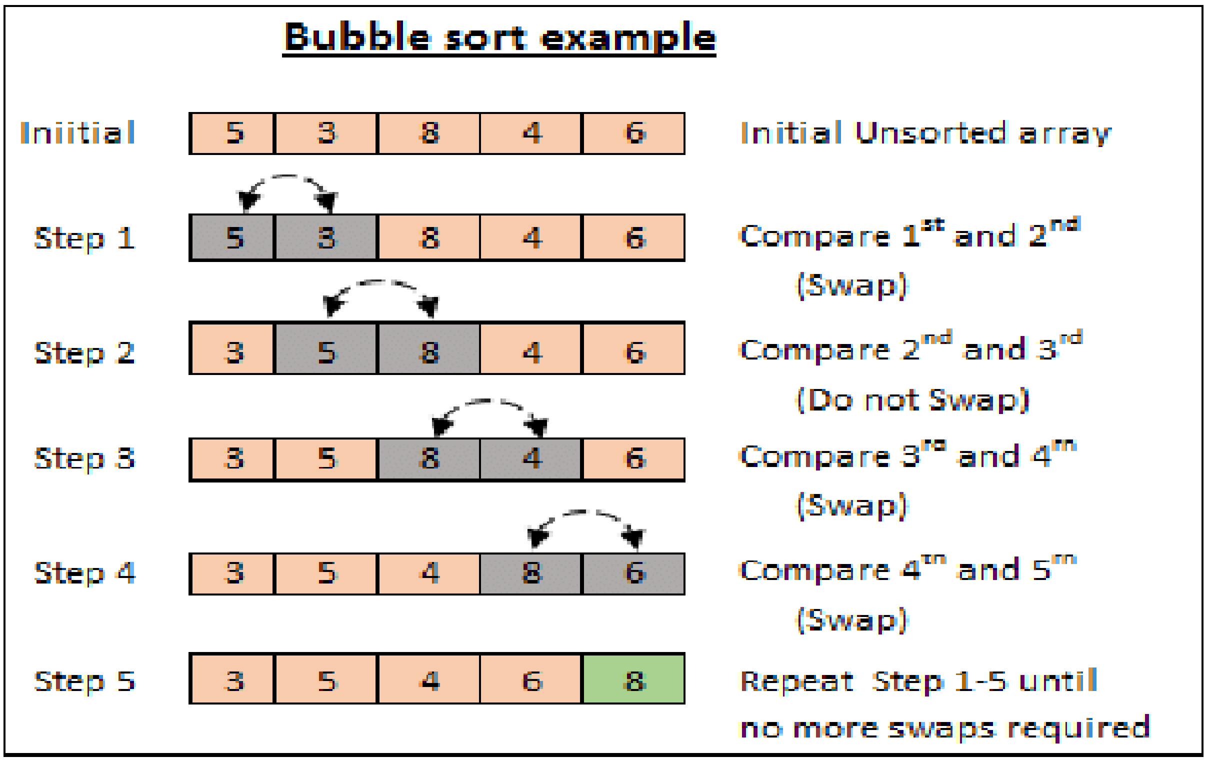
Bubble Sort



Algorithm

```
for (int pass=1 ; pass<=n-1 ; ++pass)
{
flag=0 // flag denotes are there any swaps done in pass
for (int i=0 ; i<=n-2 ; ++i)
{
if(A[i] > A[i+1])
{
swap(i,i+1,A);
flag=1 // After swap, set flag to 1
}
}
if(flag == 0) break; // No swaps indicates we can terminate loop
}
void swap(int x, int y, int[] A)
{
int temp = A[x];
A[x] = A[y];
A[y] = temp;
return;
}
```

Time Complexity Analysis-

- Bubble sort uses two loops- inner loop and outer loop.
- The inner loop deterministically performs O(n) comparisons.

Worst Case-

- In worst case, the outer loop runs O(n) times.
- Hence, the worst case time complexity of bubble sort is $O(n \times n) = O(n^2)$.

Best Case-

- In best case, the array is already sorted but still to check, bubble sort performs O(n) comparisons.
- Hence, the best case time complexity of bubble sort is O(n).

Average Case-

- In average case, bubble sort may require (n/2) passes and O(n) comparisons for each pass.
- Hence, the average case time complexity of bubble sort is $O(n/2 \times n) = \Theta(n^2)$.

The following table summarizes the time complexities of bubble sort in each case-

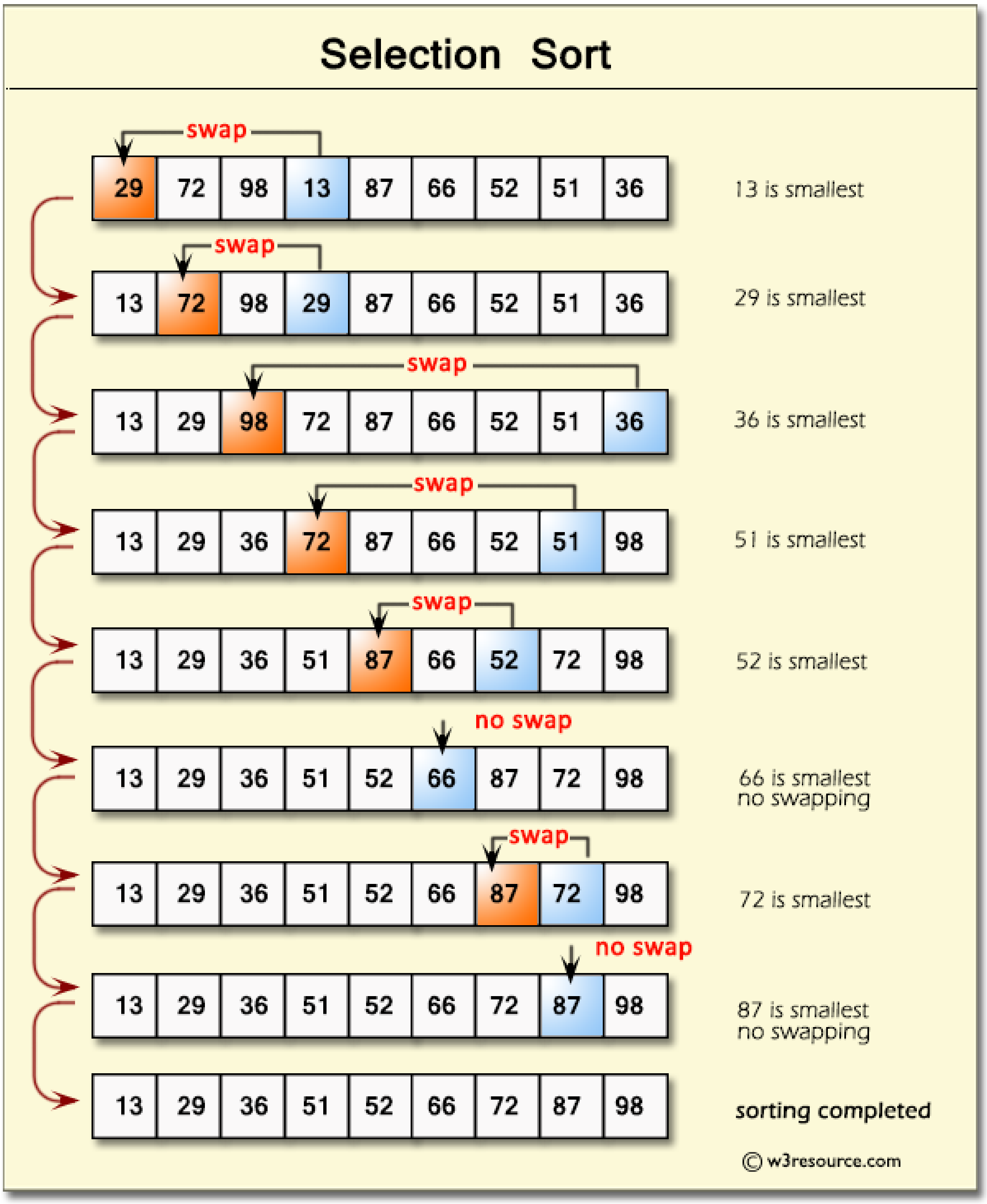
	Time Complexity
Best Case	**O(n)**
Average Case	**$\Theta(n^2)$**
Worst Case	**$O(n^2)$**

From here, it is clear that bubble sort is not at all efficient in terms of time complexity of its algorithm.

