Problem #2

(a)
I also uploaded a python file to Canvas, please kindly check.

```python
# HW1 Problme 2: Hybrid Sorting
# For the hybrid_sort part,
# I used Copilot to generate the code and modified it myself.
import heapq
import math


def quick_sort(arr: list) -> list:
    '''
    Quick sort to sort an array arr of n elements in ascending order.
    '''
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)


def insertion_sort(arr: list) -> list:
    '''
    Insertion sort to sort an array arr of n elements in ascending order.
    '''
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            # Move elements of arr[0..i-1], that are greater than key,
            # to one position ahead of their current position.
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr


def heapsort(arr):
    '''
    Heapsort using heapq module.
    '''
    h = []
    for value in arr:
```

```
        heapq.heappush(h, value)
    return [heapq.heappop(h) for _ in range(len(h))]


def hybrid_sort(arr) -> list:
    '''
    A hybrid sorting algorithm that combines the quick sort, insertion sort,
        and heap sort.
    The algorithm uses the quick sort to sort the array
        until the depth reaches 2 * math.log2(n), then it switches to the
        heap sort to sort the array.
    When the size of the array is less than or equal to 16,
        the algorithm uses the insertion sort to sort the array.
    '''
    def _hybrid_sort_helper(arr, depth, max_depth):
        # If the size of the array is less than or equal to 16,
        # simply use the insertion sort to sort the array.
        if len(arr) <= 16:
            return insertion_sort(arr)
        # If the depth is greater than the max_depth,
        # use the heap sort to sort the array afterwards.
        if depth > max_depth:
            return heapsort(arr)

        # Otherwise, use the quick sort to sort the array.
        # aka, if low < high:
        pivot = arr[len(arr) // 2]  # Choose the middle element as the pivot.
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]

        return (_hybrid_sort_helper(left, depth + 1, max_depth) +
                middle +
                _hybrid_sort_helper(right, depth + 1, max_depth))

    max_depth = 2 * math.log2(len(arr))
    return _hybrid_sort_helper(arr, 0, max_depth)
```

(b)

Time complexity is $O(n) = n \cdot \log n$.

Best-case scenario: the time complexity is $O(n) = n \cdot \log n$.

In the best case, the array is nearly sorted and only insertion sort and quick sort will be used. In addition, we know that the time complexity for insertion sort is $O(1)$, and the time complexity for quick sort is $O(n) = n \cdot \log n$ in the best case. Since $n \cdot \log n$ grows faster than constant time, we could say the time complexity for above hybrid sort in the best case, is $O(n) = n \cdot \log n$.

Average-case scenario: the time complexity is still $O(n) = n \cdot \log n$.

In the average case, the pivot picked up in the quick sort would be nearly balanced, and the time complexity for insertion sort will be $O(n) = n$. When the depth exceeds $2 \cdot \log n$, we would switch to heap sort, the time complexity of which is $O(m) = \log m$ for subarrays of length m. Therefore, the time complexity for above hybrid sort in the average case, is still $O(n) = n \cdot \log n$.

Worst-case scenario: the time complexity is still $O(n) = n \cdot \log n$.

In the worst case, the quick sort would picke the most unbalanced pivot, which leads to $O(n) = n^2$. However, since we have designed the algorithm as if the depth is exceeding $2 \cdot \log n$ then we would switch to heap sort, we could avoid such worst case, leaving the time complexity of quick sort to be $O(n) = n \cdot \log n$. In addition, the time complexity of heap sort is $O(n) = n \cdot \log n$ as well. Finally, if the length of the subarray is equal to or less than 16, the insertion sort would be used. Insertion sort has a time complexity of $O(1)$ as for small array, and $O(n) = n$ on average. Therefore, we could say that the time complexity for above hybrid sort even in the worst case, is still $O(n) = n \cdot \log n$.

In conclusion, the time complexity for this hybrid sort, is $O(n) = n \cdot \log n$.

