By: **Sethu Senthil**

HW 4

Question 1:

Algorithm:

```
def find_fixed_point(arr):
    # Initialize low and high pointers
    low = 0
    high = len(arr) - 1

    # Iterate until low and high pointers meet (Binary Search)
    while low <= high:  # O(log n) - Binary search time complexity
        # Calculate mid index
        mid = (low + high) // 2  # O(1)

        # Check if the element at mid is equal to mid
        if mid == arr[mid]:
            return mid
        # If mid is less than the element at mid, search on the left side
        elif mid < arr[mid]:
            high = mid - 1  # O(1)
        # If mid is greater than the element at mid, search on the right side
        else:
            low = mid + 1  # O(1)

    # Return -1 if no fixed point is found
    return -1  # O(1)
```

1) This method is like finding a special number in a sorted list. It starts in the middle and checks if the number matches its position. If not, it figures out if the special number is on the left or right side and keeps going until it finds the special number or concludes it's not there. Since this algorithm is basically a fork of binary search, it has a O(log n) time complexity.
2) Using the masters theorem
    - $T(n) = aT\frac{n}{b} + f(n)$ → Masters Theorm
    - a = 1 → (1 sub problem per call)
    - b = 2 → (binary search halves in every call)

- f(n) = O(1) → all other operations (like dividing) are constant time operations
- This falls under case 2 of the master theorem: O(log n)

2)

a) Strassen algorithm. This algorithm is a divide-and-concur algorithm that is used to efficiently calculate the product of two matrices.

Using this algorithm we can reduce the computational complexity to just 5 multiplication through the following.

- Let A be a 2x2 matrix
- A =

a	b
c	d

- A^2 =

a	b
c	d

×

a	b
c	d

- Product via Strassen algorithm

$a^2 + bc$	$ab + bd$
$ac + cd$	$bc + d^2$

Strassen algorithm intermediate products:

$p_1 = a * (d - b)$

$p_2 = (a + b) * d$

$p_3 = (c - d) * a$

$p_4 = c * (a + b)$

$p_5 = (a + d) * (c + d)$

Using the intermediate products, we can calculate each block in the matrices

$a^2 + bc = p_1 + p_3$
$ab + bd = p_1 + p_2$
$ac + cd = p_4 + p_1$
$bc + d^2 = p_5 - p_1$

Since we only have 5 intermediate products, we only utilized 5 multiplications. Although we used more arithmetic operations than the naive approach, we assume that the time complexity to perform multiplication costs significantly higher than the time complexity for addition and subtraction.

b) This is not true. We do not get the same type of problem as seen below if we consider matrix A from the previous part:

Matrix square from the previous part:

$a^2 + bc$	$ab + bd$
$ac + cd$	$bc + d^2$

Matrix based on the modified algorithm:

$A^2 + BC$	$B(A + D)$
$C(A + D)$	$BC + D^2$

Although we do end up getting 5 subproblems it is not the same as the subproblems from part A. In fact, we get an incorrect answer. Since matrix operations aren't communicative, it is not the same (so we can't distribute and prove that they are the same, it won't be applicable in this case as the order of the operations matters)

c) If squaring a $2n \times 2n$ matrix can be done in $O(nc \log n)$ time, and constructing the matrices C and D and combining their results takes

polynomial time (which is the case here), then the overall time complexity of multiplying two n × n matrices using this approach is also O(nc log n).

d) Therefore, if squaring matrices can be done in O(nc log n) time, then matrix multiplication can also be done in O(nc log n) time, as shown

3.

a) If x is a majority element in an array it means the following

- More than half the elements in the array are equal to x, in mathematical terms: at least (n/2 + 1) are x
- An example: If we take an array of 6 [x,x,x,x,9,7]
    - x is the majority element
    - If we take the first half of the array : [x,x,x], x is still the majority element
    - If we take the second half of the array [x,9,7], x is not the majority element of the array
    - This fits the statement that x is the majority element in the first half or the second half of the array
- Proof using counter positive:
    - Assume x is a majority element in the array A[0, ..., n-1], but it is not a majority element in either the first half or the second half of the array.
    - let's also assume x is not a majority element in the first half.
    - This means that the count of occurrences of x in the first half of the array must be less than or equal to half of the length of the first half of the array, which is n/2.
    - Since x is not a majority element in the second half, the count of the second half of x must be less than or equal to half of the length of the second half of the array, which is n/2.
    - Therefore, count of first half of x + count of second half of x ≤ n/2 + n/2 = n

