

**SUBMITTED BY : BISMA AZHAR**

**SUBMITTED TO: SIR JAMAL ABDUL AHAD**

**COURSE TITLE: DATA STRUCTURE**

**ROLL NO: 14641**

**SUBMISSION DATE: 22 OCT 2024**

**FOUNDATION**

**UNIT NO 1**

**THE ROLE OF ALGORITHMS IN COMPUTING**

* **PART 1**

**ALGORITHMS**

**EXERCISE**

**1.1-1**

**Question:**

**1.1-1:-** Describe your own real word example that require sorting. Describe that one requires finding the shortest distance between two points ?

**ANSWER:**

**Real-World Example:**

* **Sorting:**  
  Patient data in hospitals are frequently classified by treatment priority. Each patient is given a priority depending on severity, with the most critical cases at the top of the list. Sorting these information enables clinicians to treat critical situations quickly, hence increasing response times and overall patient care.

**Real-world Example**

* **Requiring thes hortest distance:**

Finding the quickest path between two points is critical in urban planning for effective road development. For example, calculating the shortest way between a hospital and neighboring residential areas can aid in the optimization of routes For example, calculating the shortest way between a hospital and neighboring residential areas can aid in the optimization of routes for emergency responders, lowering travel time and enhancing accessibility during crucial situations.

**1.1-2:-** Other than speed, what other measures of efûciency might you need to consider in a real-world setting?

**ANSWER:**

**Flexibility:**

Adapting fast to changes, such as shifting client preferences or supplier constraints, can help a business stay operational even when conditions change unexpectedly.

**Environmental Impact:**

Reducing emissions and waste promotes sustainability and often lowers expenses. For example, environmentally responsible manufacturing processes can result in long-term cost savings and a great public image.

**Reliability and consistency:**

Ensuring that processes operate without frequent disruptions is critical, particularly in industries such as healthcare and manufacturing where dependable performance is required.

**Scalability:**

As demand increases, efficiency should be maintained. For example, a company should be able to accommodate additional clients without losing quality or increasing delays.

**Reliability and consistency:**

Ensuring that processes operate without frequent disruptions is critical, particularly in industries such as healthcare and manufacturing where dependable performance is required.

**Cost efficiency:**

Cost efficiency entails lowering operational costs such as fuel in transportation, labor charges, and material costs in manufacturing. Efficient systems seek to accomplish desired outcomes with minimal investment, which is critical for profitability and sustainability.

**Customer Satisfication:**

Efficiency can also be gauged by how well a product or service meets customer needs. High customer satisfaction often correlates with efficient processes

.

**1.1-3**:- Select a data structure that you have seen, and discuss its strengths and limitations.

**ANSWER:**

**Data Structure: Array Strength.**  
**Fast Access:**

Arrays provide constant-time (O(1)) access to elements by index, making them ideal for fast data retrieval.   
**Simple Memory Allocation:**

Arrays use contiguous memory, which is easy to allocate and allows for efficient data management.   
**Limitations:**   
  
**Fixed Size:**

Once declared, the size of an array is fixed, limiting flexibility when adding new components.   
**Costly insertions and deletions:**

Adding or removing members from the middle of an array requires element shifting,which takes time (O(n).

**1.1-4:-** How are the shortest-path and travelling-salesman problems given above similar? How are they different?

ANSWER:  
the Shortest Path Problem and the Travelling Salesman Problem (TSP):

# Similarities:

1. **Graph-Based**:

Both problems use graphs where nodes represent locations and edges represent paths.

1. **Minimization Goal**:

Both aim to minimize travel costs, whether it’s distance or time.

### Differences:

**Objective**:

* 1. **Shortest Path Problem**: Finds the shortest route from one specific starting point to a specific destination.
  2. **Travelling Salesman Problem**: Requires visiting multiple locations exactly once and returning to the starting point.

**Complexity**:

* 1. **Shortest Path Problem**: Can be solved efficiently with algorithms like Dijkstra’s.
  2. **Travelling Salesman Problem**: NP-hard, meaning it’s much harder to solve, especially as the number of locations increases.

**Output**:

* 1. **Shortest Path Problem**: Provides a single shortest path between two points.
  2. **Travelling Salesman Problem**: Provides a complete tour visiting all points exactly once and returning to the start.

**1.1-5:-** Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is “approximately” the best is good enough.

**ANSWER:**

**Real-world Problem Requiring the Best Solution:**

**EXAMPLE: Air Traffic Control**

Air traffic control relies heavily on ensuring the safe and efficient movement of airplanes. Controllers must calculate the best flight routes and landing sequences for several planes approaching a crowded airport. A minor inaccuracy might cause accidents or major interruptions, thus only the finest solution is acceptable to ensure air travel safety and efficiency.

**Real-world Problem Allowing for an Approximately Best Solution:**

**EXAMPLE :Restaurant Menu Optimization.**

In a restaurant, optimizing the menu to increase customer pleasure and revenues results in an approximately optimal solution. By evaluating sales data and customer comments, the restaurant can change its menu to include popular meals without requiring the ideal match. This adaptable strategy encourages creativity while also aiming for a balanced and lucrative meal.  
  
  
  
  
**1.1-6:-** Describe a real world problem in which sometimes the entire input is available before you need to solve the problem but others times the input is not entirely available in advance and arrives over times.

ANSWER:

**Real-world Problem: Supply Chain Management**  
In supply chain management, firms frequently encounter varied conditions regarding the availability of input data.  
**Entire Input Avaliable:**  
Companies may receive complete data on inventory levels, supplier capabilities, and client demand before making decisions. For example, during a planned manufacturing run, a manufacturer can optimize production scheduling and inventory management by analyzing all available inputs.

**Input Arriving Over Time:**

During unanticipated occurrences, such as a sudden rise in demand or supply chain disruptions (such as natural catastrophes or geopolitical concerns), businesses may only receive partial information at first. For example, a store may not know how much stock will be available from suppliers owing to delays or shortages, forcing them to adjust their plans as new information becomes available over time.

**EXERCISE 2:-**

**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

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 Exlorer:**

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](https://en.wikipedia.org/wiki/Video_codec) to achieve video decoding and some for [recommending new content](https://help.netflix.com/en/node/100639).

**Any Game:**

Applies [clipping algorithm](https://en.wikipedia.org/wiki/Clipping_(computer_graphics)) to discard objects that are outside the viewport.

