Indian Institute of Technology Ropar

SORTING NOTES

MD RIZWAN AHMAD

September 3, 2024

Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm. The algorithm works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

Algorithm Description

The Bubble Sort algorithm can be described as follows:

Algorithm 1 Bubble Sort

Input: An array A of n elements
Output: A sorted array A
for $i = 1$ to $n - 1$ **do**
    for $j = 1$ to $n - i$ **do**
        if $A[j] > A[j+1]$ **then**
            Swap $A[j]$ and $A[j+1]$
        end if
    end for
end for
Return A

The algorithm proceeds by iteratively comparing adjacent elements in the array. If the current element is greater than the next element, they are swapped. After each iteration, the largest element in the unsorted portion of the array "bubbles up" to its correct position.

Complexity Analysis

Case	Time Complexity
Best Case	$O(n)$
Worst Case	$O(n^2)$
Average Case	$O(n^2)$
Space Complexity	$O(1)$

Table 1: Time and Space Complexity of Bubble Sort

Bubble Sort is an in-place sorting algorithm, meaning it does not require any additional memory beyond the input array.

Bubble Sort with Flag Optimization

If in a pass there is no swapping then this means that the array has been already sorted. We can use the flag variable to check if whether the swap is done in pass or not. If there has not been any swap then there is no need to continue further steps.

Bubble Sort Implementation

```
// Bubble Sort
void bubbleSort(int arr[], int n)
{
    for (int pass = 1; pass < n; pass++)
    {
        bool swapped = false;
        for (int i = 0; i < n - pass; i++)
        {
            if (arr[i] > arr[i + 1])
            {
                swap(arr[i], arr[i + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Insertion Sort

Insertion Sort is a simple and intuitive sorting algorithm. It builds the final sorted array one item at a time by repeatedly picking the next item from the unsorted portion and inserting it into its correct position in the sorted portion. This algorithm is particularly useful for small datasets or partially sorted arrays.

Algorithm Description

The Insertion Sort algorithm can be described as follows:

Algorithm 2 Insertion Sort

Input: An array A of n elements
Output: A sorted array A
for $i = 1$ to $n - 1$ **do**
  Set $key \leftarrow A[i]$
  Set $j \leftarrow i - 1$
  while $j \geq 0$ **and** $A[j] > key$ **do**
    $A[j + 1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  end while
  $A[j + 1] \leftarrow key$
end for
Return A

The algorithm iterates through each element of the array, using the current element as the ‘key‘. It then compares the ‘key‘ with the elements in the sorted portion of the array and shifts elements to make space for the ‘key‘.

Insertion Sort Summary

Aspect	Details
Best Case	$O(n)$ (Already sorted)
Worst Case	$O(n^2)$ (Reverse sorted)
Average Case	$O(n^2)$ (Random order)
Space Complexity	$O(1)$ (In-place)
Advantages	Simple Adaptive Stable
Disadvantages	Inefficient for large n Many comparisons/shifts

Table 2: Summary of Insertion Sort Algorithm

Insertion Sort Implementation

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

Listing 1: Insertion Sort

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum element from the unsorted portion of the array and moving it to the beginning. This process is repeated for each position in the array until it is fully sorted.

Algorithm Description

The Selection Sort algorithm can be described as follows:

Algorithm 3 Selection Sort

Input: An array A of n elements
Output: A sorted array A
for $i = 0$ to $n - 1$ **do**
    Set $min\_index \leftarrow i$
    for $j = i + 1$ to n **do**
        if $A[j] < A[min\_index]$ **then**
            Set $min\_index \leftarrow j$
        end if
    end for
    Swap $A[i]$ and $A[min\_index]$
end for
Return A

The algorithm iterates through the array, selecting the minimum element from the unsorted portion and placing it in the correct position. This process continues until the array is sorted.

Complexity Analysis

Case	Time Complexity
Best Case	$O(n^2)$
Worst Case	$O(n^2)$
Average Case	$O(n^2)$
Space Complexity	$O(1)$

Table 3: Time and Space Complexity of Selection Sort

Selection Sort is an in-place sorting algorithm and does not require any additional memory beyond the input array.

Selection Sort Implementation

