CS-2023 Data Structures And Algorithms

Complexity Analysis Take Home Assignment

Sajeev Kugarajah (210554M)

(Q1)

1. **Little Oh notation (o)**
   - Little oh notation used to describe an upper bound on the growth of an algorithm which cannot be tight. in other words, loose upper bound.

     o(g(n)) = {f(n): for any positive constant c>0 there exists an integer n_0>0 such that
     $0 \leq f(n) < c.g(n)$ for all $n \geq n_0$}
   - Example:

     $2n = o(n^2)$

     let c>0 be arbitrary
     we know 2n>0 for all n>0
     now, $2n < c.n^2$
     $0 < c.n^2 - 2n$
     $0 < n(c.n - 2)$
     but, $n > 0$
     so, $n.c - 2 > 0$
     $n > 2/c$
     take, $n_0 = 2/c + 1$

     since c > 0 is arbitrary,

     for all c > 0, there exists $n_0 = 2/c + 1$ s.t for all $n \geq n_0$, $0 \leq 2n < c.n^2$

2. **Big Omega notation (Ω)**
   - Big Omega notation represents the lower bound of the running time of an algorithm. this provides the best case time complexity for a given algorithm.

     Ω(g(n)) = {f(n): there exists positive constant c,n_0 s.t
     $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$}

   - Example:

$n^2 = \Omega(3n - 2)$

$n^2 \geq 3n - 2$
$n^2 - 3n + 2 \geq 0$
$(n - 1)(n - 2) \geq 0$

so, $c = 1$, $n_0 = 2$: $0 \leq c.(3n - 2) \leq n^2$ for all $n \geq n_0$

3. Little Omega notation (ω)

- Very similar to the big omega notation, but it represents the loose lower bound of an algorithm like little oh notation.

  $\omega(g(n)) = \{f(n)$: for all $c > 0$, there exists $n_0 > 0$ s.t
  $0 \leq c.g(n) < f(n)$ for all $n \geq n_0\}$

- Example:

  $n^2/2 = \omega(n)$

  let $c > 0$ be arbitrary,
  $n^2/2 > 0$ and $n \geq 0$
  $c.n < n^2/2$
  $0 < n(n/2 - c)$
  so, $n > 2c$
  take $n_0 = 2c + 1$

  since c is arbitrary,
  for all $c > 0$, there exists $n_0 = 2c + 1$ s.t : $0 \leq c.n < f(n^2/2)$ for all $n \geq n_0$

(Q2)

Theta(θ)	**Big Oh(O)**	**Little Oh(o)**	**Big Omega(Ω)**	**Little Omega(ω)**
Asymptotic tight bound	Asymptotic upper bound May or may not be tight	Asymptotic upper bound (Loose bound)	Asymptotic lower bound (May or may not be tight)	Asymptotic lower bound (Loose bound)
$\Theta(g(n)) = \{f(n)$: there exists $c_1, c_2, n_0 > 0$ s.t $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0\}$	$O(g(n)) = \{f(n)$: there exists $c, n_0 > 0$ s.t $0 \leq f(n) \leq c.g(n)$ for all $n \geq n_0\}$	$o(g(n)) = \{f(n)$: there exists $c, n_0 > 0$ s.t $0 \leq f(n) < c.g(n)$ for all $n \geq n_0\}$	$\Omega\ (g(n)) = \{f(n)$: there exists $c, n_0 > 0$ s.t $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0\}$	$\omega\ (g(n)) = \{f(n)$: there exists $c, n_0 > 0$ s.t $0 \leq c.g(n) < f(n)$ for all $n \geq n_0\}$
Exact time complexity	Worst case time complexity	Approximates worst case time complexity	Best case time complexity	Approximately best case time complexity
$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$ $f(n) = o(g(n))$ iff $g(n) = \omega\ (f(n))$ $f(n) = \theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega\ (g(n))$				

(Q3)

1.