**1.2-2:-**Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in 8n^2 steps, while merge sort runs in 64n lg n steps. For which values of n does insertion sort beat merge sort?

To find the values of nnn for which insertion sort beats merge sort, we need to determine when the time complexity of insertion sort is less than that of merge sort. Given:

**Insertion Sort:** 8n2

**Merge Sort:** 64nlgn

We are looking for values of nnn such that:

8n2<64nlgn

Dividing both sides by 8n (assuming n>0), we get:

n<8lgn

Now we need to solve this inequality, n<8lgn, to find the values of n for which it holds.

Approach:

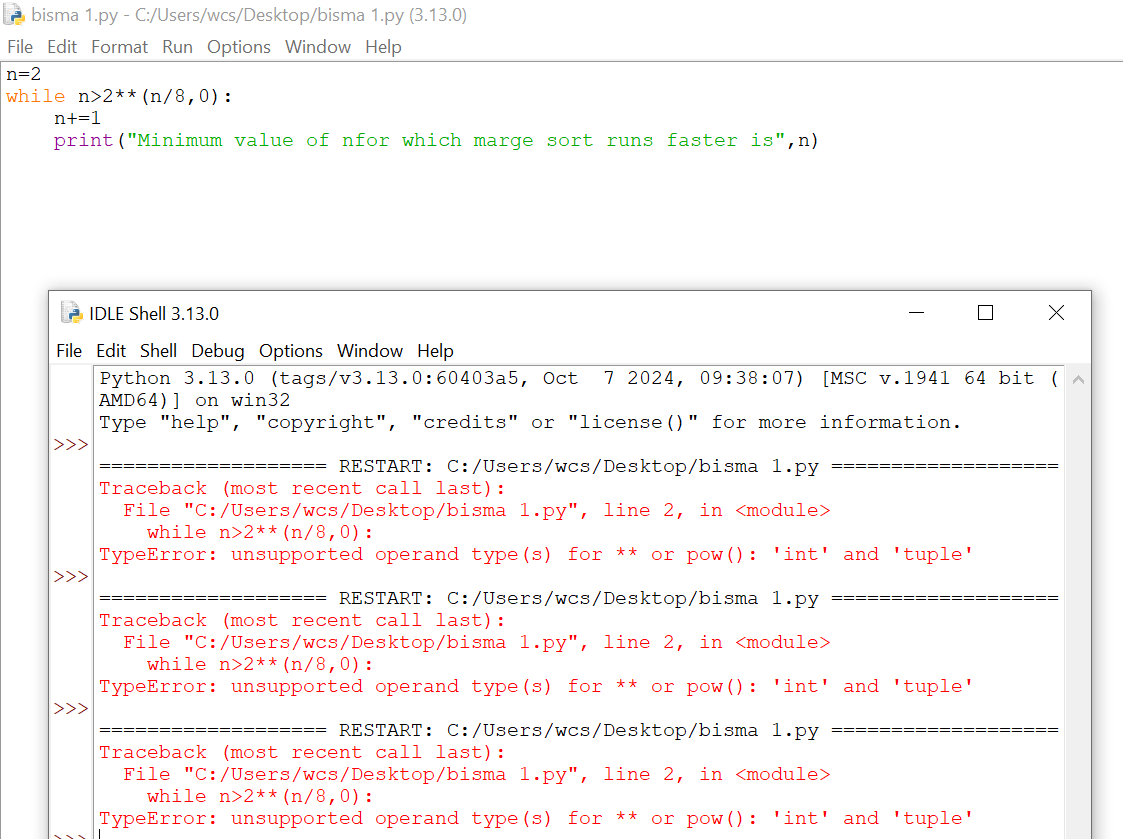
We can approximate the values of nnn for which n<8lgn by testing different values of n and checking the inequality. However, let's also approach it theoretically to get a rough sense of where the inequality might hold:

* **For small values of n** (like 1 through 10), we can check explicitly.
* **For larger values of n**, since nnn grows faster than lgn, there will be a point where n exceeds 8lgn, so we’ll search for that threshold.

Let's calculate and verify a few values of nnn to pinpoint this threshold.

The inequality n<8lgn holds for values of n up to 43. Therefore, insertion sort outperforms merge sort for input sizes n≤43 on this machine. For n>43, merge sort becomes more efficient

**Python Code:**

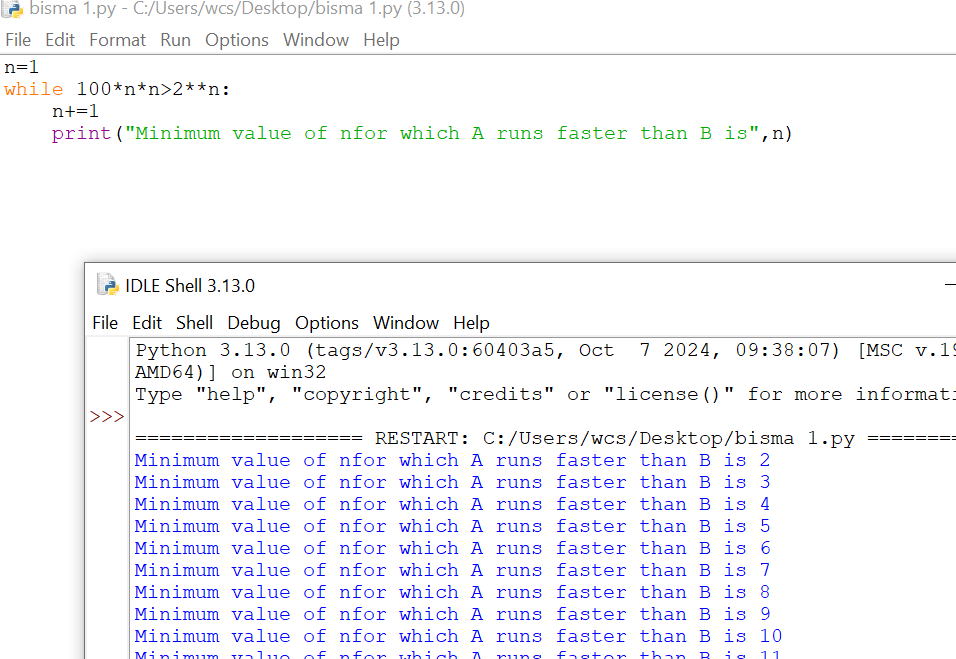


# 1.2-3:-What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2^n on the same machine?

**ANSWER:**

We want that 100n 2 < 2 n. note that if n = 14, this becomes 100(14)2 = 19600 > 2 14 = 16384. For n = 15 it is 100(15)2 = 22500 < 2 15 = 32768. So, the answer is n = 15.

**Phython Code:**



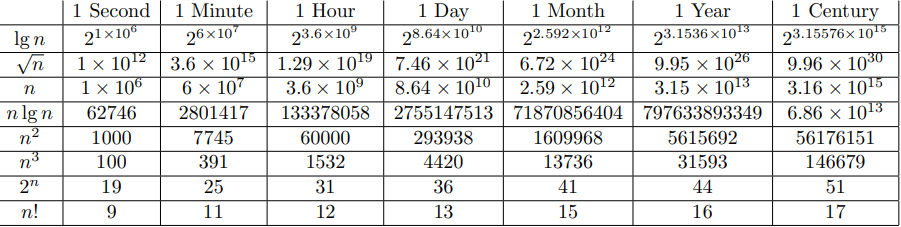
PROBLEM 1.1:

For each function f .n/ and time t in the following table, determine the largestsize n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f .n/ microseconds.

ANSWER:

We assume a 30 day month and 365 day year.

**TABLE:**



**CHAPTER NO 2:**

**2.1-1:-** Using Figure 2.2 as a model, illustrate the operation of Insertion-Sort Insertion-Sort  on the array A=⟨31,41,59,26,41,58⟩A=⟨31,41,59,26,41,58⟩.

**ANSWER:**

**Insertion Sort Process:**

**1.Initial Array**

𝐴 = ⟨ 31 , 41 , 59 , 26 , 41 , 58 ⟩

A=⟨31,41,59,26,41,58⟩

**2.Step One (Index 1):**

1. Current element: 41.
2. Compare to 31. There's no need to swap.

**Array:** 𝐴= ⟨ 31 , 41 , 59 , 26 , 41 , 58 ⟩

A=⟨31,41,59,26,41,58⟩

**3.Step Two (Index 2):**

1. Current Element: 59
2. Compare to 41. There's no need to swap.

**Array after swaps:**

𝐴= ⟨ 31 , 41 , 59 , 26 , 41 , 58 ⟩

A=⟨31,41,59,26,41,58⟩

**4.Step three (Index 3):**

1. Current Element: 26
2. Compare to 59, 41, and 31. Swap with each.

**Array after swaps:**

𝐴 = ⟨ 26 , 31 , 41 , 59 , 41 , 58 ⟩

A=⟨26,31,41,59,41,58⟩

**5.Step four (Index 4):**

1. Current element: 41.
2. Compare to 59. Swap.
3. Compare to 41. There will be no future swaps.

**Array after swaps:**

𝐴 = ⟨ 26 , 31 , 41 , 41 , 59 , 58 ⟩

A=⟨26,31,41,41,59,58⟩

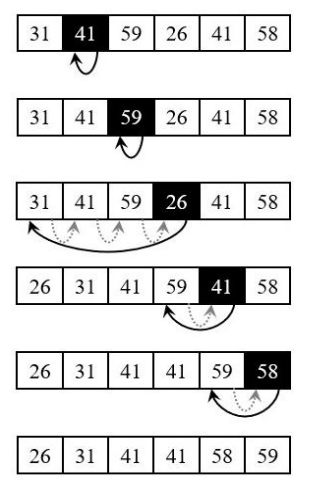
**6.Step 5 (index 5):**

1. Current Element: 58
2. Compare to 59. Swap.
3. Compare to 41. There will be no future swaps.

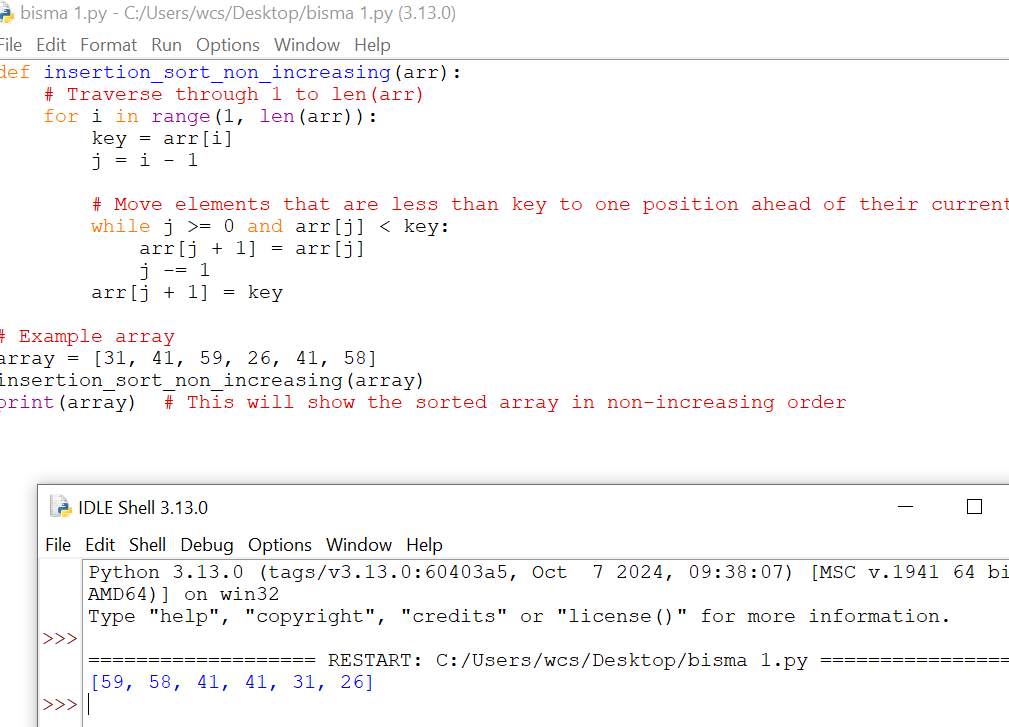
**Finally sorted array:**

𝐴 = ⟨ 26 , 31 , 41 , 41 , 58 , 59 ⟩

A=⟨26,31,41,41,58,59⟩

  
  
  
**2.1-2:-** Rewrite the Insertion-SortInsertion-Sort procedure to sort into non-increasing instead of non-decreasing order.

**PYTHON CODE:**

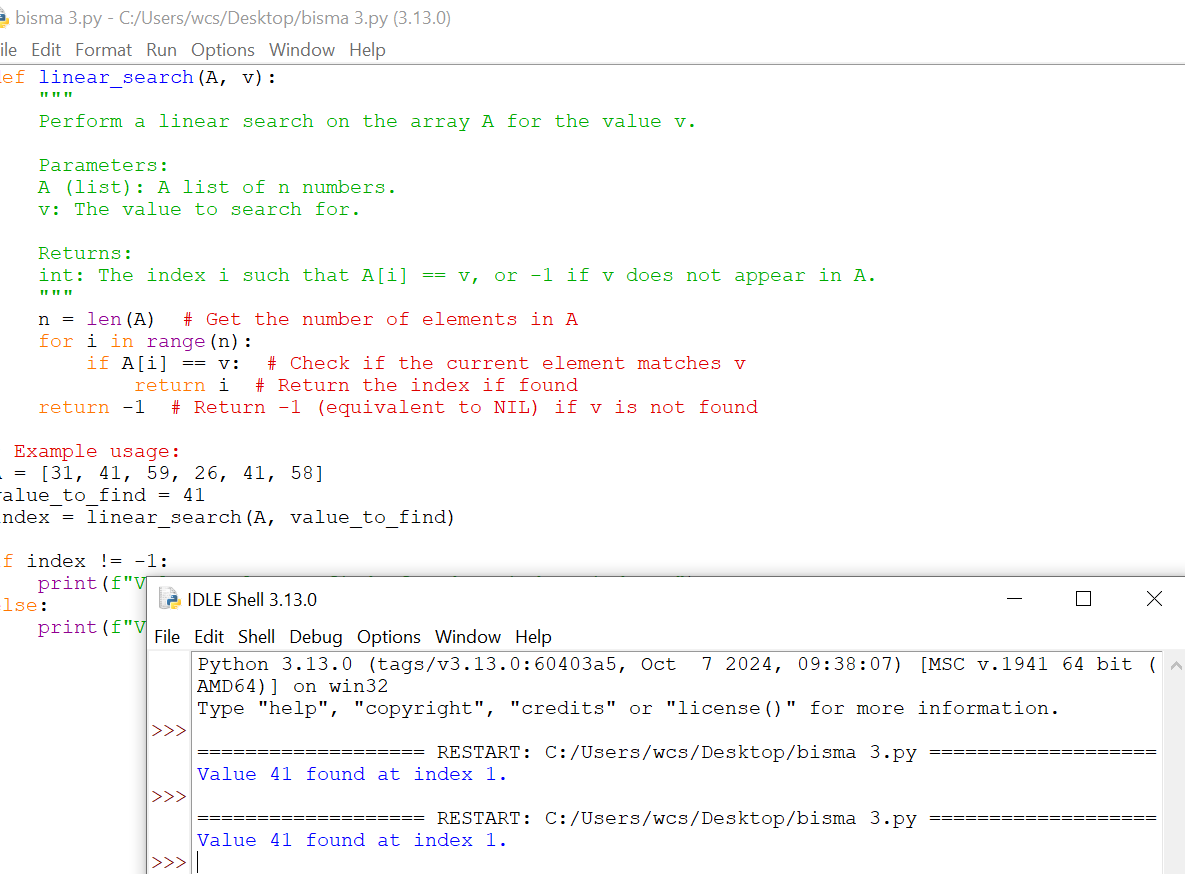
**  
  
  
2.1-3:-** Consider the searching problem:

Input: A sequence of n numbers A=⟨a1,a2,…,an and a value v.

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

Write pseudocode for linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

**PYTHON CODE:**



**Loop Invariant.**

:At the start of each iteration of the loop (before the if statement), the value v is not found in the first i-1 members of the array A.:

**Initialization:**

The loop invariant is true prior to the first iteration of the loop.

**Maintenance:**

Assuming the loop invariant is true at the start of the i-th iteration (i.e., v is not found in the first i-1 elements):

If 𝐴 [ 𝑖 ] = = 𝑣 A[i]==v, the algorithm yields 𝑖 i, and we have obtained the value v.

If 𝐴 [ 𝑖 ] ≠ 𝑣 A[i]  =v, we go to the next iteration. Since v was not discovered in the first 𝑖 i elements, the invariant remains valid for the next iteration (when 𝑖 i grows by 1).

The loop invariant applies to iterations 𝑖 and 𝑖+1.

**Termination:**

The loop might end in one of two ways:

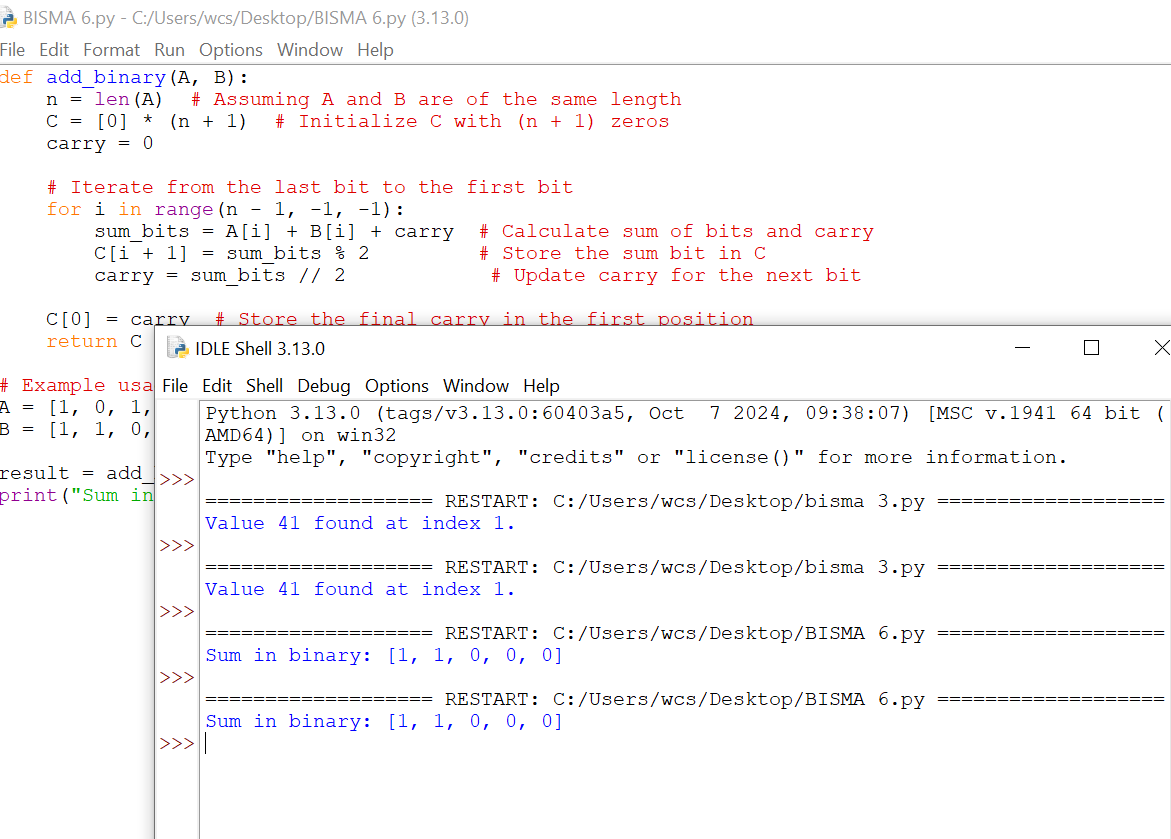
1. The program finds the value v in the array and returns the index i. In this situation, the loop invariant ensures that 𝑣 = 𝐴 [ 𝑖 ] v=A[i].
2. The loop ends when 𝑖 = 𝑛 + 1 (i=n+1), indicating that all 𝑛 elements in the array have been verified. If v was not discovered, the loop invariant indicates that it was not present in any of the n items. The algorithm then returns NI

**Conclusion:**

We may infer that the linear search algorithm is proper because we demonstrated that the loop invariant is true for initialization, is preserved during each iteration, and yields a valid conclusion at termination. The function successfully provides an index 𝑖 i such that 𝑣 = 𝐴 [ 𝑖 ] v=A[i], or NIL if 𝑣 v does not occur in the sequence 𝐴 A.

**2.1-4:-**Consider the problem of adding two n-bit binary integers, stored in two nn element arrays AA and BB. The sum of the two integers should be stored in binary form in an (n+1) element array C. State the problem formally and write pseudocode for adding the two integers.

**PYTHON CODE:**



**EXERCISE NO 2:-**

2.2-1:-Express the function n3/1000−100n2−100n+3 intermsof Θ notation.

**ANSWER:**

**FUNCTION:**

**f(n)=1000n3​−100n2−100n+3**

in terms of Θ notation, we need to identify the term with the highest growth rate as n→∞n , since Θ notation describes the asymptotic behavior.

**Identify the Dominant Term:**

The terms in f(n) are:

* **n^3/1000**
* **−100n^2**
* **−100n-**
* **+3**
* Among n^3/1000 these is the term with the highest degree n^3, so it will dominate the function's growth rate as n becomes large.

**Express in O Notation:**

*:* Since n^3/1000​ is the leading term and constant factors are ignored in Θ notation, we have:

**f(n)=Θ(n^3)**

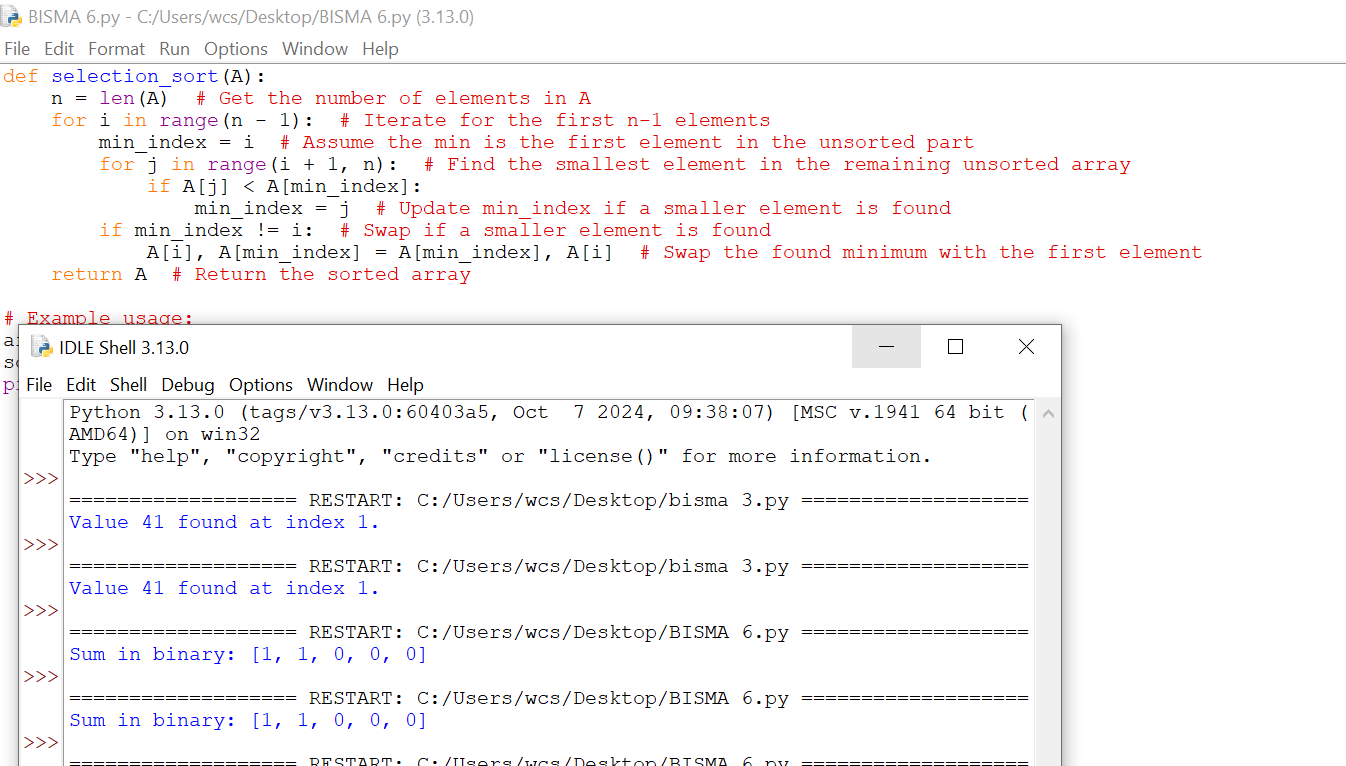
**Final Answer:**

**f(n)=Θ(n^3)**

This means f(n) grows asymptotically at the same rate as n^3 as n→infinity

**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.

**PYTHON CODE:**

****

**Loop Invariant:**

**Why run for only the first 𝑛 - 1 elements?**

The selection sort algorithm only sorts the first 𝑛 - 1 n−1 elements, as the last member is always the largest in the array. As a result, the last unsorted element is correctly positioned without the need for any additional checks or swaps.

**Running Times of Selection Sort**

1. The best-case running time is: Θ ( 𝑛 2 ) Θ(n 2).
2. The worst-case running time is: Θ ( 𝑛 2 ) Θ(n 2)

In selection sort, the number of comparisons remains constant regardless of the original order of the elements, resulting in a consistent quadratic running time.

**2.2-3:-**Consider linear search again (see [Exercise 2.1-3](https://atekihcan.github.io/CLRS/02/E02.01-03/)). 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 Θ-notation?

ANSWER:.

**Linear Search Overiview:**

In linear search, we look for an element v in an array A by checking each element one by one from the start to the end of the array.

**Worst Case:**

**Definition**:

* The worst case occurs when:
  + The element v is not in the array, or
  + The element v is the last element in the array.

**Elements Checked:**

* : In both scenarios, you need to check all n elements.

**Running Time:O(n)**

**Average Case:**

**Definition:**

The average case assumes that v is equally likely to be at any position in the array.

**Elements Checked:**

* If v is found, on average, you'll check about half of the elements, which is n/2.

**Running Time:Θ(n)**

Even though you check about half of the elements on average, the time complexity is still linear, as it grows proportionally with the size of the array.

**Conclusion:**

1. **Worst-case**: Θ(n) (check all elements)
2. **Average-case**: Θ(n) (check about half, but still linear)

Both the worst-case and average-case running times for linear search are linear because they grow with the size of the input array.

**EXERCISE 3:**

2.3-1 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; 49i.?

**Answer:**

To demonstrate the merge sort algorithm on the given array ℎ 3 , 41 , 52 , 26 , 38 , 57 , 9 , 49 𝑖 h3,41,52,26,38,57,9,49i, we will follow the divide-and-conquer strategy of merge sort, which involves recursively dividing the array into smaller subarrays until they are of size one, and then merging those subarrays back together in a sorted manner.

**Solution:**

1. **Divide Step:**

Initial array: 3,41,52,26,38,57,9,49

* Split into: 3,41,52,26 and 38,57,9,49.
* Further divide each half until we have individual elements:
* 3,41,52,26,38,57,9,49

**2. Conquer Step:**

**Merge the single elements:**

* Merge 3 and 41 to get 3,41
* Merge 52 and 26 to get 26,52
* Merge 38 and 57 to get 38,57
* Merge 9 and 49 to get 9,49

**Merge pairs of subarrays:**

* Merge 3,41 and 26,52, to get 3,26,41,52
* Merge 38,57and 9,49 to get 9,38,49,57

**Final Merge:**

* Merge 3,26,41,52 and 9 ,38,49,57 to get 3,9,26,38,41,49,52,57.
* The final sorted array is 3,9,26,38,41,49,52,57.

**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.

**Answer:**

**Inductive case:**

ifp≤rifp<rq=⌊(p+r)/2⌋q≥pq+1≤rMERGE−SORT(A,p,q)MERGE−SORT(A,q+1,r)else returnno recursive call has p>rp≤r holds in new recursive callifp≤rifp<rq=⌊(p+r)/2⌋q≥pq+1≤rMERGE−SORT(A,p,q)MERGE−SORT(A,q+1,r)else returnno recursive call has p>rp≤r holds in new recursive call

**Base case:**

MERGE−SORT(A,1,n)n≥1p≤rholdsMERGE−SORT(A,1,n)n≥1p≤rholds

"if p≠rp=r" suffices to ensure that no recursive call has p>rp>r.

**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.

To demonstrate the validity of the MERGE method, we may define loop invariants for the three while loops in lines 12-18, 20-23, and 24-27, and show how they ensure the procedure generates a sorted merged array.

**Loop invariant for lines 12-18 (Main While Loop):**

Loop Invariant: At the start of each iteration of the while loop (lines 12-18), the following criteria are true:

1. The elements in **𝐴 [ 𝑝 … 𝑘 − 1 ] A[p…k−1]** are sorted.
2. The items in **𝐴 [ 𝑝 … 𝑘 − 1 ] A[p…k−1]** are the smallest **𝑘 − 𝑝 k−p** elements from the initial subarrays **𝐿 L** and **𝑅 R**.
3. The elements **𝐿 [ 𝑖 ] L[i]** and **𝑅 [ 𝑗 ] R[j],** where 𝑖 i and 𝑗 j are the current indices in L and R, respectively, have not yet been placed in 𝐴 A.

**Proof using the invariant:**

* Initialization: Before the loop begins, 𝑘 = 𝑝 k=p, and 𝐴 [ 𝑝 … 𝑘 − 1 ] A[p…k−1] is empty, therefore it is trivially sorted.
* Maintenance: Each iteration, the smallest unmerged element from 𝐿 [ 𝑖 ] L[i] and 𝑅 [ 𝑗 ] R[j] is chosen and placed at 𝐴 [ 𝑘 ] A[k]. This decision preserves the sorted order in 𝐴 [ 𝑝 … 𝑘 ] A[p…k] and increases 𝑘 k by one.
* Termination: The loop ends when 𝑖 > length ( 𝑿 ) i>length(L) or 𝑗 > length ( 𝑅 ) j>length(R), indicating that all entries in one of the arrays have been combined. A[p…k−1] contains the smallest elements from both arr ays in sorted order.
* oop invariant for lines 12-18 (Main While Loop)
* Loop Invariant: At the start of each iteration of the while loop (lines 12-18),the following criteria are true:

**Conclusion:**

Correctness of the Merge ProcedureBy constructing and maintaining these loop invariants, we guarantee that:

The merged array 𝐴 [ 𝑝 … 𝑟 ] A[p…r] includes items from 𝐿 L and 𝑅 R.

The elements in A[p…r] are sorted.

2.3-4 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:**

To prove by induction that the solution of the recurrence 𝑇 ( 𝑛 ) = 2 𝑇 ( 𝑛 / 2 ) + 𝑛 T(n)=2T(n/2)+n for 𝑛 ≥ 2 n≥2 (where 𝑛 n is a power of 2) is 𝑇 ( 𝑛 ) = 𝑛 lg ⁡ 𝑛 T(n)=nlgn

**Base Case:**

For ❑� = 2 and n = 2:

**𝑇 ( 2 ) = 2 T(2)=2**

and 𝑛 lg ⁡ 𝑛 = 2 lg ⁡ 2 = nlgn=2lg2=2. Thus, 𝑇(2 ) = 2 T(2)=2, implying the base case.

**Inductive Step:**

Assume 𝑇 ( 𝑘) = 𝑘 lg ⁡ 𝑘.

T(k) = klgk for 𝑘 = 2 and 𝑚 = 2. We must demonstrate that 𝑇 ( 2 𝑘 ) = 2 𝑘 lg ⁡ ( 2 𝑘 ) T(2k)= 2klg(2

**Using recurrence:**

𝑇 ( 2 𝑘 ) = 2 𝑇 ( 𝑘 ) + 2 𝑘 T(k)=2T(k)+2k.

According to the induction hypothesis, 𝑇 (𝑘) = 𝑘 lg ⁡ 𝑘 T(k) = klgk. Substitute this into the recurring event.

𝑇 ( 2 𝑘 ) = 2 ( 𝑘 lg ⁡ 𝑘 ) + 2 𝑘 = 2 𝑘 lg ⁡ 𝑘 + 2 𝑘 = 2 𝑘 ( lg ⁡ 𝑘 + 1 ) = 2 𝑘 lg ⁡ ( 2 𝑘 )

T(2k)=2(klgk)+2k=2klgk+2k=2k(lgk+1)=2klg(2k)

Thus, 𝑇 ( 𝑛 ) = 𝑛 lg ⁡ 𝑛 T(n)=nlgn holds for 𝑛 = 2 𝑘 n=2k.

**Conclusion:**

Induction yields the solution to the recurrence: 𝑇 ( 𝑛 ) = 𝑛 lg ⁡ 𝑛 T(n)=nlgn for all 𝑛 n that are p

**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.**

**Answer:**

**Algorithm Steps:**

**Sort the Array:**

Sort SSS in ascending order. This takes Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn) time.

**Use Two Pointers:**

**Initialize two pointers:**

1. left at the start (index 0).
2. right at the end (index n−1n - 1n−1).

**While left < right, do the following:**

1. Calculate the sum: sum=S[left]+S[right]\text{sum} = S[\text{left}] +S[\text{right}]sum=S[left]+S[right]
2. If sum=x\text{sum} = xsum=x, return true.
3. If sum<x\text{sum} < xsum<x, increment left.
4. If sum>x\text{sum} > xsum>x, decrement right.

**If no pair is found, return false.**

**Pseudocode:**

plaintext

Copy code

TWO-SUM(S, x)

1. Sort S

2. left = 0

3. right = n - 1

4. while left < right5. sum = S[left] + S[right]

6. if sum == x

7. return true

8. else if sum < x

9. left = left + 1

10. else

11. right = right – 1

12. return false

**Time Complexity:**

Sorting: Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn)

Two-pointer search: O(n)O(n)O(n)

**Overall:**

Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn) time complexity.

he total worst-case time complexity of the algorithm is:

Θ(nlog⁡n)+O(n)=Θ(nlog⁡n)\Theta(n \log n) + O(n) = \Theta(n \logn)Θ(nlogn)+O(n)=Θ(nlogn)

**CHAPTER 3:**

**CHARACTERIZING RUNNING TIME**

* **PART 1**

**NOTATIONS**

**EXERCIES**

**3.1-1**

**QUESTION**

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

**Explanation**

* The lower-bound argument for insertion sort is typically based on comparisons. For an input array of size , insertion sort has a worst-case time complexity of because each element could potentially need to be compared to all elements before it in a reverse-sorted array.
* For this question, you’re asked to adjust the argument for cases where isn’t a multiple of 3. However, this doesn’t fundamentally change the number of comparisons required for insertion sort; the time complexity will still be even if is not a multiple of 3. We can implement insertion sort in Python and analyze the comparison count for different input sizes.

**Python Code**

def insertion\_sort(arr):

comparisons = 0 # Initialize the comparison counter

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

# Move elements of arr[0..i-1], that are greater than key, to one position ahead

# of their current position

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

comparisons += 1 # Increment comparisons each time a comparison is made

arr[j + 1] = key

comparisons += 1 # Final comparison where arr[j] <= key

return arr, comparisons

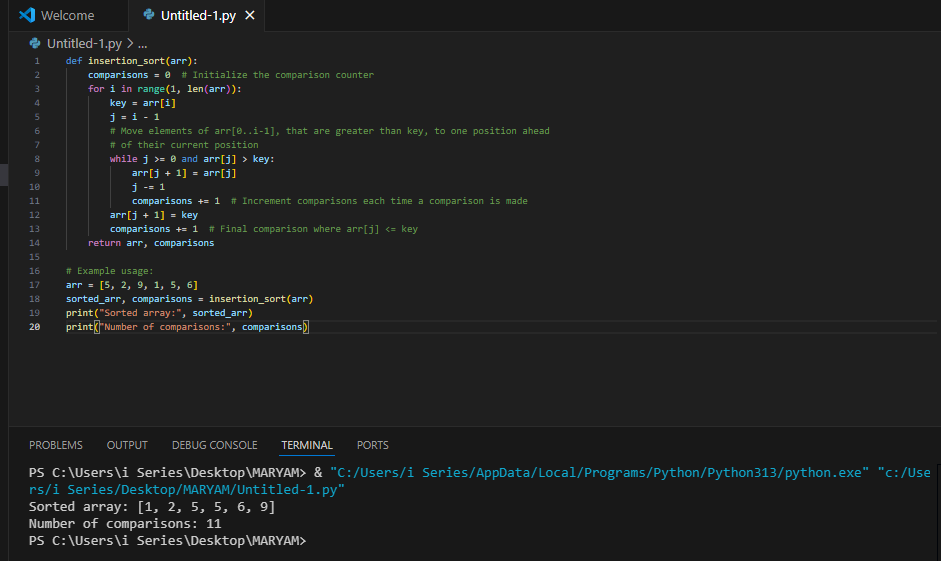
# Example usage:

arr = [5, 2, 9, 1, 5, 6]

sorted\_arr, comparisons = insertion\_sort(arr)

print("Sorted array:", sorted\_arr)

print("Number of comparisons:", comparisons)



**3.1-2**

**QUESTION**

* Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm.

**Explanation**

Selection sort also has a worst-case time complexity of , as it repeatedly searches for the minimum element in the unsorted part of the array and places it in the correct position. For each element, selection sort performs a linear scan to find the minimum, leading to approximately comparisons in total.

* Let's implement selection sort in Python and count the number of comparisons.

**Python Code**

def selection\_sort(arr):

comparisons = 0

for i in range(len(arr)):

min\_idx = i

for j in range(i + 1, len(arr)):

comparisons += 1 # Each comparison to find the minimum

if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i] # Swap the found minimum

return arr, comparisons

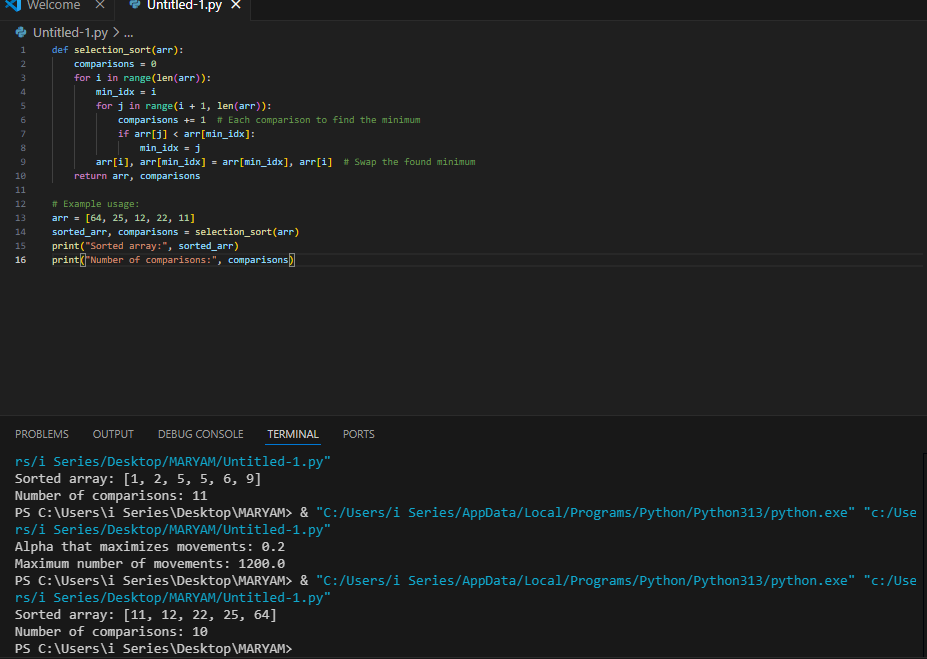
# Example usage:

arr = [64, 25, 12, 22, 11]

sorted\_arr, comparisons = selection\_sort(arr)

print("Sorted array:", sorted\_arr)

print("Number of comparisons:", comparisons)



**3.1-3**

**QUESTION**

* Suppose is a fraction in the range . Show how to generalize the lower-bound argument for insertion sort to consider an input in which the largest values start in the first positions. Determine an additional restriction on , and find the value of that maximizes the number of times the largest values must pass through each of the middle array positions.

**Explanation**

This question requires us to analyze insertion sort in a case where the largest values are placed in the first positions of the array. When largest values are in the front, these values need to be moved towards the end. Each of these elements would require comparisons to move through the middle section of the array, which has a size of .

* To maximize the number of movements, we need to find the value of that maximizes the number of comparisons needed to move the elements through .
* Let’s break down the logic in Python without going into detailed mathematical proof, focusing on simulating different values of .

**Python Code**

def count\_movements(alpha, n):

middle\_section = (1 - 2 \* alpha) \* n

largest\_elements = int(alpha \* n)

movements = 0

# Simulate the movement of the largest elements through the middle section

for i in range(largest\_elements):

movements += middle\_section # Each large element moves through the middle section

return movements

# Test different values of alpha

n = 100 # Example array size

alphas = [i / 10 for i in range(1, 10)] # Test alpha values from 0.1 to 0.9

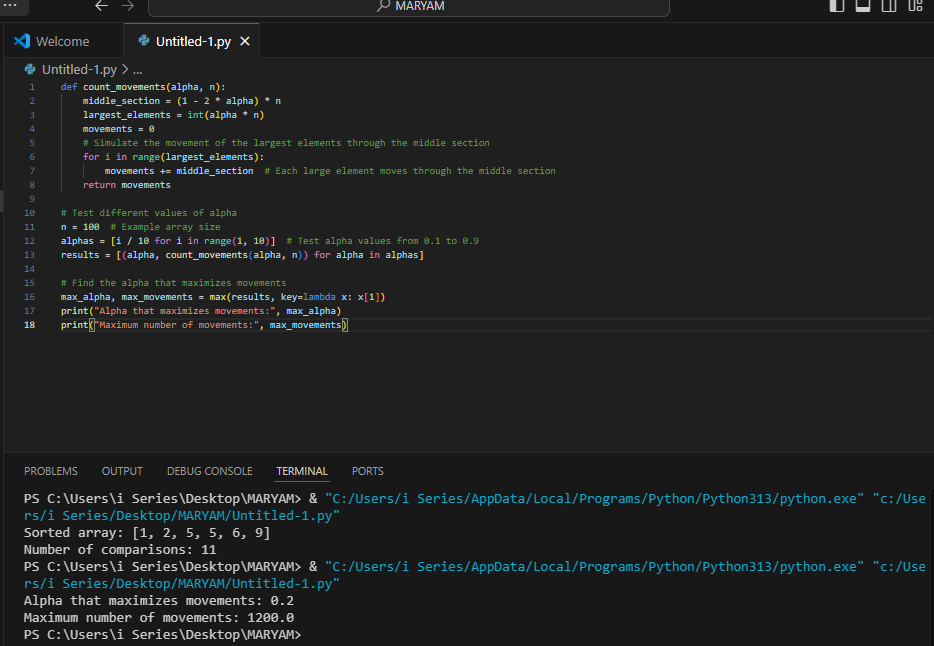
results = [(alpha, count\_movements(alpha, n)) for alpha in alphas]

# Find the alpha that maximizes movements

max\_alpha, max\_movements = max(results, key=lambda x: x[1])

print("Alpha that maximizes movements:", max\_alpha)

print("Maximum number of movements:", max\_movements)



This code tests different values of to find which maximizes the number of movements through the middle section. It’s a way to empirically find the optimal without rigorous proof. The exact behavior may depend on how insertion sort handles comparisons with this specific arrangement, but this code simulates the movements across different values.