Question 1

A+B.

1. $64^{\log_4 n} = O(n^4)$

הוכחה:

נפתח את הביטוי בשמאל: $64^{\log_4 n}=(4^3)^{\log_4 n}=4^{\log_4(n^3)}=n^3$

נראה כי $n^3=O(n^4)$

יהיו $c=1$, $n_0=1$ כך שלכל $n\ge n_0$ מתקיים $n^3\le c\cdot n^4=n^4$

אזי $n^3\in O(n^4)$ ■

2. $3^n=O(2^n)$

הפרכה:

נניח בשלילה כי מתקיים $3^n=O(2^n)$. כלומר קיימים $c>0, n_0\in\mathbb{R}$ כך שלכל $n\ge n_0$ מתקיים $3^n\le c\cdot 2^n$.

נפתח את הביטוי ונקבל $(\frac{3}{2})^n\le c$. מתקיים $\lim_{n\to\infty}(\frac{3}{2})^n=+\infty$

ואילו c הוא קבוע, אזי הגענו לסתירה ■

3. $n^2\log(n)+n\log^2(n)=O(n^2\log(n))$

הוכחה

נשים לב כי לכל $n\ge 1$, מתקיים $\log(n)<n$. על כן מתקיים

גם $n\log^2(n)<n^2\log(n)$.

נפתח את הביטוי $n^2\log(n)+n\log^2(n)\le 2n^2\log(n)$.

כלומר עבור $c=2$, $n_0=1$, מתקיים

$n^2\log(n)+n\log^2(n)\le c\cdot n^2\log(n)$, כנדרש ■

4. $f_1(n)=O(g_1(n)) \wedge f_2(n)=O(g_2(n)) \to f_1\circ f_2(n)=O(g_1\circ g_2(n))$

הפרכה:

נפריך את הטענה באמצעות דוגמא.

נגדיר $f_1(n)=g_1(n)=2^n$, $g_2(n)=n$, $f_2(n)=2n$.

נשים לב שמתקיים $f_1=O(g_1)$ וגם $f_2=O(g_2)$.

נפתח את הביטויים $f_1\circ f_2=2^{2n}=4^n$, $g_1\circ g_2(2^n)$

ולא קיימים $c>0, n_0\in\mathbb{R}$ כך שלכל $n\ge n_0$ $4^n\le c\cdot 2^n$. לכן הטענה שגויה ■

ב

1. הוכחה:

נסמן $M=\max\{a_1,\dots,a_n\}$, $\sum_{i=1}^n a_i$. נראה כי קיים חסם עליון ותחתון.

מהגדרת max מתקיים שלכל $i\in N$ $a_i\le M$. כלומר מתקיים הביטוי:

$\sum_{i=1}^n a_i\le\sum_{i=1}^n M=M\cdot n$. משמע, $S=O(n\cdot m)$

יש חסם עליון

בשביל לחסום תחתון, נסכום את האיברים שגדולים מ-$c\cdot M$

גדול מהביטוי $b\cdot n\cdot c\cdot M$. ונסמן $S_2\ge b\cdot n\cdot c\cdot M$. מתקיים $S\ge S_2$

מאחר ש-S_2 הוא סכום של קבוצה חלקית מהסדרה.

על כן $b\cdot n\cdot c\cdot M\le S\le n\cdot M$ והוא חסום כדרוש ■

2. $n\log(n)=O(\log(n!))$

הוכחה:

יהי $a_i=\log(i)$ סדרה. נשים לב שמתקיים $\sum_{i=1}^n a_i=\log(n!)$,

וכי $\max\{a_1,\dots a_i\}=\log(n)$. על כן, לפי ההוכחה ב-1 מתקיים

$\log(n!)=\Theta(n\cdot\log(n))$ כנדרש ■

3. הוכחה

מאחר ש-$k\ge 1$, מקסימום איברי הסדרה $\{a_i\}_{i=1}^n=i^k$ הוא n^k.

