((x + y)\). Algebraic expansion of powers of a binomial is done through the Binomial Theorem:

\[
(x + y)^n = \sum_{m=0}^{n} \binom{n}{m} x^{n-m} y^m
\]

\[
\binom{n}{m} = \text{Binomial Coefficient} = \frac{n!}{m!(n-m)!}
\]

**Example:**

\[
(x + y)^4 = \binom{4}{0} x^4 + \binom{4}{1} x^3 y + \binom{4}{2} x^2 y^2 + \binom{4}{3} xy^3 + \binom{4}{4} y^4
\]

\[
= x^4 + 4x^3 y + 6x^2 y^2 + 4xy^3 + y^4
\]

The coefficients are represented by the Pascal Triangle:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

The Binomial Coefficients can be computed using the following recurrence relation:

\[
\binom{n}{m} = \begin{cases} 
\binom{n-1}{m} + \binom{n-1}{m-1} & n \neq m, \quad m \neq 0, \\
\binom{n}{0} = 1 & \end{cases}
\]

A recursive algorithm to compute the binomial coefficients proves to be exponential. A dynamic programming (DP) algorithm using an \(n \times m\) table is of complexity \(O(nm)\).

Using a table \(n \times m\), \(n = 0.5, m = 0.3\), trace the DP algorithm to find \(\binom{5}{3}\)

**Solution:**

**Algorithm**

Table \(T[n,m]\)

```plaintext
for (i = 0 to \(n - m\))  T[i, 0] = 1;
for (i = 0 to \(m\)) T[i, i] = 1;
for (j = 1 to \(m\))
    for (i = \(j + 1\) to \(n - m + j\))
        T[i, j] = T[i - 1, j - 1] + T[i - 1, j];
return T[n, m];
```
Hence \( \binom{5}{3} = 10 \)

2. Consider the following function:

\[
F(n) = \sum_{i=1}^{n-1} F(i)F(i-1) \quad \text{for } n > 1 \quad \text{with } F(0) = F(1) = 2
\]

Consider \( T(n) \) to be the number of arithmetic operations used to compute the function.
a) Show that a direct recursive algorithm would give an exponential complexity.
b) Explain how, by not re-computing the same \( F(i) \) value twice, one can obtain an algorithm with \( T(n) = O(n^2) \).
c) Give an algorithm for this problem that only uses \( O(n) \) arithmetic operations.

**Solution:**
a) A direct recursive algorithm is:

```plaintext
F(n)
{
    if (n == 0) OR (n == 1) return 2;
    sum = 0;
    for i = 1 To n-1 { sum = sum + F(i) * F(i-1); }
    return sum;
}
```

Analysis gives

\[
T(n) = \sum_{i=1}^{n-1} \{ T(i) + T(i-1) + 2 \} = 2 \sum_{i=0}^{n-1} T(i) - T(n-1) - T(0) + 2(n-1)
\]

\[
T(n-1) = 2 \sum_{i=0}^{n-2} T(i) - T(n-2) - T(0) + 2(n-2)
\]

\[
T(n) - T(n-1) = 2T(n-1) - T(n-1) + T(n-2) + 2
\]

\[
T(n) = 2T(n-1) + T(n-2) + 2 \geq 2T(n-1) = \Omega(2^n)
\]

b) A DP algorithm using a table \( T[ ] \) and with complexity \( O(n^2) \) is:

```plaintext
F(n)
{
    T[0]=T[1]=2;
    for i = 2 To n
```
c) A DP algorithm using a table T[ ] and with complexity $O(n)$ is:

```c
F(n) {
    T[0] = T[1] = 2;
    for i = 2 To n
        T[i] = T[i-1] + T[i-1] * T[i-2];
    return T[n];
}
```

3. Consider evaluating the following recursive function:

$$\text{Fun}(n) = n \sum_{i=0}^{n-1} \text{Fun}(i) + n \quad \text{for } n > 0 \quad \text{with } \text{Fun}(0) = 0$$

The first few terms are: 0, 1, 4, 18, 96, ....

a) Show that a direct recursive algorithm would give an exponential complexity $O(2^n)$ in the number of arithmetic operations.

b) Implement a DP algorithm with $T(n) = O(n^2)$

c) Give a DP algorithm for this problem that only uses $O(n)$ arithmetic operations.