Space Complexity Analysis-

- Bubble sort uses only a constant amount of extra space for variables like flag, i, n.
- Hence, the space complexity of bubble sort is O(1).
- It is an in-place sorting algorithm i.e. it modifies elements of the original array to sort the given array

Selection Sort



1. SELECTION SORT(arr, n)

```

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT

SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
     SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

Time Complexity Analysis-

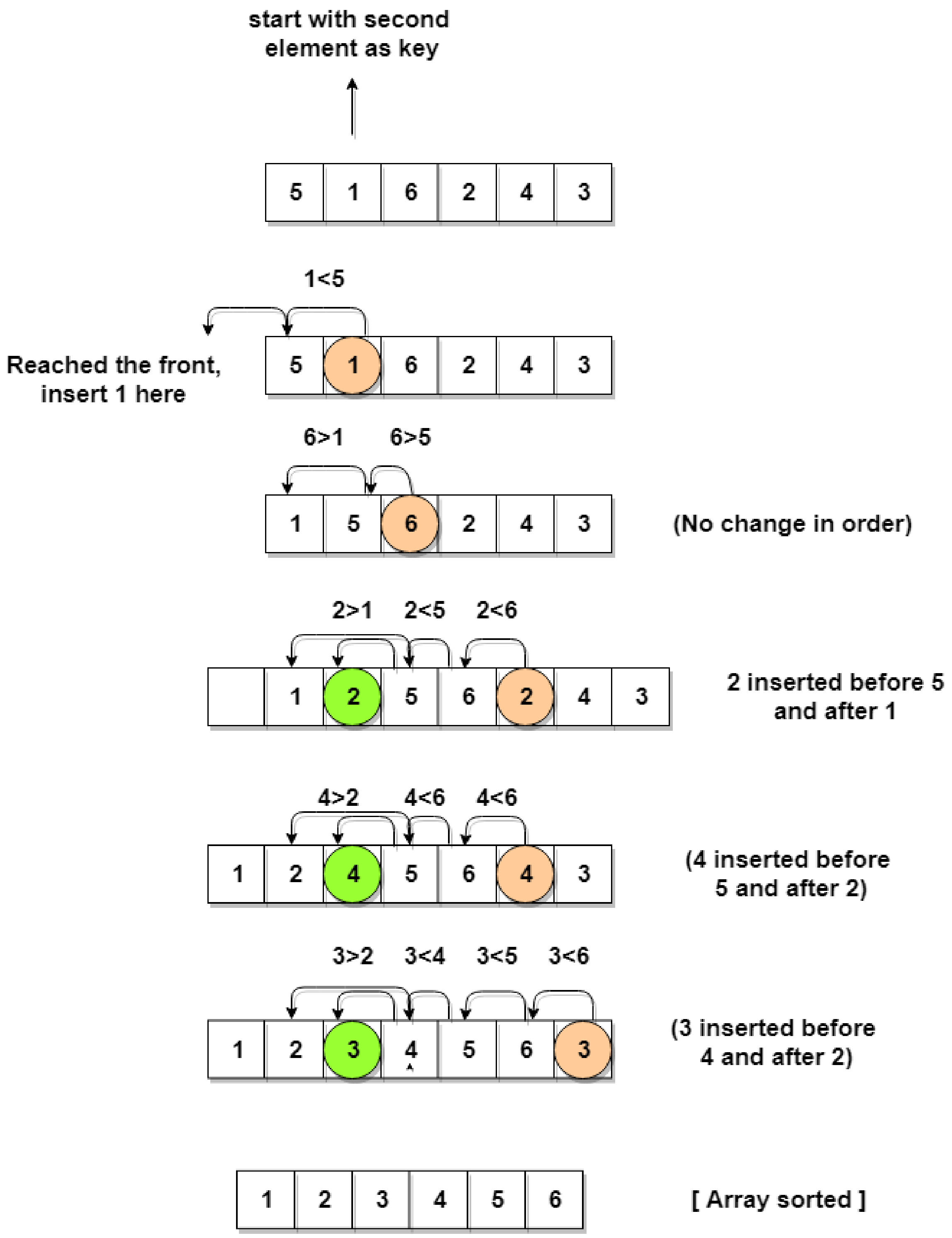
- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has O(n^2) time complexity.

	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2

Space Complexity Analysis-

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be O(1).

Insertion Sort



Algorithm

```
for (i = 1 ; i < n ; i++)
{
key = A [ i ];
j = i - 1;
while(j > 0 && A [ j ] > key)
{
A [ j+1 ] = A [ j ];
j--;
}
A [ j+1 ] = key;
}
```

Time Complexity Analysis-

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has O(n^2) time complexity.

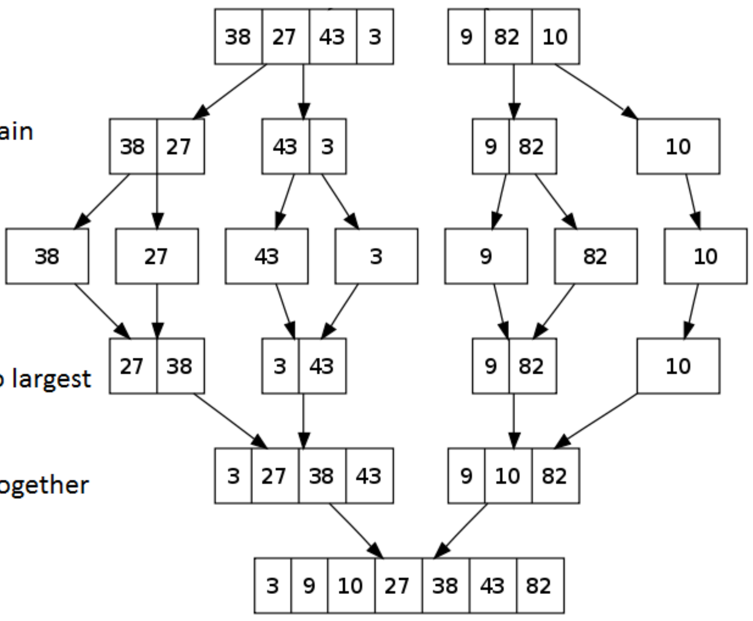
	Time Complexity
Best Case	n
Average Case	n^2
Worst Case	n^2

Space Complexity Analysis-

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be O(1).

Merge Sort

1. Divide the array into two parts
2. Divide the array into two parts again
3. Break each element into single parts
4. Sort the elements from smallest to largest
5. Merge the divided sorted arrays together
6. The array has been sorted



Algorithm

```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

```
/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];

    i = 0, /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */

    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {

        a[k] = RightArray[j];
        j++;
        k++;
    }
}
```

Time Complexity Analysis of Merge Sort

Now, let us follow up with the steps. our very own first step was to divide the input into two halves which comprised us of a logarithmic time complexity ie. log(N) where N is the number of elements.

our second step was to merge back the array into a single array, so if we observe it in all the number of elements to be merged N, and to merge back we use a simple loop which runs over all the N elements giving a time complexity of O(N).
finally, total time complexity will be - step -1 + step-2

General analysis
T(N) = Time Complexity for problem size N
T(n) = Θ(1) + 2T(n/2) + Θ(n) + Θ(1)
T(n) = 2T(n/2) + Θ(n)

Let us analyze this step by step:
T(n) = 2 * T(n/2) + 0(n)

STEP-1 Is to divide the array into two parts of equal size .
2 * T(n/2) --> Part 1
STEP-2 Now to merge baiscall traverse through all the elements.
constant * n --> Part 2
STEP-3 --> COMBINE 1 + 2
T(n) = 2 * T(n/2) + constant * n --> Part 3
Now we can further divide the array into two halfs if size of the partition arrays are greater than 1. So,

n/2/2--> n/4
T(N) = 2 * (2 * T(n/4) + constant * n/2) + constant * n
T(N) = 4 * T(n/4) + 2 * constant * n

