Sarah Pham (slp0042)

Homework 2 - Intro to Algorithms

Notes from class:

Θ = Is in-between the two ( = )

T(n) = O(g(n)) = T(n) is less than or equal to cn(0) ( < )

T(n) = o(g(n)) = T(n) is less than cg(n) for all positive constants, cg(n) grows way faster than t(n)

Omega = If you can find there exists a constant c and large value of n, if you can show that T(n) grows faster than cg(n), you can claim it is bounded from below by g(n) (>)

T(n) = w(g(n)) = T(n) is bounded from below by a quadratic

* Polynomial is less efficient than logarithm, poly grows faster than polylog
* Exponential is less efficient, logarithm is much faster
* Factorial (n!) is more efficient than exponential
* Log < nb < cn < n! < nn

**1a. f(n) = 100n + logn and g(n) = n + (logn)2**

If you look at the graph for these two equations, it looks something like this, f(n) and g(n) look very similar:   
Chart, line chart

Description automatically generated

One way to find out whether a function is smaller than the other is through division and finding the limit. If the answer is more towards infinity, then the numerator grows faster, if its closer to 0 then the denominator bounds the numerator.

= = = = = =

= = 100

Since the limit of f(n)/g(n) is equal to a constant 100, that means that f(n) = Θ(g(n)).

**1b. f(n) = logn and g(n) = log(n2)**

The graph is below.

Chart

Description automatically generated with low confidence   
We also know that log(n2) = 2log(n) from Logarithm rules, so therefore the only difference between log(n) and 2log(n) is the constant 2, so therefore **f(n) = Θ(g(n)).**

**1c. f(n) = n2/logn and g(n) = n(logn)2**

The graph is below, and we can see that f(n) is less efficient.

Graphical user interface, application

Description automatically generated

So let’s do the same strategy as the first problem,  
 = = =

= = = = =

Since the limit of f(n)/g(n) is equal to infinity, that means that f(n) grows faster and therefore  
**f(n) = Ω(g(n))**

**1d. f(n) = n(1/2) and g(n) = (logn)5**

From the graph below, it looks f(n) is growing faster than g(n) until n=~20

Chart

Description automatically generated with medium confidence

We also know that exponentials are not as efficient as logarithms, so therefore **f(n) = Ω(g(n)).**

**1e. f(n) = n2nand g(n) = 3n**

The graph below shows that g(n) grows faster than f(n):

Chart, line chart

Description automatically generated

If we find the limit of f(n)/g(n), it will eventually go to 0, which means that the denominator (g(n)) grows faster. So therefore, **f(n) = O(g(n))**.

**2a.** Let’s look at the method step by step.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Code | Meaning | Cost |
| 1 | If i=j then return A[i] | If the array has 1 element left, then return the array | 7 |
| 2 | else |  | 0 |
| 3 | K = i + floor((j-i)/2) | This step calculates the middle indices for the array | 8 |
| 4 | Temp1 = Mystery(A[i…k]) | Temp1 calls the recursive function to only look at the left half of the given array | 5 |
| 5 | Temp2 = Mystery(A[(k+1)...j]) | Temp2 calls the recursive function only look at the right half of the given array | 5 |
| 6 | If temp1<temp2 then return temp1 else return temp2 | Return the smallest integer | 7 |
| 7 | end |  | 0 |
| **Total** |  |  |  |

To summarize, this recursive algorithm returns the smallest integer in the array.

**2b.**

Since this is a divide-and-conquer algorithm we know that the recurrence case is:

Text

Description automatically generated with medium confidence

T(n) = 7 = Θ(1) when n = 1

* The constant cost would be 7 because looking at the first step we would need to:
  1. Get i
  2. Get j
  3. Compare i = j
  4. Get i
  5. Identify memory location A[i]
  6. Grab data at A[i]
  7. Return A[i]

and T(n) = 2T(n/2) + Θ(n) when n > 1

* Suppose Θ(n) = 18
* T(n) = 2(Tn/2) + 18

**2c.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Level | Level # | Total # of Recursive Executions at this level | Input size to each recursive execution | Work done by each recursive execution, excluding the recursive calls | Total work done by the algorithm at this level |
| Root | 0 | 1 = 2i  = 20 | n = n/2i | 18 | c \* 2i = 18 \* 1 = 18 |
| One level below root | 1 | 2 | n/2 | 18 | 18\*2 |
| Two levels below root | 2 | 4 | n/4 | 18 | 18\*4 |
| The level just above the base case level | Log2(n-1) | 2log2(n-1) | n/2log2(n-1) | 18 | 18\*2log2(n-1) |
| Base case level | Log2n | 2log2n | n/2log2n | 7 | 7\*2log2n |

