Data Structure Algorithms & Applications (CT-159)
Lab 06

Elementary Sorting Techniques

Objectives

The objective of the lab is to get students familiar with Bubble sort, insertion sort, and selection sort techniques.

Tools Required

Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Introduction

A sorting algorithm is a procedure used to arrange the elements of a list in a particular order, typically either ascending or descending. Sorting algorithms optimize other algorithms that require sorted data, like search algorithms or data analysis techniques.

Selection Sort

Selection sort is a simple, comparison-based sorting algorithm that divides the input array into two parts: the sorted part and the unsorted part. The sorted part is built up from left to right at the front of the array, while the unsorted part occupies the rest. The algorithm repeatedly selects the

smallest (or largest, depending on sorting order) element from the unsorted part and swaps it with the first element of the unsorted part, thus expanding the sorted part by one element.

Selection Sort Algorithm

1. Start with the first element (index 0) and consider it the minimum.
2. Search the unsorted part of the array to find the smallest element.
3. Swap the smallest element with the first element of the unsorted part.
4. Move the boundary between the sorted and unsorted parts one element to the right.
5. Repeat the process for the next position until the entire array is sorted.

Time Complexity: Best case: $O(n^2)$, Worst case: $O(n^2)$, Average case: $O(n^2)$

Space Complexity: O(1) (In-place sorting)

Properties:

- Not stable: It does not preserve the relative order of elements with equal keys.
- Not adaptive: It performs the same number of comparisons regardless of the input's initial order.

Selection sort is easy to implement but inefficient on large lists, making it more suitable for small datasets or educational purposes.

Example 01: A C++ program for implementation of selection sort

```
#include <iostream>
using namespace std;
//Swap function
void swap(int &xp, int &yp){
	int temp = xp;
	xp = yp;
	yp = temp;
}
void selectionSort(int* arr, int n){
	int i, j, min_idx;
	// One by one move boundary of unsorted subarray
	for (i = 0; i < n-1; i++){
	// Find the minimum element in unsorted array
		min_idx = i;
		for (j = i+1; j < n; j++)
			if (arr[j] < arr[min_idx])
				min_idx = j;
		// Swap the found minimum element with the first element
		if(min_idx!=i)
			swap(arr[min_idx], arr[i]);  }
}
//Function to print an array
void printArray(int *arr, int size){
	int i;
	for (i=0; i < size; i++)
		cout << arr[i] << " ";
	cout << endl;  }
// Driver program to test above functions
int main(){
	int arr[] = {64, 25, 12, 22, 55, 4, 11};
	int n = sizeof(arr)/sizeof(arr[0]);
	selectionSort(arr, n);
	cout << "Sorted array: \n";
```

```
        printArray(arr, n);
        return 0;
}
```

Bubble Sort

Bubble sort is a simple, comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted. The algorithm gets its name because larger elements "bubble" to the top (end of the list) with each pass through the array.

Bubble Sort Algorithm

1. Start at the beginning of the array.
2. Compare each pair of adjacent elements.
3. If the current element is greater than the next one (for ascending order), swap them.
4. Repeat the process for each pair until you reach the end of the list, ensuring the largest element is in its correct position.
5. After each complete pass through the list, the next largest element will have moved to its correct position, so the next pass can ignore the last sorted elements.
6. Continue repeating passes until no swaps are needed, indicating the list is fully sorted.

Time Complexity: Best case (already sorted): O(n) (optimized version that stops early if no swaps are made), Worst case: $O(n^2)$, Average case: $O(n^2)$

Space Complexity: O(1) (In-place sorting)

Properties:

- Stable: It preserves the relative order of elements with equal keys.
- Not adaptive (in its basic form): Although an optimized version can detect an already sorted array, the standard algorithm does not take advantage of existing order.

Bubble sort is primarily useful for educational purposes and small datasets, but it's inefficient for large datasets due to its quadratic time complexity.

Example 02: A C++ program for implementation of Bubble sort

```
#include <iostream>
using namespace std;
// A function to implement bubble sort
void bubbleSort(int* arr, int n){
        int i, j;
        for (i = 0; i < n - 1; i++)
                // Last i elements are already in place
                for (j = 0; j < n - i - 1; j++)
                        if (arr[j] > arr[j + 1])
                                swap(arr[j], arr[j + 1]);
}
// Function to print an array
void printArray(int* arr, int size){
        int i;
        for (i = 0; i < size; i++)
                cout << arr[i] << " ";
        cout << endl;  }
// Driver code
int main(){
        int arr[] = { 5, 1, 4, 2, 8};
        int N = sizeof(arr) / sizeof(arr[0]);
        bubbleSort(arr, N);
        cout << "Sorted array: \n";
        printArray(arr, N);
        return 0;
}
```

Insertion sort

Insertion sort is a straightforward, comparison-based sorting algorithm that builds the sorted array one element at a time. It works by repeatedly taking one element from the unsorted portion of the array and inserting it into its correct position within the sorted portion.