```
//selection sort
void selectionSort(int arr[],int n)
{
    for(int i=0;i<n;i++)
    {
        int ind=i;
        int element=arr[ind];
        for(int j=i+1;j<n;j++)
        {
            if(arr[j]<element)
            {
                element=arr[j];
                ind=j;
            }
        }
        swap(arr[i],arr[ind]);
    }
}
```

Listing 2: Selection Sort

Merge Sort

Merge Sort is a comparison-based sorting algorithm that uses the divide and conquer technique. It divides the array into two halves, recursively sorts each half, and then merges the sorted halves to produce the final sorted array.

Algorithm Description

The Merge Sort algorithm can be described as follows:

Algorithm 4 Merge Sort

```
Input: An array A of n elements
Output: A sorted array A
if n > 1 then
    Divide A into two halves: A_1 and A_2
    Sort A_1 using MergeSort
    Sort A_2 using MergeSort
    Merge A_1 and A_2 into A
end if
Return A
```

Merge Sort recursively divides the array into halves until it reaches individual elements. Then, it merges these elements in a sorted manner. This merging process combines two sorted arrays into one sorted array.

Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$
Worst Case	$O(n \log n)$
Average Case	$O(n \log n)$
Space Complexity	$O(n)$

Table 4: Time and Space Complexity of Merge Sort

Merge Sort is a stable sorting algorithm with a consistent time complexity of $O(n \log n)$ across all cases. It requires additional space proportional to the size of the array due to the merging process.

Merge Sort Implementation

```
void merge(int arr[],int low,int mid,int high)
{
    int n=high-low+1;
    int temp[n];
    int first=low;
    int second=mid+1;
    int ind=0;

    while(first<=mid and second<=high)
    {
        if(arr[first]<=arr[second])
            temp[ind++]=arr[first++];
        else
            temp[ind++]=arr[second++];

    }

    while(first<=mid)
        temp[ind++]=arr[first++];
    while(second<=high)
        temp[ind++]=arr[second++];

    ind=0;
    first=low;
    while(ind<n)
        arr[first++]=temp[ind++];
}

void mergeSort(int arr[],int low,int high)
{
    if(low<high)
    {
        int mid=low+(high-low)/2;
        mergeSort(arr,low,mid);
        mergeSort(arr,mid+1,high);
        merge(arr,low,mid,high);
    }
}

void mergeSort(int arr[],int n)
{
    mergeSort(arr,0,n-1);
}
```

Listing 3: Merge Sort

Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It first builds a heap from the input data and then repeatedly extracts the maximum element from the heap and rebuilds the heap until it is empty. Heap Sort is efficient and has a good performance profile.

Algorithm Description

The Heap Sort algorithm can be described as follows:

Algorithm 5 Heap Sort

```
Input: An array A of n elements
Output: A sorted array A
Build a max heap from the array
for i = n − 1 to 1 do
    Swap A[0] and A[i]
    Reduce the heap size by 1
    Heapify the root of the heap
end for
Return A
```

Heap Sort involves two main operations: building a max heap and performing heap sort. The max heap is used to repeatedly extract the maximum element and place it in its correct position in the array.

Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$
Worst Case	$O(n \log n)$
Average Case	$O(n \log n)$
Space Complexity	$O(1)$

Table 5: Time and Space Complexity of Heap Sort

Heap Sort has a time complexity of $O(n \log n)$ in the best, worst, and average cases. It is an in-place sorting algorithm, meaning it requires only a constant amount of additional space.

Heap Sort Implementation

```
void heapify(int arr[],int right)
{

    int ind=right;
    while(ind)
    {
        int i=ind;
        while(i)
        {
            int parent=(i-1)/2;
            if(arr[parent]<arr[ind])
                swap(arr[parent],arr[i]);
            i=(i-1)/2;
        }
        ind--;
    }
}

void heapSort(int arr[],int n)
{
    int last=n-1;
    heapify(arr,n-1);
    while(last)
    {
        swap(arr[0],arr[last]);
        last--;
        heapify(arr,last);
    }
}
```

Listing 4: Heap Sort

Quick Sort

Quick Sort is a comparison-based sorting algorithm that follows the divide and conquer technique. It selects a 'pivot' element from the array and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Algorithm Description

The Quick Sort algorithm can be described as follows:

Algorithm 6 Quick Sort

```
Input: An array A of n elements
Output: A sorted array A
if n > 1 then
    Choose a pivot element
    Partition the array into two sub-arrays
    Sort the first sub-array
    Sort the second sub-array
end if
Return A
```

Quick Sort divides the array into smaller partitions around a pivot element. The pivot is used to partition the array into elements less than and greater than the pivot, which are then recursively sorted.

Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$
Worst Case	$O(n^2)$
Average Case	$O(n \log n)$
Space Complexity	$O(\log n)$

Table 6: Time and Space Complexity of Quick Sort

Quick Sort has a best and average-case time complexity of $O(n \log n)$, but in the worst-case scenario, it can degrade to $O(n^2)$, especially if the pivot selection is poor. The space complexity is $O(\log n)$ due to the recursive stack space.

Quick Sort Implementation

