https://github.com/warishaeman28/first-repository.git

CHAPTER#1

EXERCISE#1.1

QUESTION# 1

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

ANSWER

Sorting

Definition
The arrangement of data in a preferred order is called sorting in the data structure. By sorting data, it is easier to search through it quickly and easily.
Example
The simplest example of sorting is a dictionary.

- ***Real world example of sorting***

DICTIONARY

A real-world example of sorting is a dictionary (like a glossary or a contact list). In a dictionary, words are sorted alphabetically to make it easier to locate definitions.

IMPORTANCE OF SORTING HERE

Sorting is fundamental here because it allows users to quickly search for terms by simply navigating in alphabetical order, rather than searching randomly through each entry.

OTHER EXAMPLE ARE:

Library Catalogs:

Books are sorted by genre, then alphabetically by author or title, so visitors can quickly locate titles of interest.

Employee Database in Companies:

HR systems sort employees by last name, department, or role, enabling quick access to employee information for administrative purposes.

- ***ONE THAT REQUIRE SHORTEST DISTANCE BETWEEN TWO POINTS***

Shortest Distance Example:

In a food delivery service, finding the shortest route from the restaurant to the customer's location minimizes delivery time and fuel consumption, leading to better customer satisfaction and cost savings.

Emergency Services:

Ambulances, fire trucks, and police cars rely on shortest-path calculations to reach emergency locations quickly, sometimes even taking shortcuts or restricted roads to save time.

Network Routing in Internet Traffic:

Routers use shortest-path algorithms to direct data packets between servers, minimizing delay and optimizing data flow across the network.

QUESTION#2

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

ANSWER

Accuracy:

In some areas, like predicting the weather or forecasting financial markets, accuracy is critical. It's okay if these calculations take a bit longer, because mistakes could lead to serious problems.

Maintainability:

For long-term projects, keeping code easy to understand and update is important. Well-organized code may not be the fastest, but it's easier to fix and improves reliability over time.

User Experience:

In apps and websites, good user experience might mean adding animations or visuals that slow things down a little. The focus is on making the experience enjoyable, even if it's not the fastest.

Data Consistency:

In systems that work across many computers, it's important that data stays consistent. The system must keep data up-to-date across all parts and avoid any data conflicts.

QUESTION #3

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

ANSWER

Definition

The arrangement of data in a preferred order is called sorting. By sorting data, it is easier to search through it quickly and easily.

OR

Sorting

is the process of arranging elements in a specific order, typically in ascending or descending order. Sorting is essential in computer science and various real-world applications because it allows for faster searching, better data organization, and easier analysis.

(means putting items in a specific order, like arranging numbers from smallest to largest or names alphabetically. Sorting helps make data easier to work with and understand)

Strengths of Sorting:

1. **Faster Searching improve efficiency**: When things are sorted, we can find what we need more quickly. Sorted data allows for faster searching methods, like binary search, which significantly speeds up finding specific elements.
2. **Better Organization**: Sorting makes data look neat and organized, which helps when we are trying to understand or share information.
3. **Easier to Analyze**: When items are in order, it's easier to spot patterns or compare them.
4. **Reduces Complexity in Some Algorithms**: Algorithms that rely on ordered data, such as certain search or merge algorithms, become simpler and faster when data is sorted.
5. **Simplifies Other Tasks**: Many tasks are easier to do when data is sorted, like combining two lists.

Limitations of Sorting:

1. **Time-Consuming for Big Lists**: Sorting large amounts of data can take a lot of time and computer power.
2. **Needs Extra Space**: Some sorting algorithm need extra memory, which can be a problem if there's not much available.
3. **Slows Down Real-Time Systems**: In systems where data keeps changing (like live stock prices), sorting constantly can slow things down.
4. **Choice of Sorting Algorithm Matters**: Different sorting algorithms work better with different data types and structures, and choosing the wrong algorithm can result in poor performance.

