Name : Sardar Hassan Aftab

Roll no : 14873

Subject : DSA

Semester : 3

Section : D

Chapter : 1

The Role of Algorithms in Computing

Exercises

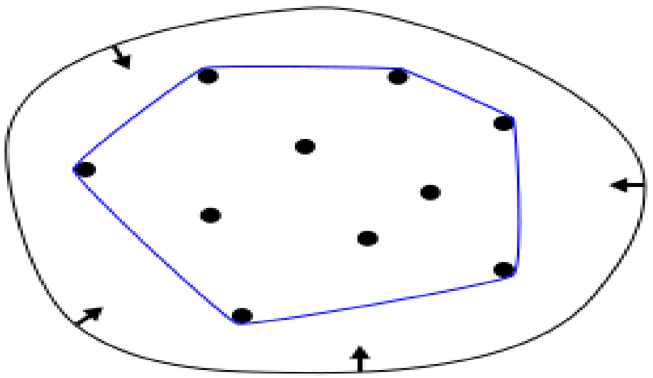
EX : 1.1-1

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

Solution:

The contact book in our phones needs to be sorted alphabetically so that we can find person phone number easily. Some other examples are flights, movie tickets they are all sorted by time. Even our daily schedules are sorted by time.

Convex hull can be defined as a step of points, a line connecting the points in a convex polygonic way, so that all the points are inside it.



In image processing application Convex hull is used.

Ex : 1.1-2

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

Solution :

Memory :

This is probably the most obvious one other than speed. Not only we want to utilize available memory efficiently, we might want to reduce number of memory accesses, avoid leaking memories etc.

One example is learning based approaches often needs to work with very large amount of data. If the dataset is larger than available system RAM, we will have to design our algorithm to be able to run out of core.

Power Consumption :

This is particularly relevant when we are developing our algorithm for portable devices like smartphones or smartwatches etc. In such sometimes we want to settle for relatively less power hungry algorithm algorithm that does the job good enough instead of going with more power consuming complex algorithms that produce better results.

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.

Ex : 1.1-3

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

Solution :

I would give the answer for the data structure Array.

Strengths of Array :

1. Accessing any element by index is simple, O(1)$O(1)$ time complexity
2. No additional memory required to store address

Limitations of Array:

1. Addition or removal of elements from any index but the last means re-arranging the whole list, O(n)$O(n)$ time complexity
2. Accessing an element by value means traversing the list, O(n)$O(n)$ time complexity
3. Needs contiguous memory.

Ex : 1.1-4

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

Solution :

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.

Ex : 1.1-5

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

Solution :

Taking the example of sorting…

Best sorting algorithm is required when we need to sort a huge amount of data as soon as possible. For example, Sorting blog posts by date or items in an online store by price.

“Approximately” best sorting algorithm will work when we need to sort a small amount of data and/or without any time constraint. For example, analyzing data of a medical survey to make a conclusion after few days. Or finding out the ranking in a competition where 5 people participated.

Ex : 1.1-6

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 ?

Solutuon :

A real-world example is **package delivery routing** for companies like UPS or Amazon.

In the **static case**, the company can plan the most efficient routes in advance, minimizing travel time and fuel costs since all delivery information is known upfront. Algorithms like the Traveling Salesperson Problem (TSP) can be used to generate optimal routes.

In the **dynamic case**, routes need to be continuously updated as new orders or changes come in throughout the day. The challenge is to maintain efficiency while being flexible enough to adapt to unexpected or last-minute deliveries, requiring real-time decision-making algorithms.

EX : 1.2-1

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

Solution :

If you think about it, such examples are everywhere these days, much more than when CLRS was first published (1990). Here are few such examples…

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.

Ex : 1.2-2

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?

Solutuon :

For insertion sort to beat merge sort for inputs of size *n*, 8n^2 must be less than 64 nlgn .

$$8n^2 < 64\, n \lg n$$

$$= \cancel{8n}.n < \cancel{8n} . 8 \lg n$$

$$= n \quad < 8 \lg n$$

$$= n/8 \; < \lg n$$

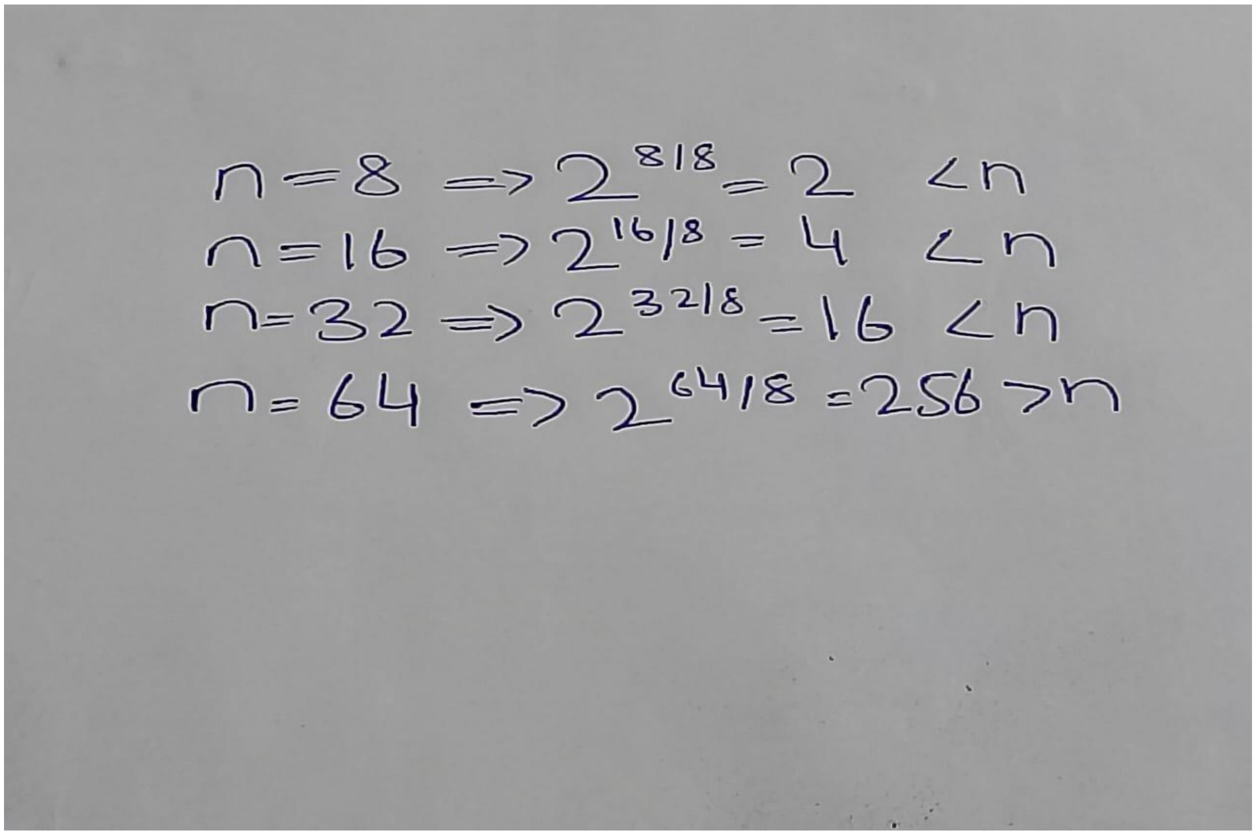
$$2^{n/8} < n$$

This is not a purely polynomial equation in n. To find the required range of values of n, there are a few different methods we can use…

Calculation

It is obvious that insertion sort runs at quadratic time which is definitely worse than merge sort's linearithmic time for very large values of n*n*. We know for n=1, merge sort beats insertion sort. But for values greater than that, insertion sort beats merge sort. So, we will start checking from n=2 and go up to see for what value of n merge sort again starts to beat insertion sort.