**Solution**

a) A direct recursive algorithm would be:

```c
int Fun(int n) {
    if (n == 0) return 0;
    else {
        int sum = 0;
        for(int i = 0; i < n; i++) sum = sum + Fun(i);
        return n * sum + n;
    }
}
```
**Answer:**
To show that this algorithm is exponential, consider the additions in the loop and the two additions outside:

\[
T(n) = \sum_{i=0}^{n-1} \{T(i) + 1\} + 2 \quad \text{for } n > 0 \text{ with } T(0) = 0
\]

\[
T(n - 1) = \sum_{i=0}^{n-2} \{T(i) + 1\} + 2 \quad \text{for } n > 1
\]

\[
T(n) - T(n - 1) = T(n - 1) + 1
\]

\[
T(n) = 2T(n - 1) + 1 \quad \text{for } n > 1 \quad \text{with } T(1) = 3
\]

Using sheet we get
\[
T(n) = 3 \cdot 2^{n-1} + \sum_{i=2}^{n} 2^{n-i} - 1 = O(2^n)
\]

b) A Dynamic Programming approach would use an array to store previous results:

```c
int Fun(int n)
{
    int a[Max], sum;   a[0] = 0;
    if (n == 0) return a[0];
    else {
        for (int i = 1; i <= n; i++)
            {  sum = 0;  for (int j = 0; j < i; j++) sum = sum + a[j];
                a[i] = i * sum + i; }
        return a[n];
    }
}
```

\[
T(n) = \sum_{i=1}^{n} \left\{ \sum_{j=0}^{i-1} 1 + 2 \right\} = \sum_{i=1}^{n} (i + 2) = n(n + 1)/2 + 2n = O(n^2)
\]

with an extra space of size \( \approx n \)

c) **Hint:** Find the relation between \( \text{Fun}(n) \) and \( \text{Fun}(n-1) \). In this case you may even be able to do without the table and obtain a linear algorithm.
\[ \text{Fun}(n) = n \sum_{i=0}^{n-1} \text{Fun}(i) + n \quad \text{for } n > 0 \quad \text{with } \text{Fun}(0) = 0 \]

\[ \text{Fun}(n) \neq n \sum_{i=0}^{n-1} \text{Fun}(i) + 1 \]

\[ \text{Fun}(n-1)/(n-1) = \sum_{i=0}^{n-2} \text{Fun}(i) + 1 \]

Subtracting and re-arranging we get:

\[ \text{Fun}(n) = \frac{n^2}{n-1} \text{Fun}(n-1) \quad \text{for } n > 1 \quad \text{with } \text{Fun}(0) = 0, \text{Fun}(1) = 1 \]

Hence, for \( n = 2 \) and above, we compute the new value by multiplying the old value by a factor.

**Algorithm**

\[ \text{fun}(n) \]
\{  
  if \((n==0)||(n==1))\) return \(n; \)  
  else  
    \{  
      \(f=1;\)  
      for \(i = 2 \) to \(n\)  
        \(f = (i \times i)/(i-1)*f;\)  
        return \(f;\)  
      \}
  \}

4. A similar problem is to evaluate the function:

\[ C(n) = \left(2/n\right)\sum_{i=0}^{n-1} C(i) + n \quad \text{with } C(0) = 1 \]

5. **Longest Common Subsequence (LCS)**
   Given two sequences \( A = \{a_1, \ldots, a_m\} \) and \( B = \{b_1, \ldots, b_n\}. \) Find the length of the longest sequence that is a subsequence of both \( A \) and \( B. \) For example, if \( A = \{aaadebcbac\} \) and \( B = \{abcadebcbec\}, \) then \( \{adebcb\} \) is subsequence of length 6 of both sequences.
   Give a Dynamic Programming algorithm together with its analysis.
Solution:
LCS Length (A[1..m],B[1..n])
T = array [0..m, 0..n]
For i=0 to m do T(i,0)=0
For j=0 to n do T(0,j)=0
For i=1 to m do
    For j=1 to n do
        if a(i) = b(j) then T(i,j)=T(i-1,j-1) + 1
        else T(i,j)= MAX( T(i, j-1), T(i-1, j))
return T(m,n)

This algorithm is O(nm)

6. The Shortest Common Supersequence (SCS)
Given two sequences X = <x1,...,xm> and Y = <y1,...,yn>, a sequence U = <u1,...,uk> is a common supersequence of X and Y if U is a supersequence of both X and Y.
The shortest common supersequence (SCS) is a common supersequence of minimal length. In the shortest common supersequence problem, the two sequences X and Y are given and the task is to find a shortest possible common supersequence of these sequences. In general, the SCS is not unique.
For two input sequences, an SCS can be formed from an LCS easily. For example, if X[1..m] = abcbdab and Y[1..n] = bdcaba, the LCS is Z[1..r] = bcba. By inserting the non-lcs symbols while preserving the symbol order, we get the SCS: U[1..t] = abdcabdab.
The Length of the SCS is SCS Length(X,Y) = n+m-Length LCS(X,Y)

