|  |  |
| --- | --- |
| **Data Structures & Algorithms**  Diploma in IT, CSF  Year 2 (2024/25) Semester 4 | **Week 14** |
| **1 Hour** |
| **Tutorial 11 – Sorting** | |

1. Trace the insertion sort as it sorts the following array into ascending order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 20 | 80 | 40 | 25 | 60 | 40 |

* Compare second element 80 with 20, no need swap as 80 > 20
* Compare third element 40 with 80, move 80 to the right [20, 40, 80, 25, 60, 40]
* Compare fourth element 25 with 80, 40, 20. Move 80 and 40 to the right, insert 25 [20, 25, 40, 80, 60, 40]
* Compare the fifth element 60, compare 60 with 80, move 80 to the right and insert 60 [20, 25, 40, 60, 80, 40]
* Compare the sixth element 40, compare with 80, 60, 40, move 80 and 60 to the right, insert 40 [20, 25, 40, 40, 60, 80]

1. Trace the selection sort as it sorts the following array into ascending order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 7 | 12 | 24 | 4 | 19 | 32 |

* Find the smallest element in the array (4), swap it with 1st element (7) [4, 12, 24, 7, 19, 32]
* Find the smallest element in the remaining array (7), swap it with the 2nd element (12) [4, 7, 24, 12, 19, 32]
* Find the smallest element in the remaining array (12), swap it with the 3rd element (24) [4, 7, 12, 24, 19, 32]
* Find the smallest element in the remaining array (19), swap it with the 4th element [4, 7, 12, 19, 24, 32]
* Find the smallest element in the remaining array (24), since it is in correct position, no swap needed
* The last element is in correct position
* [4, 7, 12, 19, 24, 32]

1. Apply the selection sort and insertion sort to
2. An inverted array

|  |  |  |  |
| --- | --- | --- | --- |
| 8 | 6 | 4 | 2 |

Selection Sort:

Step 1:

* Find the smallest element (2) in the entire array
* Swap it with the first element (8) – [2, 6, 4, 8]

Step 2:

* Find the smallest element (4) in the remaining array [6, 4. 8]
* Swap it with the second element (6) – [2, 4, 6, 8]

Step 3:

* Find the smallest element (6) in the remaining array [6, 8]
* Since it is already in the correct position, no swap is needed [2, 4, 6, 8]

Step 4:

* The last element (8) is already in the correct position [2, 4, 6, 8]

Insertion Sort:

Step 1:

* Compare the second element (6) with the first element (8)
* Swap them since 6 < 8 – [6, 8, 4, 2]

Step 2:

* Compare the third element (4) with 8 and 6
* Shift 8 and 6 to the right, and insert 4 in the correct position [4, 6, 8, 2]

Step 3:

* Compare the fourth element (2) with 8, 6 and 4
* Shift 8, 6, 4 to the right, and insert 2 in the correct position [2, 4, 6, 8]

1. An ordered array

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 6 | 8 |

Selection Sort:

Step 1:

* Find the smallest element (2) in the entire array
* No swap needed since already correct position

Step 2:

* Find the second smallest element (4) in the remaining array [4, 6, 8]
* No swap needed

Step 3:

* Find the smallest element (6) in the remaining array [6, 8]
* No swap needed

Step 4:

* The last element (8) is already in the correct position

Insertion Sort:

Step 1:

* Compare the second element (4) with the first element (2).
* No swap is needed since 4 > 2. Result: [2, 4, 6, 8]

Step 2:

* Compare the third element (6) with the previous elements (4 and 2).
* No swap is needed since 6 > 4. Result: [2, 4, 6, 8]

Step 3:

* Compare the fourth element (8) with the previous elements (6, 4, and 2).
* No swap is needed since 8 > 6. Result: [2, 4, 6, 8]

What can you conclude about the differences in performance between insertion sort and selection sort based on the order of the items?

Selection sort performs unnecessary comparisons regardless of input order, time complexity stays as O(n^2). Insertion sort detects the array is already sorted and avoids unnecessary shifts, time complexity approaches O(n).

1. Trace the mergesort algorithm as it sorts the following array into ascending order.

List the calls to mergesort and to merge in the order they occur.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 20 | 80 | 40 | 25 | 60 | 30 |

Step 1:

* Split the array into two halves : [20, 80, 40] and [25, 60, 30]
* Call 1 – mergeSort([20, 80, 40])
* Call 2 – mergeSort([25, 60, 30])

Step 2: Left Half - [20, 80, 40]

1. Split [20, 80, 40] into [20] and [80, 40]

* Call 3 – mergeSort([20]) -> Base Case
* Call 4 – mergeSort([80, 40])

1. Split [80, 40] into [80] and [40]

* Call 5 – mergeSort([80]) -> Base Case
* Call 6 – mergeSort([40]) -> Base Case

1. Merge [80] and [40] into [40, 80]
   * Merge 1 – merge([80], [40]) -> [40, 80]
2. Merge [20] and [40, 80] into [20, 40, 80]
   * Merge 2 – merge([20], [40, 80]) -> [20, 40, 80]

Step 3: Right Half - [25, 60, 30]

1. Split [25, 60, 30] into [25] and [60, 30]
   * Call 7 – mergeSort([25]) –> Base Case
   * Call 8 – mergeSort([60, 30])
2. Split [60, 30] into [60] and [30]
   * Call 9 – mergeSort([60]) -> Base Case
   * Call 10 – mergeSort([30]) -> Base Case
3. Merge [60] and [30] into [30, 60]
   * Merge 3 – merge([60], [30]) -> [30, 60]
4. Merge [25] and [30, 60] into [25, 30, 60]
   * Merge 4 – merge([25], [30, 60]) -> [25, 30, 60]

Final Merge:

Merge [20, 40, 80] and [25, 30, 60] into [20, 25, 30, 40, 60, 80]

Merge 5 -> merge([20, 40, 80], [25, 30, 60]) -> [20, 25, 30, 40, 60, 80]

1. When sorting an array by mergesort,
2. Do the recursive calls to mergesort depend on the values in the array, the number of items in the array, or both. Explain.

