Data Structure

Prof. Vijya Tulsani, Assistant Professor
IT and Computer Science

Parul® University

DIGITAL LEARNING CONTENT

CHAPTER-4

Sorting , Searching and File Structure

What is Sorting?

- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.
- Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order.
- Sorting arranges data in a sequence which makes searching easier.

Complexity of Sorting algorithm

- The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted.
- The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems.
- The most noteworthy of these considerations are:
- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program
- The amount of memory necessary for running the program

Types of Sorting

- Selection Sort
- Bubble sort
- Insertion sort
- Quick Sort
- Shell Sort
- Merge Sort etc.

Image source : Youtube video

Selection Sort

- In selection sort the list is divided into two sub-lists sorted and unsorted.
- These two lists are divided by imaginary wall.
- We find a smallest element from unsorted sub-list and swap it to the beginning. And the wall moves one element ahead, as the sorted list is increases and unsorted list is decreases.

Selection Sort

- Assume that we have a list on n elements.
- By applying selection sort, the first element is compared with all remaining (n-1) elements. The smallest element is placed at the first location.
- Again, the second element is compared with remaining (n-1) elements.
- At the time of comparison, the smaller element is swapped with larger element. Similarly, entire array is checked for smallest element and then swapping is done accordingly.
- Here we need n-1 passes or iterations to completely rearrange the data.

Image source : Google

Algorithm Steps

- Pass1: Find the Location LOC of the smallest in the List N elements A[0],A[1]….A[N], and then interchange A[LOC] and A[0].Then A[0] is Sorted.

- Pass 2: Find the Location LOC of the smallest in the List N-1elements A[1],A[2]….A[N], and then interchange A[LOC] and A[1]. Then A[0],A[1] is Sorted. Since A[0]<=A[1].

- Pass N-1: Find the Location LOC of the smallest in the List A[N-1],A[N]and then interchange A[LOC] and A[N-1].Then A[0],A[1]……A[N-1]is Sorted. Since A[N-2]<=A[N-1].

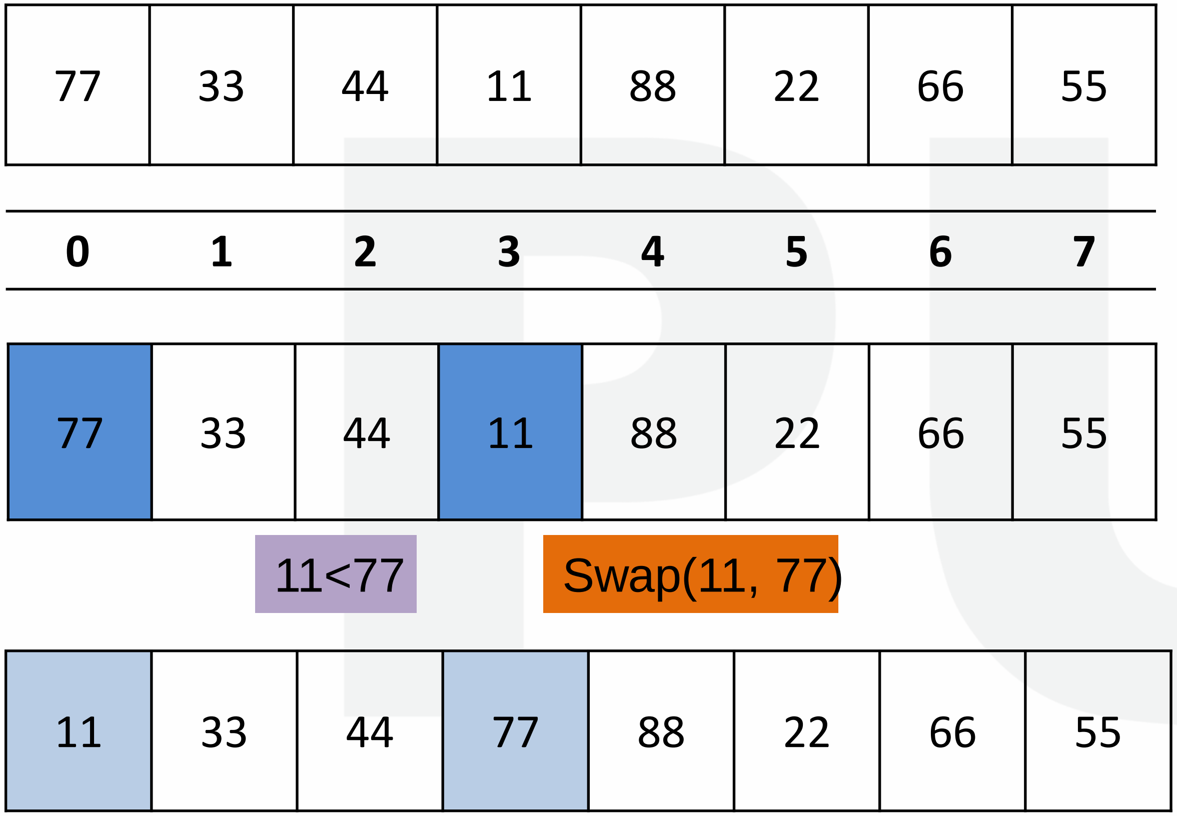
- Thus A is Sorted after N-1 passes**.**

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Example of Selection Sort



```
for (c = 0; c < (n - 1); c++)
{
   min = c;
   for (d = c + 1; d < n; d++)
   {
     if (a[min] > array[d])
       min = d;
   }
  interchange(a[min],a[c]);
}
```

Image source : Google

Example of Selection Sort

11	33	44	77	88	22	66	55

0	1	2	3	4	5	6	7
11	33	44	77	88	22	66	55

33<22

Swap(33, 22)

11	22	44	77	88	33	66	55

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Example of Selection Sort

11	22	44	77	88	33	66	55

0	1	2	3	4	5	6	7
11	22	44	77	88	33	66	55

44<33

Swap(44, 33)

11	22	33	77	88	44	66	55

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Example of Selection Sort

11	22	33	77	88	44	66	55

0	1	2	3	4	5	6	7
11	22	33	77	88	44	66	55

77<44

Swap(77, 44)

11	22	33	44	88	77	66	55

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Example of Selection Sort

11	22	33	44	88	77	66	55

0	1	2	3	4	5	6	7

11	22	33	44	88	77	66	55

88<55

Swap(88, 55)

11	22	33	44	55	77	66	88

Image source : Google

Example of Selection Sort

11	22	33	44	55	77	66	88

0	1	2	3	4	5	6	7
11	22	33	44	55	77	66	88

77<66

Swap(77, 66)

11	22	33	44	55	66	77	88

Image source : Google

Example of Selection Sort

Sorted List

11	22	33	44	55	66	77	88

Image source : Google

Algorithm

```
Selection_sort(A,n)
{
For i=0 to n-2
imin=i;
For j=i+1 to n-1
{
if(A[imin]>A[j])
imin=j;
}
Swap(A[i],A[imin])
}
```

Complexity: O(n2)

Image source : Youtube video

Code

```
#include <stdio.h>
int main()
{
  int array[100], n, c, d, position, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0; c < (n - 1); c++) // finding minimum element (n-1) times
  {
    position = c;
    for (d = c + 1; d < n; d++)
    {
      if (array[position] > array[d])
        position = d;
    }
    if (position != c)
    {
      t = array[c];
      array[c] = array[position];
      array[position] = t;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```

Image source : Youtube video

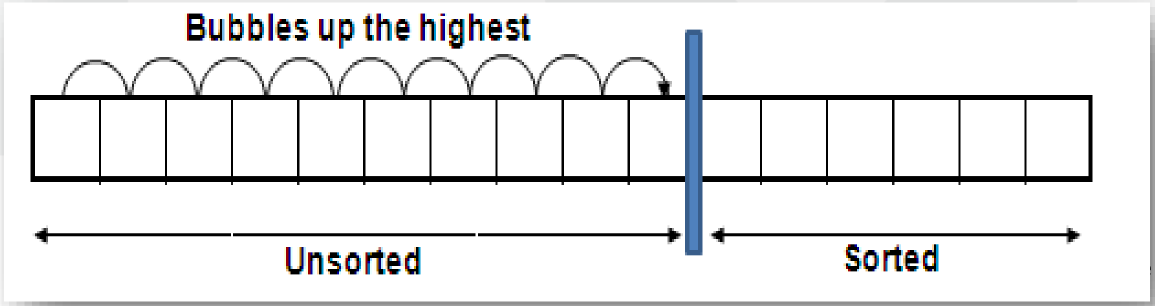
Bubble Sort

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Image source : Youtube video

Bubble Sort

- In bubble sort method the list is divided into two sub-lists sorted and unsorted.
- The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead.
- The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled.
- This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done.
- Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.



: Youtube video

Example

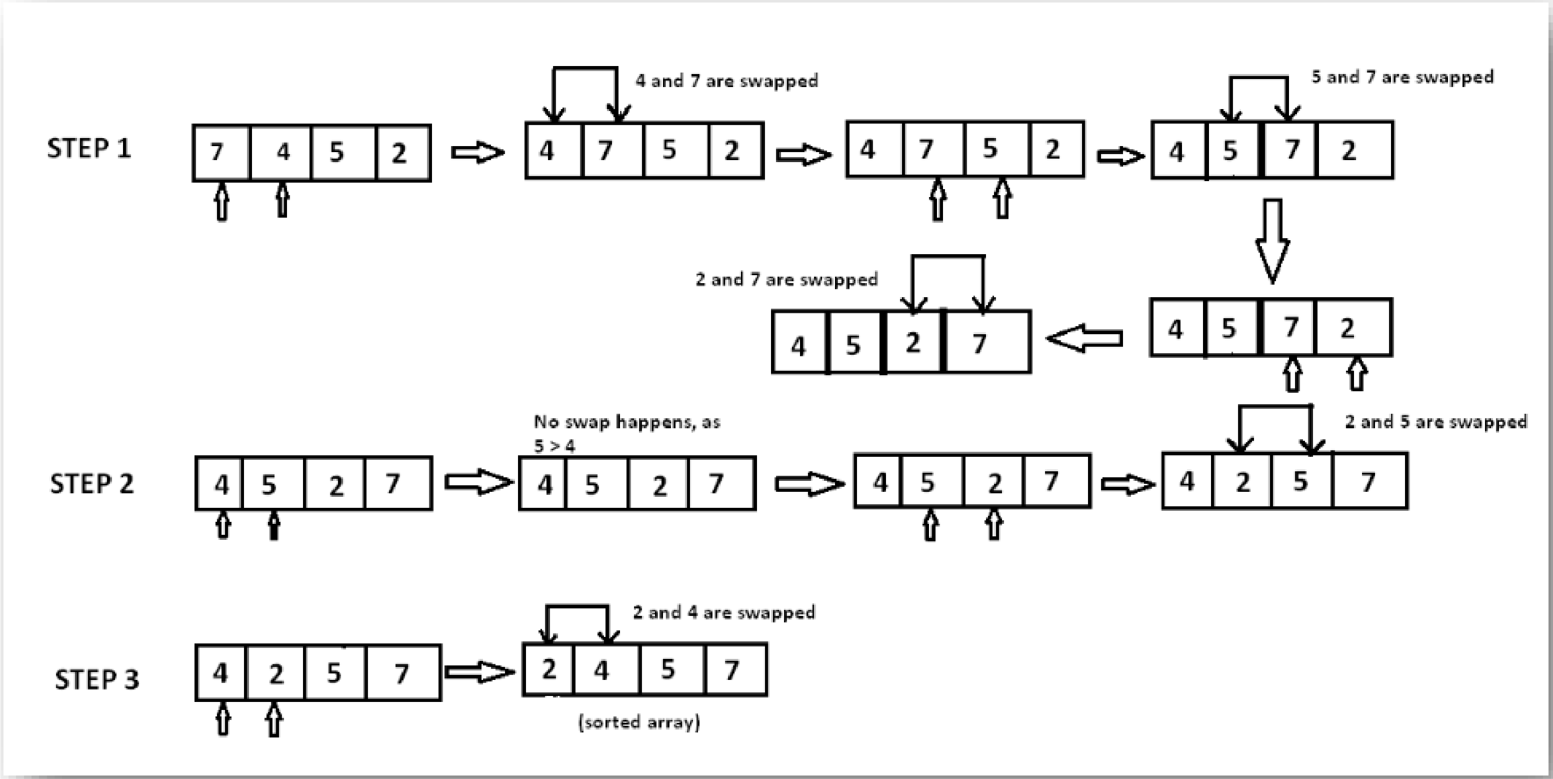


Image source : Youtube video

Bubble sort algorithm

- Algorithm for Bubble Sort:

```
Bubble_Sort ( A [ ] , N )
Step 1 : Repeat For i = 0 to N – 1 Begin
Step 2 : Repeat For J = 1 to N – i - 1 Begin
Step 3 : If ( A [ J ] < A [ J – 1 ] )
Swap ( A [ J ] , A [ J – 1 ] ) End For
End For
Step 4 : Exit
```

Image source : Youtube video

Code

