aryanrao-HW1

1.

(a) 8n3 + 9n2 + 5 ∈ *O*(n3)

Proof:

f(n) = 8n3 + 9n2 + 5 ≤ 8n3 + 9n3 + 5n3

f(n) = 8n3 + 9n2 + 5 ≤ 22n3

= c . n3

= c . g(n)

Choose c = 22, N = 1

(b) 22n+2 ∈ *O*(22n+1)

We need to show that for every constant c ≥ 1 and every threshold N > 0, there exists an n > N for which ƒ(n) ≥ c.g(n). Alternatively, we can use a proof by contradiction.

Assume there exist constants c ≥ 1 and N > 0 such that for all n > N, ƒ(n) < cg(n).

Let's consider n = N + 1. We have:

ƒ(N + 1) = 22N+1+2 =(22N+3)

> (22N+2) (since 2N+3 > 2N+2)

Now, let's consider c = 1. For n = N + 1, we have:

c.g(n) = 1 \* g(n) = g(n) = 22n+2

Since ƒ(N + 1) > cg(n) (which is 22N+2 )this contradicts our assumption.

Therefore, there does not exist any constant c ≥ 1 and threshold N > 0 for which ƒ(n) < cg(n) holds for all n > N. This implies that ƒ(n) = 22n+2 does not belong to O(22n+1)

(c) To prove that if ƒ ∈ O(g) and h is any positive-valued function, then fh ∈ O(gh):

Since ƒ ∈ O(g), there exists a constant c ≥ 1 such that for every n ≥ 1, ƒ(n) ≤ cg(n).

Now, we can prove that fh ∈ O(gh) by showing that for the same constant c, there exists a constant c2 ≥ 1 such that for every n ≥ 1, fh(n) ≤ c2gh(n).

Let's consider c2 = c. For every n ≥ 1, we have:

fh(n) = ƒ(n)h(n) ≤ cg(n)h(n)

(since ƒ(n) ≤ cg(n) for every n ≥ 1)

= c(gh(n))

Therefore, for c2 = c and n ≥ 1, we have fh(n) ≤ c2gh(n).

This proves that fh ∈ O(gh).

2.

(a) r = 0;

for i in the range [1, n] {

for j in the range [i, n]

for k in the range [1, j-i]

r++;

print (r);

}

Suppose the ‘k’ loop takes c1 time per iteration, ‘j’ loop takes c2 and ‘i’ loop takes c3 respectively.

T(n) =

Expanding,

T(n) =

T(n) = c3.n + n(n+2). c2 + n(n2 -1). c1

T(n) = c3.n + c2() + c1( )

T(n) = c3.n + c2() + c2() + c1() - c1()

Because n3 is of the highest order, T(n) = *O*(n3)

(b) x = Math.pow(2, n)

for (i = 1; i <=x; i = i\*2)

for j in range [1, i]

Constant Number of Operations

Outer loop (i) = 1to x, *i* = *i* \* 2

Inner loop (j) = 1 *to i*

x = *2n, n* =

x = = n

T(n) of outer loop = *O*(n)

Inner loop: number of iterations = 1 to *i = O(i)*

*i* = *i* \* 2:

20 + 21 + 22 + … + 2n-1

The number of iterations is 2n-1. Hence, T(n) = *O*(2n)

The total efficiency: *O*(n + 2n) = *O*(2n)

(c) i = n

while i >= 1

for j in range [1, i]

Constant Number of Operations

i = i / 2

Outer loop will run as long as ’i’ is greater than or equal to 1.

Inner loop is from 1 to ‘i.’

While loop: *i* = halves each time.

The outer loop is having a complexity: *O*

The inner loop is having a complexity: i = n ,

The number of iterations in each iteration of the outer loop decreases as i gets divided by 2. In the first iteration, the inner loop runs n times, then n/2 times, then n/4 times, and so on. The total number of iterations for the inner loop can be approximated as the sum of the geometric series n + n/2 + n/4 + ... +2n, which is 2n. Therefore, the runtime of the inner loop is O(n).

Combining these steps, the overall runtime of the given algorithm as a function of n is O(log n) \* O(n) = O(n log n)

Therefore, the algorithm's runtime has a Big-O upper bound of O(n log n).

3.

function polynomial( A, t){ //function to calculate the polynomial

n = length[A]

p\_total = 0 // variable to return the total value of polynomial

for i = 0 to n-1{ //assuming stating index is 0

p\_total = p\_total + A[i] \* Math.pow( t , i)// Math.pow is not a primitive function

}

return p\_total //return final value

}

The Big- O for this would be O(n) because the for loop will run ‘n’ times.

Big-O = O(n)

4.

function findMedian(A) {

int n = A.length

sort(A); // Sort the array in non-decreasing order

if n % 2 = 1{

// Array size is odd

return A[n / 2]; // Return the middle element as the median

}

else {

// Array size is even

int mid1 = n / 2

int mid2 = mid1 - 1

return (A[mid1] + A[mid2]) / 2 // Return the average of the two middle elements as the median

}

}

Pseudo code for a sort method as runtime is dominated by it:

Function MergeSort(A):

if length(A) <= 1:

return A

mid = length(A) / 2

left = A[0:mid] // Divide the array into two halves

right = A[mid:length(A)]

// Recursively sort the two halves

left = MergeSort(left)

right = MergeSort(right)

return Merge(left, right) // Merge the sorted halves

Merge(left, right):

merged = [] // Create an empty array to store the merged result

i = 0 // Index for the left array

j = 0 // Index for the right array

// Compare elements from the left and right arrays and merge them in sorted order

while i < length(left) and j < length(right):

if left[i] <= right[j]:

merged.append(left[i])

i = i + 1

else:

merged.append(right[j])

j = j + 1

// Append any remaining elements from the left array

while i < length(left):

merged.append(left[i])

i = i + 1

// Append any remaining elements from the right array

while j < length(right):

merged.append(right[j])

j = j + 1

return merged

Sorting the array A takes O(n log n) time in the worst case, where n is the size of the array.

Returning the middle element or the average of two middle elements takes constant time, denoted as O(1).

Therefore, the overall runtime of the algorithm is O(n log n) due to the sorting step, which dominates the runtime.

Hence, the worst-case runtime of the algorithm, expressed as a function of the size of the input array, is O(n log n).