Chapter 1

Introduction

1.1 Motivation behind Sorting

Sorting is a fundamental operation in computing. Hardly a month goes by without any research reported on the problem. While it is important to thus understand the basic principles behind classic sorting algorithms, the questions in this experiment let you think of current problems. Another important aspect which can be observed from this experiment is to think of large data sets. How classic computation such as sorting should be designed for those large data sets.

1.2 Introduction of Sorting

Sorting [1] is defined as arranging an unordered collection of data into particular order. Order can be monotonically increasing or decreasing. Suppose \( M = (p_1, p_2, p_3, \ldots, p_n) \) be a sequence of ‘\( n \)’ elements in unsorted manner, sorting transforms ‘\( M \)’ into a monotonically increasing sequence \( S' = (p'_1, p'_2, p'_3, \ldots, p'_n) \). Sorting has two categories, internal sorting and external sorting [2]. Sorting algorithm can be sort the data as comparison-based and non-comparison-based. In comparison-based sorting algorithm [3], sort the unordered data by comparing the pairs of
data repeatedly, and if the data are out of order then exchange them to each other. This exchanging operation of this sorting is called compare-exchange. Non-comparison-based [4] sorting algorithms sort the data by using certain well known properties of the data, such as the binary representation or data distribution. Sorting algorithms have four types of performance measures which are stability, adaptivity, time complexity, space complexity [5].

**Definition 1 (Stability)** A sorting algorithm is stable [6] if it preserves the order of duplicate keys, or stability means that equivalent elements retain their relative positions, after sorting.

**Definition 2 (Adaptivity)** A sorting algorithm is adaptive [7] if it sorts the sequences that are close to sorted faster than random sequences. A sorting algorithm [8] is denoted adaptive if the time complexity is a function depending on the size as well as the pre-sortedness of the input sequence.

**Definition 3 (Time complexity)** Time complexity [9] of an algorithm signifies the total time required by the program to run to completion. Algorithms have different cases of complexity [10] which are best case, average case, and worst case. The time complexity of an algorithm is represented using the asymptotic notations [11]. Asymptotic notations provide the lower bound and upper bound of an algorithm.

**Definition 4 (Space complexity)** Space complexity [12] of any algorithm is also important, and it is the number of memory cells which an algorithm needs. Space complexity calculated by both auxiliary space and space used by the input.

### 1.3 Sorting Algorithms

In this section, the working of some traditional and popular sorting algorithms have been explained with the help of algorithms and examples.
1.3.1 Insertion Sort

Insertion sort [11] algorithm works efficiently for a small number of elements. In insertion sort [13] algorithm we sort the one element at a time. We can use the

Algorithm 1 Insertion Sort Algorithm

<table>
<thead>
<tr>
<th>INPUT: Unssorted list of n items</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT: Sorted list of n items</td>
</tr>
</tbody>
</table>

for \( m = 0 \) to length[\( A \)] do
  \( \text{lock} = A[m] \)
  Arrange \( A[m] \) in sorted order
end for

\( i = m - 1 \)

while \((i > 0 \text{ and } A[i] > \text{lock})\) do
  \( A[i + 1] = A[i] \)
  \( i = i - 1 \)
  \( A[i + 1] = \text{lock} \)
end while

insertion sort algorithm, when we sort a deck of cards. In this algorithm we pick an element from the list and place it in the correct location in the list. Process will be repeated till there is no more unsorted items remained. It is an adaptive sorting algorithm, it takes \( O(n) \) time when data is nearly sorted. Insertion sort algorithm is stable, and also it is an online sorting. The insertion sort algorithm can be depicted as follows; for the value \( m = 2, 3, \ldots, n \). Where \( n = \text{Length} [A] \).