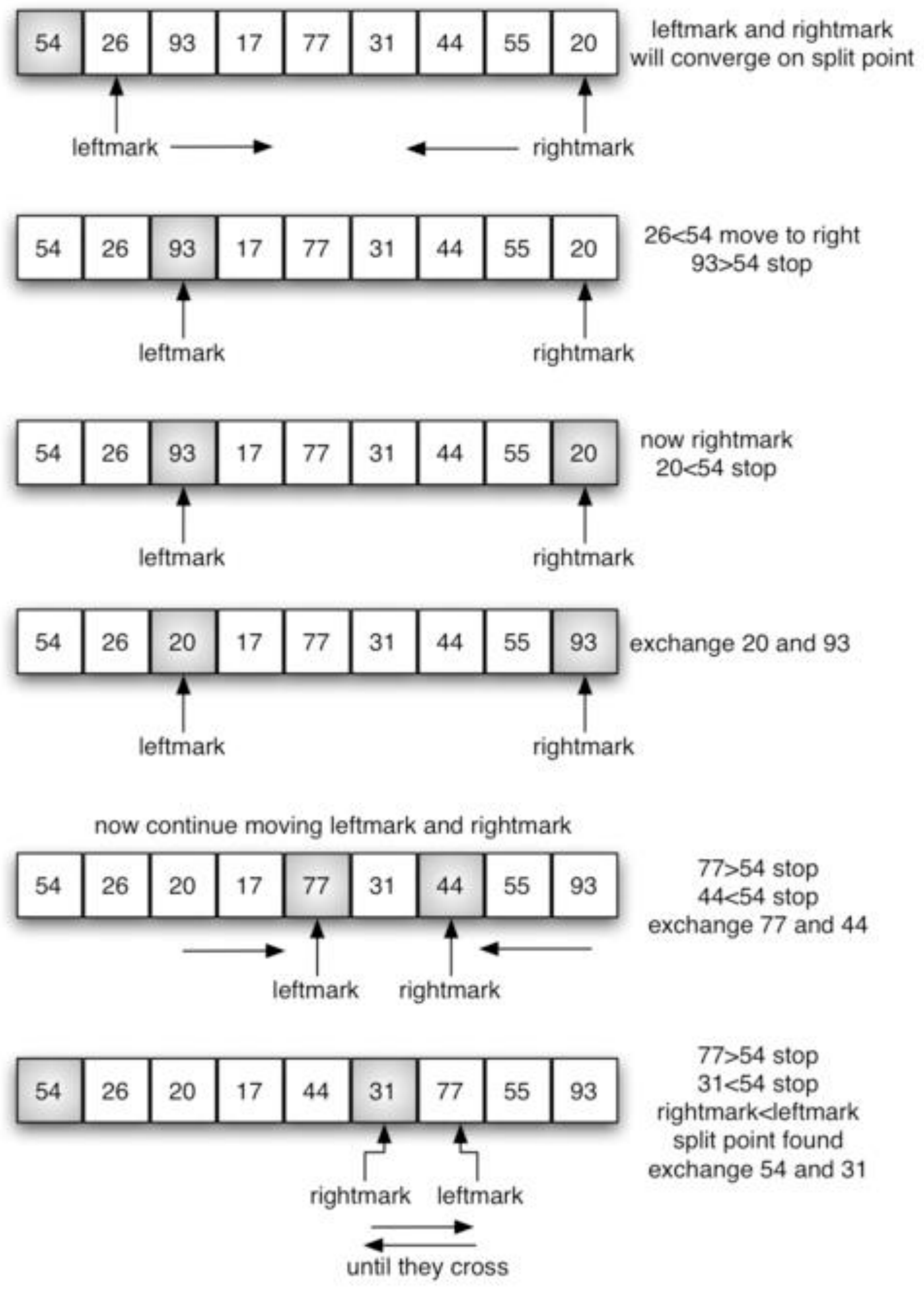
For this we can say that:
Where n can be subtituted to 2^k and the value of k is logN
T(n) = 2^k * T(n/(2^k)) + k * constant * n

Hence,
T(N) = N * T(1) + N * logN
= O(N * log(N))

CASE	TIME COMPLEXITY	# COMPARISONS
Worst Case	O(N logN)	N logN
Average Case	O(N logN)	0.74 NlogN
Best Case	O(N logN)	0.50 NlogN

- Space Complexity: O(N)

Quick Sort



After that partition

P-1: 31,26,20,17,44

Then 54

p-2: 77,55,93

and do as above.

Algorthm:

- a = Linear Array in memory
- beg = Lower bound of the sub array in question
- end = Upper bound of the sub array in question

```
Partition_Array (a , beg , end , loc)
Begin
Set left = beg , right = end , loc = beg
Set done = false
While (not done) do
While ( (a[loc] <= a[right] ) and (loc ≠ right) ) do
Set right = right - 1
end while
if (loc = right) then
Set done = true
else if (a[loc] > a[right]) then
Interchange a[loc] and a[right]
Set loc = right
end if
if (not done) then
While ( (a[loc] >= a[left] ) and (loc ≠ left) ) do
Set left = left + 1
end while
if (loc = left) then
Set done = true
else if (a[loc] < a[left]) then
Interchange a[loc] and a[left]
Set loc = left
end if
end if
end while
End
```

Quick Sort Analysis-

- To find the location of an element that splits the array into two parts, O(n) operations are required.
- This is because every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.

- If the array is split approximately in half (which is not usually), then there will be $\log_2 n$ splits.
- Therefore, total comparisons required are $f(n) = n \times \log_2 n = O(n\log_2 n)$.

Best and Average Case:

Order of Quick Sort = $O(n\log_2 n)$

Worst Case-

- Quick Sort is sensitive to the order of input data.
- It gives the worst performance when elements are already in the ascending order.
- It then divides the array into sections of 1 and (n-1) elements in each call.
- Then, there are (n-1) divisions in all.
- Therefore, here total comparisons required are $f(n) = n \times (n-1) = O(n^2)$.

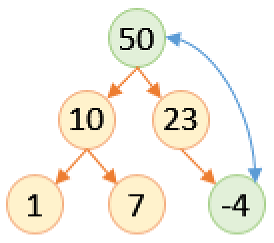
Order of Quick Sort in worst case = $O(n^2)$

Space Complexity

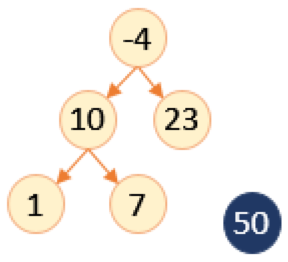
- O(N)
- as we are not creating any container other then given array therefore Space complexity will be in order of N

Heap Sort

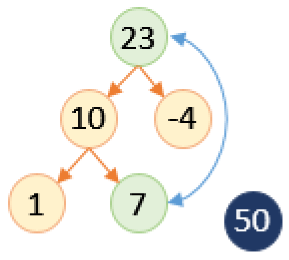
Step 1: Initial Max Heap



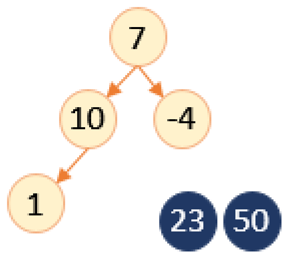
Step 2



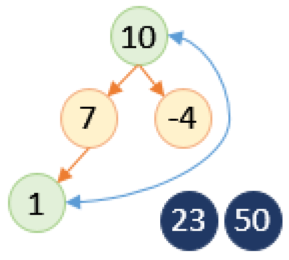
Step 3: Max Heap



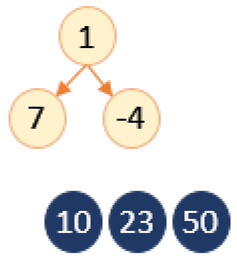
Step 4



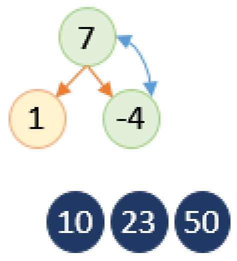
Step 5: Max Heap



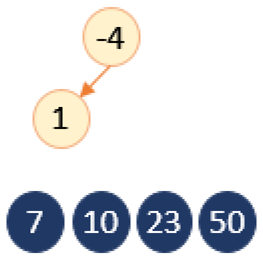
Step 6



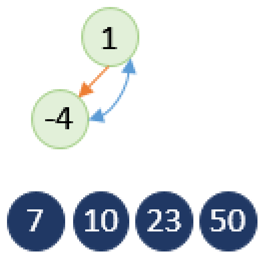
Step 7: Max Heap



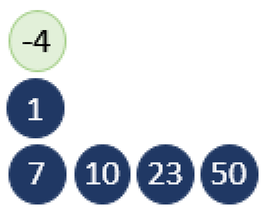
Step 8



Step 9: Max Heap



Step 10



Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
End
```

BuildMaxHeap(arr)

