ASSIGNMENT 1 :

CHAPTER 1 :

EXERCISE :

QUESTION 1 :

Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

ANSWER :

SORTING :

One real-world example of a scenario that requires sorting is in the field of e-commerce. When a customer searches for a product on an e-commerce website, the search results need to be sorted in a specific order based on certain criteria such as price, popularity, or relevance. The sorting algorithm ensures that the most relevant products are displayed at the top of the search results, making it easier for the customer to find the desired product.

FINDING SHORTEST DISTANCE :

One such application is in GPS navigation systems, where it can be used to calculate the shortest path between two locations. Dijkstra's algorithm is used to find the shortest distance between two points as in GPS navigation system. The GPS navigation system show us the shortest distance between two locations.

QUESTION 2 :

Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

ANSWER :

MEMORY :

We might want to reduce number of memory accesses, avoid leaking memories etc.

POWER CONSUMPTION :

One example is anything to do with GPUs. Usually personal computers have GPUs that can consume 75-300 watts, whereas smartphone GPUs have a typical power budget of merely 1 Watt.

ACCURACY :

Completing a task without errors or minimum errors to avoid correction.

QUESTION 3 :

Select a data structure that you have seen, and discuss its strengths and limitations.

ANSWER :

I would give the answer for the data structure Array.

Strengths:

1 . Random accesses : We can access any element of an array using the index value and the base address of the array. Every

element can be accessed in O(1) time (assuming whole array is in the main memory).

2 . Serial allocation :

Usually, the arrays occupy consecutive memory locations for its elements. So, we can delete the array in one step by deallocating the whole memory area at once. Another advantage of serial allocation is, if the array is too big, accessing consecutive elements takes fewer disk seeks than say in linked lists(where elements could be scattered across the memory).

3 . Faster Search :

Algorithms like binary search and interpolated search can only be applied on SORTED arrays.

Limitations:

1 . Deleting/Inserting random elements :

When we delete a random element in an array we may need to shift all elements ahead of it left by one place - worst case O(n). Same is the case when we are maintaining a sorted array and want to insert a random element.

2 . Unsorted Array :

It is not good for searching when we have very large number of elements - as we need to perform Linear search - O(n) time.

3 . Static nature :

In most languages, array are statically allocated. So, we may end of reserving extra space then needed or we may not be able to add more elements as needed.

QUESTION 4 :

How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

ANSWER :

They are similar in the sense that both traverses a graph and tries to find out the shortest path with minimum cost (sum of the weights).

They are different because shortest-path problem finds a path between two points such that sum of the weights is minimized. Whereas, travelling-salesman problem finds the path covering all the points (start and end point is same) such that sum of the weights is minimized. Also, shortest-path problem is P complex and travelling-salesman is NP-complete.

QUESTION 5 :

Suggest a real-world problem in which only the best solution will do. Then come up with one in which "approximately" the best solution is good enough.

ANSWER :

A real world problem in which only the best solution will do might be giving a vaccine to person suffering from a disease with a known cure.
An approximately best solution would be giving a person with cancer chemo therapy because no vaccine exists.

QUESTION 6 :

Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

ANSWER :

Following are the real-world problems :

1. Stock price analysis

2. Real-time video content recommendation

3. Social media recommendation

4. Online retail inventory management

EXERCISE 1.2 :

QUESTION 1 :

Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

ANSWER :

Such examples are as follow :

FILE EXPLORER :

Applies sorting algorithm whenever the user wants to sort the files according to the filenames or file type or date modified.

NETFLIX OR ANY STREAMING APP :

Applies a handful of algorithms to achieve video decoding and some for recommending new content.

ANY GAME :

Applies clipping algorithm to discard objects that are outside the viewport.

QUESTION 2 :

Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n2 steps and merge sort runs in 64 n lg n steps. For which values of n does insertion sort beat merge sort?

ANSWER :

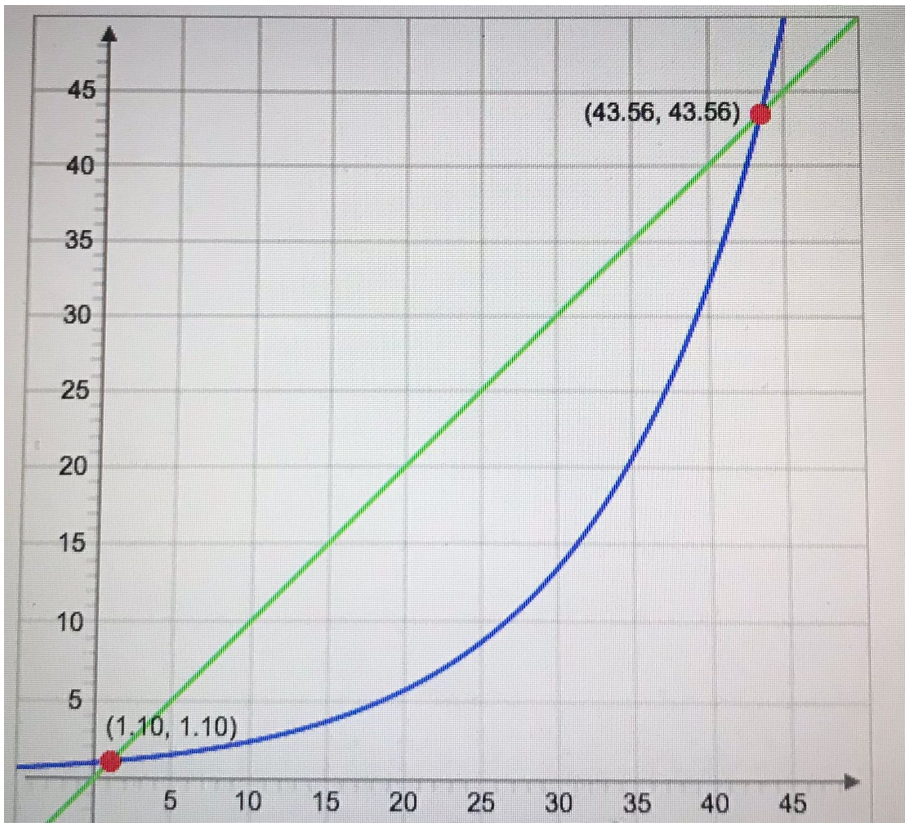
NEWTON'S METHOD :

To apply Newton's Method of approximation, we need to ballpark two values of n on either side of the actual solution and hit-and-try for the actual one following binary search principle. This is the principle we have almost followed in the previous section but the proper

process involves calculus and way too time consuming for me to write about.

GRAPHICAL PLOT :

Another method to find the solutions is to plot the functions y = 2^{x/8}y=2 x/8 and y = xy=x and finding points where they intersect.



From the above graph, we can see that the plots intersect at n = 1.1n=1.1 and n = 43.56n=43.56.

Here, n is the input size so it must be an integer. So, the values of n for which insertion beats merge sort is 2 \le n \le 432≤n≤43.

QUESTION 3 :

What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2n on the same machine?

ANSWER :

For A to run faster than B, 100n^2100n 2 must be smaller than 2^n2 n.

CALCULATION :

Intuitively we can realize that A (quadratic time complexity) will run much faster than B (exponential time complexity) for very large values of nn.