Insertion Sort Algorithm

1. Start with the second element (index 1) of the array. Consider the first element (index 0) as the sorted portion.
2. Compare the current element with the elements in the sorted portion.
3. Shift all elements in the sorted portion that are larger than the current element to the right.
4. Insert the current element into its correct position.
5. Repeat this process for each element in the unsorted portion until the entire array is sorted.

Time Complexity: Best case: O(n) (when the array is already sorted), Worst case: $O(n^2)$ (when the array is sorted in reverse order), Average case: $O(n^2)$

Space Complexity: O(1) (In-place sorting)

Properties:

- Stable: It preserves the relative order of elements with equal keys.
- Adaptive: Performs better on nearly sorted or small data sets.

Insertion sort is most efficient for small or nearly sorted data, and it is commonly used as a building block for more complex algorithms like Timsort.

Example 03: A C++ program for insertion sort

```
#include <iostream>
using namespace std;
// Function to sort an array using insertion sort
void insertionSort(int* arr, int n){
	int i, key, j;
	for (i = 1; i < n; i++){
		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 && arr[j] > key){
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}
void printArray(int* arr, int n){
	int i;
	for (i = 0; i < n; i++)
		cout << arr[i] << " ";
	cout << endl;
}
int main(){
	int arr[] = { 12, 11, 13, 5, 6 };
	int N = sizeof(arr) / sizeof(arr[0]);
	insertionSort(arr, N);
	printArray(arr, N);
	return 0;
}
```

Visualization of the above mentioned sorting techniques is available at :
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Exercise

1. If the array is already sorted, we don't want to continue with the comparisons. This can be achieved with modified bubble sort. Update the code in example 02 to have a modified bubble sort function.

2. Given an array **arr[ ]** of length **N** consisting cost of **N** toys and an integer **K** the amount with you. The task is to find maximum number of toys you can buy with **K** amount.
   Test Case: Input: N = 7, K = 50, arr[] = {1, 12, 5, 111, 200, 1000, 10}, **Output:** 4
   Explanation: The costs of the toys. You can buy are 1, 12, 5 and 10.

3. Create a single class Sort, which will provide the user the option to choose between all 3 sorting techniques. The class should have following capabilities:
- Take an array and a string (indicating the user choice for sorting technique) as input and perform the desired sorting.
- Should allow the user to perform analysis on a randomly generated array. The analysis provides number of comparisons and number of swaps performed for each technique.
- After printing all the results in the main program, highlight the best and worst techniques.

4. Given an array of integers arr, sort the array by performing a series of **pancake flips**. In one pancake flip we do the following steps:
- Choose an integer k where 1 <= k <= arr.length.
- Reverse the sub-array arr[0...k-1] (**0-indexed**).
  For example, if arr = [3,2,1,4] and we performed a pancake flip choosing k = 3, we reverse the sub-array [3,2,1], so arr = [1,2,3,4] after the pancake flip at k = 3. Return *an array of the* k*-values corresponding to a sequence of pancake flips that sort* arr. Any valid answer that sorts the array within 10 * arr.length flips will be judged as correct.
  Example 1: Input: arr = [3,2,4,1], **Output:** [4,2,4,3]
  Explanation: We perform 4 pancake flips, with k values 4, 2, 4, and 3.
  Starting state: arr = [3, 2, 4, 1]
  After 1st flip (k = 4): arr = [1, 4, 2, 3]
  After 2nd flip (k = 2): arr = [4, 1, 2, 3]
  After 3rd flip (k = 4): arr = [3, 2, 1, 4]
  After 4th flip (k = 3): arr = [1, 2, 3, 4], which is sorted.

5. Given an array nums with n objects colored red, white, or blue, sort them inplace so that objects of the same color are adjacent, with the colors in the order red, white, and blue. We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively. You must solve this problem by writing a sort function.
   Example 1: Input: nums = [2,0,2,1,1,0], **Output:** [0,0,1,1,2,2]
   Example 2: Input: nums = [2,0,1], **Output:** [0,1,2]

Lab 04 Evaluation		
Student Name:		**Student ID:** **Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		