```
BuildMaxHeap(arr)
    heap_size(arr) = length(arr)
    for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

MaxHeapify(arr,i)

```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

Worst Case Time Complexity of Heap Sort

The worst case for heap sort might happen when all elements in the list are distinct. Therefore, we would need to call `max-heapify` every time we remove an element. In such a case, considering there are 'n' number of nodes-

- The number of swaps to remove every element would be log(n), as that is the max height of the heap
- Considering we do this for every node, the total number of moves would be n * (log(n)).

Therefore, the runtime in the worst case will be O(n(log(n)).

Best Case Time Complexity of Heap Sort

The best case for heapsort would happen when all elements in the list to be sorted are identical. In such a case, for 'n' number of nodes-

- Removing each node from the heap would take only a constant runtime, O(1). There would be no need to bring any node down or bring max valued node up, as all items are identical.
- Since we do this for every node, the total number of moves would be n * O(1).

Therefore, the runtime in the best case would be O(n).

Average Case Time Complexity of Heap Sort

In terms of total complexity, we already know that we can create a heap in O(n) time and do insertion/removal of nodes in O(log(n)) time. In terms of average time, we need to take into account all possible inputs, distinct elements or otherwise. If the total number of nodes is 'n', in such a case, the `max-heapify` function would need to perform:

- log(n)/2 comparisons in the first iteration (since we only compare two values at a time to build max-heap)
- log(n-1)/2 in the second iteration
- log(n-2)/2 in the third iteration
- and so on

So mathematically, the total sum would turn out to be-

(log(n))/2 + (log(n-1))/2 + (log(n-2))/2 + (log(n-3))/2 + ...
Upon approximation, the final result would be
=1/2(log(n!))
=1/2(n∗log(n)−n+O(log(n)))

Considering the highest ordered term, the average runtime of `max-heapify` would then be O(n(log(n)).
Since we call this function for all nodes in the final `heapsort` function, the runtime would be (n * O(n(log(n))). Calculating the average, upon dividing by n, we'd get a final average runtime of O(n(log(n))

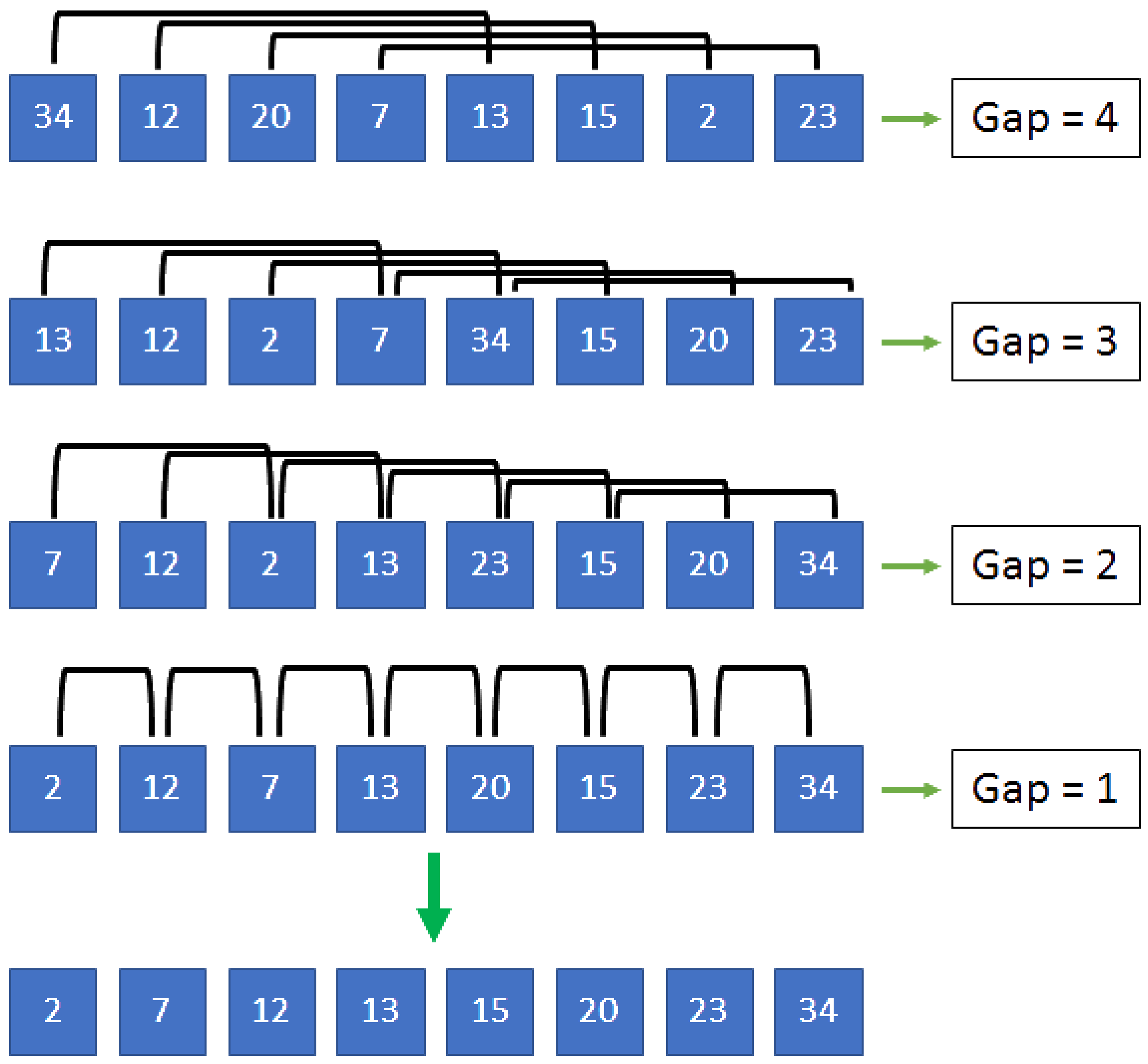
Space Complexity of Heap Sort

Since heapsort is an in-place designed sorting algorithm, the space requirement is constant and therefore, O(1). This is because, in case of any input-

- We arrange all the list items in place using a heap structure
- We put the removed item at the end of the same list after removing the max node from the max-heap.