Notice that for n<8, 2^n/8 will be a fraction. So, let's start with n=8*n*=8 and check for values of n*n* which are powers of 2.



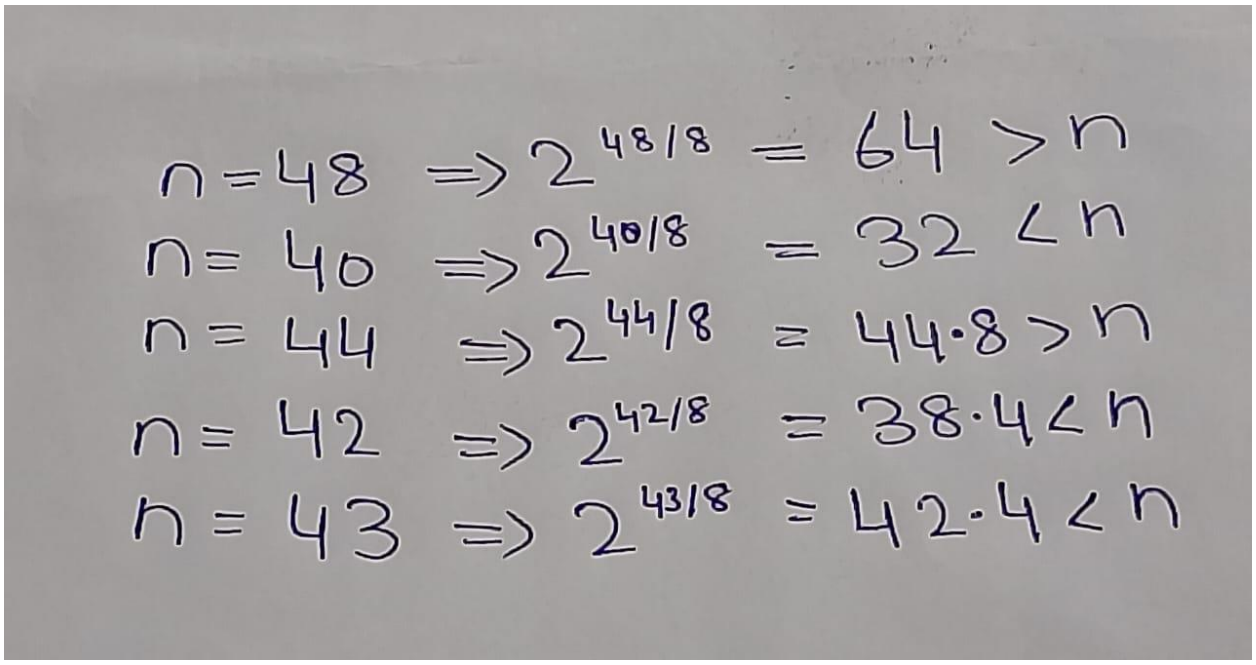
$n = 8 \Rightarrow 2^{8/8} = 2 < n$

$n = 16 \Rightarrow 2^{16/8} = 4 < n$

$n = 32 \Rightarrow 2^{32/8} = 16 < n$

$n = 64 \Rightarrow 2^{64/8} = 256 > n$

Note that we don’t need to continue anymore as we have found an approximate range of values for nn where merge sort starts to beat insertion sort; somewhere between 32 and 64. Let’s do what we were doing but now we are going to try middle value of either range, repeatedly (or in other words, binary search, if you have been reading ahead).



$n = 48 \Rightarrow 2^{48/8} = 64 > n$

$n = 40 \Rightarrow 2^{40/8} = 32 < n$

$n = 44 \Rightarrow 2^{44/8} = 44.8 > n$

$n = 42 \Rightarrow 2^{42/8} = 38.4 < n$

$n = 43 \Rightarrow 2^{43/8} = 42.4 < n$

So, at n=44, merge sort starts to beat insertion sort again. Therefore, for $2 \leq n \leq 43$, insertion sort beats merge sort.

Ex : 1.2-3

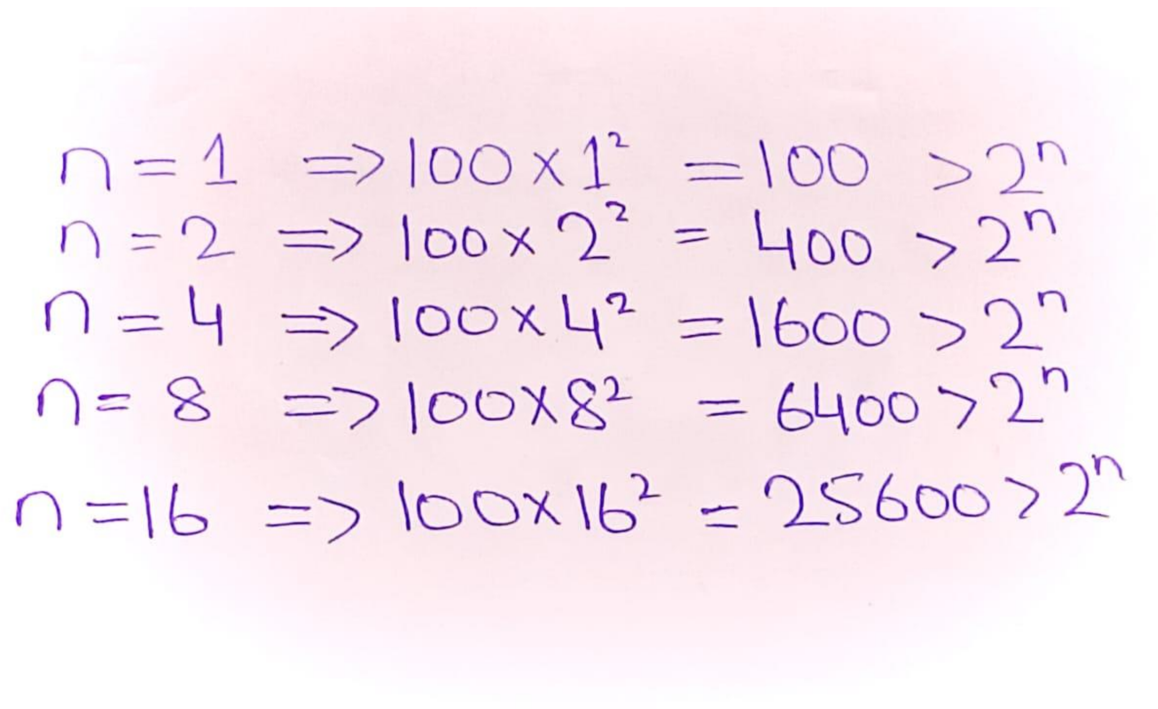
What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2 n on the same machine?