QUESTION #4

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

The **shortest-path problem** and the **traveling salesman problem** both aim to minimize the distance traveled, but they differ in purpose and complexity.

Similarities:

- Both problems involve finding optimized routes within a network or graph.
- They both seek efficiency in terms of distance or cost, aiming to reduce travel distance or time.

(Both problems involve moving through a graph and finding the path with the lowest total distance or cost (the sum of the weights).

Difference:

Shortest-path problem

- The shortest-path problem seeks the path between two specific points that has the smallest possible sum of weights, focusing on one route from start to finish.

Traveling salesman problem (TSP),

- The traveling salesman problem (TSP), however, requires visiting every point in the graph once and returning to the starting point, all while minimizing the total distance.

Additionally, the shortest-path problem is generally easier to solve (classified as P), while the TSP is much more complex (classified as NP-complete).

Question#5

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

ANSWER:

Best Solution Needed:

- **Flight Scheduling for Airlines**: Coordinating flights, gate assignments, and crew schedules requires the best solution to avoid delays, overbookings, or customer inconvenience. In this case, optimizing all variables precisely is necessary to ensure smooth operations.
- **Life-Support Systems in Healthcare**: Life-support machines, such as ventilators or heart-rate monitors, require precise algorithms to maintain accuracy.

A small deviation could have life-threatening consequences, so these systems demand exact, high-quality solutions.

Approximate Solution Acceptable:

- **Map Navigation for Driving Directions**: When suggesting a route, small inaccuracies acceptable, as the impact on travel time is minimal for most cases. Approximate or heuristic-based solutions are fast and work well for general navigation.
- **Weather Forecasting Beyond Short-Term**: For long-term forecasts, such as seasonal predictions, an approximate solution is often acceptable. While exact accuracy isn't possible due to the chaotic nature of weather systems, approximate trends or patterns provide sufficient guidance.

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:

Entire Input Available:

In a university admission system, all applicants' information is typically available after the application deadline, allowing the system to process and evaluate candidates at once.

Input Arrives Over Time:

In a stock trading platform, real-time price updates continuously arrive, requiring the system to make decisions with partial data to maximize profits or minimize losses.

- ***When all information is available upfront***

At the start of the month, the store has all the information about stock levels, planned promotions, and expected customer demand. With all this information, they can plan orders and make sure shelves are well-stocked in advance.

- ***When information arrives over time:***

During the month, unexpected things might happen, like sudden increases in demand or delivery delays. As new information comes in, the store updates its orders and adjusts stock to make sure they have enough items without overstocking.

EXERCISE #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

Application: Navigation applications (e.g., Google Maps, Waze)

Function of the Algorithms:

1. ***Route Optimization:***

Algorithms Used: Dijkstra's algorithm, A* (A-star) algorithm.

Function:

These algorithms find the shortest or fastest route from the user's current location to the desired destination by evaluating multiple paths. They consider various factors such as distance, traffic conditions, and road types to provide the most efficient route.

2. ***Traffic Prediction:***

Algorithms Used: Machine learning algorithms, regression analysis.

Function:

Navigation apps analyze historical traffic data and real-time information to predict traffic conditions. By learning from past trends, these algorithms help in estimating travel times and suggesting alternate routes when delays are expected.

3. ***Dynamic Rerouting:***

Algorithms Used: Graph algorithms and real-time data processing.

Function:

When the user is on the route, the app continuously monitors traffic and other conditions. If a better route becomes available (e.g., due to an accident ahead), the app can quickly recalculate and suggest a new route to save time.

QUESTION#2

Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n 2 steps and merge sort runs in 64n lg n steps. For which values of n does insertion sort beat merge sort?

