CSCI203: Algorithms and Data Structures

Assignment 1 Solutions

Question 1a and 1b:

No.

Date

Q1a) 1. 10^{10} : This is a constant value, so growth rate is $O(1)$

2. $(\sqrt{2})^{\log_2 \log_2 n} = (\log_2 n)^{\log_2 \sqrt{2}}$: This is a log-logarithmic growth, which is a very slow growth rate.

3. $\log_2 \log_2 ((n+10)^n) \approx \log_2(\log_2 n)$: This is a log-logarithmic growth, which is a very slow growth rate but faster than $(\log_2 n)^{\log_2 \sqrt{2}}$.

4. $\log_2 (\log_2 n)^3 = 3 \cdot \log_2 (\log_2 n)$: This is a log-logarithmic growth, which is a very slow growth rate but faster than $\log_2 \log_2 ((n+10)^n)$

5. $2^{\log_2 \log_2 n} = \log_2 n$: This is a logarithmic growth, which is faster than log-logarithmic growth.

6. $\sqrt{16n^2} = 4n$: This is a linear growth, so growth rate is $O(n)$

7. $\sum_{k=1}^{n} = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$: This is a quadratic growth, so growth rate is $O(n^2)$

8. $27^{\log_3 n} = (3^3)^{\log_3 n} = n^3$: This is a cubic growth, so growth rate is $O(n^3)$

9. $2^{3n} = (2^3)^n = 8^n$: This is exponential growth, which is faster than all polynomial growth rate.

10. $(n-3)!$: This is a factorial growth, it is faster than exponential growth.

b) static int doIt (int n){

for $i \leftarrow 1$ to 10 do : runs exactly 10 times, which is constant, ∴ cost: $O(1)$

for $k \leftarrow 1$ to n do : runs n times for each iteration of the outer loop ∴ cost: $O(n)$

$j \leftarrow 1; m \leftarrow n$: outer loop runs 10 times, inner loop runs n times ∴ cost: $O(n)$

while $j < m$ do : for each iterations of the inner loop, the while loop runs $O(\log n)$ times, since the inner loop run n times, the total cost: $O(n \log n)$

$m \leftarrow (m+j)/2$: Same as while loop cost : $O(n \log n)$

∴ The overall complexity of doIt() is $O(1) + O(n) + O(n) + O(n \log n) + O(n \log n) = O(n \log n)$

Static int myMethod (int n){

$i \leftarrow 1$: execute once ∴ cost : $O(1)$

while $(i < n)${ : doubles on every iteration ∴ cost : $O(\log n)$

doIt(n) : called on every iteration of while loop, ∴ cost: $O(\log n) \times O(n \log n) = O(n \log^2 n)$

$i \leftarrow i \times 2$: This executes $O(\log n)$ times, ∴ cost: $O(\log n)$

}

return 1; : execute once, ∴ cost : $O(1)$

}

∴ the overall complexity of myMethod() is $O(1) + O(\log n) + O(n \log^2 n) + O(\log n) + O(1)$
$= O(n \log^2 n)$

A'ZONE

Question 1c:

c) $f(n) = \frac{n^2}{2} + n\lg n$

$g(n) = 5n(4+2^{\lg n})$

$f(n) \in O(g(n))$ if there exist a non-negative integer n_0 and positive real constant C such that $f(n) \leq g(n)$ for all $n \geq 0$

$\therefore \frac{n^2}{2} + n\lg n \leq c(5n(4+2^{\lg n}))$

$= \frac{n^2}{2} + n\lg n \leq c(20n + 5n^2)$

$\frac{f(n)}{g(n)} = \frac{\frac{n^2}{2} + n\lg n}{20n + 5n^2}$

$= \frac{\frac{1}{2} + \frac{\lg n}{n}}{\frac{20}{n} + 5}$

$= \frac{\frac{1}{2}}{5}$

$= \frac{1}{10}$ when $n = \infty$, $\frac{\lg n}{n} = 0$ and $\frac{20}{n} = 0$

Since $\frac{f(n)}{g(n)} = \frac{1}{10}$ when $n = \infty$, there exist a constant $C = 1$ or $C \geq 1$ such that $f(n) \leq c\,g(n)$ for all sufficiently large n, therefore $f(n) \in O(g(n))$

So final answer is yes, $\frac{n^2}{2} + n\lg n \in O(5n(4+2^{\lg n}))$

Question 2a:

No.

