Week 11

Sorting

(Merge Sort, Counting Sort, Radix Sort, Bucket Sort)

Merge Sort

Merge Sort

Merge sort is a *divide* and *conquer* based sorting algorithm.

Merge Sort

Merge sort is a *divide* and *conquer* based sorting algorithm.

Divide: Divide the input array into two almost equal halves (i.e., two sub arrays of almost equal size).

Merge Sort

Merge sort is a *divide* and *conquer* based sorting algorithm.

***Divide*: Divide the input array into two almost equal halves (i.e., two sub arrays of almost equal size).**

***Conquer*: Sort the smaller subarrays, and merge the two sorted subarrays to produce a combined sorted array.**

Merge Sort

Merge sort is a *divide* and *conquer* based sorting algorithm.

***Divide*: Divide the input array into two almost equal halves.**

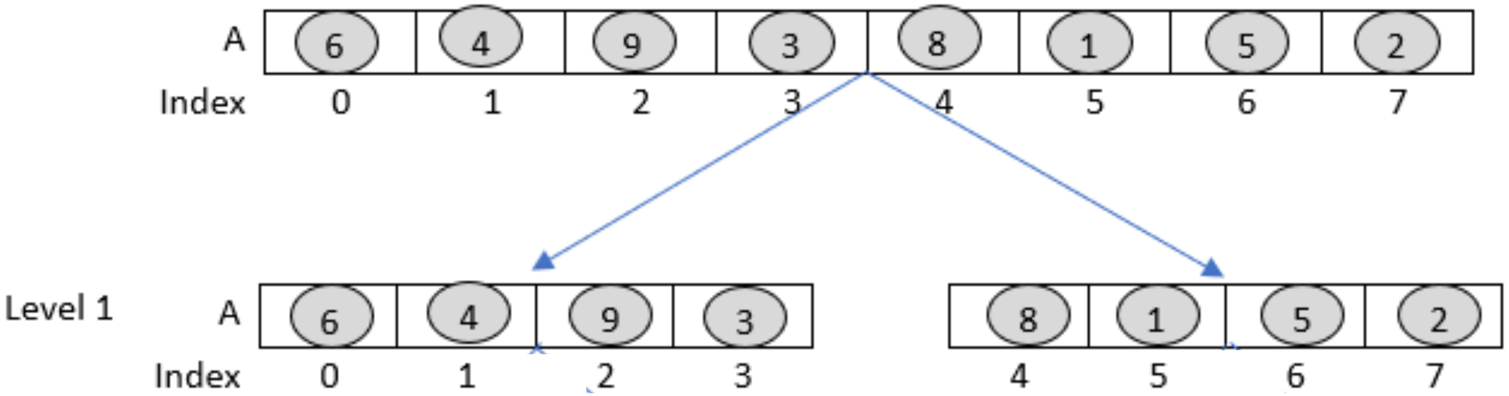
***Conquer*: Sort the smaller subarrays, and merge the two sorted subarrays to produce a combined sorted array.**

It is an external memory algorithm. That means, algorithm can process the data collection part by part

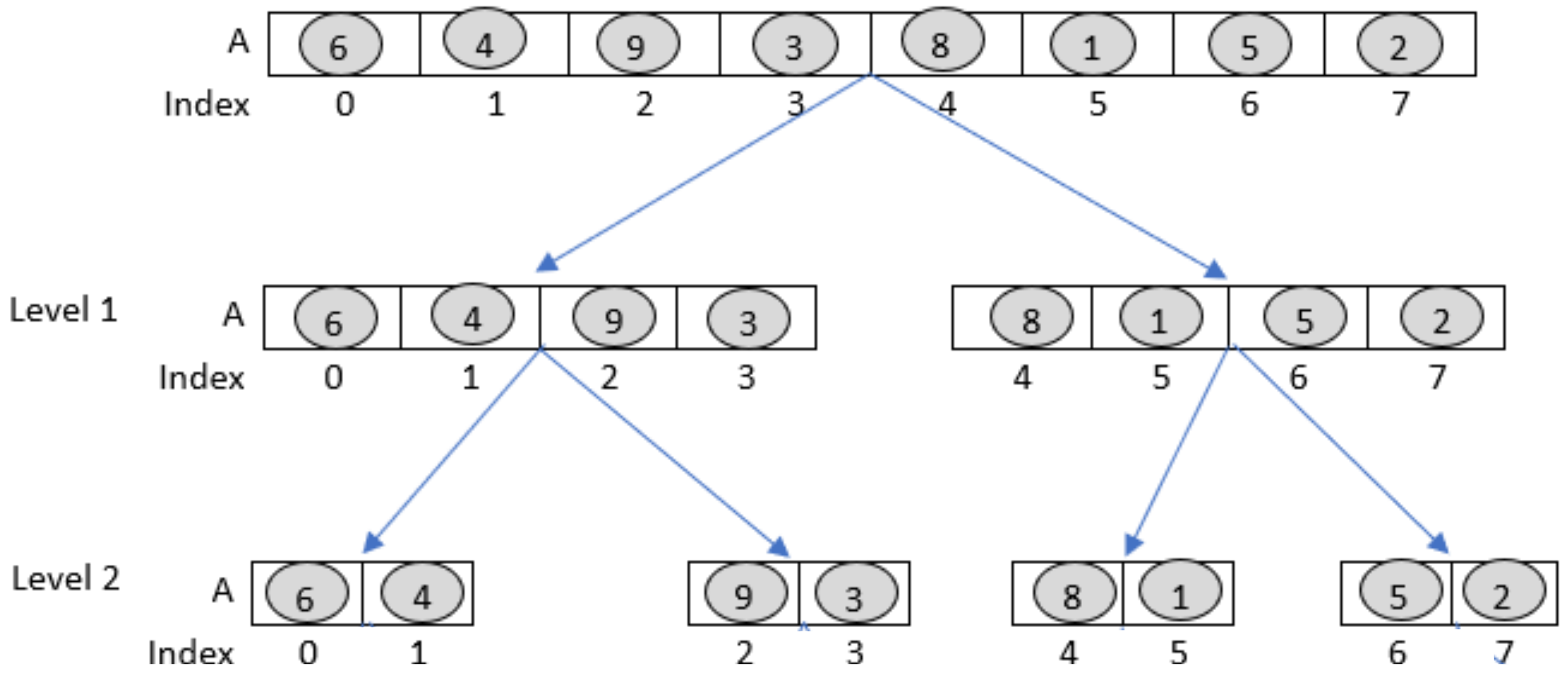
Merge Sort

A	6	4	9	3	8	1	5	2
Index	0	1	2	3	4	5	6	7

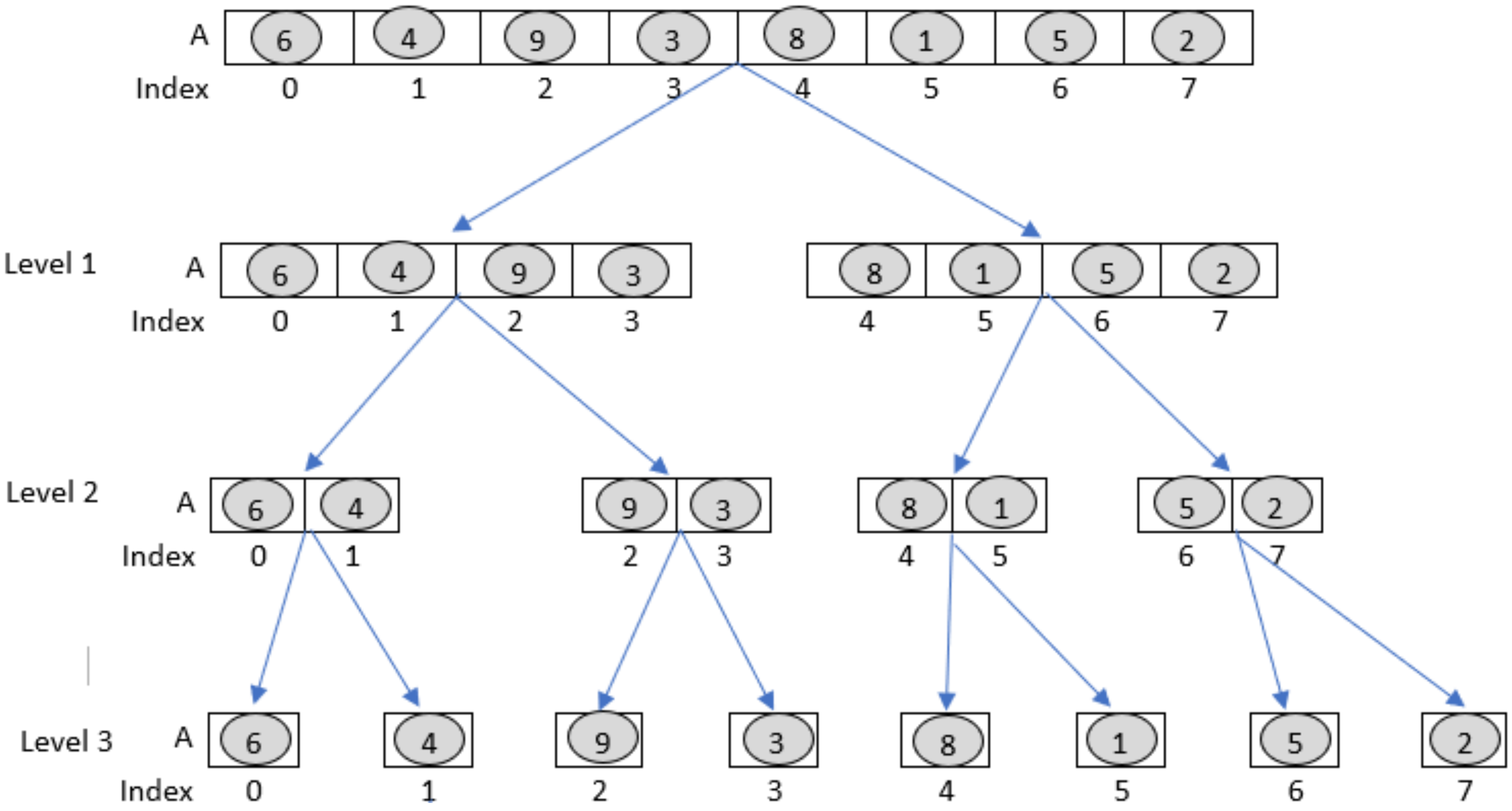
Merge Sort – Divide into smaller problems



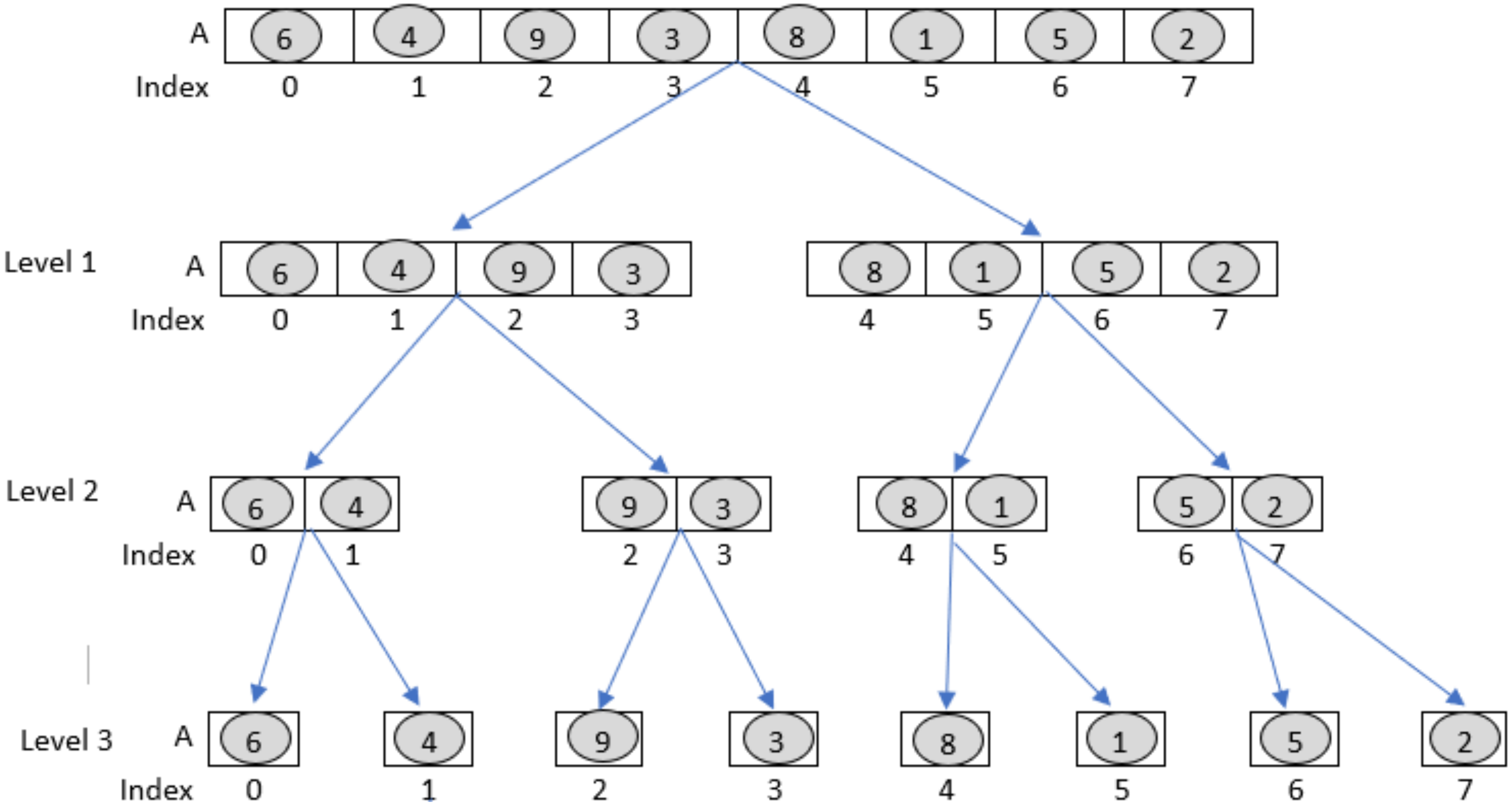
Merge Sort – Divide into smaller problems



