Analysis of Algorithms

The natural logarithm of $(1+x)$, i.e. $\ln (1+x)$ for $-1 < x < 1$ can be evaluated by the approximation:

$$p(x) = \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + \frac{x^n}{n}$$

Consider the variable $x$ to be of type float. The value of $x^i$ is computed by a function `pow(x,i)` using $(i - 1)$ float multiplications. The algorithm is:

```c
float p = 0; float s = -1.0;
for (int i = 1; i <= n; i++)  { s = -s ;    p = p + pow(x,i) / i * s ; }
```

(a) What is the number of float arithmetic operations for a single iteration (i) of the loop?

(b) What is the total number of such operations $T(n)$ done by the algorithm, and what is its complexity (Big-O)?

(c) A faster algorithm is:

```c
float p = 0; float s = -1.0;
for (int i = 1; i <= n; i++)  { s = -s * x ;   p = p + s / i  ; }
```

Why is this algorithm faster than the direct one? (explain by comparing the two Big-O’s).

Answer:

(a) Number of float arithmetic operations for a single iteration (i) of the loop is $4 + (i - 1) = i + 3$

(b) $T(n) = 1 + \text{sum from } i = 1 \text{ to } n \text{ of } (i + 3) = 1 + n(n+1)/2 + 3n = O(n^2)$

(c) The number of float arithmetic operations inside loop is (4), and the loop is done (n) times so that $T(n) = 1 + 4n = O(n)$. The second algorithm is faster because $O(n) < O(n^2)$.

Find the Big-O for the following number of operations:

1. $T(n) = n^3 + 100n \log n + 500 = O(n^3)$
2. $T(n) = 4^n + n^3 = O(4^n)$
3. $T(n) = 0.01n \log n + 8 \log n = O(n \log n)$
4. $T(n) = 1 + 3 + 9 + 27 + \ldots + 3^{n-1} = (3^n - 1) / 2 = O(3^n)$
The multiplication of two square matrices $A_{n \times n}$ and $B_{n \times n}$ produces a matrix $C_{n \times n} = A \times B$ whose elements are given by:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}, \quad i, j = 0, \ldots, n - 1$$

Write the algorithm to receive $A$, $B$, and return $C$ using the above definition. Find the number of arithmetic operations done by this algorithm as a function of $n$.

**Answer:**

```plaintext
Algorithm MatrixMult (A[n][n], B[n][n], C[n][n])
for i = 0 to n-1
   for j = 0 to n-1
      sum = 0
      for k = 0 to n-1 sum = sum + A[i][k] * B[k][j]
      C[i][j] = sum
```

**Analysis:**

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 2 = 2n^3 = O(n^3)$$

Suppose program (A) takes $2^n/16$ units of time and program (B) takes $16n^2$ units:

1. For what values of $n$ does program (A) take less time than (B)?
2. For each of these programs, how large a problem can be solved in $2^{20}$ time units?

**Solution:**

1. At small $n$ (say $n = 4$), algorithm (A) takes 1 time unit while algorithm (B) takes a longer time of 256 units. At large $n$, algorithm (A) takes more time than (B) because $O(2^n) > O(n^2)$. They would spend the same time at a value of $n$ such that $2^n / 2^4 = 2^4 n^2$, $2^n = 8 n^2$.

   Taking Logs we get $n = 8 + 2 \log n$

   Excluding $n = 1$ then we must have $n > 10$

   Trial and error gives $n = 16$

   Hence program (A) will take less time than (B) for $n < 16$

2. Algorithm (A) takes $2^{20}$ time units to solve a problem of size $n = 24$, and algorithm (B) will take the same time to solve a problem of a bigger size of $n = 256$ because it is faster.