נבחר $b=\frac{1}{2}$ כך שמתקיים שלכל $i\ge\frac{n}{2}$ $a_i=i\ge(\frac{n}{2})^k=\frac{n^k}{2^k}$.

מאחר ש-k קבוע, ניתן לבחור $c=\frac{1}{2^k}$. לפי ההוכחה ב-1 מתקיים

$P_k(n)=\Theta(n\cdot n^k)$, וכן $P_k(n)=\Theta(n^{k+1})$ כנדרש ■

4. הוכחה:

מאחר שלכל i מתקיים $i^k\le n^k$, מתקיים כי: $S=\sum_{i=1}^n 2^i\cdot i^k\le n^k\cdot\sum_{i=1}^n 2^i$

לפי נוסחת סכום לסדרה הנדסית $\sum_{i=1}^n 2^i=2\cdot\frac{2^n-1}{2-1}=2(2^n-1)<2^{n+1}$

ולכן מתקיים כי $S\le 2^{n+1}\cdot n^k=2(2^n\cdot n^k)$, משמע $S=O(2^n\cdot n^k)$ חסם עליון

מאחר שהאיבר המקסימלי בסכום הוא $2^n\cdot n^k$ כלומר מתקיים

$S\ge 2^n\cdot n^k$ חסם תחתון והביטוי חסום עם 2 חסמים מאותו

סדר גודל כדרוש ■

C.

1. The worst-case scenario complexity is $O(n^2)$. let n_0 = initial n. since n is halved on each iteration the number of iterations is $log(n_0)$. the for loops runs n times so the total iterations are $\{n_0/2 + n_0/4 + ... \}$. Which converges to n_0. In the worst case scenario, ***i in L*** is true for every i. So , because of the append method the total complexity will be $O(n^2)$.
2. The worst case scenario is $O(n * log^2(n))$. the outer loop runs n-500 times meaning $O(n)$. the middle loops runs log m times which in the worst-case is $log(n)$. who lower while loops runs $log(n)$ times. So in total the worst-case complexity is $O(n * log^2(n))$
3. The worst-case scenario complexity is $O(n^2)$. the outer loops runs n times and the inner loop run over ***L[:i+1]*** and since i approaches n, the inner loops cost is $O(n)$. so the total complexity is $O(n^2)$.

Question 2

a. In the in-class version of binary search, we were dealing with natural numbers (indexes in an array) so we determined the middle with floor division (\\). So for example if we search for the middle of 3, we'd get 1. Hence the addition of 1 to solve for that. In this case however, we're dealing with real numbers so we can just use regular division.

b.

The function call statement:

```
find_root(lambda x: x**2 - 4, 0, 3)
```

The console prints from the function

searching in (0 , 3)
searching in (1.5 , 3)
searching in (1.5 , 2.25)
searching in (1.875 , 2.25)
searching in (1.875 , 2.0625)
searching in (1.96875 , 2.0625)
searching in (1.96875 , 2.015625)
searching in (1.9921875 , 2.015625)
searching in (1.9921875 , 2.00390625)
searching in (1.998046875 , 2.00390625)
searching in (1.998046875 , 2.0009765625)
searching in (1.99951171875 , 2.0009765625)
searching in (1.99951171875 , 2.000244140625)
searching in (1.9998779296875 , 2.000244140625)
searching in (1.9998779296875 , 2.00006103515625)
searching in (1.999969482421875 , 2.00006103515625)
searching in (1.999969482421875 , 2.0000152587890625)
searching in (1.9999923706054688 , 2.0000152587890625)