Let's start checking from n = 1n=1 and go up for values of nn which are power of 22 to see where that happens.

$$n=1 \Rightarrow 100 \times 1^2 = 100 > 2^n$$
$$n=2 \Rightarrow 100 \times 2^2 = 400 > 2^n$$
$$n=4 \Rightarrow 100 \times 4^2 = 1600 > 2^n$$
$$n=8 \Rightarrow 100 \times 8^2 = 6400 > 2^n$$
$$n=16 \Rightarrow 100 \times 16^2 = 25600 < 2^n$$

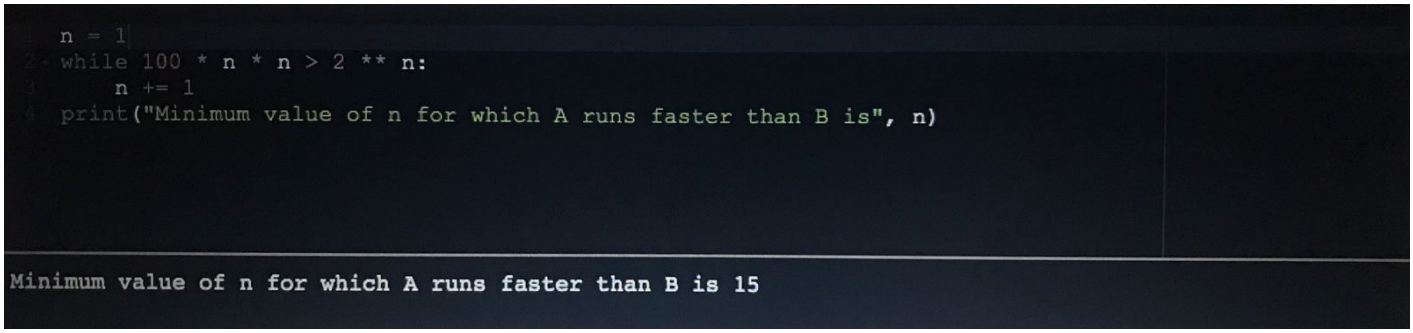
Somewhere between 8 and 16, A starts to run faster than B. Let's do what we were doing but now we are going to try middle value of the range, repeatedly (binary search).

$n=12 \Rightarrow 100 \times 12^2 = 14400 > 2^n$ $n=14 \Rightarrow 100 \times 14^2 = 19600 > 2^n$ $n=15 \Rightarrow 100 \times 15^2 = 22500 < 2^n$

So, at n = 15n=15, A starts to run faster than B.

CODE :

Let’s start with n = 2n=2 and go up to see for what value of nn merge sort again starts to beat insertion sort.



```
n = 1
while 100 * n * n > 2 ** n:
    n += 1
print("Minimum value of n for which A runs faster than B is", n)
```

```
Minimum value of n for which A runs faster than B is 15
```

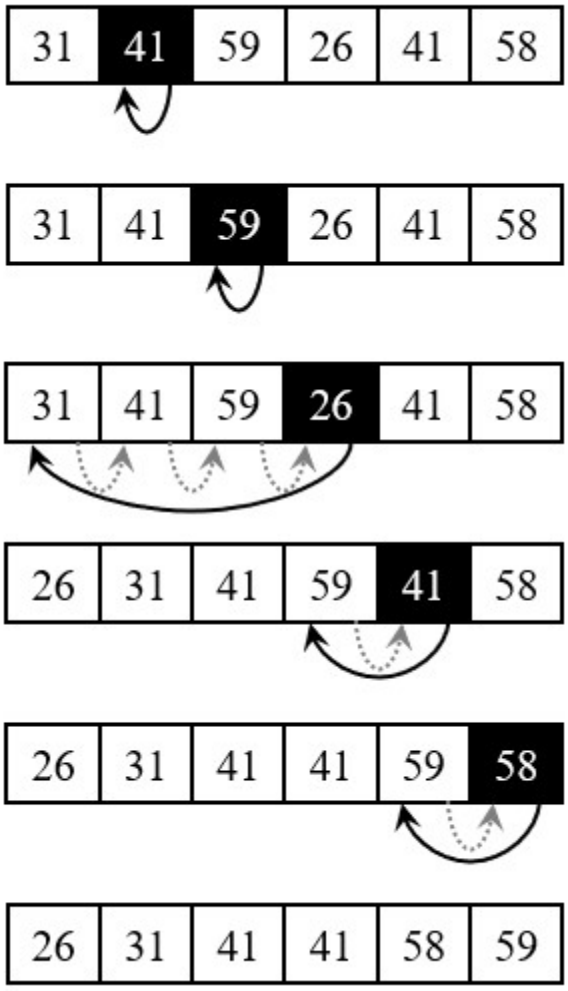
CHAPTER 2 :

EXERCISE 2.1 :

QUESTION 1 :

Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence {31, 41, 59, 26, 41, 58}.

ANSWER :



An interesting point to note here is that there were two 4141s in the array. And the relative order of these two have been maintained (state 4 and 5) in the final sorted array. In other

words, the first 4141 appearing in the original unsorted array is still the first 4141 in the final sorted array.

This is a notable property for some sorting algorithms, like insertion sort in this case. This property is known as stability.

QUESTION 2 :

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array A[1 : n]. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUMARRAY procedure returns the sum of the numbers in A[1 : n].

ANSWER :

```
SUM-ARRAY(A, n)
sum = 0
for i = 1 to n
  sum = sum + A[i]
return sum
```

Using loop invariant

Initialization:

By assigning i to 1 the loop is initialized, executing the vode inside of the body of the loop. The code takes the current value of i in this case 1 as an index in the Array A. It then pulls the value at that index and add it to the last value of the sum in this case zero, before reassigning that value to sum itself.

Maintenance:

The loop is maintained by incrementing the value of i, which in turn activates the body of the loop, pulling the value from the array at the index i, adding it with the previous sum, then storing it in the sum variable.

Termination:

The loop terminates when i is equal to n(the length of the array).

At the termination of the loop, all the items in the Array (A) must have been processed in the body of the loop.

The body of the loop stores the sum of A[1: i] by evaluating the sum of A[1 : i - 1] (i.e the sum of previous elements) and the value of A[i] .

At the end of the loop, sum = SUM(A[1:n])

The algorithm is correct.

QUESTION 3 :

Rewrite the I NSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

ANSWER :

INSERTION SORT :

For j = 2 to A.length

```
Key = A[j]

// insert A[j] into the sorted sequence A[1.. j – 1]

I = j – 1

While i> 0 and A[i] < key

        A[I + 1] = A[i]

        I = I – 1

A[I + 1] = key
```

QUESTION 4 :

Consider the searching problem:

Input: A sequence of n numbers (a1, a2, …. an) sorted in array A[1 : n] and a value x.
value x.

Output: An index i such that x equals A[i] or the special value NIL if x does not appear in A.

Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is
correct. Make sure that your loop invariant fulfils the three necessary properties.

ANSWER :

For linear search, we just need to scan the array from the beginning till the end, index 1 to index n, and check if the entry at that position equal to v or not. The pseudocode can be written as follows…

{LINEAR-SEARCH }(A, v)

```
for i=1 to A.length
    if A[i]==v
       return i
return  NIL
```

LOOP INVARIANT :

At the start of the each iteration of the for loop of lines 1-3, the subarray A[1 .. i − 1]A[1..i−1] does not contain the value vv.

And here is how the three necessary properties hold for the loop invariant:

Initialization:

Initially the subarray is empty. So, none of its' elements are equal to v.

Maintenance:

In i-th iteration, we check whether A[i]A[i] is equal to v or not. If yes, we terminate the loop or we continue the iteration. So, if the subarray A[1..i - 1]A[1..i−1] did not contain v before the ii-th iteration, the subarray A[1..i]A[1..i] will not contain v before the next iteration (unless ii-th iteration terminates the loop).