```
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Listing 5: Quick Sort

Implementing the Quick Sort Alogrithm using the Median of Medians Algorithm

The Median of Medians algorithm can be employed in Quick Sort to improve pivot selection. This technique provides several advantages:

- **Guaranteed Linear Time Selection:** The Median of Medians algorithm guarantees a worst-case linear time complexity, $O(n)$, for selecting the pivot.
- **Improves Worst-Case Time Complexity:** By ensuring better pivot selection, it helps Quick Sort achieve $O(n \log n)$ time complexity in the worst case, thus avoiding the $O(n^2)$ worst-case performance.
- **More Balanced Partitions:** It facilitates dividing the array into more balanced partitions, leading to more efficient sorting.

```
#include<bit/stdc++.h>
using namespace std;


int medianOfMedians(int arr[], int low, int high) {
    int n = high - low + 1;
    if (n <= 5) {
        int temp[n];
        for (int i = 0; i < n; i++)
            temp[i] = arr[low + i];
        sort(temp, temp + n);
        return temp[n / 2];
    }

    int numMedians = (n + 4) / 5;
    int median[numMedians];
    for (int i = 0; i < numMedians; i++) {
        int subLow = low + i * 5;
        int subHigh = min(subLow + 4, high);
        int subSize = subHigh - subLow + 1;
        int temp[5];
        for (int j = 0; j < subSize; j++)
            temp[j] = arr[subLow + j];
        sort(temp, temp + subSize);
        median[i] = temp[subSize / 2];
    }

    return medianOfMedians(median, 0, numMedians - 1);
}
```

```cpp

// Hoare partition scheme
int hoarePartition(int arr[], int low, int high, int
    pivotIndex) {
    int pivot = arr[pivotIndex];
    swap(arr[pivotIndex], arr[low]);
    int i = low - 1;
    int j = high + 1;
    while (true) {
        do { i++; } while (arr[i] < pivot);
        do { j--; } while (arr[j] > pivot);
        if (i >= j) return j;
        swap(arr[i], arr[j]);
    }
}

// QuickSort function using median of medians for pivot
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = medianOfMedians(arr, low, high);
        int pivotNewIndex = hoarePartition(arr, low, high,
            pivotIndex);
        quickSort(arr, low, pivotNewIndex);
        quickSort(arr, pivotNewIndex + 1, high);
    }
}

void quickSort(int arr[], int n) {
    quickSort(arr, 0, n - 1);
}

// Testing the quickSort function
int main() {
    int arr[] = {3, 6, 8, 10, 1, 2, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, n);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Listing 6: Quick Sort

Cycle Sort

Cycle Sort is an in-place, unstable sorting algorithm that minimizes the number of memory writes. It is particularly useful when writing to memory is a costly operation. It guarantee to sort the array with the minimum number of swaps.

Algorithm Description

The Cycle Sort algorithm can be described as follows:

Algorithm 7 Cycle Sort

Input: An array A of n elements
Output: A sorted array A
for $cycle\_start = 0$ to $n - 2$ **do**
    Set $item \leftarrow A[cycle\_start]$
    Find the correct position for *item* in the array
    if *item* is already in the correct position **then**
        Continue to next cycle
    end if
    while *item* is not in the correct position **do**
        Swap *item* with the element in its correct position
        Find the new correct position for *item*
    end while
end for
Return A

Cycle Sort works by finding the correct position for each element and moving it to that position in the array, ensuring minimal memory writes.

Cycle Sort Summary

Aspect	Details
Best Case	$O(n^2)$ (Few cycles)
Worst Case	$O(n^2)$ (Many cycles)
Average Case	$O(n^2)$
Space Complexity	$O(1)$ (In-place)
Advantages	Minimum memory writes In-place
Disadvantages	Quadratic time complexity Unstable

Table 7: Summary of Cycle Sort Algorithm

Code Implementation

```
void cycleSort(int arr[],int n)
{
    for(int cs=0;cs<n-1;cs++)
    {
        int currentItem=arr[cs];
        int pos=cs;

        //Find the number of elements in the array that is
            smaller than the current element
        for(int i=cs+1;i<n;i++)
            pos=arr[i]<currentItem?pos+1:pos;
        swap(currentItem,arr[pos]);