```
# Bubble Sort
#include <stdio.h>
int main()
{
  int a[100], n, c, d, position, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
  for (i = 0; i <= n-1; i++)
// finding maximum element (n-1) times
  {
      for (j =1; j <= n – i - 1; j++)
     {
      if (a[j] < a[j-1])
       {
           t = a[j];
           a[j] = a[j-1];
           a[j-1] = t;
       }
     }
   }

printf("Sorted list in ascending order:\n");
  for (i = 0; i < n; i++)
    printf("%d\n", a[i]);
  return 0;
}
```

Image source : Youtube video

Insertion Sort

- In insertion sort the element is inserted at an appropriate place similar to card insertion.
- Here the list is divided into two parts sorted and unsorted sub-lists.
- n each pass, the first element of unsorted sub list is picked up and moved into the sorted sub list by inserting it in suitable position.
- Suppose we have ‘n’ elements, we need n-1 passes to sort the elements.

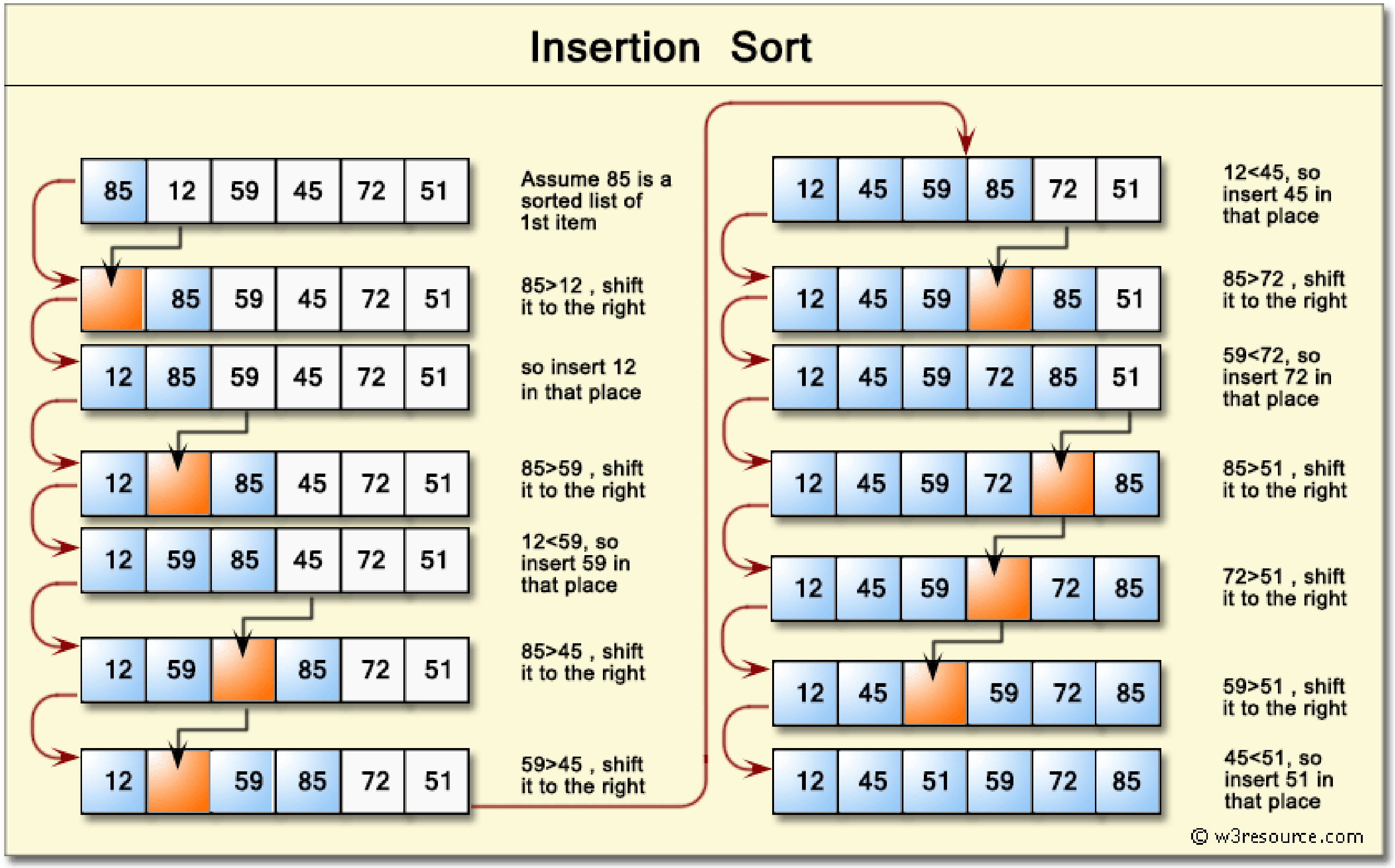
Image source : Youtube video

Algorithm Steps

- Pass 1: A[0] by itself is trivially sorted.
- Pass 2: A[1] is inserted either before or after A[0] so that; A[0], A[1] is Sorted.
- Pass 3: A[2] is inserted into its proper place in A[0], A[1] that is before A[0] , between A[0] and A[1], or after A[1], So that A[0], A[1], A[2] is sorted.
- Pass N: A[N-1] is inserted into the proper place in A[0], A[1], A[2]....A[N-1] is Sorted.

Image source : Youtube video

Insertion Sort


Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

ıbe video

Parul® University

DIGITAL LEARNING CONTENT

Example

78	23	45	8	32	36

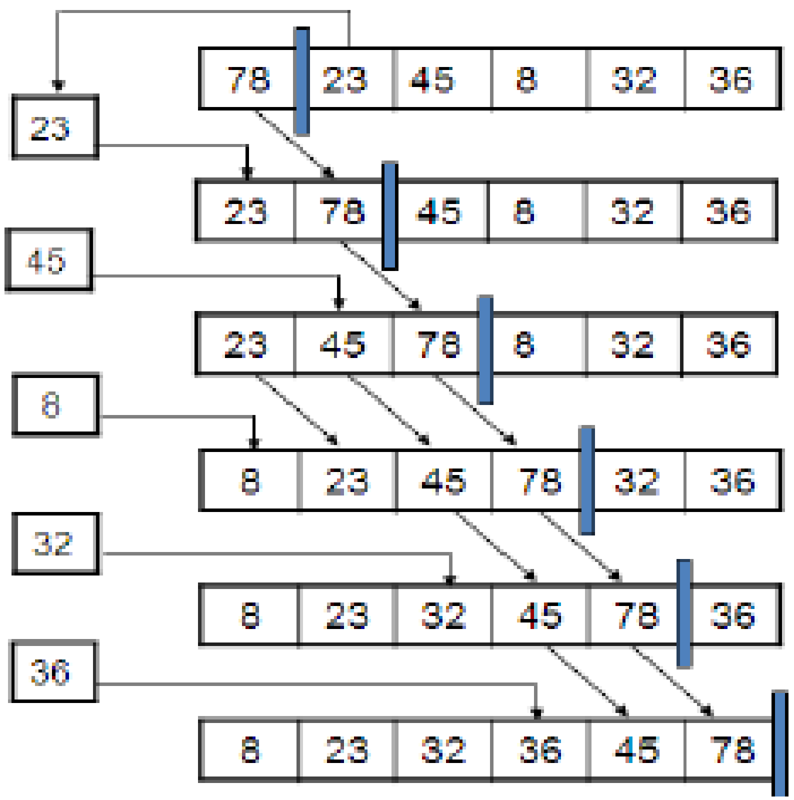


Image source : Youtube video

Example

5	2	4	6	1	3

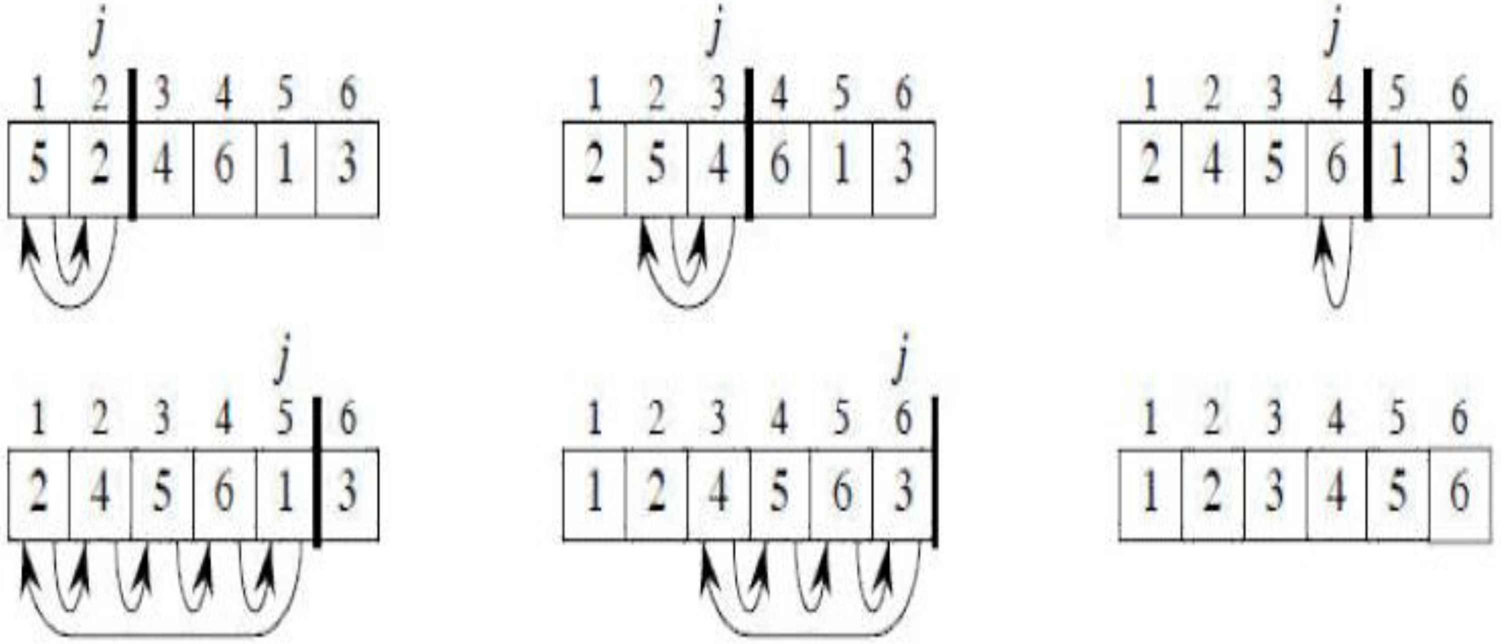


Image source : Youtube video

Types of Sorting

```
Insertion_sort(A, n)
{
For i=1 to n-1
{
Value= A[i]
Key=i
While(key>=0 && A[key-1]>value)
{
A[key]=A[key-1]
Key=key-1
}
A[key]=value
}
```

Complexity: O(n2)

Image source : Youtube video

C program for insertion sort

```
#include <math.h>
#include <stdio.h>
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        /* Move elements of arr[0..i-1], that are
          greater than key, to one position
ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```
// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
/* Driver program to test insertion sort */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    printArray(arr, n);
    return 0;
}
```

Image source : Youtube video

Quick Sort

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Image source : Youtube video

Quick Sort

- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Image source : Youtube video

Quick Sort Pivot Algorithm

Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point left and right of the list excluding pivot
Step 3 – left points to the low index
Step 4 – right points to the high
Step 5 – while value at left is less than pivot move right
Step 6 – while value at right is greater than pivot move left
Step 7 – if both step 5 and step 6 does not match swap left and right
Step 8 – if left ≥ right, the point where they met is new pivot

Image source : Youtube video

Pseudo code