* Recursive calls to merge sort depend only on the number of items in the array, not values in the array.

Structure of Algorithm:

* Merge sort divides the array into two halves repeatedly, regardless of the actual values in the array
* Recursive calls are determined by the size of the array: If the array has more than one element, it is split into smaller subarrays

No Dependence on Values:

* Values of the elements do not affect how the arrays is divided. Each split is based purely on the size of the array, with the left and right halves processed recursively

1. In what step of mergesort are the items in the array actually swapped (that is sorted)? Explain.

Items in the array are actually swapped (that is , sorted) during the merge step of mergesort

1. Merge Step:
   * After the array is recursively divided into single-element subarrays, the merge step combines these subarrays back into sorted order
   * At each level of recursion, two sorted subarrays are merged into one larger sorted array
   * During this merging process, elements are compared and placed in their correct positions in a array, effectively sorting them
2. Sorting does not happen during division:
   * Splitting of the array during recursive calls only prepares the array for sorting: no comparisons or swaps occur during this phase
   * Sorting only happens when the subarrays are combined in the merge step
3. Discuss the performance of the mergesort algorithm.

* Merge Sort has a consistent time complexity of O (n log n) across all cases, making it reliable and predictable
* Major trade-off is its space complexity of O(n), which makes it less suitable for scenarios where memory is a concern
* Most efficient and stable sorting algorithms, especially for large data sets or linked lists

1. Trace the quicksort as it sorts the following array into ascending order. List the calls to quickSort and to partition in the order in which they occur.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 20 | 80 | 40 | 25 | 60 | 10 | 15 |

Step 1: Partition entire array

* Pivot: 15 (Last element)
* Rearrange so that elements smaller than 15 are on the left, and elements greater than 15 are on the right
* Result after partitioning -> [10, 15, 40, 25, 60, 80, 20]
* Pivot 15 is now in its correct position (index 1)

Call 1: partition([20, 80, 40, 25, 60, 10, 15]) -> [10, 15, 40, 25, 60, 80, 20]

Next Calls: quicksort([10]) (left) and quicksort([40, 25, 60, 80, 20]) (right)

Step 2: QuickSort on the right subarray [40, 25, 60, 80, 20]

* Pivot: 20 (Last element)
* Rearrange so that elements smaller than 20 are on the left
* Result after partitioning -> [20, 25, 60, 80, 40]
* Pivot 20 is now in its correct position (index 2)

Call 2: partition([40, 25, 60, 80, 20]) -> [20, 25, 60, 80, 40]

Next Calls: quicksort([]) (left) and quicksort([25, 60, 80, 40]) (right)

Step 3: QuickSort on [25, 60, 80, 40]

* Pivot: 40 (last element)
* Rearrange so that elements smaller than 40 are on the left
* Result after partitioning -> [25, 40, 80, 60]
* Pivot 40 is now in its correct position (index 3)

Call 3: partition([25, 60, 80, 40]) -> [25, 40, 80, 60]

Next calls: quicksort([]) (left) and quicksort([80, 60]) (right)

Step 4: QuickSort on [80, 60]

* Pivot: 60 (last element)
* Rearrange so that elements smaller than 60 are on the left
* Result after partitioning -> [60, 80]
* Pivot 60 is now in its correct position (index 4)

Call 4: partition([80, 60]) -> [60, 80]

Next calls: quickSort([]) (left) and quickSort([]) (iright)

Final Sorted Array: [10, 15, 20, 25, 40, 60, 80]

1. You can choose any array item as the pivot for quicksort. Simply interchange items so that your pivot is in theArray[first]. One way to choose a pivot is to take the middle value of the three values theArray[first], theArray[(first + last)/2, and theArray[last]. How many recursive calls are necessary to sort an array of size n if you always choose the pivot in this way?
2. Choosing the median point:

* By choosing the median of the first, middle and last elements as the pivot, the pivot is more likely to be closer to the middle value of the array
* Strategy helps reduce the likelihood of encountering worst-case scenarios, such as when pivot consistently partitions the array into highly unbalanced halves (e.g, one side empty and the other containing nearly all elements)

1. Number of recursive calls:

* Recursive calls depend on how the array is partitioned at each step
* Ideally, median-of-three pivot selection tends to divide the array into two nearly equal halves

Best Case: log n recursive calls

Worst Case: n recursive calls (less likely)

Average Case: log n recursive calls

1. Describe the case in which quick sort has the worst performance.

Happens when pivot chosen at each step results in highly unbalanced partitions. Leads to degenerate case where one partition contains nearly all the elements, while other is empty or contains very few elements. QuickSort behaves like a less efficient sorting algorithm with O(n^2) time complexity

Case for worst performance:

1. Pivot selection issues:
   1. If the pivot is consistently the smallest or largest element of the array (or close to it) during every partition
   2. This can happen if the array is already sorted (ascending or descending) and the pivot is always chosen as:
      1. First element
      2. Last element
      3. Another extreme value of the array (first or last in subarray)
2. Unbalanced Partitions:
   1. After each partition, one partition contains n-1 elements, and the other contains only 1 or 0 elements
   2. Creates recursion tree with a depth of n, resulting in O(n^2) comparisons

Therefore:

* Array already sorted or nearly sorted
* Consistently chooses poor pivot -> unbalanced partitions