        //Repeating the process untill the correct element
            comes into my position
        while(pos!=cs)
        {
            pos=cs;
            for(int i=cs+1;i<n;i++)
                pos=arr[i]<currentItem?pos+1:pos;
            swap(currentItem,arr[pos]);
        }
    }
}
```

Listing 7: Cycle Sort

Counting Sort

Counting Sort is a non-comparative integer sorting algorithm that sorts an array by counting the occurrences of each unique element. It is efficient when the range of input values (k) is not significantly greater than the number of elements (n).

Algorithm Description

The Counting Sort algorithm can be described as follows:

Algorithm 8 Counting Sort

Input: An array A of n elements, with each element in the range 0 to k
Output: A sorted array A
Create a count array C of size $k+1$ and initialize all elements to 0
for each element x in A **do**
    Increment $C[x]$
end for
for $i = 1$ to k **do**
    $C[i] \leftarrow C[i] + C[i-1]$ {Cumulative sum}
end for
for each element x in A, in reverse order **do**
    Place x in the sorted array at the position $C[x] - 1$
    Decrement $C[x]$
end for
Return the sorted array

Counting Sort works by counting the occurrences of each unique element and using these counts to determine the correct positions of elements in the sorted array.

Counting Sort Summary

Aspect	Details
Best Case	$O(n+k)$
Worst Case	$O(n+k)$
Average Case	$O(n+k)$
Space Complexity	$O(n+k)$
Advantages	Linear time complexity for small k Stable sorting algorithm
Disadvantages	Not comparison-based Limited to integers within a known range

Table 8: Summary of Counting Sort Algorithm

Counting Sort code Implementation

```
void countSort(int arr[], int n, int k)
{
    int count[k];
    for(int i=0;i<k;i++)
        count[i]=0;
    for(int i=0;i<n;i++)
        count[arr[i]]++;

    for(int i=1;i<k;i++)
        count[i]=count[i-1]+count[i];

    int output[n];
    //We are traversing from the end of the array to make the sorting algorithm stable
    for(int i=n-1;i>=0;i--){
        output[count[arr[i]]-1]=arr[i];
        count[arr[i]]--;
    }
    for(int i=0;i<n;i++)
        arr[i]=output[i];

}
```

Listing 8: Counting Sort

Radix Sort

Radix Sort is a non-comparative sorting algorithm that processes each digit of the numbers, starting from the least significant digit (LSD) to the most significant digit (MSD) or vice versa. It uses a stable sorting algorithm like Counting Sort as a subroutine to sort the elements based on individual digits.

Algorithm Description

The Radix Sort algorithm can be described as follows:

Algorithm 9 Radix Sort

```
Input: An array A of n elements, with each element in the range 0 to k
Output: A sorted array A
Determine the maximum number of digits, d, in the elements of A
for each digit i from least significant to most significant do
    Use a stable sorting algorithm (e.g., Counting Sort) to sort the array
    based on the ith digit
end for
Return the sorted array
```

Radix Sort processes the digits of the numbers one by one and sorts the array according to these digits using a stable sorting algorithm.

Radix Sort Summary

Aspect	**Details**
Best Case	$O(d \cdot (n + k))$
Worst Case	$O(d \cdot (n + k))$
Average Case	$O(d \cdot (n + k))$
Space Complexity	$O(n + k)$
Advantages	Linear time complexity for fixed d No comparison-based sorting Stable
Disadvantages	Limited to integers or strings Requires additional memory

Table 9: Summary of Radix Sort Algorithm

Radix Sort Code Implementation

```
void countSort(int arr[],int n,int exp)
{
    int count[10]={0};
    int output[n];

    for(int i=0;i<n;i++) count[(arr[i]/exp)%10]++;
    for(int i=1;i<10;i++) count[i]=count[i]+count[i-1];

    for(int i=n-1;i>=0;i--)
    {
        output[count[(arr[i]/exp)%10]-1]=arr[i];
        count[(arr[i]/exp)%10]--;
    }

    for(int i=0;i<n;i++)
        arr[i]=output[i];

}

void radixSort(int arr[],int n)
{
    //Find the max element for the number of passes
    int mx=arr[0];
    for(int i=1;i<n;i++)
        mx=arr[i]>mx?arr[i]:mx;