```
1. While data[left] <= data[pivot]
     ++left
2. While data[right] > data[pivot]
     --right
3. If left < right
     swap data[left] and data[right]
4. While left < right, go to 1.
5. Swap data[right] and data[pivot]
```

Image source : Youtube video

Quick Sort

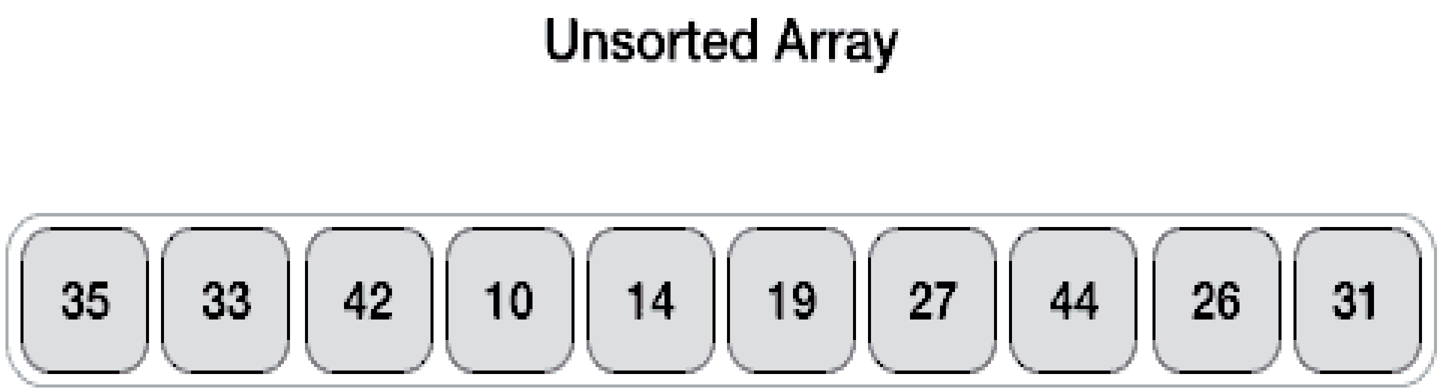


Image source : Youtube video

Quick Sort

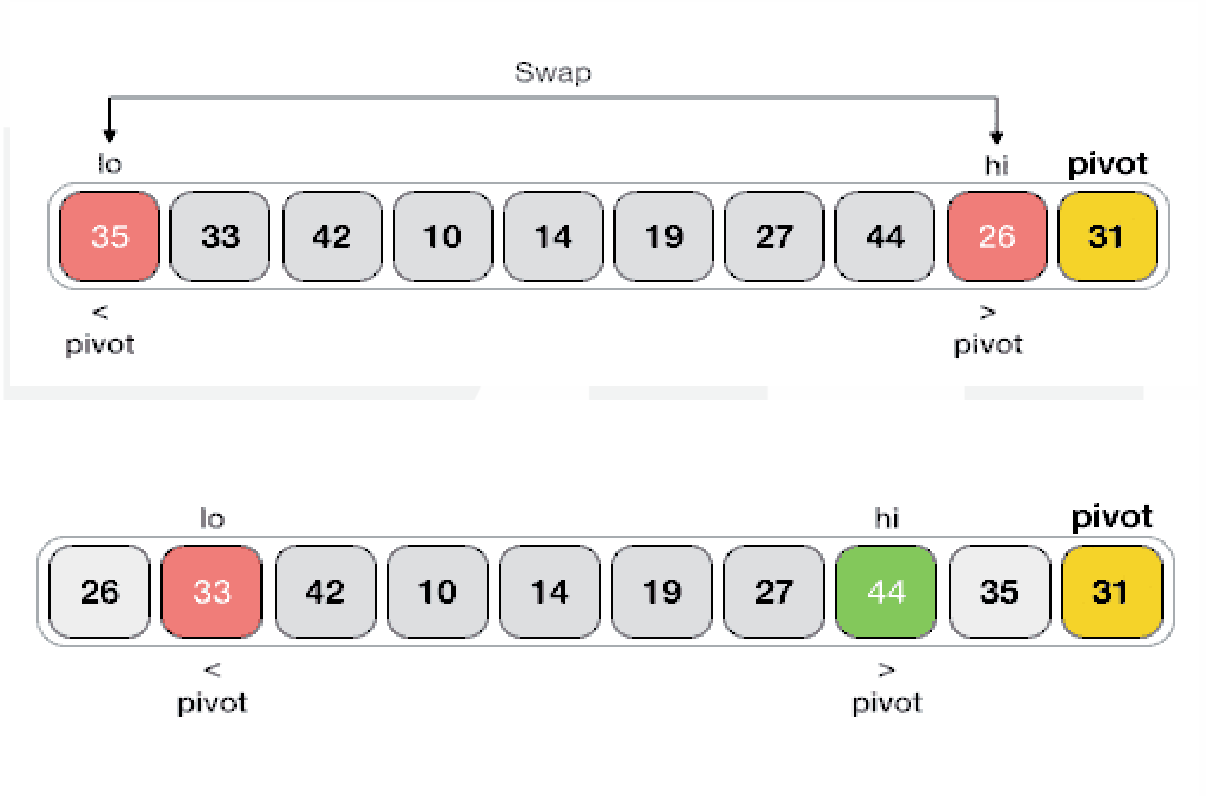
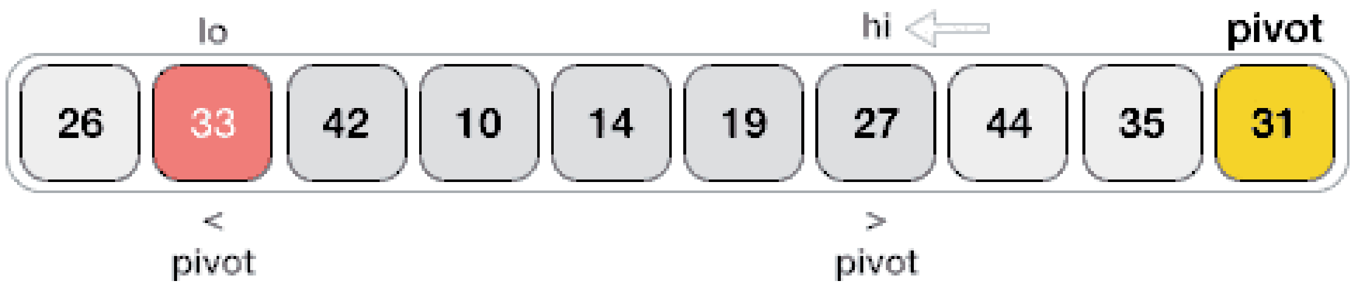
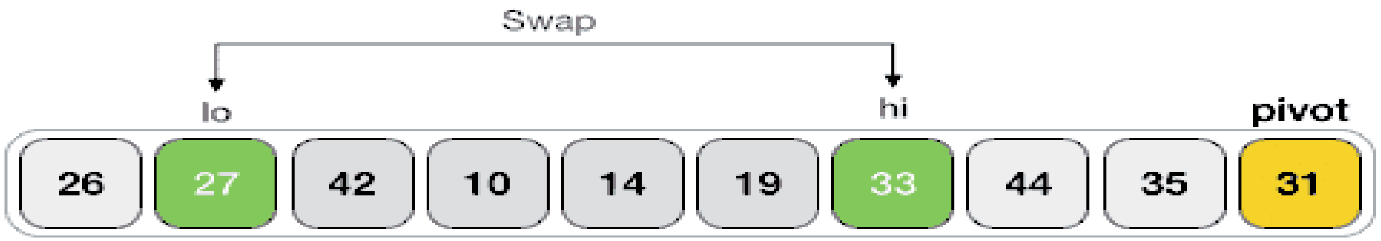


Image source : Youtube video

Quick Sort


lo
hi
pivot
26
33
42
10
14
19
27
44
35
31
<
pivot
>
pivot


Swap
lo
hi
pivot
26
27
42
10
14
19
33
44
35
31

Quick Sort

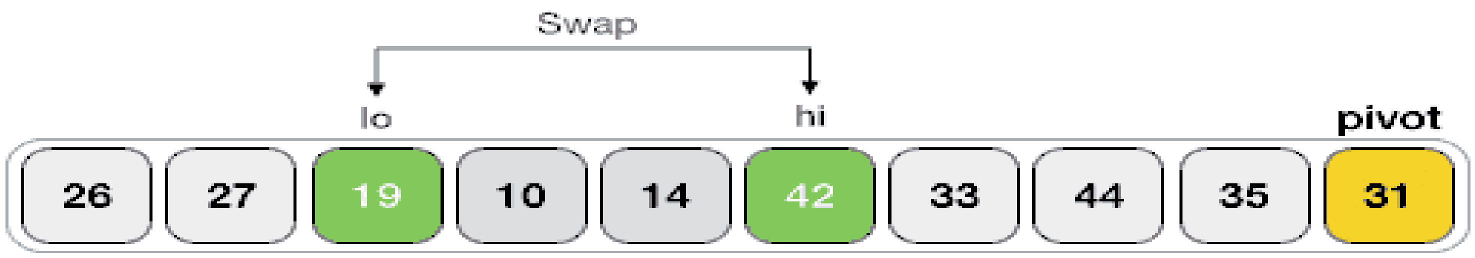
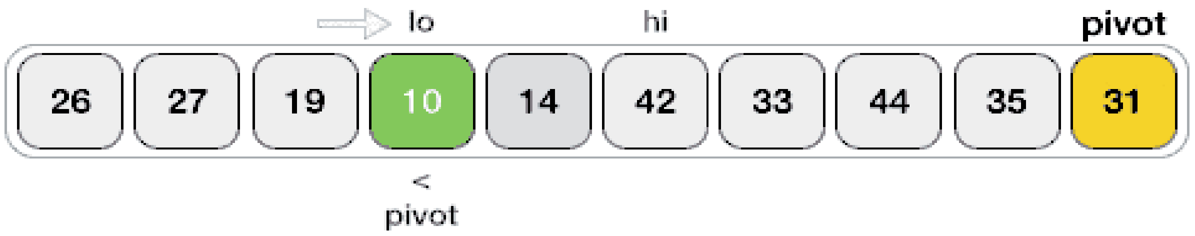


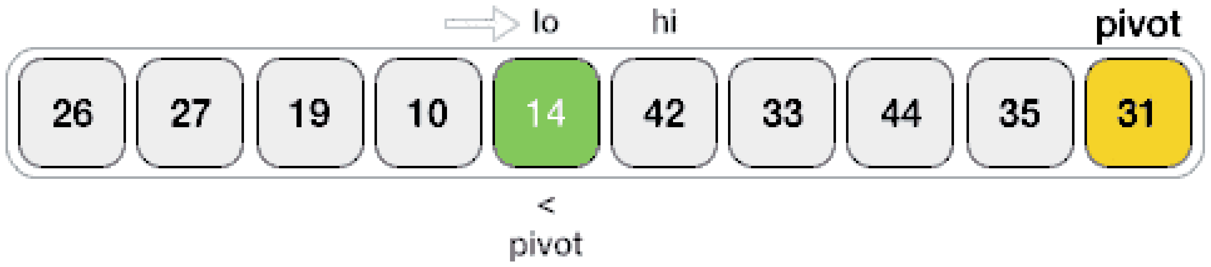
Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Quick Sort





ɔutube video

Parul® University

DIGITAL LEARNING CONTENT

Quick Sort

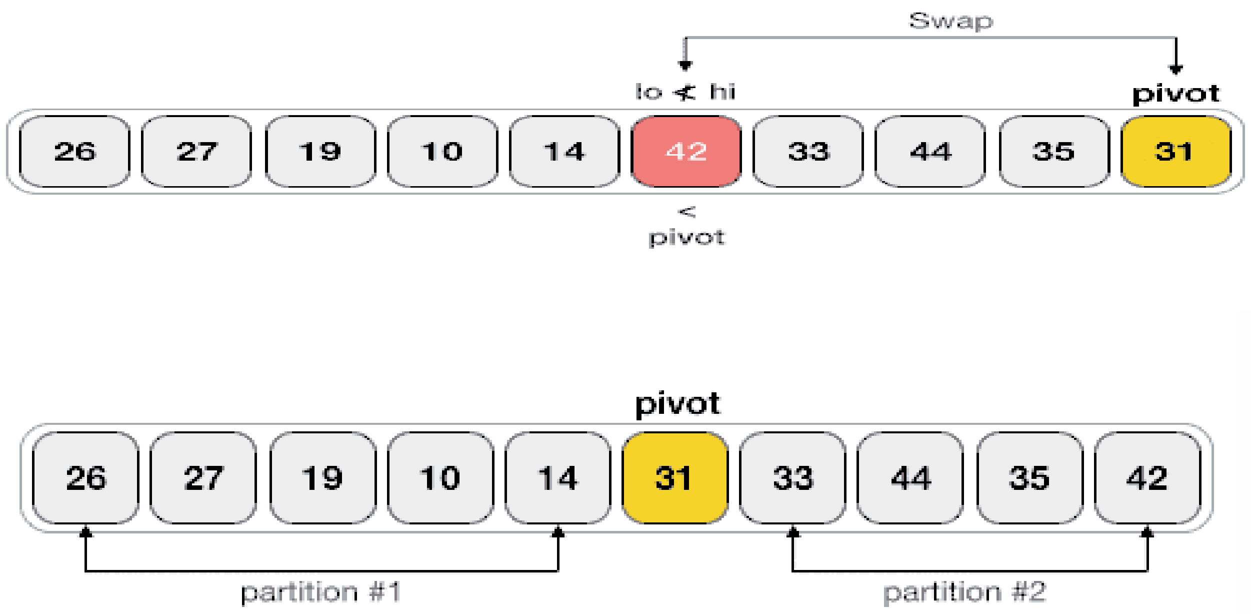


Image source : Youtube video

Algorithm

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1 )
        quicksort(A, p + 1, hi)
algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo - 1
    for j := lo to hi - 1 do
        if A[j] < pivot then
            i := i + 1
            swap A[i] with A[j]
    if A[hi] < A[i + 1] then
        swap A[i + 1] with A[hi]
    return i + 1
```

Image source : Youtube video

Complexity

- Mathematical analysis of quick sort shows that the on average, the algorithm takes 0(n logn) comparisons to sort n items.
- In the worst case, it makes 0(n2) comparisons, though this behavior is rare.

Image source : Youtube video

Quicksort Algorithm

Given an array of n elements (e.g., integers):

- If array only contains one element, return
- Else
  - ✓ pick one element to use as pivot.
  - ✓ Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - ✓ Quicksort two sub-arrays
  - ✓ Return results

Image source : Youtube video

Example

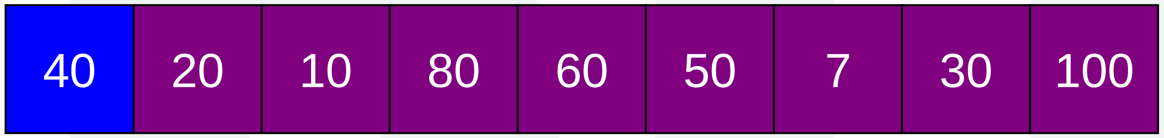
- We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Image source : Youtube video

Pick Pivot Element

- There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:



40	20	10	80	60	50	7	30	100

Image source : Youtube video

Partitioning Array

- Given a pivot, partition the elements of the array such that the resulting array consists of:
  1. One sub-array that contains elements >= pivot
  2. Another sub-array that contains elements < pivot
- The sub-arrays are stored in the original data array.
- Partitioning loops through, swapping elements below/above pivot.