Time Complexity of Insertion Sort

- **Best Case:** In the above algorithm line 1 to 7 is the outer loop and line 5 to 7 is the inner loop. Insertion sort have best case when the data is already sorted or nearly sorted, and for the best case the inner loop never executed in the algorithm. So the comparison will be like that, we need 1-comparison to compare first element and 2-comparison to compare second element and \( n \)-comparison to compare \( n \)-elements.

\[
1 + 2 + 3 + \ldots + n \quad (1.3.1)
\]

\[
T(n) = \Omega(n) \quad (1.3.2)
\]
• **Average Case:** If we compare the average case with worst case, then we find that, the average case comes to be same as the worst case. Suppose if there are ‘n’ numbers and the numbers chosen randomly and apply an insertion sort. Then how much time algorithm will take to determine the sub array $A[1...m-1]$ to insert element $A[m]$. In average case, we divide the elements in two halves, in one half elements are in $[1... m-1]$ and these elements are less than $A[m]$, and in another half the elements are greater. In average case, we also check one half of the sub array $A[1...m-1]$ and so $t_m$ is become $m/2$. Then we find the resulting average case execution time to be a quadratic function of the input size ‘n’, which is same as the worst case execution time. i. e.

$$T(n) = \Theta(n^2)$$ (1.3.3)

• **Worst Case:** Insertion sort worst case occurs if the array is re-versed sorted that is in decreasing order. In the worst case, inner loop is executed exactly $m-1$ times for every iteration of the outer loop. Calculation of number of comparison of an array ‘n’ elements in worst case will be: to insert the first element comparison is not necessary, and to insert the second element one comparison is needed and so on, and to insert the last element $(n-1)$ comparisons is required at most

$$Total : 1 + 2 + 3 + ..... + (n - 1) = O(n^2)$$ (1.3.4)

**Space Complexity of Insertion Sort**

Auxiliary space complexity of insertion sort is $O(1)$, i.e. insertion sort having a constant space complexity.

### 1.3.2 Selection Sort

In the selection sort [11] algorithm, smallest item will be selected and swapped by the item which is the filled in the next position. Selection sort working is that: we search the smallest element through the entire array, once we find it,
swap item with the smallest element in the position of the first element of the array. After that we search for the second smallest element in the remaining array and exchange it with the second element and so on. Selection sort is not stable sorting, and it is also not adaptive sorting. Selection sort comes in the categories of comparison based sort. Selection sort algorithm can be depicted as follows, in the algorithm length \([B] = n\), where ‘n’ is the input data which is used for sorting in the algorithm.

**Algorithm 2 Selection Sort Algorithm**

**INPUT:** Unsorted list of \(n\) items

**OUTPUT:** Sorted list of \(n\) items

\[
\begin{align*}
n &\leftarrow \text{length}[B] \\
\text{for } m &\leftarrow 1 \text{ to } n - 1 \text{ do} \\
 & \quad \text{smallest} \leftarrow m \\
 & \quad \text{for } i \leftarrow m + 1 \text{ to } n \text{ do} \\
 & \quad \quad \text{if } B[i] < B[\text{smallest}] \text{ then} \\
 & \quad \quad \quad \text{smallest} \leftarrow i \\
 & \quad \quad \text{exchange } B[m] \leftrightarrow B[\text{smallest}] \\
 & \quad \text{end if} \\
 & \text{end for} \\
\end{align*}
\]

**Time Complexity of Selection Sort**

- **Best Case, Average Case, Worst Case** There is difficult to analyzing the time complexity in selection sort algorithm [10], because there are no loops depend on the item in the given array \(n-1\) comparison will be taken to selecting the lowest element. And to select the lowest element, we require scanning all \(‘n’\) elements and then lowest element swapped to the first position. After that we again scan the remaining \(n-1\) elements to find the next lowest element, and also further scan the elements till there are no more items to swap, so the time complexity of selection sort will be

\[
T(n) = (n - 1) + (n - 2) + \ldots + 2 + 1 = n\left(\frac{n - 1}{2}\right) = \Theta(n^2)
\]

Comparisons. In any cases of selection sort (worse case, best case or average case) the number of comparisons between elements is the same. So in all the
three cases selection sort have the time complexity:

\[ T(n) = \Theta(n^2) \]  

(1.3.6)

**Space Complexity of Selection Sort**

Auxiliary space complexity of selection sort is \( O(1) \) i.e. selection sort having a constant space complexity.