**2d.**

Order of complexity is:

T(n) = 7(2log2n) + 18\*i

We also know that 2log2n is n, since n is the total number of recursions and from class we know that:

A picture containing shape

Description automatically generated

So we can rewrite/simplify our complexity to be:

T(n) = 7n + 18[ (2logn-1)/(2-1) ]

T(n) = 7n + 18\* (2logn-1) We know 2logn = n

T(n) = 7n + 18 \* (n - 1)

T(n) = 7n + 18n - 18 = **25n -18**

**3.** T(n) = 7T(n/8) + cn; T(1) = c

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Level | Level # | Total # of Recursive Executions at this level | Input size to each recursive execution | Work done by each recursive execution, excluding the recursive calls | Total work done by the algorithm at this level |
| Root | 0 | 1 = 7i  = 70 | n/80 = n | cn | cn\*(70) = cn |
| One level below root | 1 | 71 = 7 | n/81 = n/8 | c\*(n/8) | c\*(n/8)\*(7) = 7c\*(n/8) =  (7/8)cn |
| Two levels below root | 2 | 72 = 49 | n/82 = n/64 | c\*(n/64) | c\*(n/64)\*(49)=  (7/8)2cn |
| The level just above the base case level | Log8(n-1) | 7n-1 = 7log8(n-1) | n/8n-1 = n/8log8(n-1) | c\*(n/8log8(n-1)) | c\*(n/8log8(n-1))\*( 7log8(n-1)) =  (7/8) log8(n-1)cn |
| Base case level | Log8n | 7n = 7log8n | n/8n = n/8log8n = n/nlog88 = 1 | c\*(1) = c | c\*(1) \* (7log8n) = c\*nlog87 |

T(n) = c\*nlog87 + i\*cn The bounds for the sum are from [0, log8(n-1)]

T(n) < c\*nlog87 + i\*cn We set bounds to infinity to use geometric series

T(n) = c\*nlog87 + cn[ 1/(1-(7/8)) ]

**T(n) = c\*nlog87 + 8cn**

**4.** T(n) = 3T(n/3) + 5; T(1) = 5

Statement of what you must prove:

T(n) = O(n) where T(n) ≤ cn - 5

We guessed O(n) because the given equation does not end with “+n” which would be O(nlogn), so we know that it’s smaller than O(nlogn)

Base Case Proof:

T(1) = 5 ≤ (c - 5) holds when c is at least 10

Inductive Hypothesis:

We substitute T(n/3) ≤ cn/3 - 5

Inductive Step:

T(n) ≤ 3\* ((cn/3) - 5) + 5

T(n) ≤ cn - 15 + 5

T(n) ≤ cn -10 <- This is true, so therefore the proof holds

Value of c:

c ≥ 10

**5.** f(n) = O(s(n)) means that f(n) is dominated by some scaled version of s(n) (i.e. a\*s(n)) and vice versa with g(n) = O(r(n)). In other words, there exists constants a,b >= 0 such that f(n) ≤ a \*s(n) for all n >= b.

With the above definition in mind, we can create some arbitrary value for s(n) and r(n),

Let’s say that s(n) = n + 1 and r(n) = n,

Therefore,

f(n) = O(s(n)) = O(n+1) = O(n) + O(1) = n

g(n) = O(r(n)) = O(n) = n

Plugging in our values,

f(n) - g(n) = O(s(n) - r(n))

n - n = O(n+1-n)

0 = O(1)

0 ≠ 1

Therefore, f(n) - g(n) ≠ O(s(n) - r(n))

**6.** T(n) = T(n/2) + T(n/4) + T(n/8) + T(n/8) + n; T(1) = c

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Level | Level # | Total # of Recursive Executions at this level | Input size to each recursive execution | Work done by each recursive execution, excluding the recursive calls | Total work done by the algorithm at this level |
| Root | 0 | 1 | 1 | (refer to Recursion tree) | n |
| One level below root | 1 | 4 | 4 |  | n |
| Two levels below root | 2 | 5 | 16 |  | n |

Diagram

Description automatically generated

**6.3.** Shallowest part = log8n because n/8 is quite small versus n/2

**6.4.** Depth = log2n because n/2 is bigger than n/8

**6.5.** O(nlogn) because we know that any equation that ends with “+n” is nlogn.

**7.** Statement of what you have to prove:

T(n) = O(nlogn) where T(n) ≤ cn\*logn

Base Case Proof:

T(1) = c ≤ (c\*log(1)) Log(1) = 0 because of log rules

T(1) = c ! ≤ 0 0 is not greater than c, but we still need to prove the base case, so lets increment

T(2) = c + 2 ≤ (c(2)\*log(2)) Substitute 2 for n

T(2) = 2 ≤ 2c\*log2 - c We want to isolate c to one side to find the value

T(2) = 2 ≤ c(2log2-1) Factor out c

T(2) = 2/(2log2-1) ≤ c Isolated c

This is true; therefore, the base case holds.

