Problem #1

(a)

Proof by induction.

The condition in line 3 is ,

When the index of the last element is at least 2 greater than the first element, namely at least there are 3 elements in the array would we call the function recursively.

The base case is when the array contains only 1 or 2 elements.

Base case 1, when :

The subarray only contains 1 element,

The array is sorted.

Base case 2, when :

The algorithm checks if and swaps them if necessary.

After above step, the array is sorted.

Assume New-Sort correctly sorts and where , namely one third of the array’s length, and is the index of the first element. Now we only need to prove this algorithm correctly sorts as well.

is sorted,

is sorted too.

is sorted, ,

is sorted too.

is sorted, is sorted,

is sorted.

Combined with base cases, we have proved that New-Sort correctly sorts the input array .

(b)

The recurrence for the worst-case running time is where , and .

From the pseudocode we know that the New-Sort divides the array into three parts and recursively sorts the first two-thirds, the last two-thirds, and then the first two-thirds again. This gives us the following recurrence relation:

, where stands for the efforts we spend on initializing variables, etc. According to the Master Theorem, we know that the watershed function is , and .

grows asymptotically and polynomially faster than ,

Case 1 applies in this case.

.

(c)

No, the students do not deserve an A in the course, since this New-Sort is slower than insertion sort, merge sort, heapsort and quicksort.

From previous homework and learning, we know that the time complexity of insertion sort, merge sort, heapsort and quicksort in the worst case is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Heap Sort | Quick Sort |
| Time Complexity |  |  |  |  |

Problem #2

(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)