Solution :

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

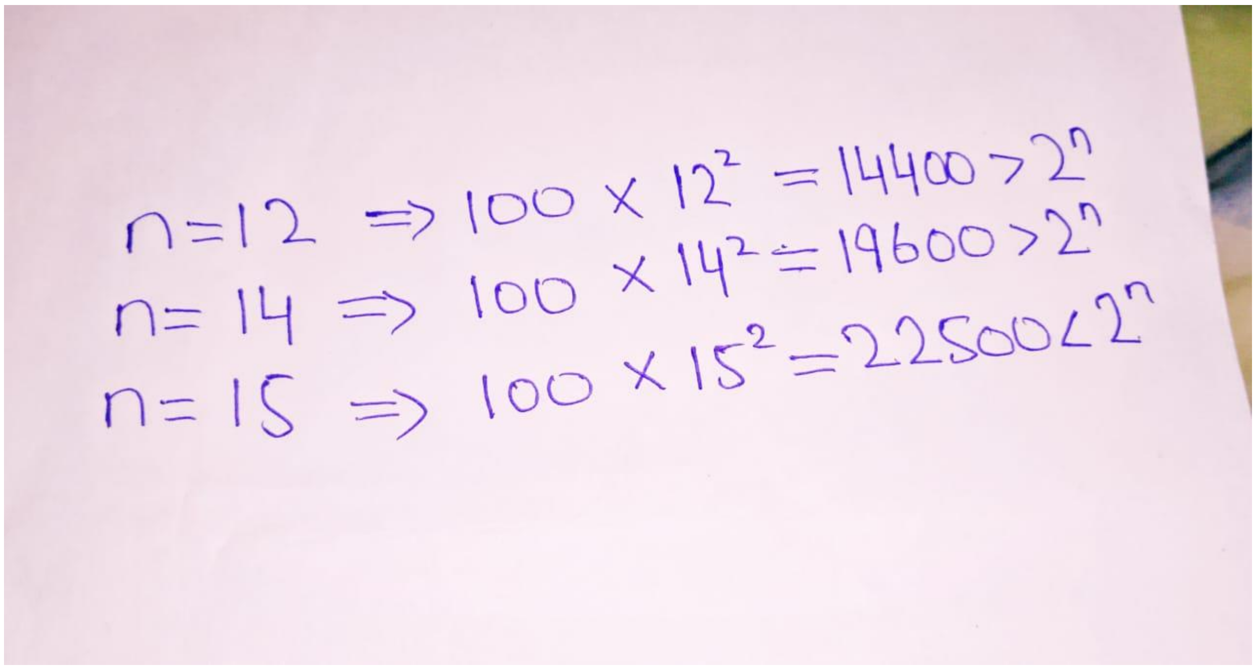
Intuitively we can realize that A (quadratic time complexity) will run much faster than B (exponential time complexity) for very large values of n*n*.

Let's start checking from n=1*n*=1 and go up for values of n*n* 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=15*n*=15, A starts to run faster than B.

Chapter : 2

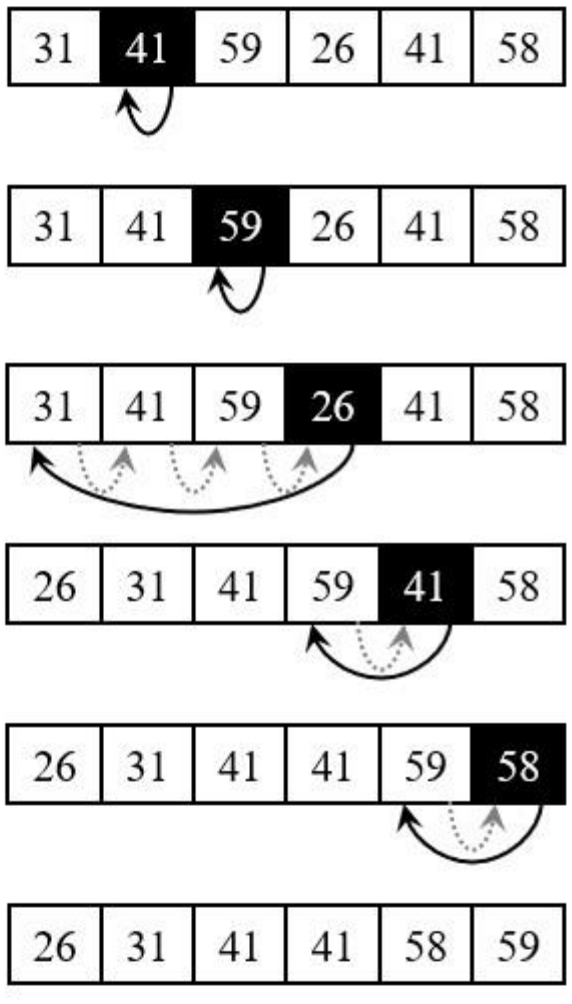
GETTING STARTED

EX :2.1-1

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; 58i.

Solution :

Using Figure 2.2 as a model, illustrate the operation of Insertion-SortInsertion-Sort on the array A=⟨31,41,59,26,41,58⟩.



EX : 2.1-2

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 [1: n]

Solution :

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

The procedure accepts and Array (A) and the length of the array(n) 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.

Ex : 2.1-3

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

Solution :

We just need to reverse the comparison of A[i] and key in line 5 as follows…

Insertion-Sort (A)

```
1   for j=2 to A.length
2       key=A[j]
3       // Insert A[j] into the sorted sequence A[1 .. j−1]
4       i=j−1
5       while i>0 and A[i]<key
6           A[i+1]=A[i]
7           i=i−1
8       A[i+1]=key
```

Ex : 2.1-4

Consider the searching problem:

Input: A sequence of n numbers (a^1 , a^2 …..a^n) stored in array A[1:n] and a 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 fulûlls the three necessary properties.