searching in (1.9999923706054688 , 2.0000038146972656)
searching in (1.9999980926513672 , 2.0000038146972656)
searching in (1.9999980926513672 , 2.0000009536743164)
searching in (1.9999995231628418 , 2.0000009536743164)
searching in (1.9999995231628418 , 2.000000238418579)
searching in (1.9999998807907104 , 2.000000238418579)
searching in (1.9999998807907104 , 2.0000000596046448)
searching in (1.9999999701976776 , 2.0000000596046448)
searching in (1.9999999701976776 , 2.000000014901161)
searching in (1.9999999925494194 , 2.000000014901161)
searching in (1.9999999925494194 , 2.0000000037252903)
searching in (1.9999999981373549 , 2.0000000037252903)
searching in (1.9999999981373549 , 2.0000000009313226)
searching in (1.9999999995343387 , 2.0000000009313226)
searching in (1.9999999995343387 , 2.0000000002328306)
searching in (1.9999999998835847 , 2.0000000002328306)
searching in (1.9999999998835847 , 2.0000000000582077)
searching in (1.9999999999708962 , 2.0000000000582077)

c. For epsilon = 10**-1000, the function never reaches the root of the function because epsilon is orders of magnitude smaller than the smallest float in 64-bit representation. So, the "most" f(M) can be is equal to epsilon but never smaller.

d. The upper limit on the algorithm's accuracy is the result of 64-bit float representation. In the range of 0-1 the margin for error is 2^{-52} but in the range 2-3 the margin of error is 2^{-51}. Hence, the upper limit for 0-3 is too 2^{-51}.

e. Having the interval be between 2 consequent powers of 2 (specifically 2,1) means that there are 2^{52} possible values between the 2 numbers. This means that every bisection cuts that in half. So using $2^{52}/2^{k}$ we can calculate that the upper limit on the number of iterations to be 52.

Question 3

a. Implemented in python file
b. Implemented in python file
c. Implemented in python file
d. The function's complexity is $O(kn + 5^{k})$. disregarding the complexity of creating the helper list, the first action is to iterate over ***lst*** n times. And for each of the items in the list, go into ***string_to_int()*** which has a complexity of k since we're looping over each of the k characters. This part's complexity is $n * k$. Then we loop over the 5^{k} long helper. So, in total, the complexity $kn + 5^{k}$.
e. Implemented in python file

f. The function's time complexity is $O(5^k * kn)$. we iterate every natural number up to 5^k. for each iteration we loop over ***lst*** (of length n) for each of which we, in turn, receive the ***string_to_int()*** value (function with a complexity of k since wer'e looping over a "k" long string). In total the time complexity is $O(5^k * kn)$.
As for the memory complexity, we're using $O(k)$ because the ***string_to_int()*** function is creating a "k" long dictionary.

Question 4

1. The time complexity of the function is $O(n)$. compared to a classic binary search implementation, where the complexity of each iteration is constant ($O(1)$), in the case of this function, slicing the list if a costly operation that is equal to a geometric series ($n/2\ +\ n/4\ +\ n/8$ ....) and as we know, this converges to n. Hence, the $O(n)$ complexity
2.
    a. Here's a rundown of points 2-6 and what might be wrong with them

        - Valid. The mid-point is necessary for the quicksort algorithm to work at $O(log(n))$.
        - We don't check ***lst[mid] == s*** because we expect s not to move but because the mid point is the point we're currently looking into. It's entirely possible for s not to have moved but rather, other elements have moved. What should have been said is that s didn't move relative to it's neighbours.
        - The reasoning is valid. If s is not at mid check around it.
        - The part contains an error. Since ***lst[mid-1]*** might be smaller than s like in the case where ***lst=[2,1,3,5,4] s=2***. After landing on 3, ***lst[mid-1] = 1*** and so, we go into the else and never reach 2.
        - Valid, but only in the case that all other errors are fixed.otherwise we might never reach the number and return None as if it's not there

    b. The code doesn't consider some edge-cases here. In Part 5 There's a risk of ***lst[mid-1]*** being out of bounds say if mid = 0.
    c. *Implemented in python file*

Question 5

Implemented in python file