Design and Analysis of Algorithms

Art of Computation

PROF. MAHESH GOYANI

Department of Computer Engineering
L D College of Engineering
Ahmedabad, INDIA

E-Mail: mgoyani@gmail.com

Web Site: www.maheshgoyani.in

Lecture Notes on

Sorting Techniques

Introduction To Sorting

What is Sorting ?

Definition:
Sorting is to organize a collection of data elements based on the order of a comparable property of each element.

Data Element: A unit of information

Comparable Property: A property in each element that can be used to compare element A with element B

Collection: Sorting works on a collection of data elements, not on a single data element

Classification

Internal Sorting:

All the data to sort is stored in memory at all times while sorting is in progress

Classification

Internal Sorting:

All the data to sort is stored in memory at all times while sorting is in progress

External Sorting:

Data is stored outside memory (like on disk) and only loaded into memory in small chunks.

External sorting is usually applied in cases when data can't fit into memory entirely.

Inversion

Definition:

Let a = $\{a_0, a_1 \ldots, a_{n-1}\}$ be a finite sequence. An inversion is a pair of index positions, where the elements of the sequence are out of order. Formally, an inversion is a pair (i, j), where $i < j$ and $a_i > a_j$.

The exact number of steps required by sorting algorithm is given by the number of inversions of the sequence

Inversion

Example: Let a $= \{ 5, 7, 4, 9, 7 \}$. Then, $(0, 2)$ is an inversion, since $a_0 > a_2$, namely $5 > 4$. Also, $(1, 2)$ is an inversion, since $7 > 4$, and $(3, 4)$ is an inversion, since $9 > 7$. There are no other inversions in this sequence.

Inversion

Example: Let a $= \{ 5, 7, 4, 9, 7 \}$. Then, (0, 2) is an inversion, since $a_0 > a_2$, namely $5 > 4$. Also, (1, 2) is an inversion, since $7 > 4$, and (3, 4) is an inversion, since $9 > 7$. There are no other inversions in this sequence.

i	0	1	2	3	4	5	6	7
a_i	5	7	0	3	4	2	6	1
V_i	0	0	2	2	2	4	1	6

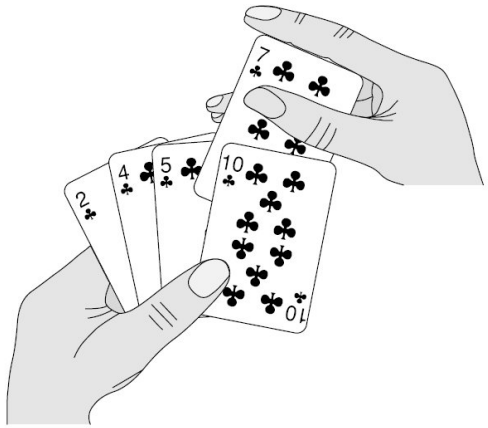
Properties of Algorithm

1. **Inplace:** Only requires a constant amount O(1) of additional memory space
2. **Stable:** Does not change the relative order of elements with same keys
3. **Online:** Sort a list as it arrives
4. **Adaptive:** Performance adapts to the initial order of elements
5. **Incremental:** Builds the sorted sequence one number at a time

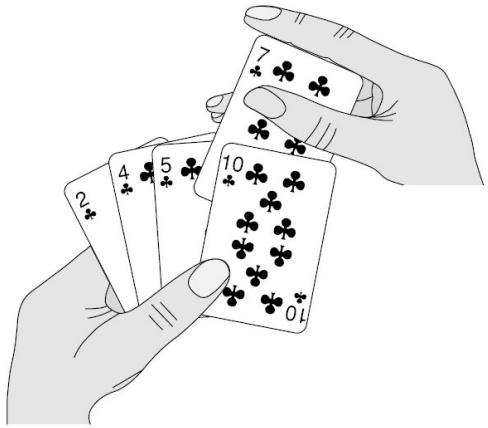
Insertion Sort

Introduction

It works the way you might sort a hand of playing cards:



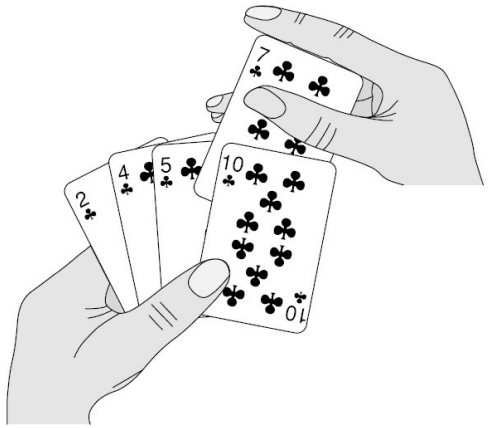
Introduction



It works the way you might sort a hand of playing cards:

1. We start with an empty left hand **(sorted array)** and the cards face down on the table **(unsorted array)**.

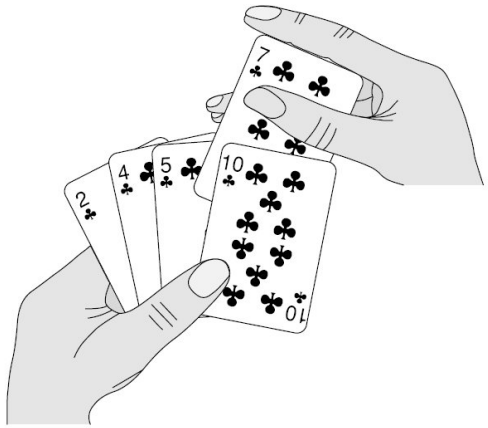
Introduction



It works the way you might sort a hand of playing cards:

1. We start with an empty left hand **(sorted array)** and the cards face down on the table **(unsorted array)**.
2. Then remove one card **(key)** at a time from the table, and insert it into the correct position in the left hand.

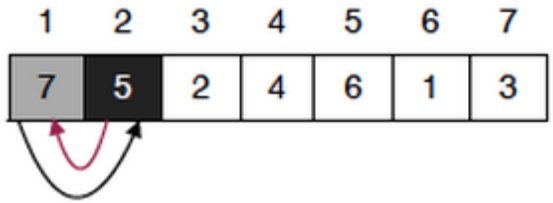
Introduction



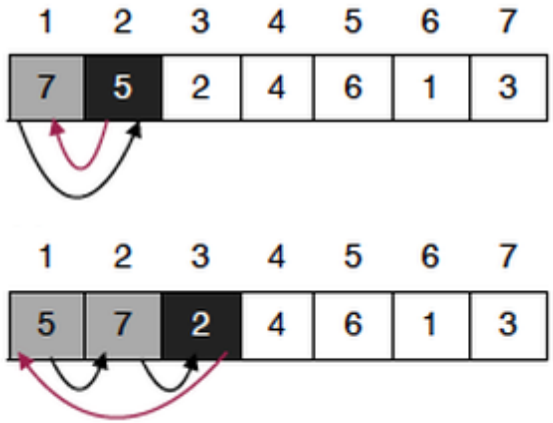
It works the way you might sort a hand of playing cards:

1. We start with an empty left hand **(sorted array)** and the cards face down on the table **(unsorted array)**.
2. Then remove one card **(key)** at a time from the table, and insert it into the correct position in the left hand.
3. To find the correct position for the key, we compare it with each of the cards already in the hand, from right to left.

