**PAST QUESTIONS (150 Points Total):**

1. **Algorithm Understanding and Analysis (10 points)**

Given below is a recursive algorithm to reverse an array of characters.

**Reverse**(S[i…j]: nonempty character array)

C: character

1. if j – i + 1 == 1 then
2. return

else

1. C = S[i]
2. Reverse(S[i+1..j])
3. For k=(i+1) to j
4. S[k – 1] = S[k]
5. S[j] = C
6. Return

Understand the mechanics of this algorithm. To demonstrate your understanding, draw the recursion tree of the algorithm operating on the input array S=[a,b,c] in the space below. Then answer these questions:

1 If the input array contains n characters, which of the following statements is true?

1. j-i=n B. j-i-1=n C. j-i+1=n D. j-i+2=n E. None of these

2 What is the cost of single execution of step 1?

1. 5 B. 4 C. 6 D. 3 E. None of these

3 What is the approximate complexity of the loop in steps 5-6?

1. O(1) B. O(n) C. O(7n) D. O(n2) E. None of these

4 Exactly how many times are steps 5 & 6 executed?

1. n & n-1 B. n-1 & n-2 C. n+1 & n D. (n/2) & (n/2)-1 E. None of these

5 What steps are executed when the input is a base case?

1. 1 & 2 B. 3 & 4 C. 5 & 6 D. 7 & 8 E. None of these
2. **Algorithm Analysis (15 points)** This question does not require you to show your work.

**(1 point each)**

1. Exactly how many times will the loop statement “for i=n down to 3” execute?
2. 1 B. n C. n-1 D. n+1 E. None of these
3. Exactly how many times will the loop statement “for i=1 to (n-3)” execute?
4. n-2 B. n+2 C. n-1 D. n+1 E. None of these
5. Exactly how many times will the loop statement “for i=(n-3) down to 3” execute?
6. n-3 B. n+3 C. n-4 D. n+4 E. None of these

**(2 points each)**

1. Exactly how many times will the statement “do something” execute?

i=n //n: integer > 0

while i>0

do something

i=i-1

1. 1 B. n C. n-1 D. n+1 E. None of these
2. Exactly how many times will the statement “until i<0” execute?

i=n //n: integer > 0

repeat

do something

i=i-2

until i<0

1. floor(n/2) B. floor(n/2)+1 C. ceiling(n/2) D. ceiling(n/2)+1 E. None of these
2. Exactly how many times will the statement #2 execute?
3. for i=1 to (n+1)
4. for j=(m+1) down to 1
5. do something
6. (n+2)(m+2) B. n(m+1) C. (n+1)m D. (n+1)(m+2) E. None of these

**(3 points each)**

1. Calculate the T(n) for the algorithm fragment below.
2. for i=1 to n
3. for j=i to n
4. temp=temp+A[i]+A[j]

T(n)=\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Write the summation expression for the exact number of times “do something” will be executed. You do not have to expand or simplify the summation. Be sure to show the range (or limits) of each summation.
2. for i=1 to n
3. for j=1 to (n-i)
4. do something

Your answer: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Computational Problem Solving: Specification & Strategy Design (10 points)**

This question does not require you to show your work.

A well-defined computational problem specification is given below:

Input: An unsorted array A of integers with no duplicates.

Output: A non-negative integer.

Correctness Criterion: The output must equal the **maximum** of the absolute values of the differences between any two **consecutive** pairs of **integers** when the integers in the array are considered in the **descending** order.

Examples: A=[-5, -4, -3, 0, 1, 2, 3]; correct answer = |0-(-3)|=3.

A=[15, 3, 7, 1, 0, -4, 16]; correct answer = |15-7|=8.

1 What is this problem’s inherent complexity?

1. Ω(n) B. Ω(n2) C. Ω(nlogn) D. Ω(logn) E. None of these

2 Consider these two computational strategies to solve this problem:

Strategy I: Look at every possible pair of integers from the array, compute the absolute value of the difference between the two numbers in each pair, and return the maximum value found.

Strategy II: Locate the largest and the smallest integers in A and return the difference between the two, i.e., largest-smallest.

1. Strategy I is correct and Strategy II is correct.
2. Strategy I is incorrect and Strategy II is incorrect.
3. Strategy I is correct and Strategy II is incorrect.
4. Strategy I is incorrect and Strategy II is correct.
5. None of the above.

3 True or False? Circle one.

A correct and reasonably efficient strategy to solve this problem is to sort the array in the descending order and then calculate the difference between every consecutive pair of integers in the sorted array and return the maximum difference.

4 What is the best possible efficiency of an algorithm implementing the strategy outlined in Q. 3 above using any of the sorting algorithms we have discussed in class?