Date

2a) function Sum (A, left, right)

if left > right : time component $\therefore$ cost: $O(1)$

return 0 : constant operation for base case $\therefore$ cost: $O(1)$

else if left = right : time component $\therefore$ cost: $O(1)$

return A[left] : Accessing an array element is constant time operation $\therefore$ cost: $O(1)$

mid = floor ((left/right)/2) : Arithemic and floor operation are constant time operation $\therefore$ cost: $O(1)$

lsum = Sum (A, left, mid)

rsum = Sum (A, mid+1, right) } : This is recursive call, the problem size is halved on each recursive call $\therefore$ cost: $O(\frac{n}{2})$

return lsum + rsum + A[mid] : adding three value is a constant-time operation $\therefore$ cost: $O(1)$

$\therefore$ recurrance relation is $T(n) = 2T(\frac{n}{2}) + C$

expanding the reccurance relation : $T(n) = 2T(\frac{n}{2}) + C$

$= 2[2T(\frac{n}{2^2}) + C] + C$

$= 2^2T(\frac{n}{2^2}) + 3C$

$T(\frac{n}{2^2}) = 2^2[2T(\frac{n}{2^3}) + C] + 3C$

$= 2^3T(\frac{n}{2^3}) + 4C + 3C$

Generalised the recurrence Relation : $T(n) = 2^k T(\frac{n}{2^k}) + (2^k - 1)C$

The recursive call will stop when the base condition is met, which is $\frac{n}{2^k} = 1$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \lg n$

Subsititute k into generalised recurrence Relation : $T(n) = 2^k T(\frac{n}{2^k}) + (2^k - 1)C$

$= 2^{\lg n} T(\frac{n}{2^{\lg n}}) + (2^{\lg n} - 1)C$

$= nT(\frac{n}{n}) + (n-1)C$

$T(1) = 1 \leftarrow = nT(1) + cn - C$

$= n + cn - C$

$\therefore$ the running time efficiency is $O(n)$.

The worst case asymptotic complexity of this algorithm is the same as the algorithm that linearly iterates through the list. $\therefore$ The worst-case asymptotic complexity is $O(n)$

The best-case asymptotic complexity of this algorithm only reduces the constants associated with non-recursive work, but does not fundamentally change the reccurence $\therefore$ The best-case asymptotic complexity is $O(n)$

A'ZONE

Question 2b and 3 part i, ii, iii:

b) The best-case inputs and worst-case inputs are the same because the algorithm always ~~diva~~ divides the input into halves recursively until the base cases are reached. All n elements of the input are processed regardless of their values or arrangement and there is no condition to stop recursion early based on input properties.

3) i) $T(n) = 4T(\frac{n}{4}) + \frac{n}{\log_2 n}$, and $T(1) = 1$

$a = 4, b = 4, c = 1, p = -1$

$\frac{a}{b^c} = \frac{4}{4^1} = 1, p = -1$

$\therefore T(n) = O(n^{\log_b a} \times \log_b \log_b n)$

$= O(n^{\log_4 4} \times \log_4 \log_4 n)$

$= O(n \log_4 \log_4 n)$ *

ii) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$, and $T(1) = 1$

$a = 2, b = 4, c = \frac{1}{2}, p = 0$

$\frac{a}{b^c} = \frac{2}{4^{\frac{1}{2}}} = 1, p = 0$

$\therefore T(n) = O(n^{\log_b a} \times \log_b^{p+1} n)$

$= O(n^{\log_4 2} \times \log_4^1 n)$

$= O(n^{\frac{1}{2}} \log_4 n)$ *

iii) $T(n) = 2T(\frac{n}{2}) + n \lg^2 n$, and $T(1) = 1$

$a = 2, b = 2, c = 1, p = 2$

$\frac{a}{b^c} = \frac{2}{2^1} = 1$, ~~mm~~, $p = 2$

$\therefore T(n) = O(n^{\log_2 2} \times \log_2^3 n)$

$= O(n \log_2^3 n)$ *

Question 3 part iv, v, vi:

No.