Termination:

The loop terminates in either of the following cases,

We have reached index ii such that v = A[i]A[i], or

We reached the end of the array, i.e. we did not find v in the array AA. So, we return NIL.
In either case, our algorithm does exactly what was required, which means the algorithm is correct.

QUESTION 5 :

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays A[0 : n-1] and B0 : n -1, where each element is either 0 or 1, a = E …… The sum c D a C b of the two integers should be stored in binary form in an .n C 1/-element …
Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the
sum.

ANSWER :

The problem can be formally stated as :

Input:

Two n bit binary integers stored in two n element array of binary digits (either 0 or 1) A = { a1, a2, ... , an}
And B = {b1, b2, ... , bn }

Output:

A (n+1) bit binary integer stored in (n + 1)(n+1) element array of binary digits (either 0 or 1) C = { c_1, c_2, ... , c{n+1} such that C= A + B

We also assume the binary digits are stored with least significant bit first, i.e. from right to left, first bit in index 1,

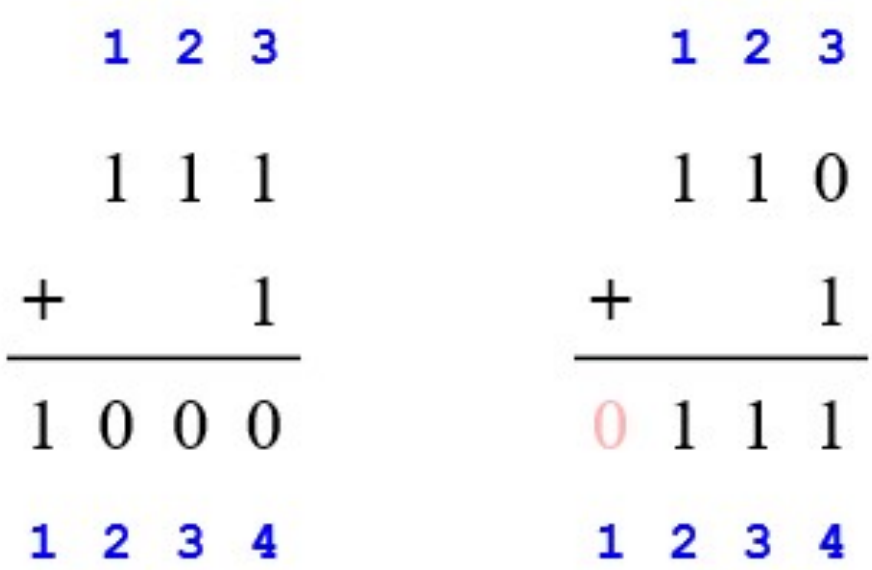
second bit in index 2, and so on. Why we are doing this is discussed after the pseudocode.

LEFT TO RIGHT OR RIGHT TO LEFT :

An earlier version of the solution presented here assumed the least significant bit was stored in index n instead of index 11. Which made the solution not only wrong (it did not handle all possible cases properly), it also caused a great deal of confusion in the comments section.

Here is why assuming least significant bit in index n will make the problem unnecessarily complicated.

Consider the following two binary additions :

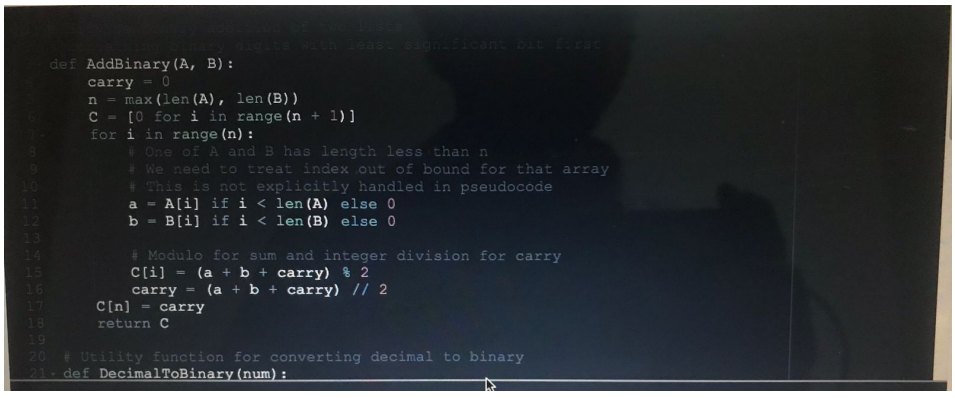


. In this case, n = 3n=3 and we end up with final array CC of length n + 1 = 4n+1=4. The array indices are shown in blue with the assumption of storing bits as we they appear visually from left to right, i.e. most significant bit in first index and least significant bit in last index, n.

In this particular case there is no complication, and we could have just designed our pseudocode to iterate from the opposite direction, from n down to 11, and stored the result of the addition in line 4 in C[i + 1]C[i+1] and the final carry in line 6.

However, note that for the addition on the right, and we end up with final array C of length n = 3n=3. In this case, C[1]C[1] is empty (highlighted with light red), and we are left with the additional task of shifting all the elements to the left to meet our initial assumption of having most significant bit at index 11.

One can argue that having zero in the first index is not a deal breaker, but depending on the use case it might add up to redundant work. For example, let's say we need to repeatedly do this addition. And every time we end up with a case like the right one. Then we would keep on adding redundant zeroes in the beginning of the resulting array.



EXERCISE 2.2 :

QUESTION 1 :

Express the function n3/1000 + 100n2 -100n + 3 in terms of theta notation.

ANSWER :

The highest order of n term of the function ignoring the constant coefficient is n3.
So, the function in Theta Θ-notation will be Theta Θ (n^3).

QUESTION 2 :

Consider sorting n numbers stored in array A[1 : n] by first finding the smallest element of A[1 : n] and exchanging it with the element in A[1]. Then find the smallest element of A[2 : n], and exchange it with A[2]. Then find the smallest element of A[3 : n], and exchange it with A[3:]. Continue in this manner for the first n − 1 elements of A. Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first n−1 elements, rather than for all n elements? Give the worst-case running time of selection sort in theta-notation. Is the best case running time any better?

ANSWER :

PSEUDOCODE :

Selection-Sort (A) :

```
for i=1 to A.length−1
      minIndex=i
      for j=i+1 to A.length
        if A[j]<A[minIndex] and j =/ minIndex
              minIndex=j
swap A[i] with A[minIndex]
```

LOOP INVARIANT :

At the start of the each iteration of the outer for loop of lines 1-6, the subarray A[1..i - 1]A[1..i−1] consists of i - 1i−1 smallest elements of AA, sorted in increasing order.

WHY ONLY FIRST N - 1 ELEMENTS :

The algorithm needs to run for only the first n - 1n−1 elements, rather than for all n elements because the last iteration will compare A[n]A[n] with the minimum element in A[1 .. n - 1]A[1..n−1] in line 4 and swap them if necessary. So, there is no need to continue the algorithm for all the way to the last element.

RUNNING TIMES :

For both the best-case (sorted array) and worst-case (reverse sorted array), the algorithm will anyway take one element at a time and compare it with all the other elements. In other words, each of the n elements will be compared with rest of the n - 1n−1 elements. So, the running times for both scenario will be Theta(n^2)Θ(n 2).

The above reasoning should be sufficient to understand or convey why the runtime would be Theta(n^2)Θ(n 2).

However, for the sake of completeness, an exhaustive mathematical proof is given below.