1. O(logn) B. O(n2) C. O(nlogn) D. O(1) E. None of these

5 Another strategy to solve this problem, without sorting, is:

1. Scan the input array A to determine the min and max integers in it.
2. Calculate the positive or negative offset needed to map the min integer to 1.
3. Copy the array A into another array B of size max-min+1 (initialized with -∞ in every cell) so that each integer in cell A[i] is copied to B[A[i]+offset]
4. Scan B from cell 1 to cell (max-min+1) to find the max difference between consecutive integer pairs, skipping over any cells with -∞.

What is the most accurate statement about this strategy?

1. This strategy is incorrect.
2. This strategy is correct.
3. This strategy is correct but less efficient than the strategy in Q. 3 above.
4. This strategy is correct and more efficient than the strategy in Q. 3 above.
5. None of these.
6. **Computational Problem Solving: Algorithm Design and Correctness (15 points)**

This question does not require you to show your work.

Consider the previous problem and the following strategy to solve this problem: Sort the array in the descending order and then calculate the difference between every consecutive pair of integers in the sorted array and return the maximum difference. Translate this strategy into an algorithm, partial pseudocode for which is given. Complete the missing steps.

**Max\_Difference**(A[p…r]: array of integer with no duplicates) returns non-negative integer

1. Merge-Sort(A) //sort A in descending order
2. Answer = -∞
3. for i=\_\_\_ to \_\_\_\_\_\_\_
4. difference=\_\_\_\_\_\_\_\_\_\_\_\_ //compute consecutive pair difference
5. if \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ then
6. Answer = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
7. return Answer

To prove that this algorithm is correct, we must show that (1) it will halt, (2) that step 1 will correctly sort A in the descending order and (3) steps 2-7 will correctly determine and return the maximum of the absolute values of the differences between every consecutive pairs of integers in the sorted array. (1) Since Merge-Sort is a called algorithm, we may assume that it will halt. Since the rest of the algorithm contains only finite steps, including a for loop with finite bounds, the algorithm will halt. (2) Since Merge-Sort is a called algorithm, we may also assume that it correctly sorts A in the descending order. (3) In order to prove that steps 2-7 will correctly determine and return the maximum of the absolute values of the differences between every consecutive pairs of integers in the sorted array, one can apply the Loop Invariant (LI) technique. The LI to use for this purpose is: Before the k-th execution of the for loop, answer will contain the maximum of the absolute values of the differences between every consecutive pairs of integers in the sorted subarray A[p…(p+k-1)].

Confirm that this is the right LI to use to prove the correctness of steps 2-7 of the algorithm by filling in the blanks and/or answering the questions below regarding a LI proof of correctness.

Initialization:

Before the 1st execution of the loop, according to the LI, answer should contain the maximum of the absolute values of the differences between every consecutive pairs of integers in the sorted subarray A[\_\_\_\_\_\_\_\_]. Since this subarray contains only one integer, the maximum of the absolute values of the differences between every consecutive pairs of integers is undefined and answer contains an undefined value – so the LI is trivially true.

Maintenance:

Before the 2nd execution of the loop, according to the LI, answer should contain the maximum of the absolute values of the differences between every consecutive pairs of integers in the sorted subarray A[\_\_\_\_\_\_]. Explain in your own words why the LI is true in this case.

Suppose Maintenance is proven (you do not have to provide the proof). I.e., suppose it has already been shown that if the LI holds true before the k-th execution of the loop, then it will be true before the next, (k+1)-th, execution of the loop.

Termination:

The loop body will execute exactly \_\_\_\_\_\_\_\_\_\_ times. Before the next execution (which will not actually happen because the loop will exist), the LI holding true implies that answer would contain the maximum of the absolute values of the differences between every consecutive pairs of integers in the sorted subarray A[\_\_\_\_\_\_\_]. This is in fact the maximum of the absolute values of the differences between every consecutive pairs of integers in the entire sorted array. Therefore the value returned by the algorithm, answer, is correct.

1. **(11 points)**

Merge Sort is an O(nlog2n) sorting algorithm because it splits the input array into two equal halves and recursively sorts each half and then merges the two halves in linear time, i.e., its recurrences are T(n<=1)=c; T(n)=2T(n/2)+cn, n>1. Suppose you modify that algorithm so it splits the array into three equal parts and recursively sorts each part and then merges the three parts in linear time. The recurrences of this new algorithm will be:

T(n<=1)=c; T(n)=3T(n/3)+cn, n>1

You can use the Recursion Tree method to solve these recurrences. As part of that, fill in the empty cells of the table below; you do not need to add up the last column and simplify it to calculate the full expression for T(n). Just by looking at the last column that you filled in, come up with an informed guess about the **complexity order** of this modified Merge Sort algorithm.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Level | Level Number | Total # of Executions at This Level | Input Size to Each Execution | Work done by each execution, excluding the recursive calls | Total work done by the algorithm at this level |
| 0 | 0 |  |  |  |  |
| 1 | 1 |  |  |  |  |
| 2 | 2 |  |  |  |  |
| The level just above the base case level | (log3n)-1 |  |  |  |  |
| Base case level | log3n |  |  |  |  |