### 1.3.3 Bubble Sort

The basic idea underlying the bubble sort [4] is to pass through the list of elements sequentially several times. In each pass, we compare each element in the array and interchanging the two elements if they are not in proper order. Bubble sort is a stable sorting algorithm. It is an adaptive sorting algorithm, it takes \( O(n) \) time when data type is nearly sorted, it is also a comparison based sorting algorithm.

Bubble sort algorithm can be depicted as follows: In the algorithm length \([\text{Array}] = n\), where ‘n’ is the input data which is used for sorting in the algorithm.

**Algorithm 3** Bubble Sort Algorithm

**INPUT:** Unsorted list of \( n \) items  
**OUTPUT:** Sorted list of \( n \) items  

```
for i ← 1 to length[Array] - 1 do
  for m ← to length[Array] - i do
    if array[m] > array[m + 1] then
      exchange array[m] ↔ array[m + 1]
    end if
  end for
end for
```

**Time Complexity of Bubble Sort**

- **Best Case:** Bubble sort have best case when the data in the array is already sorted or nearly sorted. In the best case, where the array is already sorted
algorithm will terminate after the first iteration and no swap will be made and the one iteration will take \( n-1 \) comparison, so the

\[
T(n) = \Omega(n)
\]

(1.3.7)

- **Average Case:** Bubble sort average case is \( \Theta(n^2) \), which is same as worst case of bubble sort.
- **Worst Case:** In the worst case of bubble sort elements compares till \( n-1 \) iterations with the following comparisons.

\[
T(n) = (n - 1) + (n - 2) + ... + 2 + 1 = n\left(\frac{n-1}{2}\right) = O(n^2)
\]

(1.3.8)

**Space Complexity of Bubble Sort**

Auxiliary space complexity of bubble sort is \( O(1) \), i.e. bubble sort having a constant space complexity.

### 1.3.4 Heap Sort

Heap sort algorithm is sorted the items based on the data structure heaps. Heaps are two types: 1. A max heap 2. A min heap. Heap sort should satisfy the heap property. Heap property: [15]

- All nodes are either greater than or equal to or less than or equal to of each of its children, according to a comparison the node will define for the heap.
- Heaps with a mathematical \((\geq)\) comparison predicate are called max-heaps. Figure 1.1 shows the max-heap.
- Heaps with a mathematical \((\leq)\) comparison predicate are called min-heaps. Figure 1.2 shows the min-heap.
Heap sort works as: [9]

- Construct a heap,
- Add each item to its (maintaining the heap property),
- After adding the all items, we remove them one by one (restoring the heap property as each one is removed).

Heap sort is not a stable sorting algorithm. Heap sort is not an adaptive sorting algorithm. A heap sort algorithm can be depicted as follows:
**Algorithm 4** Heap Sort Algorithm

**INPUT:** Unsorted list of \( n \) items

**OUTPUT:** Sorted list of \( n \) items

\[
\text{BuildHeap}(B) \\
\text{for } i \leftarrow \text{length}(B) \text{ down to } 2 \text{ do} \\
\quad \text{exchange } B[1] \leftrightarrow B[i] \\
\quad \text{heap-size}[B] \leftarrow \text{heap-size}[B] - 1 \\
\quad \text{Heapify}(B, 1) \\
\end{for}
\]

**Algorithm 5** Heapify \((B, i)\)

**INPUT:** Unsorted list of \( n \) items

**OUTPUT:** Sorted list of \( n \) items

\[
l \leftarrow \text{left}[i] \\
r \leftarrow \text{right}[i] \\
\text{if } l \leq \text{heap-size}[B] \text{ and } B[l] > B[i] \text{ then} \\
\quad \text{largest} \leftarrow l \\
\text{else} \\
\quad \text{largest} \leftarrow i \\
\text{end if} \\
\text{if } r \leq \text{heapsize}[B] \text{ and } B[i] > B[\text{largest}] \text{ then} \\
\quad \text{largest} \leftarrow r \\
\text{end if} \\
\text{if } \text{largest} \neq i \text{ then} \\
\quad \text{exchange } B[i] \leftrightarrow B[\text{largest}] \\
\quad \text{Heapify}(B, \text{largest}) \\
\text{end if}
\]