Image source : Youtube video

DIGITAL LEARNING CONTENT

Quick Sort

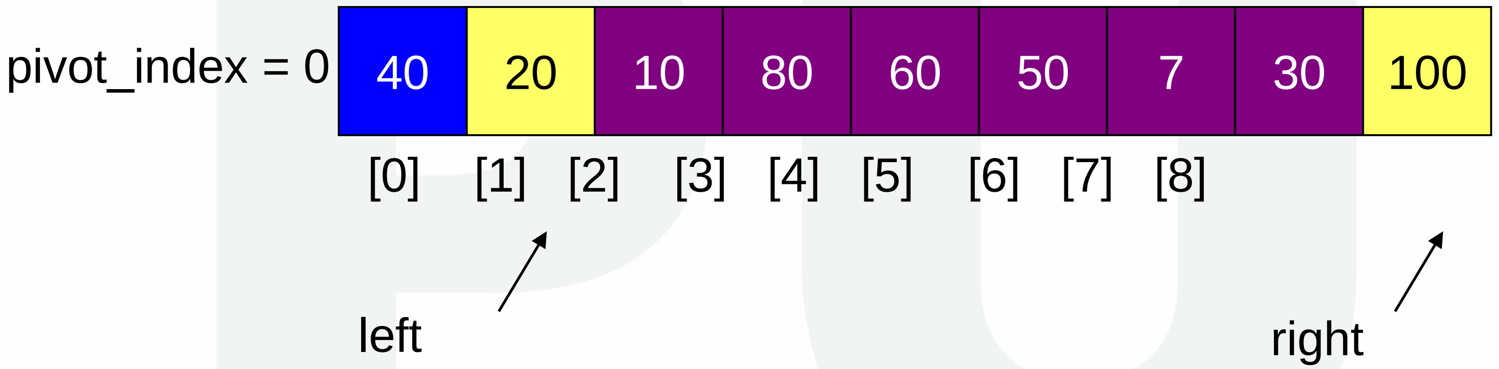


Image source : Youtube video

Quick Sort

1. While data[left] <= data[pivot]
   ++left

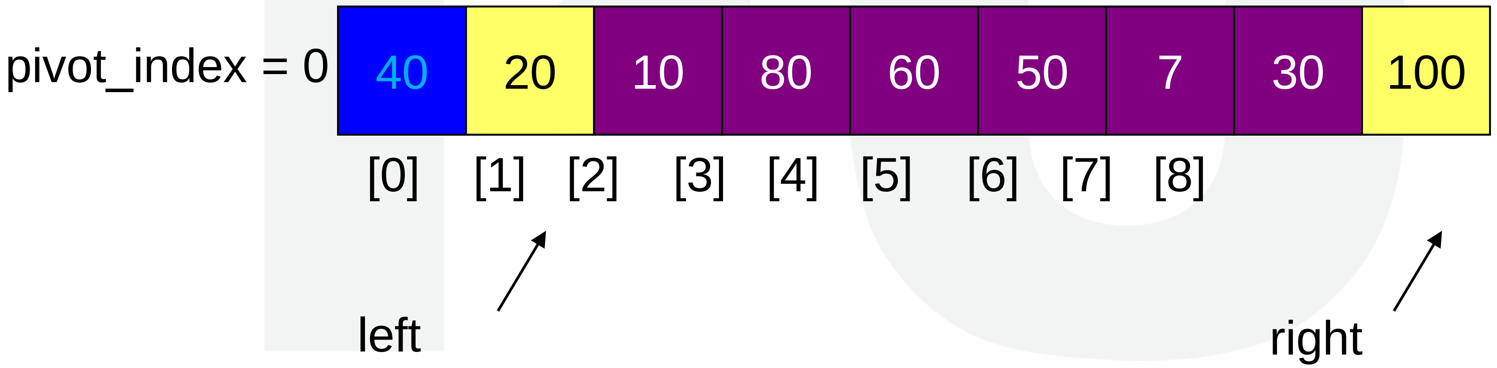


Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Quick Sort

1. While data[left] <= data[pivot]
   ++left

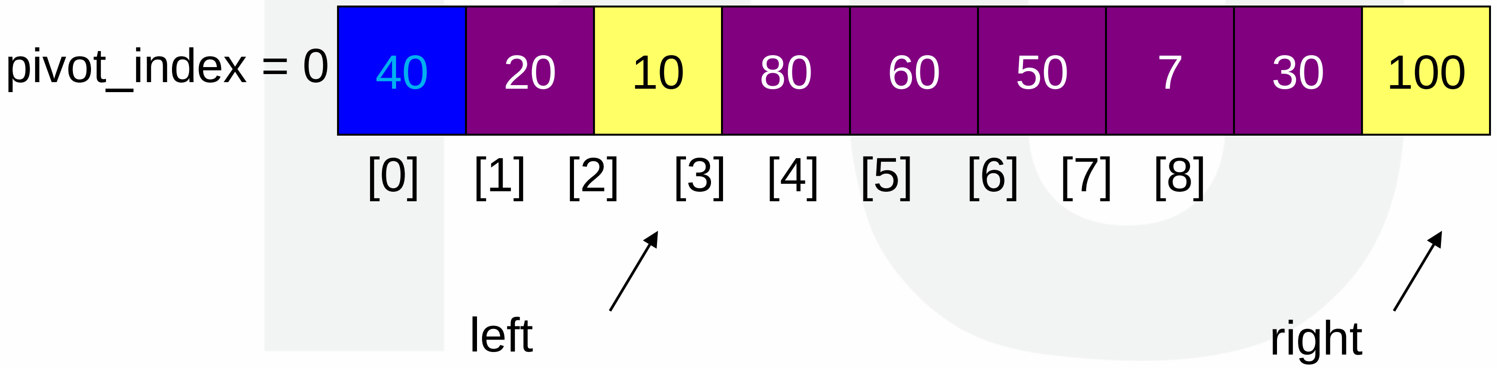


Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Complexity

1. While data[left] <= data[pivot]
   ++left

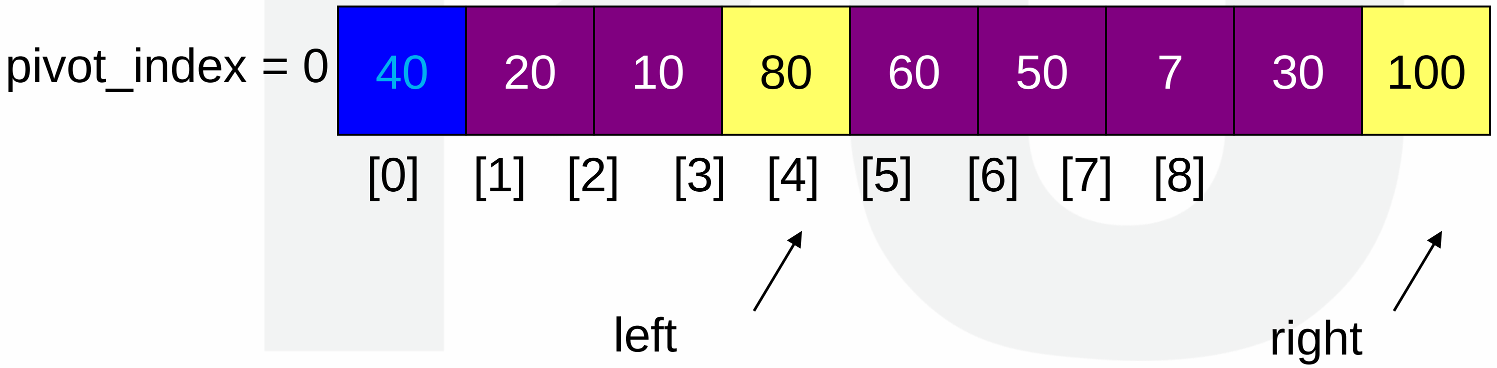


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
        ++left
2. While data[right] > data[pivot]
        --right
```

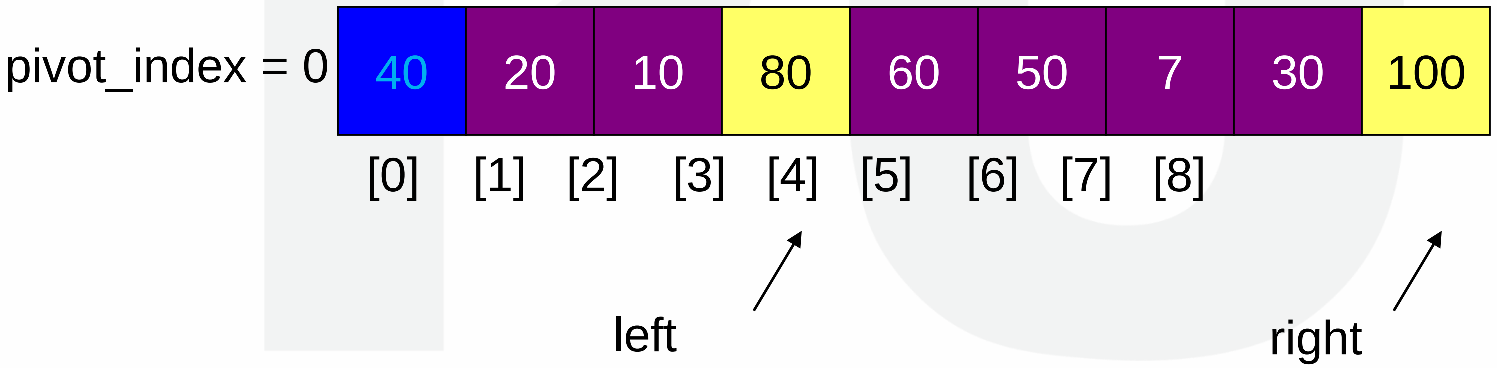


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
```

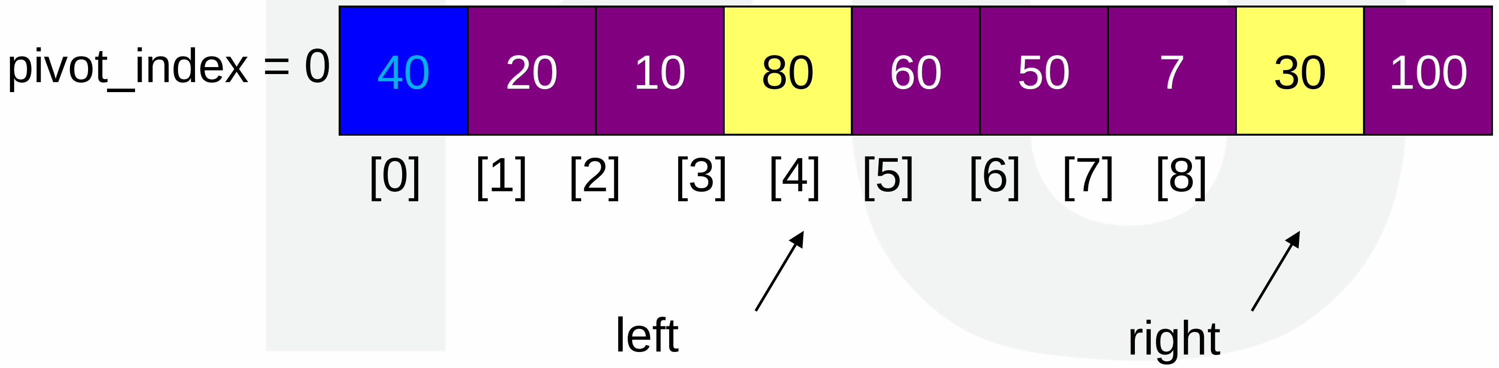


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
       ++left
2. While data[right] > data[pivot]
       --right
3. If left < right
       swap data[left] and data[right]
```

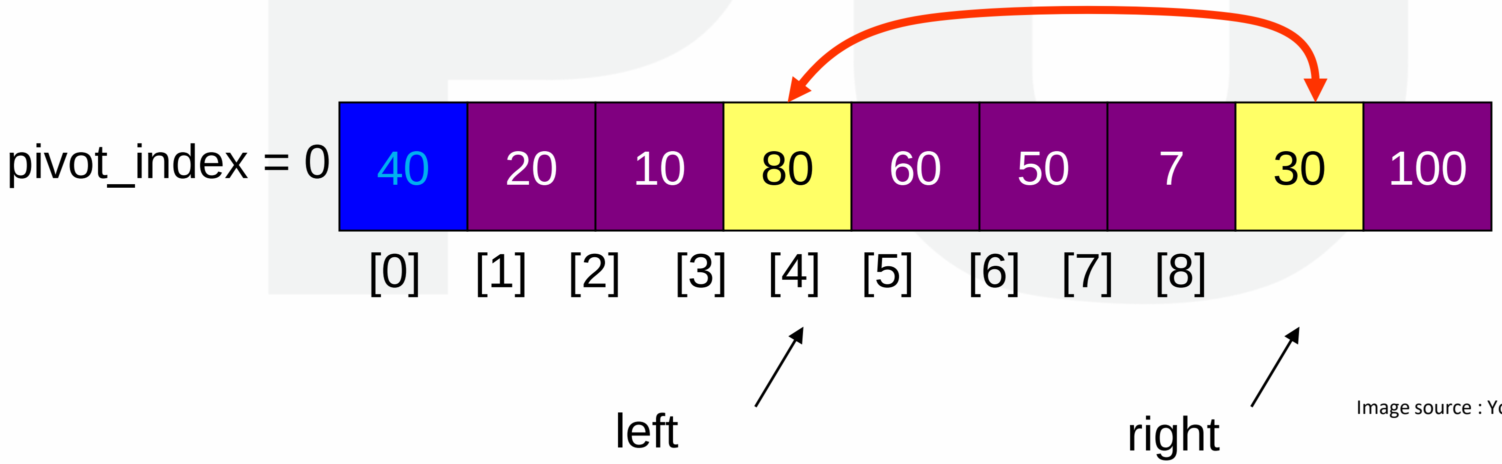


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
       ++left
2. While data[right] > data[pivot]
       --right
3. If left < right
       swap data[left] and data[right]
```

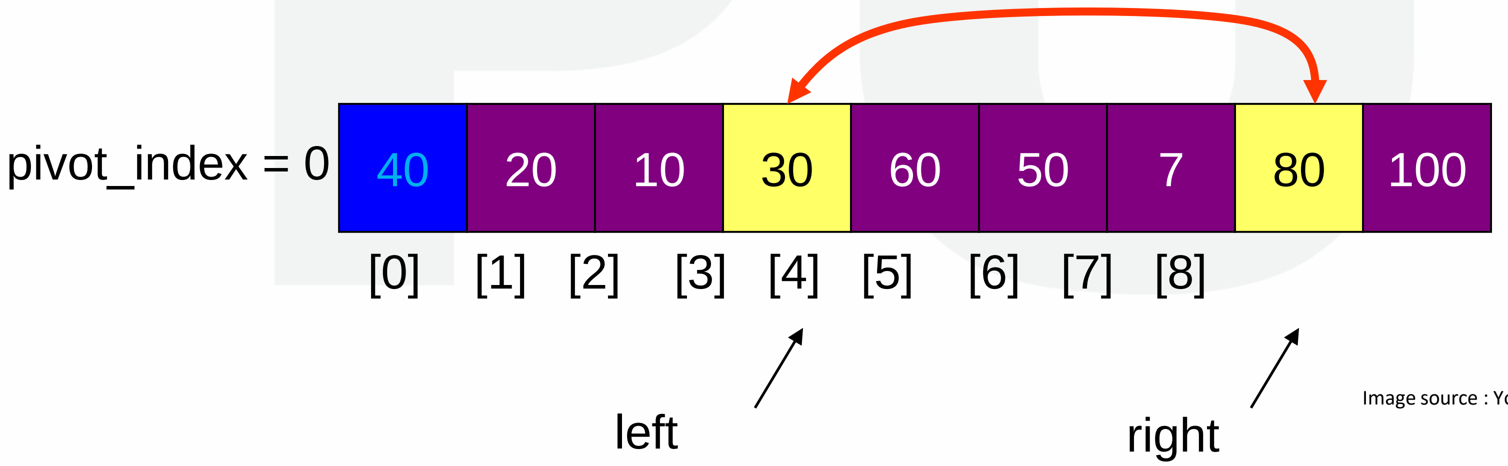


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

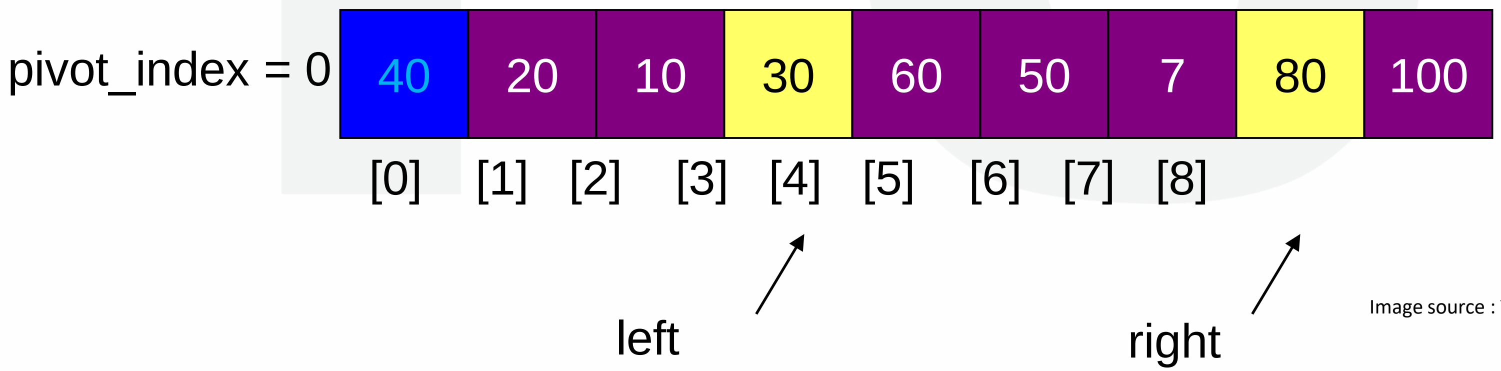


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

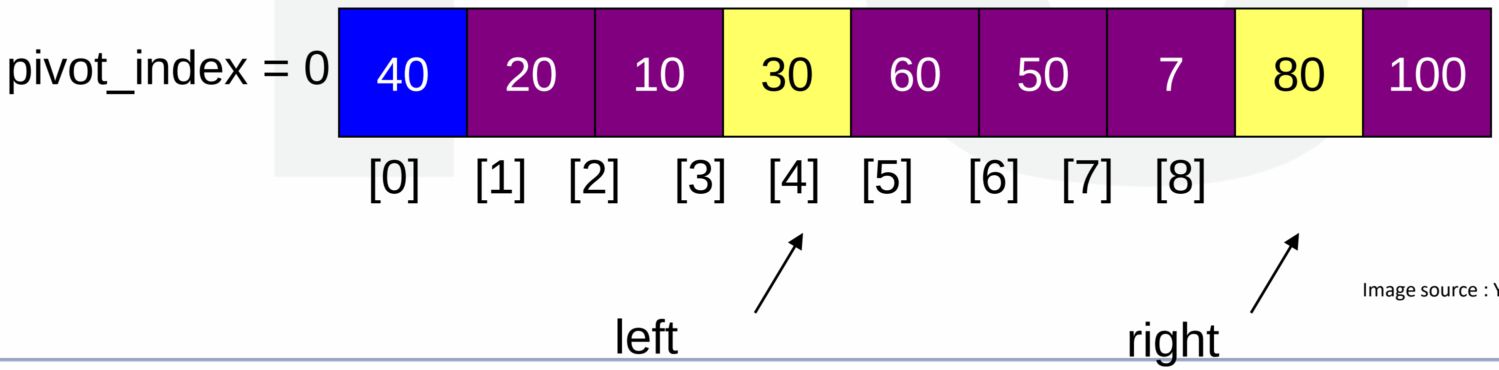


Image source : Youtube video

Complexity

```
→ 1.  While data[left] <= data[pivot]
          ++left
  2.  While data[right] > data[pivot]
          --right
  3.  If left < right
          swap data[left] and data[right]
  4.  While right > left, go to 1.
```