This algorithm’s complexity order: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **(12 points)**

**Mystery(n: non-negative integer)**

1 if n==0 then return(n+1)

2 else return Mystery(n-1)

Develop and state the two recurrence relations of this algorithm (determine the integer costs; return has a cost of 1; cost of recursive calls to be stated using the function T; cost of parameter computation in recursive calls must be included):

T(0) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

T(n) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, n>0

Solve these recurrence relations using the backward substitution method

T(n) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (copied from your answer above)

T(n-1) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Substituting T(n-1) into the expression for T(n), state T(n) in terms of T(n-2) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

T(n-2) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Substituting T(n-2) into the expression for T(n), state T(n) in terms of T(n-3) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

T(n-3) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Substituting T(n-3) into the expression for T(n), state T(n) in terms of T(n-4) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Identify the pattern (the pattern need not be stated). It should be pretty obvious.

Apply the pattern to write T(n) in terms of the base case T(0)=T(n-n) and simplify.

T(n) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **(5 points)**

Solve the recurrence relations from the previous question using the forward substitution method

T(n) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (copied from your answer above)

T(1) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

T(2) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

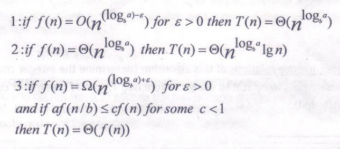
T(3) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Identify the pattern (the pattern need not be stated). It should be pretty obvious. Keeping all terms separate (not combining) may help you see the pattern.

Apply the pattern to write T(n) in terms of n.

T(n) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **(6 points) The Master Theorem:**



Apply the Master Theorem above to calculate the complexity order of the divide & conquer recursive algorithm with T(n)=16T(n/4)+n2.

a = \_\_\_\_ b = \_\_\_\_ logba = \_\_\_\_ f(n) = \_\_\_\_

The case that applies (1, 2 or 3): \_\_\_\_\_\_\_\_\_\_ T(n) = Ө(\_\_\_\_\_\_\_)

1. **(16 points)**

**Partition**(A[p…r])

1 x = A[r]

2 i = p – 1

3 for j = p to r – 1

1. if A[j] <= x
2. then i = i + 1
3. Swap A[i] and A[j]

7 swap A[i+1] and A[r]

8 return i + 1

Consider the fastest algorithm to solve the selection problem:

To find the i-th smallest number in A[p…r] containing r-p+1=n distinct numbers:

1. Base Case: if (r-p+1)==1, return A[p]. Else, divide input array A into (floor(n/5)+1) partitions of 5 numbers each – the last partition will have (n mod 5) numbers.
2. Sort each partition using Insertion Sort.
3. Determine the median of each partitions.
4. *Call itself recursively* to find the median of the (floor(n/5)+1) medians.
5. Partition A using the median of medians as the pivot (swap the last number and the median of medians and then call **Partition**).
6. Let k = size of (left partition)+1. If i==k, answer is A[i]. If i<k, *call itself recursively* on the left partition with the same i-value. If i>k, *call itself recursively* on the right partition with a new i-value I’=(i-k)

Suppose A[1…13] = [3, 6, 7, 5, 9, 12, 19, 17, 15, 16, 20, 21, 22] and i=7

Show A after steps 1 & 2 have been executed:

A = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the medians computed by the algorithm in steps 3 & 4?

Median of the first 5-sized partition of A = \_\_\_\_

Median of the second 5-sized partition of A = \_\_\_\_

Median of the remaining numbers of A = \_\_\_\_

Median of medians = \_\_\_\_

Show A after step 5 is executed:

A = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Step 6:

k = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

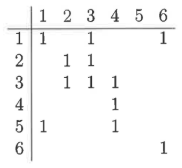
What happens next? (Circle one):

Answer is found

Algorithm recursively calls itself on the left partition

Algorithm recursively calls itself on the right partition

1. **(5 points)** Is counting sort stable? If so, prove it; if not, prove that or show a counterexample.
2. **(10 points)** For the directed graph whose adjacency matrix is given below, draw the predecessor tree that would be generated by Breadth First Search. You can assume that the adjacency list for a given node is explored in order of increasing number of the head vertex for each edge (That is, if vertex 7 has edges to vertices 10, 3, 2, and 5, those will be explored in the order 2, 3, 5, 10). As in 22.2© in the textbook, if there is an entry in row 1, col 4, that means there is a directed edge from vertex 1 to vertex 4.