Time Complexity of Heap Sort

- **Best Case, Average Case, Worst Case**
  1. \( \Theta(n) \) trips through the loop.
  2. \( T(n)_{\text{Maxheapify}} = \Theta(\log n) \).
  3. \( T(n)_{\text{Heapsort}} = \Theta(n) \times \Theta(\log n) = \Theta(n \log n) \)

Space Complexity of Heap Sort

The auxiliary space complexity of heap sort is \( O(1) \). i.e. it is having constant space complexity.
1.3.5 Shell Sort

Shell sort [3] is the diminishing increment sort. Shell sort is a generalization of bubble sort or insertion sort. The shell sort makes many passes through the data in to array, and each time sort the numbers that are equally distanced. The shell sort algorithm is similar to bubble sort in the sense that, it also moves elements by the exchanges. It begins by comparing elements that are at a distance ‘d’.

In each pass of shell sort the value of ‘d’ is reflected to half i.e. in each pass, we compare the each element that is located ‘d’ position away from it, after comparing the element exchanges are made if required. In the next iteration the value of ‘d’ will get change, the algorithm terminates when $d = 1$.

The example of shell sort is explained in Figure 1.3 to Figure 1.5.

12, 9, -10, 22, 2, 35, 40

d = n/2 = 7/2 = 3

Pass 1 is completed in Figure 1.3. Now the value of $d = 2$
Pass 2 is completed in Figure 1.4. Now the value of $d = 1$.

Pass 3 is completed, and the algorithm got terminated as $d = 1$. Finally, we got the sorted list -10, 2, 9, 12, 22, 35, and 40 Shell sort is no preserved the stability. It is an adaptive algorithm: it takes $O(n\log n)$ time when data is nearly sorted. The shell sort algorithm can be depicted as follows:

**Algorithm 6 Shell Sort Algorithm**

**INPUT:** Unsorted list of $n$ items

**OUTPUT:** Sorted list of $n$ items

for each(gap in gaps) do
    for $m = gap$; $m < n$; $m + = 1$ do
        temp = $a[m]$
    end for
end for

for $q = m$; $q > = gap$ and $a[q - gap] > temp$; $q -$ = gap do
    $a[q] =$ $a[q - gap]$
    $a[q] =$ temp
end for
Time Complexity of Shell Sort

- **Best Case:** Shell sort have best case, when the data is already sorted, because the number of passes will be less in this case. Passes = \( n \), for 1 sorts with item 1 apart (last step) \( 3 \times \frac{n}{3} \), for 3 sorts with items 3 apart (next-to-last step) \( 7 \times \frac{n}{7} \), for 7 sorts with items 7 apart. \( 15 \times \frac{n}{15} \), for 15 sorts with items 15 apart +.....Each term is \( 'n' \). The question arise how many terms are there. The value of \( 'k' \) such that \( 2^k - 1 < n \). So \( k < \log(n + 1) \), meaning that the sorting time in the best case is less than

\[
n \times \log(n + 1) = \Omega(n \log n)
\]  

(1.3.9)

- **Average Case:** The average case time complexity of shell sort is depends on the gap sequences between the elements.

- **Worst Case:** The worst case is similar as average, but overall computation is differ. Passes \( \leq n^2 \), for 1 sort with item 1 apart (last step). \( 3 \times (\frac{n}{3})^2 \), for 3 sorts with items 3 apart (next-to-last step). \( 7 \times (\frac{n}{7})^2 \), for 7 sorts with items 7 apart. \( 15 \times (\frac{n}{15})^2 \), for 15 sorts with items 15 apart +.... So the number of passes is bounded by

\[
n^2 \times \left( 1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + .... \right) < n^2 \times \left( 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + .... \right) = n^2 \times 2 = O(n^2)
\]  

(1.3.10)

Space Complexity of Shell Sort

The auxiliary space complexity of shell sort is \( O(1) \). i.e. it is having constant space complexity.