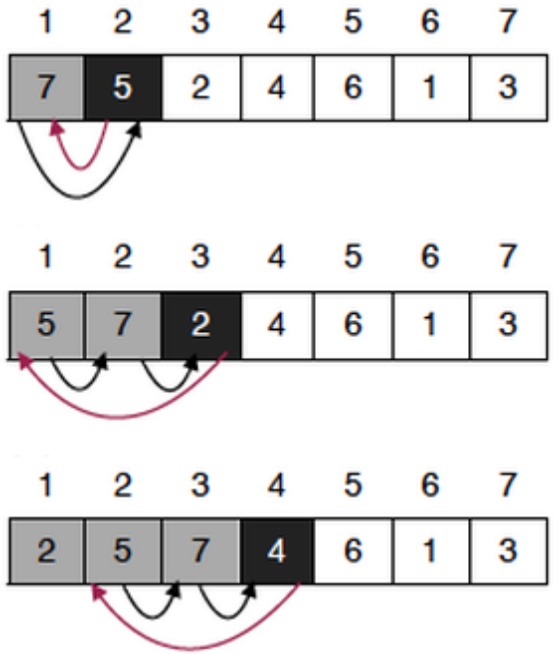
Simulation



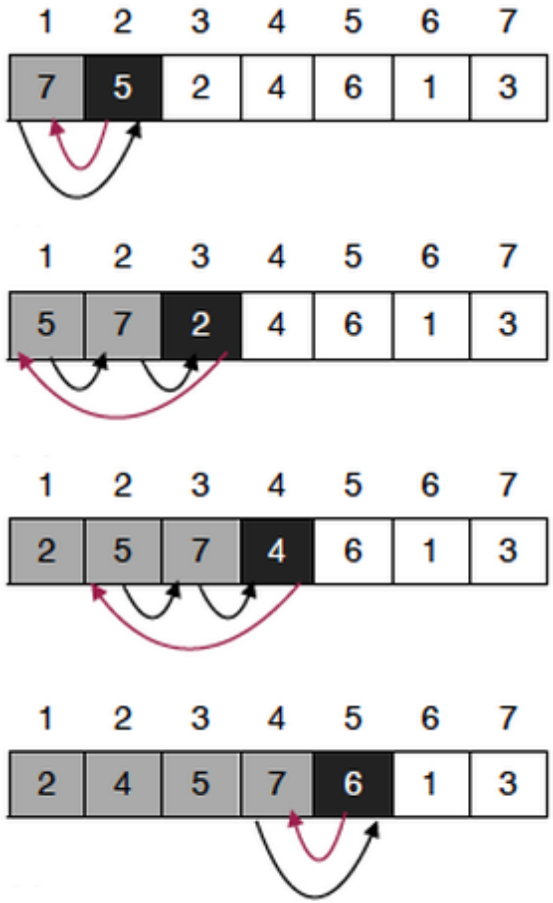
Simulation



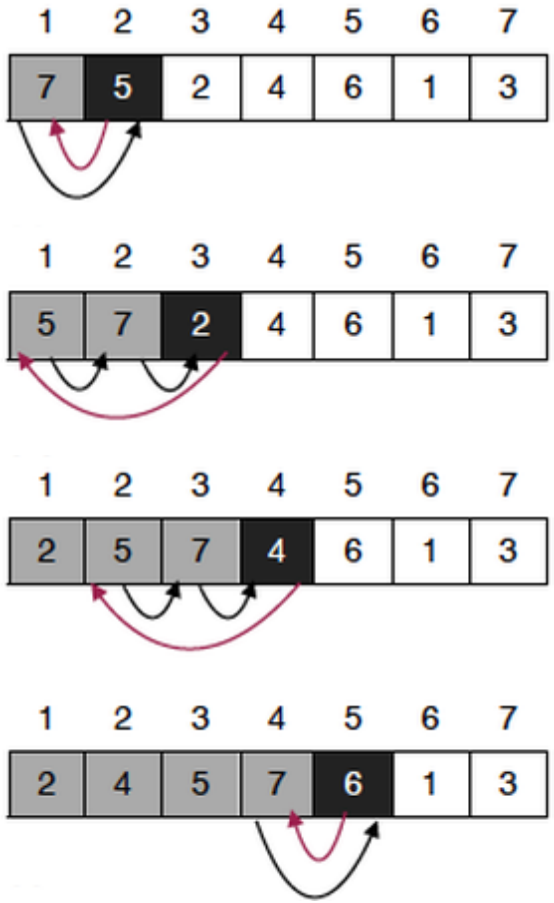
Simulation

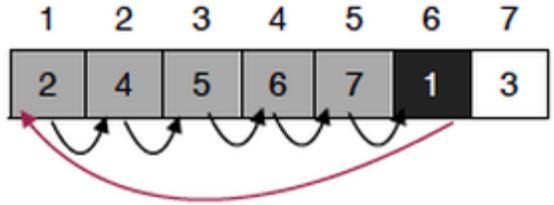


Simulation

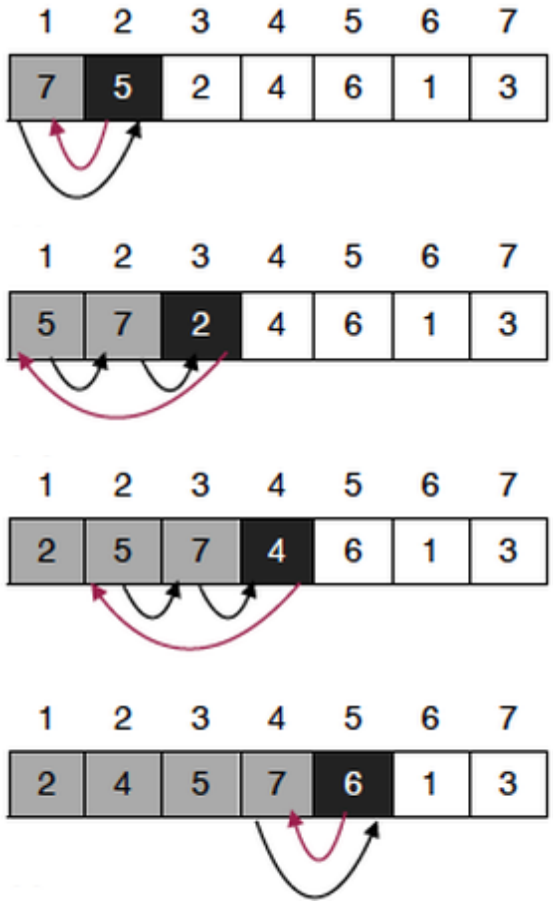

1 2 3 4 5 6 7
7 5 2 4 6 1 3
1 2 3 4 5 6 7
5 7 2 4 6 1 3
1 2 3 4 5 6 7
2 5 7 4 6 1 3
1 2 3 4 5 6 7
2 4 5 7 6 1 3

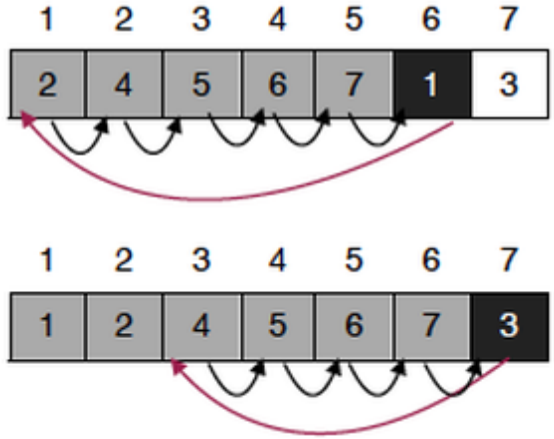
Simulation


1 2 3 4 5 6 7
7 5 2 4 6 1 3
1 2 3 4 5 6 7
5 7 2 4 6 1 3
1 2 3 4 5 6 7
2 5 7 4 6 1 3
1 2 3 4 5 6 7
2 4 5 7 6 1 3

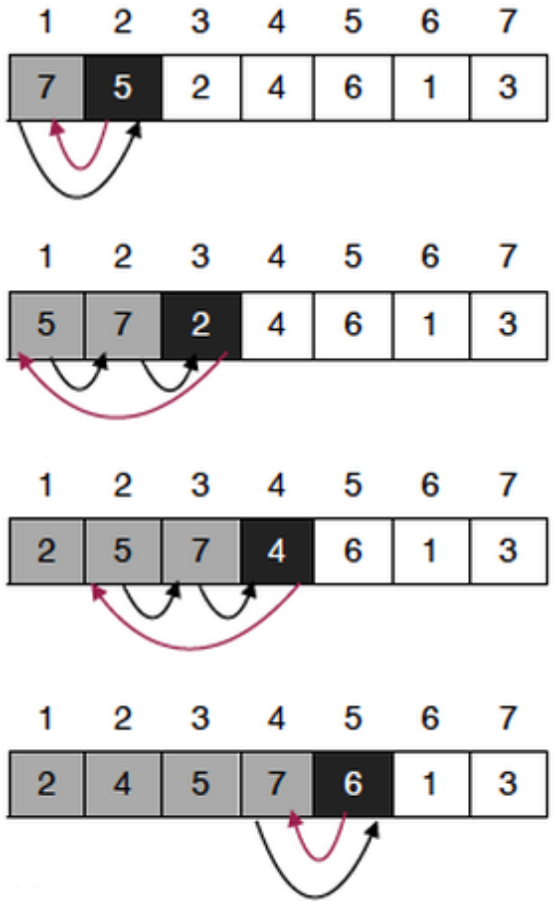

1 2 3 4 5 6 7
2 4 5 6 7 1 3

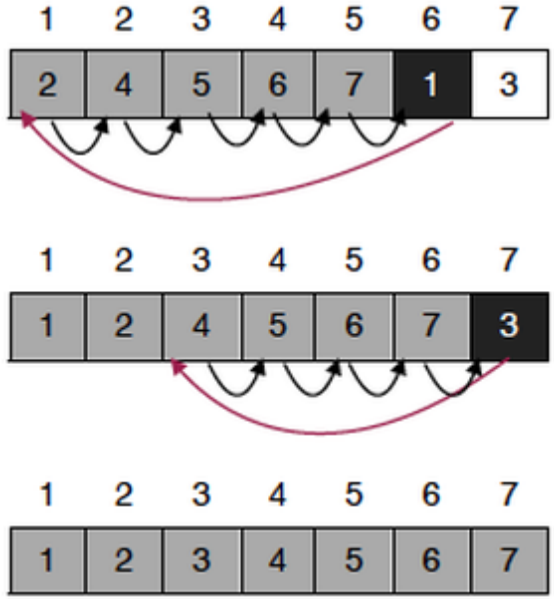
Simulation


1 2 3 4 5 6 7
7 5 2 4 6 1 3
1 2 3 4 5 6 7
5 7 2 4 6 1 3
1 2 3 4 5 6 7
2 5 7 4 6 1 3
1 2 3 4 5 6 7
2 4 5 7 6 1 3


1 2 3 4 5 6 7
2 4 5 6 7 1 3
1 2 3 4 5 6 7
1 2 4 5 6 7 3

Simulation


1 2 3 4 5 6 7
7 5 2 4 6 1 3
1 2 3 4 5 6 7
5 7 2 4 6 1 3
1 2 3 4 5 6 7
2 5 7 4 6 1 3
1 2 3 4 5 6 7
2 4 5 7 6 1 3


1 2 3 4 5 6 7
2 4 5 6 7 1 3
1 2 3 4 5 6 7
1 2 4 5 6 7 3
1 2 3 4 5 6 7
1 2 3 4 5 6 7

Complexity Analysis

Instruction	Cost	Time
for $j \leftarrow 2$ *to* $length(A)$	c_1	n
$key \leftarrow A[j]$	c_2	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
while$(i > 0$ *and* $A[i] > key)$	c_4	$\sum_{j=2}^{n} t_j$
$A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
end		
$A[i+1] \leftarrow key$	c_7	$n-1$
end		

$$T(n) = c_1 * n + c_2 * (n-1) + c_3 * (n-1) + c_4 * \sum_{j=2}^{n} t_j + c_5 * \sum_{j=2}^{n}(t_j-1) + c_6 * \sum_{j=2}^{n}(t_j-1) + c_7 * (n-1)$$

Best Case Analysis

$$T(n) = c_1 * n + c_2 * (n-1) + c_3 * (n-1) + c_4 * \sum_{j=2}^{n} t_j + c_5 * \sum_{j=2}^{n} (t_j - 1) + c_6 * \sum_{j=2}^{n} (t_j - 1) + c_7 * (n-1)$$

Best case for insertion sort occurs when data is already sorted, and so while loop condition would be false for each element. For this case $t_j = 1$

$$T(n) = c_1 * n + c_2 * (n-1) + c_3 * (n-1) + c_4 * \sum_{j=2}^{n} 1 + c_5 * \sum_{j=2}^{n} (0) + c_6 * \sum_{j=2}^{n} (0) + c_7 * (n-1)$$

Where $\sum_{j=2}^{n} 1 = (n-1)$

$$= c_1 * n + c_2 * (n-1) + c_3 * (n-1) + c_4 * (n-1) + c_7 * (n-1)$$
$$= c_1 n + c_2 n - c_2 + c_3 n - c3 + c_4 n - c_4 + c_7 n - c_7$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c3 + c_4 + c_7)$$
$$= an + b, \text{ Which is linear function of n}$$

So, $\mathbf{T(n) = O(n)}$

Worst Case Analysis

$$T(n) = c_1 * n + c_2 * (n-1) + c_3 * (n-1) + c_4 * \sum_{j=2}^{n} t_j + c_5 * \sum_{j=2}^{n} (t_j - 1) + c_6 * \sum_{j=2}^{n} (t_j - 1) + c_7 * (n-1)$$

Worst case for insertion sort occurs when data is inversely sorted, and so while loop condition would be true j times for each element. For this case $t_j = j$

$$T(n) = c_1 * n + c_2 * (n-1) + c_3 * (n-1) + c_4 * \sum_{j=2}^{n} j + c_5 * \sum_{j=2}^{n} (j - 1) + c_6 * \sum_{j=2}^{n} (j - 1) + c_7 * (n-1)$$

Where $\sum_{j=2}^{n} j = (2 + 3 + ... + n) - 1 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$

$$\sum_{j=2}^{n} (j-1) = 1 + 2 + 3 + ... + (n-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n-1)}{2}) + c_6(\frac{n(n-1)}{2}) + c_7(n-1)$$
$$= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 \frac{n^2}{2} + c_4 \frac{n}{2} - c_4 + c_5 \frac{n^2}{2} - c_5 \frac{n}{2} + c_6 \frac{n^2}{2} - c_6 \frac{n}{2} + c_7 n - c_7$$
$$= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}) n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7) n + (-c_2 - c_3 - c_4 + c_7)$$
$$= an^2 + bn + c, \text{ Which is quadratic function of n}$$

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

Average Case Analysis

- Average case is often roughly as bad as worst case.
- On an average, half of the elements are greater than $A[j]$ and other is less than that. So, $t_j = \frac{j}{2}$. It again turns out to be quadratic function of n.
- So, T(n) = $O(n^2)$

Assumption

- Ignore actual cost of operations
- Ignore abstract cost C
- Consider only leading term of polynomial
- Drop the constant if any

Properties

- In place: Require O(1) Extra space
- Stable: Does not change relative order of elements with same keys
- $O(n^2)$ comparison and swaps in worst case
- Online: Can sort a list as it receives
- Adaptive: O(n) when nearly sorted, $O(n^2)$ otherwise
- Incremental: Sort one element at a time
- Low overhead
- Simple implementation
- Efficient for small or nearly sorted data set
- More efficient than other quadratic algorithms

Properties

- Often used as the recursive base case (when the problem size is small) for higher overhead divide and conquer sorting algorithms, such as merge sort or quick sort
- It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.
- For data sets that are already substantially sorted, the time complexity is O(n + d), where d is the number of inversions
- Insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort

Properties

- Good quicksort implementations use insertion sort for arrays smaller than a certain threshold, commonly between 8 to 10.
- Insertion sort only scans as many elements as needed to determine the correct location of the $(k+1)^{st}$ element, while selection sort must scan all remaining elements
- For random sequence insertion sort usually perform about half as many comparisons as selection sort.
- If the input array is reverse-sorted, insertion sort performs as many comparisons as selection sort.
- If the input array is already sorted, insertion sort performs as few as n-1 comparisons