(c)

Sort Algorithm		Hybrid Sort	Quicksort	Heapsort	Insertion Sort
Size	Array Type				
100	nearly_sorted	0.000035	0.000057	0.000022	0.000024
	random	0.000078	0.000070	0.000031	0.000137
	reverse_sorted	0.000045	0.000048	0.000025	0.000247
1000	nearly_sorted	0.000431	0.000685	0.000231	0.000429
	random	0.000903	0.001053	0.000265	0.017418
	reverse_sorted	0.000699	0.000665	0.000326	0.031863
10000	nearly_sorted	0.006604	0.008466	0.002956	0.005592
	random	0.011669	0.013253	0.003493	1.880364
	reverse_sorted	0.008117	0.008313	0.004177	3.804048
100000	nearly_sorted	0.083590	0.104143	0.036055	0.041171
	random	0.154533	0.172997	0.053966	193.347324
	reverse_sorted	0.102506	0.104657	0.051455	386.607629

(d)

From the list in (c), we know that the hybrid sort algorithm is almost always the second fastest algorithm, regardless of small sized array or large sized array. Also, as proved in (b), this kind of sort performs well no matter the array is nearly sorted, random or completely reversed. Although hybrid sort still cannot beat heap sort, it truly is efficient than insertion sort and quicksort.

On the other hand, quicksort also provided an acceptable result. Sometimes it is faster than the hybrid sort, but overall, it shows similar performance with hybrid sort, which also accounts for the fact that those two algorithms have the same time complexity.

As for heapsort, it is the fastest algorithm of sorting any size array. Especially in terms of large sized array, heapsort is nearly as twice quick as the second fastest algorithm.

Finally, obviously insertion sort is a good algorithm for small, nearly sorted array. But once it comes to large, reversed arrays or random arrays, insertion sort would perform terribly. As it is shown from above list, it took nearly 6 minutes to sort a completely revised 100,000-sized array.

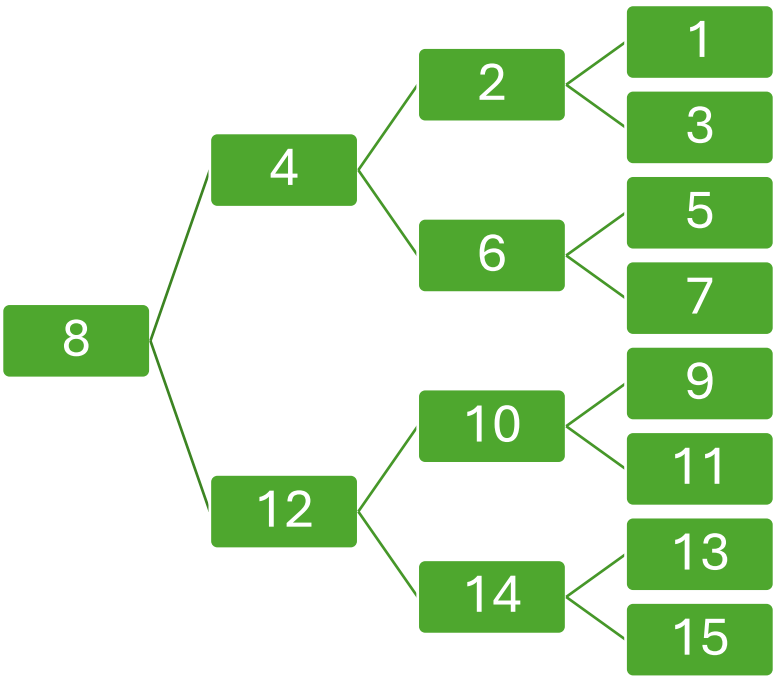
Regarding real-world applications, because of the adaptability of hybrid sort, I would say it would be particularly useful in systems where the data size can vary from time to time. Since the ability to switch between sorting algorithms based on the recursion depth and array size ensures efficient sorting with minimal latency. For example, hybrid sort would be much helpful in terms of managing flood detection data.

Problem #3

(a)

(b)

The time complexity of Binary Search is $O(n) = \log n$. So, when searching a number from 15 numbers, the target must be found within $\log 15 < \log 16 = 4$ times. A guessing number tree is as follows:



In other words, at first let us suppose the tentative target number is 8, if the real answer is smaller than the tentative number, then let the tentative number be 4, which is the middle

number of all numbers smaller than 8. Repeat this process. Apply the same logic for cases where the real number is greater than the tentative number. As such, all possibilities could be listed as above, and no matter what number the computer holds, we could "guess" which one it is within at most 4-time tries.