Solution :

For *linear search*, we just need to scan the array from the beginning till the end, index 11 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*)

```
1  for i=1 to A.length
2      if A[i]==v
3          return i
4  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] does not contain the value v.

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] 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] did not contain v before the i-th iteration, the subarray A[1..i] will not contain v before the next iteration (unless i-th iteration terminates the loop).

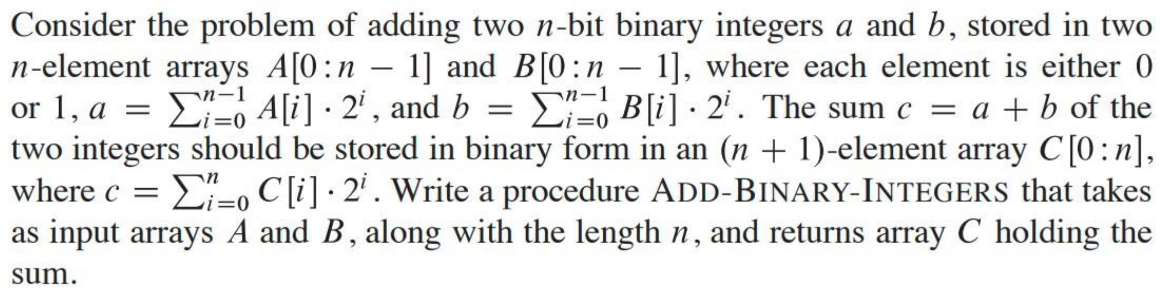
Termination:

The loop terminates in either of the following cases,

- We have reached index i*i* such that v = A[i], or
- We reached the end of the array, i.e. we did not find v in the array A. So, we return NIL.

In either case, our algorithm does exactly what was required, which means the algorithm is correct.

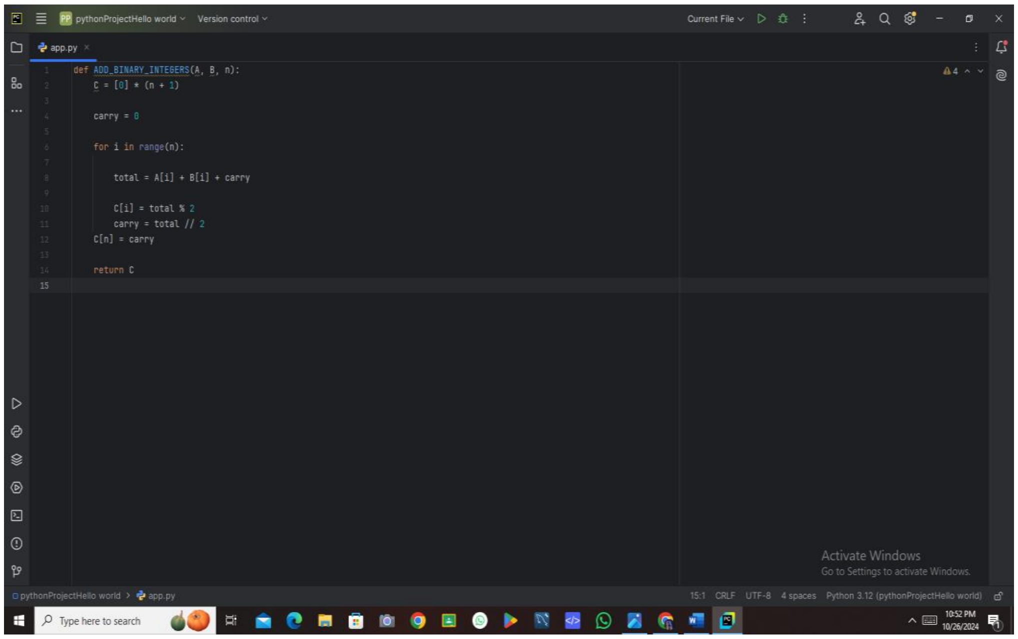
Ex :2.1-5



Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. 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.

Solution :

Here's a procedure for adding two n-bit binary integers stored in arrays A and B, producing the result in array C. This algorithm handles the carry that may result from adding corresponding bits.



```python
def ADD_BINARY_INTEGERS(A, B, n):
    C = [0] * (n + 1)

    carry = 0

    for i in range(n):

        total = A[i] + B[i] + carry

        C[i] = total % 2
        carry = total // 2
    C[n] = carry

    return C
```

1 Initialization:

- An array C of size n+1 is created to hold the sum.
- A carry variable is initialized to zero.

2 Bit Addition:

- The loop iterates from 0 to n-1, which corresponds to the indices of the bits in A and B.
- For each bit index i, it computes the sum of the corresponding bits in A and B along with the carry.
- The result for each bit is stored in C[i], and the new carry is calculated for the next higher bit.

3□ Final Carry:

- After the loop, any remaining carry is stored in C[n], which accounts for an overflow (e.g., when both A and B are the maximum possible n-bit values).

Exercise 2.2-1

Express the function n^3 /1000−100n^2−100 n+3 in terms of Θ-notation.

Solution :

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

Exercise 2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A, and exchange it with A[2]. 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 best-case and worst-case running times of selection sort in Θ-notation.

Solution :

Pseudocode

```
Selection Sort (A)