Inductive Hypothesis:

We substitute T(n/2) ≤ cn/2 \* log(n/2) as well as the other inputs:

T(n/4) ≤ cn/4 \* log(n/4)

T(n/8) ≤ cn/8 \* log(n/8)

Inductive Step:

T(n) ≤ cn/2 \* log(n/2) + cn/4 \* log(n/4) + cn/8 \* log(n/8) + n (Plug in inductive hypothesis)

T(n) ≤ c[(n/2) log(n/2) + (n/4) log(n/4) + (n/8) log(n/8)] + n (Factor out c)

T(n) ≤ c[(n/2)(logn - log2) + (n/4)(logn - log4) + (n/8)(logn - log8)] + n (Log rules, division=minus)

T(n) ≤ c[(nlogn/2 - nlog2/2) + (nlogn/4 - nlog4/4) + (nlogn/8 - nlog8/8)] + n (Distribute)

This is true and therefore the inductive step holds.

**8a.**

Notes:

* If nlogba grows faster than f(n) by a rate of a polynomial, T(n) = Θ(nlogba)
* If f(n) is identical to Θ(nlogba) then you can claim that T(n) = Θ(nlogba \*logn)
* If f(n) grows faster, then you can indicate that T(n) = Θ(f(n))

T(n) = 2T(99n/100) + 100n

We know that; f(n) = 100n, a = 2, b = 100/99

nlog100/992 = n68.96 We want to compare this value with f(n), is f(n) bounded?

Case 1 -> n69 grows faster than 100n so we use case 1

f(n) = 100n = O(n69-ε) where ε > 1

Let’s say that ε = 68, therefore

f(n) = O(n69-68) = O(n1) = O(n), therefore f(n) = Θ( nlog100/992)

**8b.** T(n) = 16T(n/2) + n3lgn

f(n) = n3lgn, a = 16, b = 2

nlog216 = n4

Case 1 -> Graphically, n4 grows faster than n3lgn

f(n) = n3lgn = O(n4-ε) where ε > 0

f(n) = O(n4-0.5) = O(n3.5) Plugging in ε = 0.5, we can see here that f(n) is bounded by n3.5.

The reason the master method does not omit this type of problem is because we have shown that O(n3.5) is polynomially larger by n0.5/logn

Therefore, f(n) = Θ(n4)

**8c.** T(n) = 16T(n/4) + n2

f(n) = n2, a = 16, b = 4

nlog416 = n2

f(n) = n2 = nlog416

Case 2 -> Since f(n) is the same as nlog416, therefore T(n) = Θ( nlog416 logn)

**9.** T(n) = 2T(n-1) + 1; T(0) = 1

Notes from class:

* Backwards Substitution: Expand right hand side, express T(n) in closed-form formula
* Forward Sub: Start with T(base case), calculate non-base case until pattern is found

**Backwards:**

**1.** We will show 3 expansions to find a pattern

T(n) = 2T(n-1) + 1 We want to find T(n-1), so let’s find it and substitute

T(n-1) = 2T(n-2) + 1 We have found T(n-1), but we also need T(n-2), let’s find that

T(n-2) = 2T(n-3) + 1 We found T(n-2), but we also need T(n-3)

T(n-3) = 2T(n-4) + 1 We found T(n-3), now to plug in all the values

T(n) = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 = 4T(n-2) + 3 Plugging in T(n-1)

* 4T(n-2) + 3 =
* 22T(n-2) + 21 + 20 T(n) Can also be rewritten as

T(n) = 4(2T(n-3) + 1) + 3 = 8T(n-3) + 7 Plugging in T(n-2)

* 8T(n-3) + 7 =
* 23 T(n-3) + 22 + 21 + 20  T(n) Can also be rewritten as

T(n) = 8(2T(n-4) + 1) + 7 = 16T(n-4) + 15 Plugging in T(n-3)

* 16T(n-4) + 15 =
* 24T(n-4) + 23 + 22 + 21 + 20 T(n) Can also be rewritten as

We see a pattern here, which will continue until we reach the base case.

