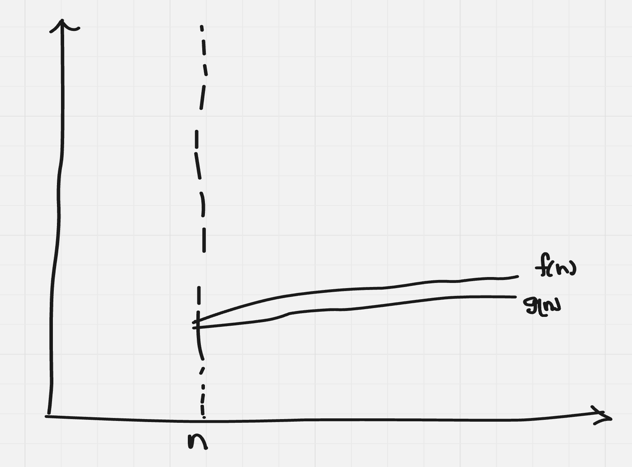
**COMP3112 Analysis of Algorithms Fall 2024 HW #1**

**Q1) 𝑓(𝑛) = 𝑛𝑙𝑜𝑔2𝑛 𝑔(𝑛) = 𝑛𝑙𝑜𝑔 𝑛 h(𝑛) = 𝑛2**

**a. 𝑓(𝑛)** ∈ **θ(𝑔(𝑛))**

**metin, ekran görüntüsü, diyagram, öykü gelişim çizgisi; kumpas; grafiğini çıkarma içeren bir resim

Açıklama otomatik olarak oluşturuldu **

**c1​⋅g(n)≤f(n)≤c2​⋅g(n)**

the formula is from second Pdf in the lecture.

There is just a constant between g(n) and f(n).

el yazısı, yazı tipi, diyagram, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

So, We can say that **𝑓(𝑛)** ∈ **θ(𝑔(𝑛))** is true.

**b. 𝑓(𝑛)** ∈ **𝑂(h(𝑛))**

**çizgi, diyagram, öykü gelişim çizgisi; kumpas; grafiğini çıkarma, taslak içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**𝑓(𝑛) = 𝑛𝑙𝑜𝑔2𝑛** (linearithmic)

**h(𝑛) = 𝑛2** (quadratic)

h(n) grows more faster than f(n). ( 1 < n < logaritmic < linearithmic < quadratic < exponential < factorial )

So, We can say that **𝑓(𝑛)** ∈ **𝑂(h(𝑛))** is true.

**Q2)**

// Input: Array A of integers, integer k

// Output: Return the beginning index of the sequence. Return -1 if there is no such sequence in A such that their sum is equal to k.

**Algorithm** find (A, k)

x ← 0

for i ← 0 to n(A.length) - 1 do

sum ← 0

for j ← i to n(A.length) - 1 do

sum ← sum + A[j]

if sum = k then

return i

return -1

We have to look at every number in the array. So, ***θ(n2)* (worst-case) algorithm**

**Q3)**

// Our aim is to find the most frequent element

// input: an array of integers A[ ]

// output: return the most frequent element in the array

**Algorithm** mostFrequentElement(A)

maxFrequent ← 0

maxFrequentedNumber ← A[0]

for i ← 0 to n(A.length) – 1 do

currentFrequent ← 0

for j ← 0 to n(A.length) – 1 do

if A[i] = A[j] then

currentFrequent ← currentFrequent + 1

if currentFrequent > maxFrequent then

maxFrequent ← currentFrequent

maxFrequentedNumber ← A[i]

return maxFrequentedNumber

brute-force *θ(n2)* (worst-case) algorithm

there are 2 for loop. External loop runs n times and inner loop runs n times. So, complexity is *θ(n2).*

Best Case: We should look all numbers in array. So best case complexity is *θ(n2).*

Average Case: Generally operation is *n2*. So, avg complexity is *θ(n2).*

**Q4)**

// rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements.

// input: an array of integers A[ ]

// output: return rearranged A[ ]

**Algorithm** rearrangeArray(A)

B ← new array (B’s size is A.length)

j ← 0

for i ← 0 to n(A.length) - 1 do

if A[i] < 0 then

B[j] ← A[i]

j ← j + 1

for k ← 0 to n(A.length) - 1 do

if A[k] >= 0 then

B[j] ← A[k]

j ← j + 1

return B

My algorithm is working in linear time complexity. There are 2 for loops but compexity is θ(n) because loops are not nested.

It has to go every numbers in the arrays so best case and average case are also θ(n).

**Q5)**

//Computes the minimum distance between the closest pair of n points placed on the x-axis, with the divide-and-conquer technique.

//Input: An array representing n points on x-axis.

//Output: The distance between the closest point pair.

**Algorithm** findMinDistance(P[0..n-1])

If n <= 2 then

return 0

if n = 3 then

if P[0] = P[1] then

return 0

if P[0] < P[1] then

return (P[1] - P[0])

if P[0] > P[1] then

return (P[0] - P[1])

mid ← n / 2

leftMinDistance ← findMinDistance (P[0..mid - 1])

rightMinDistance ← findMinDistance (P[mid..n - 1])

if leftMinDistance < rightMinDistance then

exactlyMinDistance ← leftMinDistance

if leftMinDistance >= rightMinDistance then

exactlyMinDistance ← rightMinDistance

for i ← 0 to mid - 1 do

for j ← mid to n - 1 do

if (P[j] - P[i]) < exactlyMinDistance then

exactlyMinDistance ← P[j] - P[i]

return exactlyMinDistance

Analysis: sorting process is O(nlogn). Here divide and conquer process is wanted. Then I divided 2 parts the array and I run both 2 arrays with recursively. All dividig process runs O(n) and recursive is logn.

Finally I check the array from 0 to mid-1 and from mid to n-1. This process is O(n).

taslak, çizgi, çizim, diyagram içeren bir resim

Açıklama otomatik olarak oluşturuldu

As a result of this analysis, the total time complexity of the algorithm is O(nlogn).

This question was the hardest one. But I focused a lot, I hope my all answers are correct.

**Q6)**

1. **What does this algorithm compute?**

The algorithm checks whether n is even, odd or zero. If n is odd or zero, function returns 0. Also if n is even this function computes n2 + (n−2)2 + (n−4)2 + ... + 4 + 0 and returns result.

1. **Which strategy does it follow to solve the problem?**

This algorithm uses an recursive strategy to solve the problem. At each step, the function sums the square of n and returns f(n - 2). when n = 0, func returns 0. Finaly, func computes n2 + (n−2)2 + (n−4)2 + ... + 4 + 0 and returns result.

**c) What is the time-complexity of this algorithm? What is its basic operation? Comparison, Multiplication, Addition? Does your complexity analysis depend on which basic operation you choose? Please explain and validate your answer**

Each step algorithm cals f(n - 2) so function is completed approximately n/2. So complexity of algorithm is O(n/2). To be honest constant values are not important for complexity, So we can say that complexity of algorithm is O(n).

Basic operations are comparison, multiplication and addition.

Comparison: n is even or odd

Multiplication: n\*n

Addition: n2 + (n−2)2 + (n−4)2 + ... + 4 + 0

The time complexity 𝑂(𝑛) of this algorithm remains the same regardless of the chosen basic operation. Each recursive call involves a constant amount of work (one multiplication and one addition), with approximately 𝑛 / 2 calls in total. Thus, the growth rate stays linear as 𝑛 increases, making 𝑂(𝑛) the final complexity independent of the basic operation chosen.

**İhsan Eren Erben ----- 22SOFT1055**