Properties

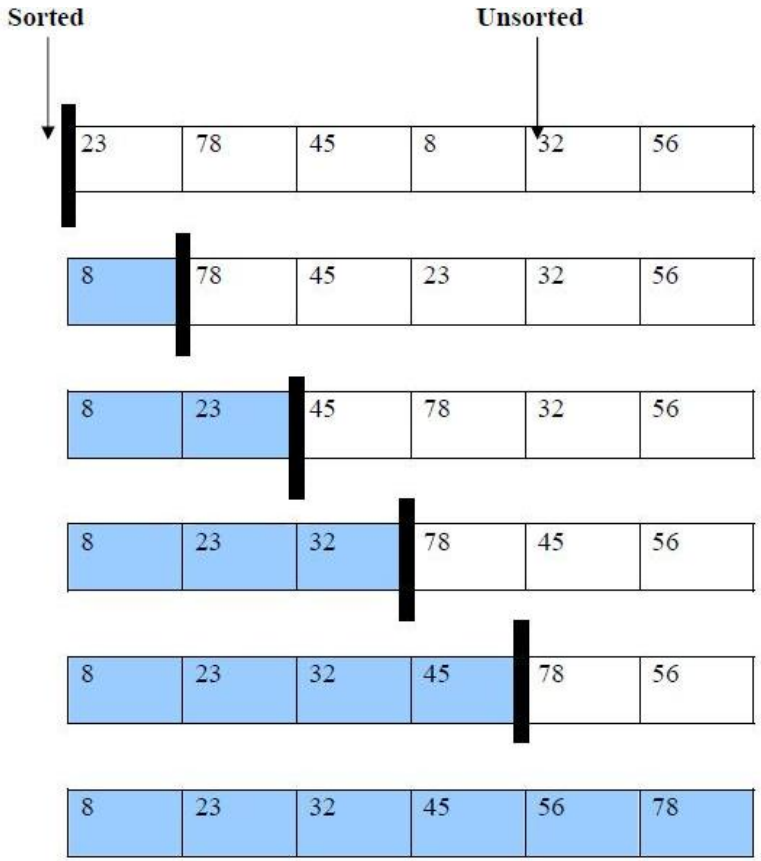
- While insertion sort typically makes fewer comparisons than selection sort, it requires more writes
- Insertion sort: $O(n^2)$ Writes, Selection sort: O(n) Writes
- Selection sort is preferable in where writing to memory is significantly more expensive (eg. EEPROM or flash memory)
- Variation - Shell Sort: Compares elements separated by distance that decreases on each path
- Number of swaps can be reduced by calculating the position of multiple elements before moving
- For example, if the target position of two elements is calculated before they are moved, the number of swaps can be reduced by about 25% for random data.

Selection Sort

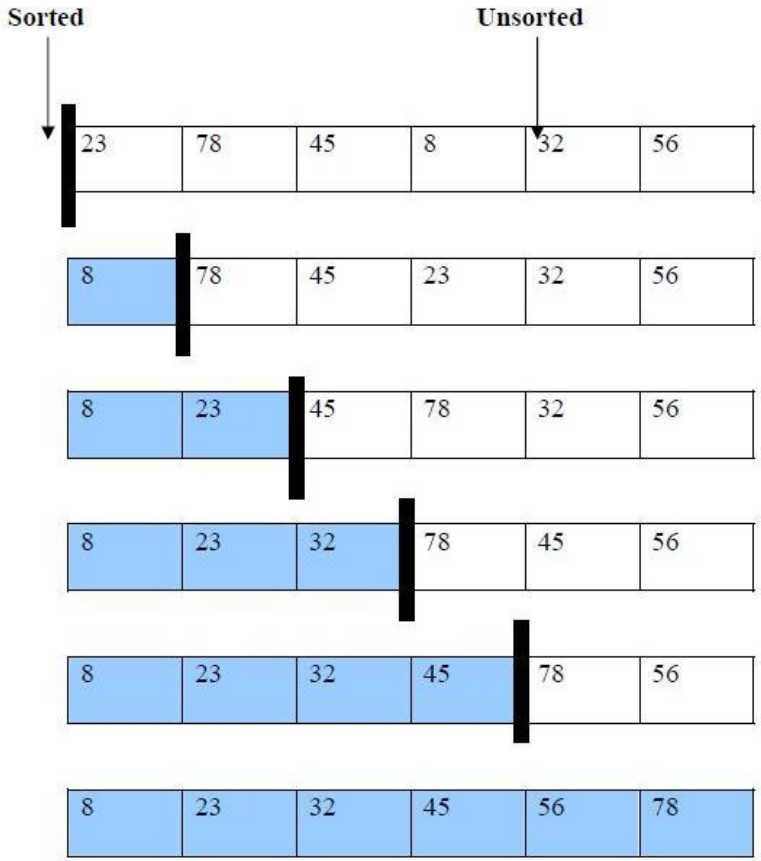
Definition

- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

Pseudo Code


Sorted
Unsorted
23 78 45 8 32 56
8 78 45 23 32 56
8 23 45 78 32 56
8 23 32 78 45 56
8 23 32 45 78 56
8 23 32 45 56 78

Pseudo Code



SELECTION_SORT(A)

for $i \leftarrow 0$ *to* $length[A] - 1$
    $min \leftarrow A[i]$
    for $j \leftarrow (i + 1)$ *to* $length[A]$
        if $A[j] < min$
            $min \leftarrow A[i]$
        end
    end
    $swap(A[i], min)$
end

Simulation

Pass	Data Sequence					Comparisons
1	74	25	16	22	10	(n-1)
2	10	25	16	22	74	(n-2)
3	10	16	25	22	74	.
4	10	16	22	25	74	.
5	10	16	22	25	74	1
6	10	16	22	25	74	0

$$
\begin{aligned}
T(n) &= 1 + 2 + 3 + \ldots + (n-1) \\
&= \sum (n-1) \\
&= \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \\
T(n) &= O(n^2)
\end{aligned}
$$

Stable and Non Stable Implementation

Non Stable Version:
Swap min and A[i]
Number of Writes: O(n)

```
64 25 12 22 11
11 25 12 22 64
11 12 25 22 64
11 12 22 25 64
11 12 22 25 64
```

Stable Version:
Insert min in front of A[i]
Number of Writes: $O(n^2)$

```
64 25 12 22 11
11 64 25 12 22
11 12 64 25 22
11 12 22 64 25
11 12 22 25 64
```

Complexity Analysis

- Selection sort makes $(n-1)$ steps of outer loop. Every step of outer loop requires finding minimum in unsorted part. Summing up, (n - 1) + (n - 2) + ...+ 1, results in $O(n^2)$ comparisons.
- Worst case occurs if the array is already sorted in descending order.
- Nonetheless, the time require by selection sort algorithm is not very sensitive to the original order of the array to be sorted.
- Best Case = Average Case = Worst Case = $O(n^2)$
- Number of swaps may vary from zero (in case of sorted array) to (n - 1) (in case array was sorted in reversed order), which results in O(n) number of swaps.

Properties

- In place - O(1) Extra Space: Advantage over other algorithm when auxiliary memory is limited
- Not Adaptive
- Not stable
- Can be implemented as a stable sort if, rather than swapping in step 2, the minimum value is inserted into the first position (that is, all intervening items moved down)
- It requires a data structure that supports efficient insertions , such as a linked list, or it leads to performing $O(n^2)$ writes.
- $O(n^2)$ Comparison
- $O(n)$ Swap

Properties

- Section sort is a method of choice for sorting files with very large objects (records) and small keys
- Inefficient for large list due to $O(n^2)$ complexity, generally perform worse than insertion sort
- Selection sort: O(n) Swaps, Insertion sort: (n^2) swaps
- Among simple average-case (n^2) algorithms, selection sort almost always outperforms bubble sort and gnome sort.
- Choice of selection when if writes are significantly more expensive than reads, (EEPROM or Flash memory), where every write lessens the lifespan of the memory.
- Heap allows finding next lowest element in $O(log_2 n)$ time instead of O(n), reducing the total running time to $O(nlog_2 n)$.

Properties

- Variant Cocktail Sort finds minimum and maximum values in every pass.
- It reduces number of scans by a factor of 2, eliminating some loop overhead but not actually decreasing the number of comparisons or swaps.
- Note: Cocktail sort more often refers to a bidirectional variant of bubble sort.
- Theoretically Cycle Sort is optimal in number of swaps
- Selection sort almost always far exceeds (and never beats) cycle sort

Properties

- Bingo Sort: A variant of selection sort orders items by first finding the least value, then repeatedly moving all items with that value to their final location and find the least value for the next pass.
- This is more efficient than selection sort if there are many duplicate values.
- Selection sort does one pass for each **item**
- Bingo sort does one pass for each **value**

Bubble Sort

Introduction

- The bubble sort is also known as the **Ripple Sort**.
- Sometimes *incorrectly* referred as **Sinking Sort**
- It Compares each pair of adjacent items and swapping them if they are in the wrong order
- The algorithm gets its name from the way larger elements bubble up to the top of the list.
- In every iteration K, list is divided into two parts, 1 to n-K, and K to n. Left side sub list contains unsorted data and right side list contain sorted data.
- It has ability to detect the list is sorted

Pseudo Code

BUBBLE_SORT(A)

for $i \leftarrow 1$ *to* $(n-1)$
    for $j \leftarrow 1$ *to* $(n-i)$
        if $A[j] > A[j+1]$
            $A[i] \leftrightarrow A[j+1]$
        end
    end
end

Simulation

Sort the sequence: 6 1 2 3 5 4

6 **1** 2 3 5 4 (Swap)
1 **6** **2** 3 5 4 (Swap)
1 2 **6** **3** 5 4 (Swap)
1 2 3 **6** **5** 4 (Swap)
1 2 3 5 **6** **4** (Swap)
1 2 3 5 4 6

Simulation

Sort the sequence: 6 1 2 3 5 4

6 1 2 3 5 4 (Swap)
1 **6 2** 3 5 4 (Swap)
1 2 **6 3** 5 4 (Swap)
1 2 3 **6 5** 4 (Swap)
1 2 3 5 **6 4** (Swap)
1 2 3 5 4 6

1 2 3 5 4 6 (No Swap)
1 **2 3** 5 4 6 (No Swap)
1 2 **3 5** 4 6 (No Swap)
1 2 3 **5 4** 6 (Swap)
1 2 3 4 **5 6** (No Swap)
1 2 3 4 5 6

Complexity Analysis

- Bubble sort makes (n-1) steps of outer loop. Every step of outer loop requires finding maximum in unsorted part. Summing up, (n-1) + (n-2) + ...+ 1, results in $O(n^2)$ number of comparisons
- Best Case = Average Case = Worst Case = $O(n^2)$
- Number of swaps may vary from zero (in case of sorted array) to (n - 1) (in case array was sorted in reversed order), which results in O(n) number of swaps.

Turtle and Rabbit Example

- **Turtle Example:**
  Thought array $\{2, 3, 4, 5, 1\}$ is almost sorted, it takes $O(n^2)$ iterations to sort an array. Element 1 is a turtle.

Turtle and Rabbit Example

- **Turtle Example:**
  Thought array $\{2, 3, 4, 5, 1\}$ is almost sorted, it takes $O(n^2)$ iterations to sort an array. Element 1 is a turtle.

- **Rabbit Example:**
  Array $\{6, 1, 2, 3, 4, 5\}$ is almost sorted too, but it takes O(n) iterations to sort it. Element $\{6\}$ is a rabbit.

  This example demonstrates adaptive property of the bubble sort.

Properties

- O(1) Extra Space
- $O(n^2)$ Comparisons
- $O(n^2)$ Swaps
- Adaptive: O(n) when nearly sorted
- If the algorithm is realized with a *bigger* and *not a bigger or same* clause, it is stable.