    //Number of passes
    for(int exp=1;mx/exp>0;exp*=10)
        countSort(arr,n,exp);
}
```

Listing 9: Radix Sort

Bucket Sort

Bucket Sort is a distribution-based sorting algorithm that divides an array into a number of buckets. Each bucket is then sorted individually, either using another sorting algorithm or by recursively applying the bucket sort. The sorted buckets are then combined to form the final sorted array.

Algorithm Description

The Bucket Sort algorithm can be described as follows:

Algorithm 10 Bucket Sort

```
Input: An array A of n elements, uniformly distributed over some range
Output: A sorted array A
Create k empty buckets
for each element A[i] in A do
    Insert A[i] into the appropriate bucket
end for
for each bucket do
    Sort the bucket using a suitable sorting algorithm (e.g., Insertion Sort)
end for
Concatenate the sorted buckets to obtain the sorted array
Return the sorted array
```

The algorithm works by distributing the elements into different buckets based on a uniform distribution. Each bucket is then sorted, and the sorted buckets are concatenated to form the final sorted array.

Bucket Sort Summary

Aspect	**Details**
Best Case	$O(n + k)$
Worst Case	$O(n^2)$ (All elements in one bucket)
Average Case	$O(n + k)$
Space Complexity	$O(n + k)$
Advantages	Efficient for uniformly distributed data Simple to implement Suitable for parallel processing
Disadvantages	Performance depends on distribution of data Not suitable for large ranges of data Requires additional memory for buckets

Table 10: Summary of Bucket Sort Algorithm

Bucket Sort Code Implementation

```
void bucketSort(int arr[],int n,int k)
{
    //Find the maximum element in the array
    int maxVal=arr[0];
    for(int i=1;i<n;i++)
        maxVal=arr[i]>maxVal?arr[i]:maxVal;
    maxVal++;

    //Creating the k buckets
    vector<int> bkt[k];

    //Filling the data into the buckets
    for(int i=0;i<n;i++)
    {
        int bi=(k*arr[i])/maxVal;
        bkt[bi].push_back(arr[i]);
    }

    //Sort the buckets
    for(int i=0;i<k;i++)
        sort(bkt[i].begin(),bkt[i].end());

    int index=0;
    //Joining the buckets
    for(int i=0;i<k;i++)
    {
        for(int j=0;j<bkt[i].size();j++)
            arr[index++]=bkt[i][j];
    }
}
```

Listing 10: Bucket Sort

Summary of Sorting Algorithms

Algorithm	Best Case	Worst Case	Average Case	Stable	In-Place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	No
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Yes
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	No	Yes
Cycle Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	Yes	No
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	Yes	No
Bucket Sort	$O(n + k)$	$O(n^2)$	$O(n + k)$	Yes	No

Table 11: Summary of Sorting Algorithms

Algorithm	Advantages	Disadvantages
Bubble Sort	Simple to understand and implement. Works well for small or nearly sorted datasets.	Very inefficient for large datasets due to high time complexity.
Insertion Sort	Efficient for small datasets or nearly sorted data. Adaptive and stable.	Inefficient for large datasets due to quadratic time complexity.
Selection Sort	Simple to implement and has a predictable runtime. Minimizes the number of swaps.	Poor performance on large datasets, and it is not stable.
Merge Sort	Consistent performance with $O(n \log n)$ time complexity. Stable and works well on large datasets.	Requires additional space proportional to the size of the array, making it non-in-place.
Heap Sort	Efficient with $O(n \log n)$ time complexity. In-place sorting with no additional memory requirement.	Not stable and can have a slower performance in practice compared to Quick Sort.
Quick Sort	Very efficient for large datasets with average $O(n \log n)$ time complexity. In-place sorting with good cache performance.	Worst-case time complexity of $O(n^2)$, though this can be mitigated with techniques like the median of medians. Not stable.
Cycle Sort	Minimizes the number of writes, making it useful when memory write operations are costly. In-place sorting.	Poor time complexity for large datasets and not stable.
Counting Sort	Very efficient for small range of integer keys. Stable sorting algorithm with linear time complexity.	Requires additional space proportional to the range of input values. Not suitable for large ranges of input values.
Radix Sort	Efficient for sorting large numbers with small keys. Stable and can be faster than comparison-based sorts.	Requires additional space and is less efficient for large key sizes or very large datasets.
Bucket Sort	Effective for sorting uniformly distributed data. Linear time complexity when the input is uniformly distributed.	Performance degrades if the data is not uniformly distributed. Requires additional space for buckets.

Table 12: Advantages and Disadvantages of Sorting Algorithms