- **Contradiction**: But this contradicts the fact that x is a majority element in the entire array, as it implies that the total count of x in the array is less than or equal to n, which is not true.
- Hence, our initial assumption that x is not a majority element in either the first half or the second half of the array is false. Therefore, if x is a majority element in the array, it must also be a majority element in either the first half or the second half of the array.

b)
Python Code:

```
def is_majority_element(arr, x):
    count = 0  # O(1)
    n = len(arr)  # O(1)

    # Count occurrences of x in the array
    for a in arr:  # O(n)
        if a == x:  # O(1)
            count += 1  # O(1)

    # Check if x is a majority element
    if count > n // 2:  # O(1)
        return True  # O(1)
    else:  # O(1)
        return False  # O(1)
```

In this Python code, we are basically using a linear searching-based algorithm (aka just a regular loop), traversing every element in the array. Since the array has n elements, the time complexity is O(n). This is a pretty simple straightforward algorithm.

c)
Python Code:

```
def is_majority_element(arr, x):
    count = 0  # O(1)
    for a in arr:  # O(n)
        if a == x:  # O(1)
            count += 1  # O(1)
```

```
    return count > len(arr) // 2  # O(n)

def majority_element_dc(arr):
    # Base case: If the array has only one element, it is the majority element
    if len(arr) == 1:  # O(1)
        return arr[0]  # O(1)

    # Divide: Split the array into two halves
    mid = len(arr) // 2  # O(1)
    left_half = arr[:mid]  # O(n)
    right_half = arr[mid:]  # O(n)

    # Conquer: Recursively check if x is a majority element in both halves
    left_majority = majority_element_dc(left_half)  # T(n/2)
    right_majority = majority_element_dc(right_half)  # T(n/2)

    # Combine: Check if either left_majority or right_majority is a majority element in
the entire array
    if is_majority_element(arr, left_majority):  # O(n)
        return left_majority  # O(1)
    elif is_majority_element(arr, right_majority):  # O(n)
        return right_majority  # O(1)
    else:  # O(1)
        return None  # O(1)
```

This is a more time-efficient version of the previous algorithm using divide and concur. In this algorithm, we split it into three different steps:

- **Divide Step:** Divide the array A into two halves, A[0, ..., mid] and A[mid+1, ..., n-1], where mid is the midpoint of the array.
- **Conquer Step**: Recursively check if the candidate element x is a majority element in both halves of the array.
- **Combine Step**: Combine the results from the two halves to determine if x is a majority element in the entire array.

To use master's theorem to prove the time, we first need to take the recurrence relation for this algorithm which is T(n) = **2**T(n/**2**) + **O(n)**, since In the Conquer step, the algorithm recursively solves two subproblems of size n/2 each (half of the array). In the Combine step, the results of the

subproblems are combined. This step takes linear time O(n) because it involves comparing the majority elements found in the subarrays to determine if they are majority elements in the entire array. Then using the master theorem it falls into case 2 as c is greater than log_a b, hence the running time of the algorithm is O(n log n).

d)

Proof:

- Suppose A has a majority element, let this element be denoted as “x”
- Then we have 2 cases
    - If x is present in a pair, we keep one occurrence of it and discard the other element, as described in the algorithm procedure.
    - If x is not present in any pair, then it must be the unpaired singleton element in A.
- In either case, x contributes at least one occurrence in L.
- Since x is a majority element in A, its occurrences outnumber the occurrences of any other element.
- Hence, x will also be a majority element in L.

e)

Python Code:

```
def find_majority_element(arr):
    # Base case: if the array has only one element, it is the majority element
    if len(arr) == 1:  # O(1)
        return arr[0]  # O(1)

    # Pair up the elements of the array
    pairs = []  # O(1)
    i = 0  # O(1)
    while i < len(arr) - 1:  # O(n)
        if arr[i] == arr[i+1]:  # O(1)
            pairs.append(arr[i])  # O(1)
            i += 2  # O(1)
        else:
            i += 2  # O(1)

    # If there's an unpaired element, add it to the pairs list
    if i == len(arr) - 1:  # O(1)
```

```
        pairs.append(arr[i])  # O(1)

    # Recursive call with the pairs list
    return find_majority_element(pairs)  # T(n/2)
```

This is another implementation to solve the problem, except this time it uses a different method using the techniques described in the problem statement by first pairing up the elements, and then computing each pair. It is clear that this is done in linear time as the highest big O() function is O(n) when analyzing the algorithm's time complexity on each step. This is because the program utilizes a while loop that iterates through the whole array (in the worst-case scenario).