Properties

- Due to its simplicity, bubble sort is often used to introduce the concept of sorting algorithm
- **Donald Knuth**, in his famous book *The Art of Computer Programming*, concluded that "The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems"
- Bubble sort interacts poorly with modern CPU hardware
- It has ability to detect that the list is sorted
- It requires at least twice as many writes as insertion sort.
- Experiments by Astrachan show bubble sort is roughly 5 times slower than insertion sort and 40% slower than selection sort

Properties

- Large elements at the beginning of list move quite faster towards end. **(Rabit / Hare)**
- Small elements at the end of the list move extremely slower towards beginning. **(Turtle)**
- Cocktail sort goes from beginning to end, and then reverses itself, going end to beginning. It can move turtles fairly well
- But it retains $O(n^2)$ worst-case complexity
- Comb sort compares elements separated by large gaps, and can move turtles extremely quickly before proceeding to smaller and smaller gaps to smooth out the list.
- Its average speed is comparable to faster algorithms like quicksort

Properties

- Bubble sort is also efficient when one random element needs to be sorted into a *sorted list*, provided that new element is placed at the beginning and not at the end.
- If the random element is placed at the end, bubble sort loses its efficiency because each element greater than it must bubble all the way up to the top.

Merge Sort

Fundamentals

- Merge sort was invented by John von Neumann in 1945.
- Divide and Conquer Approach
- Most implementations produce a stable sort
- It incorporates two main ideas to improve its runtime:
  1. A small list will take fewer steps to sort than a large list.
  2. Fewer steps are required to construct a sorted list from two sorted lists than from two unsorted lists
- If the list is of length 0 or 1, then it is already sorted. Otherwise:
  1. Divide the unsorted list into two sub lists of about half the size.
  2. Sort each sub list recursively by re-applying the merge sort
  3. Merge the two sub lists back into one sorted list.

Approaches

1. **Top Down Approach:**
   It uses recursion to divide the list into sub-lists, then merges sublists during returns back up the call chain

2. **Bottom Up Approach:**
   It treats the list as an array of n sublists of size 1, and iteratively merges sub-lists back and forth between two buffers

Pseudo Code

MERGE_SORT(A, p, r)

if $p < r$
    $q = floor((p + r)/2)$
    $MERGE\_SORT(A, p, q)$
    $MERGE\_SORT(A, q + 1, r)$
    $MERGE(A, p, q, r)$

Pseudo Code

MERGE(A, p, q, r)

$n_1 \leftarrow (q - p + 1)$
$n_2 \leftarrow (r - q)$

for $i \leftarrow 1$ *to* n_1
    $L[i] \leftarrow A[p + i - 1]$

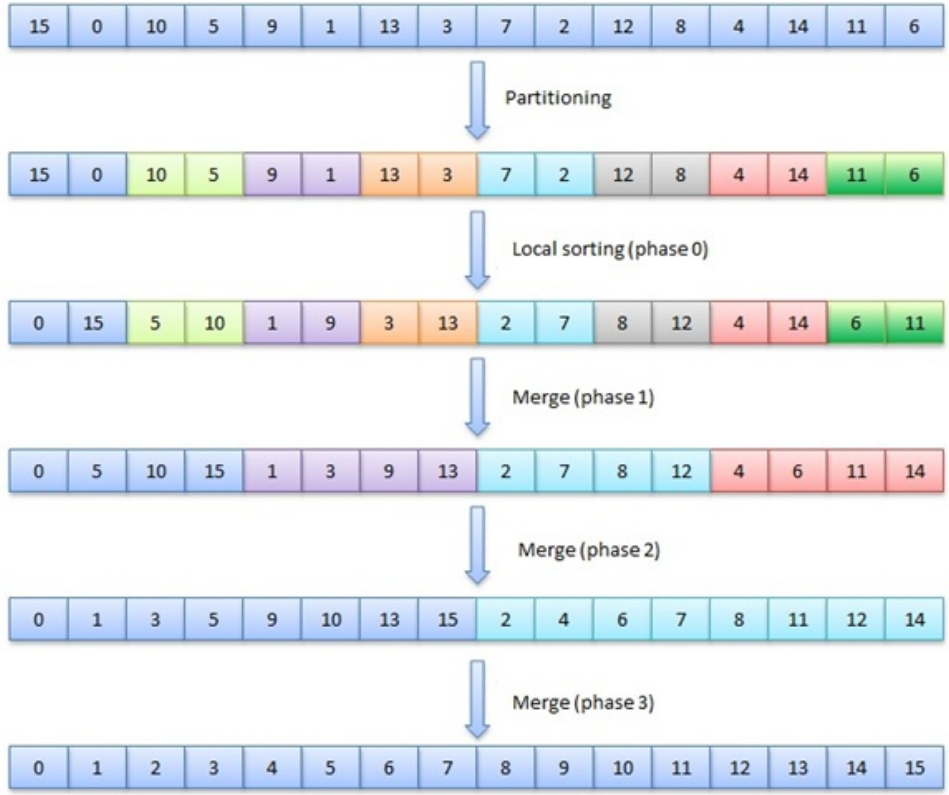
for $j \leftarrow 1$ *to* n_2
    $R[i] \leftarrow A[q + j]$

$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$

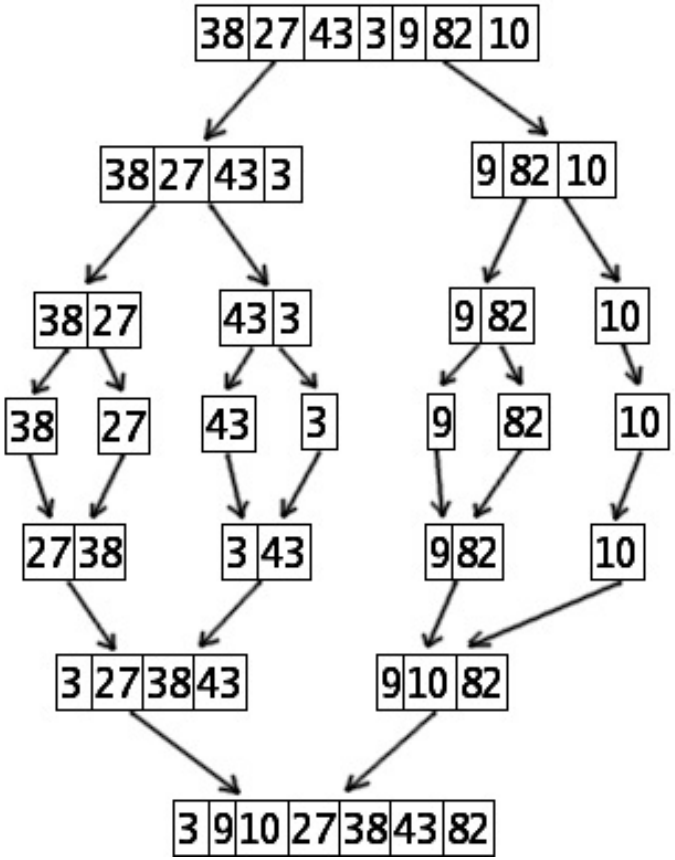
$i \leftarrow 1$
$j \leftarrow 1$

for $k \leftarrow p$ *to* r
    if $L[i] \leq R[j]$
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
    end
end

Simulation


15 0 10 5 9 1 13 3 7 2 12 8 4 14 11 6
Partitioning
15 0 10 5 9 1 13 3 7 2 12 8 4 14 11 6
Local sorting (phase 0)
0 15 5 10 1 9 3 13 2 7 8 12 4 14 6 11
Merge (phase 1)
0 5 10 15 1 3 9 13 2 7 8 12 4 6 11 14
Merge (phase 2)
0 1 3 5 9 10 13 15 2 4 6 7 8 11 12 14
Merge (phase 3)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Simulation


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Recurrence Formulation

- The divide step just computes the middle of the sub array, which takes constant time. Thus, $D(n) = \theta(1)$.
- We recursively solve two sub problems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- We have already noted that the MERGE procedure on an n-element sub array takes time $\theta(n)$, so $C(n) = \theta(n)$.

$$\mathrm{T(n)} = \begin{cases} 0 & \text{if } n \leq 1 \\ 2T(n/2) + D(n) + C(n) & \text{else} \end{cases}$$

$$T(n) = 2T(n/2) + \theta(1) + \theta(n)$$

$$T(n) = 2T(n/2) + n$$

Complexity Analysis

$$T(n) = 2T(n/2) + n$$

$$= 2[2T(n/4) + n/2] + n$$
$$= 4T(n/4) + 2n$$

$$= 4[2T(n/8) + n/4] + n$$
$$= 8T(n/8) + 3n$$

.

.

$$= 2^k T(\tfrac{n}{2^k}) + k.n$$

Complexity Analysis

$T(n) = 2T(n/2) + n$

$= 2[2T(n/4) + n/2] + n$
$= 4T(n/4) + 2n$

$= 4[2T(n/8) + n/4] + n$
$= 8T(n/8) + 3n$
.
.

$= 2^k T(\frac{n}{2^k}) + k.n$

$Assume\ n = 2^k \Longrightarrow k = log_2 n$

$= 2^k T(\frac{n}{2^k}) + k.n$
$= 2^k T(\frac{2^k}{2^k}) + k.n$
$= 2^k T(1) + k.n$
$= k.n \qquad (T(1) = 0)$
$= n.log_2 n \qquad (k = log 2_n)$

$T(n) = nlog_2 n$

Compute Steps

Hit The Bottom:

$T(16) = 2T(8) + 16$

$T(8) = 2T(4) + 8$

$T(4) = 2T(2) + 4$

$T(2) = 2T(1) + 2$

$T(1) = 0$

Compute Steps

Hit The Bottom:

$T(16) = 2T(8) + 16$

$T(8) = 2T(4) + 8$

$T(4) = 2T(2) + 4$

$T(2) = 2T(1) + 2$

$T(1) = 0$

Bounce Back:

$T(1) = 0$

$T(2) = 2T(1) + 2 = 2(0) + 2 = 2$

$T(4) = 2T(2) + 4 = 2(2) + 4 = 8$

$T(8) = 2T(4) + 8 = 2(8) + 8 = 24$

$T(16) = 2T(8) + 16 = 2(24) + 16 = 64$

Properties

- Merge sort's most common implementation does not sort in place
- In-place, Stable sorting is possible but is more complicated, and usually a bit slower with $O(nlog_2 n)$ time
- Like the standard merge sort, in-place merge sort is also stable.
- Merge sort is more efficient than quicksort for sequentially accessed data structures like tape.
- In the worst case, merge sort does about 39% fewer comparisons than quicksort does in the average case
- Merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, merge sort's worst case is found simultaneously with quick sorts best case.

Properties

- In terms of moves, merge sort's worst case complexity is same as quick sorts best case, and merge sort's best case takes about half as many iterations as worst case.
- Recursive implementations of merge sort make **(2n - 1)** calls in the worst case, compared to quick sorts **n**.
- Although heapsort has the same time bounds as merge sort, it requires only O(1) auxiliary space instead of merge sort's O(n), and is often faster in practical implementations.
- It makes between $0.5 * log_2 n$ and $log_2 n$ comparisons per element, and between $log_2 n$ and $1.5 * log_2 n$ swaps per element.
- Takes O(n) extra space to sort an array
- Takes $O(log_2 n)$ extra space to sort a linked list

Properties

- Cache-aware versions of the merge sort algorithm, whose operations have been specifically chosen to minimize the movement of pages in and out of a machine's memory cache.
- For example, the **Tiled Merge Sort** algorithm stops partitioning subarrays when subarrays of size S are reached.
- Where S is the number of data items fitting into a CPU's cache.
- Each of these subarrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion.
- This algorithm has demonstrated better performance on machines that benefit from cache optimization.