Answer

Insertion Sort: insertion(n)= **8n^2**

Merge Sort: merge(n)= **64nlog2^n**

We want to find the values of n for which:

8n2 < 64nlog2n

Step 1: Simplify the Inequality

Dividing both sides by 8n gives:

n < 8log2n

Step 4: Test Some Values

Let's evaluate the inequality for different values of nnn:

For n=1

1 < 8log2(1)

⇒1<0(False)

For n=2

2<8log(2)

⇒2 < 8×1⇒2<8(True)

Therefore, **insertion sort beats merge sort for values of n in the range 2≤n**

QUESTION#3

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

ANSWER

100n^2 < 2^n.....eq(1)

For A to run faster than B, 100n2must be smaller than 2n

Put some value in ep 1

Let's start checking from n=1 and go up for values of n which are power of 2to see where that happens.

- **n=1:**

100(1)^2=100(1)^2 =100 and 2^1

=100>2

So 1 is not correct

- ***n*=5:**

100(5)^2< 2^5= 2500 and 2^5

=2500>32

So 5 is not correct

- ***n*=10 :**

100(10)^2<2^10

=10000>1024

So 10 is not correct

- **n=15:**

100(15)^2 < 2^15

=22500<32768

So, the smallest n for which 100n^2<2^n is **n=15.**

function f(n)*f*(*n*) and time t*t* in the following table, determine the largest size n*n* of a problem that can be solved in time t*t*, assuming that the algorithm to solve the problem takes f(n)*f*(*n*) microseconds.

Problem 1

Function	**1 second**	**1 minute**	**1 hour**	**1 day**
lgn*lgn*	21062106	26×10726×107	23.6×10923.6×109	28.64×101028.64×1
n*n*	10121012	3.6×10153.6×1015	1.3×10191.3×1019	7.46×10217.46×102
n*n*	106106	6×1076×107	3.6×1093.6×109	8.64×10108.64×101
nlgn*nlgn*	6.24×1046.24×104	2.8×1062.8×106	1.33×1081.33×108	2.76×1092.76×109

Function	1 second	1 minute	1 hour	1 day
n2*n*2	1,000	7,745	60,000	293,938
n3*n*3	100	391	1,532	4,420
2n2*n*	19	25	31	36
n!*n*!	9	11	12	13

CHAPTER#2

Exercise #1

Question #1

Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence h31;41;59;26;41;58 i .

• ***31***	• ***41***	• ***59***	• ***26***	• ***41***	• ***58***
• ***26***	• ***31***	• ***41***	• ***59***	• ***41***	• ***58***
• ***26***	• ***31***	• ***41***	• ***41***	• ***59***	• ***58***

• ***26***	• ***31***	• ***41***	• ***41***	• ***58***	• ***59***

CODE

```
Users > Shehryar > Desktop > projecy.py > ...
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Input list
arr = [31, 41, 59, 26, 41, 58]
sorted_arr = insertion_sort(arr)
print("Sorted array:", sorted_arr)
```

Output

```
PS C:\Users\Shehryar>
& C:/Users/Shehryar/AppData/Local/Programs/Python/Python312/python.exe c:/Users
Sorted array: [26, 31, 41, 41, 58, 59]
PS C:\Users\Shehryar>
```

Question#2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array AŒ1 Wn. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in AŒ1 W n.

Code

```python
def sum_array(A, n):
    total = 0
    for i in range(n):
        total += A[i]
    return total

# Test
A = [1, 2, 3, 4, 5]
n = len(A)
print("Sum of array:", sum_array(A, n))
```

Output

```
PS C:\Users\Shehryar> & C:/Users/Shehryar/AppDa
Sum of array: 15
PS C:\Users\Shehryar>
```

Question #3

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

Code

```python
def insertion_sort_desc(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] < key:  # Change '>' to '<
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

arr = [31, 41, 59, 26, 41, 58]
sorted_arr = insertion_sort_desc(arr)
print("Sorted array in descending order:", sorted_arr)
```