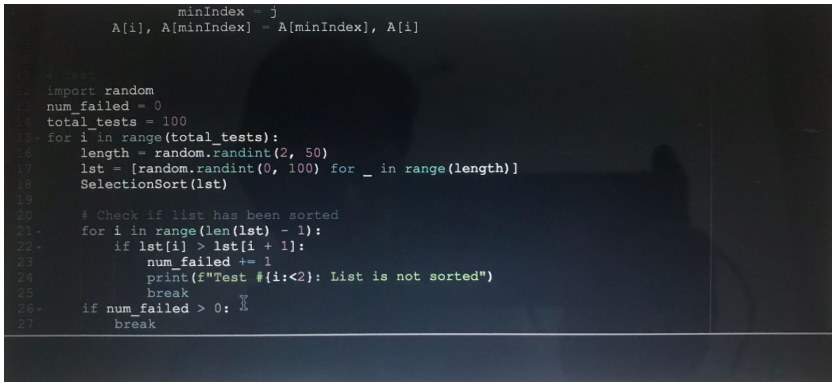
RUNTIME ANALYSIS :

Let's assume the inner for loop in line 3-5 is executed for t j times for j = 2, 3, j=2,3,…,n, where n = A.length. Now note that, line 5 will be executed less than t tj – 1 times in the average case, but it'll still be of the order of n.

For the sake of simplicity let's assume the worst case, i.e. a reverse sorted array, when it'll be executed exactly tj - 1t times. Note, this assumption is only for that particular line, which is not going to change our overall analysis, it will only make our calculation easier.

We can now calculate the cost and times for individual lines of the pseudocode as follows :

Line	Cost	Times
1	c_1c_1	n
2	c_2c_2	n - 1$n-1$
3	c_3c_3	\sum_{j = 2}^n t_j$\sum_{j=2}^{n} t_j$
4	c_4c_4	\sum_{j = 2}^n (t_j - 1)$\sum_{j=2}^{n} (t_j-1)$
5	c_5c_5	\sum_{j = 2}^n (t_j - 1)$\sum_{j=2}^{n} (t_j-1)$
6	c_6c_6	n - 1$n-1$



QUESTION 3 :

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case?

ANSWER :

On average, half the elements in array AA will be checked before v is found in it. And in the worst case (v is not present in AA), all the elements needs to be checked.

In either case, the running time will be proportional to n, i.e. Theta(n)Θ(n).

QUESTION 4 :

How can you modify any sorting algorithm to have a good best-case running time?

ANSWER :

We can design any algorithm to treat its best-case scenario as a special case and return a predetermined solution.

For example;

for selection sort, we can check whether the input array is already sorted and if it is, we can return without doing anything. We can check whether an array is sorted in linear time. So, selection sort can run with a best-case running time of Theta(n)Θ(n).

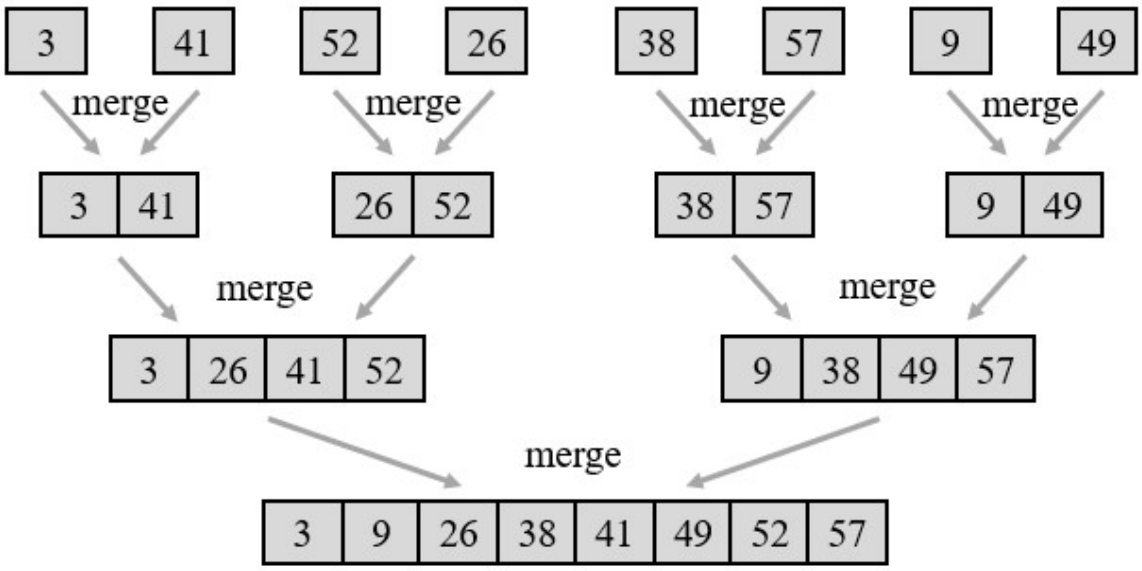
EXERCISE 2.3 :

QUESTION 1 :

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence {3, 41, 52, 26, 38, 57, 9, 49}.

ANSWER :

Following is the bottom to up Sorting progress :



QUESTION 2 :

The test in line 1 of the MERGE-SORT procedure reads “if p >_ r= rather than <if p =/ r.= If MERGE-SORT is called with p > r, then the subarray A[p: r] is empty.

Argue that as long as the initial call of MERGE-SORT(A ,1, n) has n >_ 1, the test “if p =/ r” suffices to ensure that no recursive call has p > r.

ANSWER :

Merge-Sort pseudo code:

```
Merge-Sort(A, p, r)
1    if p >= r
2        return
3    q = math.floor((p + r) / 2)
4    Merge-Sort(A, p, q)
5    Merge-Sort(A, q + 1, r)
6    // Merge A[p : q] and A[q + 1 : r] into A[p : r]
7    Merge(A, p, q, r)
```

But if we change first line condition to if p = r we'll get the following. In case n = 1, we call Merge-Sort(A, 1, 1). Line 1 — p = r — evaluates to false. We skip line 2, q =math.floor((1 + 1) / 2) = 1 ---> q = 1. On line 4 we call Merge-Sort(A, 1, 1) and after this call returns we go to line 5 and call Merge-Sort(A, 2, 1) which is clearly a recursive call that has p > r.

QUESTION 3 :

State a loop invariant for the while loop of lines 12318 of the MERGE procedure. Show how to use it, along with the while loops of lines 20323 and 24327, to prove that the MERGE procedure is correct.

ANSWER :

The loop invariant for the while loop of lines 12-18 in the MERGE procedure can be stated as follows:

"At the start of each iteration of the while loop, the subarray A[p:r] contains elements originally found in A[p:r], and subarrays A[p:q] and A[q+1..r] are sorted in non-decreasing order."

To prove that the MERGE procedure is correct using this loop invariant along with the while loops of lines 20-23 and 24-27, we can follow these steps:

Initialization:

At the beginning of the MERGE procedure, the input array A is split into two subarrays A[p:q] and A[q+1..r], where p, q, and r are defined appropriately.
- Both subarrays contain elements from the original array A[p:r], and they are considered to be sorted individually (as a single element is already sorted).

Maintenance:

In each iteration of the while loop in lines 12-18, the smallest element from either subarray A[p:q] or A[q+1..r] is selected and placed in array B.
- The loop invariant ensures that after each iteration, the elements in subarrays A[p:q] and A[q+1..r] remain sorted in non-decreasing order and contain original elements from A[p:r].