Properties

Application :

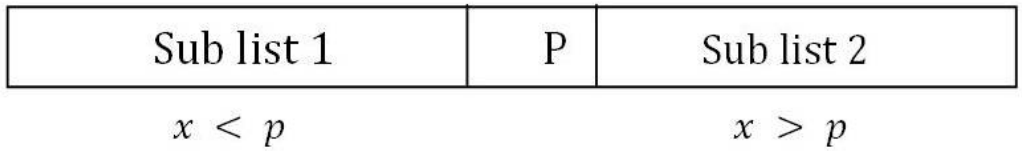
1. Merge Sort is useful for sorting linked lists in $O(nlog_2 n)$ time. Other $nlog_2 n$ algorithms like Heap Sort, Quick Sort cannot be applied to linked lists.
2. Merge sort is used to solve Inversion Count Problem
3. Used in External Sorting

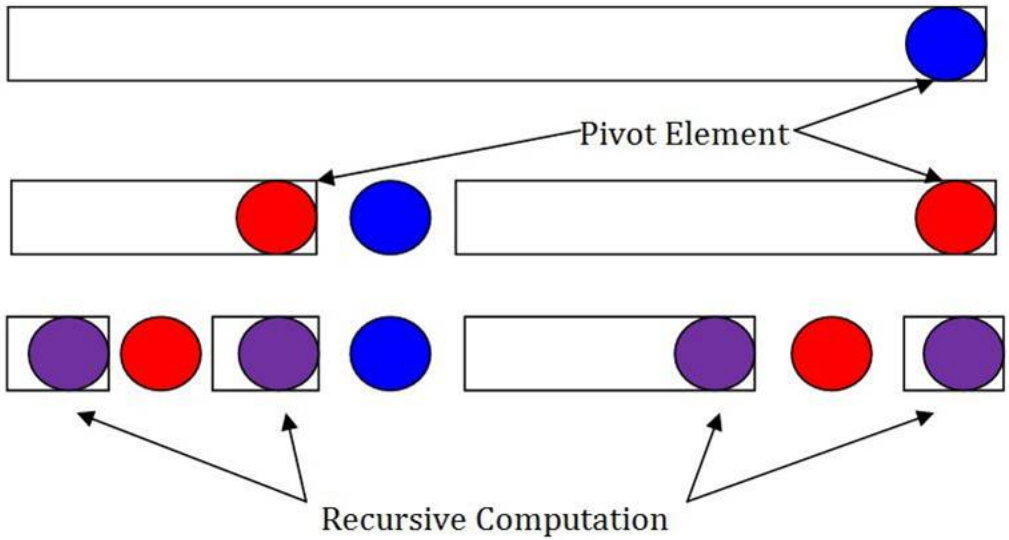
Quick Sort

Introduction

- Invented by Tony Hoare in 1960
- Also known as **Partition Exchange Sort**
- Divide and conquer based approach
- **Procedure:**
  1. Pick an element, called a pivot, from the list
  2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way).
  3. Recursively apply the above steps to the sub-list
- Partition is dynamically carried out
- In merge sort, division of array is based in position of array elements, but in quick sort, it is based on actual value of elements

Simulation


Sub list 1
P
Sub list 2
$x < p$
$x > p$


Pivot Element
Recursive Computation

Pseudo Code

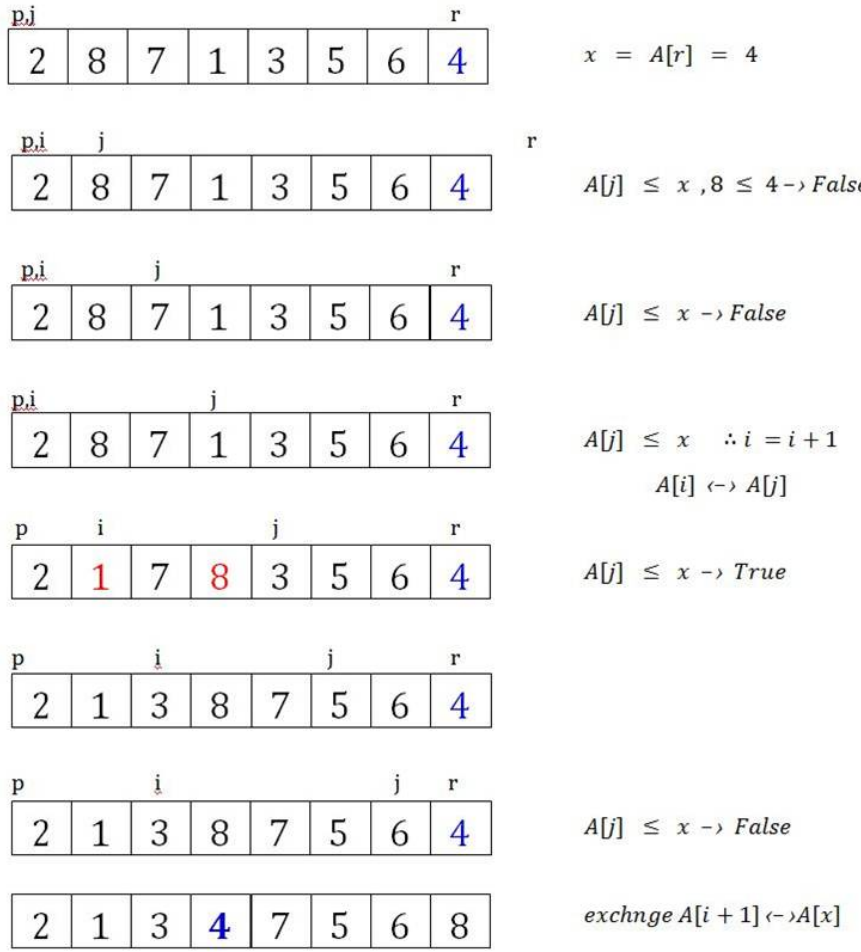
QUICKSORT(A, p, r)

```
if p < r then
    q ← PARTITION(A, p, r)
    QuickSORT(A, p, q-1)
    QuickSORT(A, q+1, r)
```

PARTITION(A, p, r)

```
QUICKSORT(A, p, r) x ← A[r]
i ← p − 1
for j ← p to r − 1
    if A[j] ≤ x then
        i ← i + 1
        A[i] ↔ A[j]
    end
end
A[i + 1] ↔ A[r]
return (i + 1)
```

Simulation



PARTITION(A, p, r)

$QUICKSORT(A, p, r)\ x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ *to* $r - 1$

  if $A[j] \leq x$ **then**

    $i \leftarrow i + 1$

    $A[i] \leftrightarrow A[j]$

  end

end

$A[i + 1] \leftrightarrow A[r]$

return $(i + 1)$

Recurrence Formulation

- **Divide:** Entire array is is scanned to fix up the exact location of PIVOT point. Thus, $D(n) = \theta(n)$.
- **Conquer:** We recursively solve two sub problems, each of size roughly n/2, which contributes 2T(n/2) to the running time.
- **Combine:** In each recursive call, PIVOT point automatically comes to its correct place, so $C(n) = \theta(1)$.

$$\mathrm{T(n)} = \begin{cases} 0 & \text{if } n \leq 1 \\ 2T(n/2) + D(n) + C(n) & \text{else} \end{cases}$$

$$T(n) = 2T(n/2) + \theta(n) + \theta(1)$$

$$T(n) = 2T(n/2) + n$$

Best Case Analysis

- Best case for Quick sort occurs when list spits from middle.
- In this case, Quick sort performs identical to Merge Sort

$$T(n) = 2T(n/2) + n$$

$$= 2[2T(n/4) + n/2] + n$$
$$= 4T(n/4) + 2n$$

$$= 4[2T(n/8) + n/4] + n$$
$$= 8T(n/8) + 3n$$

.

.

$$= 2^k T(\tfrac{n}{2^k}) + k.n$$

Best Case Analysis

- Best case for Quick sort occurs when list spits from middle.
- In this case, Quick sort performs identical to Merge Sort

$$T(n) = 2T(n/2) + n$$

$$= 2[2T(n/4) + n/2] + n$$
$$= 4T(n/4) + 2n$$

$$= 4[2T(n/8) + n/4] + n$$
$$= 8T(n/8) + 3n$$

.

.

$$= 2^k T(\tfrac{n}{2^k}) + k.n$$

$$Assume\ n = 2^k \Longrightarrow k = log_2 n$$

$$= 2^k T(\tfrac{n}{2^k}) + k.n$$
$$= 2^k T(\tfrac{2^k}{2^k}) + k.n$$
$$= 2^k T(1) + k.n$$
$$= k.n \qquad (T(1) = 0)$$
$$= n.log_2 n \qquad (k = log2_n)$$

$$T(n) = nlog_2 n$$

Worst Case Analysis

- Worst case for Quick sort occurs when list is sorted
- So that, after each iteration, one sublist contains (n-1) elements and second sublist contains 0 element
- In this case, Quick sort performs identical to Insertion Sort

Recurrence for worst case: T(n) = T(n-1) + n, T(1) = 1

$T(n-1) = T(n-2) + (n-1)$

$So, \;\; T(n) = T(n-2) + (n-1) + n$

$T(n-2) = T(n-3) + (n-2)$

$So, \;\; T(n) = T(n-3) + (n-2) + (n-1) + n$

.

.

$T(n) = T(1) + 2 + ... + (n-2) + (n-1) + n$

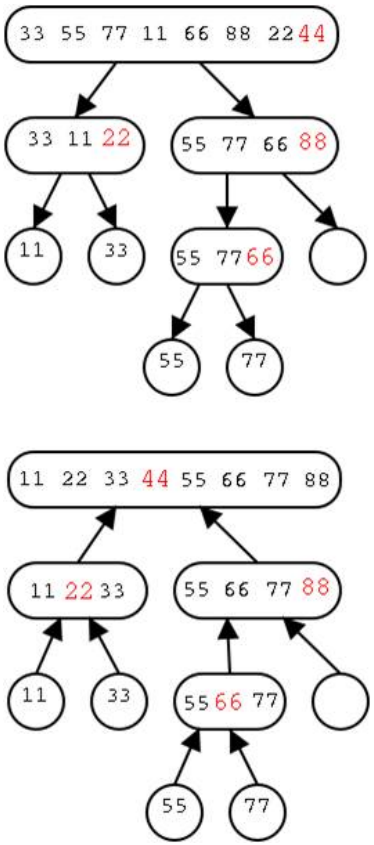
$T(n) = \sum n = O(n^2)$

Best Case Analysis Using Master Method

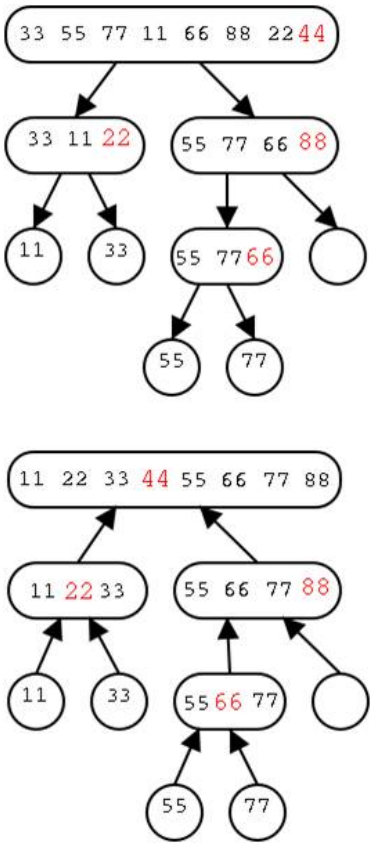
$T(n) = 2T(\frac{n}{2}) + n$
$a = 2,$
$b = 2,$
$f(n) = n,$
$d = 1$

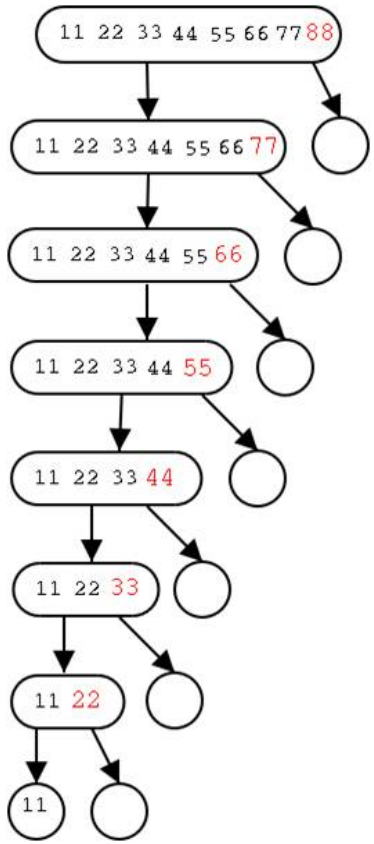
$Here,\ a = b^d$
$\therefore\ T(n) = \theta(n^d.logn)$
$\therefore\ T(n) = (n.logn)$

Best Case / Worst Case Partitioning



Best Case / Worst Case Partitioning





Randomized Quick Sort

- Use middle element of array of pivot.
- Use a random element of array as the pivot.
- Take the medium of three elements (first , last , middle) as pivot.

