Name : Deshmukh Mehmood Rehan

MIS No. : 612303050

SY Computer Science – Div 1

Searching Algorithms:

Bubble Sort:

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

Implementation

```
void bubble_sort(int arr[], int n){
    int i, j;
    for(i = 0; i < n; i++){
        int swapped = 0;
        for(j = 0; j < n - 1 - i; j++){
            if(arr[j] > arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
                swapped = 1;
            }
        }
        if(!swapped) break;
    }
}
```

Time Complexity

Best Case: O(n) : When the list is already sorted.

Average Case: O(n^2) : When the list is in random order.

Worst Case: O(n^2) : When the list is in reverse order.

Performance

Data Set Size	Time (ms)
100	3.069218
200	2.520787
500	5.269694
1000	6.446332
2000	17.133653
5000	53.386605
10000	143.240873
20000	690.219696
50000	4926.238435

Insights

Bubble sort is one of the slowest sorting algorithms, with a time complexity of O(n^2) in the worst and average cases. It is not suitable for large data sets due to its poor performance. The algorithm performs better on nearly sorted data.

Selection Sort

Selection sort is an in-place comparison sorting algorithm that divides the input list into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted. The algorithm selects the smallest element from the unsorted sublist and swaps it with the leftmost unsorted element. The selection sort algorithm is simple and easy to implement, but it is inefficient on large lists and generally performs worse than insertion sort.

Implementation

```
void selection_sort(int arr[], int n){
    int i, j, min;
    for(i = 0; i < n; i++){
        min = i;
        for(j = i+1; j < n; j++){
            if(arr[j] < arr[min]) min = j;
        }
        swap(&arr[min], &arr[i]);
    }
}
```

Time Complexity

Best Case: O(n^2) : When the list is in reverse order.

Average Case: O(n^2) : When the list is in random order.

Worst Case: O(n^2) : When the list is already sorted.

Performance

Data Set Size	Time (ms)
100	2.562486
200	3.856803
500	3.774940
1000	7.764609
2000	16.781381
5000	37.272564
10000	61.085738
20000	184.295726
50000	1106.673971

Insights

Selection sort is another slow sorting algorithm with a time complexity of O(n^2) in all cases. It is not suitable for large data sets due to its poor performance. The algorithm performs better on nearly sorted data compared to random or reverse-sorted data.

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted list one item at a time. It iterates through the list, removing one element at a time and finding the correct position to insert it in the sorted part of the list. The algorithm is efficient for small data sets and is more efficient than bubble sort and selection sort.

Implementation

```
void insertion_sort(int arr[], int n){
    int i, j;
    for(i = 1; i < n; i++){
        int key = arr[i];
        j = i - 1;
        while(j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Time Complexity

Best Case: O(n) : When the list is already sorted.

Average Case: O(n^2) : When the list is in random order.

Worst Case: O(n^2) : When the list is in reverse order.

Performance

Data Set Size	Time (ms)
100	2.492995
200	2.588347
500	2.908154
1000	4.759833
2000	13.697476
5000	32.283147
10000	42.610466
20000	145.498880
50000	788.663037

Insights

Insertion sort is more efficient than bubble sort and selection sort, with a time complexity of O(n^2) in the worst and average cases. It is suitable for small data sets and nearly sorted data. The algorithm performs better on nearly sorted data compared to random or reverse-sorted data.

Quick Sort

Quick sort is a comparison-based sorting algorithm that uses a divide-and-conquer strategy to sort the elements. It picks an element as a pivot and partitions the array around the pivot, such that all elements smaller than the pivot are on the left, and all elements larger than the pivot are on the right. The algorithm then recursively sorts the subarrays on the left and right of the pivot. Quick sort is efficient and widely used due to its average-case time complexity of O(n log n).

Implementation

```
void partition(int arr[], int n, int *pivot){
    int i = 1, j = n - 1;
    int p = arr[0];
    while(i <= j){
        while(i < n && arr[i] <= p) i++;
        while(j > 0 && arr[j] > p) j--;
        if(i < j) swap(&arr[i], &arr[j]);
    }
    swap(&arr[0], &arr[j]);
```

```
    *pivot = j;
}

void quick_sort(int arr[], int n){
    if(n <= 1) return;
    int pivot;
    partition(arr, n, &pivot);
    quick_sort(arr, pivot);
    quick_sort(arr + pivot + 1, n - pivot - 1);
}
```

Time Complexity

Best Case: O(n log n) : When the pivot divides the array into two equal parts.

Average Case: O(n log n) : When the pivot divides the array into two parts of nearly equal size.

Worst Case: O(n^2) : When the pivot is the smallest or largest element in the array.

Performance

Data Set Size	Time (ms)
100	2.632061
200	2.946879
500	2.827336
1000	4.171350
2000	7.473192
5000	11.589450
10000	11.626805
20000	15.337132
50000	30.941726

Insights

Quick sort is one of the most efficient sorting algorithms with an average-case time complexity of O(n log n). It is suitable for large data sets and performs well on random data. The choice of the pivot element can affect the performance of the quick sort algorithm.

Heapsort

Heapsort is a comparison-based sorting algorithm that uses a binary heap data structure to sort the elements. It builds a max heap from the input array and repeatedly extracts the maximum element from the heap and places it at the end of the array. The algorithm then reduces the heap size and maintains the heap

property to sort the remaining elements. Heapsort has a time complexity of O(n log n) and is an efficient sorting algorithm.

Implementation