Version 1

<table>
<tr><th>Line No</th><th>Code</th><th>Cost (C_i)</th><th>Times (T_i)</th></tr>
<tr><td>1</td><td>for j = A.length to 2 do</td><td>C1</td><td>n</td></tr>
<tr><td>2</td><td>swapped = false</td><td>C2</td><td>n-1</td></tr>
<tr><td>3</td><td>for i = 2 to j do</td><td>C3</td><td>n(n+1)/2 – 1</td></tr>
<tr><td>4</td><td>swapped = false</td><td>C4</td><td>n(n-1)/2</td></tr>
<tr><td>5</td><td>if (A|i — 1|> A[i]) then</td><td>C5</td><td>n(n-1)/2</td></tr>
<tr><td>6</td><td>temp = A|i|</td><td>C6</td><td>n(n-1)/2</td></tr>
<tr><td>7</td><td>A|i| = A|i — 1|</td><td>C7</td><td>n(n-1)/2</td></tr>
<tr><td>8</td><td>A|i — 1| = temp</td><td>C8</td><td>n(n-1)/2</td></tr>
<tr><td>9</td><td>swapped = true</td><td>C9</td><td>n(n-1)/2</td></tr>
<tr><td>10</td><td>if (! swapped) then</td><td>C10</td><td>n(n-1)/2</td></tr>
<tr><td>11</td><td>break;</td><td>C11</td><td>0</td></tr>
<tr><td>12</td><td>n = newLimit</td><td>C12</td><td>n-1</td></tr>
</table>

T(n) = C1.n + C2.(n-1) + C3.(n(n+1)/2 - 1) + (C3+C4+C5+C6+C7+C8+C9+C10)(n(n-1)/2) + C12(n-1)

= C13.n + C14.n^2 + C15

T(n) = O(n^2)

Version 2

<table>
<tr><th>Line No</th><th>Code</th><th>Cost</th><th>Times</th></tr>
<tr><td>1</td><td>n = A.length</td><td>C1</td><td>1</td></tr>
<tr><td>2</td><td>do</td><td>C2</td><td>n</td></tr>
<tr><td>3</td><td>swapped = false</td><td>C3</td><td>n-1</td></tr>
<tr><td>4</td><td>for i = 2 to n do</td><td>C4</td><td>n(n+1)/2</td></tr>
<tr><td>5</td><td>if (A|i — 1| >A[i]) then</td><td>C5</td><td>n(n-1)/2</td></tr>
<tr><td>6</td><td>temp = A|i|</td><td>C6</td><td>n(n-1)/2</td></tr>
<tr><td>7</td><td>A|i| = A|i - 1|</td><td>C7</td><td>n(n-1)/2</td></tr>
<tr><td>8</td><td>A|i - 1| = temp</td><td>C8</td><td>n(n-1)/2</td></tr>
<tr><td>9</td><td>swapped = true</td><td>C9</td><td>n(n-1)/2</td></tr>
<tr><td>10</td><td>newLimit = i-1</td><td>C10</td><td>n(n-1)/2</td></tr>
<tr><td>11</td><td>n = newLimit</td><td>C11</td><td>n-1</td></tr>
<tr><td>12</td><td>while swapped</td><td>C12</td><td>n</td></tr>
</table>

T(n) = C1 + C2.n + C3(n-1) + C4(n(n+1)/2) + (C5+C6+C7+C8+C9+C10)(n(n-1)/2) + C11(n-1) + C12.n

= C13.n^2 + C14.n + C15

T(n) = O(n^2)

2.

Since both algorithms are having $O(n^2)$, there is no difference between the worst time complexities of both algorithms.

3.

Yes, instead of calculating every steps we can calculate using the loops and their nested loops and recursive terms.

The inner loop interactions will be multiplied and outer loops interactions will be added.

For example:

```
    for i=1 to n        --------------------- 1
        //////
        for j=1 to n/2     ---------- 2
            //////
        end for
    end for
    for m=1 to n/4 ----------  3
        //////
    end for
```

loop 1 will run n times, inner loop 2 will run n/2 times and outer loop 3 will run n/4 times

so, time complexity T(n) $= O(n.n/2 + n/4)$

$= O(n^2)$

in this way we can easily calculate the worst case time complexity as we need the largest term of the function.