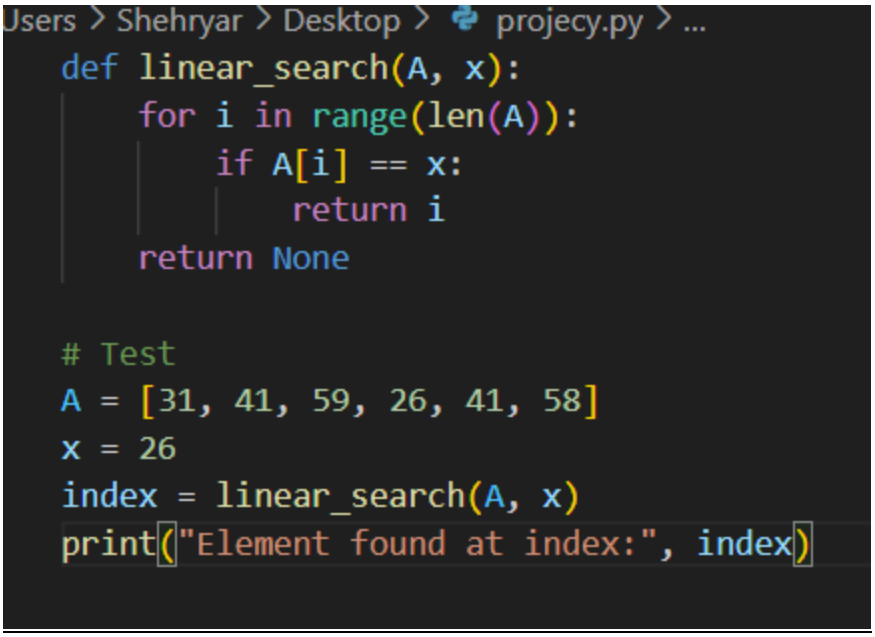
Output

```
Sorted array in descending order: [59, 58, 41, 41, 31, 26]
PS C:\Users\Shehryar>
```

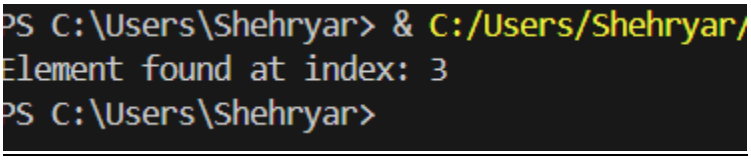
Question #4

2.1-4 Consider the searching problem: Input: A sequence of n numbers ha 1 ; a 2 ; :::; a n i stored in array AŒ1 W n] and a value x . Output: An index i such that x equals AŒi] or the special value NIL if x does not appear in A . Write pseudocode for linear search,

Code



Output



Question#5

Code

```python
def add_binary_integers(A, B, n):
    # Initialize the result array C with n+1 bits to st
    C = [0] * (n + 1)
    carry = 0

    # Iterate from the least significant bit to the most
    for i in range(n - 1, -1, -1):
        # Calculate the sum of A[i], B[i], and carry
        total = A[i] + B[i] + carry
        C[i + 1] = total % 2
        carry = total // 2
    C[0] = carry

    return C
# Test example
A = [1, 0, 1, 1]  # Binary for 11
B = [1, 1, 0, 1]  # Binary for 13
n = len(A)
result = add_binary_integers(A, B, n)
print("Sum of binary integers:", result)  # Output =24
```

Output

```
PS C:\Users\Shehryar> & C:/Users/Shehryar/A
Sum of binary integers: [1, 1, 0, 0, 0]
```

EXERCISE 2.2

Question 1

Express the function n3/1000+100n2−100n++3 in terms of Θ-notaion.

answer

Θ(n3)

Question 2

Consider sorting nn numbers stored in array A[1:n]A[1:n] by first finding the smallest element of A[1:n]A[1:n] and exchanging it with the element in A[1]A[1]. Then find the smallest element of A[2:n]A[2:n], and exchange it with A[2]A[2]. Then find the smallest element of A[3:n]A[3:n], and exchange it with A[3]A[3]. Continue in this manner for the first n−1n−1 elements of AA. 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−1n−1 elements, rather than for all nn elements? Give the worst-case running time of selection sort in ΘΘ-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 to A.length()

  if A[j]<A[minindex]

  minindex = j

  swap(A[i],A[minindex])

  loop invariant:

  At the start of ith iteration, A[1:i−1]consist of the i−1 elements of A[1:n] and it is at sort order.

subarray A[1:n−1],thebiggestelementmustbeA[1:n−1],thebiggestelementmustbeA[n]$, and then A[1:n] are sort.