Properties

- Generally not Stable
- Not adaptive
- Massively Recursive
- Extremely fast
- Very complex
- Simple version requires O(n) extra storage space
- Additional memory allocations required can also drastically impact speed and cache performance
- More complex version uses an in-place partition algorithm that require $O(log_2 n)$ space (Recursion - Stack)

Properties

- Running time of quick sort depends on whether partitioning is balanced or unbalanced and this in turn, depends on which elements are used for partitioning.
- If the partitioning is balanced then quick sort runs asymptotically as fast as merge sort.
- If the partitioning is unbalanced, however it can run asymptotically as slow as Insertion sort.
- **Optimization:**
  1. To make sure at most O(log N) space is used, recurse first into the smaller half of the array, and use a *tail call* to recurse into the other
  2. Use insertion sort for small array (Size: 8 - 10), which has a smaller constant factor and is thus faster on small arrays

Properties

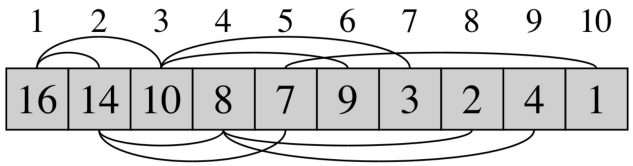
- Like mergesort, quicksort can be implemented as an in-place stable sort,but this is seldom done.
- Although quicksort can be written to operate on linked lists, it will often suffer from poor pivot choices without random access.
- The main disadvantage of mergesort is that, when operating on arrays, efficient implementations require O(n) auxiliary space, whereas the variant of quicksort with in-place partitioning and tail recursion uses only $O(log_2 n)$ space.
- When a stable sort is not needed, quick sort is an excellent general-purpose sort
- Never used in applications which require *Guaranteed Response Time*

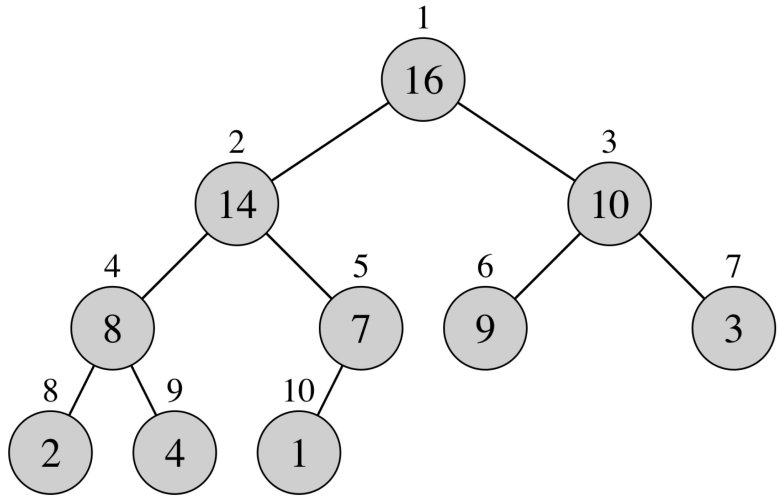
Heap Sort

Introduction

- It uses heap data structure, that can be viewed as a nearly complete binary tree
- **Procedure:**
  1. Build heap out of the data.
  2. sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array.
  3. The heap is reconstructed after each removal.
- The array can be split into two parts, the sorted array and the heap.
- Like merge sort, but unlike insertion sort, heap sorts running time is $O(nlog_2 n)$.
- Like insertion sort, but unlike merge sort, heap sort sorts in place.

Heap Data Structure

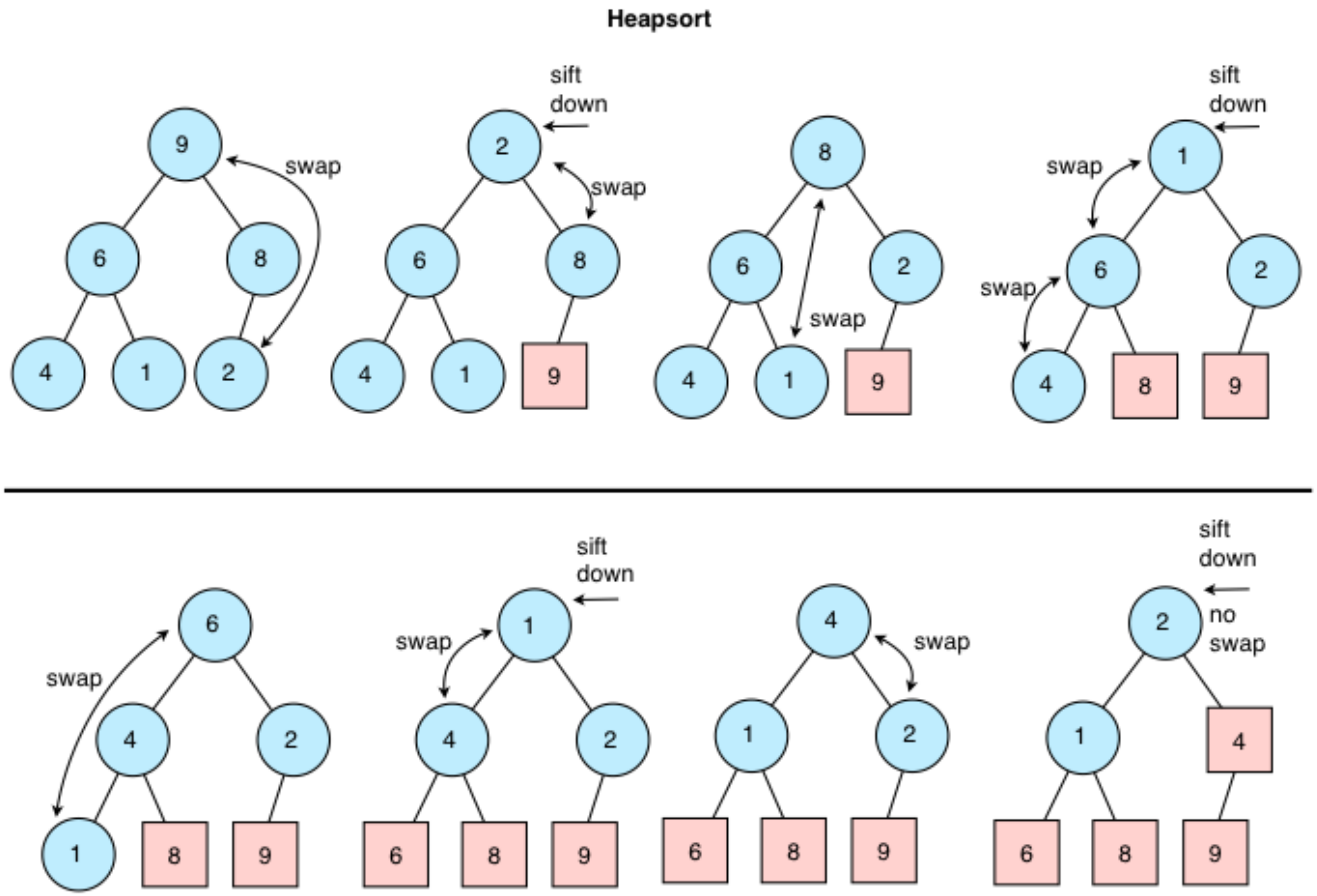




Heap Data Structure

- Height of a tree be a number of edges on the longest simple downward path from a root to a leaf.
- Heap of height h has the minimum number of elements when it has just one node at the lowest level
- Minimum number of nodes possible in a heap of height h is 2^h
- heap of height h, has the maximum number of elements when its lowest level is completely filled.
- in this case, heap is a complete binary tree of height h and hence has $2^{h+1} - 1$ nodes.
- Height of n element heap is $log_2 n$
- Heap with n elements: $2^h \leq n \leq 2^{h+1} - 1$

Simulation


Heapsort
swap
sift down
swap
swap
sift down
swap
swap
sift down
swap
swap
sift down
no swap

Index

```
PARENT(i)
    return floor(i/2)

LEFT(i)
    return floor(2i)

RIGHT(i)
    return floor(2i + 1)
```

Heap Properties

MAX HEAP:

- Property: $A[parent[i]] \geq A[i]$
- Largest element is at the root
- Used in Heap Sort

MIN HEAP:

- Property: $A[parent[i]] \leq A[i]$
- Smallest element is at the root
- Used in Priority Queue

Heap with n node has max height $O(log_2 n)$

Routines

- MAX_HEAPIFY: Runs in $O(log_2 n)$ time and is key to maintaining max-heap property.
- BUILD_MAX_HEAP: Runs in $O(n)$ time and produces a max-heap from unsorted array
- HEAP_SORT: Runs in $O(nlog_2 n)$ time and sorts array in place
- HEAP_EXTRACT_MAX: Runs in $O(n)$ time.

Heap Data Structure

MAX_HEAPIFY(A, i)

$l \leftarrow LEFT[i]$

$r \leftarrow RIGHT[i]$

if $l \leq heap\_size[A]$ *and* $A[l] > A[i]$ **then**

    $largest \leftarrow l$

else

    $largest \leftarrow i$

if $r \leq heap\_size[A]$ *and* $A[r] >$

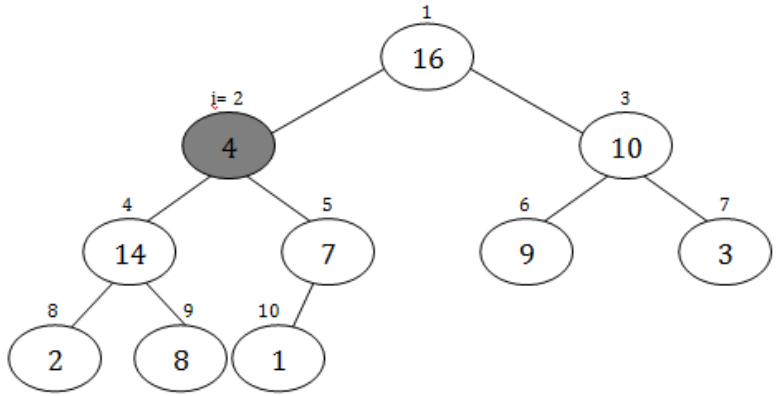
$A[largest]$ **then**

    $largest \leftarrow r$

if $largest \neq i$ **then**

    $A[i] \leftrightarrow A[largest]$

$MAX\_HEAPIFY(A, largest)$



Heap Data Structure

In above figure $heap\_size \leftarrow 10$

$i = 2$ $\quad A[i] = 4$

$l = LEFT[i] = 4$ $\quad A[l] = 14$

$r = RIGHT[i] = 5$ $\quad A[r] = 7$

$l \leq heap\_size$ *and* $A[l] >$ $A[i]$ **TRUE**

$largest = l, \quad Solargest = 4$

$A[largest] = 14$

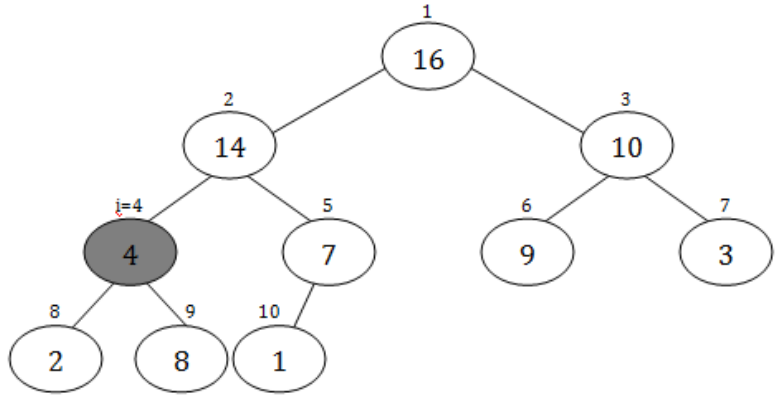
$r \leq heap\_size[A]$ *and* $A[r] >$ $A[largest]$ **FALSE**

