INTRO TO DATA STRUCTURES CS-501-J

Assignment 1

1. **Easy: Understanding Bubble Sort : Manually sort the array ` [22, 24, 8, 11, 3] ` using the bubble sort technique. write down each step of the process.**

Solution:

Bubble Sort:

Definition: Bubble Sort is a simple sorting algorithm technique used to sort the adjacent elements repeatedly if they are in wrong order (not in ascending order) and placing those elements in ascending order.

Working: Bubble sort is a step-by-step process starting from the first element it compares the adjacent element and swaps the element if they are in wrong order. This process continues until the entire list is sorted. During each iteration the largest unsorted element gradually takes place at the end of the list.

Example: consider the unsorted array [22,24,8,11,3]

Step 1: Given initial array **[22,24,8,11,3]**

Bubble sort with 'n' elements require 'n-1' passes. Here there are 5 elements so 5-1 = 4 passes required to get the sorted algorithm.

Pass 1:

Compare 22 and 24 (no swap required) [22,24,8,11,3]
Compare 24 and 8 (swap required as 24 is greater than 8) [22,8,24,11,3]
Compare 24 and 11(swap)[22,8,11,24,3]
Compare 24 and 3(swap)[22,8,11,3,24]
Now the array after first pass [22,8,11,3,24]

Pass 2:

Compare 22 and 8(swap)[8,22,11,3,24]
Compare 22 and 11(swap)[8,11,22,3,24]
Compare 22 and 3(swap)[8,11,3,22,24]
Compare 22 and 24(no swap)[8,11,3,22,24]
After second pass the array is [8,11,3,22,24]

Pass 3:

Compare 8 and 11(no swap)[8,11,3,22,24]
Compare 11 and 3(Swap)[8,3,11,22,24]
Compare 22 and 24(no swap)[8,3,11,22,24]
After third pass the sorted array is [8,3,11,22,24]

Pass4:

Compare 8 and 3(Swap)[3,8,11,22,24]
Compare 8 and 11 , 11 and 22 , 22 and 24(no swap)[3,8,11,22,24]
The final sorted array after all 4 required passes is**[3,8,11,22,24]**

2. **Intermediate: Trace the Bubble Sort Provided the unsorted array ` [5, 3, 1, 12] ` and trace the bubble sort algorithm step by step, showing the changes in the array after each pass.**

Solution :
Given initial array is **[5,3,1,12]**
Step 1: the size of the array is 4
a[0] = 5, a[1] = 3, a[2] = 1,a[3] =12
at initial the temporary value ‘t=a[0]’
if a[0]>a[1] then swap a[0] with a[1] then assign it to ‘t’
a[0]=5,a[1]=3 condition true then swap a[0]=3, a[1]=5 now ‘t=3’
after first trace the array is [3,5,1,12]
Step 2: a[0]=3,a[1]=5,a[2]=1,a[3]=12
A[1]>a[2] condition true then swap and assign it to ‘t’
Swap [3,1,5,12] t=1
A[2]>a[3] condition fails no swap and ‘t’ is past value
Swap [1,3,5,12]
The final sorted array is **[1,3,5,12]**

3. Intermediate: Code Implementation

Implement the bubble sort algorithm in C++. Provide them with the following unsorted array: ` [24,12,35,23,45,34,20,48] `. Code from scratch and test it to ensure it works correctly.

```
#include <iostream>
using namespace std;
int main(){

int a[]={24,12,35,23,45,34,20,48};
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];
}
}
```

Explanation:-
int a[]={24,12,35,23,45,34,20,48};
int getArrayLength = sizeof(a) / sizeof(int);
"The array integer 'a' is initialized with given values, and 'getarraylength' is calculated to record the length of the given array”.

```
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;
```