1.3.6 Counting Sort

Counting sort [16] is a linear time sorting, counting sort is an algorithm that sorts the items according to keys. Counting sort is used to sort those items, when they belong to a fixed and finite set. Example of items belongs to a fixed
interval are \(m_1\) to \(m_2\), and example of items belongs to finite set will be no limit. The algorithm makes two passes one for ‘\(A\)’ and one for ‘\(B\)’. If the size of items belongs to fixed interval range ‘\(m\)’ will be less than size of input ‘\(n\)’, then time complexity will be \(O(n)\). Counting Sort preserves the stability. The efficiency of counting sort is better for the number of objects, when its range is less than input data. The algorithm of counting sort is as follows:

**Algorithm 7 Counting Sort Algorithm**

**INPUT:** Unsorted list of \(n\) items

**OUTPUT:** Sorted list of \(n\) items

```plaintext
for i = 1 to \(m\) do
    \(X[i] = 0\)
end for

for j = 1 to length[\(A\)] do
    \(X[A[j]] = X[A[j]] + 1\)
    \(X[i]\) here elements equal to \(i\) contains by \(X[i]\)
end for

for i = 2 to \(m\) do
    \(X[i] = X[i] + X[i-1]\)
    \(X[i]\) here number of elements less than or equal to \(i\) contains by \(X[i]\)
    \(B[C[A[j]]] = A[j]\)
    \(X[A[j]] = X[A[j]] - 1\)
end for
```

**Time Complexity of Counting Sort**

- **Best Case:** If the size of items belongs to fixed interval range ‘\(m\)’ will be less than size of input ‘\(n\)’, then time complexity will be \(T(n) = \Omega(n)\).
- **Average Case:**
  \[\Theta(n) + \Theta(k) = \Theta(n + k)\]  \hspace{1cm} (1.3.11)
- **Worst Case:**
  \[O(n) + O(k) = O(n + k)\]  \hspace{1cm} (1.3.12)

**Space Complexity of Counting Sort**

The auxiliary space complexity of counting sort is \(O(n+k)\), because the counting sort needs \(O(n)\) auxiliary space for the array and ‘\(k\)’ is the number of key
1.3.7 Quick Sort

Quick sort [11] uses the divide and conquer technique [38]. Quick sort first divides the given array of data into two smaller sub-arrays: the low elements and the high elements, sub-arrays are recursively sorted. The steps of quick sort are [39]:

1. Quick sort first chooses the pivot from the list.
2. Reorder the array, and in the array the values which are less than the pivot come before the pivot and the values which are greater than the pivot come after the pivot and the equal values can go either way.
3. The above two steps are recursively applied to the sub-array of data with smaller data and separately to the sub-array of data with greater values.

Quick sort is not a stable sorting algorithm. Quick sort is not an adaptive sorting algorithm; it is a comparison based sorting. The quick sort algorithm can be depicted as follows. Partitioning the array: The main thing in the algorithm is the partition procedure, which rearranges the sub array $A[p...r]$.

**Algorithm 8 Quick Sort Algorithm**

**INPUT:** Unsorted list of $n$ items  
**OUTPUT:** Sorted list of $n$ items  

```
if $p < r$ then  
  $q = \text{Partition}(A, p, r)$  
  Quick sort($A, p, q - 1$)  
  Quick sort($A, q + 1, r$)  
end if  
Partition($A, p, r$)  
$x = A[r]$  
$i = p - 1$  
for $j = p$ to $r - 1$ do  
  if $A[j] \leq x$ then  
    $i = i + 1$  
    Exchange $A[i + 1]$ with $A[r]$  
    return $i + 1$  
  end if  
end for
```
Time Complexity of Quick Sort[17]

- **Best Case:** Best case of quick sort occurs when the sub arrays are completely balanced every time. Each sub array has $\leq n/2$ elements. Get the recurrence:

\[
T(n) = 2T\left(\frac{n}{2}\right) + \Omega(n) \tag{1.3.13}
\]

Where $\Omega(n)$ time is the partitioning cost. By case 2 of master theorem [11] the equation (1.2.1) has the solution:

\[
T(n) = \Omega(n\log n) \tag{1.3.14}
\]

- **Average Case:** In average case, we produce the good and bad splits by partition the sub array.