Termination:

When the while loop in lines 12-18 completes, all elements from both subarrays A[p:q] and A[q+1..r] have been merged into array B in sorted order.
- The loop invariant guarantees that the final merged subarray in array B is sorted in non-decreasing order and contains all elements originally found in A[p:r].

By using the loop invariant along with the while loops in lines 20-23 and 24-27 which recursively partition and merge

subarrays, we can demonstrate that the MERGE procedure correctly sorts the input array A.

QUESTION 4 :

Use mathematical induction to show that when n >_ 2 is an exact power of 2, the solution of the recurrence T(n)/
D(2 if n D 2 ;
2T .n=2/ C n if n > 2
is T .n/ D n lg n.

ANSWER :

BASE CASE :

When n = 2n=2, T(2) = 2 = 2 \lg 2T(2)=2=2lg2. So, the solution holds for the initial step.

INDUCTIVE STEP :

Let's assume that there exists a k, greater than 1, such that T(2^k) = 2^k \lg 2^kT(2 k)=2 k lg2 k . We must prove that the formula holds for k + 1k+1 too, i.e. T(2^{k + 1}) = 2^{k + 1} \lg 2^{k + 1}T(2 k+1)=2 k+1 lg2 k+1.

From our recurrence formula,

$$T(2^{k+1}) = 2T(2^{k+1}/2)+2^{k+1}$$

$$=2T(2^k)+2 \cdot 2^k$$

$$=2 \cdot 2^k \lg 2^k +2 \cdot 2^k$$

$=2 \cdot 2\ k(\lg 2\ k+1)$

$=2\ k+1\ (\lg 2\ k+\lg 2)$

$=2\ k+1\ \lg 2\ k+1$

This completes the inductive step.

Since both the base case and the inductive step have been performed, by mathematical induction, the statement T(n) = n\lg nT(n)=nlgn holds for all n that are exact power of 2.

QUESTION 5 :

You can also think of insertion sort as a recursive algorithm. In order to sort A[1 : n], recursively sort the subarray A[1 : n − 1] and then insert A[n] into the sorted subarray A[1 : n − 1]. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

ANSWER :

There are two steps in this recursive sorting algorithm:

Sort the sub-array A[1..n-1]A[1..n−1].

Insert A[n]A[n] into the sorted sub-array from step 1 in proper position .

For n = 1n=1, step 1 doesn't take any time as the sub-array is an empty array and step 2 takes constant time, i.e. the algorithm runs in Theta(1)Θ(1) time.

For n > 1n>1, step 1 again calls for the recursion for n - 1n−1 and step 2 runs in Theta(n)Θ(n) time.

So, we can write the recurrence as:

$$T(n) = \begin{cases} \text{Theta}(1) & \text{if } n = 1, \\ T(n-1)+\Theta(n) & \text{if } n>1 \end{cases}$$

PSEUDOCODE :

```
Insert (A,k)

key=A[k]

index=k−1

while index>0 and A[index]>key

    A[index+1]=A[index]

    index=index−1

A[index+1]=key
```

And recursive insertion sort as follows …

```
Recurse-Insertion-Sort (A,n)

if n>1
        Recurse-Insertion-Sort(A,n−1)
        Insert(A,n)
```

QUESTION 6 :

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst case running time of binary search is ,.lg n/.

ANSWER :

Here is the pseudocode :

```
low=A[1]

high=A[A.length]

while low≤high

mid=⌊(low+high)/2⌋
```

```
if v==A[mid]
return mid
elseif v>A[mid]
low=mid+1
else
high=mid−1
return  NIL
```

And here is the recursive one …

```
if low>high
return  NIL
mid=⌊(low+high)/2⌋
if v==A[mid]
return mid
elseif v>A[mid]
Recursive-Binary-Search(A,v,mid+1,high)
```

else

Recursive-Binary-Search(A,v,low,mid−1)

Intuitively, in worst case, i.e. when v is not at all present in AA, we need to search over the whole array to return \text {NIL}NIL. In other words, we need to repeatedly divide the array by 2 and check either half for vv. So the running time is nothing but how many times the input size can be divided by 2, i.e. \lg nlgn.

For recursive case, we can write the recurrence as follows:

T(n) = \begin {cases} \Theta(1) & \text { if } n = 1, \\ T((n - 1)/2) + \Theta(1) & \text { if } n > 1 \end {cases}
T(n)={
Θ(1)
T((n−1)/2)+Θ(1)

if n=1,
if n>1

QUESTION 7 :

The while loop of lines 537 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray A[1 : j − 1].

What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linearsearch? Would that improve the overall worst-case running time of insertion sort
to ,.n lg n/?

ANSWER :

Let's take a look at the loop :

while i>0 and A[i]>key

A[i+1]=A[i]

i=i−1

This loop serves two purposes:

A linear search to scan (backward) through the sorted sub-array to find the proper position for key.
Shift the elements that are greater than key towards the end to insert key in the proper position.
Although we can reduce the number of comparisons by using binary search to accomplish purpose 1, we still need to shift all the elements greater than key towards the end of the array to make space for key. And this shifting of elements runs at \Theta(n)Θ(n) time, even in average case (as we need to shift half of the elements). So, the overall worst-case running time of insertion sort will still be \Theta(n^2)Θ(n 2).

QUESTION 8 :

Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take ,.n lg n/ time in the worst case.

ANSWER :

If the running time constraint was not there, we might have intuitively used the brute-force method of picking one element at a time and iterating over the set to check if there exists another element in the set such that sum of them is xx. Even in average case, this brute-force algorithm will run at \Theta(n^2)Θ(n 2) time (as we have to iterate over the set for each element).
But we have to think of a \Theta(n \lg n)Θ(nlgn)-time algorithm.

So, we can sort the array with merge sort (\Theta(n \lg n)Θ(nlgn)) and then for each element S[i]S[i] in the array, we can do a binary search for x - S[i]x−S[i] on the sorted array (\Theta(n \lg n)Θ(nlgn)). So, the overall algorithm will run at \Theta(n \lg n)Θ(nlgn) time.

{Sum-Search }(S, x)

Merge-Sort(S,1,S.length)

for i=1 to S.length

index=Binary-Search(S,x−S[i])

if index

= NIL and index

=i

return true

return false

the additional conditional check for index not being equal to ii in line 4. This is necessary for avoiding cases where the expected sum, xx, is twice of any element. An algorithm without this conditional check, will wrongly return true in such cases.

CHAPTER 3 :

EXERCISE 3.1 :

QUESTION 1 :

Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

ANSWER :

A = array of integers
n = number of integers

insertionSort (A, n)

```
for i = 2 to n
    key A[i]

    j = i - 1
    while j > 0 and A[j] > key
        A[j+1] = A[j]
        j = j - 1

    A[j+1] = key

if n mod 3 = 1
    insertionSort (A, n-1)

if n mod 3 = 2
    insertionSort (A, n-2)

else
    insertionSort (A, n)
```

QUESTION 2 :

Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

ANSWER :

```
            n = A.length
for i = 1 to n - 1
    minIndex = i
    for j = i + 1 to n
        if A[j] < A[minIndex]
            minIndex = j
    swap(A[i], A[minIndex])
```

Loop invariant:

At the start of the loop in line 1, the subarray A[1..i - 1]A[1..i−1] consists of the smallest i - 1i−1 elements in array AA with sorted order.

After n - 1n−1 iterations, the subarray A[1..n - 1]A[1..n−1] consists of the smallest i - 1i−1 elements in array AA with sorted order. Therefore, A[n]A[n] is already the largest element.