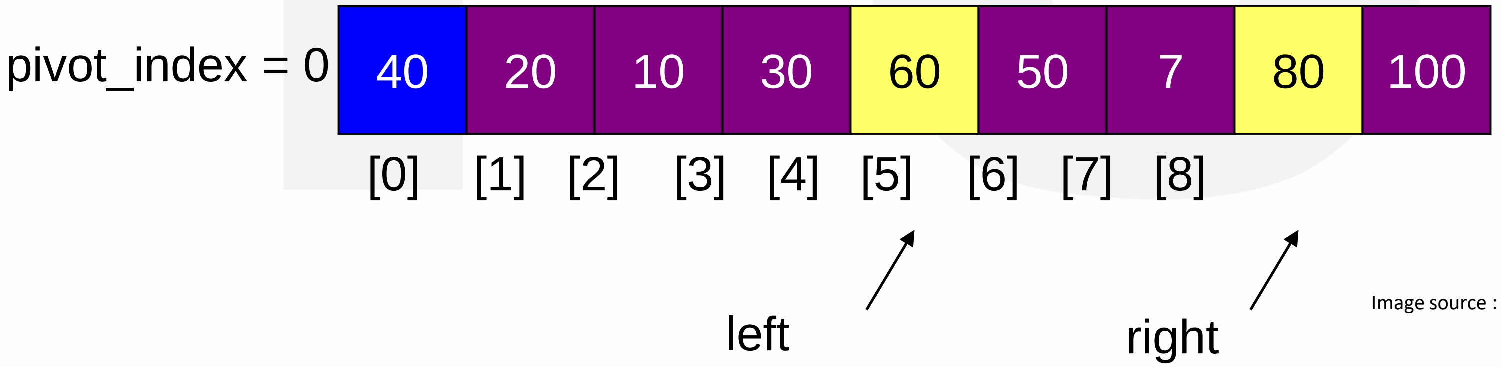


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

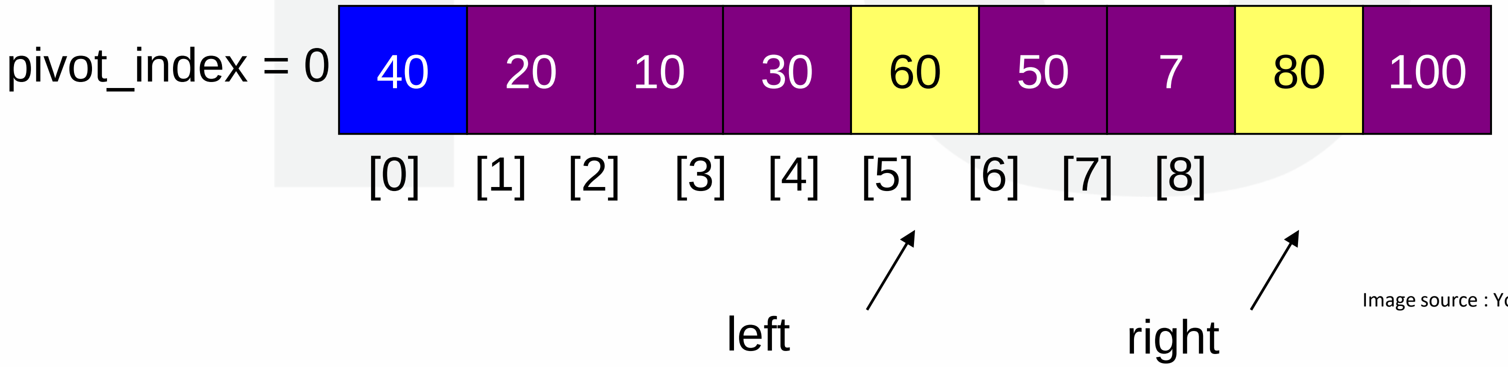


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
       ++left
2. While data[right] > data[pivot]
       --right
3. If left < right
       swap data[left] and data[right]
4. While right > left, go to 1.
```

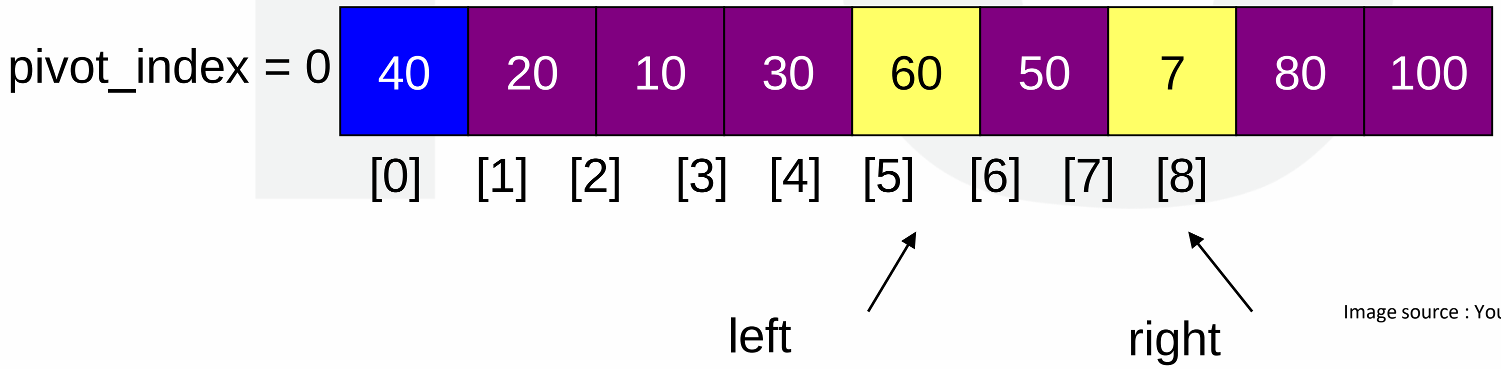


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

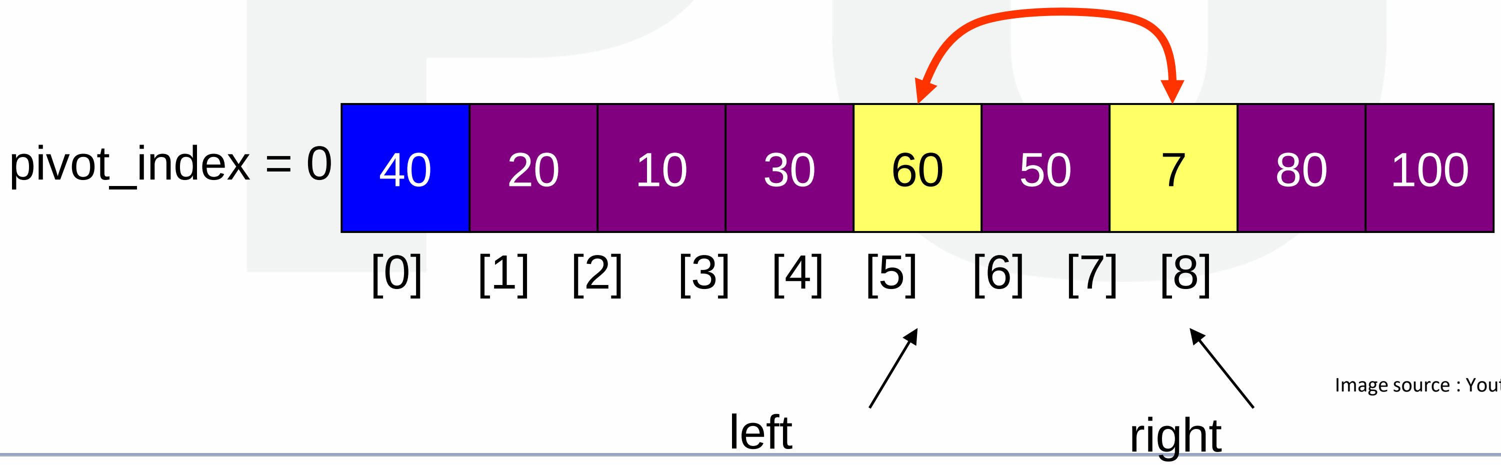


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

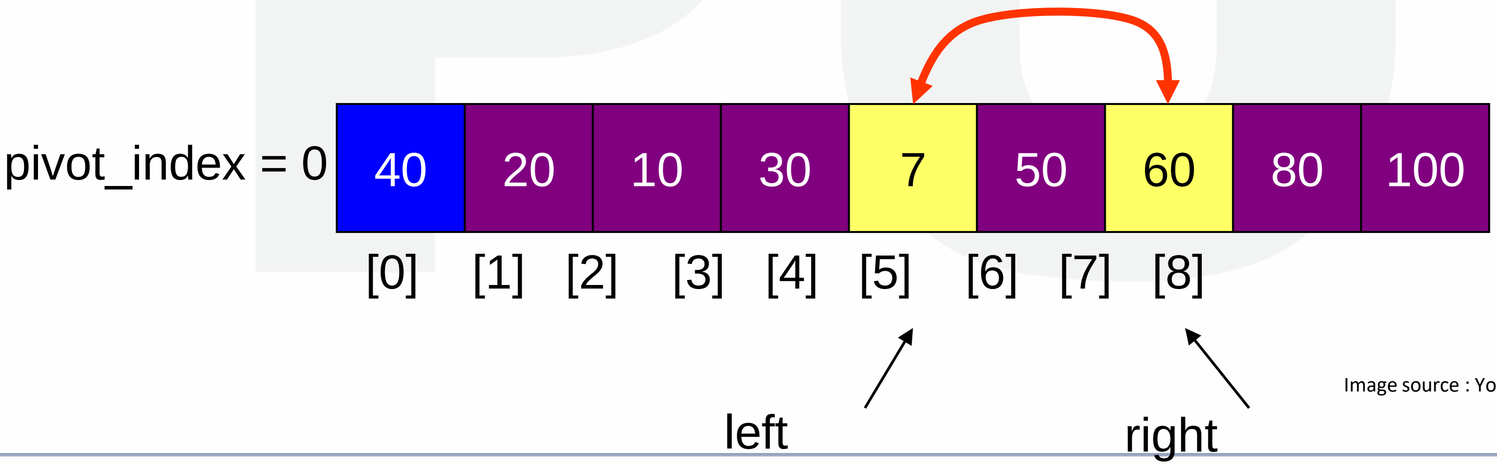


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

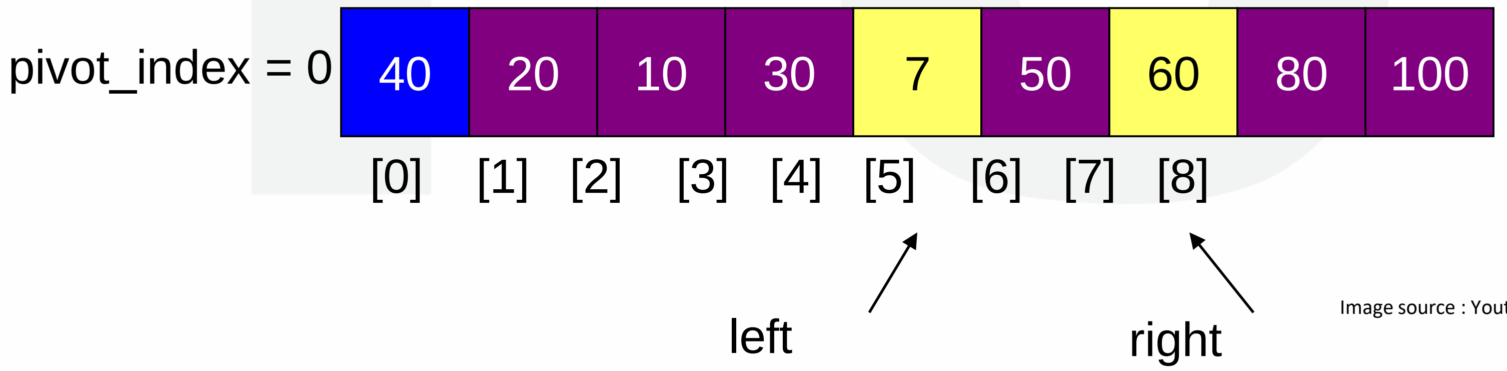


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
           ++left
2. While data[right] > data[pivot]
           --right
3. If left < right
           swap data[left] and data[right]
4. While right > left, go to 1.
```