1. **(10 points)** How could you implement a (non-priority) queue (FIFO) using a priority queue? Implement the ENQUEUE(n) and the n = DEQUEUE() operations using the priority queue operations defined in section 6.5 of the textbook.
2. **(10 points)** Give pseudocode to modify quick sort to sort in decreasing rather than increasing order?
3. **(15 points)** If, for a given input data set, the PARTITION routine in quicksort gave you a partition into an n-1 element subarray and a 0 element subarray every other time it was called, and two (n-1)/2-sized subarrays the other times it was called, what is the running time of quick sort with that data? Justify your answer.

**PAST ANSWERS:**

1. 1 – C.

2 – A.

3 – B.

4 – A.

5 – A.

1. 1 – C.

2 – A.

3 – C.

4 – B.

5 – B.

6 – D.

7 – 10n2 + 11n + 2

8 – (j=1 to n) (n – i)

1. 1 – A.

2 – B.

3 – True

4 – C.

5 – B.

1. Line 3 – p, r – 1

Line 4 – |A[i] – A[i+1]|

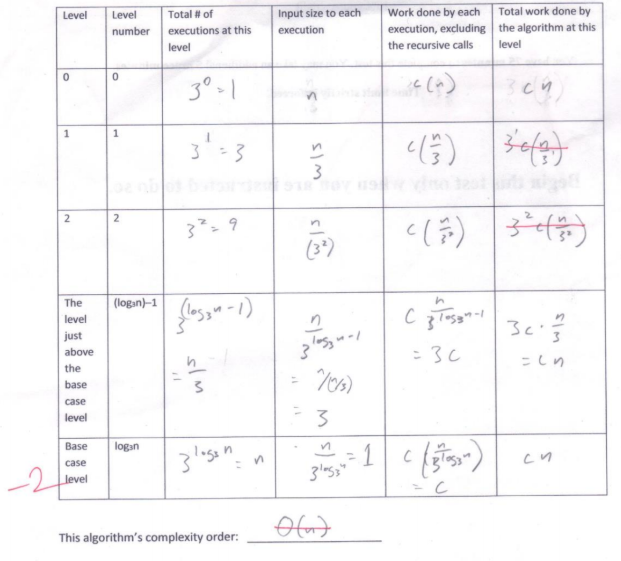
Line 5 – difference > answer

Line 6 – difference

Initialization – p…p

Maintenance – p…p+1, Before the 2nd execution of the loop the sorted subarray only has one pair sorted in descending order, so after steps 4 through 6 execute, answer must contain the absolute values of the differences since there is only one value.

Termination – r – p, p…r

1. 
2. 1st Part – 5

2nd Part – T(n-1) + 5

3rd Part – T(n-1) + 5

4th Part – T(n-2) + 5

5th Part – T(n) = (T(n-2) + 5) + 5

6th Part – T(n-3) + 5

7th Part – T(n) = (T(n-3) + 5) + 5 + 5

8th Part – T(n-4) + 5

9th Part – T(n) = (T(n-4) + 5) + 5 + 5 + 5

10th Part – T(n-n) + 5n = 5n + 5 = 5(n+1)

1. 1st Part – T(n-1) + 5

2nd Part – T(0) + 5 = 5 + 5 = 10

3rd Part – T(1) + 5 = 5 + 5 + 5 = 15

4th Part – T(2) + 5 = 5 + 5 + 5 + 5 = 20

5th Part – 5(n+1)

1. 1st Part – 16

2nd Part – 4

3rd Part – 2

4th Part – n2

5th Part – 2

6th Part – n2lgn

1. 1st Part – [3, 5, 6, 7, 9, 12, 15, 16, 17, 19, 20, 21, 22]

2nd Part – 6

3rd Part – 16

4th Part – 21

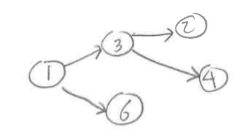
5th Part – 16

6th Part – [3, 5, 6, 7, 9, 12, 15, 16, 17, 19, 20, 21, 22]

7th Part – 8

8th Part – Algorithm recursively calls itself on the left partition

1. Counting sort is stable (the rest of the answer is not legible).



1. Add a global variable key and implement MIN-HEAP. Key = 0 initially, on attribute key

ENQUEUE(n)

MIN-HEAP-INSERT(A, n)

Key++

DEQUEUE()

Return HEAP-EXTRACT-MIN(A)

1. Only the PARTITION method needs to be modified as such:

PARTITION(A, p, r):

X = A[r]

i = p – 1

for j = p to r – 1

if A[j] >= x

i = i + 1

exchange A[i] with A[j]

exchange A[i+1] with A[r]

return i + 1