4)

a) Using Master Theorm: a = 27, b = 3, f(n) = $17n^3$, d = 3 * 17

$log_b(a) = log_3(27) = log_3(3^3) = 3$

Therefore, Case 1 since $d > log_b(a)$, hence: T(n) = $O(n^3)$

b) Using Master Theorm: a = 2, b = 7, f(n) = $\sqrt{n}$, d = ½ = 0.5

$log_b(a) = log_7(2) = 0.3562$

Therefore, Case 1 since $d > log_b(a)$, hence: T(n) = $O(\sqrt{n})$

c) Using iteration technique to find pattern:

$T(n) = T(n-1) + c^n$

$= [T(n-2) + c^{n-1}] + c^n$

$= T(n-2) + c^{n-1} + c^n$

$= T(n-3) + c^{n-2} + c^n$

.

.

.

$= T(1) + c^2 + c^3 + \ldots. c^n$

Then, let's sum the geometric series $c^2 + c^3 + \ldots. c^n$:

$c^2 + c^3 + \ldots. c^n = c^2(1 + c + c^2 + \ldots + c^{n-2}) = c^2 \frac{1-c^{n-1}}{1-c}$

Hence, $T(n) = T(1) + c^2 \frac{1-c^{n-1}}{1-c}$

Simplifaction through substitution as T(1) = O(1):

$T(n) = O(1) + c^2 \frac{1-c^{n-1}}{1-c}$

The tightest upper bound is $O(c^n)$ when ignoring constants and lower order terms.

d) Using iteration technique to find pattern:

$$T(n) = T(n-1) + n^c$$
$$= [T(n-2) + (n-1)^c] + n^c$$
$$= T(n-2) + (n-1)^c + n^c$$
$$= T(n-3) + (n-2)^c + (n-1)^c + n^c$$
.
.
.
$$= T(1) + 1^c + 2^c + \ldots n^c .$$

If C ≥ 1, all the terms in the sum are at least n^c which means:

$$1^c + 2^c + \ldots n^c = n^{c+1}$$

Therefore, T(n) = $O(n^{c+1})$ is the tightest upper bound for the recurrence relation.

e) Using Master Theorm: a = 7, b = 4, f(n) = n, d = 1

$$log_b(a) = log_4(7) = 1.403$$

Since d (1) < $log_b(a)$ (1.4), it falles under the 3rd case. Hence, O($n^{log_4(7)}$)

f) Using Master Theorm: a = 1, b = 2, f(n) = 2.25^n

$$log_b(a) = log_2(1) = 0$$

When comparing with the log value with f(n), since 2.25^n grows faster than any polynomial n^k (k being any constant), we have f(n) = 2.25^n = $\Omega(n^0)$.

Hence, we are in the 3rd case of the Master theorem:

$$T(n) = \Theta(f(n)) = \Theta(2.25^n)$$

Therefore, the tightest upper bound for T(n) is $O(2.25^n)$.

g) Using Master Theorm: a = 49, b = 36, f(n) = $n^{3/2}\log(n)$

$log_{36}(49) = 1.08603$

3rd case of master theorm: O($n^{log_{36}(49)}$)

h) Base Case: For n = 1, T(1) = O(1) according to hint.

Hypothesis: Assume that T(k) ≤ ck for all k < n, where c is a constant.

Induction:

$T(n) = T(\frac{n}{3}) + T(\frac{n}{6}) + T(\frac{n}{12}) + n$

$\leq c(\frac{n}{3}) + c(\frac{n}{6}) + c(\frac{n}{12}) + n$ $[using\ hypothisis]$

$= \frac{7}{12}cn + n$

$= (\frac{7}{12}c + 1)\, n$

Let $c_1 = \frac{7}{12}c + 1$. We know that $c_1 > 0$, since c > 0.
Hence, by induction T(n) ≤ c_1n for some constant $c_1 > 0$.
Hence, T(n) = O(n).

i) Base Case: n = 1, T(1) = O(1 log 1) = O(1) according to hint.

Hypothesis: Assume that T(k) ≤ ck log k for all k < n, where c is a constant.

Induction:

$T(n) = T(\frac{3n}{7}) + T(\frac{4n}{7}) + \theta(n)$

$\leq c(\frac{3n}{7log(\frac{3n}{7})} + c(\frac{4n}{7\,log(4n/7)}) + cn$ $[using\ hypothisis]$

$= \frac{3}{7}cn\,log(\frac{3n}{7}) + \frac{4}{7}cn\,log(\frac{4n}{7}) + cn$

$= \frac{7}{7}cn\,log\,n = cn\,log\,n$

Let $c_1 = c$. We know that $c_1 > 0$ since $c > 0$.
Hence, by induction, we see that $T(n) \leq c_1 n \log n$ for some constant $c_1 > 0$.