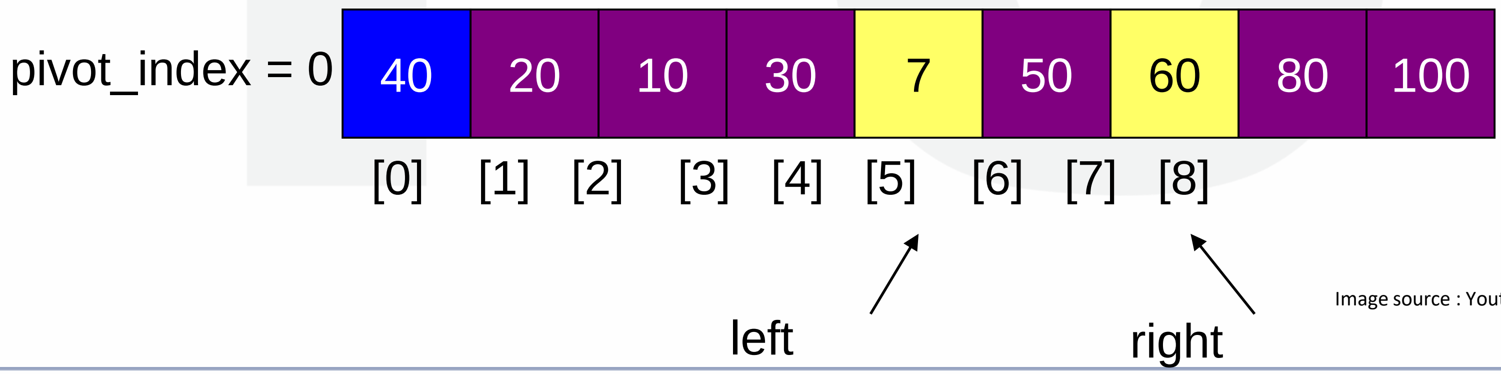


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
       ++left
2. While data[right] > data[pivot]
       --right
3. If left < right
       swap data[left] and data[right]
4. While right > left, go to 1.
```

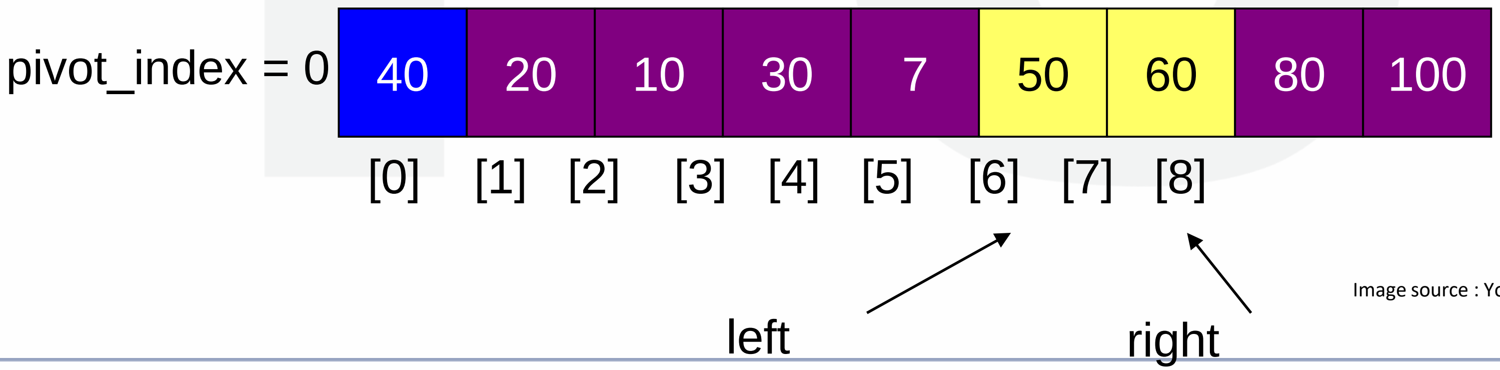


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

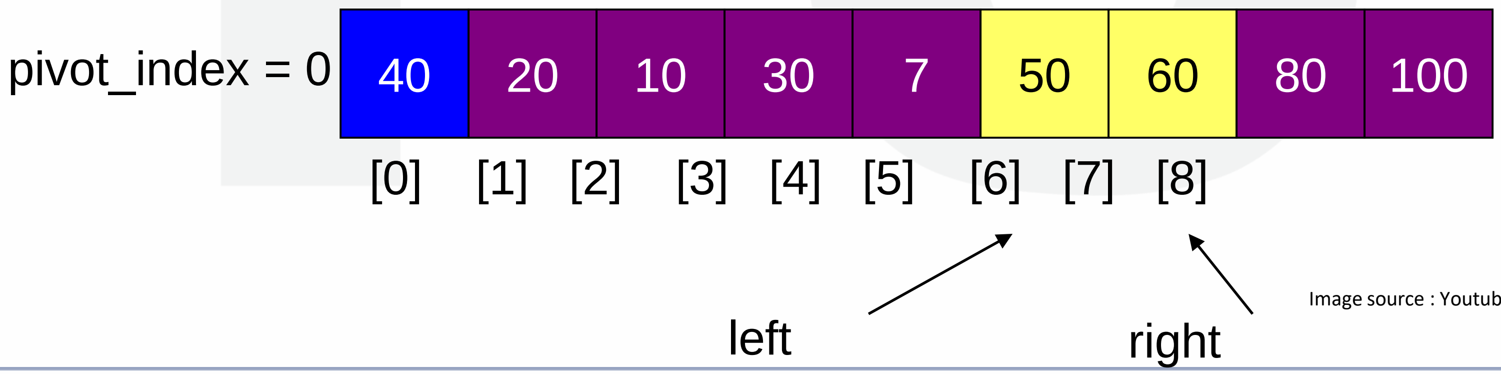


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

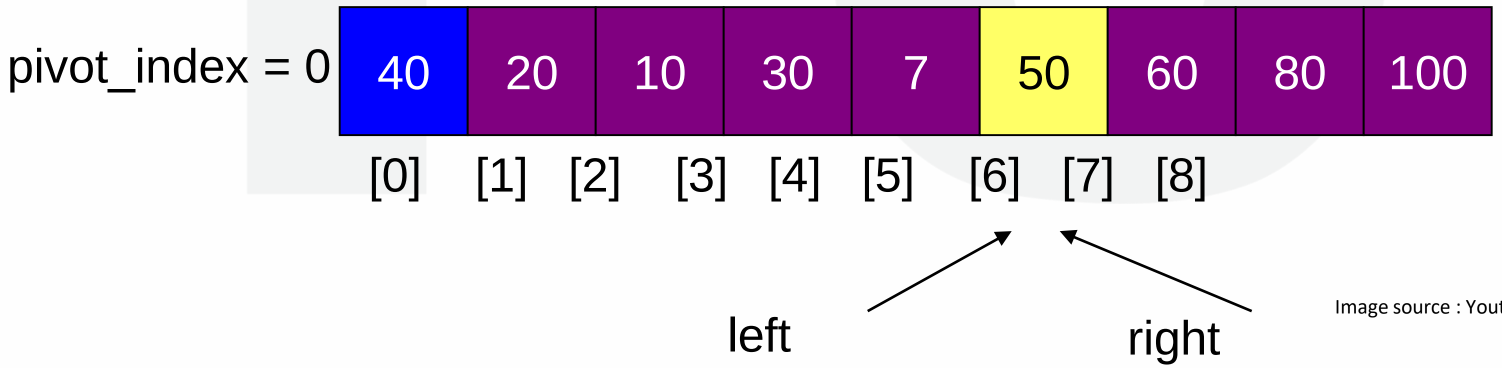


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

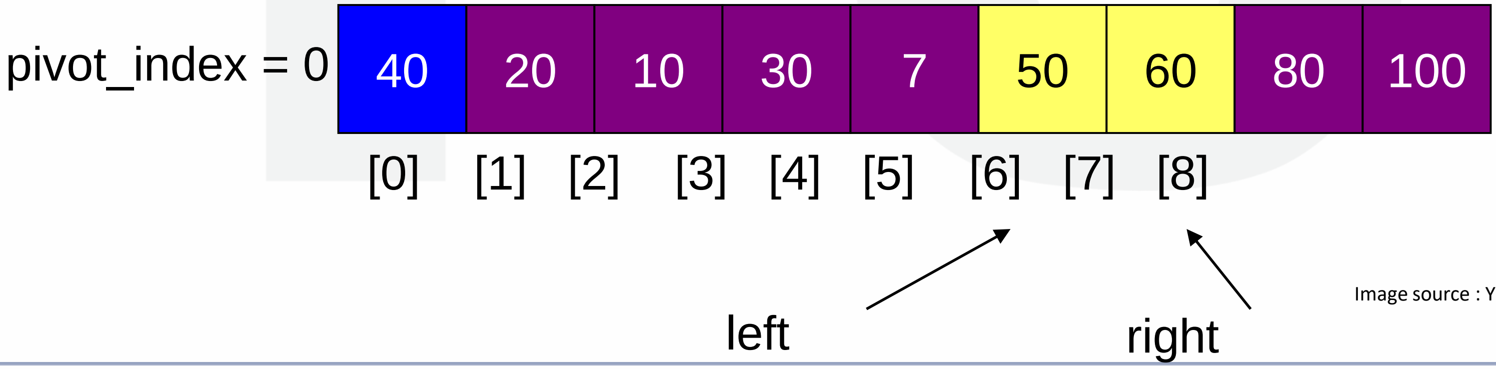


Image source : Youtube video

Complexity

```
1.    While data[left] <= data[pivot]
          ++left
2.    While data[right] > data[pivot]
          --right
3.    If left < right
          swap data[left] and data[right]
4.    While right > left, go to 1.
```

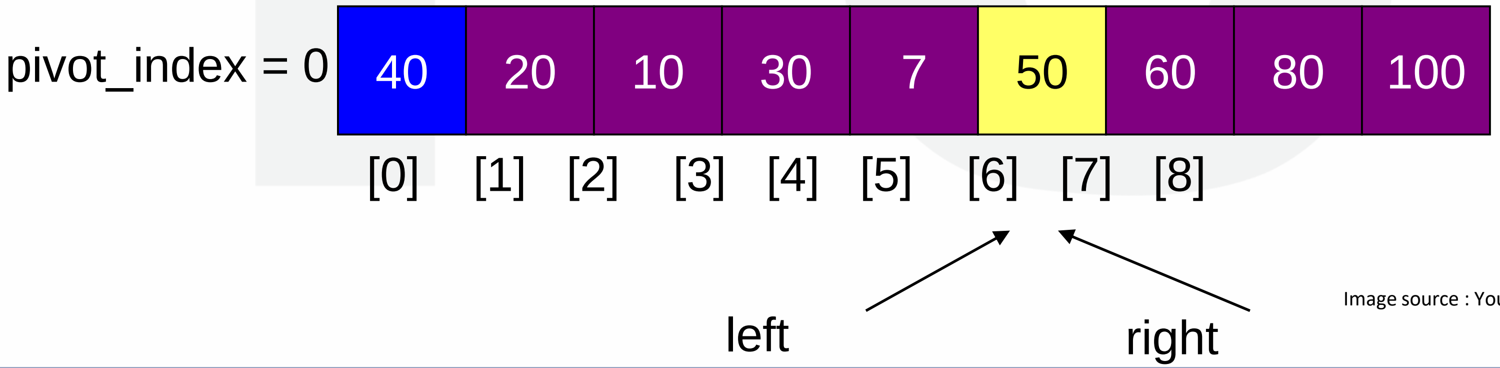


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

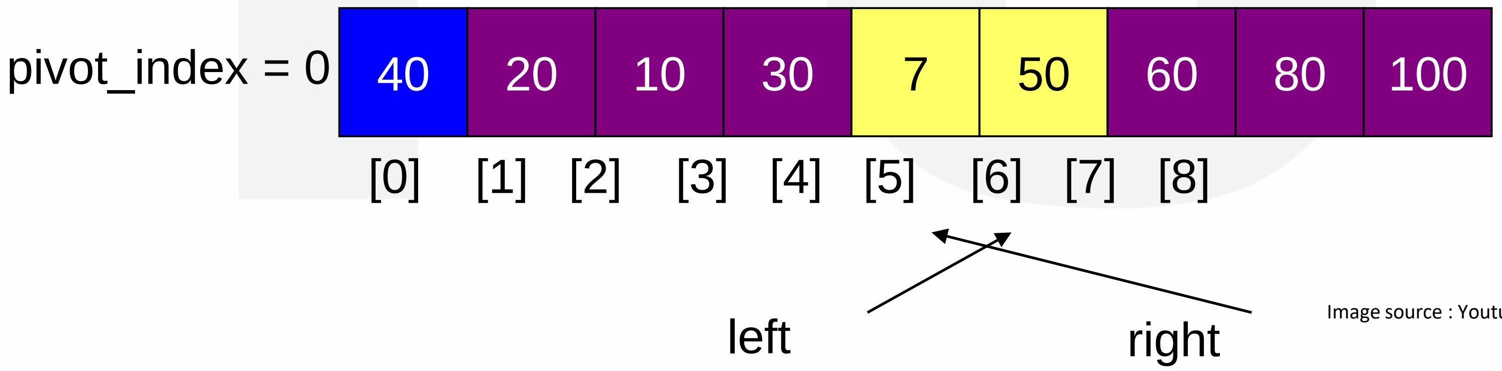


Image source : Youtube video

Complexity

```
1.  While data[left] <= data[pivot]
        ++left
2.  While data[right] > data[pivot]
        --right
3.  If left < right
        swap data[left] and data[right]
4.  While right > left, go to 1.
```

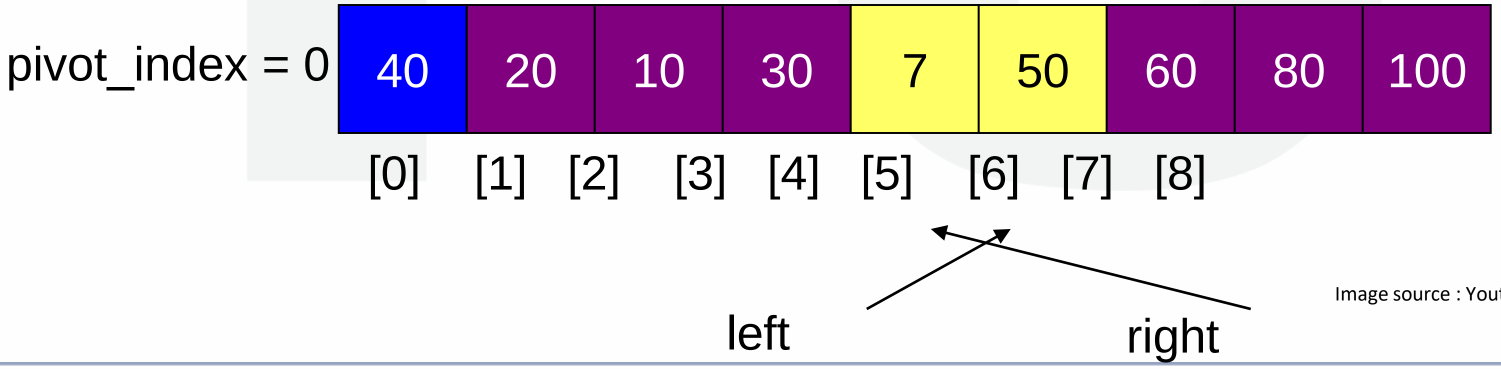


Image source : Youtube video

Complexity

```
1. While data[left] <= data[pivot]
        ++left
2. While data[right] > data[pivot]
        --right
3. If left < right
        swap data[left] and data[right]
4. While right > left, go to 1.
```

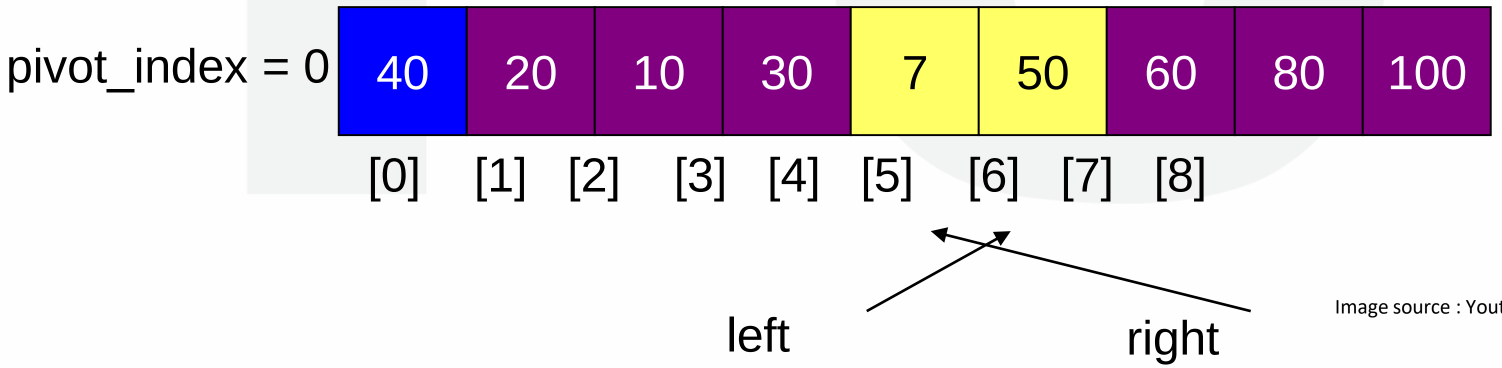


Image source : Youtube video

Complexity