- **worse case:**

in each iteration, j increase from ii to n, running time=(n)+(n−1)+...+(2)=Θ(n2)

- **best case:**

the same as worst case since in each iteration in inner for loop, j must go from i to n to get the mindindex.

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? ,-notation, give the average-case and worst-case running times of linear search. Justify your answers.

Answer

- ***average-case:***

running time∑i=1ni/n=(n+1)/2=Θ(n)running time∑i=1ni/n=(n+1)/2=Θ(n)

- ***worst-case:***

running time=n=Θ(n)running time=n=Θ(n)

Question 4

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

. Randomly Generate a Solution

Modify the sorting algorithm so that it begins by **randomly permuting** the elements of the array, AAA.

2. Check if the Solution is Correct

After the random permutation, **check if the array is sorted**. This check can be done in O(n)O(n)O(n) time.

3. Output or Proceed

- **If the array is sorted**: Output A and halt the algorithm. In this best-case scenario, the running time will only be Θ(n).
- **If the array is not sorted**: Proceed with the standard sorting algorithm, such as Selection Sort, to sort the array as usual.

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 h3;41;52;26;38;57;9;49 i .

Answer

[3|41|52|26|38|57|9|49]

↓

[3|41|52|26] [38|57|9|49]

↓

[3|41] [52|26] [38|57] [9|49]

↓

[3] [41] [52] [26] [38] [57] [9] [49]

↓

[3|26] [41|52] [38|57] [9|49]

↓

[3|26|41|52] [9|38|49|57]

↓

[3|9|26|38|41|49|52|57]

Question 2

The test in line 1 of the MERGE-SORT procedure reads "if p≥rp≥r" rather than "if p≠rp=r." If **MERGE-SORT** is called with p>rp>r, then the subarray A[p:r]A[p:r] is empty. Argue that as long as the initial call of MERGE-SORT(A, 1, n) has n≥1n≥1, the test "if p≠rp=r" suffices to ensure that no recursive call has p>rp>r.

- **inductive case:**

$$\textit{if } p \le r$$

$$\textit{if } p < r$$

$$q = \lfloor (p+r)/2 \rfloor$$

$$q \ge p$$

$$q+1 \le r$$

$$MERGE\text{-}SORT(A,p,q)$$

$$MERGE\text{-}SORT(A,q+1,r)$$

else return

no recursive call has p>r

$p \le r$ holds in new recursive call

- **base case:**

MERGE−SORT(A,1,n)

n≥1

p≤rholds

"if p≠*r*" suffices to ensure that no recursive call has p>r.

Question 3

State a loop invariant for the while loop of lines 12–18 of the MERGE procedure. Show how to use it, along with the while loops of lines 20–23 and 24–27, to prove that the MERGE procedure is correct.

Answer

- **loop invariant:**

  at the start of each iteration, A[p:k−1]A[p:k−1] consist of the smallest elements of union of LL and RR at sort order.

- when the while loop of lines 12-18 finish A[p:k−1] consist of the smallest elements of union of L and R at sort order, either while loop of lines 20–23 or 24–27 will be run, L or R has the biggest rest elements in sort order. when the second while loop finish, they are added to the end of A, and A consist of all elements at sort order.

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

$$T(n)=T(2t)$$

$$=2T(2t-1)+2t$$

$$=4T(2t-2)+2\cdot 2t$$

$$=2t-1T(2)+2t\cdot lg2t$$

$$=nlgn$$

Question 5

You can also think of insertion sort as a recursive algorithm. In order to sort AŒ1 W n , recursively sort the subarray AŒ1 W n 1 and then insert AŒn into the sorted subarray AŒ1 W n 1 . Write pseudocode for this recursive version of inser- tion sort. Give a recurrence for its worst-case running time.

