1. Let *f*(*n*) and *g*(*n*) asymptotically positive functions. State whether the following statements are TRUE or FALSE. In case a statement is TRUE, prove it using definitions of *O,* Ω*,* Θ*, ω, o* etc. In case it is FALSE, provide a counter example.
2. *f*(*n*) = *O*( ( *f*(*n*) )2 ) **FALSE**

**Explanation:** from big-oh notation definition: f(n) <=c\*(f(n))2

This holds true in one case i..e

n<=c\*n2

n = O(n2)

But this holds false for an example below:

Let assume f(n) = 1/n

g(n) = (f(n))2 = 1/n2

g(n)/f(n) = (1/n)/(1/n2)

= = ∞ > 0

**Hence, big-oh is not proved, the assumption we made is wrong.**

1. *f*(*n*) = Θ( *f*(*n/*2) ). **FALSE**

**Explanation**: In order to prove f(n) = Θ( f(n/2) ), we need to prove >0 and >0

Let, f(n) = 2n

f(n/2) = 2n/2

g(n)/f(n) = 2n/2 / 2n = 2-n/2 = 1/2n/2

f(n)/g(n) = 2n/2n/2  = 2n/2

now, ∞ = 0

= 2n/2 = ∞ >0

**Hence, big-oh is not proved, the assumption we made is wrong.**

1. *f*(*n*) = Ω(√*f*(*n*)). **FALSE**

**Explanation:**

From Big-omega definition: f(n) > = c\*g(n)

= c >0

Let us assume that f(n) = 1/n

So, √f(n) = 1/ √n

for Big-Omega :

= (1/n) / (1/√n)

On solving this:

= 1/√n

= 1/∞

= 0

**Hence the assumption we made is wrong**

1. max( *f*(*n*)*, g*(*n*) ) = Θ( *f*(*n*) + *g*(*n*) ). **TRUE**

**Explanation:**

Given, T(n) = max{ f(n) + g(n)},

T(n) <= f(n) + g(n)

Therefore, T(n) = O(f(n) + g(n))

Now,

As T(n) >= f(n) and T(n) >= g(n)

T(n) + T(n) >= f(n) + g(n)

2T(n) >= f(n) + g(n)

T(n) >= ½ [f(n) + g(n)]

Therefore, T(n) = ω(f(n)+g(n))

So, T(n) = Θ { f(n) + g(n)}

1. Consider the general *k-ary* search algorithm (generalization of binary search) which splits a sorted array into *k* subsets each of approximately size *n/k*, and by making (*k*−1) comparisons, recursively searching in one of these *k* subsets. Clearly, the recurrence relation can be written as,

*T*(*n*) = *T*(*n/k*) + (*k* − 1) (1)

Here *k* is a variable, but it can be a function of *n* as well.

* 1. Suppose we set *k* = √*n* in equation 1. Find asymptotic running of *T*(*n*) in this case.

**Solution:** Substituting *k* = √*n*

*T*(*n*) = *T*(*n/* √*n)* + (√*n* − 1)

= *T*(√*n* )+ (√*n* − 1) 1

Applying replacing variables method:

Let, n = 2m

It can be written as*: √n* = 2m/2 2

On Substituting equation 2 in 1:

= *T*(2m/2 )+ (2m/2 − 1)

Rewriting the equation:

T(n) = S(m) = T(2m)

S(m) = s(m/2) + (m/2 – 1) 3

Applying masters method for the equation 3:

T(n) = a T(n/b) + f(n)

a = 1, b = 2 ,

f(m) = m/2 -1 4

*n*log*b*​*a = n*log*21 = 0*

on using case3: masters theorem:

f (n) = Ω(n logba + ɛ ) for some constant ɛ > 0. Alternatively: f(n) / nlogba = Ω(n ɛ ) Intuition: f (n) grows polynomially faster than n logba Or: f(n) dominates n logba by an n ɛ factor for some v > 0

f(m) = (2m/2 – 1)

= Ω(m 0+ɛ )

Checking the Regularity Condition:

since it is case-3, af(m/b) ≤ cf(n)

1\* 2(m/2)/2 ≤ c. 2 (m/2)

Therefore, T(n) = Θ(f(m))

S(m) = Θ(2 m/2)

**T(n) = Θ(√n)**

* 1. Now suppose we set *k* = log *n* (where the base of the logarithm is 2) in equation (1). The above recurrence relation becomes,

*T*(*n*) = *T*(*n/*log *n*) + (log *n* − 1) (2)

Explain why equation 2 cannot be solved using Master theorem.

**Solution**:

To apply masters theorem the above equation should be in the form of

T(n) = a T(n/b) + f(n)

On applying replacing variables method :

n = 2m and log n = m

*T*(*n*) = *T*(*n/*log *n*) + (log *n* − 1)

*T*(*n*) = *T*(*2m/*m) + (m− 1) 1

Even after applying replacing variables method , the equation 1 cannot be in the form where b>1 and not a constant for T(n) = a T(n/b) + f(n).Therefore, the given equation cannot be solved using Masters theorem.