```
1.   While data[left] <= data[pivot]
         ++left
2.   While data[right] > data[pivot]
         --right
3.   If left < right
         swap data[left] and data[right]
4.   While right > left, go to 1.
5.   Swap data[right] and data[pivot_index]
```

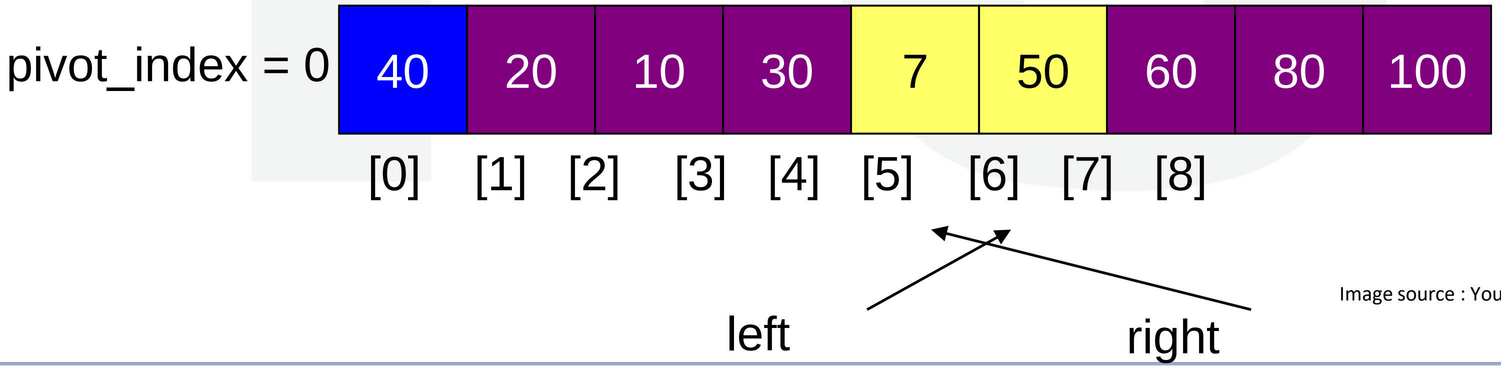


Image source : Youtube video

Complexity

```
1.   While data[left] <= data[pivot]
         ++left
2.   While data[right] > data[pivot]
         --right
3.   If left < right
         swap data[left] and data[right]
4.   While right > left, go to 1.
5.   Swap data[right] and data[pivot_index]
```

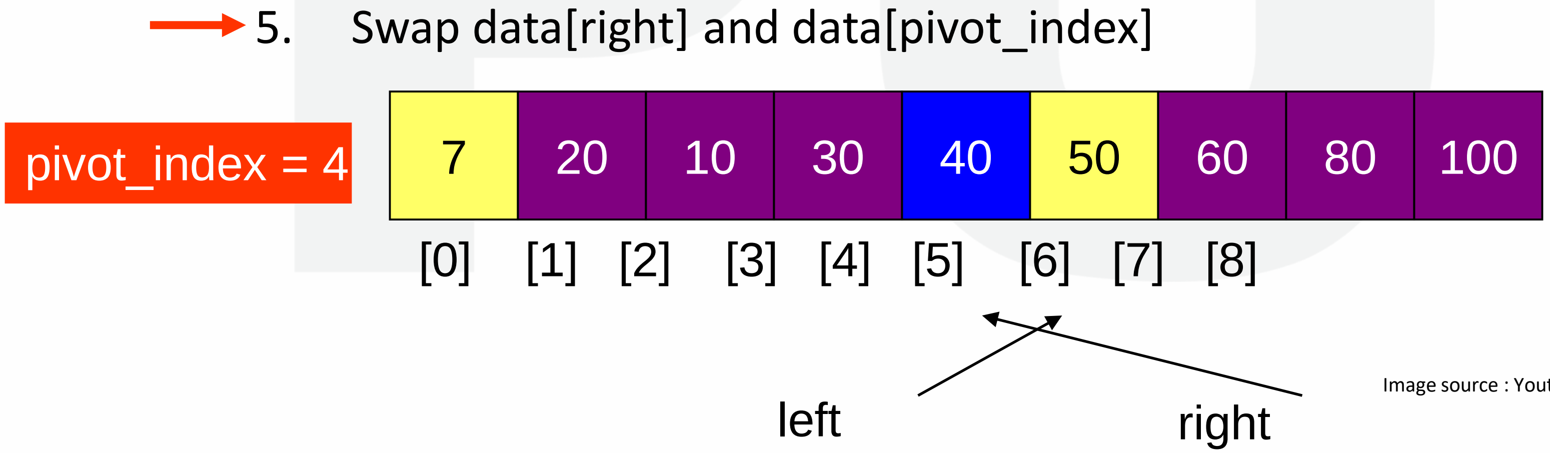


Image source : Youtube video

Complexity

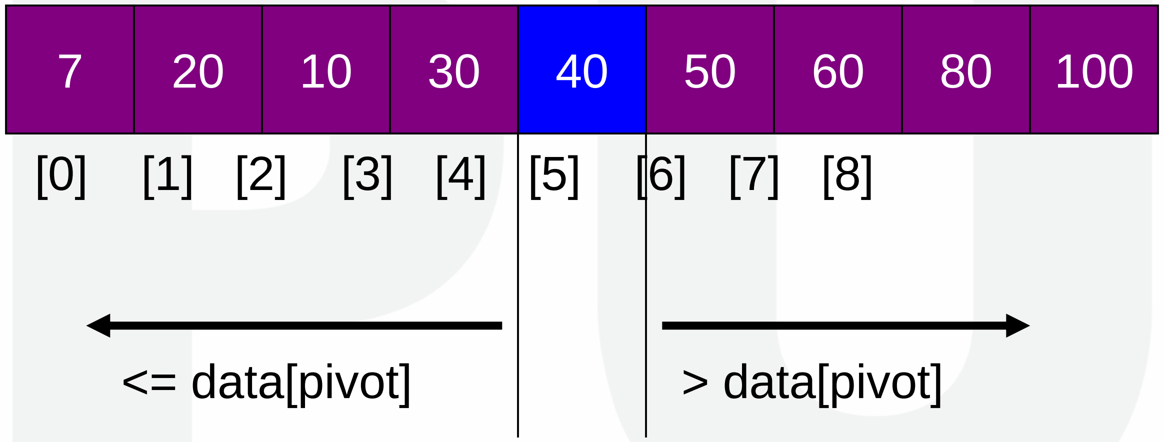


Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Recursion: Quicksort Sub-arrays

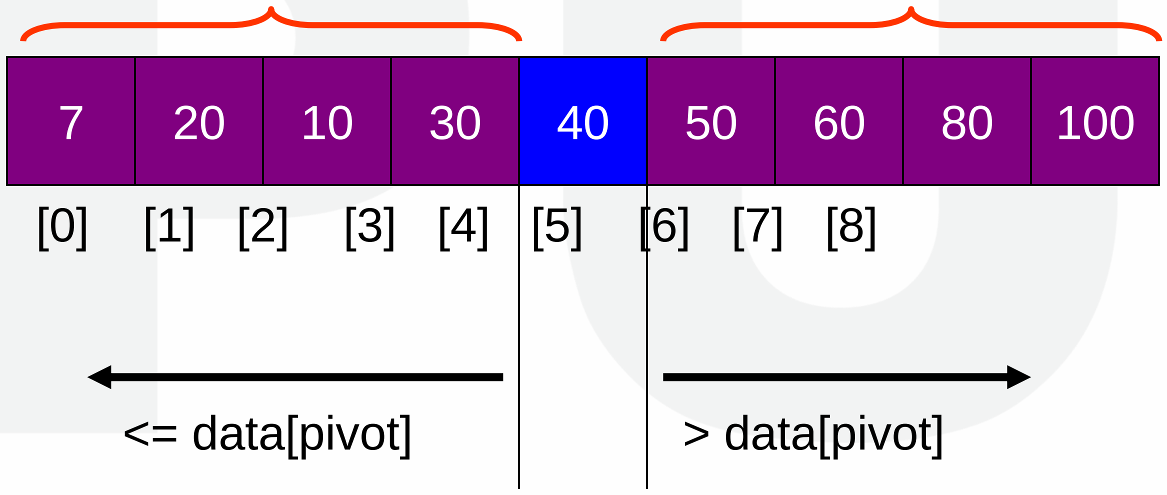


Image source : Youtube video

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
    - Recursion:

1. Partition splits array in two sub-arrays of size n/2
2. Quicksort each sub-array

- Depth of recursion tree? O(log2n)
- Number of accesses in partition? O(n)

Image source : Youtube video

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: O(n log2n)
- Worst case running time: O(n2)!!!

Image source : Youtube video

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: O(n log2n)
- Worst case running time: O(n2)!!!
- What can we do to avoid worst case?

Image source : Youtube video

Code

```
#include<stdio.h>
void quicksort(int number[25],int first,int
last)
{          int i, j, pivot, temp;
           if(first<last)
           {
                     pivot=first; i=first; j=last;
                     while(i<j)
                     {
           while(number[i]<=number[pivot]
&&i<last)
           i++;
           while(number[j]>number[pivot])
           j--;
if(i<j)
{
          temp=number[i];
number[i]=number[j];
number[j]=temp;
}
}
temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);