Date

iv) $T(n) = 2T(\frac{n}{2}) + n \lg n$, and $T(1) = 1$

$a = 2$, $b = 2$, $c = 1$, $p = 1$

$\frac{a}{b^c} = \frac{2}{2^1} = 1$, $p = 1$

$\therefore T(n) = O(n^{\log_b a} \log_b^{p+1} n)$

$= O(n^{\log_2 2} \log_2^2 n)$

$= O(n \log_2^2 n)$ *

v) $T(n) = 2T(\frac{n}{2}) + n^3$, and $T(1) = 1$

$a = 2$, $b = 2$, $c = 3$, $p = 0$

$\frac{a}{b^c} = \frac{2}{2^3} = \frac{2}{8} = \frac{1}{4}$, $p = 0$

$\therefore T(n) = O(n^c \times \log_b^p n)$

$= O(n^3 \times \log_2^0 n)$

$= O(n^3)$ *

vi) $T(n) = T(n-3) + 3n + 3$

$= T(n-1-1) + 3(n-1) + 3 + 3n + 3$

$= T(n-2) + 3(n-1) + 3 + 3n + 3$ —— ①

$T(n) = T(n-1-2) + 3(n-1-1) + 3 + 3(n-1) + 3 + 3n + 3$

$= T(n-3) + 3(n-2) + 3 + 3(n-1) + 3 + 3n + 3$ —— ②

$T(n) = T(n-k) + 3[(n-k+1) + (n-k+2) + \ldots + n] + 3k$

Since base case $T(1) = 1$, for $k = n-1$

$T(n) = T(1) + 3\underbrace{[1+2+3+\ldots+n]}_{\frac{n(n+1)}{2}} + 3(n-1)$

$= T(1) + 3(\frac{n(n+1)}{2}) + 3(n-1)$

$= 1 + \frac{3n^2+3}{2} + 3(n-1)$

$= 1 + \frac{3n^2+3n}{2} + 3n - 3$

$= \frac{3n^2+3n}{2} + 3n - 2$

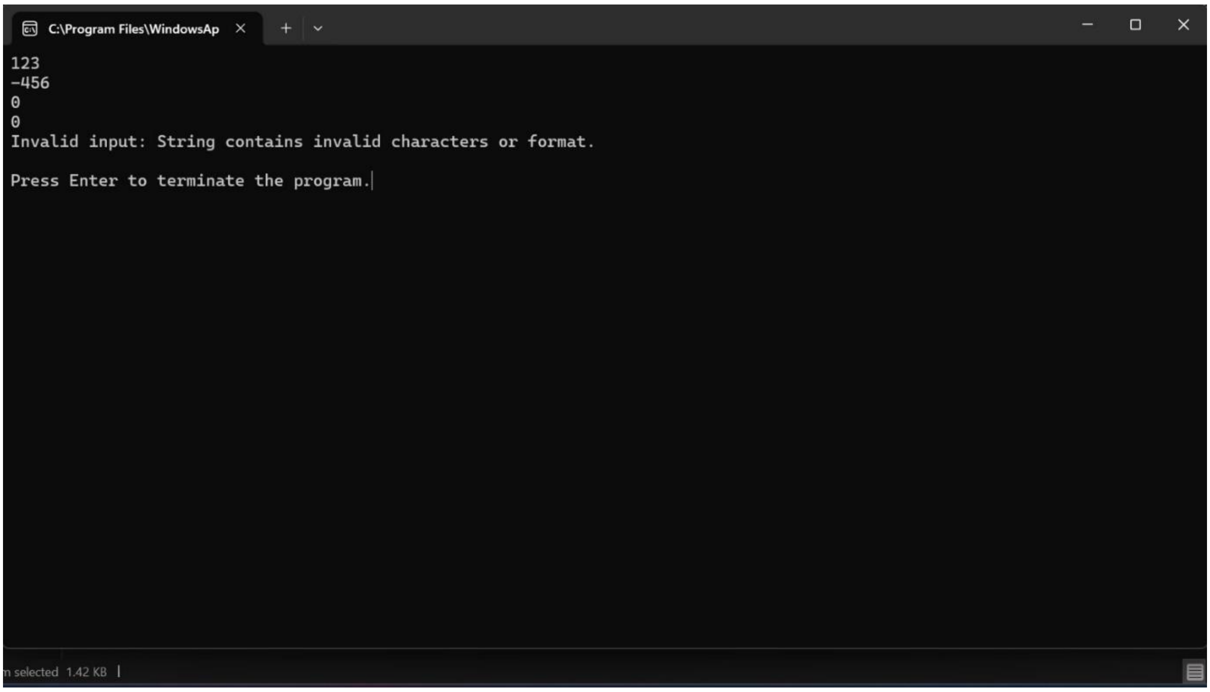
$= \frac{3}{2}n^2 + \frac{3n}{2} + 3n - 2$

$\therefore$ dominant term $= \frac{3}{2}n^2$

$T(n) = O(n^2)$ *.

Question 4a:

Screenshot of the output



Question 4b:

```
    # Recur for the rest of the string
    return recursive_convert(param[1:], result) # Recursive calls
```

The algorithm processes one character of the string in each recursive call. For a string of length n, there are n recursive calls to process all characters.

Each recursive call performs constant work (character conversion and basic arithmetic operations).

```
    digit = ord(param[0]) - ord('0')  # Convert character to integer
    result = result * 10 + digit
```

The ord(param[0]) - ord('0') and the arithmetic operations are Θ(1).

Since there are n recursive calls and each call does Θ(1)work, the total time complexity is Θ(n)