Hence, T(n) = O(n log n)

j) Unfolding the Recurrence:

- T(n) = $\sqrt{n}T(\sqrt{n} + 11n$
- T(n) = $\sqrt{n}\sqrt{\sqrt{n}}T(\sqrt{\sqrt{n}}) + 11n$
- $T(n) = \sqrt{n}\sqrt{\sqrt{n}}\sqrt{\sqrt{\sqrt{n}}}T(\sqrt{\sqrt{\sqrt{n}}}) + 11n$

.
.
.

After k steps:

$T(n) = n^{2^{-k}} T(n^{2^{-k}})$ + 11n

Continuing this until $2^{-k} = 1$, we get k = log log n.

Explanation:

- At each step, the term $T(n^{2^{-k}})$ becomes T(1) (bounded by a constant) after k = log log n steps.
- Also $n^{2^{-k}} = n^{2-log\ log\ n} = n^{\frac{1}{log\ n}}$ as k goes from 0 to log log n, $n^{1/\log n}$ becomes n.
- Hence, after unfolding k = log log n steps, we have T(n) = O(n log log n) since each term is bounded by O(n log log n).
- Therefore, T(n) = O(n log log n)

5)
Using the recursive structure of the Hadmard matrices and divide and concur approach based on the FFT algorithm (Fast Fourier Transform).

- **Base Case:** for H_0 its just a 1x1 matrix, which means it only has a single element. This means the product of H_0v is just v itself.
- **Divide and Conquer:** for H_k where $k > 0$, we can divide the matrix into four different quadrants of size $2^{k-1} * 2^{k-1}$. Let these quadrants be represented as A, B, C, and D, where each quadrant itself is a Hadmard matrix of size $2^{k-1} * 2^{k-1}$.
- **Matric-Vector Product Calculation**: Expressing H_kv as such:

  (Just handwriting the matrix as neatly as possible, but the rest is typed)

$$H_kV = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} = \begin{pmatrix} AV_1 + BV_2 \\ CV_1 + DV_2 \end{pmatrix}$$

  Recurrence: Each of the products Av_1, Bv_2, Cv_1, Dv_2 can be calculated recursively

  Complexity Analysis:

  - At each level of recursion, the problem size is divided by 2.
  - The number of recursive calls is therefore logarithmic in the vector size n (log2(n)).
  - Within each recursive call, the dominant operations are the recursive calls themselves and constant-time arithmetic operations.
  - The total number of operations is therefore bounded by O(n * log(n)), where n is the matrix size.

Python Code for this solution:

```python
def hadamard_product(Hk, v):
 """
 This computes the matrix-vector product of a modified Hadamard matrix Hk and a vector
v using the divide-and-conquer strategy.
 Time complexity: O(n log n)

 Args:
     Hk: A modified Hadamard matrix of size 2^k x 2^k.
     v: A column vector of size 2^k.

 Returns:
     The result of Hk * v as a column vector.
 """

 n = len(v)  # O(1) operation to get the vector length

 # Base case: when n is 1, no computation is needed, return the input vector directly.
 # Time complexity: O(1)
 if n == 1:
   return v

 # Recursive case: when n is greater than 1, divide the problem into halves.
 else:
   half_n = n // 2  # O(1) operation to find the half size

   # Split the input vector into two halves
   v1 = v[:half_n]  # Slicing operation, O(1)
   v2 = v[half_n:]  # Slicing operation, O(1)

   # Recursively compute the matrix-vector products for each half using the same
function
   # Time complexity: 2 * T(n/2) (recursive calls)

   product1 = hadamard_product(Hk[:half_n, :half_n], v1)
   product2 = hadamard_product(Hk[half_n:, half_n:], v2)

   # Combine the results by stacking and subtracting elements based on the matrix
structure
   # Time complexity: O(n) (concatenation and element-wise operations)
   return np.concatenate((product1 - product2, product2 - product1))
```

6)

- Worked with: Zefanya Amadeo, Eric He
- Did not receive significant help from other people

Sources:

- https://www.geeksforgeeks.org/strassens-matrix-multiplication/
- https://study.com/academy/lesson/square-matrix-definition-lesson-quiz.html#:~:text=What%20is%20squaring%20a%20matrix,is%20row%20by%20column%20multiplication.
- https://www.geeksforgeeks.org/majority-element/
- https://leetcode.com/problems/majority-element/description/
- https://cs.stackexchange.com/questions/11635/particularly-tricky-recurrence-relation-masters-theorem
- https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-multiplication/
- https://nhigham.com/2020/04/10/what-is-a-hadamard-matrix/
- https://www.geeksforgeeks.org/videos/matrix-chain-multiplication-bdrrpf/
-