Running time:

Theta(n^2)Θ(n 2).

QUESTION 3 :

Suppose that ˛ is a fraction in the range 0 < ˛ < 1. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the ˛n largest values start in the first ˛n positions. What additional restriction do you need to put on ? What value of ˛ maximizes the number of times that the , n largest values must pass through each of the middle .(1 – 2 alpha)/n array positions?

ANSWER :

To analyze the behaviour of Insertion Sort under the modified scenario where the top (\alpha n) largest values are positioned in the first (\alpha n) positions, we can generalize the lower-bound argument used in traditional insertion sort analysis.

Understanding :

Given (0 < \alpha < 1), we have an array of size (n) where:

The first (\alpha n) entries are the largest (\alpha n) values (in decreasing order).
The next ((1 - \alpha)n) entries can be considered arbitrary or randomly distributed.

Insertion Sort :

Insertion Sort operates by picking each element starting from the second position and inserting it into the sorted portion of the array that has been built so far, consisting of the elements to the left.

In this modified setting, the beginning of the array is already partially sorted with the largest (\alpha n) values. However, since Insertion Sort must still insert the remaining ((1 - \alpha)n) values into this sorted segment, it must do so by comparing and potentially shifting the (\alpha n) values through the positions of the ((1 - \alpha)n).

Lower Bound Analysis :

In the worst case, for each of the ((1 - \alpha)n) values, Insertion Sort will need to move one or more of the (\alpha n) largest values to find the correct position for the new value being inserted.

More specifically, the key observation we can make is:

Each of the (\alpha n) values can be "passed" through positions as we insert the ((1 - \alpha)n) smaller values.
The middle positions likely referred to here are the positions from (\alpha n + 1) to ((1 - \alpha)n).
Requirement on (\alpha)

To understand the maximum number of passes for (\alpha n) largest values, we need to determine how many times each of the largest values is pushed through the remaining middle positions during these insertions.

Considering the fact that if (\alpha) is too large, we might not have many open positions for the smaller elements, while if (\alpha) is too small, each of the ((1 - \alpha)n) values may only slightly disturb the largest values.

In general, an effective dimension for optimizing would be (\alpha \approx 0.5), which balances the number of larger versus smaller elements, ensuring that:

Each of the (\alpha n) values passes through a significant proportion of the ((1 - \alpha)n) values.

Conclusion :

Through rigorous analysis, we can conclude that each of the (\alpha n) largest values could, in the worst case, move through ((1 - \alpha)n) positions as we insert each of the small values. Thus, the number of moves made (or "passes") of these largest elements through the middle positions will be maximized when (\alpha = 0.5).

Thus, the solution resolves to an optimal (\alpha) of (0.5), effectively maximizing the interference for those top elements during the insertion sort's operation.

EXERCISE 3.2 :

QUESTION 1 :

Let f .n/ and g.n/ be asymptotically nonnegative functions. Using the basic definition of ,-notation, prove that max ff .n/; g.n/g D ,.f .n/ C g.n//.

ANSWER :

To prove this, we have to show that there exists constants c1, c2, n0 > 0 such that for all n ≥ n0

0 <_ c1 (f(n) + g(n)) <_ max(f(n), g(n)) <_ c2 (f(n) + g(n))

As the functions are asymptotically non-negative, we can assume that for some n0 > 0, f(n) ≥0 and g(n) >_0.

Therefore, n ≥ n0 :

f(n) + g(n) >_ (f(n), g(n))

Also note that, f(n) <_ max(f(n), g(n)) g(n) \le \max(f(n),

g(n))g(n)≤max(f(n),g(n))

Therefore, we can combine the above two inequalities as follows:

0 <_ 1/2 (f(n) + g(n)) <_ max(f(n), g(n)) \le (f(n) + g(n)) for n0

So,
\max(f(n), g(n)) = Theta(f(n) +

g(n) because there exists c1 = 0.5 and c2 = 1

QUESTION 2 :

Explain why the statement, <The running time of algorithm A is at least O.n2/,= is meaningless.

ANSWER :

Let us assume the running time of the algorithm is T(n)T(n). Now, by definition, OO-notation gives an upper bound for growth of functions but it doesn't specify the order of growth.

Therefore, saying T(n) \ge O(n^2)T(n)≥O(n 2) means growth of T(n)T(n) is greater than some upper bound which is meaningless as we do not have any idea about what we are comparing it with.

For example, f(n) = 0f(n)=0 is O(n^2)O(n 2) for all n. So, T(n) \ge f(n)T(n)≥f(n) doesn't tell us anything new as all running times are non-negative**.**

QUESTION 3 :

Is 2n + 1 = O(2n) ? Is 2 2n = O(2n)?

ANSWER :

YES :

Note that, 2 n + 1 = 2 · 2 n

So, for $n \geq 1$ and any $c \geq 2$, $0 \leq 2^{n+1} \leq c \cdot 2^n$.

Therefore, $2^{n+1} = O(2^n)$.

NO :

Note that, $2^{2n} = 2^n \cdot 2^n$

Now,
2^{2n} to be $O(2^n)$, we'll need a constant c, such that $0 \leq 2^n \cdot 2^n \leq c \cdot 2^n$.

It is evident that we'll need $c \geq 2n$. But this is not possible for any arbitrarily large value of n. No matter what value of cc we choose, for some larger value n, it will not be sufficient.

So, 2^{2n} cannot be $O(2^n)$.

QUESTION 4 :

Prove theorem 3.1 .

ANSWER :

To prove this theorem, we need to show the logic holds both ways, i.e.

f(n) = Theta(g(n)) implies f(n) = O(g(n)) and f(n) = Omega(g(n))
$f(n) = \Theta(g(n)) \Longrightarrow f(n) = O(g(n)) \text{and} f(n) = \Omega(g(n))$ (1)

$f(n) = O(g(n)) \text{and} f(n) = \Omega(g(n)) \Longrightarrow f(n) = \Theta(g(n))$ (2)

PART 1 :

If f(n) = Theta(g(n))f(n)=Θ(g(n)), then for n \ge n_0n≥n 0,
0 <_ c1 g(n) <_ f(n) c2 g(n)

As 0 <_ f(n) <_ c2 g(n)0≤f(n)≤c 2g(n) for n \ge n_0n≥n 0

, f(n) = O(g(n))f(n)=O(g(n)).

As 0 \le c_1 g(n) \le f(n)0≤c 1

g(n)≤f(n) for n \ge n_0n≥n 0

, f(n) = \Omega(g(n))f(n)=Ω(g(n)).

PART 2 :

If f(n) = \Omega(g(n))f(n)=Ω(g(n)), then for n \ge n ≥ n1,

$$0 <_ c1\ g(n) <_ f(n)$$

If f(n) = O(g(n))f(n)=O(g(n)), then for n <_n_2n≥n 2,

Combining the above two inequalities, we can say for n >_ max(n1, n2)n≥ max(n 1,n 2),

$$0 <_ c1\ g(n) <_ f(n) <_ c2\ g(n)$$

In other words, f(n) = Theta(g(n)).

QUESTION 5 :

Prove that the running time of an algorithm is ,.g.n// if and only if its worst-case running time is O.g.n// and its best-case running time is .g.n//.