The outer loop 'I' iterates across the array, while the inner loop 'j' compares and swaps elements if they are not in the correct order.
The swapped items are saved as temporary variables using Temp, an integer.

```
for (int i=0;i<getArrayLength;i++){
  cout<<a[i];
```

used to give the output of the sorted array.
Unsorted Array a[] = **[24,12,35,23,45,34,20,48]**
Pass 1 (i=0)
Compare and swap(j=0): [12,24,35,23,45,34,20,48]
Compare and swap(j=1): [12,24,35,23,45,34,20,48]
.....similar comparison and swap (j=2,3,4,5,6,7):[12,24,23,35,34,20,45,48]
Pass 2(i=1)
Compare and swap (j=2):[12,23,24,35,34,20,45,48]
Compare and swap(j=3):[12,23,24,35,34,20,45,48]
.....similar comparison and swap (j=2,3,4,5):[[12,23,24,35,34,20,45,48]
Compare and swap(j=6):[12,20,23,24,35,34,45,48]
Compare (j=7):no swap
Pass 3 (i=2)
No swaps required:[12,20,23,24,35,34,45,48]
Pass 4 (i=3)
No swaps required:[12,20,23,24,35,34,45,48]
Pass 5 (i=4)
compare and swap(j=5):[12,20,23,24,34,35,45,48]
pass 6(i=5) & pass 7(i=6)
no swaps required the array is sorted **: [12,20,23,24,34,35,45,48]**

4. Advanced: Optimization Challenge

Challenge yourself to optimize the bubble sort algorithm. Provided with the partially sorted array ` [1, 2, 3, 4, 5, 10, 9, 8, 7, 6] `. 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[] ={ 6, 4, 5, 2, 9, 1, 0 };
	int N = sizeof(arr) / sizeof(arr[0]);
	bubbleSort(arr, N);
	cout << "Sorted array: \n";
	printArray(arr, N);
	return 0;
```

}
The code includes a standard C++ library header, along with the majority of other standard headers.
The bubbleSort function implements the Bubble Sort algorithm in its optimized form.
It accepts an array arr[] and a length n as inputs.
The outer loop (i) represents each iteration of the array, while the inner loop (j) does pairwise comparison and swapping.
The swapped variable is used to determine whether any swaps were performed during a pass.
If no swaps were made by the inner loop, the algorithm exits immediately.
The printArray function is a basic utility function that prints the items of an array.
The main function determines the length of an array and initializes it with values.
Uses the bubbleSort function to sort the array.
Prints the sorted array using the printArray() method.

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 Sort:

Advantages:

Bubble Sort has a simple implementation, making it easy to comprehend and create.

In-Place Sorting: It requires no additional memory (in-place sorting), making it memory efficient.

Adaptive: It works well with partially or nearly sorted arrays.

Disadvantages:

Bubble Sort's worst-case time complexity is O(n^2), making it inefficient for huge datasets.

In fact, Bubble Sort is rarely used for sorting huge datasets due to its inefficiency, despite its ease of use.

Preference scenarios:

Bubble Sort's simplicity makes it ideal for tiny datasets or educational purposes.

Nearly Sorted Data: It may work effectively with partially or nearly sorted data.

Quicksort:

Advantages:

Efficiency: Quicksort has an average time complexity of O(n log n), making it extremely efficient.

Quicksort is an in-place sorting method that saves memory.

Adaptability: It can be used as an adaptable algorithm, which means it works well with partially sorted data.

Disadvantages:

Quicksort's worst-case time complexity is O(n^2), which is uncommon given effective pivot selection algorithms.

Stability: Quicksort is not a stable sorting algorithm.

Scenario for Preference:

Large Datasets: Quicksort is preferable for sorting large datasets due to its efficiency.

Randomized Data: It performs well with randomized or diversified datasets.

Mergesort:

Advantages

Mergesort is a stable sorting method that keeps equal elements in the same relative order.

Predictable Performance: It has O(n log n) time complexity, which makes it predictable and efficient for huge datasets.

External Sorting: Mergesort can be used to sort data that is too vast to fit in main memory.

Disadvantages:

Mergesort often uses more memory for merging, which can be a disadvantage in some cases.

Complexity: It may be more difficult to implement than simpler algorithms like Bubble Sort.

Scenario for Preference:

huge Datasets with Limited Memory: Mergesort is useful when the dataset is huge and there is enough memory for the merging stage.

External Sorting: It is appropriate for situations where sorting requires external storage.

Analysis:

Bubble Sort is simple and ideal for tiny datasets or nearly sorted data. Inefficient with huge datasets.

Quicksort is efficient for huge datasets, especially when they are random or diversified, but it has a downside.

Mergesort is stable and predictable, making it ideal for huge datasets, but it requires more resources.

B . Divya Rajya Lakshmi
0938835