By: Sethu Senthil

Colobraters: Eric He, Zefanya Amadeo

Sources: On the last page

1)

a) $2^{f(n)}$ is $O(2^{g(n)})$
- Let k = some constant
- Bound Statement: $k * 2g^{n}$
- $f(n) \leq C * g(n)$
- $2^{f(n)} \leq 2^{C * g(n)}$
- $C* g(n) \neq g(n) \rightarrow$ in some cases it works out but not all
  - Hence it is **false**

b) $f(n)^2$ is $O(g(n)^2)$
- Given: f(n) = O(g(n))
- By Big O definition: f(n) = O(g(n)) where C > 0 and n_0 exists such that $n \geq n_0$, have:
  - $f(n) \leq C * g(n)$
  - Squaring both sides: $f(n)^2 \leq C^2 * g(n)^2$
  - Inquality holds! In other words it is stil bounded when it is squared.
  - Hence it is **true**

c) $2^{f(n)}$ is $O(2^{O(g(n))})$
- $f(n) \leq C * g(n)$
- It is clearly evident that no value of provides a bound for $2^{f(n)}$ that can't exceed for any C is large enough.

d) $\log_2 f(n)$ is $O(\log_2 g(n))$ where g(n) > 1
- $f(n) \leq C * g(n)$
- Log_2 on both sides : $\log_2 f(n) \leq \log_2 C * g(n)$
- Property of Logarithms: a * b = log a + log b
- Rewritten using property: $\log_2 C * g(n) = \log_2 C + \log_2 g(n)$
  - Since adding a constant value to a function does not change its groth function (rather just changes the starting or ending point) for Big O notation, this function is indeed bounded by O(2O(g(n)))
  - Hence it is **true**

2)

a) - Total # of mulltiplcations: $\frac{n(n+1)}{2} = \frac{n^2+n}{2}$

- Big O notation for multiplications: $O(n^2)$
- Big O notation for sums: $O(n)$

3)

a) Since each multiplication takes O(n log n). Time complexity: O(n * log(n)) + O(n * log(n)) + O(n * log(n)) + … O(n * log(n))

4)

a) A =

a	b
c	d

B =

e	f
g	h

Using Matrix Multiplication Rules:

C = A * B

C =

(a * e) + (b * g)	(a * f) + (b * h)
(c * e) + (d * g)	(c * f) + (d * h)

During this computation, it is clear that 4 additions are being performed and 8 multiplications are being done.

b) Given a matrix X

X = (given from question statement)

0	1
1	1

When matrix X is multiplied by itself i times (raising it to the power of i): X^i, the matrix values exhibit an increment with the rising value of i. The computational complexity of matrix multiplication, in terms of bit

requirements, maintains proportionality to the number of multiplications and additions involved, as matrix multiplication is just an abstraction of multiplication and additions as shown in part A.

Considering O(n) matrix multiplications, where i < n, and each multiplication operation necessitates O(1) bits, it follows that all entries in X^i possess O(n) bits for when i < n.

This is clearer in the following example:
$X^2 = X * X \quad = O(1)$
$X^3 = X * X^2 = O(1 + 1) = O(2)$

c) Given The Algrothim:
   - Given X^n and n = 1, running *matrix(X, 1)* the time complexity would be O(1)

   - **If n is even**, running matrix(X, evenNumber) requires two recursive calls with n/2 as it returns X * X.
     - M(n) is defined to be the time it takes to multiply two n-bit numbers from the problem statement, and we have proven this in the previous step.
     - Putting all the information together, we can deduce that each recursive call will take O(M(n)) time. And each matrix multiplication takes O(1) time complex as shown in the previous step.
     - Putting all of this together, the time complexity for an even number for the given algorithm is: **$T(n) = 2T(\frac{n}{2}) + O(M(n))$**
   - If n is odd (which is not 1), running matrix(X, oddNumber) runs Z * Z * X where Z = matrix(X, $\frac{n-1}{2}$). This also requires two recursive calls with n/2 with each call taking O(M(n))

with each matrix multiplication taking O(1) time complexity similar to that of even numbers, which means it also has the same time complexity as even numbers.

Recursive Call Time Complexity: O(log n)

- Since at every recursive call n is divided by 2 until it reaches the base case 1 and exits the recursive stack (after computing).

Mathematical Representation:

$T(n) = 2^0 * O(M(n)) + 2^1 * O(M(\frac{n}{2})) + 2^2 * O(M(\frac{n}{4})) ... + 2^k * O(M(1))$

$= O(M(n)) (1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^k})$

$= O(M(n)) * (2 - \frac{1}{2^k})$

$= O(M(n)) * (2 - \frac{1}{n})$ $\rightarrow$ $2^k = n$ since O(log n) recursive calls

$= \mathbf{O(M(n) * \log n)}$

5)

a) In a heap of height h, the minimum node count is 2^{h-1}, and the maximum node count is 2^{h}- 1. This is attributed to the heap being a complete binary tree, ensuring each level is maximally filled.

b) Yes, it is a min-heap because the array's values construct a valid binary tree where the value of the parent node is ≤ the children's values.

c) The Heapify-UP algorithm requires log(*n*) swaps in the worst-case scenario for a heap with *n* elements. The worst case for this algorithm would be when the child node's value surpasses the parent node's value, and the grandchild node's value exceeds the child node's value (in other words a max heap). To fix this, the algorithm initiates a sequence of swaps between the parent and child nodes, followed by swaps between the child and grandchild nodes. This iterative recursive process persists until the node's value is less than its parent node's value.

6)

a) I did not work in a group

b) I did not consult anyone other than my group members (myself)

c) Non-Class Materials:

i) https://www.geeksforgeeks.org/binary-heap/

ii) https://www.geeksforgeeks.org/difference-between-min-heap-and-max-heap/

iii) https://www.mathsisfun.com/algebra/matrix-multiplying.html

iv) https://www.cuemath.com/algebra/properties-of-logarithms/

v) https://tutorial.math.lamar.edu/classes/calci/thelimit.aspx

vi) https://www.youtube.com/watch?v=HqPJF2L5h9U&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=32