7. Longest Increasing Subsequence (LIS)
The Longest Increasing Subsequence problem is to find the longest increasing subsequence of a given sequence. It also reduces to a Graph Theory problem of finding the longest path in a Directed acyclic graph.
Formally, the problem is as follows:
Given a sequence a1, a2, ...an, find the largest subset such that for every i < j, ai < aj.
A simple way of finding the longest increasing subsequence is to use the Longest Common Subsequence (Dynamic Programming) algorithm:
- Make a sorted copy of the sequence A, denoted as B. O(nlog(n)) time.
- Use Longest Common Subsequence on with A and B. O(n^2) time.

Dynamic Programming
There is a straight-forward Dynamic Programming solution in O(n^2) time. Let A be the sequence and define qk as the length of the longest increasing subsequence of A, subject to the constraint that the subsequence must end on the element ak. The longest increasing subsequence of A must end on some element of A, so that we can find its length by searching for the maximum value of q. All that remains is to find out the values qk.
But qk can be found recursively, as follows: consider the set Sk of all i < k such that ai < ak. If this set is null, then all of the elements that come before ak are greater than it,
which forces $q_k = 1$. Otherwise, if $S_k$ is not null, then $q$ has some distribution over $S_k$. By the general contract of $q$, if we maximize $q$ over $S_k$, we get the length of the longest increasing subsequence in $S_k$; we can append $a_k$ to this sequence, i.e.

$$q_k = \max(q_j \mid j \in S_k) + 1$$

If the actual subsequence is desired, it can be found in $O(n)$ further steps by moving backward through the $q$-array, or else by implementing the $q$-array as a set of stacks, so that the above "+ 1" is accomplished by "pushing" $a_k$ into a copy of the maximum-length stack seen so far.

A pseudo-code for finding the length of the longest increasing subsequence:

```java
LIS_length( a )
    n := a.length
    q := new Array(n)
    for k from 0 to n:
        max := 0;
        for j from 0 to k, if a[k] > a[j]:
            if q[j] > max, then set max = q[j].
        q[k] := max + 1;
        max := 0
    for i from 0 to n:
        if q[i] > max, then set max = q[i].
    return max;
```

8. A function $P(n, x)$ is defined for $n > 0$ to have a value at $(n)$ equal to half the product of all previous values, with $P(0, x) = x$.

- Implement a recursive algorithm to compute $P(n, x)$ for any zero or integer value of $(n)$ and give an analysis of the number of multiplications performed by this algorithm.
- Implement a Dynamic Programming algorithm that uses an array of size $(n)$ and give an analysis of the number of multiplications performed by the algorithm.

**Solution:**

A direct recursive algorithm would be:

```java
P( n, x )
{
    if (n == 0) return x;
    else
        { int m = 1;
          for(int i = 0; i < n; i++) m = m * P(i,x);
          return m/2;
        }
}
```
To show that this algorithm is exponential, consider the multiplication in the loop.

\[ T(n) = \sum_{i=0}^{n-1} T(i), \quad T(n-1) = \sum_{i=0}^{n-2} T(i) \text{ for } n > 1 \]

Subtraction gives \( T(n) = 2T(n-1), \text{ for } n > 1, \quad T(1) = 1 \)

This recurrence has a solution \( T(n) = 2^n - 1 = O(2^n) \)

A Dynamic Programming approach would use an array to store previous results:

```c
int Fun(int n)
{
    int *a = new int[n]; a[0] = 0;
    if (n == 0) return a[0];
    else {
        for (int i = 1; i <= n; i++)
            m = 1; for (int j = 0; j < i; j++) m = m * a[j];
        a[i] = m/2;
    }
    return a[n];
}
```

Analysis gives:

\[ T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2) \]

9. Suppose that the probabilities of searching for certain words in a document were:

- BAT (18 %)
- CAT (22 %)
- DOG (18 %)
- EGG (20 %)
- HAT (22 %)

Show in detail the Dynamic Programming approach to insert these words in an Optimal Binary Search Tree.

Draw the tree and compute the average search cost.

What will be the optimal tree and average search cost if all words have equal probabilities?

10. Suppose that the probabilities of occurrence of certain symbols in a message were:

- E (15 %)
- B (24 %)
- D (16 %)
- A (26 %)
- C (19 %)

Show in detail the Dynamic Programming approach to insert these symbols in an Optimal Binary Search Tree. Draw the tree and compute the average search cost.

What will be the tree and the average search cost if a greedy method is used instead?