```c
void heap_sort_max_heap(int *array, int size){
    max_heap h;
    init_max_heap(&h);

    for(int i = 0; i < size; i++){
        insert_max_heap(&h, array[i]);
    }

    for(int i = size - 1; i >= 0; i--){
        array[i] = remove_max_heap(&h);
    }

    free_max_heap(&h);

    return;
}
```

Time Complexity

Best Case: O(n log n) : When the input array is already a heap.

Average Case: O(n log n) : When the input array is random.

Worst Case: O(n log n) : When the input array is reverse sorted.

Performance

Data Set Size	Time (ms)
100	2.981336
200	4.933448
500	6.524338
1000	7.725174
2000	9.097312
5000	12.463829
10000	18.704151
20000	25.523862
50000	28.418141

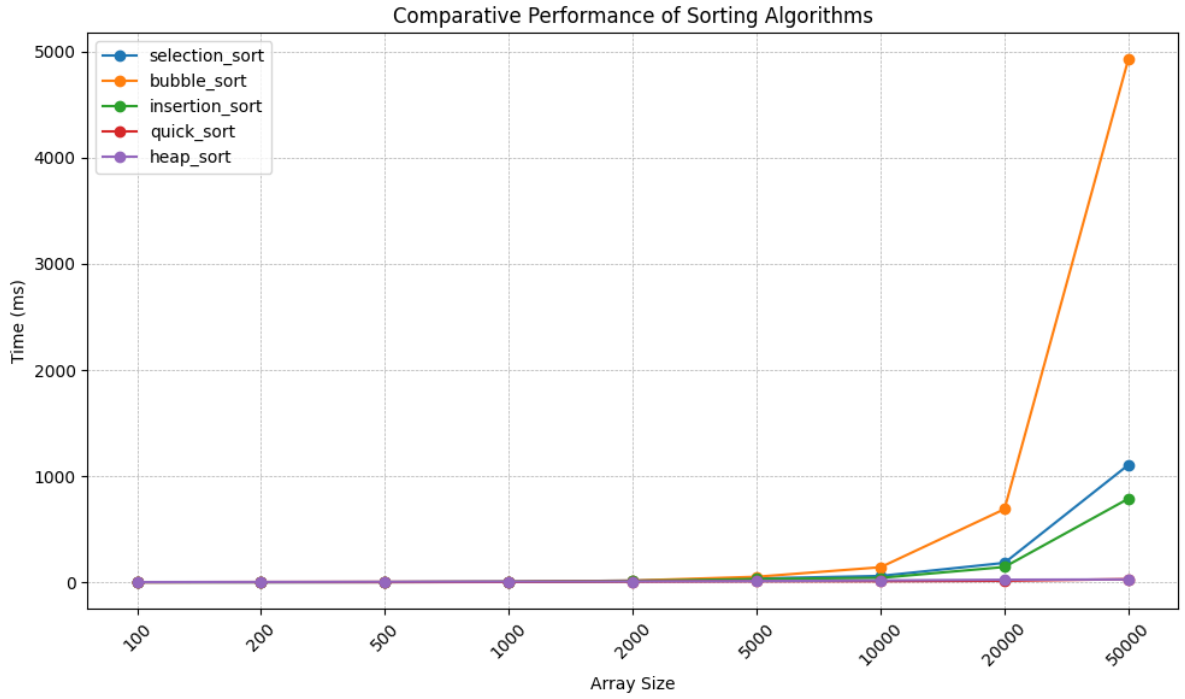
Insights

Heapsort is an efficient sorting algorithm with a time complexity of O(n log n) in all cases. It is suitable for large data sets and performs well on random data. Heapsort is often preferred in systems where memory allocation is a concern due to its in-place sorting nature.

Performance Comparison

The performance of different sorting algorithms was tested on various data sets, including randomly ordered data, nearly sorted data, and reverse sorted data. The time taken by each algorithm on each data set was recorded and analyzed.

Comparative Performance



Observations

Bubble sort and selection sort are the slowest sorting algorithms with a time complexity of O(n^2) in all cases.

Insertion sort is more efficient than bubble sort and selection sort, especially on nearly sorted data.

Quick sort is one of the most efficient sorting algorithms with an average-case time complexity of O(n log n).

Heapsort is an efficient sorting algorithm with a time complexity of O(n log n) in all cases and is suitable for large data sets.

Questions for Deeper Understanding

1. Why does quick sort have a better average case time complexity compared to bubble sort, selection sort, and insertion sort?

Quick sort has a better average-case time complexity of O(n log n) compared to bubble sort, selection sort, and insertion sort, which have an average-case time complexity of O(n^2). The key reason for this difference is the divide-and-conquer strategy used by quick sort. Quick sort divides the input array into two subarrays around a pivot element and recursively sorts the subarrays. This approach reduces the number of comparisons and swaps required to sort the elements, resulting in a more efficient algorithm.

2. Under what conditions would insertion sort outperform quick sort?

Insertion sort is more efficient than quick sort for small data sets and nearly sorted data. When the input array is already partially sorted or contains a small number of elements, insertion sort can outperform quick sort due to its simplicity and lower overhead. Quick sort, on the other hand, is more efficient for large data sets and random data due to its average-case time complexity of O(n log n).

3. Why is heapsort often preferred in systems where memory allocation is a concern?

Heapsort is often preferred in systems where memory allocation is a concern because it is an in-place sorting algorithm that does not require additional memory beyond the input array. Heapsort builds a binary heap from the input array and sorts the elements in place, making it memory-efficient compared to other sorting algorithms like quick sort that require additional memory for recursive calls and stack space. Heapsort's in-place sorting nature makes it suitable for systems with limited memory resources or strict memory constraints.