(c)

The recurrence relation for T(n) is $T(n) = T\left(\frac{n}{2}\right) + c$, where c stands for efforts we spend on guessing the number. According to the Master Theorem which takes $T(n)$ in the form of $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we know that:

$\because T(n) = T\left(\frac{n}{2}\right) + c$

$\therefore \mathrm{a} = 1, \mathrm{b} = 2 \text{ and } f(n) = c$

$\because$ The watershed function $n^{\log_b a} = n^{\log 1} = n^0 = 1$, which grows at nearly the same asymptotic rate as $f(n)$.

$\therefore$ case 2 of the Master Theorem applies to this case. That is:

$T(n) = \Theta\left(n^{\log_b a} \cdot (\log n)^{k+1}\right)$

$\because k = 0$ in this case

$\therefore T(n) = \Theta(\log n)$

Therefore, we have shown that $T(n) = \Theta(\log n)$.■

(d)

$$A(1023) = \frac{9217}{1023}$$

From the graph in (b) we know that, if the target number is 8 then it takes 1 time to guess it; if the target number is 4 or 12, then it takes 2 times (8-4/12) to guess. Similarly, it takes k times to guess a certain number, where k represents the level that number is located. Therefore, the total number of guessing a number from 15 numbers, is:

$$A(15) = \frac{1 \times 2^0 + 2 \times 2^1 + 3 \times 2^2 + 4 \times 2^3}{15}$$

$$A(15) = \frac{1 \times 1 + 2 \times 2 + 4 \times 3 + 8 \times 4}{15} = \frac{49}{15}$$

Therefore, we could say the average number of guesses of a length-n array, is:

$$A(n) = \frac{\sum_{i=1}^{\lceil \log n \rceil} i \cdot 2^{i-1}}{n}, for\ \forall n \in \mathbb{Z}^+$$

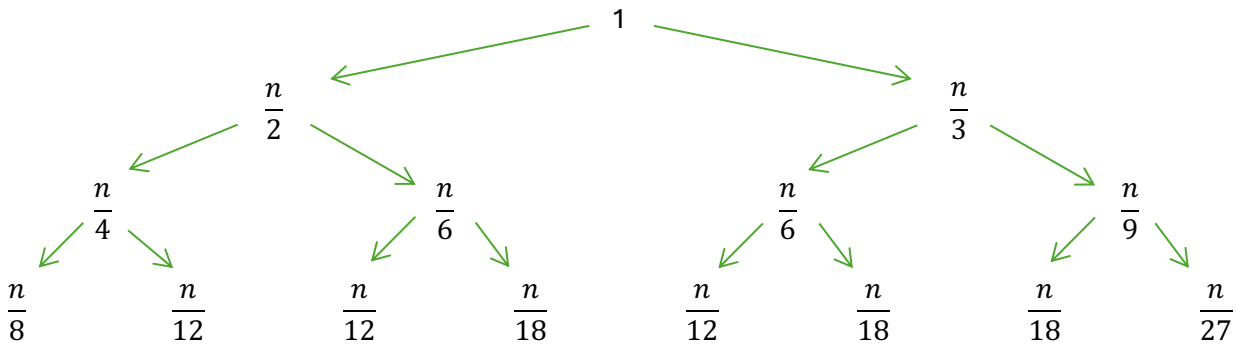
$$\therefore A(1023) = \frac{1 \times 2^0 + 2 \times 2^1 + 3 \times 2^2 + \cdots + \lceil \log 1023 \rceil \times 2^{\lceil \log 1023 \rceil}}{1023}$$

$$\therefore A(1023) = \frac{1 + 4 + 12 + \cdots + 10 \times 2^9}{1023}$$

$$\therefore A(1023) = \frac{9217}{1023}$$

Problem #4

(a)



(b)

$$T(n) = 4T(\frac{n}{3}) + 3n^2$$

(c)

$4^n = O(2^n)$ is false, and $n^{1000} = O(2^n)$ is true.

Proof by contradiction.

Assume $f(n) = 4^n$, $g(n) = 2^n$ and $f(n) = O\big(g(n)\big)$ exist, by definition we know:
$\exists\ c\ and\ n_0$ such that $0 \le f(n) \le c \cdot g(n)\ for\ \forall\ n \ge n_0$.