**2.** Writing out T(n) and simplifying the equation using page 1147Diagram

Description automatically generated with low confidence

T(n) = 2n \* T(n-n) + 2(n-1) + 2(n-2) + … + 20 This is our pattern, let’s simplify

T(n) = 2n\*T(0) + 2(n-1) + 2(n-2) + … + 20 We know that T(0) = 1 based on given relation

T(n) = 2n(1) + 2(n-1) + 2(n-2) + … + 20 Page 1147 shows Geometric Series

T(n) = 2n+1 - 1/ 2 - 1 Convert to Geometric Series

T(n) = 2n+1 - 1/1 = 2n+1-1 Simplify

**3.** Now we substitute in LHS and RHS to show LHS=RHS

T(n) = LHS = 2n+1-1

T(n) = RHS = 2T(n-1) + 1 Let’s plug-in LHS into RHS to see if we get the same results

RHS = 2T(n-1) + 1 Plugging in LHS into T(n-1)

RHS = 2(2n+1-1-1) + 1 LHS = T(n), but this is T(n-1), so we need to also -1 from 2n+1

RHS = 2\*2n - 2 + 1

RHS = 2n+1 - 1 = LHS After simplifying, RHS matches LHS!

**4.** Complexity order: T(n) = Θ(2n+1)

Forwards:

**1.** We will show 3 expansions to find a pattern

T(n) = 2T(n-1) + 1 We want to start with base case T(0)

T(0) = 1

T(1) = 2T(0) + 1 = 2(1) + 1 = 3 We know that T(0) = 1 from previously

T(2) = 2T(1) + 1 = 2(3) + 1 = 7

T(3) = 2T(2) + 1 = 2(7) + 1 = 15

… until

**2.** Writing out T(n)

T(n) = 2n+1-1 Based on the pattern from the last step, T(n) is derived

**3.** Now we substitute in LHS and RHS to show LHS=RHS

LHS = T(n)

RHS = 2T(n-1) + 1 Plugging in LHS into RHS

RHS = 2(2n+1-1-1) + 1

RHS = 2\*2n - 2 + 1

RHS = 2n+1 - 1 = LHS RHS = LHS

**4.** Complexity order: T(n) = Θ(2n+1)

**10.** T(n) = T(n-1) + n/2; T(1) = 1

Tree is below in table format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level # | Tree | # of Nodes | Sum of Row | Recursive Calls |
| 0 | n/2 | 1 | n/2 | T(n) |
| 1 | (n-1)/2 | 1 | (n-1)/2 | T(n-1) |
| 2 | (n-2)/2 | 1 | (n-2)/2 | T(n-2) |
| i | (n-i)/2 | 1 | (n-i)/2 | T(n-i) |

T(n) = We can pull out n/2

T(n) = (n/2) - (1/2) Isolate i, so that we can see T(n) easier

T(n) = (n/2)\*n - (1/2)[1+2+ … + n-1] Simplify

T(n) = n2/2 - (1 + 2 … + n)

Since our output is a quadratic, our runtime would be O(n2), which is inefficient.

**11.** T(n) = 2T(n/2) + 2nlog2n; T(2) = 4

Tree is below in table format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level # | Tree | # of Nodes | Sum of Row | Recursive Calls |
| 0 | 2nlog2n | 20 = 1 | 2nlog2n | T(n) |
| 1 | (2nlog2n)/2 (2nlog2n)/2 | 21 = 2 | 2nlog2n | T(n/2) |
| 2 | (2nlog2n)/22 (2nlog2n)/22 (2nlog2n)/22 (2nlog2n)/22 | 22 = 4 | 2nlog2n | T(n/22) |
| i | (2nlog2n)/2i … (2nlog2n)/2i | 2i | 2nlog2n | T(n/2i) |

We have a pattern of 2 = n/2i, so therefore, i = log2n-1

T(n) = We can pull out pretty much everything

T(n) = 2nlog2n Leaving us with the constant

T(n) = 2nlog2n(log2n)

T(n) = 2nlog22n Simply

Since T(n) = 2nlog22n = 2 \* O(nlog2n) = O(nlog2n) We pull out the constant

Therefore, T(n) = O(nlog2n), since 2nlog22n contains nlog2n.

**12.** It doesn’t make sense to have an algorithm have at least O(n2) because quadratic running times are the least efficient. The algorithm will take so long that it is meaningless to us. The input of n will result in the long quadratic run time, which is useless.