Therefore, we don't use any extra space when implementing this algorithm. This gives the algorithm a space complexity of O(1).

Shell Sort



Algorithm:

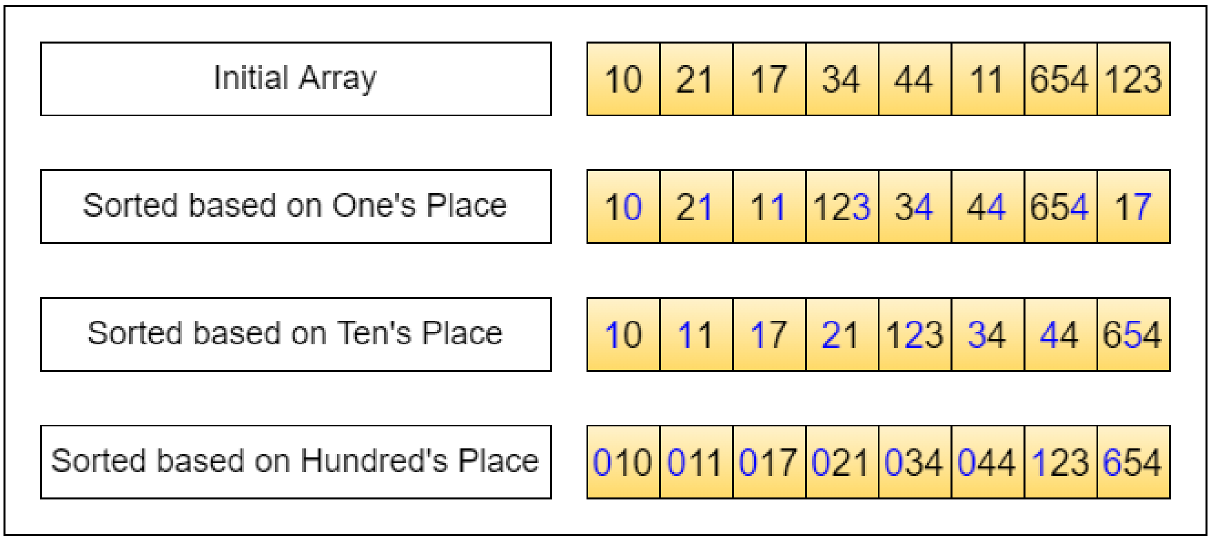
```
1. ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
2. for (interval = n/2; interval > 0; interval /= 2)
3. for ( i = interval; i < n; i += 1)
4. temp = a[i];
5. for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
6. a[j] = a[j - interval];
7. a[j] = temp;
8. End ShellSort
```

Complexity

- Worst case time complexity: **Θ(N (log N)^2)** comparisons
- Average case time complexity: **Θ(N (log N)^2)** comparisons

- Best case time complexity: **Θ(N log N)**
- Space complexity: **Θ(1)**.

Radix Sort



Initial Array	10	21	17	34	44	11	654	123
Sorted based on One's Place	10	21	11	123	34	44	654	17
Sorted based on Ten's Place	10	11	17	21	123	34	44	654
Sorted based on Hundred's Place	010	011	017	021	034	044	123	654

Algorithm:

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. **for** i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digit
   s at
7. the ith place

Time Complexity

Case	Time Complexity
Best Case	Ω(n+k)

Average Case	θ(nk)
Worst Case	O(nk)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is **Ω(n+k)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is **θ(nk)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is **O(nk)**.

. Space Complexity

Space Complexity	O(n + k)
Stable	YES

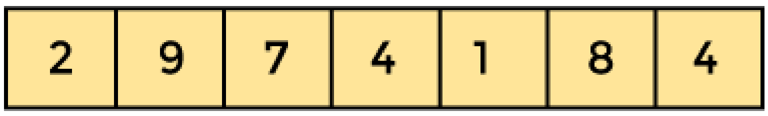
Counting Sort

Working of counting sort Algorithm

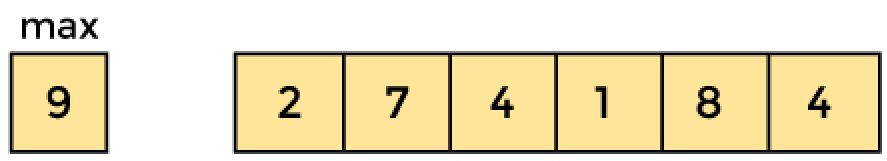
Now, let's see the working of the counting sort Algorithm.

To understand the working of the counting sort algorithm, let's take an unsorted array. It will be easier to understand the counting sort via an example.

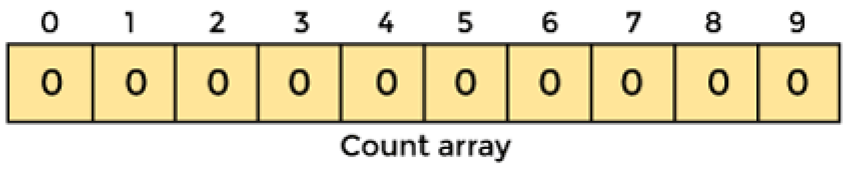
Let the elements of array are -



1. Find the maximum element from the given array. Let **max** be the maximum element.

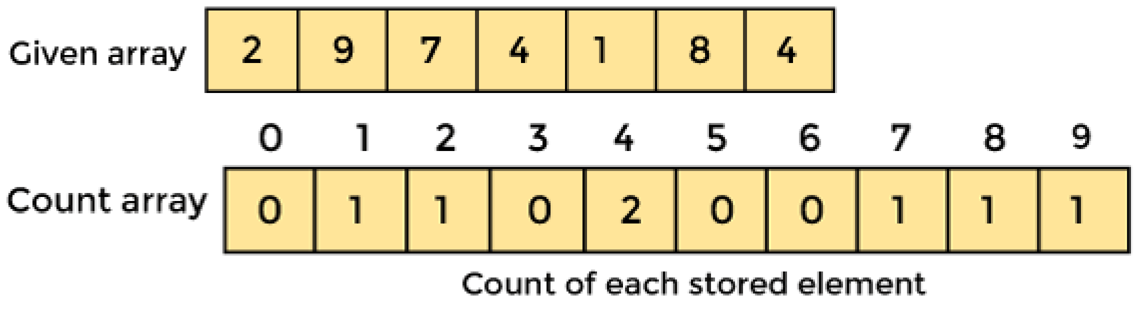


2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.



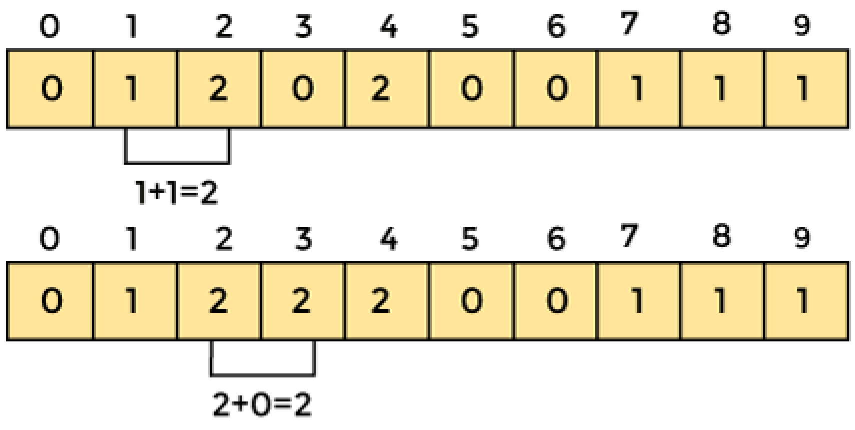
3. Now, we have to store the count of each array element at their corresponding index in the count array.

The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.

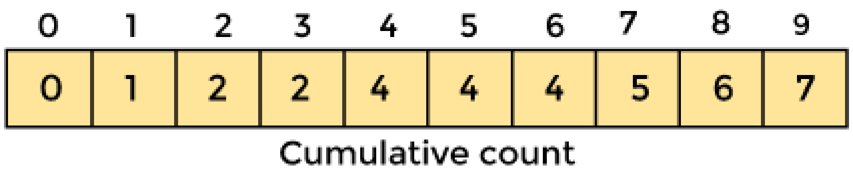


Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.

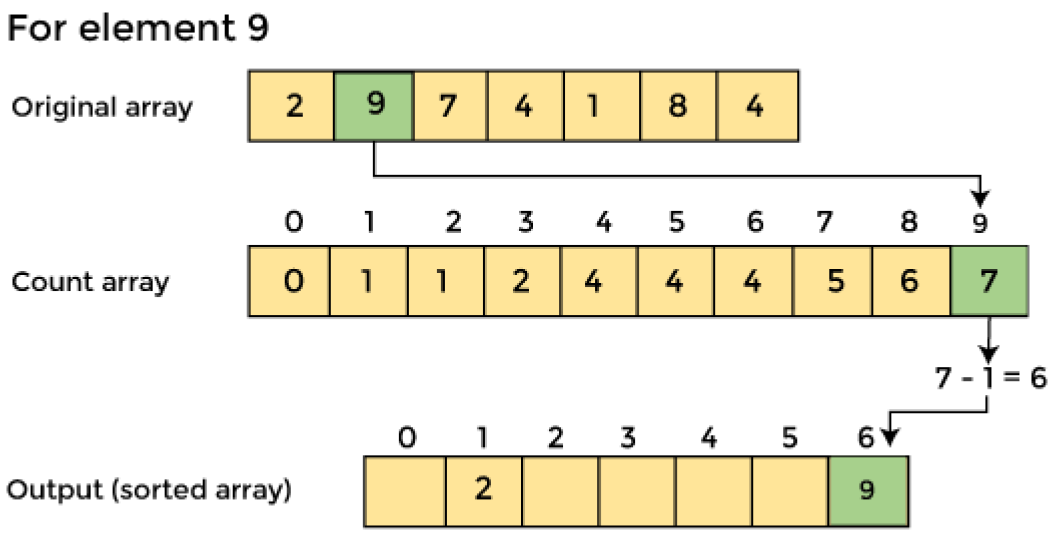
index of the sorted array.



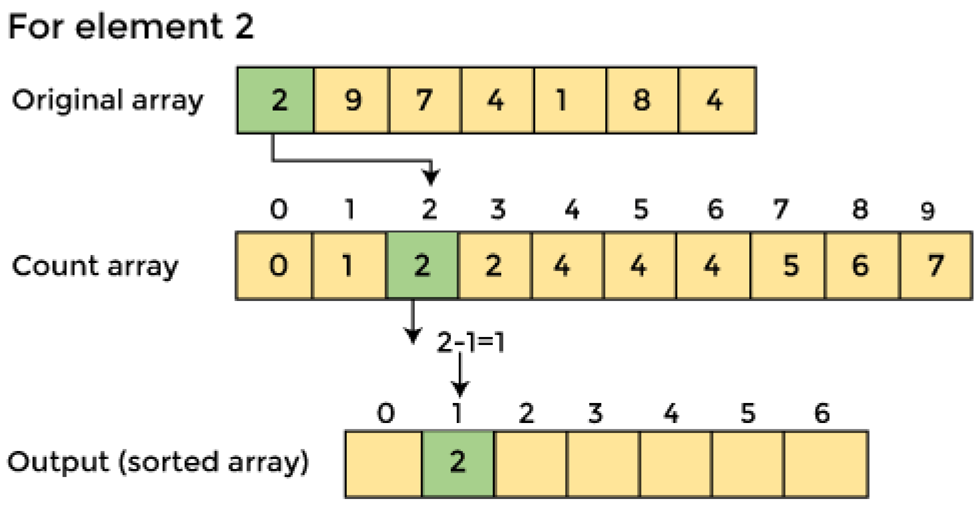
Similarly, the cumulative count of the count array is -



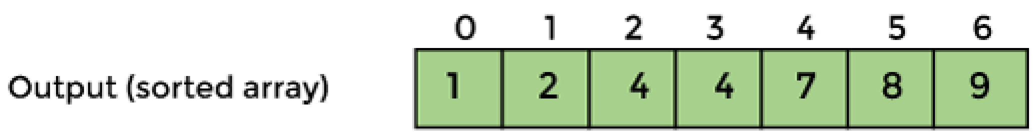
4. Now, find the index of each element of the original array



After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.



Similarly, after sorting, the array elements are -



Now, the array is completely sorted.

Algorithm:

