1.

Ans:

Scratch:

function F(n) begin

array A[1:n, 1:n]

for i := 1 to n do |

A[i, i] := 0 | 🡺 E

for l := 2 to n do | \_|

for i := 1 to n − l + 1 do begin \_| |

j := i + l − 1 | |

A[i, j] := ∞ | |

for k := i to j − 1 do | | |

A[i, j] := min{ A[i, j], A[i, k] + A[k + 1, j] + ijk} 🡺 A | 🡺 B | 🡺C | 🡺D

end | | |

return A[1, n] \_| \_|

end

Breaking up the code down.

2.

A.

B.

C.

3.

A.

Ans:

ln(ln(n)) < n < nln(n) < n^2 < n!

2^n < 4^n

Sqrt(ln(n)) < ln(n!) < └ln(n)┘!

2^ln(n) < ln(n)^ln(n)

Still need: └ln(n)┘!, (ln(n))^2, ln(n!), n^ln(ln(n)), ,2^2^n, 2^ln(n), ln(n)^ln(n), sqrt(ln(n)) ┌┐

Scratch:

2^2^n != 4^n but it might work in terms of rate of growth.

\*\*\*B. f(n) + g(n) = Ꝋ(max{f(n),g(n)})

Argument: Let us take a function, call it Y(n) = f(n) + g(n). And the definition of x(n) = Ꝋ(h(n)). This definition stays true if f(n) = O(g(n)) and g(n) = o(f(n)) because they share the same rate of growth. Conceptually, if we have big values of n, for the rate of growth to be the same, then we need to take the maximum values of f(n) + g(n). because in large values of n. The bigger rate of growth between f(n) and g(n) will overtake the other function. For example, take Y(n) = n^2 + n. The longer we go through n, the bigger the difference between n^2 and n, to the point where n doesn’t add much to it compared to n^2. This is true for even smaller differences like Y(n) = n^1.00001 + n.

This is why, f(n) + g(n) = Ꝋ(max{f(n),g(n)}) because we need to take into account the rate of the growth of the bigger function.

C.

4.

A. T(n) = 4T(n/2) + n^2

ans: Ꝋ (n^(5/2))

Masters Theorem: T(n) = aT(n/b) + f(n)

Rules

1. f(n) = 0(n^) then T(n) = Ꝋ (n^)

2. f(n) = Ꝋ ( then T(n) = Ꝋ(log(n) \* n^)

3. f(n) = Ὠ(n^) then T(n) = Ꝋ(f(n))

Scratch:

We can use the Masters Theorem where, A = 4, B = 2, and f(n) = n^2 = n^(2.5)

Applying masters theorem, and compare it to f(n). we get = 2 and compare it to f(n) = n^2.5

We compare this with n^< n^2.5 where e is a constant. So, we follow

we compare 2 to 2.5 since 2 < 2.5.

We get Ꝋ (n^(5/2)) because of rule 3.

B. T(n) = 32T(n/4) + n^2

Ans: Ꝋ(n^(5/2)\*log(n))

Scratch:

Again, using masters theorem.

A = 32, B = 4 and F(n) = n^(2.5)

So we can find . We can do this by 4^X = 32. 4^x = 4^2\*4^(1/2) or 4^x = 4^(2+1/2) = 4^(2.5).

So we get

So we get n^(2.5) and we compare it to f(n) or n^(2.5), so n^ = f(n). So we use rule 2.

Using masters theorem, we get rule 2. So the runtime is Ꝋ(n^(5/2)\*log(n))

C.T(n) = 3T(n/2) + nlogn

Ans: Ꝋ(n^1.585)

Scratch:

Using masters theorem. F(n) = nlog(n) and A = 3, B = 2

We need to find = 1.5849625007

n^1.5849625007-e = O(nlogn) because we can take any constant for e. e.g. e = .0001

so we follow rule 1.

And we get Ꝋ(n^1.585)

\*\*\*D. T(n) = 3T(n/3) + nlgn

Ans: Ꝋ(nlog^2(n))

Scratch:

Using masters theorem:

A = 3, B = 3 and f(n) = nlgn

We get

However,

Adding e to n^1 we get something that is neither over, nor under bounded with f(n).

So unfortunately, we cannot use the 3 cases. There is another use case that can be used.

Where f(n) is not a polynomial and A == B. Then we can get the answer Ꝋ((n^ if we can show that f(n) E Ꝋ((n^ for some k >= 0. In our case k = 1 because f(n) E Ꝋ(nlogn). Therefore, by this condition. We get Ꝋ(nlog^2(n))

Let us use

\*\*\*

E. T(n) = T(

\*\*\*https://www.cs.rhodes.edu/welshc/COMP355\_F17/Lecture5.pdf

5.