$\therefore 0 \le 4^n \le c \cdot 2^n$

$\therefore 0 \le (2^2)^n \le c \cdot 2^n$

$\therefore 0 \le 2^n \cdot 2^n \le c \cdot 2^n$

$\because 2^n > 0\ for\ \forall\ n$

$\therefore 0 \le 2^n \le c$

However, there will never be a c that is greater than 2^n, since 2^n grows asymptotically and polynomially faster than c. Therefore, $4^n = O(2^n)$ is false.■

Assume $f(n) = n^{1000}$, $g(n) = 2^n$ and $f(n) = O\big(g(n)\big)$ exist, by definition we know:
$\exists\ c\ and\ n_0$ such that $0 \le f(n) \le c \cdot g(n)\ for\ \forall\ n \ge n_0$.

$\therefore 0 \le n^{1000} \le c \cdot 2^n$

$\because 2^n > 0\ for\ \forall\ n$

$$\therefore 0 \le \frac{n^{1000}}{2^n} \le c$$

$\because \lim_{n\to\infty} \frac{n^a}{b^n} = 0 \; for \; a \geq 1 \; and \; b > 1$

$\therefore \lim_{n\to\infty} \frac{n^{1000}}{2^n} = 0$. In other words, when n gets very large, 2^n will always be greater than $n^{1000}, for \; some \; n > n_0$.

$\therefore \exists \; c \; and \; n_0 \; such \; that \; n^{1000} \leq c \cdot 2^n \; for \; \forall n \geq n_0$.

Therefore, $n^{1000} = O(2^n)$ is true.■

Problem #5

(a)

Loop every element after current element in the array, subtract the previous from current and store the result into a variable “max”. If the new result is greater than “max”, then update “max” with that result. Return “max” at the end. The time complexity of this brute force solution is $T(n) = O(n^2) + c$, where c stands for time assigned to initialize a variable, having two loops, etc.

(b)

```
Function FIND-MAX-PROFIT(prices):
    Define MAX-PROFIT-HELPER(prices, left, right):
        If left is equal to or greater than right:
            # It means the buying price is equal to or lower than selling price.
            # In that case, sell the stock with the same price we bought will bring the
            # best profit.
            Return 0

        # Find the mid index of the current array, which is the divide part in DAC.
        mid = ⌊(left + right) / 2⌋

        # Find the maximum profit we could have from the left half and the right half.
        left_profit = MAX-PROFIT-HELPER(prices, left, mid)
        right_profit = MAX-PROFIT-HELPER(prices, mid + 1, right)

        # Find the lowest price in the left half for us to buy,
        min_left = min(prices[left:mid + 1])
        # and the highest price in the right half for us to sell.
        max_right = max(prices[mid + 1:right + 1])

        # Find the best possible profit we could have from two halves.
        cross_profit = max_right - min_left

        # Return the max value among profit from left half, right half, and both halves.
        Return max(left_profit, right_profit, cross_profit)

    If prices is empty:
        Return 0

    Return maxProfitHelper(prices, 0, len(prices) - 1)
```

(c)

As for this algorithm, the problem is divided into two $\frac{n}{2}$ -sized sub-problems in every call, which costs $O(1)$. And within one call, there are two recursive calls for each half of the problem, which costs $2T(\frac{n}{2})$. After that, the minimum value of the left half and the maximum value of the right half would be found, which costs $2O\left(\frac{n}{2}\right) = O(n)$. Then the maximum value among maximum form the left half and the right half, with the cross profit, would be found. That costs $O(1)$. Therefore, the total efforts spend in every call, is $T(n) = 2T\left(\frac{T}{2}\right) + O(n)$.

Applying the Master Theorem to above equation.

$\because a = 2, b = 2, and\ f(n) = O(n)$ in this case, and the watershed function is $n^{\log_b a} = n$

$\therefore f(n)$ grows at the same speed as n, case 2 applies.

$\therefore T(n) = \Theta\left(n^{\log_b a} \cdot (\log n)^{k+1}\right)$ where $k = 0$,

$\therefore T(n) = \Theta(n \log n)$