Problem #1

(a)

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

(b)

Time complexity is .

Best-case scenario: the time complexity is .

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 , and the time complexity for quick sort is in the best case. Since grows faster than constant time, we could say the time complexity for above hybrid sort in the best case, is .

Average-case scenario: the time complexity is still .

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 . When the depth exceeds , we would switch to heap sort, the time complexity of which is for subarrays of length m. Therefore, the time complexity for above hybrid sort in the average case, is still .

Worst-case scenario: the time complexity is still .

In the worst case, the quick sort would picke the most unbalanced pivot, which leads to . However, since we have designed the algorithm as if the depth is exceeding then we would switch to heap sort, we could avoid such worst case, leaving the time complexity of quick sort to be . In addition, the time complexity of heap sort is 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 as for small array, and on average. Therefore, we could say that the time complexity for above hybrid sort even in the worst case, is still .

In conclusion, the time complexity for this hybrid sort, is .

(c)

Problem #3

(a)

A screenshot of a game

Description automatically generated

(b)

The time complexity of Binary Search is . So, when searching a number from 15 numbers, the target must be found within 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 , where c stands for efforts we spent on guessing the number. According to the Master Theorem which takes in the form of , we know that:

The watershed function , which grows at nearly the same asymptotic rate as .

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

in this case

Therefore, we have shown that .

(d) ?

Problem #4

(a)

(b)

(c)

is false, and is true.

Proof by contradiction.

Assume , and exist, by definition we know: such that .

However, there will never be a c that is greater than , since grows asymptotically and polynomially faster than c. Therefore, is false.

Proof by contradiction.

Assume , and exist, by definition we know: such that .

However, there will never be a c that is greater than , since grows asymptotically and polynomially faster than c. Therefore, is false.

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 , where c stands for time assigned to initialize a variable, having two loops, etc.

(b)

Function maxProfit(prices):

Define maxProfitHelper(prices, left, right):

# which means the buying price is equal to or lower than selling price.

If left is equal to or greater than right:

# 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 = maxProfitHelper(prices, left, mid)

right\_profit = maxProfitHelper(prices, mid + 1, right)

# Find the lowest price in the left half for us to buy,

# and the highest price in the right half for us to sell.

min\_left = min(prices[left:mid + 1])

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)