Merge Sort – Divide into smaller problems



Merge Sort – Divide into smaller problems

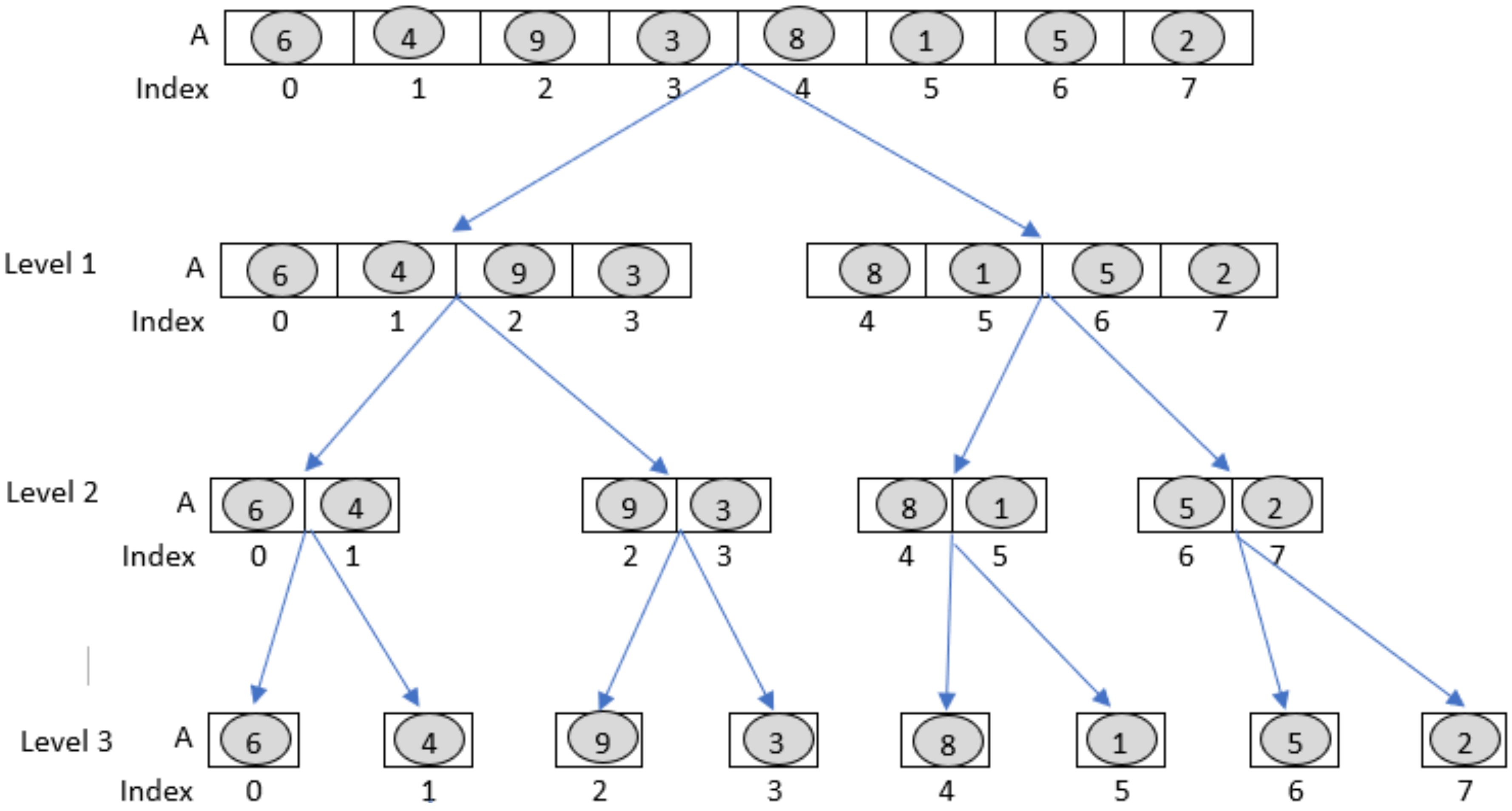


..................$\theta(2^0)$

..................$\theta(2^1)$

..................$\theta(2^2)$

Merge Sort – Divide into smaller problems



………………$\theta(2^0)$

………………$\theta(2^1)$

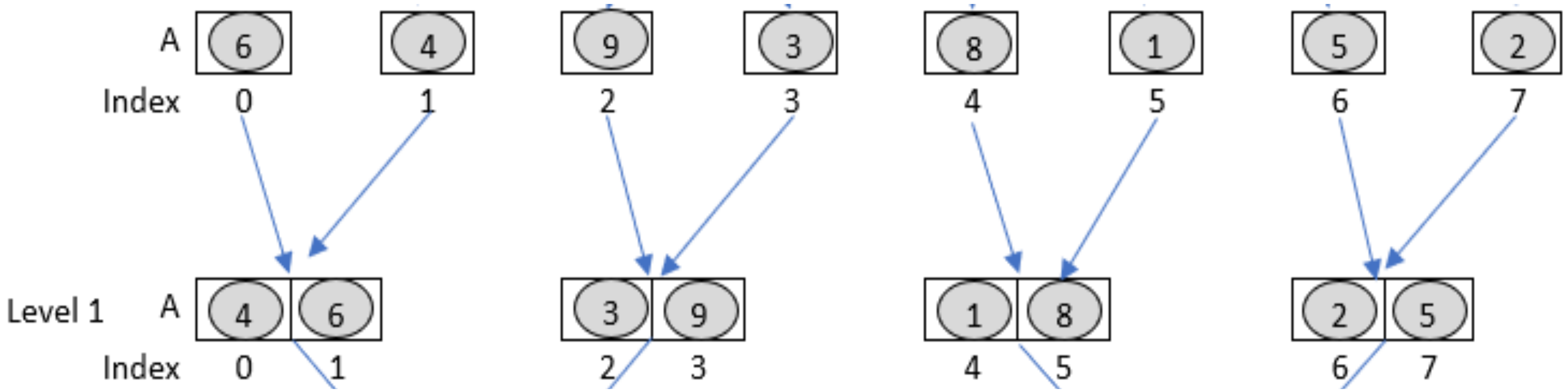
………………$\theta(2^2)$

Total: $2^0 + 2^1 + 2^2 + \cdots + 2^{l-1} = \theta(n)$

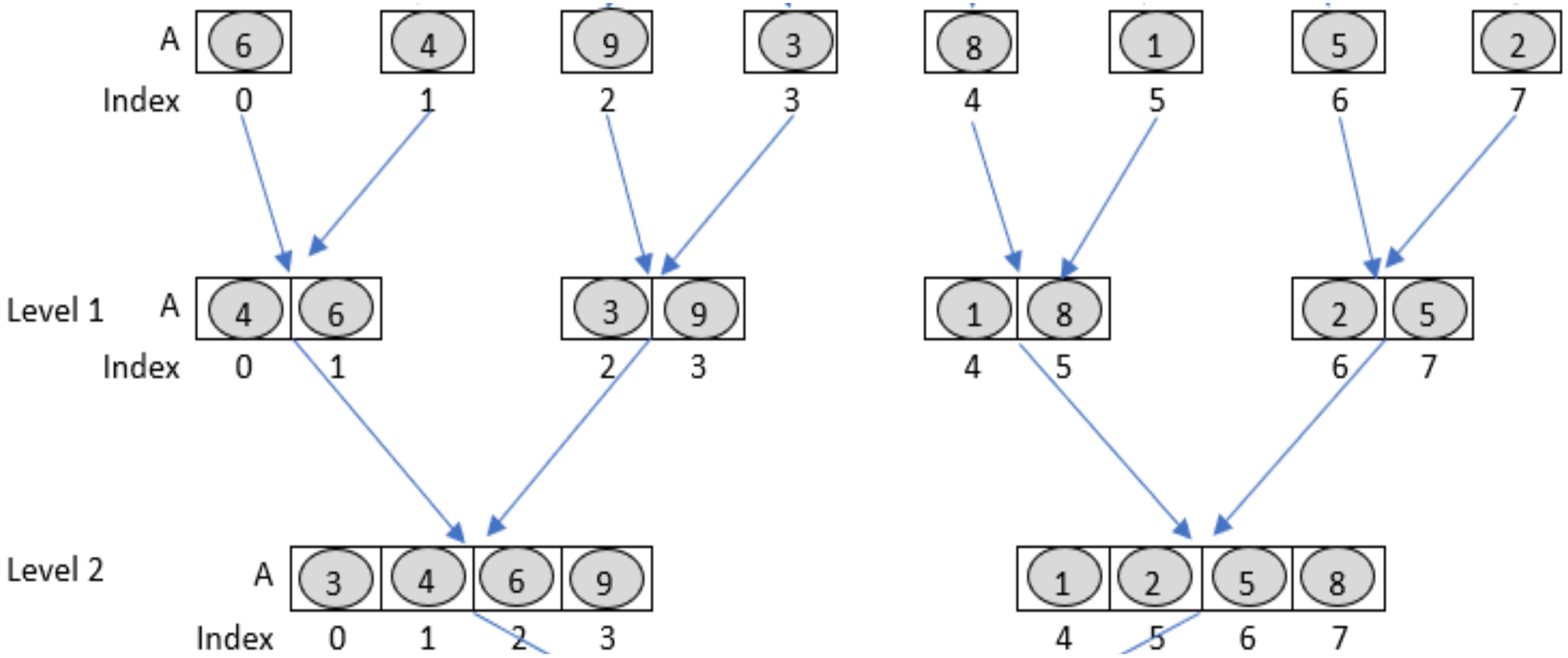
Merge Sort – Merge the smaller problems

A	6	4	9	3	8	1	5	2
Index	0	1	2	3	4	5	6	7

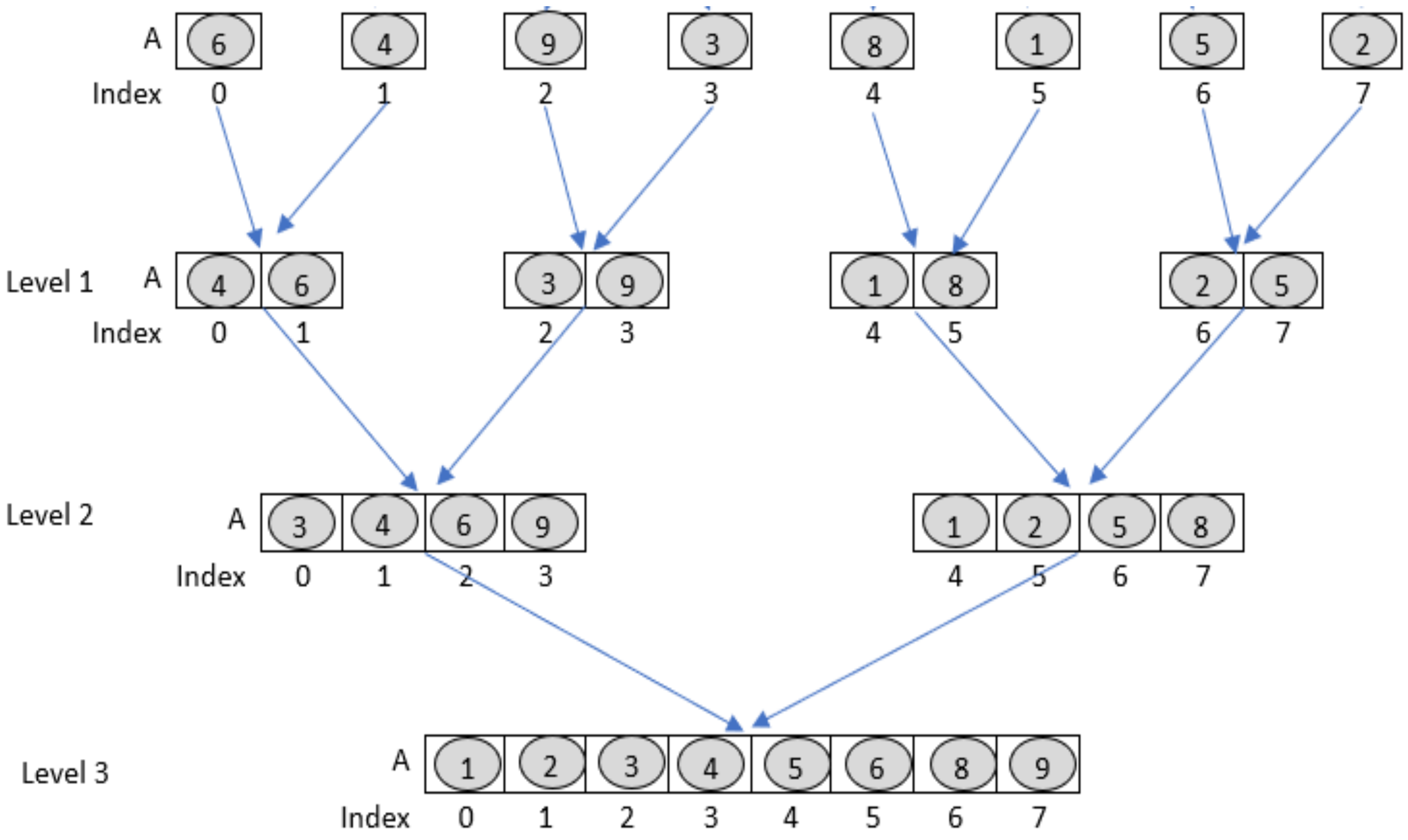
Merge Sort – Merge the smaller problems



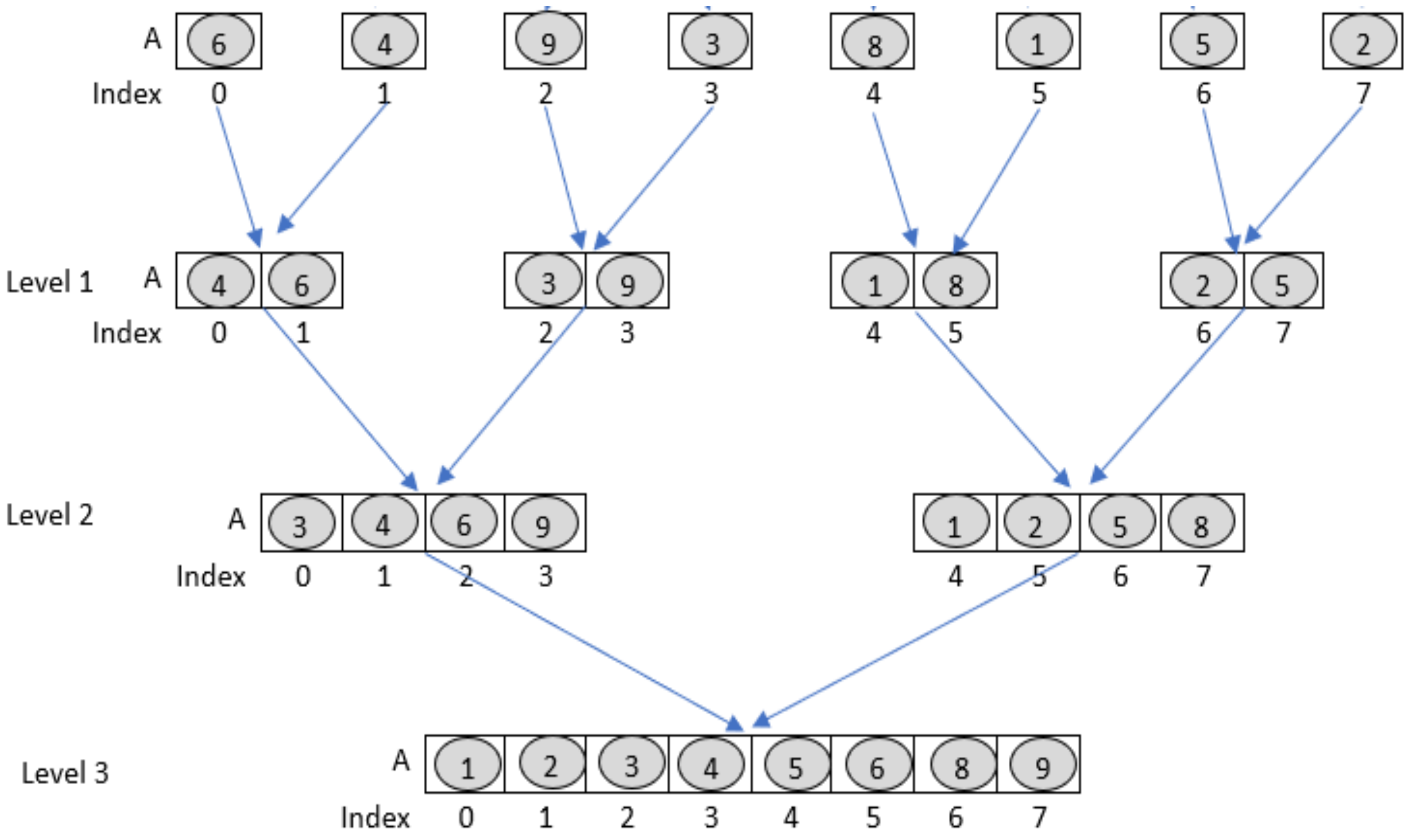
Merge Sort – Merge the smaller problems



Merge Sort – Merge the smaller problems



Merge Sort – Merge the smaller problems

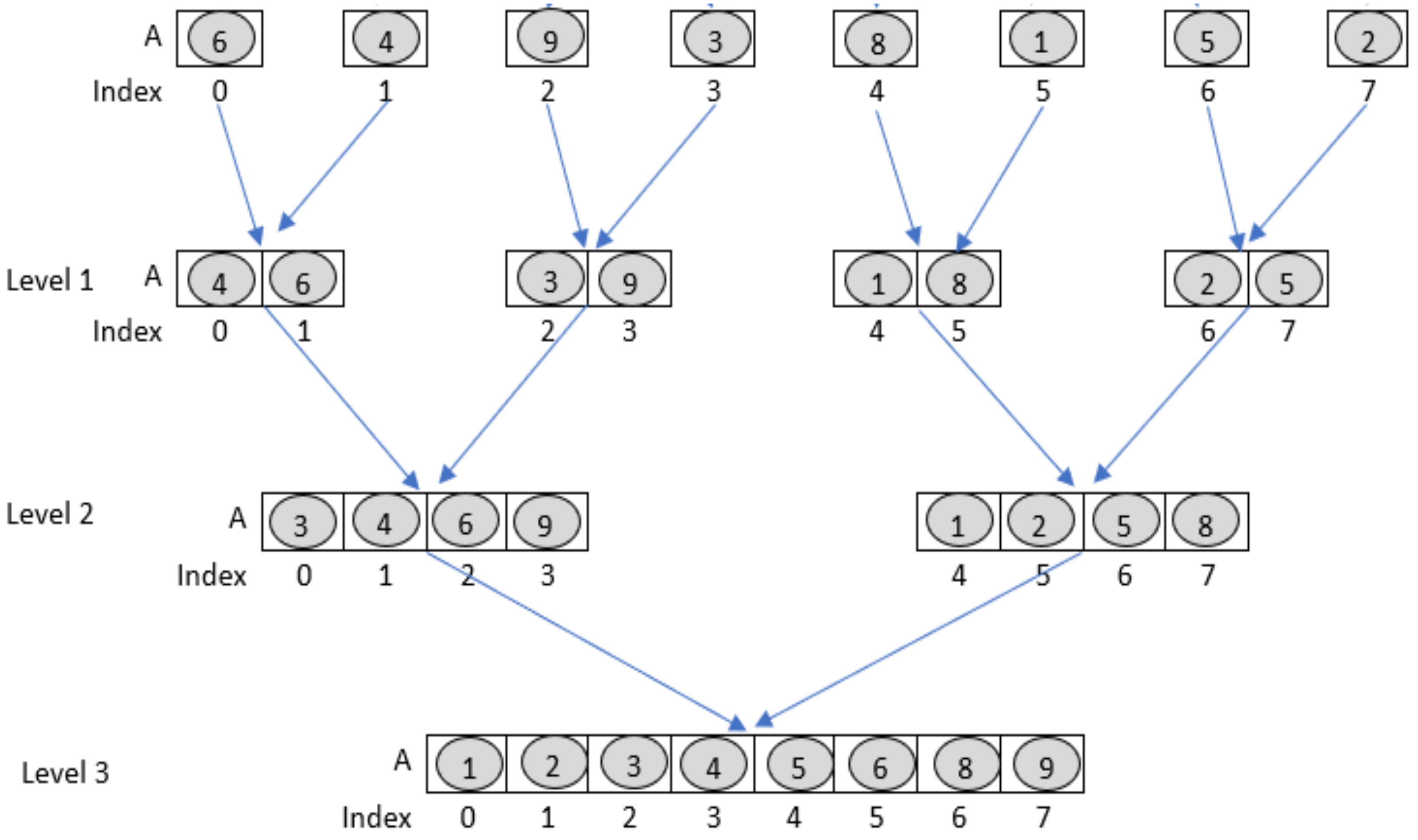


………………$\theta(n)$

………………$\theta(n)$

………………$\theta(n)$

Merge Sort – Merge the smaller problems



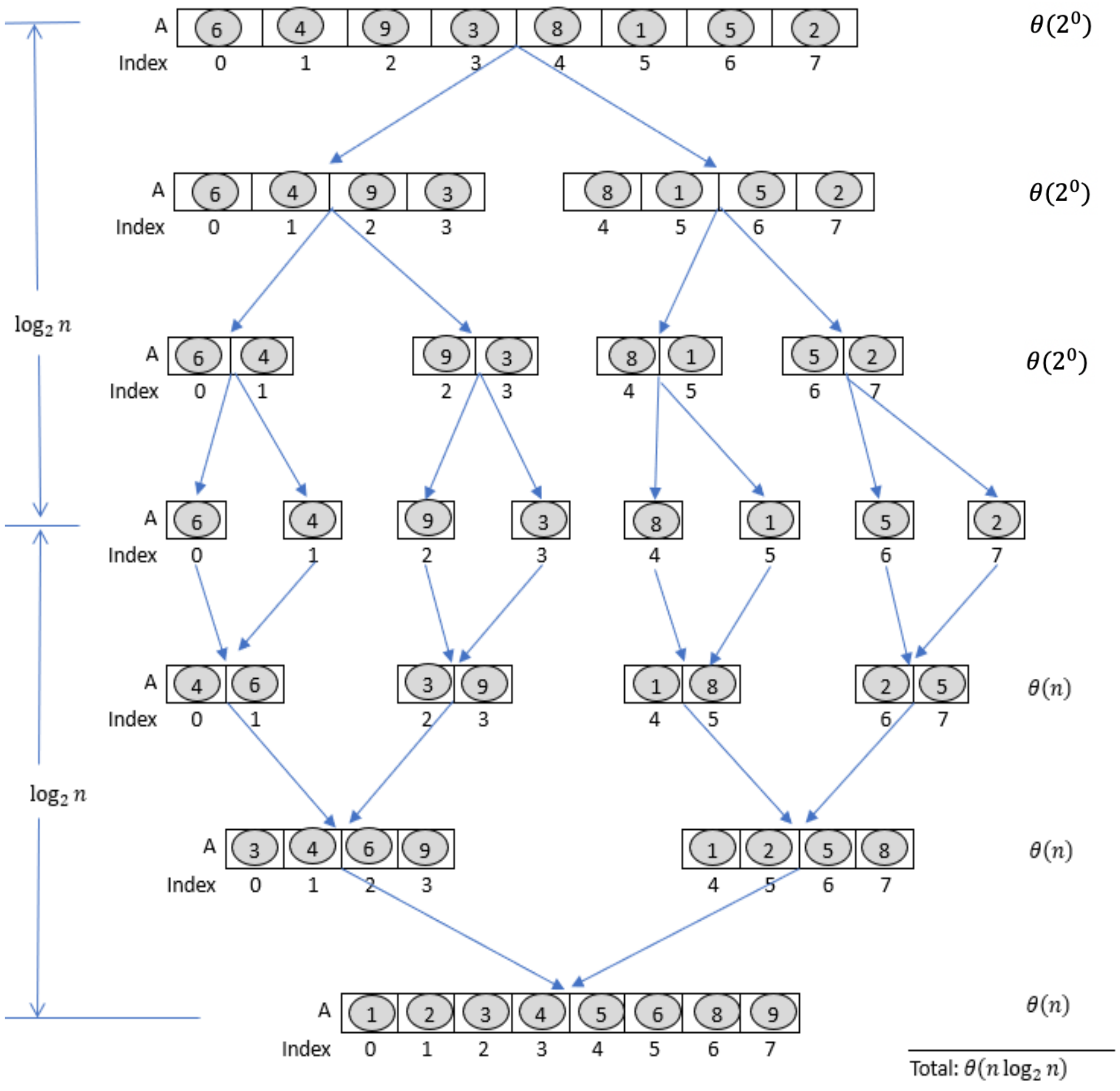
………………$\theta(n)$

………………$\theta(n)$

………………$\theta(n)$

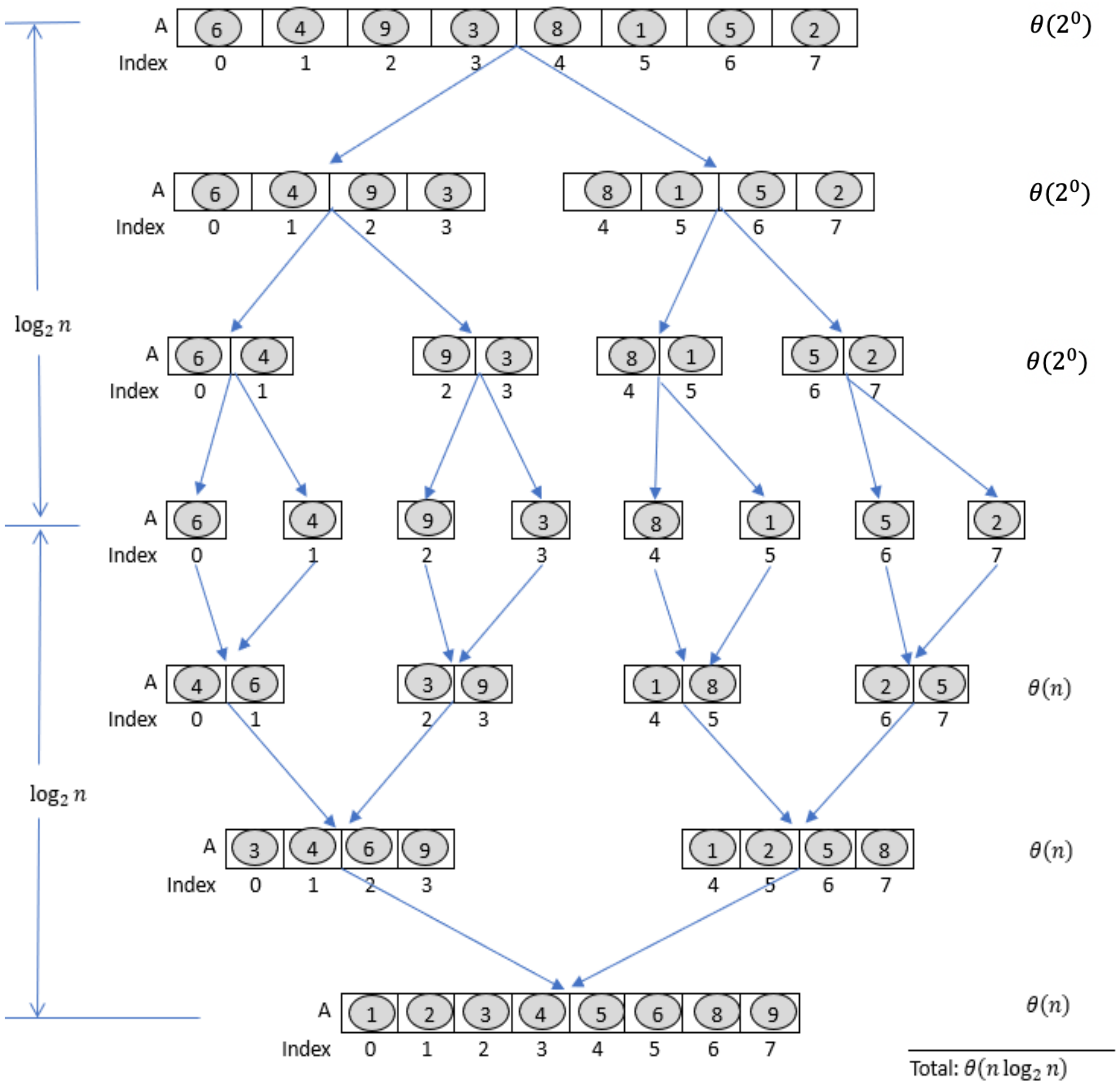
Total: $\theta(n \log_2 n)$

Merge Sort



A
Index
6 4 9 3 8 1 5 2
0 1 2 3 4 5 6 7
$\theta(2^0)$
$\theta(2^0)$
$\theta(2^0)$
$\log_2 n$
$\theta(n)$
$\theta(n)$
$\log_2 n$
1 2 3 4 5 6 8 9
$\theta(n)$
Total: $\theta(n \log_2 n)$

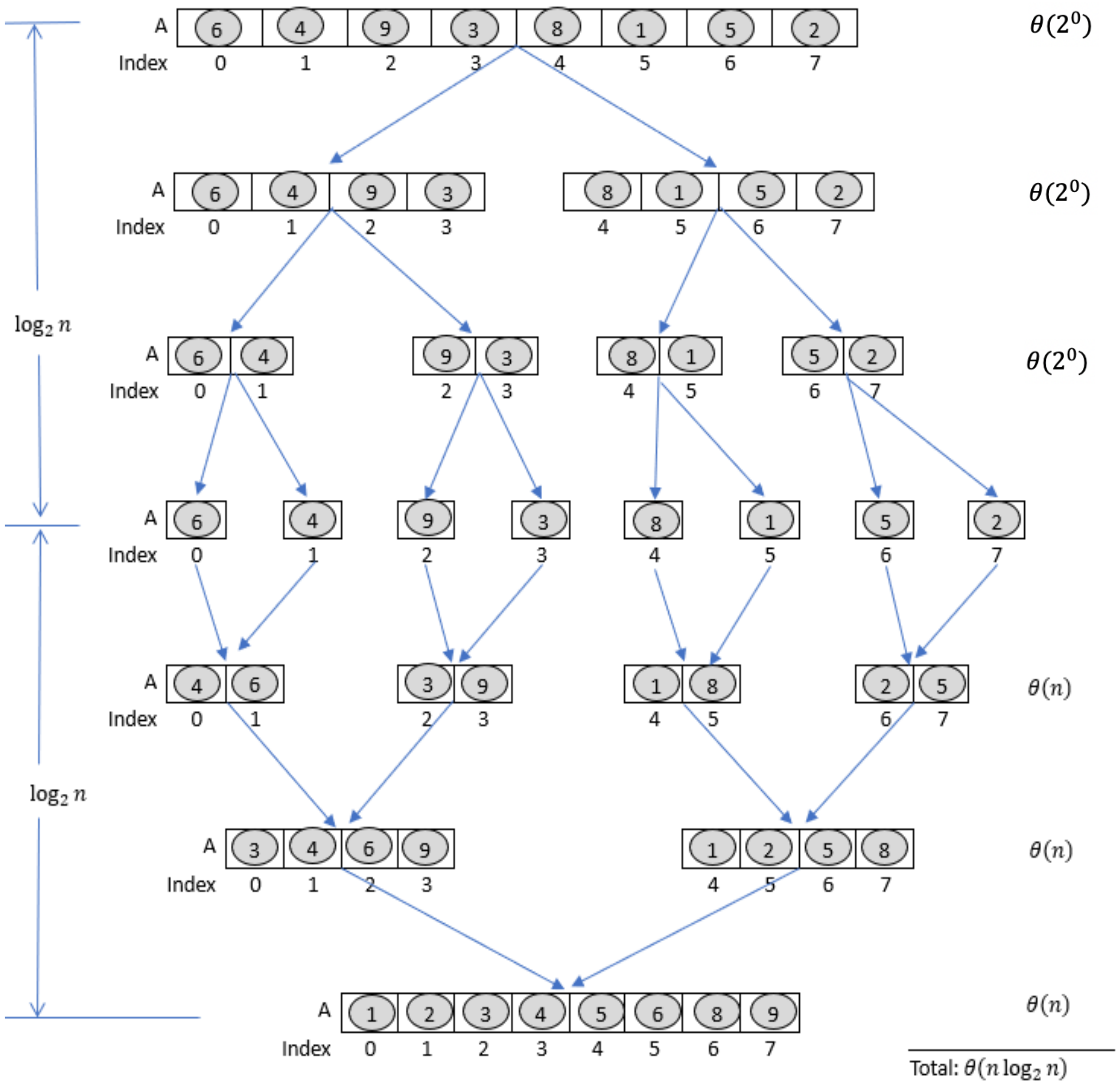
Merge Sort



Time Complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + n, T(1) = 1$$

Merge Sort

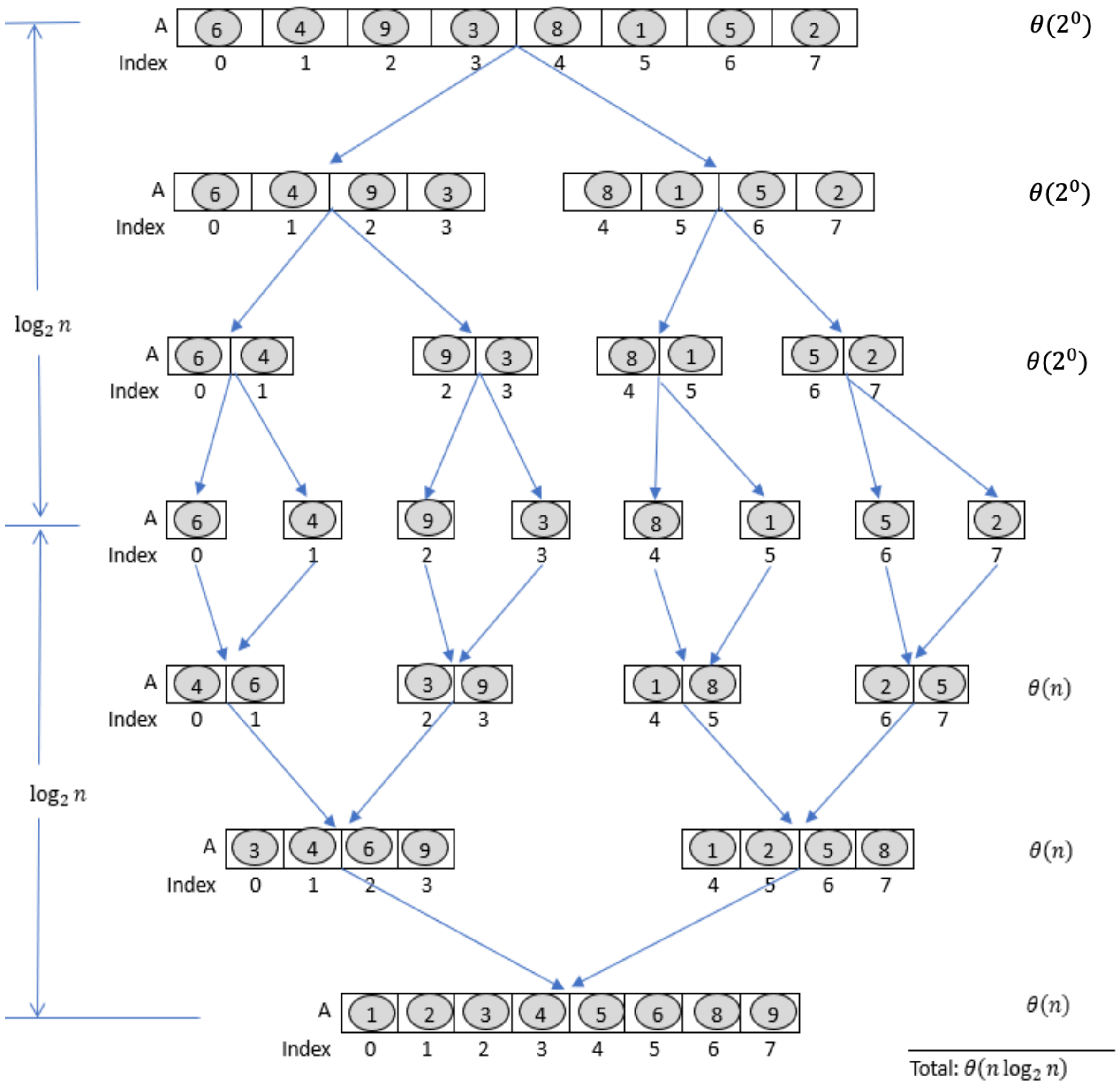


Time Complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$\Rightarrow T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2\frac{n}{2} + n$$

Merge Sort



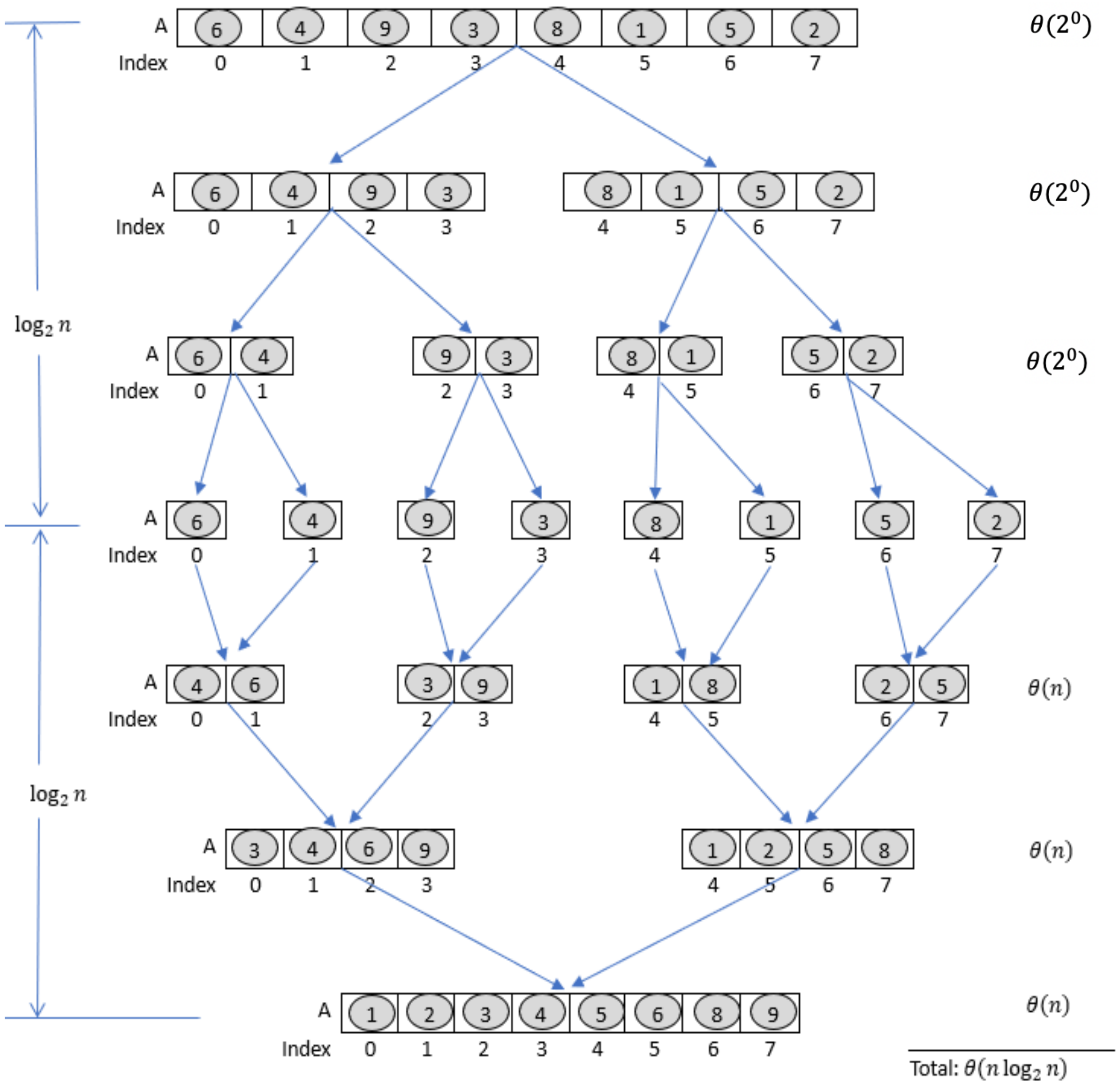
Time Complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$\Rightarrow T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2\frac{n}{2} + n$$

$$\Rightarrow T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 2^2\frac{n}{2^2} + 2\frac{n}{2} + n$$

Merge Sort



Time Complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$\Rightarrow T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2\frac{n}{2} + n$$

$$\Rightarrow T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 2^2\frac{n}{2^2} + 2\frac{n}{2} + n$$

.....

$$\Rightarrow T(n) = 2^l\, T\left(\frac{n}{2^l}\right) + n + n + \cdots + n + n$$

$$\Rightarrow T(n) = n + n + n + \cdots + n = \theta(n \log_2 n)$$

Merge Sort - Algorithm

```
void mergeSort(int A[], int first, int last){
  if(first < last){
        mergeSort(A, first, ⌈(first+last)/2⌉ − 1);
        mergeSort(A, ⌈(first+last)/2⌉, last);
        mergeSortedArrays(A, first,⌈(first+last)/2⌉, last)
  }
}
```

Merge Sort - Algorithm

```
void mergeSortedArrays(int A[], int first, int mid, int last){
    int i, j, k;
    int sl = mid - first-1;
    int sr= last - mid;
    int L[sl], R[sr];
    for (i = 0; i < sl; i++) L[i] = A[first + i];
    for (i = 0; i < sr; i++) R[i] = A[mid + i];
    i = j = 0;
    k = first;
    while (i < sl && j < sr){
        if (L[i] <= R[j]){
            A[k] = L[i];
            i++;
        }
        else{
            A[k] = R[j];
            j++;
        }
        k++;
    }
     while (i < sl){ // If R remains
        A[k] = L[i];
        i++;
        k++;
    }
    while (j < n2){ // If L remains
        A[k] = R[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(int A[], int first, int last){
   if(first < last){
            mergeSort(A, first, ⌈(first+last)/2⌉ − 1);
            mergeSort(A, ⌈(first+last)/2⌉, last);
            mergeSortedArrays(A, first, ⌈(first+last)/2⌉, last)
   }
}
```

Merge Sort – Why External Memory Sort

Lower Bound Theory

Lower Bound Theory

The lower bound theory **of a class of algorithms** for solving a particular problem defines the **lower bound time complexity** that any algorithm in the class must take to solve the given problem for an arbitrary input.

Lower Bound Theory

The lower bound theory **of a class of algorithms** for solving a particular problem defines the **lower bound time complexity** that any algorithm in the class must take to solve the given problem for an arbitrary input.

- **Problem:** Sorting
- **Class:** Comparison based sorting

Lower Bound Theory

Given a collection of n elements, there are n! number of possible permutations.

Lower Bound Theory

Given a collection of n elements, there are n! number of possible permutations.

A sorted sequence (ascending or descending) of the elements in the collection is one of the n! permutations.

Lower Bound Theory

Given a collection of n elements, there are n! number of possible permutations.

Sorted sequence (ascending or descending) of the elements in the collection is one of the n! permutations.

For example, if n is 3, and the possible elements are $\{a, b, c\}$, then its possible permutations are $(a, b, c), (a, c, b), (c, a, b), (b, c, a), (b, a, c)$, and (c, b, a).

Lower Bound Theory

Given a collection of n elements, there are n! number of possible permutations.

Sorted sequence (ascending or descending) of the elements in the collection is one of the n! permutations.

For example, if n is 3, and the possible elements are $\{a, b, c\}$, then its possible permutations are $(a, b, c), (a, c, b), (c, a, b), (b, c, a), (b, a, c)$, and (c, b, a).

If the given set is $\{a = 2, b = 5, c = 3\}$, then the sorted permutation is (a, c, b).

Lower Bound Theory

Given a collection of n elements, there are n! number of possible permutations.

Sorted sequence (ascending or descending) of the elements in the collection is one of the n! permutations.

For example, if n is 3, and the possible elements are $\{a, b, c\}$, then its possible permutations are $(a, b, c), (a, c, b), (c, a, b), (b, c, a), (b, a, c)$, and (c, b, a).

If the given set is $\{a = 2, b = 5, c = 3\}$, then the sorted permutation is (a, c, b).

If the given set is $\{a = 5, b = 1, c = 2\}$, then the sorted permutation is (b, c, a).

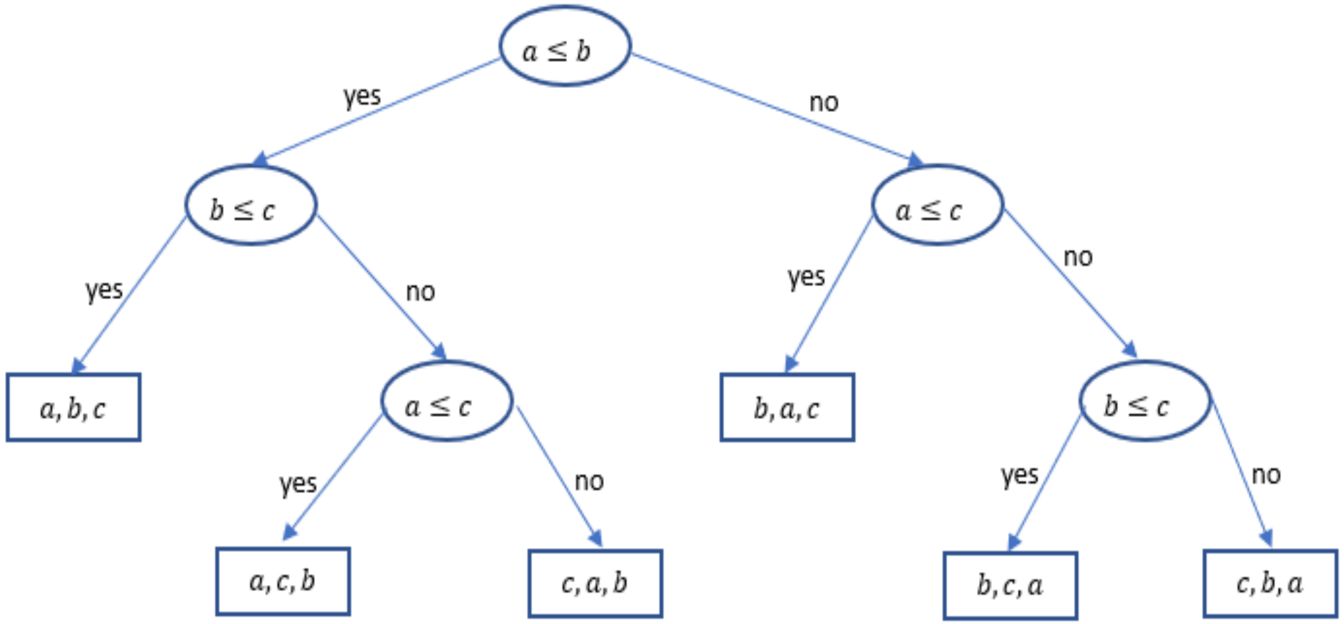
Lower Bound Theory

For a set of n elements, its permutations can be generated using **a decision tree**.

Lower Bound Theory

For a set of n elements, its permutations can be generated using a decision tree.

For example, for {a, b, c}, a decision tree to generate all possible permutation can be defined as below.

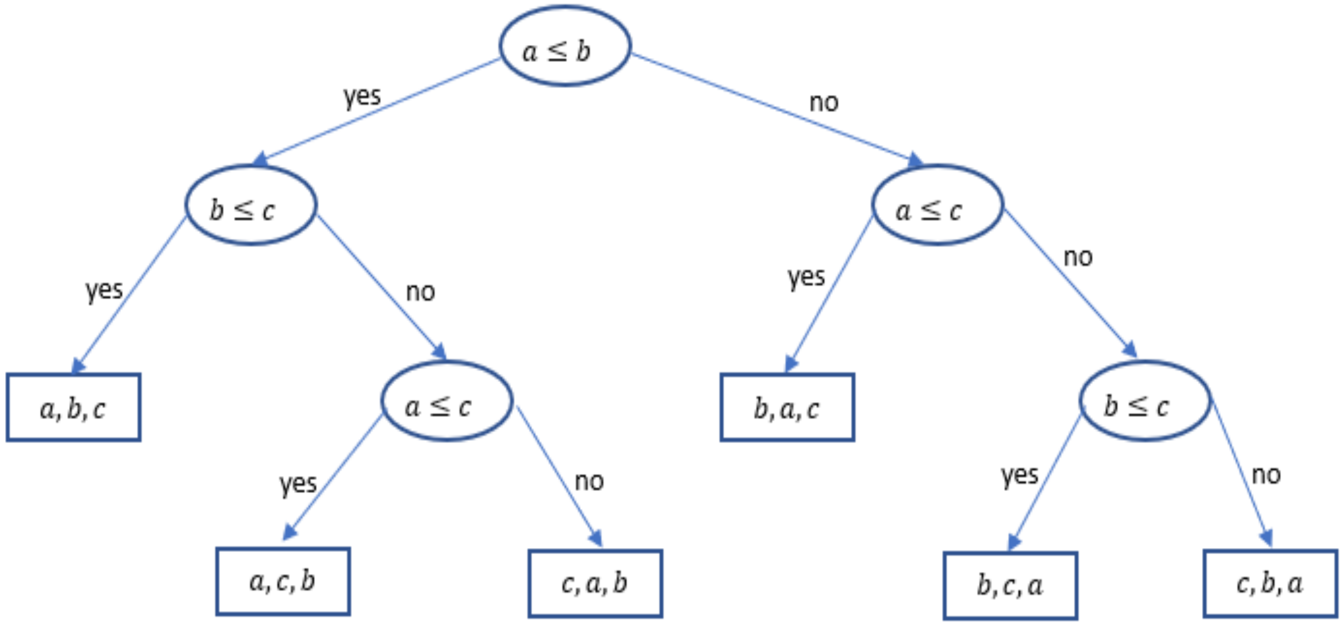


Lower Bound Theory

For a set of n elements, its permutations can be generated using a decision tree.

For example, for {a, b, c}, a decision tree to generate all possible permutation can be defined as below.

Leaf nodes are the possible permutations.



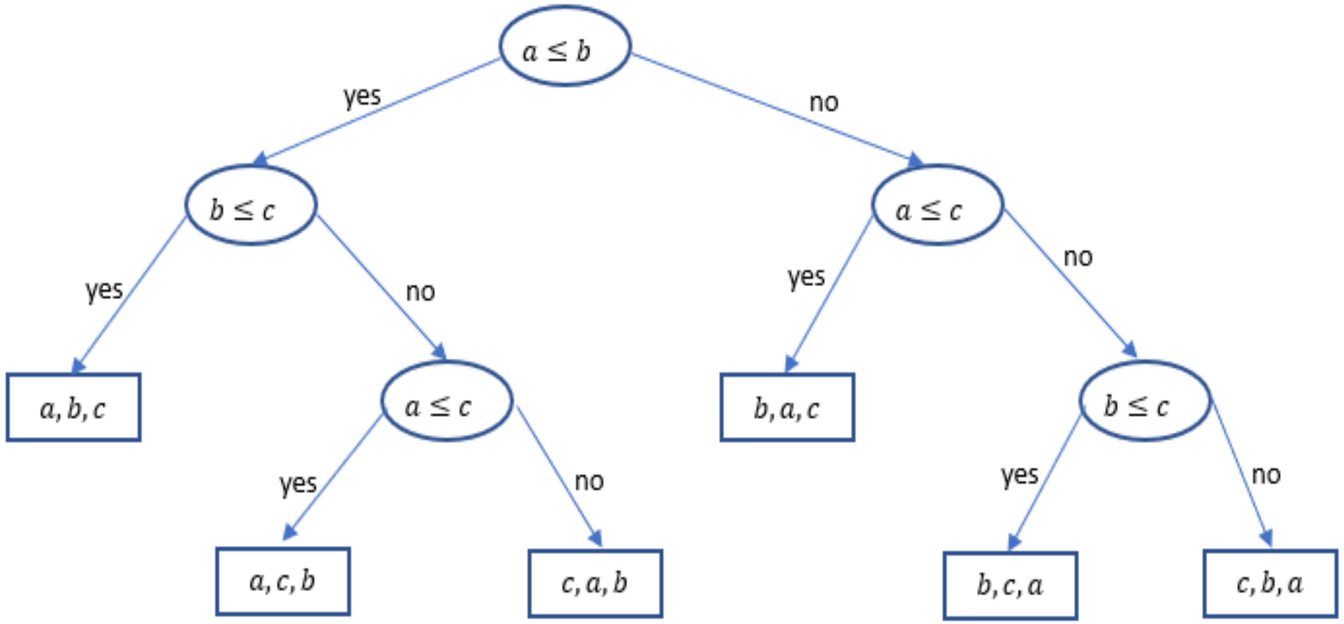
Lower Bound Theory

For a set of n elements, its permutations can be generated using a decision tree.

For example, for {a, b, c}, a decision tree to generate all possible permutation can be defined as below.

Leaf nodes are the possible permutations.

Internal nodes are the comparisons, and have two child nodes.



Lower Bound Theory

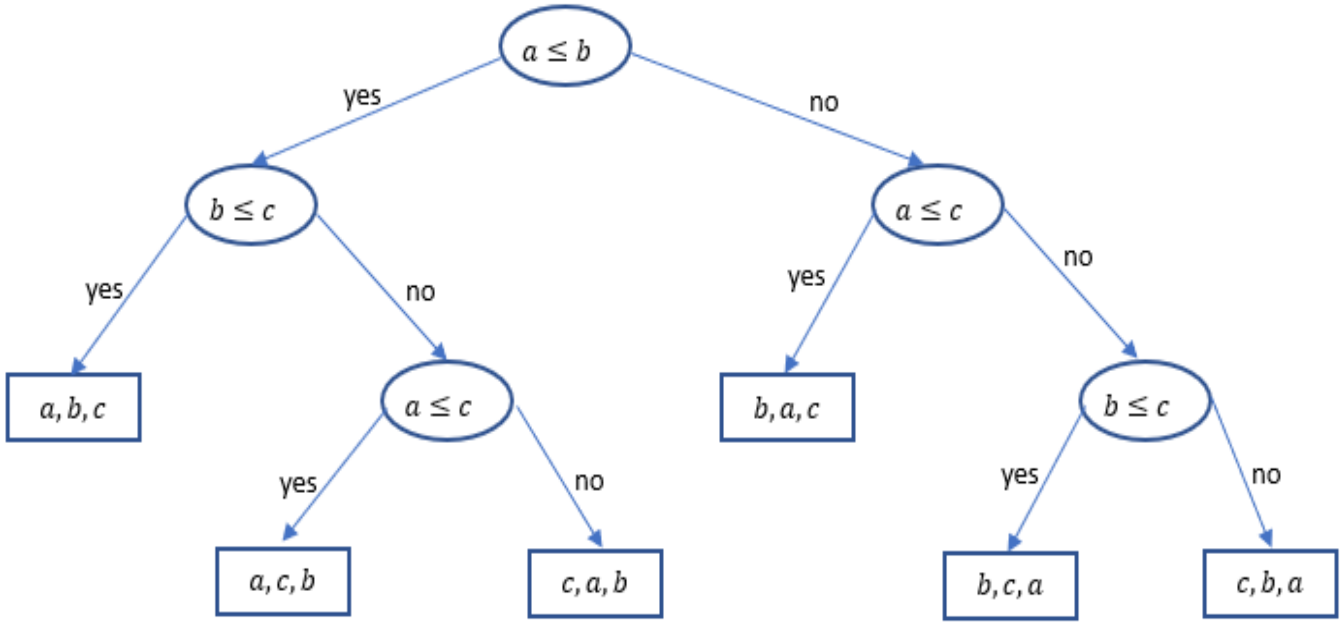
For a set of n elements, its permutations can be generated using a decision tree.

For example, for {a, b, c}, a decision tree to generate all possible permutation can be defined as below.

Leaf nodes are the possible permutations.

Internal nodes are the comparisons, and have two child nodes.

More than one decision tree possible.



Lower Bound Theory

For a set of n elements, its permutations can be generated using a decision tree.

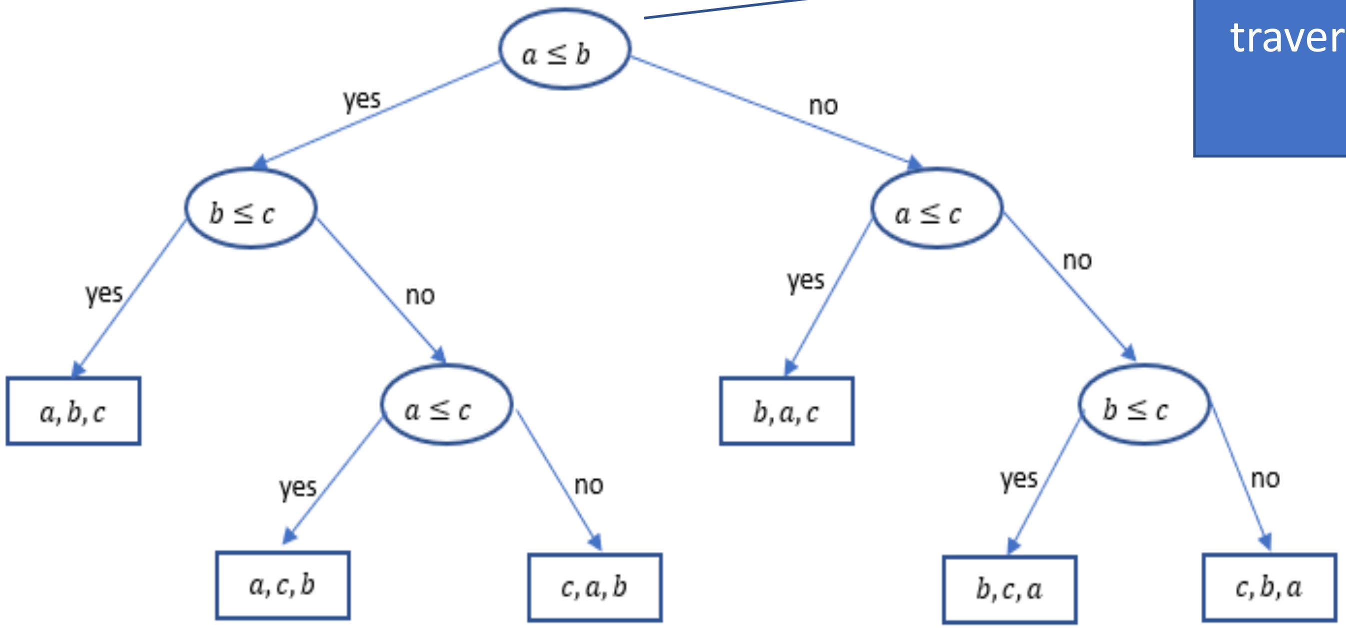
For example, for {a, b, c}, a decision tree to generate all possible permutation can be defined as below.

Leaf nodes are the possible permutations.

Internal nodes are the comparisons, and have two child nodes.

More than one decision tree possible.

Searching for a permutation is to traverse the tree from the toot to a leaf node



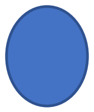
Lower Bound Theory

Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Lower Bound Theory

Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

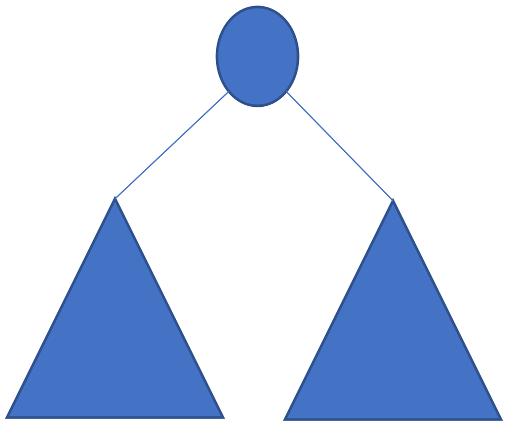


Lower Bound Theory

Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

Let us assume that it is true for $n > 1$.



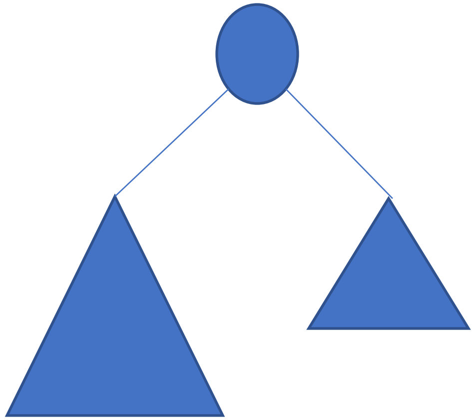
Lower Bound Theory

Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

Let us assume that it is true for $n > 1$.

It can be seen that at least one of the subtrees of root will have at least $\frac{n}{2}$ number of leaf nodes, and also less than n. That means, the height of the subtree with at least $\frac{n}{2}$ number of leaf nodes is at least $\log_2 \frac{n}{2} = \log_2 n - 1$.



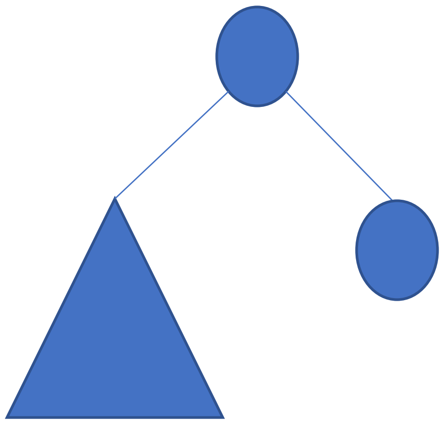
Lower Bound Theory

Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

Let us assume that it is true for $n > 1$.

It can be seen that at least one of the subtrees of root will have at least $\frac{n}{2}$ number of leaf nodes, and also less than n. That means, the height of the subtree with at least $\frac{n}{2}$ number of leaf nodes is at least $\log_2 \frac{n}{2} = \log_2 n - 1$.



Lower Bound Theory

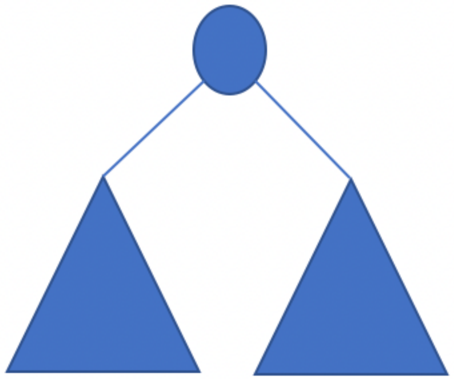
Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

Let us assume that it is true for $n > 1$.

It can be seen that at least one of the subtrees of root will have at least $\frac{n}{2}$ number of leaf nodes, and also less than n. That means, the height of the subtree with at least $\frac{n}{2}$ number of leaf nodes is at least $\log_2 \frac{n}{2} = \log_2 n - 1$.

If we add one more leaf node, in one of the subtrees, the above condition will still hold for the subtrees as $\frac{n}{2} + 1 < n$.



Lower Bound Theory

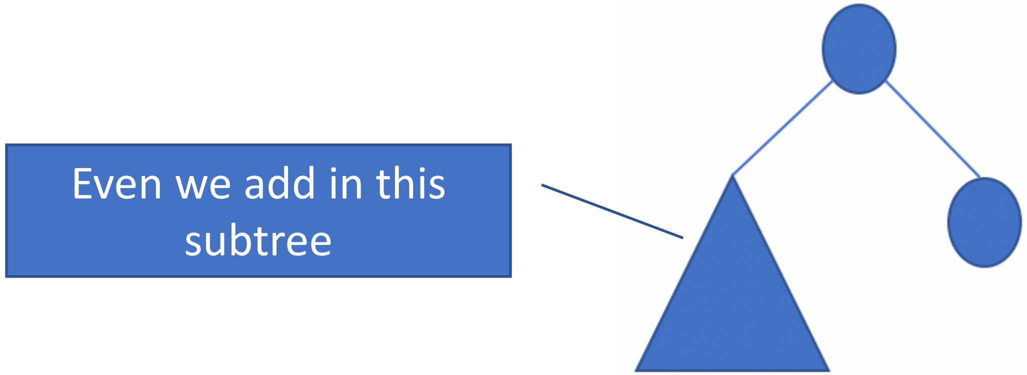
Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

Let us assume that it is true for $n > 1$.

It can be seen that at least one of the subtrees of root will have at least $\frac{n}{2}$ number of leaf nodes, and also less than n. That means, the height of the subtree with at least $\frac{n}{2}$ number of leaf nodes is at least $\log_2 \frac{n}{2} = \log_2 n - 1$.

If we add one more leaf node, in one of the subtrees, the above condition will still hold for the subtrees as $\frac{n}{2} + 1 < n$.



Lower Bound Theory

Theorem: The height $h \geq 0$ of a binary tree with $n \geq 1$ number of leaf nodes, where every non-leaf node has two child nodes, is at least $\log_2 n$.

Proof: (By induction on n) If $n = 1$, the tree has only root node, and it is also a leaf node. As the height of a leaf node is 0, it is true for $n = 1$.

Let us assume that it is true for $n > 1$.

It can be seen that at least one of the subtrees of root will have at least $\frac{n}{2}$ number of leaf nodes, and also less than n. That means, the height of the subtree with at least $\frac{n}{2}$ number of leaf nodes is at least $\log_2 \frac{n}{2} = \log_2 n - 1$.

If we add one more leaf node, in one of the subtrees, the above condition will still hold for the subtrees as $\frac{n}{2} + 1 < n$.

As the height of a binary tree is the height of the tallest child subtree plus one, the height of the binary tree with $n + 1$ leaf nodes is at least $\log_2 \frac{n+1}{2} + 1 = \log_2(n + 1)$.

What is the Lower Bound Theory?

For an arbitrary set of ***n*** elements, there are ***n!*** number of permutations and all the permutations have **equal probability of being picked up as the sorted permutation**.

What is the Lower Bound Theory?

For an arbitrary set of ***n*** elements, there are ***n!*** number of permutations and all the permutations have **equal probability of being picked up as the sorted permutation**.

To define lower bound, **we need to prove that the *expected height of any decision tree* for sorting n element is at least $log_2 n!$.**

What is the Lower Bound Theory?

Let T be a decision tree for sorting n elements. It has exactly $n!$ number of leaf nodes.

What is the Lower Bound Theory?

Let T be a decision tree for sorting n elements. It has exactly $n!$ number of leaf nodes.

Let $D(T)$ be the sum of the depths of all the leaf nodes in T.

What is the Lower Bound Theory?

Let T be a decision tree for sorting n elements. It has exactly $n!$ number of leaf nodes.

Let $D(T)$ be the sum of the depths of all the leaf nodes in T.

Let $M(T)$ be the minimum $D(T)$ over all possible decision tree with $n!$ leaf nodes.

What is the Lower Bound Theory?

Let T be a decision tree for sorting n elements. It has exactly $n!$ number of leaf nodes.

Let $D(T)$ be the sum of the depths of all the leaf nodes in T.

Let $M(T)$ be the minimum $D(T)$ over all possible decision tree with $n!$ leaf nodes.

If i be the number of leaf nodes in the left subtree of the root of T, then there will be $(n! - i)$ number of leaf node in the right subtree.

What is the Lower Bound Theory?

Let T be a decision tree for sorting n elements. It has exactly $n!$ number of leaf nodes.

Let $D(T)$ be the sum of the depths of all the leaf nodes in T.

Let $M(T)$ be the minimum $D(T)$ over all possible decision tree with $n!$ leaf nodes.

If i be the number of leaf nodes in the left subtree of the root of T, then there will be $(n! - i)$ number of leaf node in the right subtree.

Then D(T) can be defined recursively as below.

$$D(T) = D(T_i) + D(T_{n!-i}) + n!$$

The $n!$ in the above expression is additional depth of 1 from root to the roots of left and right subtree for all the $n!$ leaf nodes.

What is the Lower Bound Theory?

Let T be a decision tree for sorting n elements. It has exactly $n!$ number of leaf nodes.

Let $D(T)$ be the sum of the depths of all the leaf nodes in T.

Let $M(T)$ be the minimum $D(T)$ over all possible decision tree with $n!$ leaf nodes.

If i be the number of leaf nodes in the left subtree of the root of T, then there will be $(n! - i)$ number of leaf node in the right subtree.

Then D(T) can be defined recursively as below.

$$D(T) = D(T_i) + D(T_{n!-i}) + n!$$

The $n!$ in the above expression is additional depth of 1 from root to the roots of left and right subtree for all the $n!$ leaf nodes.

$$M(T) = arg\min_i\{D(T_i) + D(T_{n!-i}) + n!\}$$

What is the Lower Bound Theory?

Now, we need to prove that $M(T) \geq n! \log_2 n!$.

$$M(T) = arg \min_i \{D(T_i) + D(T_{n!-i}) + n!\}$$

What is the Lower Bound Theory?

Now, we need to prove that $M(T) \geq n! \log_2 n!$.

$$M(T) = arg \min_i \{D(T_i) + D(T_{n!-i}) + n!\}$$

We can prove it by induction on n. When $n = 1$, $M(T) = 0! \log_2 0! = 0$. The condition holds for $n = 1$.

What is the Lower Bound Theory?

Now, we need to prove that $M(T) \geq n! \log_2 n!$.

$M(T) = arg \min_i \{D(T_i) + D(T_{n!-i}) + n!\}$

We can prove it by induction on n. When $n = 1$, $M(T) = 0! \log_2 0! = 0$. The condition holds for $n = 1$.

Let us assume that the condition holds for $n - 1$. For n elements, we can define $M(T)$ as

$$M(T) = \arg\min_i \{D(T_i) + D(T_{n!-i}) + n!\} \geq \arg\min_i \{i \log_2 i + (n! - i) \log_2 (n! - i) + n!\}$$

What is the Lower Bound Theory?

Now, we need to prove that $M(T) \geq n! \log_2 n!$.

$$M(T) = arg \min_i \{D(T_i) + D(T_{n!-i}) + n!\}$$

We can prove it by induction on n. When $n = 1$, $M(T) = 0! \log_2 0! = 0$. The condition holds for $n = 1$.

Let us assume that the condition holds for $n - 1$. For n elements, we can define $M(T)$ as

$$M(T) = \arg\min_i \{D(T_i) + D(T_{n!-i}) + n!\} \geq \arg\min_i \{i \log_2 i + (n! - i) \log_2 (n! - i) + n!\}$$

The above expression will be minimum when $i = \frac{n!}{2}$. As $\frac{n!}{2} \leq (n-1)!$, we can write

$$M(T) \geq \frac{n!}{2} \log_2 \frac{n!}{2} + \left(n! - \frac{n!}{2}\right) \log_2 \left(n! - \frac{n!}{2}\right) + n!$$

$$= n! \log_2 \frac{n!}{2} + n! = n! \left(\log_2 n! - 1\right) + n! = n! \log_2 n!.$$

What is the Lower Bound Theory?

Now, we need to prove that $M(T) \geq n! \log_2 n!$.

$M(T) = arg \min_i \{D(T_i) + D(T_{n!-i}) + n!\}$

We can prove it by induction on n. When $n = 1$, $M(T) = 0! \log_2 0! = 0$. The condition holds for $n = 1$.

Let us assume that the condition holds for $n - 1$. For n elements, we can define $M(T)$ as

$$M(T) = \arg\min_i \{D(T_i) + D(T_{n!-i}) + n!\} \geq \arg\min_i \{i \log_2 i + (n! - i) \log_2 (n! - i) + n!\}$$

The above expression will be minimum when $i = \frac{n!}{2}$. As $\frac{n!}{2} \leq (n-1)!$, we can write

$$M(T) \geq \frac{n!}{2} \log_2 \frac{n!}{2} + \left(n! - \frac{n!}{2}\right) \log_2 \left(n! - \frac{n!}{2}\right) + n!$$

$$= n! \log_2 \frac{n!}{2} + n! = n! (\log_2 n! - 1) + n! = n! \log_2 n!.$$

So, the expected height of the decision tree which constitutes M(T) is $\frac{M(T)}{n!} \geq \log_2 n!$.

What is the Lower Bound Theory?

Determining a sorted sequence from the decision tree with $n!$ number of leaf nodes is the expected number of comparisons.

What is the Lower Bound Theory?

Determining a sorted sequence from the decision tree with $n!$ number of leaf nodes is the expected number of comparisons.

The expected number of comparison is defined by expected height of the decision tree.

What is the Lower Bound Theory?

Determining a sorted sequence from the decision tree with $n!$ number of leaf nodes is the expected number of comparisons.

The expected number of comparison is defined by expected height of the decision tree.

Therefore, lower bound of any comparable sorting algorithm for an arbitrary input set of n elements is defined by the expected height of the corresponding decision tree i.e., $\log_2 n!$.

What is the Lower Bound Theory?

Determining a sorted sequence from the decision tree with $n!$ number of leaf nodes is the expected number of comparisons.

The expected number of comparison is defined by expected height of the decision tree.

Therefore, lower bound of any comparable sorting algorithm for an arbitrary input set of n elements is defined by the expected height of the corresponding decision tree i.e., $\log_2 n!$.

$$\log_2 n! \geq \log_2 \left(n(n-1)(n-2) \ldots \left\lceil\frac{n}{2}\right\rceil\right) \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$= \frac{n}{2}[\log_2 n - 1] = \frac{n}{2}\log_2 n - \frac{n}{2}$$

$$= \Omega(n \log_2 n).$$

What is the Lower Bound Theory?

Determining a sorted sequence from the decision tree with $n!$ number of leaf nodes is the expected number of comparisons.

The expected number of comparison is defined by expected height of the decision tree.

Therefore, lower bound of any comparable sorting algorithm for an arbitrary input set of n elements is defined by the expected height of the corresponding decision tree i.e., $\log_2 n!$.

$$\log_2 n! \geq \log_2 \left(n(n-1)(n-2) \ldots \left\lceil \frac{n}{2} \right\rceil\right) \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$= \frac{n}{2}[\log_2 n - 1] = \frac{n}{2}\log_2 n - \frac{n}{2}$$

$$= \Omega(n \log_2 n).$$

Lower Bound Theory says, any comparison based sorting algorithm will take at least order of $(n \log_2 n)$ time complexity

Non-Comparison based Sorting Algorithms

Counting Sort

Counting Sort

- It is a non-comparison based sorting algorithm.
- It works in asymptotic linear time complexity.
- It works only for *real numbers*.

Counting Sort

Counting sort needs an auxiliary array **Count** which stores the count of each element.

Counting Sort

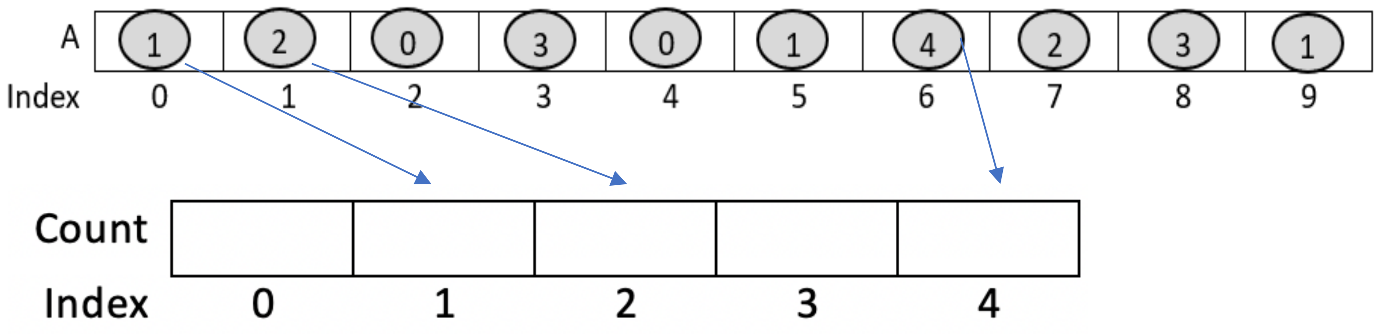
Counting sort needs an auxiliary array **Count** which stores the count of each element.

The idea is, for every element in the given array, there should be a corresponding index in Count.

Counting Sort

Counting sort needs an auxiliary array **Count** which stores the count of each element.

The idea is, for every element in the given array, there should be a corresponding index in Count.



What will count array store?

It stores the count of each element.

The i^{th} **index in Count** will store the **frequency count of i as element in A.**

A	1	2	0	3	0	1	4	2	3	1
Index	0	1	2	3	4	5	6	7	8	9

Count	2	3	2	2	1
Index	0	1	2	3	4

How do you define the Count Array?

Scenario -1

The given array has only positive values.

Scenario -1

The given array has only positive values.

A	1	2	0	3	0	1	4	2	3	1
Index	0	1	2	3	4	5	6	7	8	9

Count					
Index	0	1	2	3	4

Size of Count array = max(A) + 1

Scenario - 1

The minimum is larger than 0, and elements are not uniformly distributed over indexes

Scenario - 1

The minimum is larger than 0, and elements are not uniformly distributed over indexes

A	3	10	3	7	10	3	10	5	7	3
Index	0	1	2	3	4	5	6	7	8	9

Count				4		1		2			3
Index	0	1	2	3	4	5	6	7	8	9	10

Scenario - 2

A	3	10	3	7	10	3	10	5	7	3
Index	0	1	2	3	4	5	6	7	8	9

Count	4		1		2			3
Index	0	1	2	3	4	5	6	7

Count[i] = i + min(A)

Scenario – 3: with negative values

A	-1	10	3	7	10	3	10	5	7	3
Index	0	1	2	3	4	5	6	7	8	9

Count	1				3		1		2			3
Index	-1	0	1	2	3	4	5	6	7	8	9	10

LB = min(A)

Count	1				3		1		2			3
Index	0	1	2	3	4	5	6	7	8	9	10	11

Count[i] = i + min(A)

How to use Count array to sort an Array?

A	1	2	0	3	0	1	4	2	3	1
Index	0	1	2	3	4	5	6	7	8	9

Count	2	3	2	2	1
Index	0	1	2	3	4

Counting Sort – An Approach

Count	2	3	2	2	1
Index	0	1	2	3	4

A	0	0								
Index	0	1	2	3	4	5	6	7	8	9

Counting Sort – An Approach

Count	2	3	2	2	1
Index	0	1	2	3	4

A	0	0	1	1	1					
Index	0	1	2	3	4	5	6	7	8	9

Counting Sort – An Approach

Count	2	3	2	2	1
Index	0	1	2	3	4

A	0	0	1	1	1	2	2			
Index	0	1	2	3	4	5	6	7	8	9

Counting Sort – An Approach

Count	2	3	2	2	1
Index	0	1	2	3	4

A	0	0	1	1	1	2	2	3	3	
Index	0	1	2	3	4	5	6	7	8	9

Counting Sort – An Approach

Count	2	3	2	2	1
Index	0	1	2	3	4

A	0	0	1	1	1	2	2	3	3	4
Index	0	1	2	3	4	5	6	7	8	9

Counting Sort – An Approach

Count	2	3	2	2	1
Index	0	1	2	3	4

A	0	0	1	1	1	2	2	3	3	4
Index	0	1	2	3	4	5	6	7	8	9

This approach is simple and easy, but it will not be stable.

What is Counting Sort? - Algorithm

A	1	2	0	3	0	1	4	2	3	1
Index	0	1	2	3	4	5	6	7	8	9

Count	2	3	2	2	1
Index	0	1	2	3	4

```
1. Given an array A, construct
   Count array

2. Convert Count array to
   cumulative sum array

3. For (i=UB; i>=LB; i--)

4.        pos = Count[ A[i] ]

5.        pos = pos -1

6.        B[pos] = A[i]

7.        Count[A[i]] --

8. Move B to A
```

What is Counting Sort?

A	1	2	0	3	0	1	4	2	3	1
Index	0	1	2	3	4	5	6	7	8	9

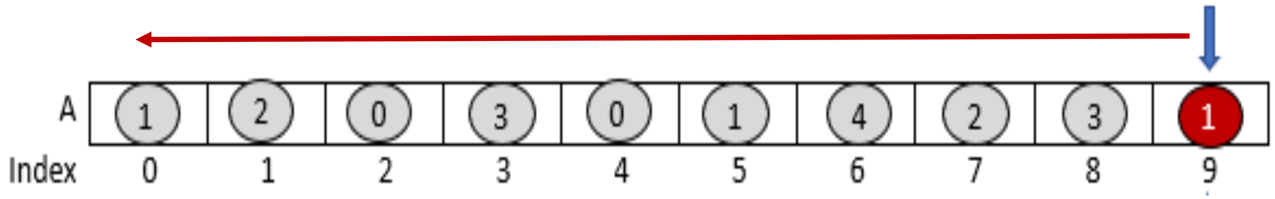
Count	2	3	2	2	1
Index	0	1	2	3	4

The counts in Count array are sequentially summed, so that Count[i] stores the cumulative sum from Count[0] to Count[i].

Count	2	5	7	9	10
Index	0	1	2	3	4

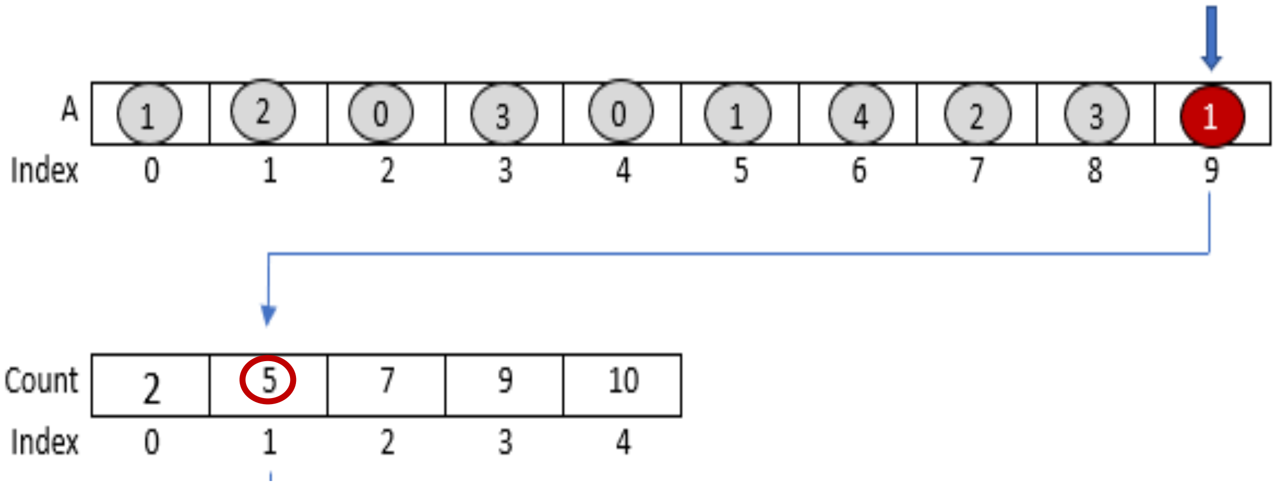
```
1. Given an array A, construct
   Count array

2. Convert Count array to
   cumulative sum array

3. For (i=UB; i>=LB; i--)

4.         pos = Count[ A[i] ]

5.         pos = pos -1

6.         B[pos] = A[i]

7.         Count[A[i]] --

8. Move B to A
```

What is Counting Sort?




```
1. Given an array A, construct
   Count array

2. Convert Count array to
   cumulative sum array

3. For (i=UB; i>=LB; i--)

4.        pos = Count[ A[i] ]

5.        pos = pos -1

6.        B[pos] = A[i]

7.        Count[A[i]] --

8. Move B to A
```

What is Counting Sort?



1. Given an array A, construct Count array

2. Convert Count array to cumulative sum array

3. For (i=UB; i>=LB; i--)

4. **pos = Count[A[i]]**

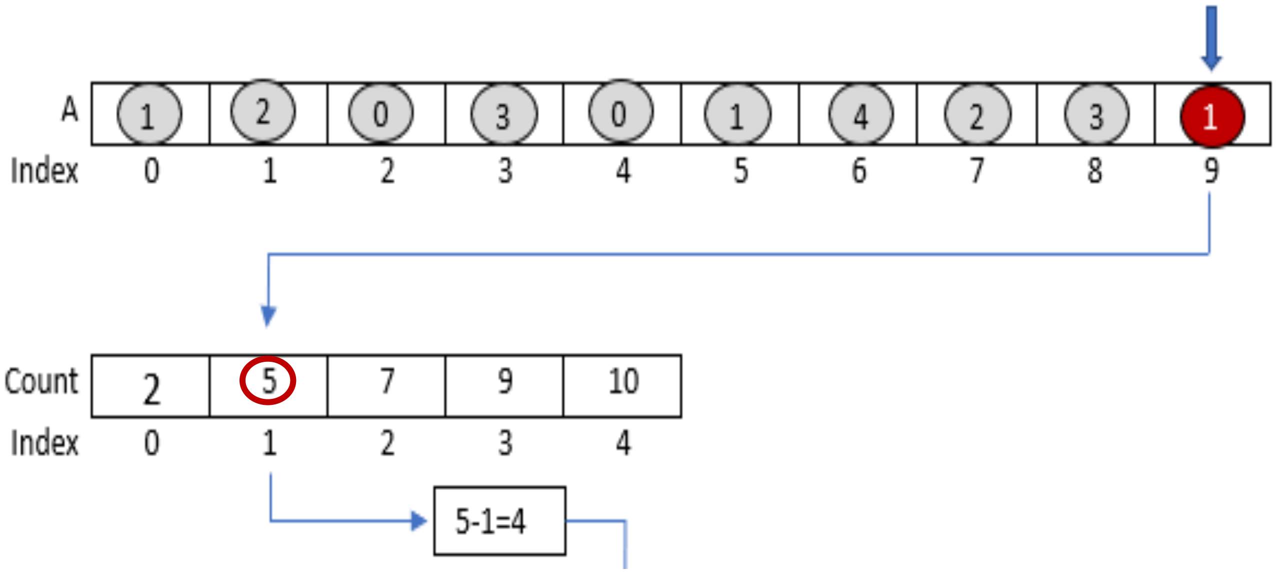
5. pos = pos -1

6. B[pos] = A[i]

7. Count[A[i]] --

8. Move B to A

What is Counting Sort?



1. Given an array A, construct Count array

2. Convert Count array to cumulative sum array

3. For (i=UB; i>=LB; i--)

4. pos = Count[A[i]]

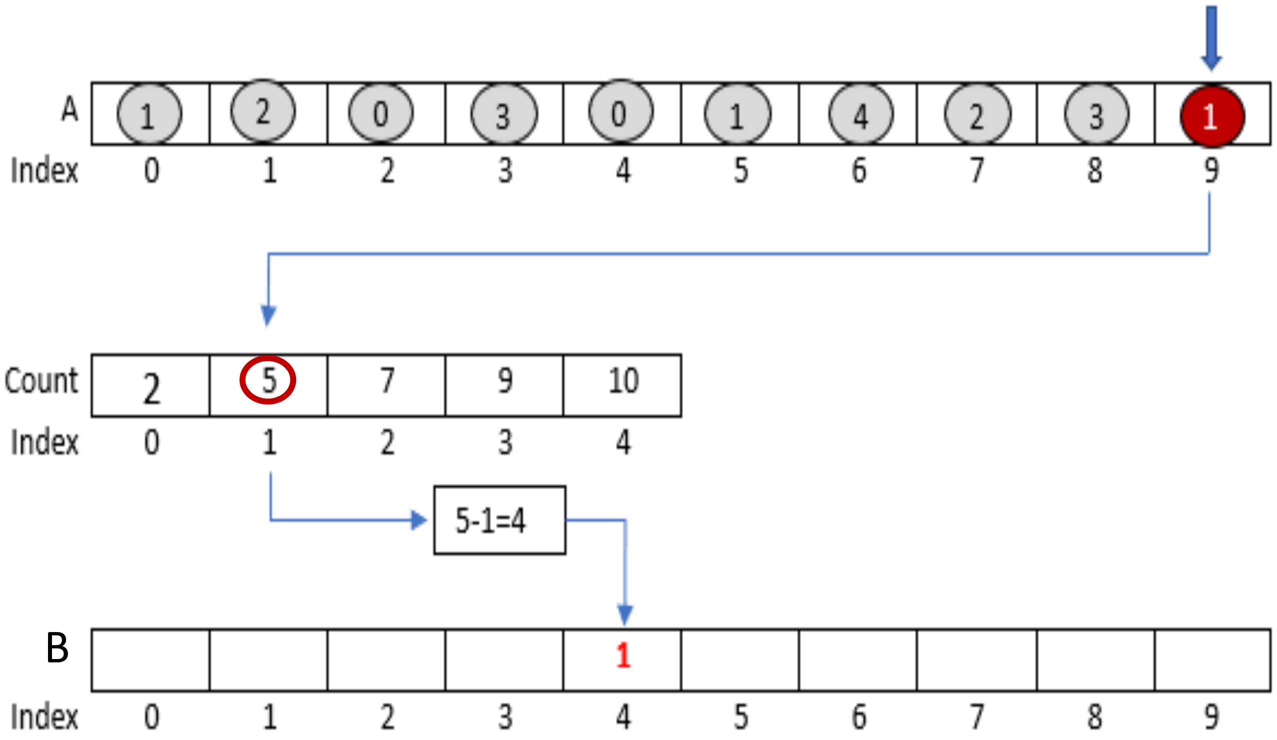
5. **pos = pos – LB - 1**

6. B[pos] = A[i]

7. Count[A[i]] --

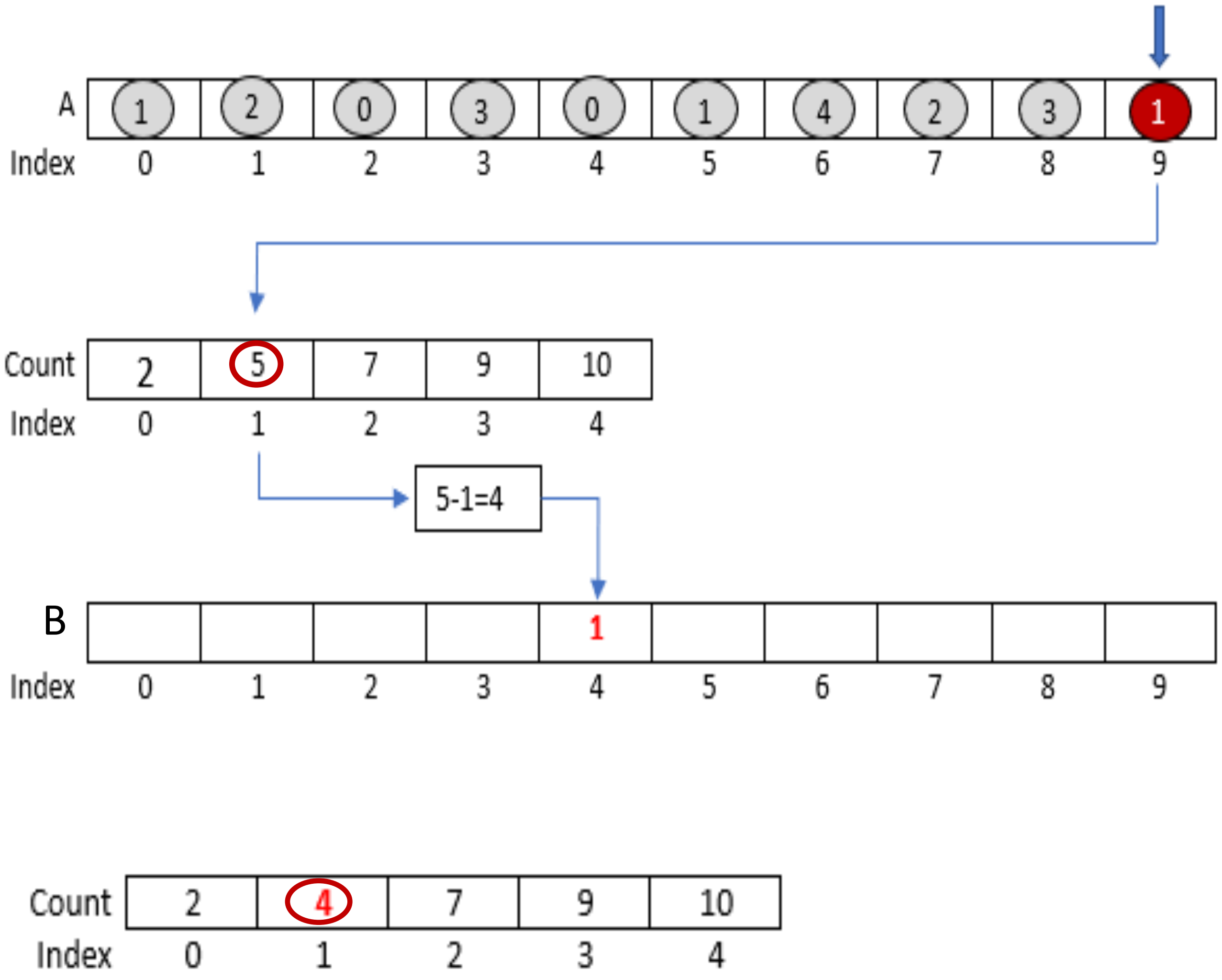
8. Move B to A

What is Counting Sort?



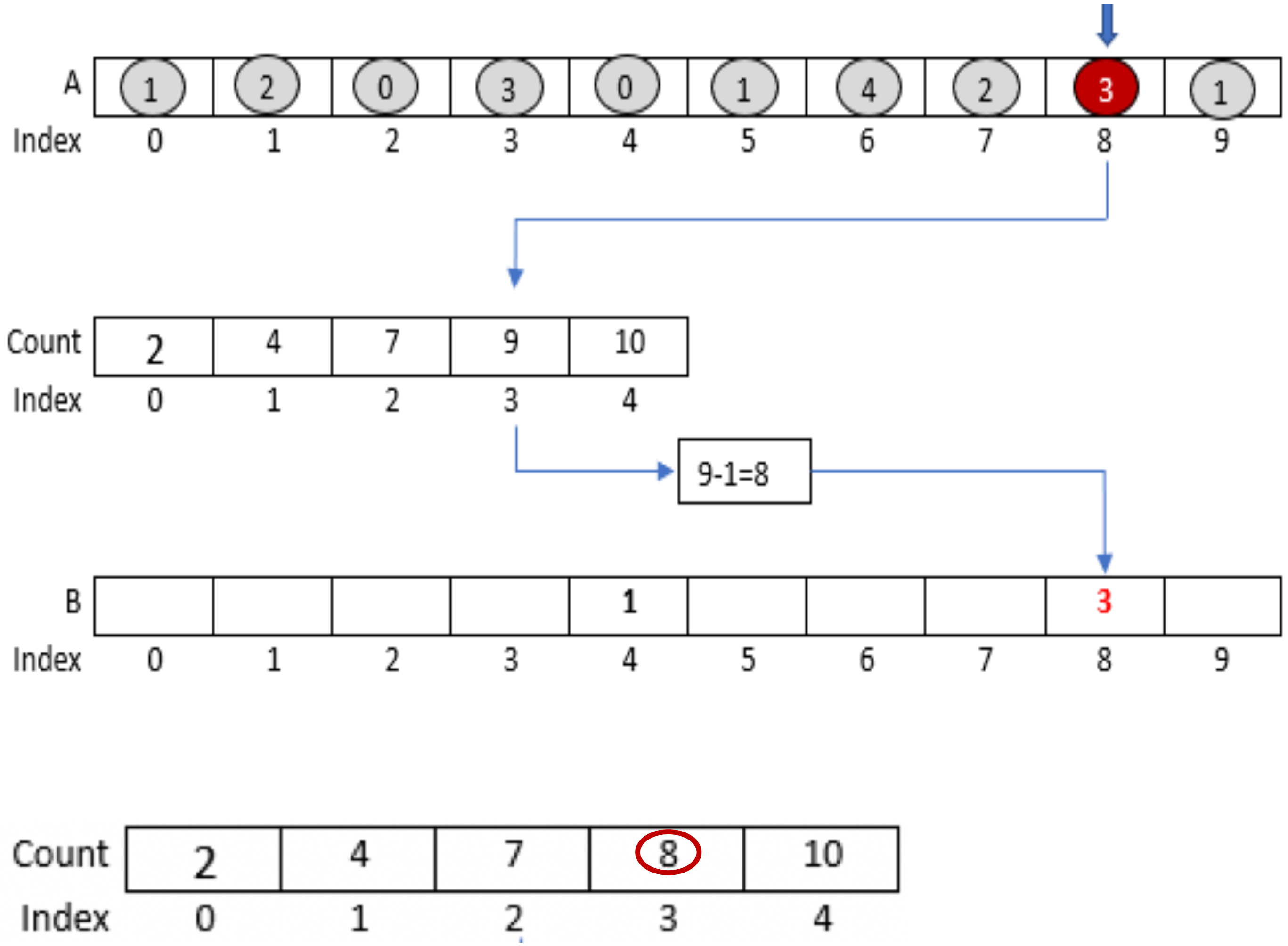
1. Given an array A, construct Count array
2. Convert Count array to cumulative sum array
3. For (i=UB; i>=LB; i--)
4. pos = Count[A[i]]
5. pos = pos – LB – 1
6. **B[pos] = A[i]**
7. Count[A[i]] --
8. Move B to A

What is Counting Sort?




```
1. Given an array A, construct
   Count array

2. Convert Count array to
   cumulative sum array

3. For (i=UB; i>=LB; i--)

4.          pos = Count[ A[i] ]

5.          pos = pos – LB - 1

6.          B[pos] = A[i]

7.          Count[A[i]] --

8. Move B to A
```

What is Counting Sort?



1. Given an array A, construct Count array

2. Convert Count array to cumulative sum array

3. For (i=UB; i>=LB; i--)

4. pos = Count[A[i]]

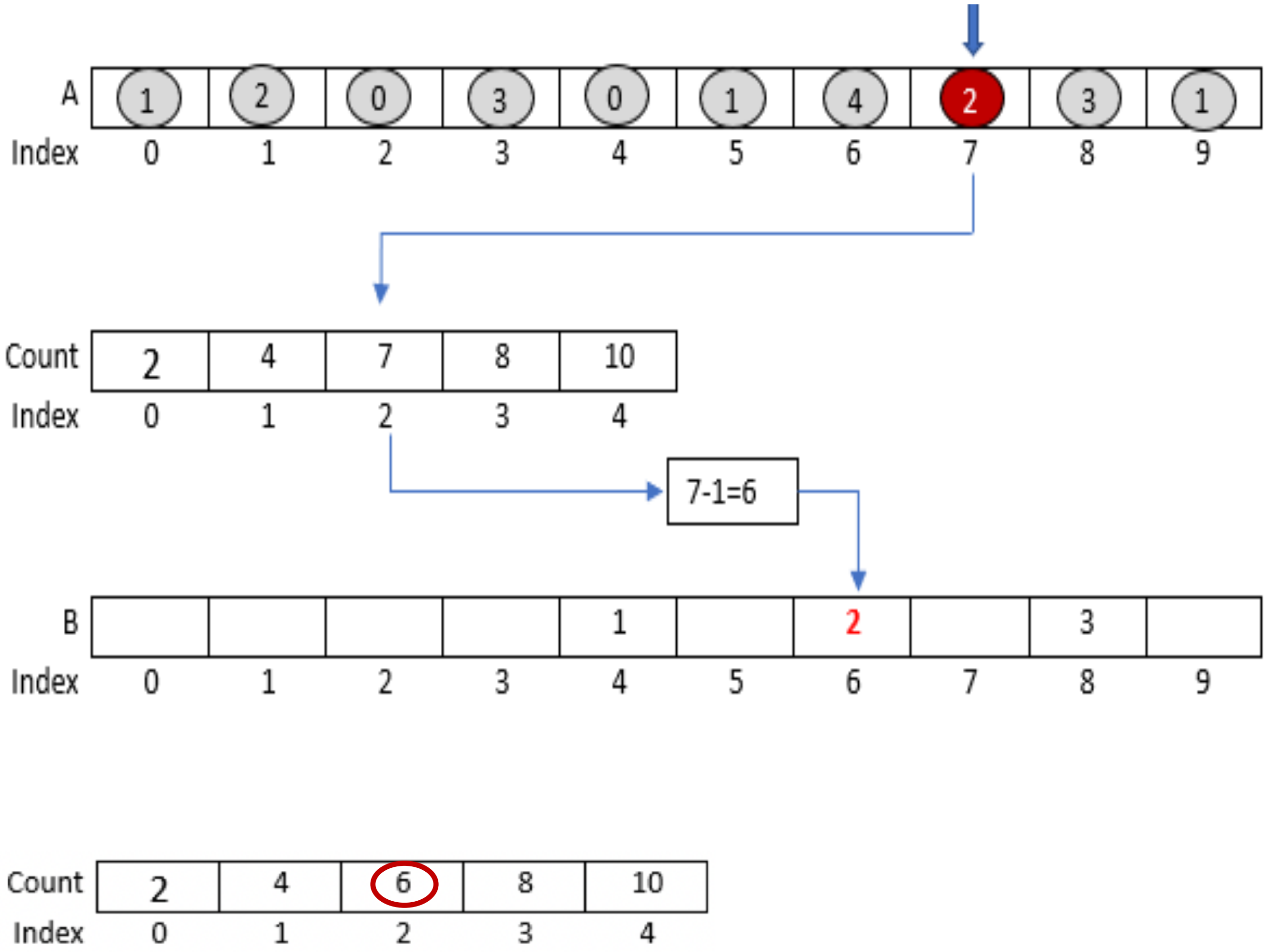
5. pos = pos – LB – 1

6. B[pos] = A[i]

7. Count[A[i]] --

8. Move B to A

What is Counting Sort?



1. Given an array A, construct Count array

2. Convert Count array to cumulative sum array

3. For (i=UB; i>=LB; i--)

4. pos = Count[A[i]]

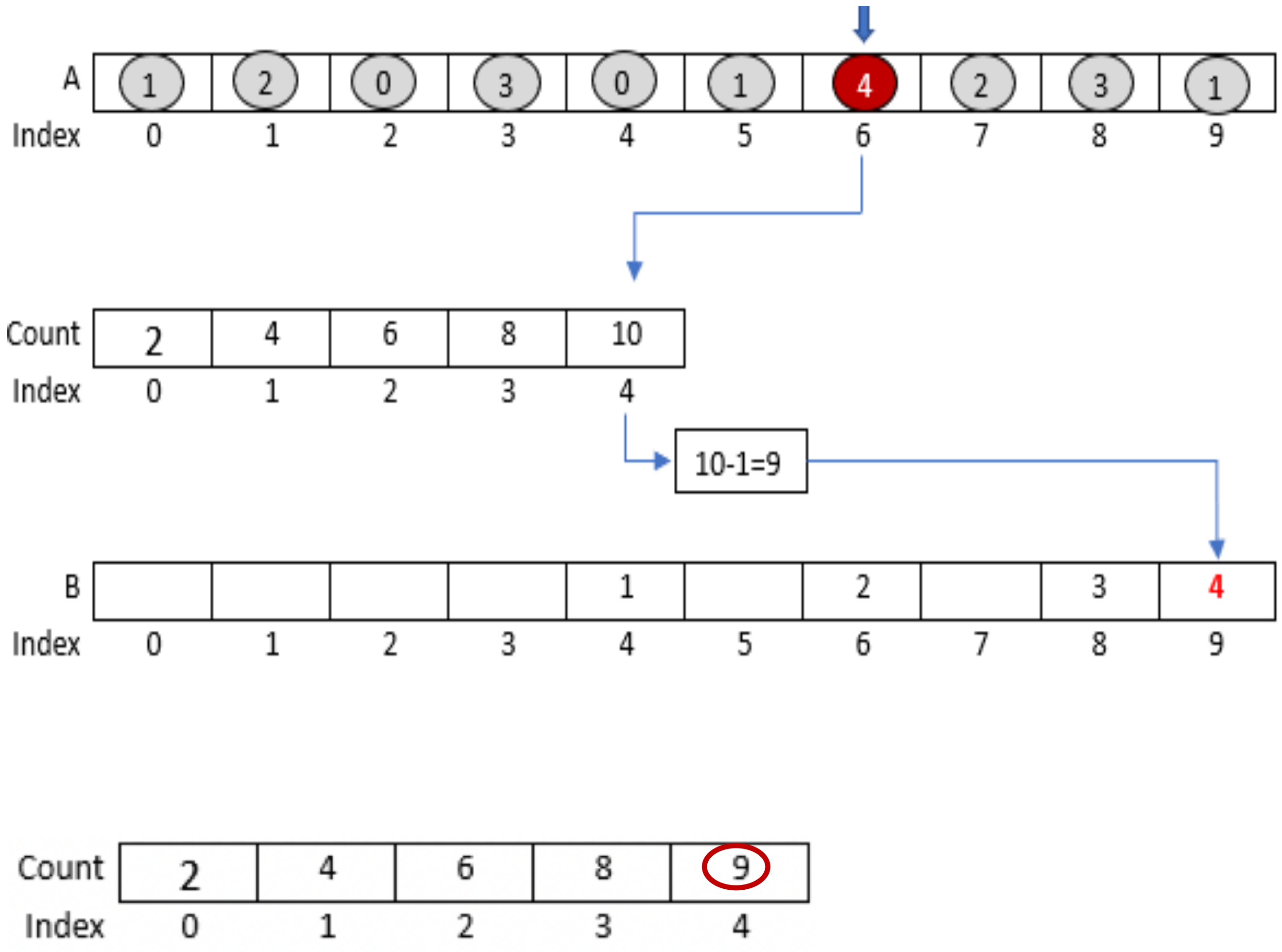
5. pos = pos – LB - 1

6. B[pos] = A[i]

7. Count[A[i]] --

8. Move B to A

What is Counting Sort?



1. Given an array A, construct Count array

2. Convert Count array to cumulative sum array

3. For (i=UB; i>=LB; i--)

4. pos = Count[A[i]]

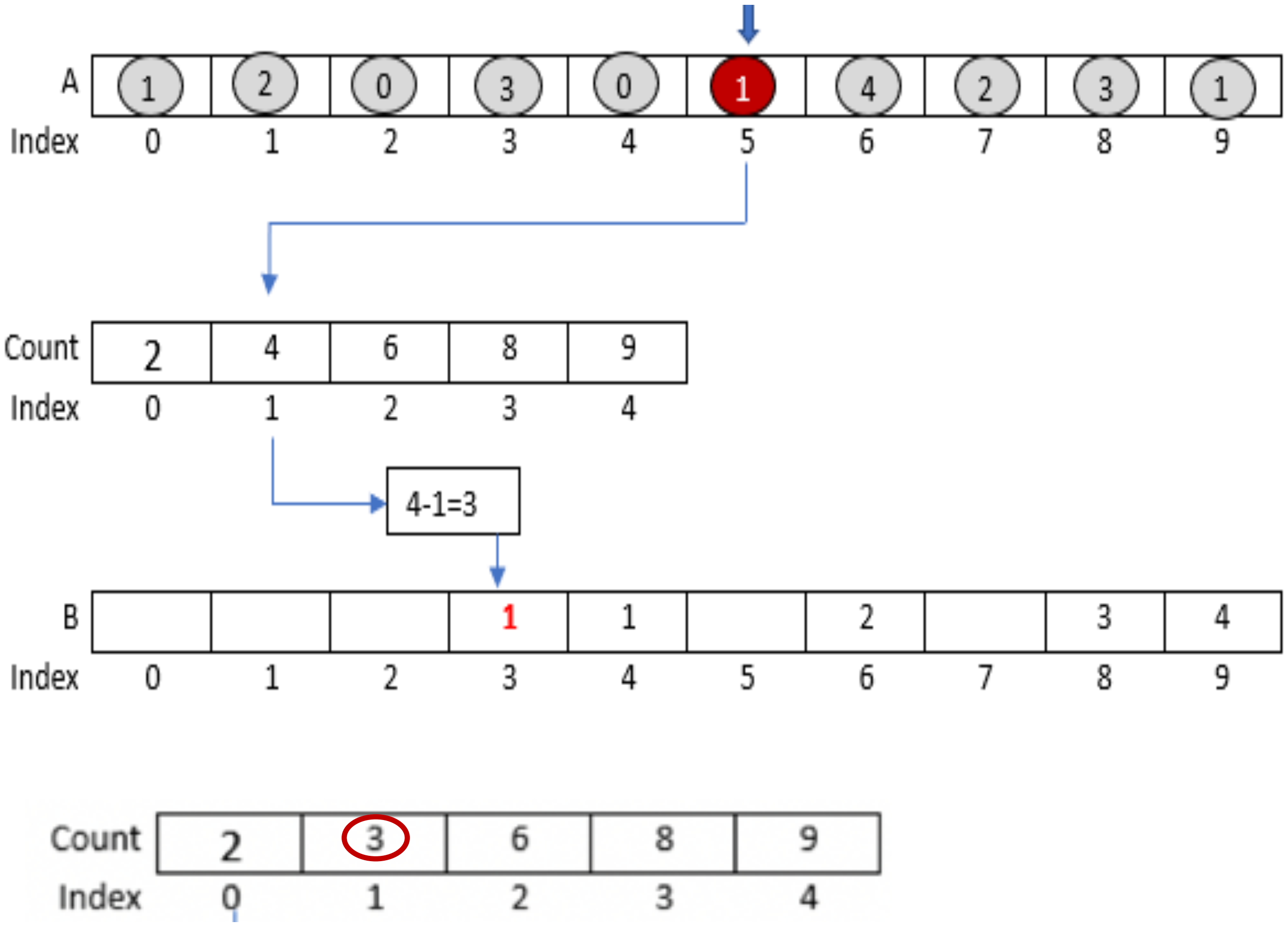
5. pos = pos – LB – 1

6. B[pos] = A[i]

7. Count[A[i]] --

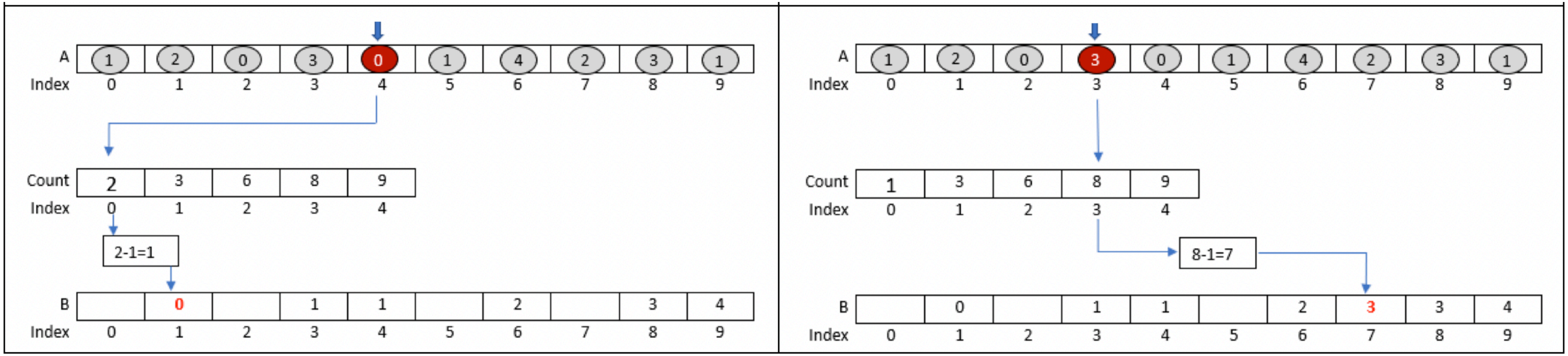
8. Move B to A

What is Counting Sort?

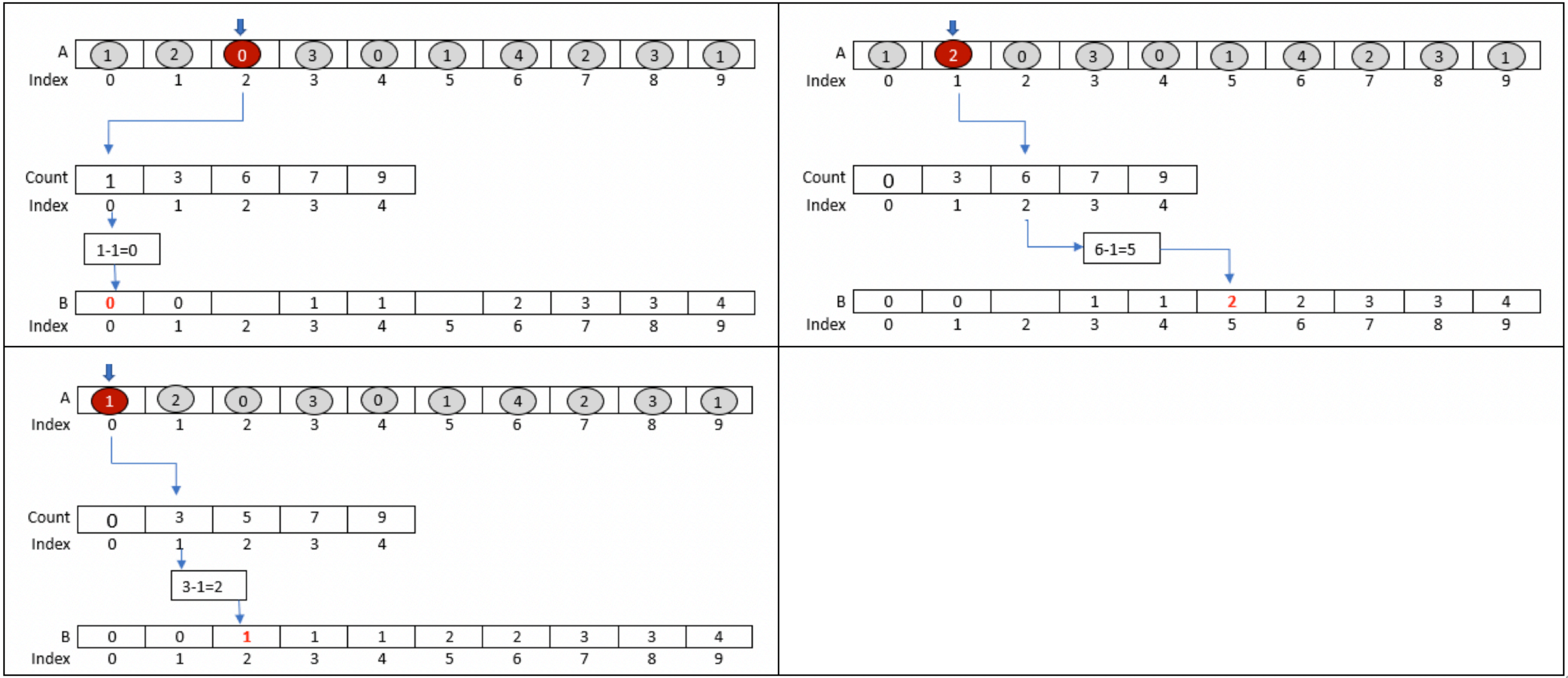


1. Given an array A, construct Count array
2. Convert Count array to cumulative sum array
3. **For (i=UB; i>=LB; i--)**
4. **pos = Count[A[i]]**
5. **pos = pos – LB – 1**
6. **B[pos] = A[i]**
7. **Count[A[i]] --**
8. Move B to A

What is Counting Sort?



What is Counting Sort?



What is Counting Sort?

Once the sorted elements are obtained in B, copy them to A to obtain the final sorted array.

```
1. Given an array A, construct
   Count array

2. Convert Count array to
   cumulative sum array

3. For (i=UB; i>=LB; i--)

4.         pos = Count[ A[i] ]

5.         pos = pos – LB - 1

6.         B[pos] = A[i]

7.         Count[A[i]] --

8. Move B to A
```

Time Complexity

Input array is scanned thrice

- First for finding the maximum value,
- Second for counting frequency, and
- Third for scan the input array for final sorting.

Time Complexity

Input array is scanned thrice

- First for finding the maximum value,
- Second for counting frequency, and
- Third for scan the input array for final sorting.

Count array is scanned once for estimating cumulative values.

Time complexity is $\theta(n+k)$, where n is the size of the input array and k is the size of Count array.

Radix Sort

Radix Sort

It is also a non-comparable sorting algorithm, where sorting is done by the index of the data elements.

Radix Sort

It is also a non-comparable sorting algorithm, where sorting is done by the index of the data elements.

$312, 20, 87, 881, 402, 7, 100, 243, 68, 524$

It processes the data elements index by index from the least significant index to the most significant index.

Radix Sort

$312, 20, 87, 881, 402, 7, 100, 243, 68, 524$

Data		
3	1	2
	2	0
	8	7
8	8	1
4	0	2
		7
1	0	0
2	4	3
	6	8
5	2	4

Radix Sort

$312, 20, 87, 881, 402, 7, 100, 243, 68, 524$

Data		
3	1	2
	2	0
	8	7
8	8	1
4	0	2
		7
1	0	0
2	4	3
	6	8
5	2	4

Sort by index 0

Place in order of occurrence

Index 0		
	2	0
1	0	0
8	8	1
3	1	2
4	0	2
2	4	3
5	2	4
	7	7
		7
	6	8

Radix Sort

$312, 20, 87, 881, 402, 7, 100, 243, 68, 524$

Data

3	1	2
	2	0
	8	7
8	8	1
4	0	2
		7
1	0	0
2	4	3
	6	8
5	2	4

Sort by index 0

Place in order of occurrence

Index 0

	2	0
1	0	0
8	8	1
3	1	2
4	0	2
2	4	3
5	2	4
	7	7
		7
	6	8

Sort by index 1

Place in order of occurrence

Index 1

1	0	0
4	0	2
		7
3	1	2
	2	0
5	2	4
2	4	3
	6	8
	7	7
8	8	1

Radix Sort

$312, 20, 87, 881, 402, 7, 100, 243, 68, 524$

Data		
3	1	2
	2	0
	8	7
8	8	1
4	0	2
		7
1	0	0
2	4	3
	6	8
5	2	4

Sort by index 0

Place in order of occurrence

Index 0		
	2	0
1	0	0
8	8	1
3	1	2
4	0	2
2	4	3
5	2	4
	8	7
		7
	6	8

Sort by index 1

Place in order of occurrence

Index 1		
1	0	0
4	0	2
		7
3	1	2
	2	0
5	2	4
2	4	3
	6	8
	8	7
8	8	1

Sort by index 2

Place in order of occurrence

Index 2		
		7
	2	0
	6	8
	8	7
1	0	0
2	4	3
3	1	2
4	0	2
5	2	4
8	8	1

Radix Sort

The time complexity of radix sort is $O(dn)$ where n is the number of elements in the array, and d is the number of digits of the largest element.

It is because, sorting of the elements by an index position can be done in linear time i.e., $O(n)$ using a linear time sorting algorithm, say bucket sort.

We need to repeat the process for d times.

So, the time complexity is $O(dn)$.

Bucket Sort

Bucket Sort

It uses a collection of **ordered buckets** which can hold data within **a defined range**.

The order and ranges of the bucket can be defined based on the elements to be sorted

Bucket Sort

It uses a collection of **ordered buckets** which can hold data within **a defined range**.

The order and ranges of the bucket can be defined based on the elements to be sorted

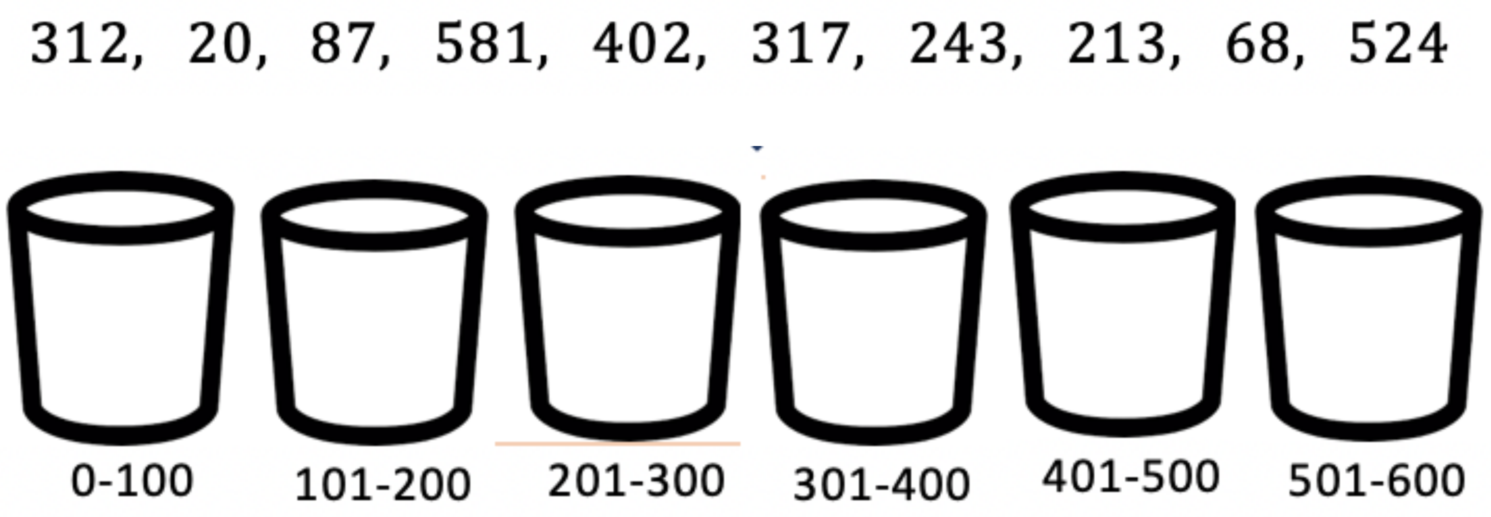
312, 20, 87, 581, 402, 317, 243, 213, 68, 524

Bucket Sort - Algorithm

1. The data elements are distributed into different ordered buckets which can hold data within a defined range.

2. The elements within each bucket are sorted using another sorting algorithm.

3. The sorted list within each of the ordered buckets are simply concatenated to get a bigger sorted list.

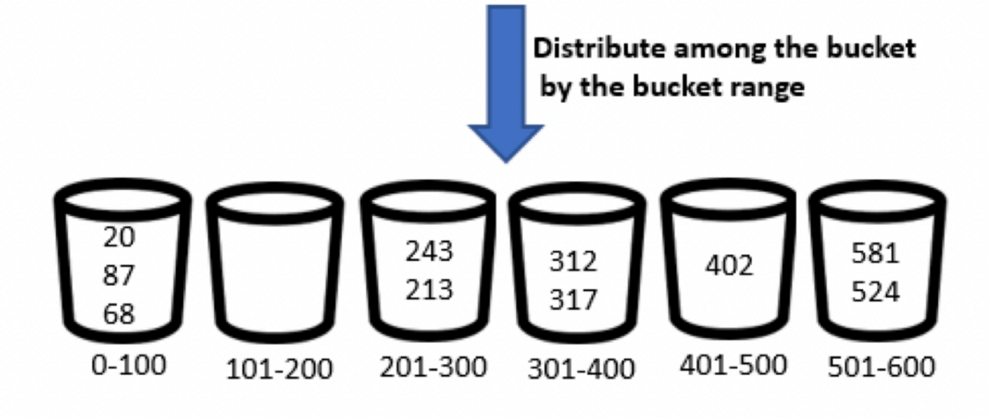


Bucket Sort - Algorithm

312, 20, 87, 581, 402, 317, 243, 213, 68, 524

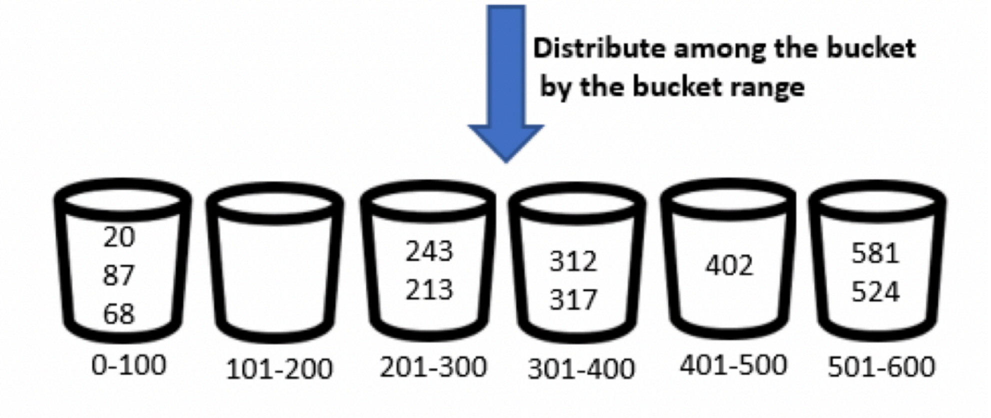
Bucket Sort - Algorithm

312, 20, 87, 581, 402, 317, 243, 213, 68, 524



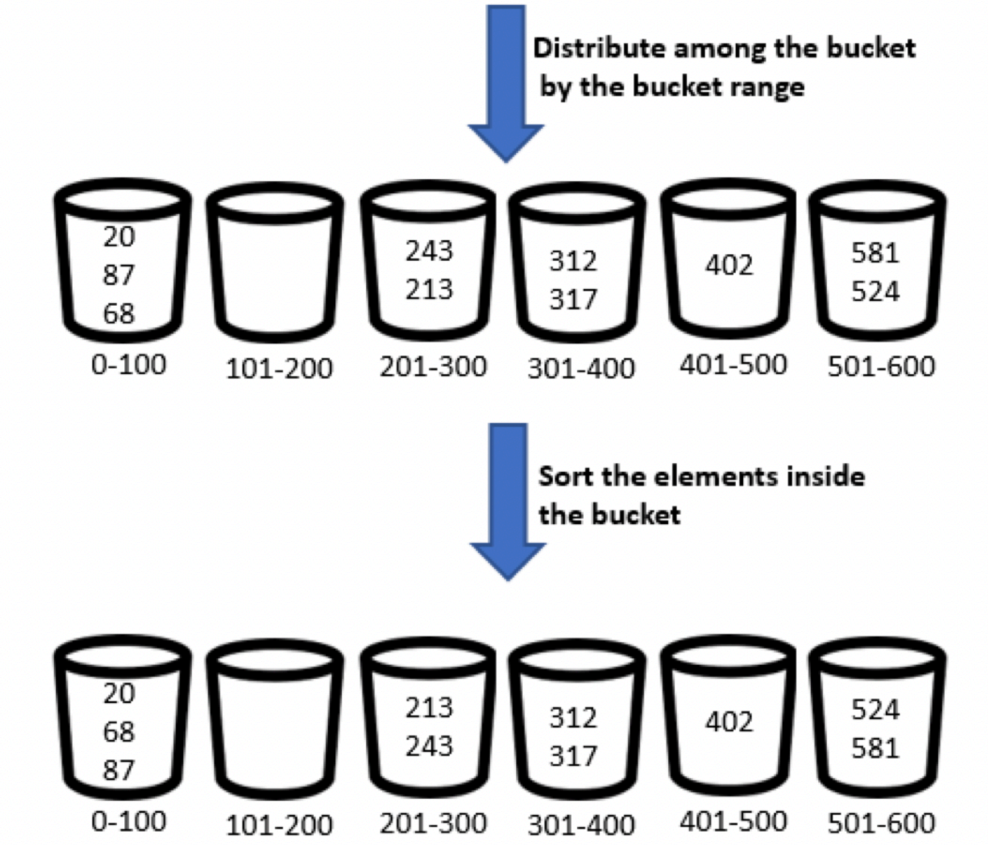
Bucket Sort - Algorithm

312, 20, 87, 581, 402, 317, 243, 213, 68, 524

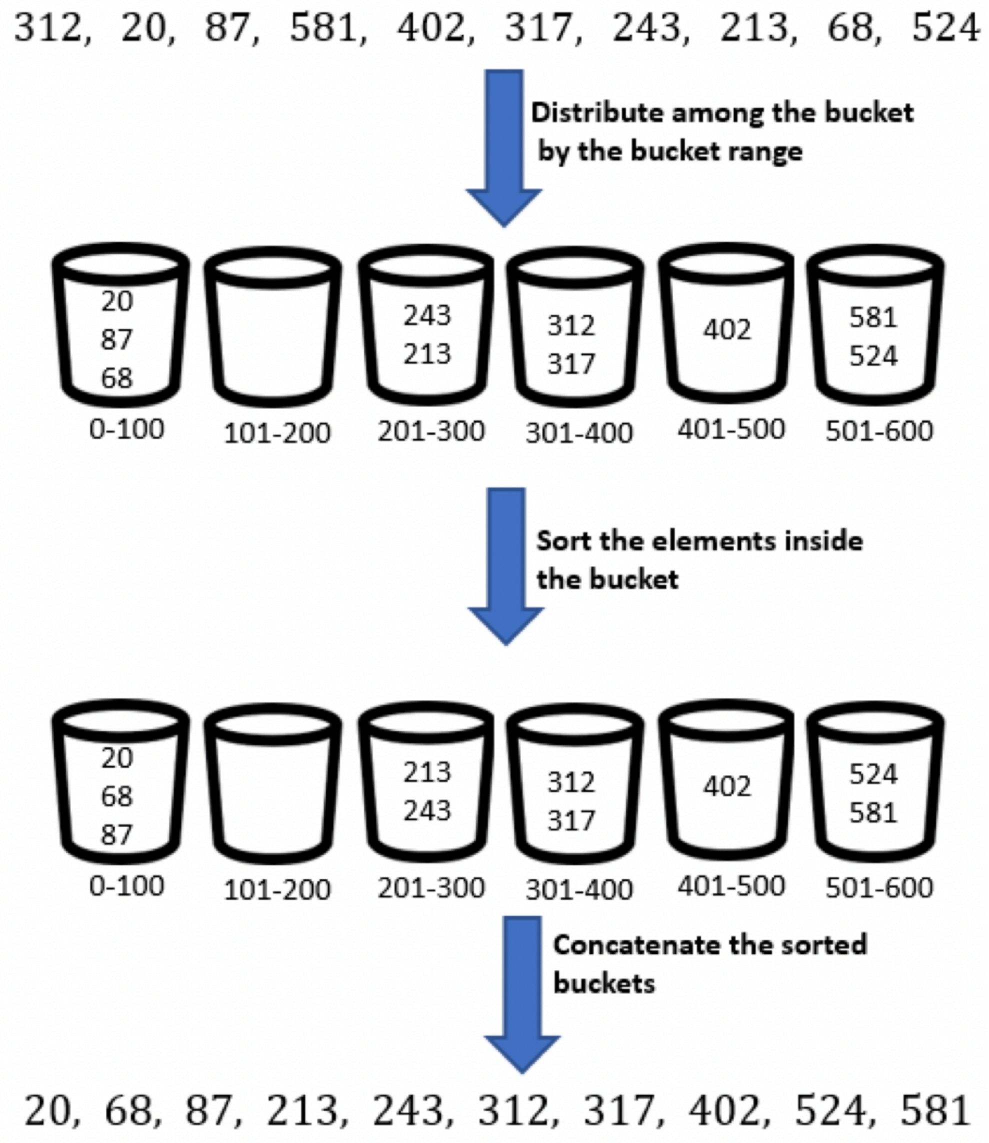


Bucket Sort - Algorithm

312, 20, 87, 581, 402, 317, 243, 213, 68, 524

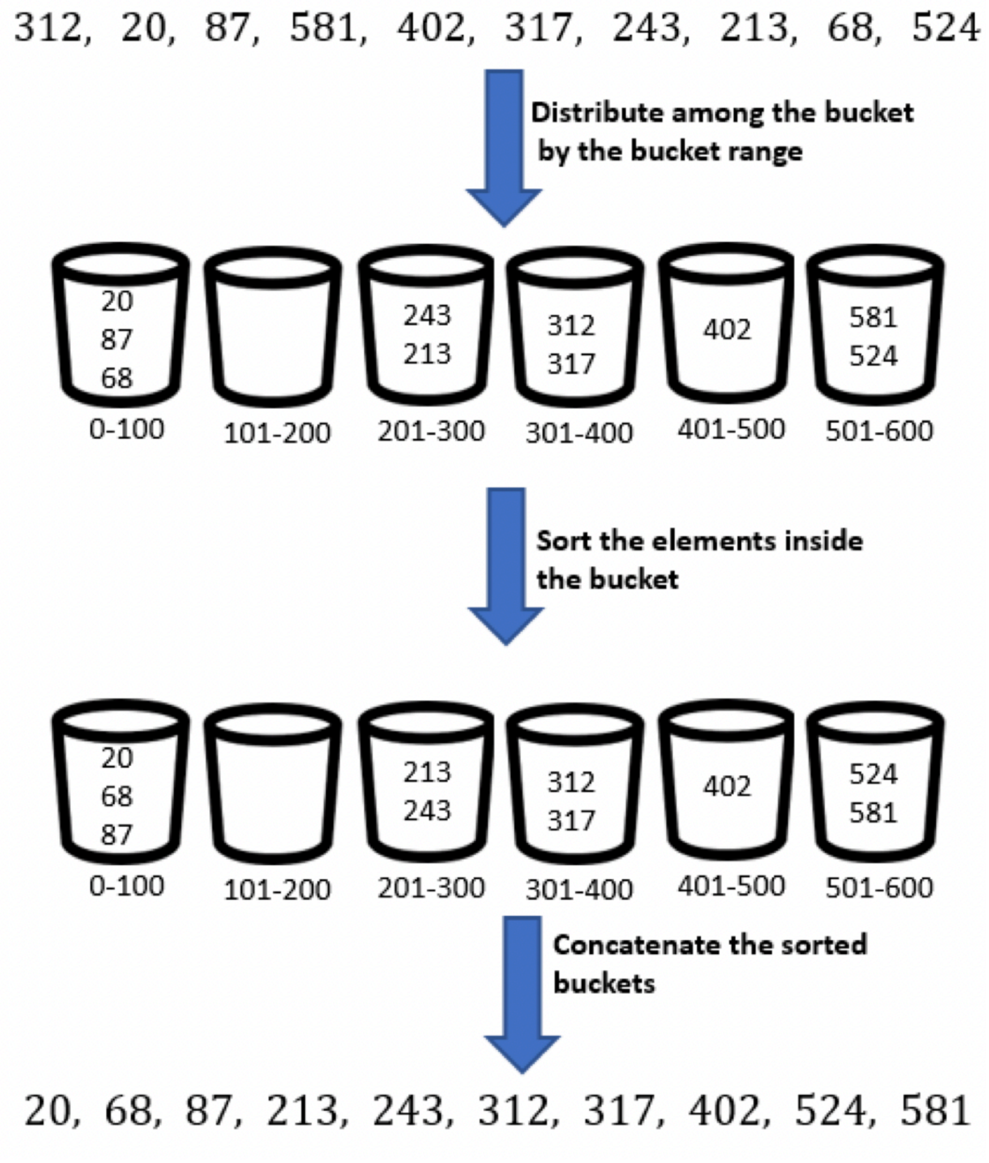


Bucket Sort - Algorithm



Some of the buckets may be empty or may hold very few data elements.

Bucket Sort - Algorithm

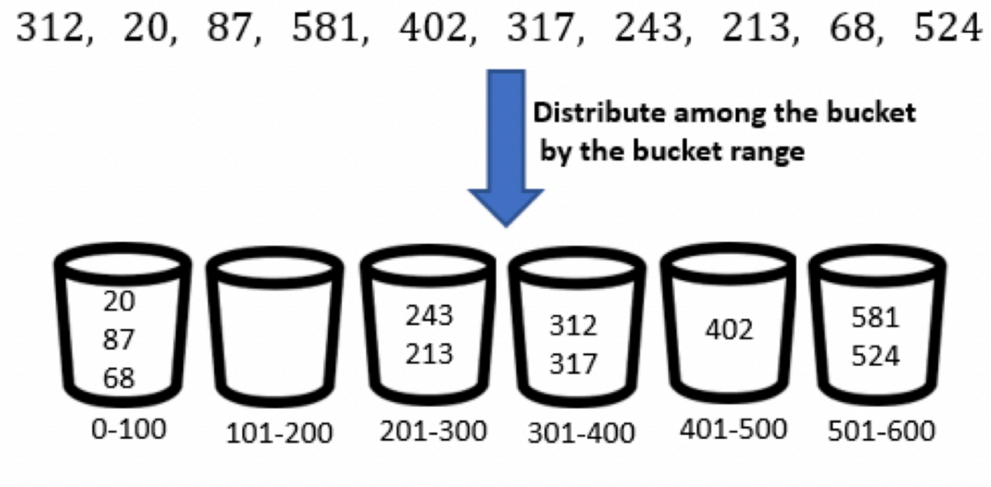


Some of the buckets may be empty or may hold very few data elements.

Time Complexity may depends on several factors such as

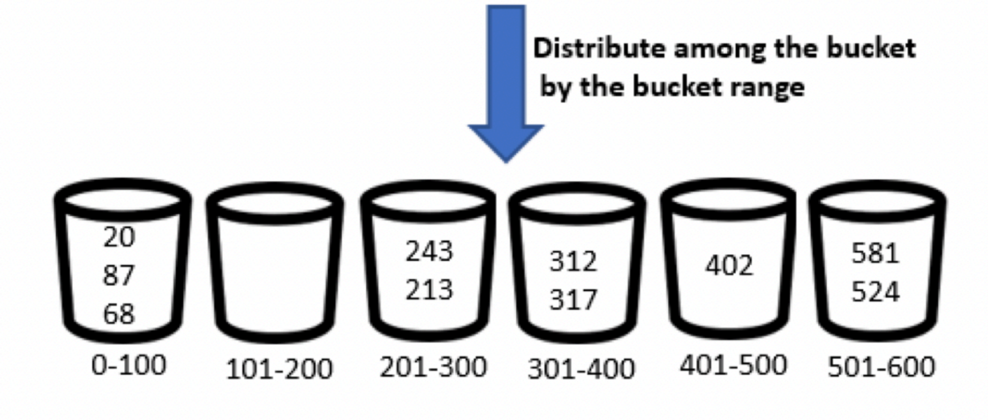
- distribution of the elements in the buckets
- Algorithms used for sorting the elements in the buckets
- Number of buckets etc.

Bucket Sort - Algorithm



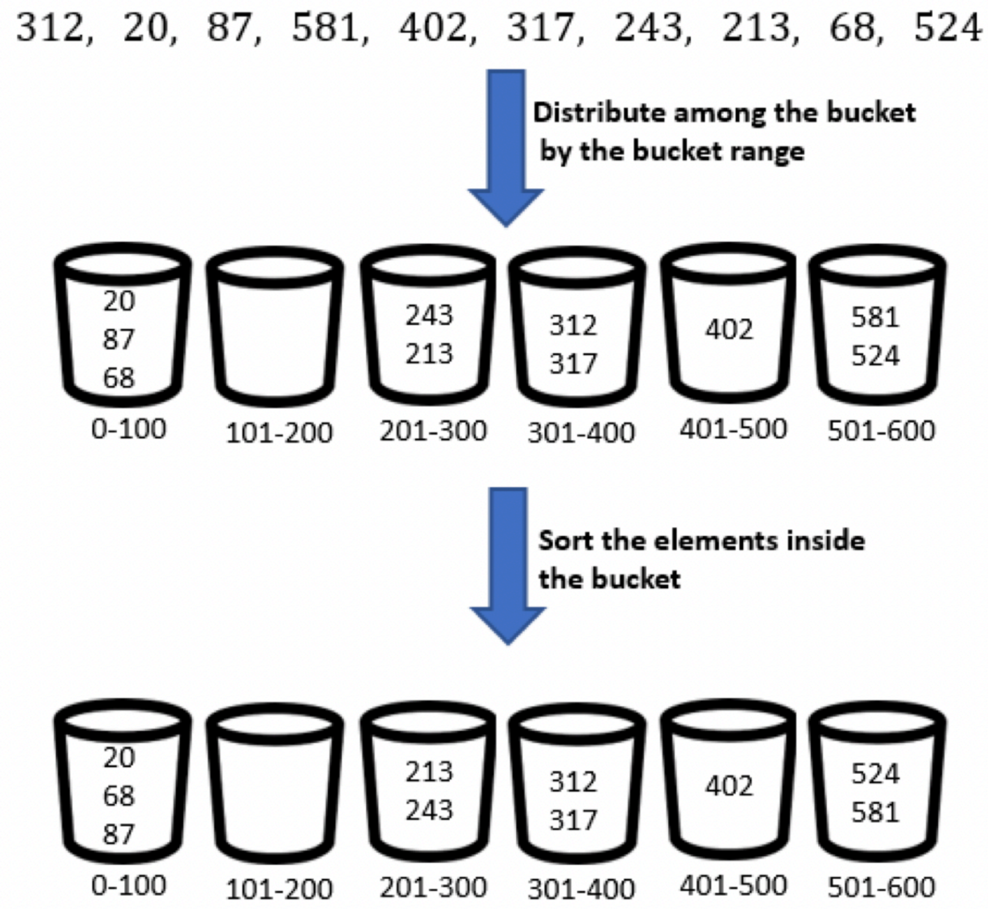
Bucket Sort - Algorithm

312, 20, 87, 581, 402, 317, 243, 213, 68, 524



It can be done in $\theta(n)$

Bucket Sort - Algorithm



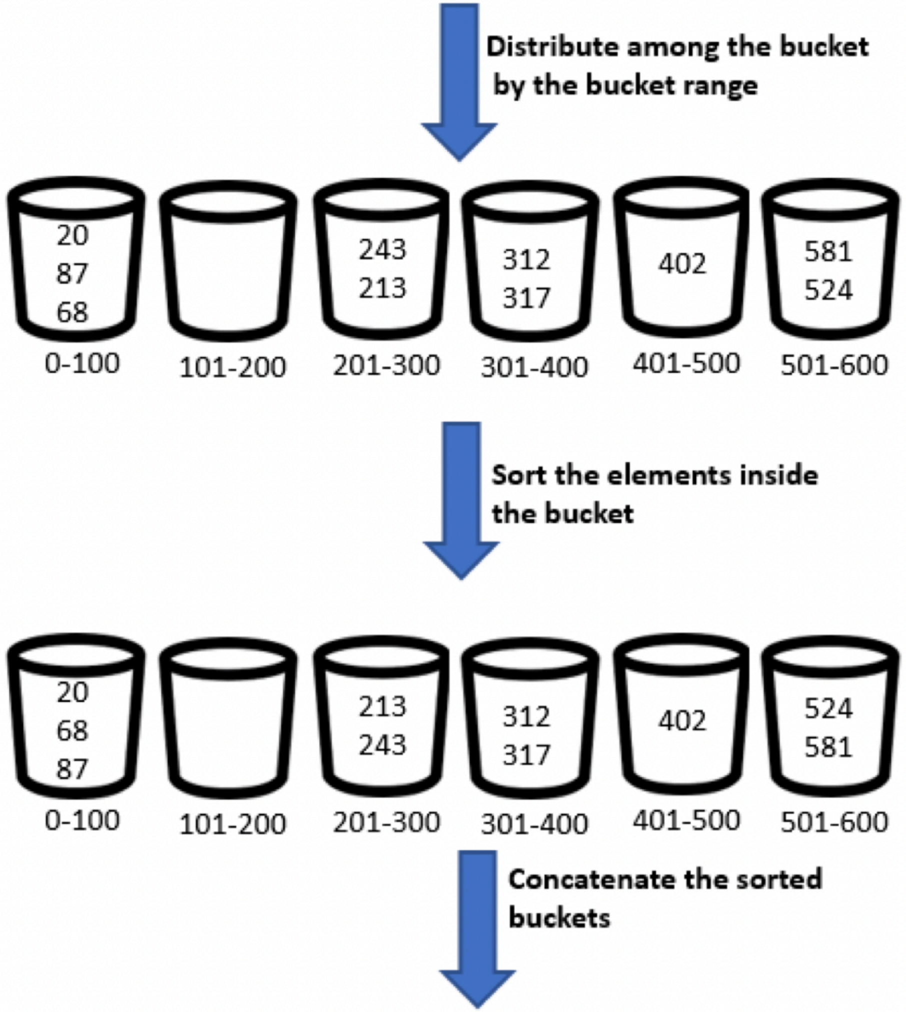
If we assume that the elements are uniformly distributed,

it can be done in $\theta\left(k\frac{n}{k}\right) = \theta(n)$

where k is the number of buckets.

Bucket Sort - Algorithm

312, 20, 87, 581, 402, 317, 243, 213, 68, 524



20, 68, 87, 213, 243, 312, 317, 402, 524, 581

It can be done in $\theta(n)$