$largest \neq i$ **TRUE**

$exchange(A[i] \leftrightarrow A[largest])$

$So, A[i] = A[2] = 14$ *and* $A[largest] = A[4] = 4$

CALL $MAX\_HEAPIFY(A, largest) = MAX\_HEAPIFY(A, 4)$



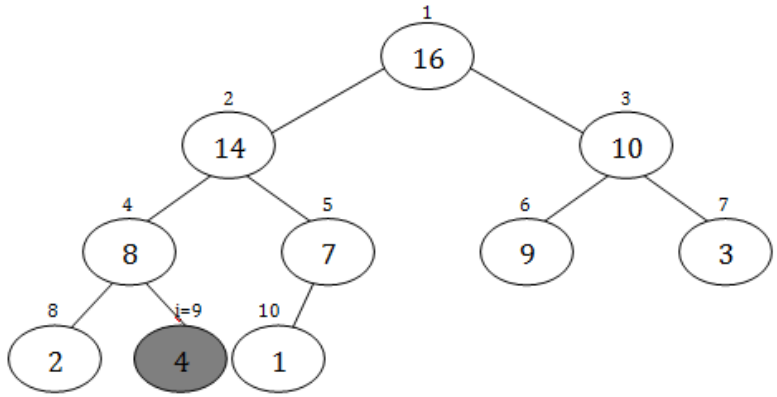
Heap Data Structure

$i = 4$ $A[i] = 4$
$l = LEFT[i] = 8$ $A[l] = 2$
$r = RIGHT[i] = 9$ $A[r] = 8$

$l \leq heap\_size$ *and* $A[l] >$
$A[i]$ **FALSE**
$largest = i$, $So largest = 4$
$A[largest] = 14$

$r \leq heap\_size[A]$ *and* $A[r] >$
$A[largest]$ **TRUE**
$largest \leftarrow r$, *So* $largest \leftarrow 9$
$A[largest] \leftarrow 8 largest \neq i$ **TRUE**
$exchange(A[i] \leftrightarrow A[largest])$
$So, A[i] = A[4] = 8$ *and* $A[largest] =$
$A[9] = 4$

CALL $MAX\_HEAPIFY(A, largest) =$
$MAX\_HEAPIFY(A, 4)$



Pseudo Code

HEAP_SORT(A)
BUILD_MAX_HEAP(A)
for $i \leftarrow length[A]$ *downto* 2
    $A[1] \leftrightarrow A[i]$
    $heap\_size[A] \leftarrow heap\_size[A] - 1$
    $MAX\_HEAPIFY(A, 1)$

BUILD_MAX_HEAP(A)
$heap\_size \leftarrow length[A]$
for $i \leftarrow ceil(length[A/2])$ *downto* 1
    MAX_HEAPIFY(A, i)

MAX_HEAPIFY(A, i)
$l \leftarrow LEFT[i]$
$r \leftarrow RIGHT[i]$
if $l \leq heap\_size[A]$ *and* $A[l] > A[i]$ **then**
    $largest \leftarrow l$
else
    $largest \leftarrow i$
if $r \leq heap\_size[A]$ *and* $A[r] > A[largest]$ **then**
    $largest \leftarrow r$
if $largest \neq i$ **then**
    $A[i] \leftrightarrow A[largest]$
$MAX\_HEAPIFY(A, largest)$

HEAP_SORT procedure takes time $O(nlog_2 n)$, since call to BUILD_MAX_HEAP takes time O(n), and each of the n - 1 calls to MAX_HEAPIFY takes time O(log n).

Properties

- In place
- Not stable
- Non adaptive
- Somewhat slower than a well-implemented Quick sort
- Advantage of a more favorable worst-case O(n log n) runtime.
- Because of the $O(nlog_2n)$ upper bound on running time and constant upper bound on auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use Heap sort
- Merge sort requires O(n) auxiliary space, but heapsort requires only a constant amount. Heapsort typically runs faster in practice on machines with *small or slow data caches*.

Properties

- Merge sort on arrays has considerably better data cache performance, often outperforming heap sort on modern computers
- Merge sort frequently accesses contiguous memory locations (good locality of reference); heap sort references are spread throughout the heap.
- Heap sort is not a stable sort; merge sort is stable.
- Merge sort parallelizes well and can achieve close to linear speedup with a trivial implementation;
- Heap sort is not an obvious candidate for a parallel algorithm.
- Merge sort is used in external sorting, heap sort is not.

Properties

- Merge sort can be adapted to operate on linked lists with O(1) extra space.
- Heap sort can be adapted to operate on doubly linked lists with only O(1) extra space overhead.

Counting Sort

Introduction

- Invented by Harold H. Seward in 1954
- Used when range of keys is relatively small and there are duplicate keys
- Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.
- Only suitable for direct use in situations where variation in keys is not significantly greater than the number of items.
- Often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

Pseudo Code

COUNTING_SORT(A,B,k)

for $i \leftarrow 0$ *to* k
    $C[i] \leftarrow 0$
for $j \leftarrow 1$ *to* $length[A]$
    $C[A[j]] \leftarrow C[A[j]] + 1$
    //C[i] now contains number of elements equal to i.
for $i \leftarrow 1$ *to* k
    $C[i] \leftarrow C[i] + C[i-1]$
    //C[i] now contains number of elements less than or equal to i
for $j \leftarrow length[A]$ *down to* 1
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]]1$

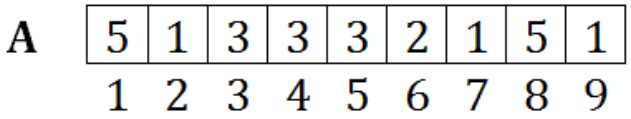
Simulation

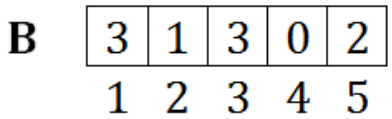
A	5	1	3	3	3	2	1	5	1
	1	2	3	4	5	6	7	8	9

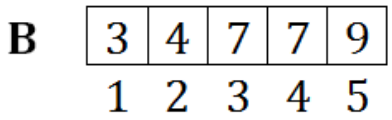
B	3	1	3	0	2
	1	2	3	4	5

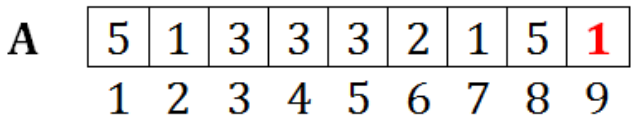
B	3	4	7	7	9
	1	2	3	4	5

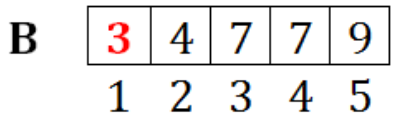
Simulation

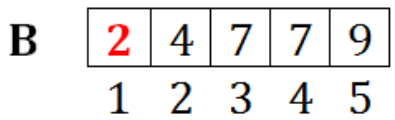

A
5 1 3 3 3 2 1 5 1
1 2 3 4 5 6 7 8 9

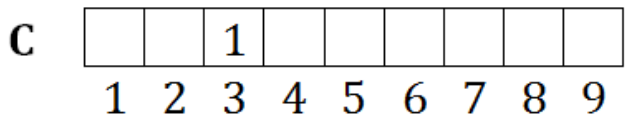

B
3 1 3 0 2
1 2 3 4 5


B
3 4 7 7 9
1 2 3 4 5

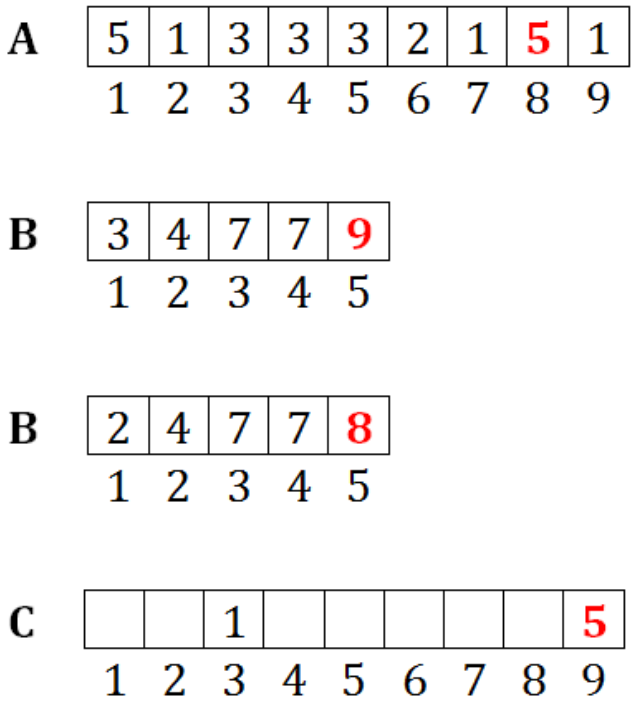

A
5 1 3 3 3 2 1 5 1
1 2 3 4 5 6 7 8 9


B
3 4 7 7 9
1 2 3 4 5


B
2 4 7 7 9
1 2 3 4 5


C
1
1 2 3 4 5 6 7 8 9

Simulation


A
5 1 3 3 3 2 1 5 1
1 2 3 4 5 6 7 8 9
B
3 4 7 7 9
1 2 3 4 5
B
2 4 7 7 8
1 2 3 4 5
C
1 5
1 2 3 4 5 6 7 8 9

Complexity Analysis

- The initialization of the *Count* array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time.
- The other two for loops, and the initialization of the output array, each take O(n) time.
- Therefore the time for the whole algorithm is the sum of the times for these steps, O(n + k).
- Because it uses arrays of length (k + 1) and n, the total space usage of the algorithm is also O(n + k).

Properties

- Stable
- Integer sorting algorithm
- It will actually not work if you try to sort anything else like *float*
- Not in place
- Modified in-place version of counting sort is not stable
- Counting sort can be made stable if we fill from backwards. If you fill in forward direction relative positions of equal elements will change
- For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient

Properties

- It is often used as a sub-routine to another sorting algorithm like Radix sort.
- Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
- When k=$O(n^2)$ or $O(n^3)$, we can say that the complexity of the counting sort is $O(n^2)$ or $O(n^3)$.
- May also be used to eliminate duplicate keys
- the algorithm can perform much worse in terms of memory usage if only single element in the input set is close to maximum range.
- Compared to counting sort, bucket sort requires preallocated memory to hold sets of items within each bucket, whereas counting sort instead stores a single number (the count of items) per bucket.

Radix Sort

Introduction

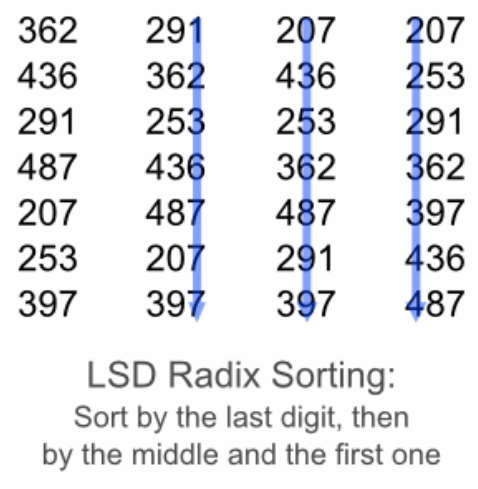
- Invented by Herman Hollerith in 1887
- Like Counting sort, Radix sort is not limited to integers only
- Classification: Least Significant Digit (LSD) radix sorts and Most Significant Digit (MSD) radix sorts.
- **Procedure:**
  1. Take the LSD of each key.
  2. Group the keys based on that digit, but keep the original order of keys. (This is what makes the LSD radix sort a stable sort).
  3. Repeat the grouping process with each more significant digit.
- Sort in step 2 is usually done using bucket / counting sort.