```
countingSort(array, n) // 'n' is the size of array
max = find maximum element in the given array
create count array with size maximum + 1
Initialize count array with all 0's
for i = 0 to n
find the count of every unique element and
store that count at ith position in the count array
for j = 1 to max
Now, find the cumulative sum and store it in count array
for i = n to 1
Restore the array elements
Decrease the count of every restored element by 1
end countingSort
```

Time Complexity

Case	Time	Complexity
Best Case	O(n + k)	
Average Case	O(n + k)	
Worst Case	O(n + k)	

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of counting sort is **O(n + k)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of counting sort is **O(n + k)**.

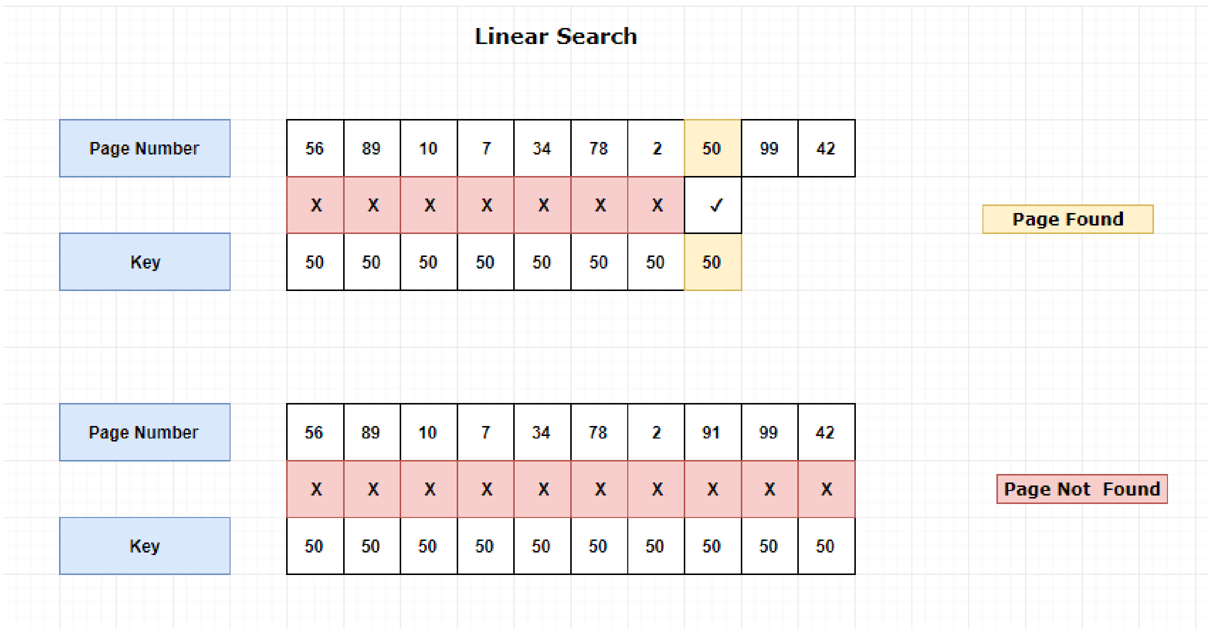
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of counting sort is **O(n + k)**.

2. Space Complexity

Space Complexity	O(max)
Stable	YES

- The space complexity of counting sort is O(max). The larger the range of elements, the larger the space complexity.

Linear Search



Algorithm:

1. Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
2. Step 1: set pos = -1
3. Step 2: set i = 1
4. Step 3: repeat step 4 while i <= n

