
CSCE 2211 Exercises
Exercises (2): Algorithm Analysis

Find positive constants c and n_0 to prove that

$T(n) = (n+1)^2 = O(n^2)$

i.e. what are c and n_0 such that

$(n+1)^2 \leq cn^2$ for $n > n_0$

Solution

Taking c = 2, we find n_0 such that $(n+1)^2 \leq 2n^2$. Hence, $n^2 + 2n + 1 \leq 2n^2$ is equivalent to $2n + 1 \leq n^2$ which is satisfied when $n > n_0 = 2$

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

$$\mathbf{1.}\ T(n) = n^3 + 100n \log n + 500 = O(n^3)$$

$$\mathbf{2.}\ T(n) = 4^n + n^3 = O(4^n)$$

$$\mathbf{3.}\ T(n) = 0.01 n \log n + 8 \log n = O(n \log n)$$

$$\mathbf{4.}\ T(n) = 1 + 3 + 9 + 27 + ... + 3^{n-1} = (3^n - 1)/2 = O(3^n)$$

The running times of certain algorithms are found to be as follows:

$T(n) = 10$ *(best case)*

$T(n) = 6 \log n^2$ *(worst case)*

$T(n) = 5 n^3$ *(always)*

What are the corresponding complexities of these algorithms?

The running times of certain algorithms are found to have the following bounds:

$T(n) \leq 5$ *for* $n \geq 2$

$T(n) \geq 2n$ *for* $n \geq 1$

$T(n) = 6 \log n$ *for* $n \geq 2$

What are the corresponding complexities of these algorithms?

Solution

$T(n) = O(1)$ $T(n) = \Omega(n)$ $T(n) = \Theta(\log n)$

Consider a randomly ordered array *a[0..n-1]* of size (*n*) elements and the following algorithm:

```
ALGORITHM FUN (a[0..n-1])
x = a0 ;
for i = 1 to n-1 do
        if (ai < x) x = ai ;
return x;
```

What does this algorithm do?

Find *T(n)* = number of comparisons done by the algorithm in the best and worst cases.

Is this algorithm tightly bound (exact) or loosely bound.

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

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

Consider the variable **x** to be of type <u>float</u>. The value of $\mathbf{x^i}$ is computed by a function **pow(x,i)** using (i -1) <u>float</u> multiplications. The algorithm is:

```
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 <u>float arithmetic operations</u> 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:

```
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).
{The sum of integers from 1 to n is equal to n(n+1) / 2}

<u>**Answer:**</u>

(a) Number of float arithmetic operations for a single iteration (i) of the loop is 4 + (i – 1) = i + 3
(b) T(n) = 1 + sum from i = 1 to n 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 + 4 n = O (n).
The second algorithm is faster because O (n) < O (n^2).

The multiplication of two square matrices $A_{n \times n}$ and $B_{n \times n}$ produces a matrix $C_{n \times n} = A*B$ whose elements are given by:

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

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.

<u>**Answer:**</u>

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
```

<u>**Analysis:**</u>

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

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 $\boldsymbol{2^n / 2^4 = 2^4 n^2}$, $\boldsymbol{2^n = 2^8 n^2}$,

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

   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.