![Figure 1.6: Split of $n$ on two consecutive levels](image)

![Figure 1.7: Split of $n$ on two consecutive levels](image)

Figure 1.6 shows the division of recursion tree at two consecutive levels. The partitioning cost shown by root of tree ‘$n$’, and the size of sub array produced $m -1$ and ‘$\theta$’, which is the worst case of quick sort. And next level have , the
sub array of size $m-1$ undergoes best case partitioning into a sub array of size $(m-1)/2-1$ and $(m-1)/2$. So if we combine the bad split and the good split then it produces three sub array and the size of sub array is 0, $(m-1)/2-1$, and $(m-1)/2$ at a total partitioning cost of:

$$
\Theta(n) + \Theta(n - 1) = \Theta(n)
$$

(1.3.15)

Figure 1.7 shows a single level partitioning that produces two sub arrays of size $(m-1)/2$ at a cost of $\Theta(n)$ Both figures result in $\Theta(n\log n)$ time, though the constant for the figure on the left is higher than that of the figure on the right. So

$$
T(n) = \Theta(n\log n)
$$

(1.3.16)

- **Worst Case:** Unbalanced sub-array produces the worst case of quick sort. It has ‘0’ element in one sub-array and $n-1$ element in the other sub-array. Then the recurrence:

$$
T(n) = T(n - 1) + T(0) + O(n)
$$

(1.3.17)

$$
T(n) = T(n - 1) + O(n)
$$

(1.3.18)

$$
T(n) = O(n^2)
$$

(1.3.19)

Where $O(n)$ is the partitioning cost, which is same as insertion sort. Quick sort have worst case, when the input is in sorted manner, but insertion sort runs in $O(n)$ time in this case. Worst case of quick sort also occurs when the array is in reverse sorted order and also when all elements are same.

**Space Complexity of Quick Sort**

Quick sort uses the constant additional space with unstable partitioning before making any recursive call. Constant amount of information is stored by quick sort for every nested recursive call. As the best case of quick sort takes at most $O(\log n)$ many nested recursive calls, it uses $O(\log n)$ space. However, if we limit the recursive calls, without Sedgwick’s trick then the worst case of quick sort could make $O(n)$ nested recursive calls and need $O(n)$ auxiliary space.
1.3.8 Radix Sort

There are two types of radix sorting [19]:

1. MSD radix sort starts sorting from the beginning of strings (Most Significant Digit).
2. LSD radix sort starts sorting from the end of strings (Least Significant Digit).

Example of radix sort explained in Figure 1.8 to Figure 1.12.

Ex. 0712, 21171, 00120, 43589, 73641, 31975, 60433

i. Padding used to make all numbers 5-digits.
ii. Start from the last most digit.
iii. We can use the buckets.

![Figure 1.8: Example of Radix Sort](image)

Now reassemble the list according to the buckets of Figure 1.8.

00120, 21171, 73641, 60433, 07125, 31975, 43589
Now reassemble the list again according to the buckets of Figure 1.9. We will get 00120, 07125, 60433, 73641, 51455, 21171, 31975, 43589.

Now reassemble the list again according to the buckets of Figure 1.10, we will get 00120, 07125, 21171, 60433, 51455, 43589, 73641, 31975.
Now reassemble the list again according to the buckets of Figure 1.11, we will get \(00120, 60433, 21171, 51455, 31975, 43589, 73641, 07125\).

Radix sort preserves the stability. Radix sort [20] is not an adaptive sort. The radix sort algorithm can be depicted as follows:
Algorithm 9 Radix Sort Algorithm

**INPUT:** Unsorted list of \( n \) items

**OUTPUT:** Sorted list of \( n \) items

RADIX SORT(X, \( n \))

\[\text{for } i = 1 \text{ to } n \text{ do}\]

\[\text{Any stable sort algorithm is used to sort array } X \text{ one digit } i.\]

\[\text{end for}\]

---

**Time Complexity of Radix Sort**

- **Best Case:** Radix sort algorithm requires ‘\( k \)’ to pass over the list of ‘\( n \)’ numbers. So the radix sort complexity = \( \Omega(nk) \), where ‘\( k \)’ is the input number of elements which is used for sorting in the algorithm, and ‘\( k \)’ is the number of digits in the longer input number. We don’t know that how ‘\( k \)’ big can be, sometimes ‘\( k \)’ can be large and it can be small. When the ‘\( k \)’ is small then \( \Omega(nk) = \Omega(n) \).

