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)

There does not exist an algorithm that is guaranteed to solve this problem in 15 or

fewer matches, since any comparison sort algorithm requires comparisons in the worst case.

Since it is a problem that we sort 8 students with a comparison sort algorithm, we could formulate this problem into sorting an array with length 8 using a comparison sort algorithm. Then assume a decision tree of height with reachable leaves corresponding to such comparison sort on 8 elements.

Each of the permutations appear as one or more leaves,

.

A binary tree of height has no more than leaves,

,

,

.

,

The height will be at least 16,

There does not exist an algorithm that is guaranteed to solve this problem in 15 or

fewer matches.

(b)

The simultaneous minimum and maximum algorithm could be applied to solve the problem here. There does not exist an algorithm that is guaranteed to solve this problem in 9 or fewer matches, since at least  comparisons are necessary to find both the maximum and minimum of  numbers in the worst case.

First of all, the pseudocode for the simultaneous minimum and maximum algorithm is as follows:

Simultaneous-Min-Max(arr):

if arr is empty:

return None, None

if the length of arr is even:

min = min(arr[0], arr[1])

max = max(arr[0], arr[1])

start\_index = 2

else:

min = max = arr[0]

start\_index = 1

for i = every other number from start\_index to length of arr – 1:

if arr[i] < arr[i + 1]:

min = min(min, arr[i])

max = max(max, arr[i + 1])

else:

min = min(min, arr[i + 1])

max = max(max, arr[i])

return min, max

Secondly, from above implementation, we know that this algorithm takes every other element as a pair and compares the two elements in that pair and compares the smaller one with the current minimum and the larger one with the current maximum, which leads to 3-time comparison.

One pair consists of two elements,

This algorithm would group the array into pairs when the length is even, and pairs and 1 element left when the length is odd.

When the length is even, the number of comparisons executed by this algorithm would be 1 comparison of the first two elements, then comparisons for all the pairs, which would be in total.

When the length is odd, the number of comparisons executed by this algorithm would be 2 more comparisons among 1 element and the minimum as well as the maximum of the n – 1 array, which is .

At least  comparisons are necessary to find both the maximum and minimum of  numbers in the worst case.

The number of the students is 8,

There should be at least comparisons to determine the highest and the lowest chess rating, namely there does not exist an algorithm that is guaranteed to solve this problem in 9 or fewer matches.

(c)

A similar algorithm to above simultaneous minimum and maximum algorithm could be applied to solve the problem here. There does not exist an algorithm that is guaranteed to solve this problem in 8 or fewer matches, since comparisons are necessary to find the second maximum of  numbers in the worst case.

First, the algorithm splits all elements into pairs of 2 elements and compares them. Then, every winner from the last round would be split into pairs again and be compared with each other.

The total number of comparisons executed by this algorithm would be .

Every element except the overall winner must lose exactly 1 comparison,

There must be comparisons in total.

This algorithm will take comparisons to find the largest of elements.

There would be rounds of comparisons, and the second largest element would be compared to some element once in every round. It must be the last element that lost to the largest as well,

Except for the last comparison where the second largest lost the largest, there should be comparisons to determine the second largest element.

In total, it takes comparisons to find the second largest of elements in the worst case.

It takes at least comparisons to find the highest chess rating player and the second highest chess rating player, and there does not exist an algorithm that could solve this problem in 8 or fewer matches.

Problem #3

(a)

, counts the occurrences of elements in ,

Let be an array of length ,

0 1 2 3 4 5

.

After calculating the running time, would be:

0 1 2 3 4 5

Let be the output array. According to the counting sort we know that:

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5

,

The out put .

(b)

Since is the array of the occurrences of elements in , we know that the value in represents the last index of the same element. So similarly, we need to start backwards to be consistent with that logic. By employing that logic, the counting sort is stable.

Let us use an example to elaborate above explanation.

Let , from (a) we know , namely, the last 0 should be put at index 0 (0-index), the last 1 should be put at index 2, and the last 2 which is the blue 2 should be put at index 5. So, the output array , which provides the stability. In controversy, if we start put elements from onwards, this algorithm will become unstable.

(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)