Answer

```
INSERTIONSORT_RECURXIVE(A,n)
if n = 1
return
INSERTIONSORT_RECURXIVE(A,n-1)
i = n-1
key = A[n]
while A[i]>key and i>=1
A[i+1]=A[i]
i--
A[i+1]=key
```

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 $\Theta(\lg n)\Theta(\lg n)$. iterative:

Answer

```
BINARY_SEARCH_ITERATIVE(A,l,r,x)
while r>=l
mid = (l+r)/2
if A[mid] == x
return mid
else if A[mid] > x
r = mid -1
else
l = mid +1
return NIL
```

recursive:

```
BINARY_SEARCH_RECURVISE(A,l,r,x)

if(l>r)

return NIL

mid= (l+r)/2

if A[mid] == x

return x

if A[mid] < x

return A[A,mid+1,r,x]

if A[mid] > x

return A[A,l,mid-1,x]
```

he worst-case is no x in A*A*, BINARY SEARCH will end al condition l>r which occurs when A[l:r] is empty.

Question 7

The while loop of lines 5-7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray A[1:j−1]A[1:j−1]. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to Θ(nlgn)

Answer

It can save search time, but can not save movement time. At worst -case, it cost j−1=Θ(j)j−1=Θ(j) time to move elements bigger than A[j]A[j] in A[1:j−1]A[1:j−1].The same as the original INSERTION-SORT
Therefore, that improve the overall worst-case running time of insertion sort to Θ(nlgn)

Question 8

Describe an algorithm that, given a set SS of n integers and another integer xx, determines whether SS contains two elements that sum to exactly xx. Your algorithm should take Θ(nlgn) time in the worst case.

solution

```
FUCTION(S,x)

  n=S.lenfth()
```

```
MERGE_SORT(S,1,n) //cost $\Theta n \lgn$
i=1
j=n
sum=A[i]+A[j]
while sum !=x and j>i //cost \Theta n
  if sum > x
    j--
  else i++
if sum == x
  return i j
else
  return NIL
```

The time complexity of the algorithm is Θ(nlgn)+Θ(n)

Problem

solution

- **a**

Each sublists of length can be sorted in Θ(k2), there are n/k sublists, so in n/k· Θ(k2)=Θ(nk)) sorst-case time.

- **b**

Each level of merging need to merge n*n* elements, and there are lg(n/k)lg(*n*/*k*)levels(since we merge each two sublists in one in each level). So the worse-case time to merge the sublists is Θ(nlg (n/k))Θ(*n*lg(*n*/*k*)).

- **c**

ifΘ(nk+nlg(n/k))=Θ(nlgn)

∃c1,c2,n0 satisfies:

c1(nlgn) ≤ nk +nlg (n/k) ≤ c 2 (nlgn) for n > n 0

(c1−1)(lgn) ≤ k−lgk ≤ (c2−1) (lgn) for n>n0

Since lgk ⋘ k ,we have k=Θ(lgn)

- **d**

Choose k be the largest size of the sublists that **insertion-sort** is faster than **merge-sort**.

Problem 2

Problem Statement: Given an array A, we want to sort it such that after sorting, A will contain all elements in non-decreasing order.

Loop Invariant for Bubble Sort

Invariant: At the start of each inner loop iteration (lines 2-4), the subarray A[j:n] retains its original elements (in a potentially different order), with A[j] as the smallest element among them.

Initialization: When j=n, A[n] is trivially the smallest.

Maintenance: If A[j−1]>A[j]A[j-1], we swap them, making A[j−1]the smallest in A[j−1:n].

Termination: When j=i, A[i:n] holds its original elements, with A[i] as the smallest.

Loop Invariant for Sorting

Invariant: At each outer loop iteration (lines 1-4), the subarray A[1:i−1] contains the smallest i−1 elements in sorted order, while A[i:n] remains unsorted.

