**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 n0>0 such that

0 ≤ f(n) < c.g(n) for all n≥n0}

* Example:

2n = o(n2)

let c>0 be arbitrary

we know 2n>0 for all n>0

now, 2n < c.n2

0 < c.n2 – 2n

0 < n(c.n - 2)

but, n > 0

so, n.c – 2 > 0

n > 2/c

take, n0 = 2/c + 1

since c > 0 is arbitrary,

for all c > 0, there exists n0 = 2/c + 1 s.t for all n ≥ n0, 0 ≤ 2n < c.n2

1. **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,n0 s.t

0 ≤ c.g(n) ≤ f(n) for all n ≥ n0}

* Example:

n2 = Ω(3n - 2)

n2 ≥ 3n -2

n2 – 3n + 2 ≥ 0

(n – 1)(n – 2) ≥ 0

so, c = 1, n0 = 2 : 0 ≤ c.(3n - 2) ≤ n2 for all n ≥ n0

1. **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.

ω(g(n)) = {f(n): for all c > 0, there exists n­0 > 0 s.t

0 ≤ c.g(n) < f(n) for all n ≥ n0}

* Example:

n2/2 = ω(n)

let c > 0 be arbitrary,

n2/2 > 0 and n ≥ 0

c.n < n2/2

0 < n(n/2 - c)

so, n > 2c

take n0 = 2c + 1

since c is arbitrary,

for all c > 0, there exists n0 = 2c + 1 s.t : 0 ≤ c.n < f(n2/2) for all n ≥ n0

(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) |
| Θ(g(n)) = {f(n): there exists c1,c2,n0 > 0 s.t 0≤c1.g(n)≤f(n) ≤c2.g(n) for all n ≥ n0} | O(g(n)) = {f(n): there exists c,n­0 > 0 s.t 0≤f(n)≤c.g(n) for all n ≥ n0} | o(g(n)) = {f(n): there exists c,n­0 > 0 s.t 0≤f(n)<c.g(n) for all n ≥ n0} | Ω (g(n)) = {f(n): there exists c,n­0 > 0 s.t 0≤c.g(n)≤f(n) for all n ≥ n0} | ω (g(n)) = {f(n): there exists c,n­0 > 0 s.t 0≤c.g(n)<f(n) for all n ≥ n0} |
| 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) = Ω(f(n))  f(n) = o(g(n)) iff g(n) = ω (f(n))  f(n) = θ(g(n)) iff f(n) = O(g(n)) and f(n) = Ω (g(n)) | | | | |

(Q3)

1.

Version 1

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

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.n2 + C15

T(n) = O(n2)

Version 2

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

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.n2 + C14.n + C15

T(n) = O(n2)

2.

Since both algorithms are having O(n2), 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(n2)

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