```
5.  Step 4: if a[i] == val
6.  set pos = i
7.  print pos
8.  go to step 6
9.  [end of if]
10. set ii = i + 1
11. [end of loop]
12. Step 5: if pos = -1
13. print "value is not present in the array "
14. [end of if]
15. Step 6: exit
```

Analysis of Best Case Time Complexity of Linear Search

The Best Case will take place if:

- The element to be search is on the first index.

The number of comparisons in this case is 1. Thereforce, Best Case Time Complexity of Linear Search is O(1).

Analysis of Average Case Time Complexity of Linear Search

Let there be N distinct numbers: a1, a2, ..., a(N-1), aN

We need to find element P.

There are two cases:

- Case 1: The element P can be in N distinct indexes from 0 to N-1.
- Case 2: There will be a case when the element P is not present in the list.

There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

If element P is in index K, then Linear Search will do K+1 comparisons.

Number of comparisons for all cases in case 1 = Comparisons if element is in index 0 + Comparisons if element is in index 1 + ... + Comparisons if element is in index N-1
= 1 + 2 + ... + N
= N * (N+1) / 2 comparisons

If element P is not in the list, then Linear Search will do N comparisons.

Number of comparisons for all cases in case 2 = N

Therefore, total number of comparisons for all N+1 cases = N * (N+1) / 2 + N
= N * ((N+1)/2 + 1)

Average number of comparisons = (N * ((N+1)/2 + 1)) / (N+1)
= N/2 + N/(N+1)

The dominant term in "Average number of comparisons" is N/2. So, the Average Case Time Complexity of Linear Search is O(N).

Analysis of Worst Case Time Complexity of Linear Search

The worst case will take place if:

- The element to be search is in the last index
- The element to be search is not present in the list

In both cases, the maximum number of comparisons take place in Linear Search which is equal to N comparisons.

Hence, the Worst Case Time Complexity of Linear Search is O(N).

Number of Comparisons in Worst Case: N

Analysis of Space Complexity of Linear Search

In Linear Search, we are creating a boolean variable to store if the element to be searched is present or not.

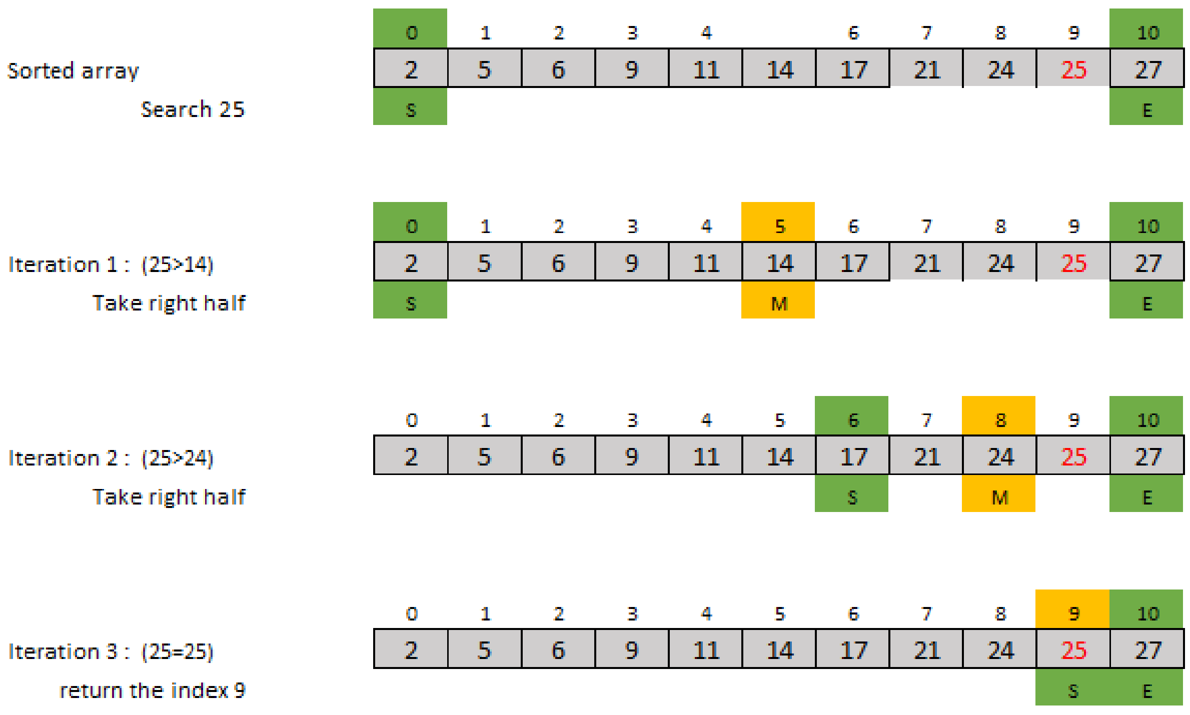
The variable is initialized to false and if the element is found, the variable is set to true. This variable can be used in other processes or returned by the function.

In Linear Search function, we can avoid using this boolean variable as well and return true or false directly.

The input to Linear Search involves:

- A list/ array of N elements
- A variable storing the element to be searched.
    - As the amount of extra data in Linear Search is fixed, the Space Complexity is O(1).
    - Therefore, Space Complexity of Linear Search is O(1).

Binary Search



Algorithm:

1. Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_boun d' is the index of the first array element, 'upper_bound' is the index of the last array el ement, 'val' is the value to search
2. Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