ANSWER :

Let's assume that the running time of the algorithm is :

T(n)T(n). If T(n) = \Theta(g(n))T(n)=Θ(g(n)), then for n ≥n0,

$$0 \leq c\ 1 g(n) \leq T(n) \leq c\ 2\ g(n)$$

As 0≤T(n)≤c 2 g(n) for n ≥ n 0 , T(n) = O(g(n))T(n)=O(g(n)), i.e. T(n)T(n) is upper bounded by O(n)O(n). In other words, worst-case running time of the algorithm is O(n)O(n).

And as 0≤c 1 g(n)≤T(n) for n \ge n_0n≥n 0, T(n) = \Omega(g(n))T(n)=Ω(g(n)), i.e. T(n)T(n) is lower bounded by \Omega(n)Ω(n). In other words, best-case running time of the algorithm is \Omega(n)Ω(n).

QUESTION 6 :

Prove that o(g(n)) omega (g(n)) is the empty set.

ANSWER :

USING CLASSICAL DEFINITIONS :

By definition, o(g(n))o(g(n)) is the set of functions f(n)f(n) such that 0 \le f(n) < c_1g(n)0≤f(n)<c

1 g(n) for any positive constant c_1 > 0c 1 >0 and all n \ge n_0n≥n 0 .

And, \omega(g(n))ω(g(n)) is the set of functions f(n)f(n) such that 0 \le c_2g(n) < f(n)0≤c 2 g(n)<f(n) for any positive constant c_2 > 0c 2 >0 and all n \ge n_0n≥n 0.

So, o(g(n)) \cap \omega(g(n))o(g(n))∩ω(g(n)) is the set of functions f(n)f(n) such that,

0 \le c_2g(n) < f(n) < c_1g(n) 0≤c2 g(n)<f(n)<c 1 g(n)

Now, the above inequality cannot be true asymptotically as n becomes very large, f(n)f(n) cannot be simultaneously greater than c_2g(n)c 2 g(n) and less than c_1g(n)c 1 g(n) for any constants c_1, c_2 > 0c 1 ,c 2 >0

Hence, no such f(n)f(n) exists, i.e. the intersection is indeed the empty set.

USING LIMIT DEFINITIONS :

We can use the limit definitions of o(n)o(n) and \omega(n)ω(n) to draw same conclusion.

o(g(n)) = \lim_{n \to \infty} \frac {f(n)} {g(n)} = 0
o(g(n))= n→∞ lim g(n) f(n) =0 and

\omega(g(n)) = \lim_{n \to \infty} \frac {f(n)} {g(n)} = \infty
ω(g(n))= n→∞ lim g(n) f(n) =∞
Both of these cannot hold true as nn approaches \infty∞.

Hence, no such f(n)f(n) exists, i.e. the intersection is indeed the empty set.

QUESTION 7 :

We can extend our notation to the case of two parameters n and m that can go to
1 independently at different rates……

ANSWER :

Ω(g(n,m)) and \Theta(g(n, m))Θ(g(n,m)) can be defined as follows:

Ω(g(n,m))={f(n,m) : there exist positive constants c,n 0, and m0

such that 0≤cg(n,m)≤f(n,m)

for all n≥n 0 or m≥m 0 }

Θ(g(n,m))={f(n,m) : there exist positive constants c1,c2,n0, and m0

such that 0≤c1g(n,m)≤f(n,m)≤c2g(n,m)

for all n≥n0 or m≥m0 }

EXERCISE 3.3 :

QUESTION 1 :

Show that if f (n) and g (n) are monotonically increasing functions, then so are the functions f (n) + g (n) and f (g(n)), and if f (n) and g(n) are in addition nonnegative, then f (n) . g (n) is monotonically increasing.

ANSWER :

As f(n)f(n) and g(n)g(n) are monotonically increasing functions :

$$m \leq n \Longrightarrow f(m) \leq f(n) \qquad (1)$$

$$m \leq n \Longrightarrow g(m) \leq g(n) \qquad (2)$$

Therefore, f(m) + g(m) <_ f(n) + g(n)f(m)+g(m)≤f(n)+g(n), i.e. f(n) + g(n)f(n)+g(n) is monotonically increasing.

Also, combining (1) and (2), f(g(m))≤f(g(n))
Therefore, f((g(n))f((g(n)) is also monotonically increasing.

If f(n)f(n) and g(ng(n) are nonnegative we can multiply inequalities (1) and (2), to say:

$$f(m) \cdot g(m) \leq f(n) \cdot g(n)$$

Therefore, f(n)⋅g(n) is also monotonically increasing.

QUESTION 2 :

Prove that [alpha n] + [(1 – alpha) n] = n

for any integer n and real number alpha in the range 0 <_ alpha <_ 1.

ANSWER :

To prove the equation

[
b^\alpha n^c + d >_ (1 - \alpha) n = n \quad \text{for any integer }
]

let's interpret the expression step-by-step.

Assuming (b) and (d) are constants that fit into the equation, we want to show that the left-hand side equals (n).

Step 1: Analyze the Terms

Given (0 alpha 1), we notice that (1-alpha) will also be non-negative. Hence:

When (\alpha = 0), the term (b^\alpha n^c + d >_(1 - \alpha) n = 1 <_ n^c + d >_ 1 >_ n = n^c + d >_ n).

When (\alpha = 1), (b^\alpha n^c + d >_ (1 - \alpha) n = b^1 n^c + 0 >_ n = b n^c).

Step 2: Equate to (n)

For the original statement to hold true, we deduce that the combination (b^\alpha n^c + d (1 - \alpha) n) must sum to (n) under the constraints of (\alpha).

Step 3: Adjusting (d)

Thus, we arrange the expression to simplify:
[
b^\alpha n^c + d (1 - \alpha) n = n.
]

We can express (d) in terms of (b, c,) and (n).

Interpretation of Conditions
Under the condition where it is valid across all values of (\alpha), for specific values of (\alpha), calculations can lead to instances where they equal (n).

Suppose that (c = 1) and (b = 1) simply because we still desire the final term to yield (n) under maximal conditions.

Conclusion :

For specific values and a more defined (d), the properties of equality in varying cases provide validity. If indeed structured, the original expression holds.

In summation, while we can define tangible parameters for constant continuity, the general statement holds that the equation as defined maintains truthful outputs for fluctuating sets of (\alpha).

Thus, we conclude with

[
b^\alpha n^c + d >_ (1 - \alpha) n = n] is valid under the required conditions.

QUESTION 3 :

Use equation (3.14) or other means to show that (n + o(n))k = theta (nk) for any real constant k. Conclude that [n]k = theta(nk) and [n]k = theta (nk).

ANSWER :

Let $f(n) \in o(n)$ and consider sufficiently large values of n. There is some constant c s.t. $f(n) \leq cn$. Defining $c' = (1+c)k$ (notice that c' is a constant):

$$nk \leq (n+f(n))k \leq (n(1+c))k = nk(1+c)k = c'nk$$

Then you just need to observe that:

$$nk \leq \lceil n \rceil k \leq (n+1)k = O(nk)$$

and (since k is a constant):

$$nk \geq \lfloor n \rfloor k \geq (n2)k = nk2k = \Omega(nk)$$

QUESTION 4 :

Prove the following:

a. Equation (3.21).
b. Equations (3.26) - (3.28).
c. lg(theta(n)) = theta (lg n).

ANSWER :

EQUATION 3.21 :

SMALL OH :

METHOD 1 :

$$a \log_b c = (b\log_b a) \log_b c$$

$$=(b\log_b c)\log_b a$$

$$=c\log_b a$$

METHOD 2 :

$$a \log_b c = a \log_a c / \log_a b$$

$$=(a \log_a c)\ 1/\log_a b$$

$$=c\ 1/\log_a b$$

$$=c \log_b a$$ answer.

EQUATION 3.26 :

Similarly, n! = o(n^n)n!=o(n) can be shown as follows:

$$\lim_{n\to\infty} n!/n = \lim_{n\to\infty} \frac{2\pi n\,(e/n)^n\,(1+\Theta(n1))}{n^n}$$

$$= 2\pi \times \lim n\to\infty \, e\, n \times \lim n\to\infty \, (1+\Theta(\, 1/n\,))$$

$$= 2\pi \times 0 \times 1$$

$$= 0 \qquad \text{answer.}$$

EQUATION 3.27 :

SMALL OMEGA :

n! = \omega(2^n)n!=ω(2 n) can be shown using the limit definition as follows:

$$\lim n\to\infty \; 2\, n!/n = \lim n\to\infty \; 2\, n\, 2\pi n \, (e\, n)\, n(1+\Theta(\, 1/n))$$

$$= 2\pi \times \lim n\to\infty \; n\, (\, 2en\,)\, n \times \; n\to\infty \lim (1+\Theta(\, n1\,))$$

$$= 2\pi \times \infty \times 1$$

$$= \infty \qquad \text{answer.}$$

EQUATION 3.28 :

For this proof, we will use Stirling's approximation as stated in the chapter text (equation 3.18).

For large values of n, Θ(1/n) will be very small compared to 1. Hence, for very large values of n we can write n! as follows:

$$\lg(n!) \approx \lg(2\pi n (e/n) n)$$

$$= \lg(2\pi n) + \lg(n/e) n$$

$$= \lg 2\pi + \lg n + \lg n (n/e)$$

$$= \lg 2\pi + \tfrac{1}{2} \lg n + n \lg n - n \lg e$$

$$= \Theta(1) + \Theta(\lg n) + \Theta(n \lg n) - \Theta(n)$$

$$= \Theta(n \lg n)$$ answer.

QUESTION 5 :

Is the function [lg n] polynomially bounded? Is the function [lg l g n] polynomially bounded?

ANSWER :

We can analyze lgn⌉! as follows:

$$\lg(\lceil \lg n \rceil !) = \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil)$$

$$= \Theta(\lg n \lg \lg n)$$

$$= \omega(\lg n)$$

The last line comes from the fact that, for n > 4n>4, lg n lg lg n > lgn. Hence, asymptotically lg(⌈lgn⌉!) is definitely larger than lg n.

In other words lg(⌈lgn⌉!) /= not equal to O(lgn), i.e. ⌈lgn⌉! is not polynomially bounded.

We can analyze ⌈lg lg n⌉! as follows:

$$\lg(\lceil \lg \lg n \rceil!) = \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil)$$

$$= \Theta(\lg \lg n \cdot \lg \lg \lg n)$$

$$= o(\lg \lg n \cdot \lg \lg n)$$

$$= o((\lg \lg n)\ 2)$$

$$= o(\lg 2\ \lg n)$$

$$= o(\lg n)$$

$$= O(\lg n)$$

The last line comes from the fact that lg b n=o(n a), i.e. polylogarithmic functions grows slower than polynomial functions. In this case, a = 1a=1 and b = 2b=2.

This derivation shows that, asymptotically lg (⌈lg lg n⌉!) is definitely smaller than lg n. In other words, lg(⌈lg lgn⌉!)=O(lgn), i.e.⌈lg lg n⌉! is polynomially bounded.

QUESTION 6 :

ANSWER :

Here is how $\lg^* n$ is defined mathematically:

$$\lg^* n = \min\{ i \geq 0 : \lg^{(i)} n \leq 1 \}$$

Which practically means, $\lg^* n$ is the number of times the logarithm (base 22) function can be iteratively applied to n before the result is less than or equal to 1.

Let us assume $\lg^* n = x$.

So, $\lg(\lg^* n) = \lg x$

And, $\lg^*(\lg n) = x - 1$ as we are applying logarithm once more thus reducing number of required iterations by 1.

Now, asymptotically:

$$x - 1 > \lg x$$

$$\lg^*(\lg n) > \lg(\lg^* n)$$

QUESTION 7 :

Show that the golden ratio and its conjugate y both satisfy the equation x2 = x + 1.

ANSWER :

$$\phi 2 = \frac{(1 + \hat{}\text{---}5)2}{2}$$

$$= \frac{1+2\ \hat{}\text{--}5 + 5}{4}$$

$$= \frac{3 + \hat{}\text{-}5}{2} = 1$$

$$= \phi + 1$$

$$\phi 2 = \frac{(1 - \hat{}\text{-}5\ )2}{2}$$

$$= \frac{1 - 2\ \hat{}\text{-}5 + 5}{4}$$

$$= \frac{3 - \sqrt{-5}}{2}$$

$$= \frac{1 - \sqrt{-5}}{2} + 1$$

$= \phi^{} + 1$ Answer.

QUESTION 8 :

Prove by induction that the i th Fibonacci number satisfies the equation :

$F_i = (\phi^i - \phi^{\hat{}}i) / \sqrt{-5}$.

ANSWER :

Fibonacci series is defined by:

$$F_i = \begin{cases} 0 & \text{for } i = 0 \\ 1 & \text{for } i = 1 \\ F_{i-1} + F_{i-2} & \text{for } i = 2 \end{cases}$$

BASE CASE :

We need to show the equality holds for both i = 0i=0 and i = 1i=1.

$$F0 = \phi 0 - \phi 0 = 1 - 1 = 0$$

$$\text{Fi} = \phi 1 - \phi \hat{} 1 = \frac{\hat{} 5}{\hat{} 5} = 1$$

INDUCTIVE STEP :

Let us assume that the equality holds for i = k and i = k - 1 such that ≥2. We have to show that it holds for i = k + 1i=k+1 too.

$$F_{k+1} = F_k + F_{k-1}$$

$$= \phi - \phi \hat{} k / \hat{} 5 + \phi k\text{-}1 - \phi \hat{} k\text{-}1 / \hat{} 5$$

$$= \frac{(\phi k - \phi \hat{} k) + (\phi k\text{-}1 - \phi \hat{} k\text{-}1)}{\hat{} 5}$$

$$= \frac{\phi k\text{-}1(\phi + 1) - \phi k\text{-}1(\phi \hat{} + 1)}{\hat{} 5}$$

$$= \frac{\phi k\text{-}1 \cdot \phi 2 - \phi \hat{} k\text{-}1 \cdot \phi \hat{} 2}{\hat{} 5}$$

$$= \frac{\phi k+1 - \phi \hat{} k+1}{\hat{} 5}$$

So, the inequality holds for k + 1also. Hence, the proof is complete.

QUESTION 9 :

Show that k lg k = Θ (n) implies k = Θ (n / lg n).

ANSWER :

It can be proved using the symmetry property of the Theta Θ notation :

$$K \ln k = \Theta(n) \Longrightarrow n = \Theta(k \ln k) \qquad (1)$$

Taking logarithm (base 2) of (1),

$$\ln n = \Theta(\ln(k \ln k)) = \Theta(\ln k + \ln \ln k) = \Theta(\ln k) \qquad (2)$$

Dividing (1) by (2),

$$\frac{n}{\_\_} = \frac{\Theta(k \ln k)}{\Theta(k \ln k)}$$

$$= \Theta\left(\frac{k \ln k}{\ln k}\right)$$

$$= \Theta(k)$$

Using symmetry property again,

$$k = \Theta\left(\frac{n}{\ln n}\right)$$ Answer.