1  for I =1 to A . length−1

2      min Index=I

3      for j=i+1 to A . length

4          if A[j]<A[min Index] and j≠min Index

5              min Index = j

6   swap A[i] with A[minIndex]
```

Loop Invariant

The loop invariant for this algorithm is: *At the start of each iteration of the outer loop (for each index iii), the subarray A[1] to A[i−1] is sorted and contains the i−1 smallest elements of the original array.*

Explanation for n−1n-1n−1 Elements

The algorithm only needs to run for the first n−1 elements because, after sorting the first n−1 elements, the last element will automatically be in the correct position. If the first n−1 elements are sorted, the largest remaining element (which will be the n-th element) must be the maximum of the entire array and will naturally be in its correct position.

Running Times in Θ-Notation

- **Best-Case Time Complexity**: Θ(n^2)
  Even in the best case (when the array is already sorted), the algorithm still examines all elements to ensure the smallest element is in the correct position, resulting in n(n−1)/2 comparisons.
- **Worst-Case Time Complexity**: Θ(n^2)
  In the worst case (when the array is in reverse order), the algorithm will perform the same number of comparisons and swaps, leading to the same complexity.

Thus, the overall time complexity for Selection Sort is Θ(n^2) in both the best and worst cases.

Ex : 2.2-3

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?

Solution :

In a linear search, on average n/2 **elements** need to be checked; 'n' being the total number of elements in the array. The worst-case scenario would require checking all 'n' elements. Both average-case and worst-case running times are Θ(n), reflecting the fact the runtime grows linearly with the size of the input sequence.

Explanation:

In a **linear search**, each element of the input sequence needs to be checked until the desired element is found. On average, assuming that the element being searched for is equally likely to be any element in the array, you would, on average, need to check n/2 elements, where 'n' represents the number of elements in the array.

In the worst case scenario, the desired element could be the last element in the sequence, or it may not be in the sequence at all. In this case, you would need to check all 'n' elements.

In terms of the running times, the average-case running time of linear search is Θ(n), because it's directly proportional to the size of the input, and on average, you are likely to inspect half of the elements. This holds more for larger arrays.

The worst-case running time is also Θ(n), because in the worst-case scenario, all elements must be checked

Ex : 2.2-4

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

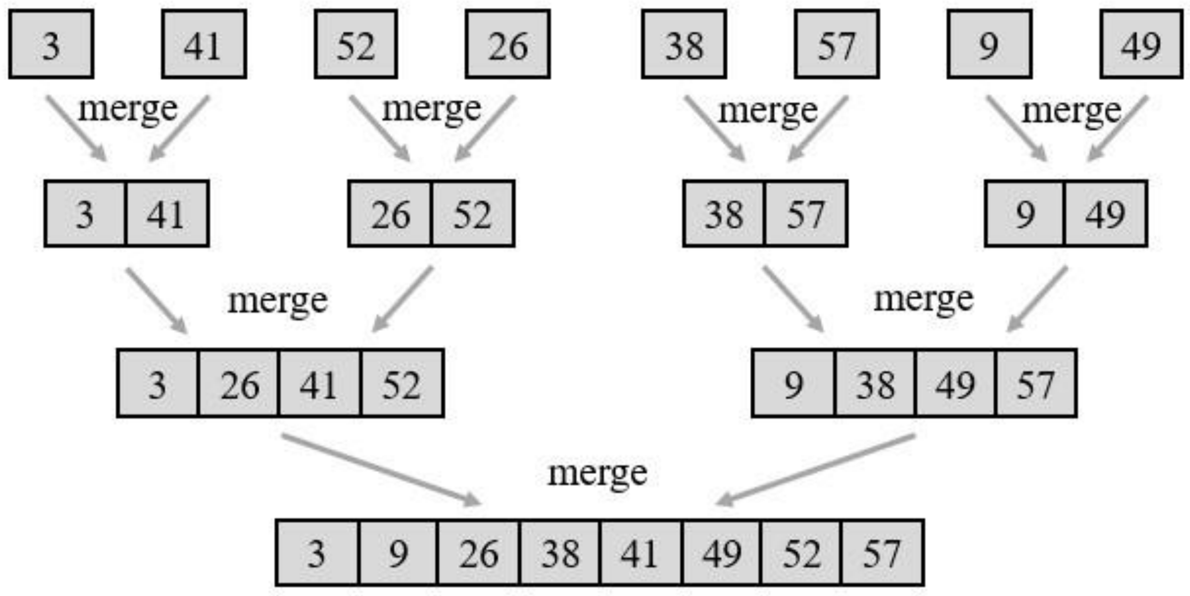
Solution :

For a good best-case running time, modify an algorithm to first randomly produce output and then check whether or not it satisfies the goal of the algorithm. If so, produce this output and halt. Otherwise, run the algorithm as usual. It is unlikely that this will be successful, but in the best-case the running time will only be as long as it takes to check a solution. For example, we could modify selection sort to first randomly permute the elements of A, then check if they are in sorted order. If they are, output A. Otherwise run selection sort as usual. In the best case, this modified algorithm will have running time Θ(n).

Exercise 2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array A=⟨3,41,52,26,38,57,9,49⟩.

Solution :



The figure is slightly different than the one in the book. The figure in the book was showing sorting progress from bottom to up. Here it is shown from top to bottom, which I feel will be more intuitive to understand.

Ex : 2.3-2

The test in line 1 of the MERGE-SORT procedure reads r, then the subarray AŒp W r� is empty. Argue that as long as the initial call of MERGE-SORT.A; 1; n/ has n 1, the test r.

Solution :

Selection Sort Algorithm

Pseudocode:

