\*\*1.

Ans: Θ(n^3)

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.

A == runs a constant of Θ(1) because it just compares 2 numbers.

B == is a for loop that runs at a runtime of

C == is a for loop that runs at a runtime of

And

D == is a for loop that runs at a runtime of

And E has a runtime of Θ(n) because it is a for loop that runs n times.

Our total runtime would be Θ(n) + Θ(D) because Θ(D) encapsulates the runtime of A, B and C.

So to find the runtime of D. Let’s examine the summations that define D.

For B we see that the constant runtime is ran a total of j-1-I times. Where j grows at a rate of i+l-1, so it grows at rate of I plus a rate of l. I grows at a rate of n-l+1, which changes with the summation of c. I is dependent on n and l where l is dependent on n. so I grows as a result of Θ(n)- Θ(n) \*\*\*which results in Θ(n). since j grows as a result of I with a rate of growth of n-l+1 and j is a result of i+l-1 we can say that j is about n-l+l and conclude that j grows as a rate of Θ(n). So

B grows as a rate of Θ(n).

So now we find the rate of growth of I, since it starts at 1 and goes up to n-l+1. We need to see how much l grows. L grows from 2 to n, so it grows at a rate of growth of Θ(n). \*\*\*we get n- Θ(n)+1 which is a rate of growth Θ(n).

So C grows at a rate of Θ(n)\*B or Θ(n)\* Θ(n) which is a rate of Θ(n^2)

And part D grows at a rate of Θ(n) because it’s a sum of l = 2 to n. so the amount of time’s it runs is based on n. so we get Θ(n)\*C or about Θ(n)\* Θ(n^2) which results in an answer of.

Θ(n^3)

2.

A. To show that a lists of size k, made up of┌(n/k)┐ lists can be sorted in worst case Θ(kn) time, we need to find the runtime of the worst case insertion sort for each list.

The worst case insertion sort for a list of size n is Θ(n^2). So, it stands to reason that the worst case for each list is Θ(k^2). However, we also need to take into account that this happens for all (n/k) lists. So the runtime is really Θ(k^2) \* ┌(n/k)┐. \*\*\* this means that we cancel out one of the k’s and multiply it with n. So we end up with a theta bound Θ(kn).

B.

This can be merged into one sorted list in Θ(nlog(n/k)) worst case time. Because we have n/k lists, and they are all sorted. Just like merge sort, we will sort two lists at a time until we get ┌(n/k)┐/2 lists, then we merge two again until we have ┌(n/k)┐/4 lists, then ┌(n/k)┐/8 lists. And so on. We keep doing this until we get a sorted sub lists of size n. This makes sense since we are merging two lists at a time. This is a logarithmic base 2 of ┌(n/k)┐ lists. This is the same as the amount of time we merge all sub lists in a set in merge sort. This is just the merges though. Every merge requires n comparisons. Which is why the runtime Is Θ(nlog(n/k)).

Now I will argue why we need n comparisons for every time we merge all sorted sub lists. Take two sub lists of size (k)

A = [A1,A2,A3,…,Ak] B = [B1,B2,B3,…,Bk]. We will measure the number of comparisons for these two lists.

We compare A1 to B1, take the smaller, B1, then A1 to B2 take the smaller, A1, then take compare A1 to B2, take the smaller, A2, then compare … and we do this until we go through the whole list for both A and B. We would need 2\*n comparisons. Which is a runtime of Θ(k). This is the case because we compare one value of A and one value of B and take the smallest. Then we ignore that one for the rest of the comparisons. So, for each comparison we place an element in B or A in a new sorted list and ignore it for the rest of the merge. That means 2\*k elements results in 2\*k comparisons.

Now remember that there are n/k lists, each with a size of k elements. With a total of n elements. Each two lists are comparing 2\*k elements, resulting in a total amount of n comparisons for every time we merge all pairs of sub lists together.

And like we found before; we merge all sub lists together log(n/k) times.

Resulting in our runtime.

C. Θ(nk + n log(n/k))

We need to find k as a function of n that is at worst Θ(n log(n)) because that is the order of growth for merge sort. To find the fastest rate of growth for k for which this holds, we need to come up with functions where if k = Θ(f(n)) the hybrid algorithm runs in O(nlogn) time and showing that if k = w(f(n)), the hybrid algorithm runs in w(nlogn) time.