```
3. Step 2: repeat steps 3 and 4 while beg <=end
4. Step 3: set mid = (beg + end)/2
5. Step 4: if a[mid] = val
6. set pos = mid
7. print pos
8. go to step 6
9. else if a[mid] > val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit
```

Best Case Time Complexity of Binary Search

The best case of Binary Search occurs when:

- The element to be search is in the middle of the list

In this case, the element is found in the first step itself and this involves 1 comparison.

Therefore, Best Case Time Complexity of Binary Search is O(1).

Average Case Time Complexity of Binary Search

Let there be N distinct numbers: a1, a2, ..., a(N-1), aN

We need to find element P.

There are two cases:

Case 1: The element P can be in N distinct indexes from 0 to N-1.
Case 2: There will be a case when the element P is not present in the list.
There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

If element P is in index K, then Binary Search will do K+1 comparisons.

This is because:

The element at index N/2 can be found in 1 comparison as Binary Search starts from middle.

Similarly, in the 2nd comparisons, elements at index N/4 and 3N/4 are compared based on the result of 1st comparison.

On this line, in the 3rd comparison, elements at index N/8, 3N/8, 5N/8, 7N/8 are compared based on the result of 2nd comparison.

Based on this, we know that:

- Elements requiring 1 comparison: 1
- Elements requiring 2 comparisons: 2
- Elements requiring 3 comparisons: 4

Therefore, Elements requiring I comparisons: 2^(I-1)

The maximum number of comparisons = Number of times N is divided by 2 so that result is 1 = Comparisons to reach 1st element = logN comparisons

 can vary from 0 to logN

Total number of comparisons = 1 * (Elements requiring 1 comparison) + 2 * (Elements requiring 2 comparisons) + ... + logN * (Elements requiring logN comparisons)

Total number of comparisons = 1 * (1) + 2 * (2) + 3 * (4) + ... + logN * (2^(logN-1))

Total number of comparisons = 1 + 4 + 12 + 32 + ... = 2^logN * (logN - 1) + 1

Total number of comparisons = N * (logN - 1) + 1

Total number of cases = N+1

Therefore, average number of comparisons = (N * (logN - 1) + 1) / (N+1)

Average number of comparisons = N * logN / (N+1) - N/(N+1) + 1/(N+1)

The dominant term is N * logN / (N+1) which is approximately logN. Therefore, Average Case Time Complexity of Binary Search is O(logN).

Analysis of Worst Case Time Complexity of Binary Search

The worst case of Binary Search occurs when:

- The element is to search is in the first index or last index

In this case, the total number of comparisons required is logN comparisons.

Therefore, Worst Case Time Complexity of Binary Search is O(logN).

Analysis of Space Complexity of Binary Search

In an iterative implementation of Binary Search, the space complexity will be O(1).

This is because we need two variable to keep track of the range of elements that are to be checked. No other data is needed.

In a recursive implementation of Binary Search, the space complexity will be O(logN).

This is because in the worst case, there will be logN recursive calls and all these recursive calls will be stacked in memory. In fact, if I comparisons are needed, then I recursive calls will be stacked in memory and from our analysis of average case time complexity, we know that the average memory will be O(logN) as well.

Amortized Analysis:

In computer science, amortized analysis is **a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute**. The motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic.

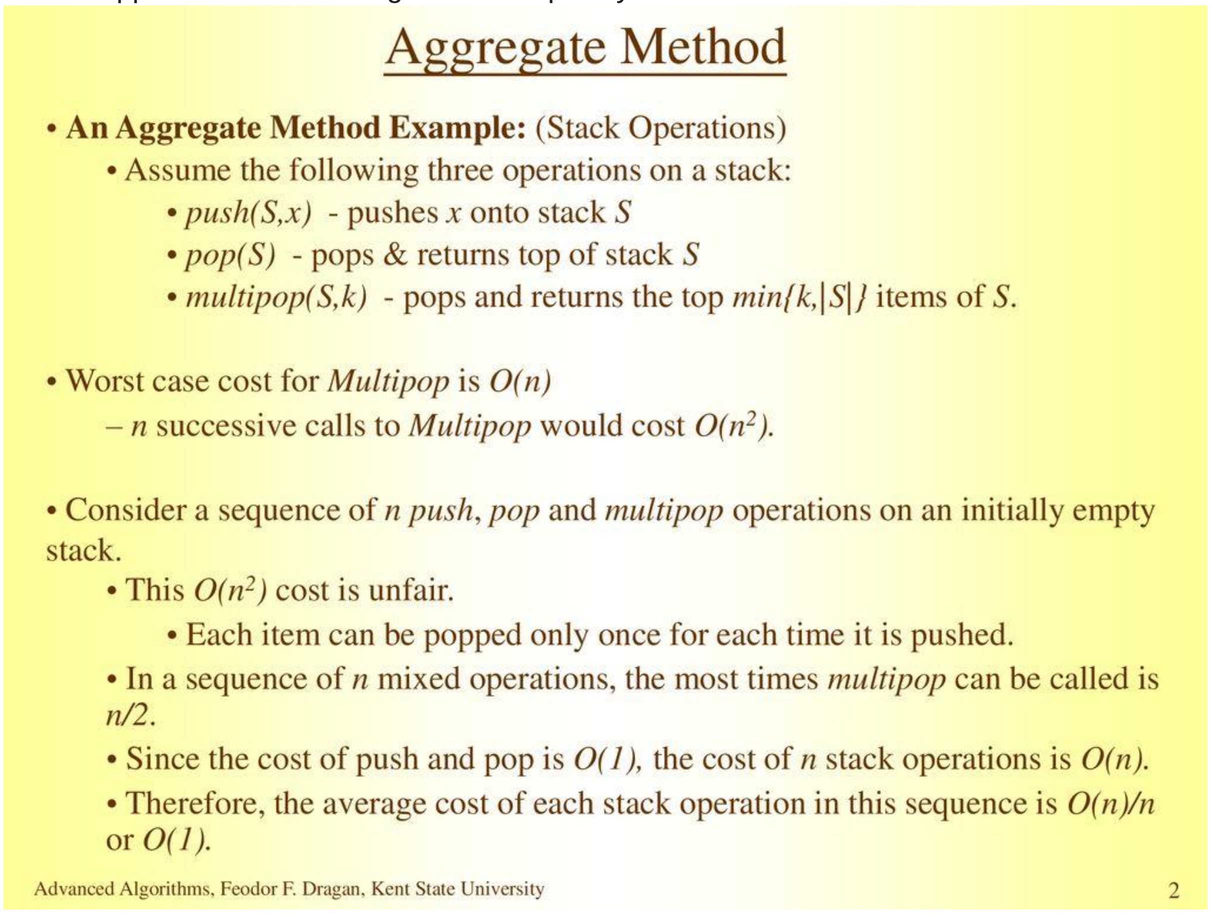
Example on Amortized Analysis

For a dynamic array, items can be inserted at a given index in O(1) time. But if that index is not present in the array, it fails to perform the task in constant time. For that case, it initially doubles the size of the array then inserts the element if the index is present.

Methods:

There are generally three methods for performing amortized analysis: **the aggregate method, the accounting method, and the potential method**. All of these give correct answers; the choice of which to use depends on which is most convenient for a particular situation.

1. The first method of amortized analysis is called the aggregate method and involves **counting out the complexity of each operation**. By expanding each case, one can try to determine a pattern and come up with an overall upper bound for the algorithm complexity.

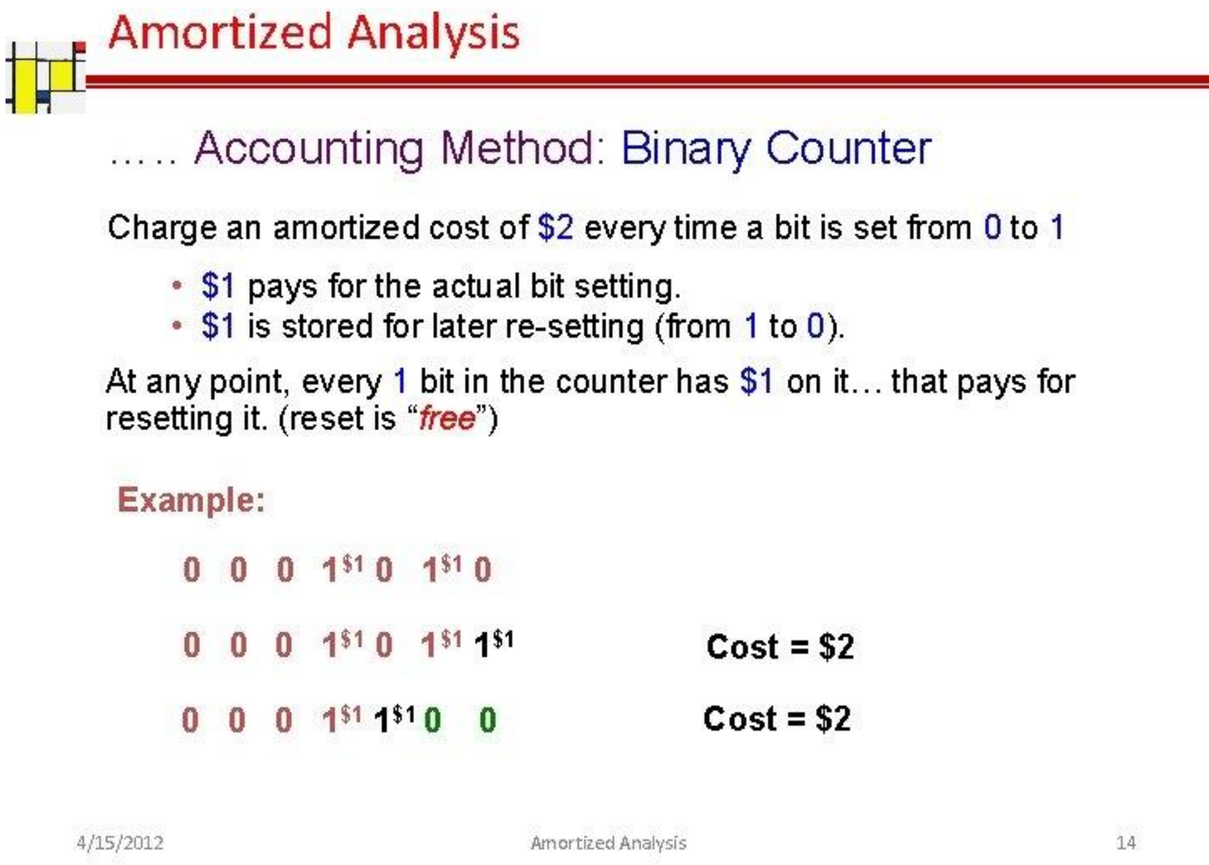


Aggregate Method

- **An Aggregate Method Example:** (Stack Operations)
  - Assume the following three operations on a stack:
    - *push(S,x)* - pushes *x* onto stack *S*
    - *pop(S)* - pops & returns top of stack *S*
    - *multipop(S,k)* - pops and returns the top *min{k,|S|}* items of *S*.
- Worst case cost for *Multipop* is $O(n)$
  - – *n* successive calls to *Multipop* would cost $O(n^2)$.
- Consider a sequence of *n push*, *pop* and *multipop* operations on an initially empty stack.
  - This $O(n^2)$ cost is unfair.
    - Each item can be popped only once for each time it is pushed.
  - In a sequence of *n* mixed operations, the most times *multipop* can be called is *n/2*.
  - Since the cost of push and pop is $O(1)$, the cost of *n* stack operations is $O(n)$.
  - Therefore, the average cost of each stack operation in this sequence is $O(n)/n$ or $O(1)$.

Advanced Algorithms, Feodor F. Dragan, Kent State University 2

2. In the field of analysis of algorithms in computer science, the accounting method is **a method of amortized analysis based on accounting**. The accounting method often gives a more intuitive account of the amortized cost of an operation than either aggregate analysis or the potential method.



Amortized Analysis

….. Accounting Method: Binary Counter

Charge an amortized cost of $2 every time a bit is set from 0 to 1

- $1 pays for the actual bit setting.
- $1 is stored for later re-setting (from 1 to 0).

At any point, every 1 bit in the counter has $1 on it… that pays for resetting it. (reset is "*free*")

Example:

```
0  0  0  1$1 0  1$1 0
0  0  0  1$1 0  1$1 1$1        Cost = $2
0  0  0  1$1 1$1 0   0         Cost = $2
```

4/15/2012 Amortized Analysis 14

3. According to computational complexity theory, the potential method is defined as **a method implemented to analyze the amortized time and space complexity of a data structure**, a measure of its performance over sequences of operations that eliminates the cost of infrequent but expensive operations

Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1)}{n}$$

We can simply the above series breaking terms 2, 3, 5, 9.. into two (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + ...)}^{[\text{Log}_2(n-1)]+1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

Amortized Cost = O(1)