- **Average Case:** The average case time complexity of radix sort is \( \Theta(kn) \), which is same as the worst case of radix sort.

- **Worst Case:** We don’t know whether ‘\( k \)’ is large or ‘\( n \)’ is large, so we keep them both. So radix sort worst case complexity = \( O(kn) \).

---

**Space Complexity of Radix Sort**

Auxiliary Space Complexity of radix sort is \( O(k + n) \), where ‘\( k \)’ is the number of buckets and ‘\( n \)’ is the input elements.
1.3.9 Bucket Sort

A Bucket sort [20] is not really a sorting algorithm because it’s not really sorts anything. It partitions an array of elements into a number of buckets, and then buckets are individually sorted, either recursively or we use some other algorithm. Bucket sort preserves the stability. Bucket sort is not an adaptive sorting algorithm. It is non-comparison based sorting algorithm. The bucket sort algorithm can be depicted as follows: In the algorithm length $|B| = n$, where ‘$n$’ is the input number of elements which is used for sorting in the algorithm.

**Algorithm 10** Bucket Sort Algorithm

**INPUT:** Unsorted list of $n$ items

**OUTPUT:** Sorted list of $n$ items

$n \leftarrow $ length[$B$]

for $i = 1$ to $n$ do

Insert $B[i]$ into list $A[B[i]/b]$, where $b$ is the bucket size.

end for

for $i = 0$ to $n-1$ do

Sort list $A$ with Insertion sort

Concatenate the lists $A[0], A[1], \ldots A[n-1]$ together in order.

end for
Time Complexity of Bucket Sort

- **Best Case:** We have a list of ‘n’ elements. Going through the list and put the elements in the correct bucket = Ω(n). Merging the buckets = Ω(k), where ‘k’ is the number of buckets.

\[ \Omega(n) + \Omega(k) = \Omega(n + k) \quad (1.3.20) \]

- **Average Case:**

\[ \Theta(n) + \Theta(k) = \Theta(n + k) \quad (1.3.21) \]

- **Worst Case:** If every element belongs to the same buckets, then what will happen? In the worst case, this would imply that we would have \( O(n^2) \) performance, because if all the element belongs to the same bucket, then use insertion sort on ‘n’ elements which is \( O(n^2) \).

Space Complexity of Radix Sort

The auxiliary space complexity of bucket sort is \( O(nk) \), because the bucket sort needs \( O(n) \) auxiliary space for the array and ‘k’ is the number of buckets.

1.4  Introduction to GPU

GPU stands for Graphics Processing Unit. In the 1999-2000 computer scientist started using the GPU to extend the range of scientific domain [21]. The term GPU was familiarize in 1999 by NVIDIA. The world first GPU was a Geforce 256. To do the GPU programming, we require the use of graphics APIs such as OpenGL and WebGL [22]. In 2002 James Fung developed OpenVIDIA. It is used for parallel GPU computer vision. The projects of OpenVIDIA implement computer observation algorithms run on graphics hardware such as OpenGL, Cg and CUDA-C [23]. In November 2006 NVIDIA launched CUDA (Compute Unified Device Architecture) [24]. It is an API (Application Programming Interface)
that allows coding the algorithms for execution on GeForce GPUs using C as a high level programming language [25]. CUDA can use with other languages also see with the help of a diagram in Figure 1.14 [26].

<table>
<thead>
<tr>
<th>GPU Computing Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>C with CUDA Extensions</td>
</tr>
<tr>
<td>NVIDIA GPU with the CUDA Parallel Computing Architecture</td>
</tr>
</tbody>
</table>

![CUDA support the various languages](image)

The parallel execution of sorting algorithms using graphics processing unit (GPU) is allowed by general purpose computing, on graphics processing unit (GPGPU) [32]. Parallel sorting algorithms having highly code are handled by GPU as a co-processor. The framework of NVIDIA’s compute unified device architecture (CUDA) release that provides free programmability of GPUs [23]. The number of stream processors are used for floating point calculations with CPUs. Parallelism is limited in a stream processor like ALUs [33]. Stream processor acts as an ideal in parallelism of floating point operations. For example, NVIDIA GTX 260 contains 250 on-board stream processors. The GPU has a much greater computation throughput compared with a CPU. NVIDIA implemented their GPGPU architecture through extensions to the C programming language to allow for simple integration with existing applications. Unlike for code running on the host supplied by full ISO C++ through NVIDIA’s CUDA [34] compiler, functions executed on the device only supports CUDA C. So we are going to develop the parallel version of merge and quick sort using GPU in the framework of NVIDIA’s CUDA C [35].