Pseudo Code

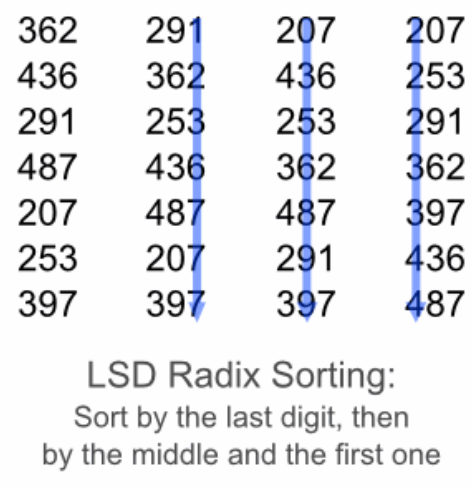
RADIX_SORT(A, d)

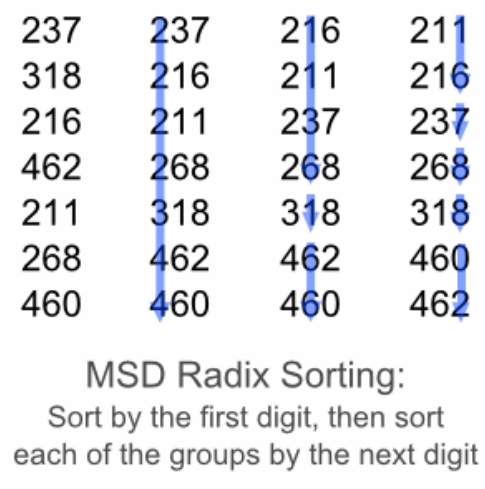
for $i \leftarrow 1$ *to* d
    Use stable sort to sort A on digit i
    \\Counting sort or bucket sort will do the job
end

Simulation


362 291 207 207
436 362 436 253
291 253 253 291
487 436 362 362
207 487 487 397
253 207 291 436
397 397 397 487
LSD Radix Sorting:
Sort by the last digit, then
by the middle and the first one

Simulation


362 291 207 207
436 362 436 253
291 253 253 291
487 436 362 362
207 487 487 397
253 207 291 436
397 397 397 487
LSD Radix Sorting:
Sort by the last digit, then
by the middle and the first one


237 237 216 211
318 216 211 216
216 211 237 237
462 268 268 268
211 318 318 318
268 462 462 460
460 460 460 462
MSD Radix Sorting:
Sort by the first digit, then sort
each of the groups by the next digit

Properties

- Radix sort efficiency is O(k.n) for n keys which have k or fewer digits.
- LSD Radix sort is fast and stable
- MSD Radix sort can be used to sort keys in lexicographic order.
- MSD Radix sort is not stable

Bucket Sort

Introduction

- Also known as **Bin Sort**
- Works by partitioning an array into a number of buckets
  1. Set up an array of initially empty buckets
  2. **Scatter:** Go over the original array, putting each object in its bucket.
  3. Sort each non-empty bucket
  4. **Gather:** Visit the buckets in order and put all elements back into the original array
- It is a distribution sort, and is a cousin of MSD Radix sort
- Generalization of pigeonhole sort.

Pseudo Code

BUCKET_SORT(A)

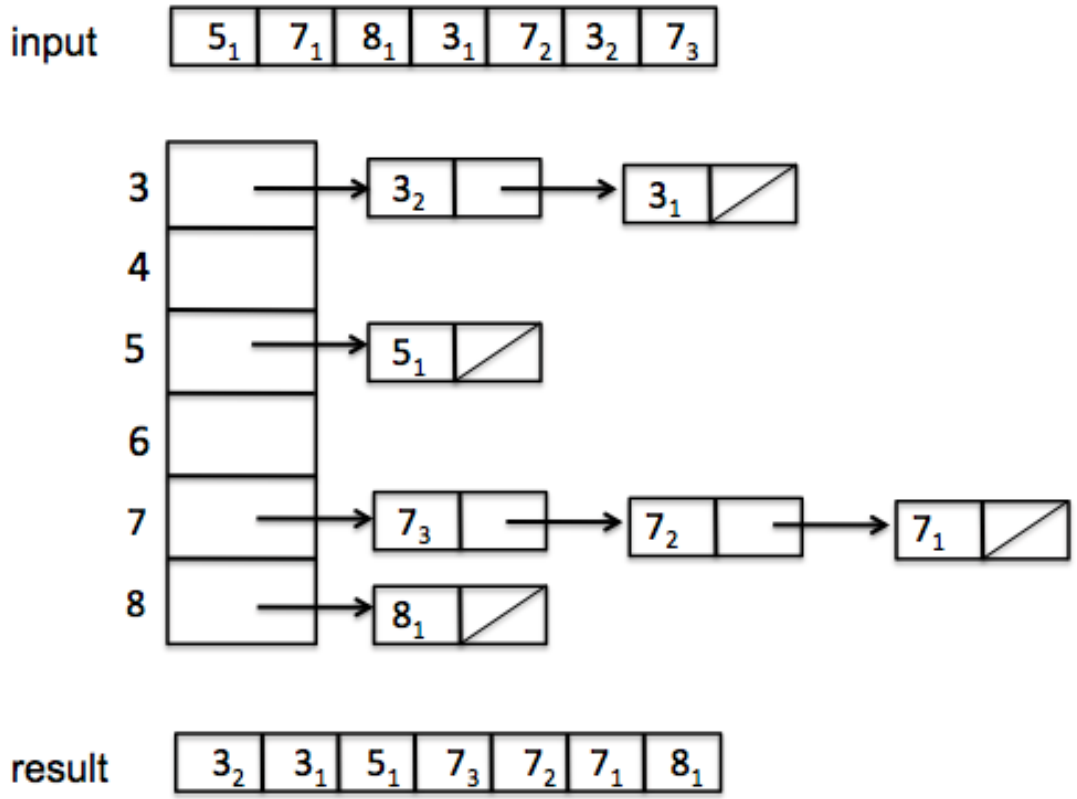
for $i \leftarrow 1$ *to* n
    insert $A[i]$ *in to* $B[MSB[A[i]]]$

for $i \leftarrow 1$ *to* $n-1$
    Sort B[i] with insertion sort or some other method

Concate the sublists B[0], B[1], B[2]....B[n-1] togather in order

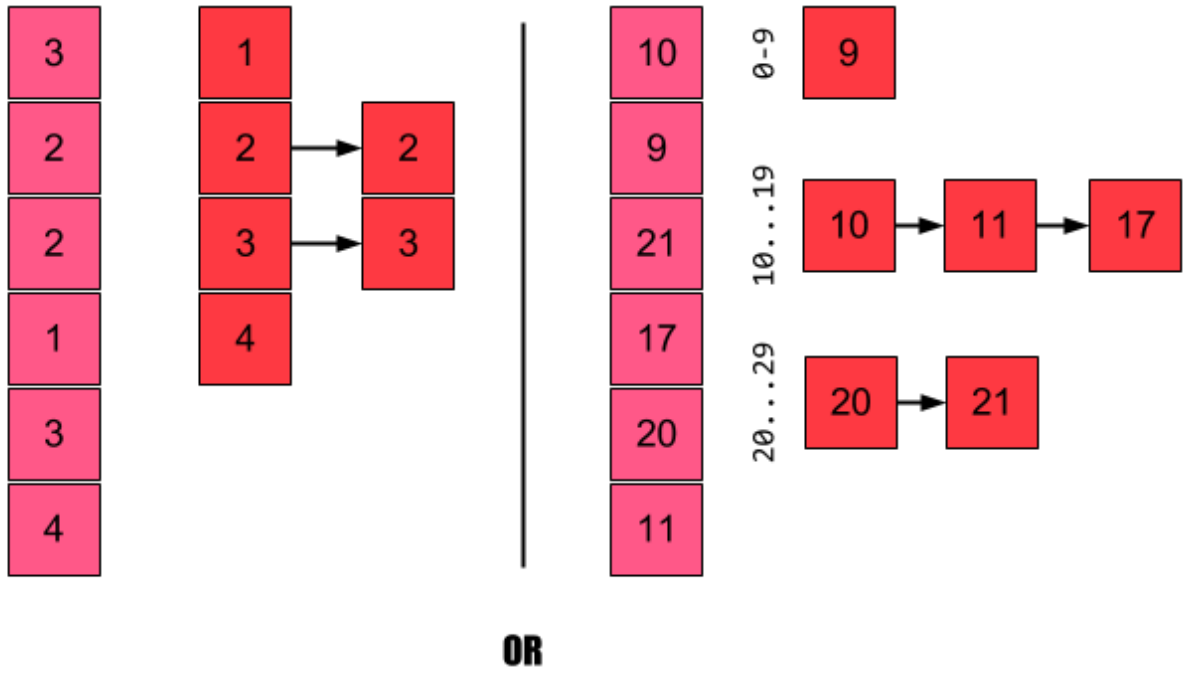
Simulation

Single bucket for **Each Value**



Simulation

Single bucket for **Range of Value**



Complexity Analysis

- The variable bucket size of bucket sort allows it to use O(n) memory instead of O(M), where M is number of distinct values; in exchange, it gives up counting sort's O(n + M) worst-case behavior.
- its worst-case performance is $O(n^2)$ which makes it as slow as bubble sort.

Properties

- Common optimization is to put the unsorted elements of the buckets back in the original array first, then run insertion sort
- Variant: Divides the value range into n buckets each of size M/n. It runs in linear time if each bucket is sorted using insertion sort
- However, the performance of this sort degrades with clustering
- Generalization of Counting sort
- As Radix sort and Counting sort, bucket sort depends so much on the input.
- Number of buckets can dramatically improve or worse the performance of the algorithm.
- Bucket sort is ideal in cases when we know in advance that the input is well dispersed.

Properties

- Bucket sort is cousin of Radix sort
- Works good when data is uniformly distributed
- Bucket sort with two buckets is effectively a version of quicksort
- Pivot value is always selected to be the middle value of the value range.
- This choice is effective for uniformly distributed inputs

Comparison

Algorithm	Stable	Inplace	Space	Best	Average	Worst
Selection Sort	Yes	Yes	$\theta(1)$	n^2	n^2	n^2
Bubble Sort	Yes	Yes	$\theta(1)$	n^2	n^2	n^2
Insertion Sort	Yes	Yes	$\theta(1)$	n	n^2	n^2
Merge Sort	Yes	No	$\theta(n)$	$nlog_2 n$	$nlog_2 n$	$nlog_2 n$
Quick Sort	No	Yes	$\theta(logn)$	$nlog_2 n$	$nlog_2 n$	n^2
Heap Sort	No	Yes	$\theta(1)$	$nlog_2 n$	$nlog_2 n$	$nlog_2 n$
Counting Sort	Yes	No	$\theta(n)$	n	n	n
Bucket Sort	Yes	No	$\theta(n)$	n	n	n
Radix Sort	Yes	Yes	$\theta(1)$	$d(n+m)$	$d(n+m)$	$d(n+m)$

Author's Profile

Academics: Prof. Mahesh Goyani has completed his graduation in Computer Engineering from SCET, VNSGU, Surat in 2005 with distinction. He received his Master Degree in field of Computer Engineering with 9.38 CPI (81.03 Perc.) from BVM College, SPU, Anand in 2009. He has secured 1st rank twice in university during his master degree. His area of interest is Image Processing, Computer Algorithms and Artificial Intelligence.

Publication: He has published many research papers in national and international journals and conferences. He was invited as a SESSION CHAIR in International Conference on Engineering, Science and Information Technology, Tirunelveli, Tamilnadu, Sept - 2011.

Membership: He is the member of technical review committee of International Journal of Computer Science and Issues (IJCSE, Mauritius), Electronics and Telecommunication Research Institute (ETRI, Korea), International journal of Engineering and Technology (IJET, Singapore), International journal of Computer Science and Information Security (IJCSIS, Pittsburg, USA). He has worked as a program committee member and reviewer in many International Conferences. He is also a member of ISTE technical society.

Website: `www.maheshgoyani.in`