```
SELECTION-SORT(A, n):
    for i from 1 to n - 1 do:
        min_index = i
        for j from i + 1 to n do:
            if A[j] < A[min_index] then:
                min_index = j
        exchange A[i] with A[min_index]
```

Loop Invariant

The loop invariant for this algorithm is: *At the start of each iteration of the outer loop (for each index iii), the subarray A[1…i−1]contains the smallest i−1 elements of the original array, sorted in non-decreasing order.*

Reason for Iterating Only the First n−1 Elements

The selection sort only needs to run for the first n−1 elements because by the time we have placed the smallest n−1elements in their correct positions, the last element will automatically be in its correct position. This is because it will be the only remaining unsorted element, which must be the largest in the original array.

Running Time Analysis

- **Worst-case Running Time**: The worst-case scenario occurs when the elements are arranged in reverse order. In this case, the algorithm still has to compare each element to find the minimum for each position. The outer loop runs n−1n - 1n−1 times, and for each iteration of the outer loop, the inner loop runs n−i times (where iii is the current index of the outer loop). The total number of comparisons can be expressed as:

Total Comparisons=(n−1)+(n−2)+…+1=(n−1)·n\2

This results in a time complexity of O(n^2)

- **Best-case Running Time**: The best-case scenario for selection sort still requires examining every element to find the minimum, even if the array is already sorted. Therefore, the best-case running time is also O(n^2)

Summary

- **Pseudocode**: Provided above.
- **Loop Invariant**: The subarray A[1…i−1]is sorted and contains the smallest i−1 elements.
- **Reason for n−1n - 1n−1**: The last element will be correctly positioned after sorting the first n−1 elements.
- **Worst-case and Best-case Running Time**: Both are O(n^2).

Ex : 2.2-3

Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence 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? What are the average-case and worst-case running times of linear search in ΘΘ-notation? Justify your answers.

Solution :

On average, half the the elements in array A will be checked before v is found in it.

And in the worst case (v is not present in A), all the elements needs to be checked.

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

Ex : 2.2-4

How can we modify almost any algorithm to have a good best-case running time?

Solution :

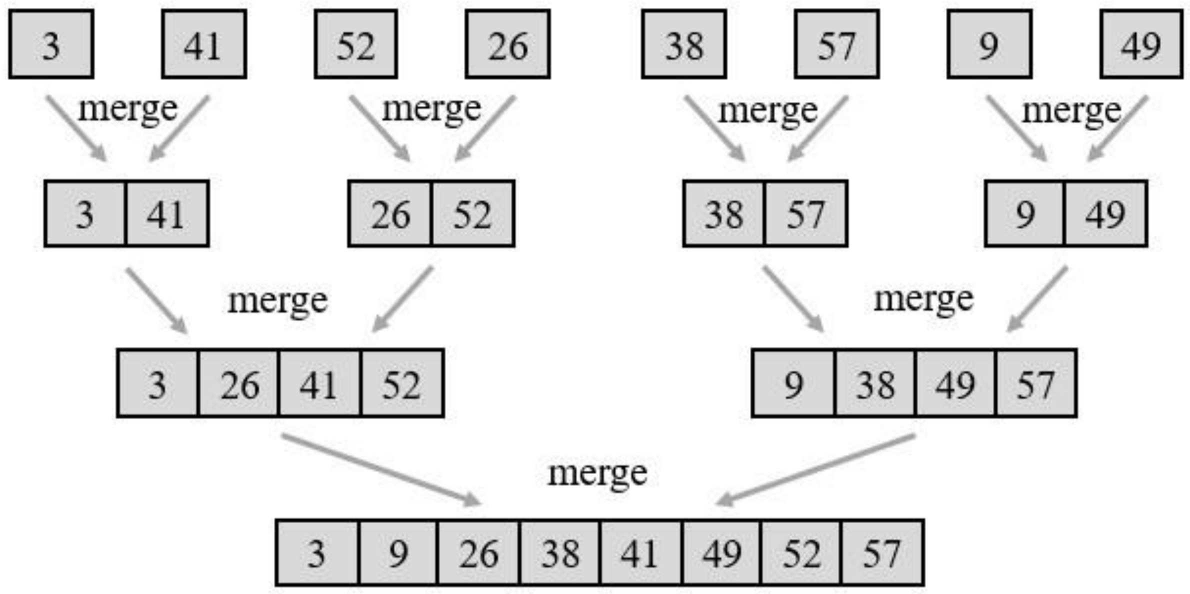
We can design any algorithm to treat its best-case scenario as a special case and return a predetermined solution.

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 Θ(n).

Ex :2.3-1

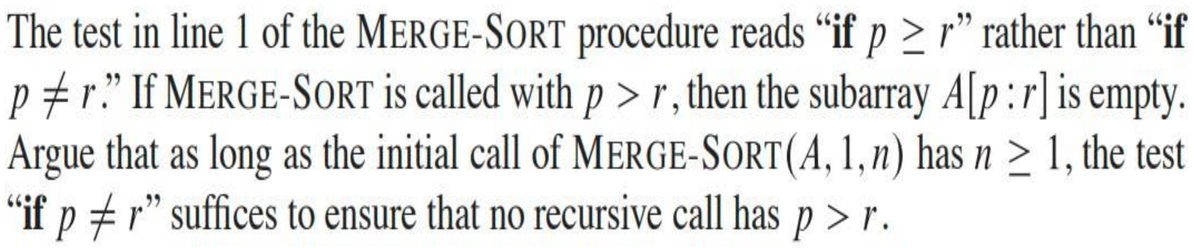
Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence (3, 41, 52, 26, 38, 57, 9, 49i).

Solution :



The figure is slightly different than the one in the book. The figure in the book was showing sorting progress from bottom to up. Here it is shown from top to bottom, which I feel will be more intuitive to understand .

Ex : 2.3-2



The test in line 1 of the MERGE-SORT procedure reads "**if** $p \geq r$" rather than "**if** $p \neq r$." 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 $n \geq 1$, the test "**if** $p \neq r$" suffices to ensure that no recursive call has $p > r$.

The MERGE-SORT algorithm is designed to sort an array by recursively dividing it into smaller subarrays. The key steps involve:

1. **Base Case**: The procedure checks if the current segment of the array defined by indices p and r is valid by evaluating the condition if p < r. If this condition is false, the procedure terminates without making further recursive calls.
2. **Initial Call**: The initial call to MERGE-SORT(A, 1, n) is made with p=1 and r=n Assuming n≥1, we have 1≤n, which means p will always be less than or equal to r in this context.
3. **Recursive Calls**: During the execution of the MERGE-SORT algorithm, the indices p and r are updated when the array is split:
    - Typically, the midpoint q is calculated as $q=\lfloor(p+r)/2\rfloor$
    - The algorithm then makes recursive calls to sort the left and right halves:
        - MERGE-SORT(A, p, q)
        - MERGE-SORT(A, q + 1, r)
4. **Index Validity**: In these recursive calls:
    - For MERGE-SORT (A, p, q), p remains unchanged, while q is always less than or equal to r.
    - For MERGE-SORT (A, q + 1, r), since q+1 is strictly greater than p (when p<r), and because q ≤ r, q+1 cannot exceed r.

Given that both p and r are updated according to these rules, and since p<r holds true in every recursive call (as derived from the splitting of the array), it follows that p cannot exceed r. Thus, the condition if p < r effectively prevents any scenario where p > r arises during the execution of the MERGE-SORT algorithm.

Conclusion

Therefore, as long as the initial call to MERGE-SORT (A, 1, n) is made with n ≥1, the test if p < r is sufficient to ensure that no recursive call results in p > r, thereby correctly handling the sorting of the array without encountering empty subarrays in inappropriate contexts.

Ex : 2.3-3

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.

Solution :

Proving Correctness Using Loop Invariants

1. **While Loop (Lines 12-18)**:
   - **Initialization**:
   - Before the first iteration, no elements are merged, so the invariant holds trivially.
   - **Maintenance**:
   - Each iteration compares the smallest elements of L and R, merging them into A while maintaining sorted order.
   - **Termination**:
   - When the loop ends, the invariant ensures the merged section of A is sorted.
2. **While Loop (Lines 20-23)**:

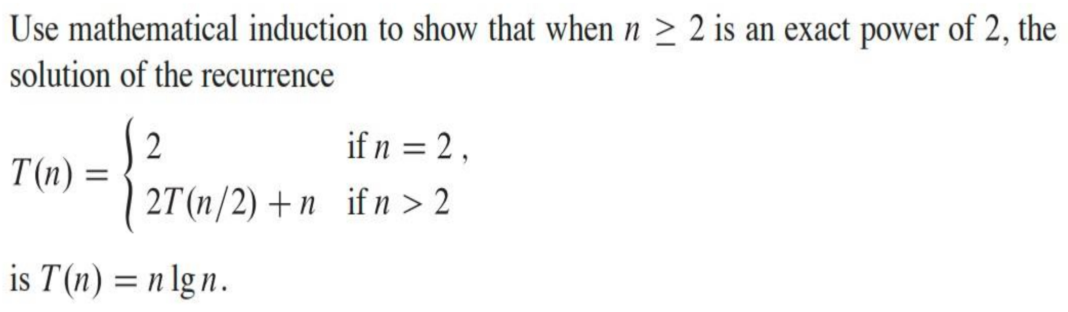
   - This loop handles any remaining elements in L. The invariant guarantees these elements are sorted and can be appended to A.

3. **While Loop (Lines 24-27)**:
   - This loop manages remaining elements in R. Again, the invariant ensures sorted order when appending to A.

Conclusion

By using the loop invariant throughout the MERGE procedure, we confirm that the merged elements are always sorted, proving the correctness of the MERGE procedure.

Ex: 2.3-4



Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 , \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is $T(n) = n \lg n$.

So;ution :

Base Case

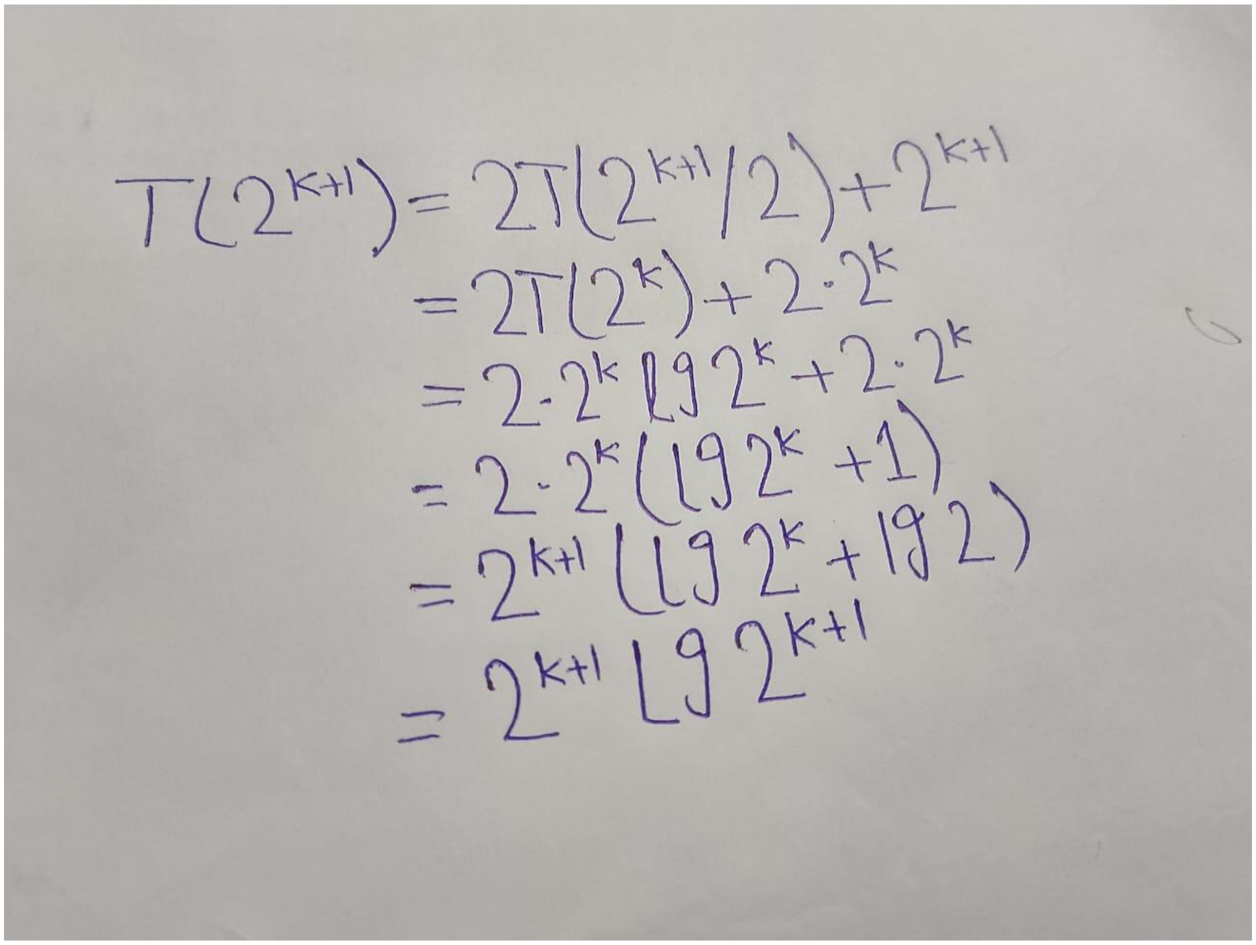
When n=2 , T(2) = 2 =2 lg 2. So, the solution holds for the initial step.

Inductive Step

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

T(2^*k*+1)= 2^*k*+1 lg 2^*k*+1.

From our recurrence formula,



$$
\begin{aligned}
T(2^{k+1}) &= 2T(2^{k+1}/2) + 2^{k+1} \\
&= 2T(2^k) + 2\cdot 2^k \\
&= 2\cdot 2^k \lg 2^k + 2\cdot 2^k \\
&= 2\cdot 2^k(\lg 2^k + 1) \\
&= 2^{k+1}(\lg 2^k + \lg 2) \\
&= 2^{k+1} \lg 2^{k+1}
\end{aligned}
$$

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 n holds for all n that are exact power of 2.

Ex : 2.3-5

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.

Solution :

```
RecursiveInsertionSort(A, n):
   if n <= 1:
      return  // Base case: a single element is already sorted


   // Recursively sort the first n-1 elements
   RecursiveInsertionSort(A, n - 1)


   // Insert the n-th element into the sorted subarray
   key = A[n - 1]
   j = n - 2


   // Shift elements of A[0..n-2] that are greater than key
   while j >= 0 and A[j] > key:
      A[j + 1] = A[j]
      j = j - 1


   A[j + 1] = key  // Place key at the correct position
```

2. Recurrence for the Worst-Case Running Time

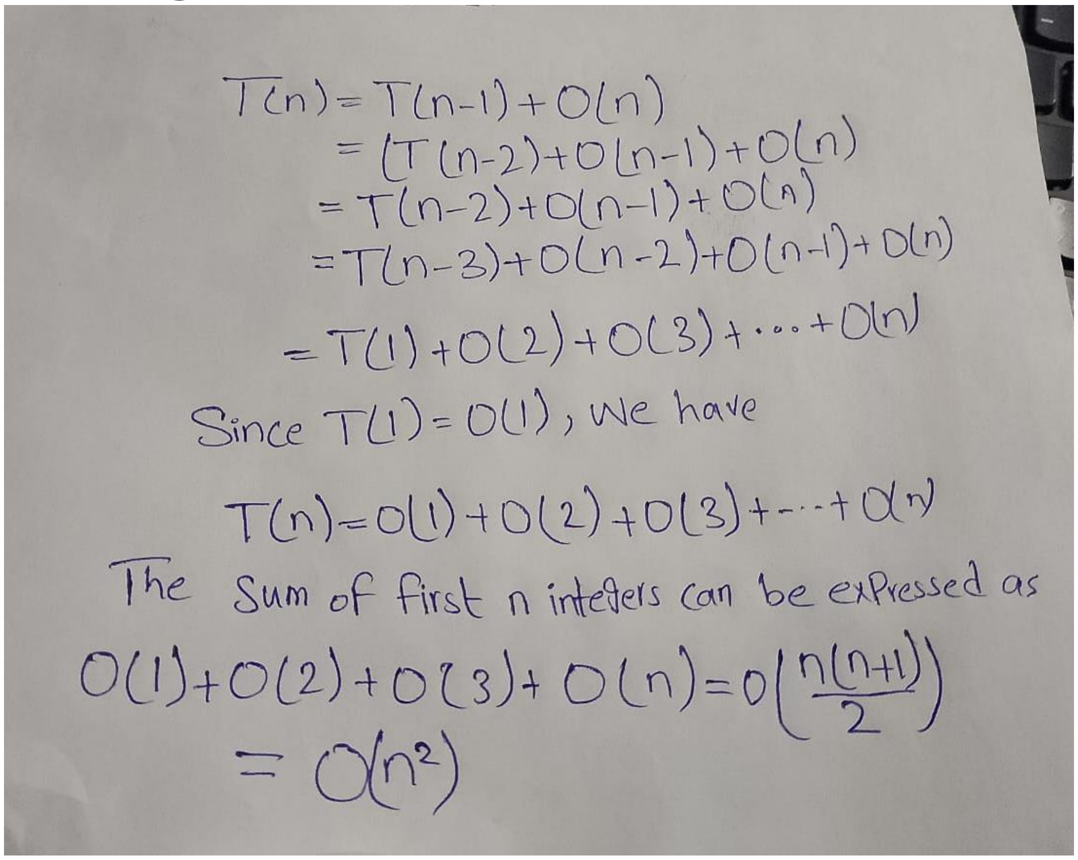
The recurrence relation for the worst-case running time T(n)T(n)T(n) of this recursive insertion sort can be expressed as:

T(n) =T (n−1) + O(n)

Explanation:

- T(n−1) represents the time taken to sort the first n−1 elements recursively.
- O(n) represents the time taken to insert the nnn-th element into its correct position in the sorted subarray, which may require shifting up to n−1 elements.

Solving the Recurrence :



$$T(n) = T(n-1) + O(n)$$
$$= (T(n-2) + O(n-1) + O(n)$$
$$= T(n-2) + O(n-1) + O(n)$$
$$= T(n-3) + O(n-2) + O(n-1) + O(n)$$
$$= T(1) + O(2) + O(3) + \cdots + O(n)$$

Since $T(1) = O(1)$, we have

$$T(n) = O(1) + O(2) + O(3) + \cdots + O(n)$$

The sum of first n integers can be expressed as

$$O(1) + O(2) + O(3) + O(n) = O\left(\frac{n(n+1)}{2}\right)$$
$$= O(n^2)$$

Conclusion

Thus, the worst-case running time of the recursive insertion sort is:
T (n) = O (n ^2).

Ex : 2.3-6

2.3-6 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 ,0 (lg n).

Solution :

Here is the pseudocode if you prefer iterative solutions …

```
Iterative-Binary-Search (A,v)
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

Recursive-Binary-Search (*A*,*v*,*low*,*high*)

```
1   if low>high
2           return  NIL
3   mid=⌊(low+high)/2⌋
4   if v==A[mid]
5         return mid
6   elseif v>A[mid]
7             Recursive-Binary-Search(A,v,mid+1,high)
8   else
9         Recursive-Binary-Search(A,v,low,mid−1)
```

Intuitively, in worst case, i.e. when v*v* is not at all present in A*A*, we need to search over the whole array to return NIL. In other words, we need to repeatedly divide the array by 2 and check either half for v*v*. So the running time is nothing but how many times the input size can be divided by 2, i.e.   lg n.
For recursive case, we can write the recurrence as follows …

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

Ex : 2.3-7

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 linear search? Would that improve the overall worst-case running time of insertion sort to 0 (n lg n)?

Solution :

```
5       while $i > 0$ and $A[i] > key$
6           $A[i+1] = A[i]$
7           $i = i - 1$
```

This loop serves two purposes:

1. A linear search to scan (backward) through the sorted sub-array to find the proper position for key.
2. 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 Θ(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 Θ(n^2).

Ex : 2.3-8

 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

Solution :

Algorithm Steps

1. Sort the Set: First, sort the array of integers. This takes O(n log n)time.
2. Use Two Pointers: Initialize two pointers:
    - One pointer (left) starts at the beginning of the sorted array.
    - The other pointer (right) starts at the end of the sorted array.
3. Iterate and Check:
    - While the left pointer is less than the right pointer:

- Calculate the sum of the elements at the two pointers: sum=S[left]+S[right.
- If sum equals x, return true (indicating that such a pair exists).
- If sum x, increment the left pointer to increase the sum.
- If x, decrement the right pointer to decrease the sum.

4. Return False: If the pointers cross without finding a valid pair, return false.

Pseudocode

plaintext

```
FindPairWithSum(S, n, x):
   // Step 1: Sort the array
   Sort(S)  // This takes O(n log n)
```

```
// Step 2: Initialize pointers
left = 0
right = n - 1

// Step 3: Iterate with two pointers
while left < right:
    sum = S[left] + S[right]

    if sum == x:
        return true  // Pair found
    else if sum < x:
        left = left + 1  // Increase sum by moving left pointer
    else:
        right = right - 1  // Decrease sum by moving right pointer

// Step 4: No pair found
return false
```

Time Complexity

- Sorting the array takes O(n log n).
- The two-pointer technique runs in O (n) since each pointer moves at most n times.

Therefore, the total time complexity is:

O (n log n)+O(n)=O(n log n)

Conclusion

This algorithm efficiently determines whether there are two elements in the set S that sum to x within the specified time complexity of O(n log n) in the worst case.

Chapter : 3

Characterizing Running Times

Exercises

Ex : 3.1-1

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

Solution :

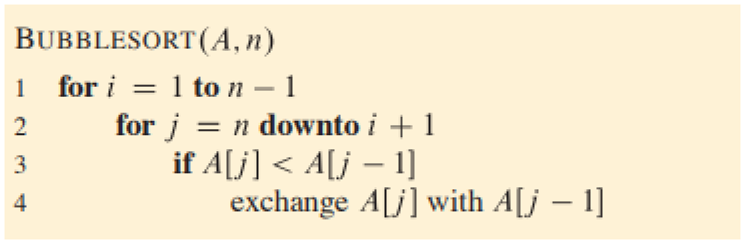
To modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3, adjust the start and end indices of the **outer loop.**

Explanation:

The **lower-bound** argument for insertion sort can be modified to handle input sizes that are not necessarily a multiple of 3 by adjusting the start and end indices for the outer loop. Instead of starting at 2 and going up to n, we can start at the first index divisible by 3 and go up to the last index less than or equal to n that is divisible by 3. This ensures that all elements are still considered in the sorting process.

Ex : 3.1-2

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



```
BUBBLESORT$(A, n)$
for $i = 1$ to $n - 1$
    for $j = n$ downto $i + 1$
        if $A[j] < A[j - 1]$
            exchange $A[j]$ with $A[j - 1]$
```

Solution :

Selection Sort Time Complexity Analysis

Overview: Selection sort divides the array into a sorted and an unsorted part, repeatedly selecting the minimum element from the unsorted portion and moving it to the sorted portion.

Steps:

1. **Outer Loop**: Iterates from the first element to the second-to-last element (n−1 times).
2. **Inner Loop**: For each position iii, finds the minimum in the unsorted part (from iii to n−1).
   - First iteration checks n−1 elements, second checks n−2n , down to 1 element.

Comparisons: The total comparisons made is:

(n−1)+(n−2)+...+1= = (n - 1) n\ 2

Time Complexity:

Overall, the time complexity is:

O(n^2)

Conclusion: Selection sort consistently runs in O(n^2) time regardless of input order.

Ex : 3.1-3

Suppose that ˛ is a fraction in the range 0 < a < 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 a? What value of a maximizes the number of times that the ˛n largest values must pass through each of the middle .(1-2a)n array positions?

Solution :

To generalize the lower-bound argument for insertion sort with $0<α<1$:

1. **Setup:**

   The largest αn values are in the first αn positions. The remaining (1−α)n values are in the last (1−α) n positions.

2. **Movement:**

   Each of the αn largest values must pass through the middle (1−2α)n positions occupied by the smaller values.

3. **Restriction:**

   To ensure that the largest values need to move past the smaller values, α\alphaα must be less than 0.5.

4. **Maximizing Movement:**

   The value of α\alphaα that maximizes the number of passes through the middle positions is α\alphaα approaching 0.5 from the left (i.e., $α→0.5^−$).

https://github.com/Sardar-Hassan661/DSA-Assingment/tree/main