The parallel computing with CUDA organizes concepts of Grid, Block and Thread which can be defined as:

- Grid: this is the group of blocks. There is no synchronization between the blocks.
• Block: This is the group of threads.
• Thread: This is the execution of the kernel.

Nowadays GPU is a big domain in parallel computing [27]. GPU works in many spheres of our daily lives. The architecture of GPU is shown in Figure 1.15.

![Figure 1.15: GPU Architecture](image)

1.4.1 Scalable Programming Model

The scalable programming model [28] allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions. A scalable programming model is shown in Figure 1.16.

24
1.5 Motivation to our approaches

Nowadays GPU is in big demand in parallel computing. Numerous researchers are working on the GPU. The programmability of the GPU is rising in the world. The rising GPU has enabled the threshold. The following point makes the user to work on the GPU rather than the CPU.

- GPUs are faster than the CPU.
- GPUs are commercially successful.
- GPUs have a disruptive innovation path.
- The GPU programming model emerging.
- GPU is massively parallel.
- It has hundreds of cores.
- It has thousands of threads.
- It is cheap.
- It is highly available.
1.6 Sorting Benchmark and Standard Dataset

In this thesis, sorting benchmark and standard dataset are used to test both the versions (sequential and parallel) of sorting algorithms. Six types of test cases are used by sorting benchmark which are Uniform, Sorted, Zero, Bucket, Gaussian, and Staggered [24][29][30]. The size of input data is varied from 100 to 10000000 and the thread in the multiple of 2 from 1 to 1024.

1. **Uniform test case:** Input values are picked randomly from 0 to $2^{32}$.

2. **Gaussian test case:** This test case consists the distribution of data created by taking the average of four randomly values picked from the uniform distribution.

3. **Zero test case:** A constant value is used as input by this test case.

4. **Bucket test case:** For $p \in n$, the input of size $n$ is split into $p$ blocks, such that the first $n/p^2$ elements in each of them are random numbers in $[0, 2^{31}/p-1]$, the second $n/p^2$ elements in $[2^{31}/p, 2^{31}/p - 1]$, and so forth.

5. **Staggered test case:** For $p \in n$, the input of size $n$ is split into $p$ blocks such that if the block index is $i \leq p/2$ all its $n/p$ elements are set to a random number in $[(2i-1)2^{31}/p, (2i)(2^{31}/p - 1)]$.

6. **Sorted test case:** Sorted uniformly distributed value has been taken as input.

Some Sorting algorithms are evaluated on four cases of standard dataset [31]. The dataset contains the 1010228 items. The four cases of dataset are:

1. Random with repeated data (Random data).
2. Reverse sorted with repeated Data (Reverse sorted data).
3. Sorted with repeated data (Sorted data).
4. Nearly sorted with repeated data (Nearly sorted data)

1.7 Hardware

In order to run proposed and designed algorithms Window 7 32-bit operating system Intel core i3 processor 530@ 2.93 GHz machine is used. System build
with GeForce GTX 460 graphic processor with \( (7 \text{ multiprocessors} \times (48) \text{ CUDA cores}) \text{ MP} = 336 \text{ CUDA cores} \). System body consists 1536 threads per multiprocessor and 1024 threads per block. CUDA runtime version of the system is 6.0. The total amount of global memory present in the system is 768 Mbytes and the total amount of constant memory is 65536 bytes. The total amount of shared memory per block is 49152 bytes. Total number of registers available per block is 32768 and warp size is 32. Maximum sizes of each dimension of a block are \( 1024 \times 1024 \times 64 \) and maximum size of each dimension of a grid is \( 65535 \times 65535 \times 65535 \).

1.8 Thesis road map

The main contribution of the thesis is explained via the following chapters: