1. **Easy: Understanding Bubble Sort : Manually sort the array ` [2, 4, 8, 1, 3] ` using the bubble sort technique. write down each step of the process.**

**Solution:**

**Example:** consider the unsorted array [2,4,8,1,3]

**Step 1**: Given initial array **[2,4,8,1,3]**

Bubble sort with ‘n’ elements require ‘n-1’ passes. Here there are 4 elements so 4-1 = 3 passes required to get the sorted algorithm.

**Pass 1:**

Compare 2 and 4 (no swap required) [2,4,8,1,3]

Compare 4 and 8 (no swap required) [2,4,8,1,3]

Compare 8 and 1(swap)[2,4,1,8,3]

Compare 8 and 3(swap)[2,4,1,3,8]

Now the array after first pass [2,4,1,3,8]

**Pass 2:**

Compare 2 and 4( no swap) [2,4,1,3,8]

Compare 4 and 1(swap)[2,1,4,3,8]

Compare 4 and 3(swap)[2,1,3,4,8]

After second pass the array is [2,1,3,4,8]

**Pass 3:**

Compare 2 and 1 (swap)[1,2,3,4,8]

Compare 2 and 3(Swap)[1,2,3,4,8]

After third pass the sorted array is [1,2,3,4,8]

The final sorted array after all 3 required passes is**[1,2,3,4,8]**

1. **Intermediate: Trace the Bubble Sort Provided the unsorted array ` [3,4,2,1,8] ` and trace the bubble sort algorithm step by step, showing the changes in the array after each pass.**

**Solution :**

Given initial array is **[3,4,2,1,8],** Taking ‘t’ as temporary variable

**Step 1:** the size of the array is 5

Compare 3 and 4 (no swap) **[3,4,2,1,8]**

Compare 4 and 2 (swap**)[3,2,4,1,8],** assign t=2.

Compare 4 and 1(swap**)[3,2,1,4,8]** assign t=1

Compare 4 and 8(no swap**)[3,2,1,4,8]**

Compare 3 and 2(swap)[**2,3,1,4,8]** assign t=2

Compare 3 and 1(swap**)[2,1,3,4,8]** assign t=1

Compare 3 and 4 $ 4 and 8(no swap**)[2,1,3,4,8]**

Compare 2 and 1(Swap**)[1,2,3,4,8]** assign t=1

Compare 2 and 3 & 3 and 4 & 4 and 8(no swap**)[1,2,3,4,8]**

The final sorted array is **[1,2,3,4,8]**

**3. Intermediate: Code ImplementationImplement the bubble sort algorithm in C++. Provide them with the following unsorted array: ` [4,3,2,1,5] `. Code from scratch and test it to ensure it works correctly.**

#include <iostream>

using namespace std;

int main(){

int a[]={4,3,2,1,5};

int getArrayLength = sizeof(a) / sizeof(int);

for (int i = 0; i < getArrayLength; i++)//up unitl the length

{

for(int j = 0;j<getArrayLength-1; j++){//up until last but one

if(a[j]>a[j+1]){

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

//printing

for (int i=0;i<getArrayLength;i++){

cout<<a[i];

}

}

**4. Advanced: Optimization Challenge**

**Challenge yourself to optimize the bubble sort algorithm. Provided with the partially sorted array ` [8,2,3,4,1] `. Optimize the algorithm to reduce the number of comparisons or swaps, making the sorting process more efficient.**

**Solution:**

Optimization is the process of making changes to an algorithm to increase its efficiency, reduce the number of operations, and improve overall performance. Optimization for the Bubble Sort algorithm usually entails cutting down on pointless swaps and comparisons.

// Optimized implementation of Bubble sort

#include <bits/stdc++.h>

using namespace std;

// An optimized version of Bubble Sort

void bubbleSort(int arr[], int n)

{

int i, j;

bool swapped;

for (i = 0; i < n - 1; i++) {

swapped = false;

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

swapped = true;

}

}

// If no two elements were swapped

// by inner loop, then break

if (swapped == false)

break;

}

}

// Function to print an array

void printArray(int arr[], int size)

{

int i;

for (i = 0; i < size; i++)

cout << " " << arr[i];

}

// Driver program to test above functions

int main()

{

int arr[] ={ 8,2,3,4,1 };

int N = sizeof(arr) / sizeof(arr[0]);

bubbleSort(arr, N);

cout << "Sorted array: \n";

printArray(arr, N);

return 0;

}

**5. Advanced: Comparison with Other Sorting Algorithms**

**Compare the bubble sort algorithm with quicksort and merge sort. Discuss the advantages and disadvantages of bubble sort in different scenarios. Additionally, analyse when it might be preferable to use other sorting algorithms.**

**Solution :**

**Bubble Sorting**

**Benefits**: Bubble Sort is easy to understand and construct because of its straightforward implementation.

**In-Place Sorting**: This sorting technique is memory economical because it doesn't require any extra memory.

**Adaptive**: It performs effectively with arrays that are almost or partially sorted.

**Cons:** Bubble Sort is inefficient for large datasets because of its O(n^2) worst-case time complexity.

Despite being simple to use, Bubble Sort is rarely utilized for sorting large datasets because of its inefficiency.

**Preference scenarios**: Bubble Sort's ease of use makes it perfect for small datasets or teaching applications.

**Nearly Sorted Data**: It might function well with data that has been partially or almost sorted.

**Quicksort:**

**Benefits**:

**Efficiency:** Quicksort is incredibly efficient, with an average time complexity of O(n log n).

Quicksort is a memory-saving in-place sorting technique.

**Adaptability**: Because it is a flexible method, it performs well even when the data is only halfway sorted.

**Cons**: Given efficient pivot selection methods, Quicksort's worst-case time complexity is O(n^2), which is unusual.

**Stability:** The sorting algorithm Quicksort is not stable.

Possible Preference Scenario:

**Big Datasets**: Because of its efficiency, Quicksort is the best option for sorting enormous datasets.

**Randomized Data**: It works best with datasets that are diverse or randomized**.**

**Mergesort:**

**Benefits**

A reliable sorting technique that maintains equal elements in the same relative order is mergesort.

Its O(n log n) time complexity ensures predictable and efficient performance even with large datasets.

**External Sorting**: To sort data that is too big to fit in main memory, use Mergesort.

**Cons**: Mergesort frequently requires additional RAM for merging, which in certain circumstances may be a drawback.

**Complexity:** Compared to simpler algorithms like Bubble Sort, it could be more challenging to implement.

**Possible Preference Scenario:**

**enormous Datasets with Limited Memory:** When a dataset is large and there is sufficient memory available for the merging step, mergesort can be helpful.

**External Sorting**: This method is suitable in scenarios where external storage is needed for sorting.

**Analysis:**

**Bubble Sort** is easy to use and works well with small datasets or data that is almost sorted. not very effective with large datasets.

Although **Quicksort** has a drawback, it is effective for large datasets, particularly those that are varied or random.

Although it needs more resources, **mergesort** is perfect for large datasets since it is reliable and steady.