Let us take the fact that Θ(nk + n log(n/k)) must be at worst Θ(nlog(n)). So nk = Θ(nlogn) or nlog(n/k) = Θ(nlogn). The largest possibility for k is Θ(logn). Because if k was bigger than that, than nk would be bigger than Θ(nlogn). And nlog(n/k) = nlog(n/logn) = nlog(n)-nlog(log(n)) which is Θ(nlogn). Because nlog(log(n)) grows slower than nlog(n) and it is also subtracting it from the runtime. Θ(nlogn) + Θ(nlogn) results in a runtime of Θ(nlogn). Which is just as good as merge sort.

3.

\*A.

Ans: ln(ln(n)) < sqrt(ln(n)) < ln^2(n) < 2^ln(n) <= n < Ln(n!) <= nln(n) < n^2 < ln(n)! < ln(n)^(ln(n)) <= n^ln(ln(n)) < 2^n < 4^n < n! < 2^2^n

n < nln(n) <

\*<= 🡺 equal to

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:

Some identities:

N^log(log(n)) = (log(n))^log(n)

n^2 = 4^log(n)

n = 2^log(n)

2^sqrt(2log(n)) = n^sqrt(2/log(n))

1 = n^(1/log(n))

log\*(log(n)) = log\*(n) – 1 for n > 1

Asymptotic bounds for Stirling’s formula are

n! = Θ(n^n+1/2 e ^−n )

log(n!) = Θ(nlog(n))

log(n)! = Θ((log(n))^(log(n) + 1/2)\*e^(-log(n)))

\*\*\*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. So, this might be a bad assumption but I will assume that A and B are both greater than 1.

I believe the best function is asymptotically greater than n^a but asymptotically smaller than b^n. would be f(n) = c^lg(n) where c is any constant greater than 1.

I believe this would be the case because we know that at any value a, as n approaches infinity, it will always end up being greater than a. And the constant C will eventually grow at a faster rate than n^a. We can also say that n^a if a is a constant will be slower than a constant to the power of an increasing number.

So, we meet the first requirement, that it is asymptotically bigger than n^a. We need to prove that it is asymptotically smaller than b^n. Well, this is simple because they both are similar. Take C^log(n) and b^n. They are both constants to the power of an increasing exponent. So it stands to reason that the faster growing exponent would result in the faster growing function overall.

So, we compare log(n) and n. And it is very clear that the faster growing algorithm between these two is n.

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).

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( runtime is more like n ^ (1/2) because that is what it is. It goes down like, 2^2^2^2^2^2^2^2 would result in about 9 runs.

Ans: Ꝋ(log\*(n))

Scratch: the rate of growth lg\*(n). meaning that lg\*(4) = 65536 or 2^2^2^2

We do this until we get down to two, because when we take the square root of 2, we get 1. Because we take the floor of the number.

This is because we split n in half until we get less than 2. So, we have lg\*(n) actions.

T(n) split into t(sqrt(n)) split into t(sqrt(sqrt(n)) etc. So we get k to be lg\*(k) = n

lets work backwards, starting from 2. because when we square root 2, we get 1. because n must be an integer. So, if we run once, we get 2^2, then we get twice we do 2^2^2, then 3 times we get 2^2^2^2. Just like how lg\*(n) grows.

For an example, take n to be 2^2^2^2. This runtime will remain true because we take square Root(n) = 2^2^2 which is one action. then again we get 2^2, that is two actions. Then again, we get 2 and that is 3 actions, then one last time we get sqrt(2) = 1. That is 4 actions. So k is lg\*(N). We end up getting a runtime of Ꝋ (lg\*(n)).

5. We can find the median element of an n element set in Ꝋ(n) time by using a pivot and moving all the elements lower than the median to the left, and the ones higher than the median to the upper. Because we are examining all the elements in the list, we need to do at least n-1 comparisons making this Ꝋ(n) time. To do this with the kth smallest element, we use the same strategy. The difference is that we take the number that place n-k elements to the right of the list or has n-k elements bigger than that member. This is if we can get the right pivot though.

So, we need to find the best way to find the pivot.

\*\*Finding the pivot:

I believe that the best way to pick a pivot, would be to choose a few elements of the array randomly and pick the number that would most likely be the k-th smallest. This is also known as the rule of 3s. So, let us say that k is less than n/3, then we would use the smallest of the three numbers we are comparing, and try that. If k is between n/3 and 2n/3 then we would use the middle element, and the last one for if k Is greater than 2n/3.