* 1. One way to solve equation 2 will be to find explicit upper bound (*O*) and lower bound (Ω) for *T*(*n*). Let us concentrate on the upper bound first. From the divide and conquer point of view, the recurrence relation *T*(*n*) = *T*( *n/*log *n* ) + (log *n* − 1) says that a problem of size *n* is divided into one sub-problem of size *n/*log *n* and “divide plus combine” step takes (log*n*−1) time. To find an upper bound of *T*(*n*) consider a separate divide and conquer algorithm which divides a problem of size *n* into one sub-problem of size *n/*2 and where the “divide plus combine” step takes log *n* time. Let the running time of this algorithm be *T*1(*n*). Since subproblem size *n/*2 is bigger than *n/*log *n* for all *n* ≥ 4*, T*(*n*) ≤ *T*1(*n*), and therefore, *T*(*n*) = *O*(*T*1(*n*)). Thus, solving *T*1(*n*) will give an upper bound of *T*(*n*). Write down the recurrence relation of *T*1(*n*) and solve for *T*1(*n*). Assume *T*1(1) = 1. Show your steps.

**Solution:**

Given, T(n) <= T1(n) for n>=4

Therefore, T(n) = O ( T1(n))

T1(n) = T1(n/2) + log(n)

Applying masters theorem:

T(n) = a T(n/b) + f(n)

a = 1, b = 2, f(n) = log (n)

*n*log*b*​*a = n*log*21 = 0*

however, f(n) ≠ Ω(n0-ɛ ) . so, we cannot apply Masters Theorem.

But we can solve using variables replacing method:

Let n = 2m , m = log(n)

T1(2m) = T1(2m/2) + m

Rewriting it:

S1(m) = T1(2m)

S1(m) = S1(m-1) + m

S1(m) = m + S1(m-1)

= m + (m-1) + S1(m-2)

= m + (m-1) + (m-2) + S1(m-3)

.

.

.

.

.

= (m(m+1))/2

Therefore, S1(m) = Θ(m2)

T1(2m) = Θ(m2)

T1(n) = T1(2m) = Θ(m2) = Θ(log2 n)

**T1 = Θ(log2 n)**

* 1. To find a lower bound of *T*(*n*) consider yet another separate divide and conquer algorithm which divides a problem of size *n* into one sub-problem of size *n/*√ *n* = √*n* and where the “divide plus combine” step takes log *n*−1 time. Let the running time of this algorithm be *T*2(*n*). Clearly, *T*(*n*) ≥ *T*2(*n*) and therefore, *T*(*n*) = Ω(*T*2(*n*)). Thus, solving *T*2(*n*) will give a lower bound of *T*(*n*). Write down the recurrence relation of *T*2(*n*) and solve for *T*2(*n*). Show your steps.

**Solution:**

Given, T(n) >= T2(n)

Therefore,by Big-Omega definition: T(n) = Θ ( T2(n))

On using changing variables method,

T2(n) = T2(√n) + log(n) – 1 1

Let n = 2t

√n = 2t/2

Log(n) = t

From above equation 1

T2(2t) = T2(2t/2) + t-1

Let S(t) = T2(2t)

So, S(t) = S(t/2) + t-1

Applying masters theorem:

Let a =1, b= 2, f(t) = t-1

t logab = t0 = 1

f(t) = t-1 = Ω(tlogab+ℇ), where ℇ>0

now, by using case3 regularity condition of masters theorem

s(t) = Θ(t)

S(t) = T2(2t) = Θ(t)

T2(n) = S(t) = T2(2t) = Θ(t)

Where t = log(n)

**T2(n) = Θ(log(n)) i.e logn ≤ T(n) ≤ log2 n**

1. We have seen that algorithm for finding strongly connected components of a directed graph *G* = (*V,E*) works as follows. In the first step, compute DFS on the reverse graph *GR* and compute post numbers, then run the undirected connected component algorithm on *G*, and during DFS and then process the vertices in decreasing order of their post number from step 1. Now professor Smart Joe claims that the algorithm for strongly connected component would be simpler if it runs the undirected connected component algorithm on *GR* (instead of *G*), but during DFS, process the vertices in increasing order of their post number from step 1.
2. Explain when the algorithm proposed by prof. Smart Joe might produce an incorrect answer.
3. With an example, (along with post numbers) show that professor Smart Joe is wrong.

**Submission:**

* All texts and diagrams must be electronically produced.
* Your name should appear on each page.
* Entire submission should be a single PDF file.
* The PDF file should be named in the format Exam01\_Lastname\_Firstname.pdf, for example Exam01\_Bose\_Sourabh.pdf.
* Submit the pdf file on blackboard.