Initialization: A[1:i−1] is empty initially.

Maintenance: After each inner loop, A[i] is the smallest in A[i:n] , keeping A[1:i] sorted.

Termination: When i=n, A[1:n] is fully sorted.

Time Complexity

Worst-case time complexity: Each outer loop iteration (line 1-4) requires n−i iterations in the inner loop (lines 2-4), each taking constant time. Thus, the worst-case time complexity is Θ(n2)Θ(n^2)Θ(n2), similar to insertion sort.

Problem 4

a.

(1,5),(2,5),(3,4),(3,5),(4,5)

b.

⟨n,n−1,...,1⟩ totally has (n2)=n(n−1)/2 inversions

c.

The running time of insertsion sort is a constant times the number of inversions, since each time exchanging elements in insertion sort reduces Inversions num by 1.

d.

```
MERGE_INVERSIONS (A,l,r, mid)
inversions = 0
n1 = mid - l
n2 = r - mid -1
L[0:n1]
for i = 0 to n1
L[i] = A[l + i]
R[0:n2]
for j = 0 to n2
R[j] = A[mid + 1 + j]
i=0
j=0
k=l
while i <= n1 and j <= n2
if L[i] <= R[j]
A[k] = L[i]
i++
k++
else
A[k] = R[j]
```

```
j++
k++
inversions + = n1 - i + 1
while i <= n1
A[k] = L[i]
k++
i++
while j <= n2
A[k] = R[j]
k++
j++
MERGE_SORT_INVERSIONS(A,l,r)
inversions = 0
if l >=r
return inversions
mid = floor((l+r)/2)
inversions + = MERGE_SORT_INVERSIONS (A, l, mid)
inversions + = MERGE_SORT_INVERSIONS (A, mid+1, r)
inversions + = MERGE_INVERSIONS (A, l, mid ,r)
return inversions
```

Chapter 3

Exercise 1

Question 1

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

Answer

At least $\lfloor n/3 \rfloor \geq 3/n-1$

elements have to pass through at least

$\lfloor n/3 \rfloor \geq 3/n-1$

the time taken by INSERTION-SORT in the worst case is at least proportional to

$(3n-1)(3n-1)=\Omega(n2)$

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

Inner loop in selection sort iterates

$(n-i+1)$ times, for i = 1 to n-1,

so the running time of selection sort is

$\sum_{I=1}^{n-1}(n-i+1) =(n+2)\ (n-1)/2=\Theta(n2))$

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 ûrst ¸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¸/n array positions***?***

Answer

At least ααn values have to pass through at least $(1-2\alpha)n$ times. insertion sort is at least proportional to $(\alpha n)(1-2\alpha)n$

$=\alpha(1-2\alpha)n^2$

$=\Omega(n^2)$

if $\alpha(1-2\alpha)=\Omega(1)$

$\max(\alpha(1-2\alpha))=1/8$ $\max(\alpha(1-2\alpha))=1/8$ when $\alpha=1/4$

Exercise 3.2

Question 1

Let f(n)$f(n)$ and g(n)$g(n)$ be the asympotically nonnegative functions. Using the basic definition of ΘΘ-notation, prove that max f(n),g(n)=Θ(f(n)+g(n))$f(n),g(n)=\Theta(f(n)+g(n))$.

Answer

Let n0 be a value that makes f(n) greater than 0 for n >n0

Then for n>n0

1/2(f(n)+g(n)) ≤max(f(n), g(n)

max(f(n), g(n)) =Θ(f(n)+g(n))

Question 2.

Explain why the statement, "The running time of algorithm A is at least O(n2) $O(n2)$,"if meaningless.

Answer

A=O(n2) $A=O(n2)$ means ∃n0, c: A≤cn2∃$n0$

A is at least O(n)$O(n)$ means A ≥ a function that ≤ cn2$A\geq$

You can say A can be any function or no function. Both are meaningless.