quicksort(number,j+1,last);
}
}
```

Image source : Youtube video

Code

```
int main()
{           int i, count, number[25];
            printf("How many elements are u going to enter?: ");
            scanf("%d",&count);
            printf("Enter %d elements: ", count);
            for(i=0;i<count;i++)
                        scanf("%d",&number[i]);
            quicksort(number,0,count-1);
            printf("Order of Sorted elements: ");
            for(i=0;i<count;i++)
                        printf(" %d",number[i]);
            return 0;
}
```

Image source : Youtube video

Shell Sort

- Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.
- This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
- This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.
- This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of O(n), where n is the number of items.

Image source : Youtube video

Working of Shell Sort

- Suppose, we need to sort the following array.
- We are using the shell's original sequence (N/2, N/4, ...1) as intervals in our algorithm.
- In the first loop, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.
- The 0th element is compared with the 4th element.
- If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.

Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Working of Shell Sort

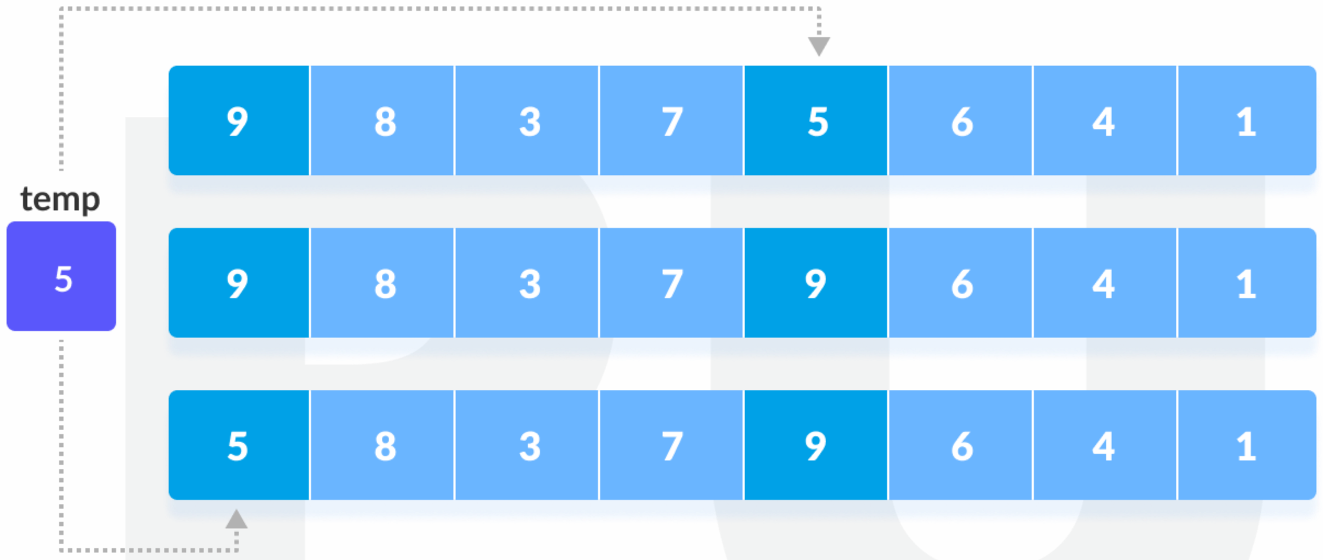


Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Working of Shell Sort

This process goes on for all the remaining elements.

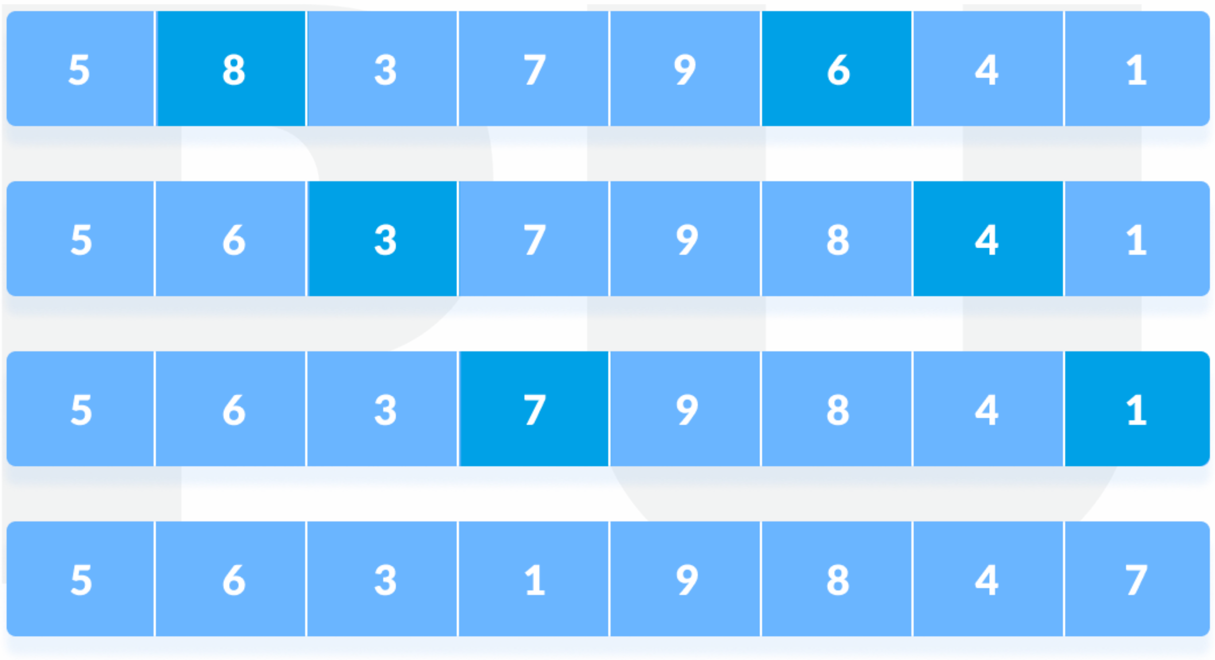
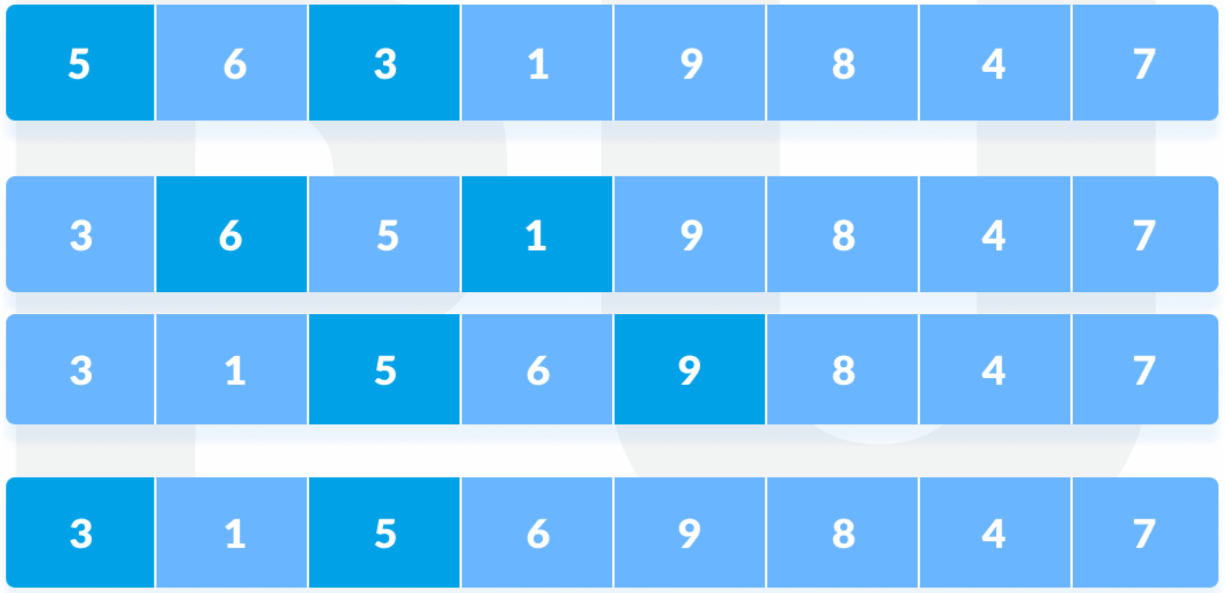


Image source : Youtube video

Working of Shell Sort

In the second loop, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.

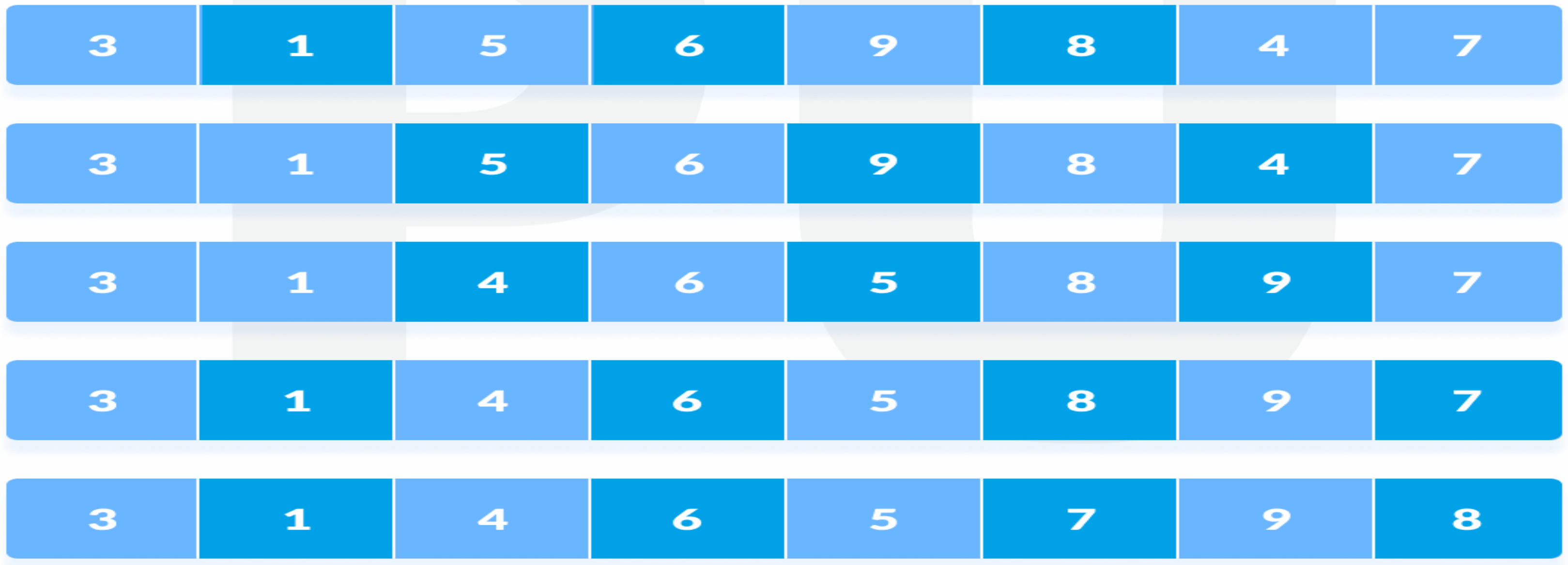


All the elements in the array lying at the current interval are compared.

Image source : Youtube video

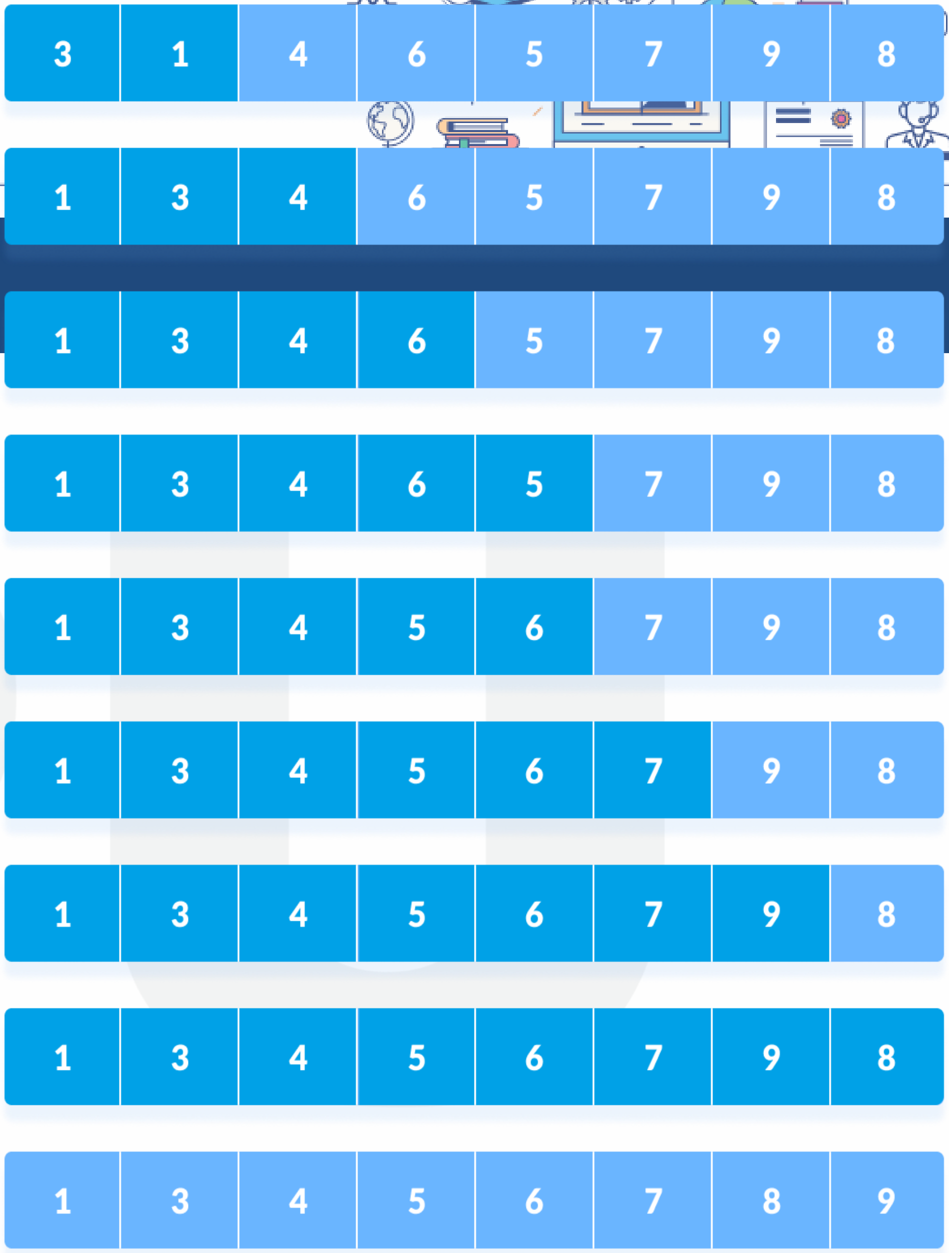
Working of Shell Sort

- The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared.
- The same process goes on for remaining elements.



Working of Shell Sort

- Finally, when the interval is N/8 = 8/8 =1 then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.



Shell Sort

- gap= Floor(N/2) N: Number of element in list
- For example:

  N=7

  gap= floor(7/2)

  =floor(3.5)

  gap=3

Image source : Youtube video

Algorithm

```
Int shellSort(int  arr[] , int n) {

   for( int gap = n / 2; gap > 0; gap = gap / 2 )
      for( int i = gap; i < n; i++ ) {
                     Int  temp =arr[i];
                     int j;
         for(j=i ; j >= gap  && arr[j - gap ] > temp ; j -= gap )
          arr[ j ] = arr[ j - gap ];
        arr[ j ] = temp;
      }
}
```

Image source : Youtube video

Code

```
#include <stdio.h>
// Shell sort
void shellSort(int array[], int n) {
  // Rearrange elements at each n/2, n/4, n/8, ... intervals
  for (int interval = n / 2; interval > 0; interval /= 2) {
    for (int i = interval; i < n; i += 1) {
      int temp = array[i];
      int j;
      for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
        array[j] = array[j - interval];
      }
      array[j] = temp;
    }
  }
}
```

Image source : Youtube video

Code

```
// Print an array
void printArray(int array[], int size) {
 for (int i = 0; i < size; ++i) {
   printf("%d  ", array[i]);
 }
 printf("\n");
}
// Driver code
int main() {
 int data[] = {9, 8, 3, 7, 5, 6, 4, 1};
 int size = sizeof(data) / sizeof(data[0]);
 shellSort(data, size);
 printf("Sorted array: \n");
 printArray(data, size);
}
```

Image source : Youtube video

Merge Sort

- **Divide And Conquer**
- Merging two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the left side then the right side and then merging the left and right back together.

Image source : Youtube video

Merge sort Example

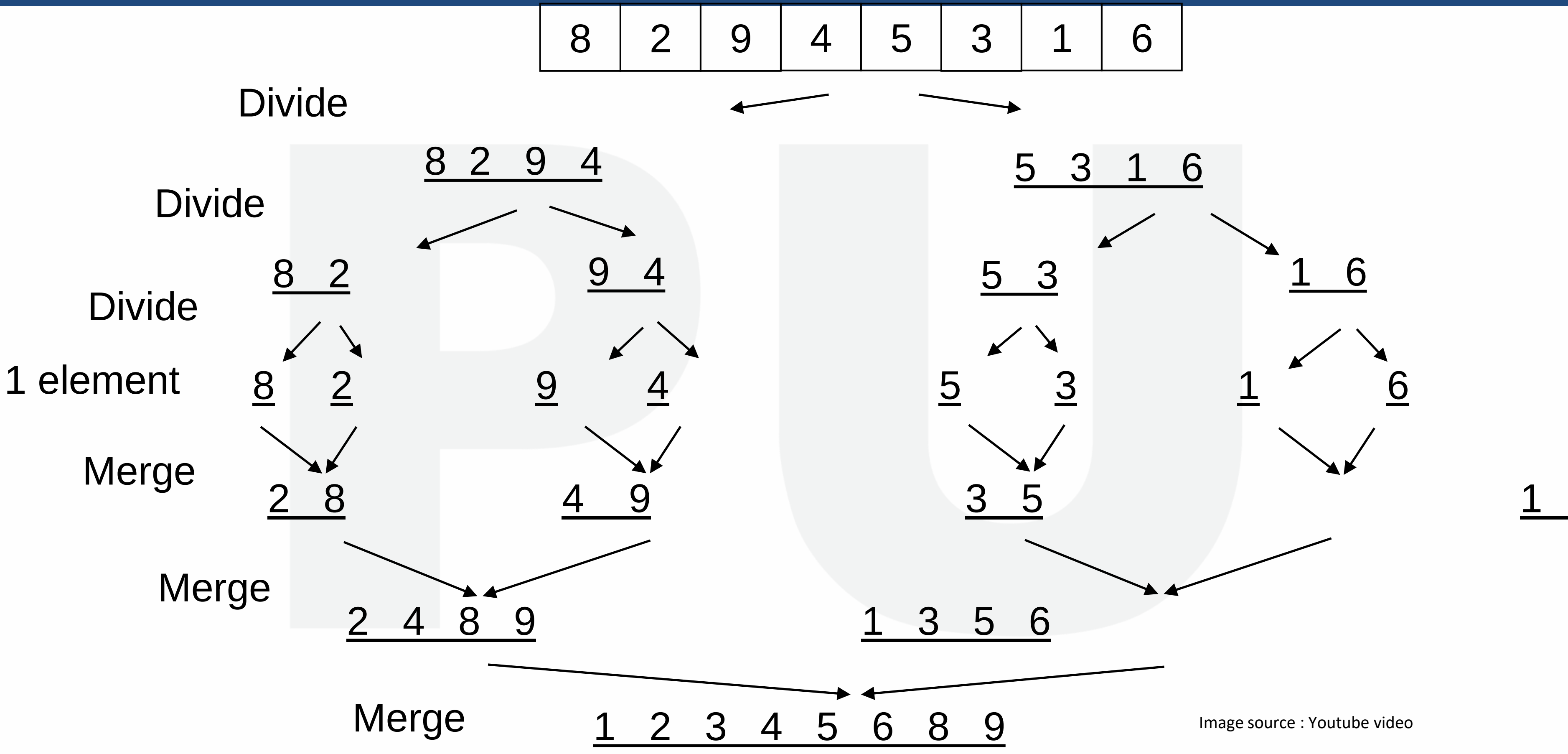
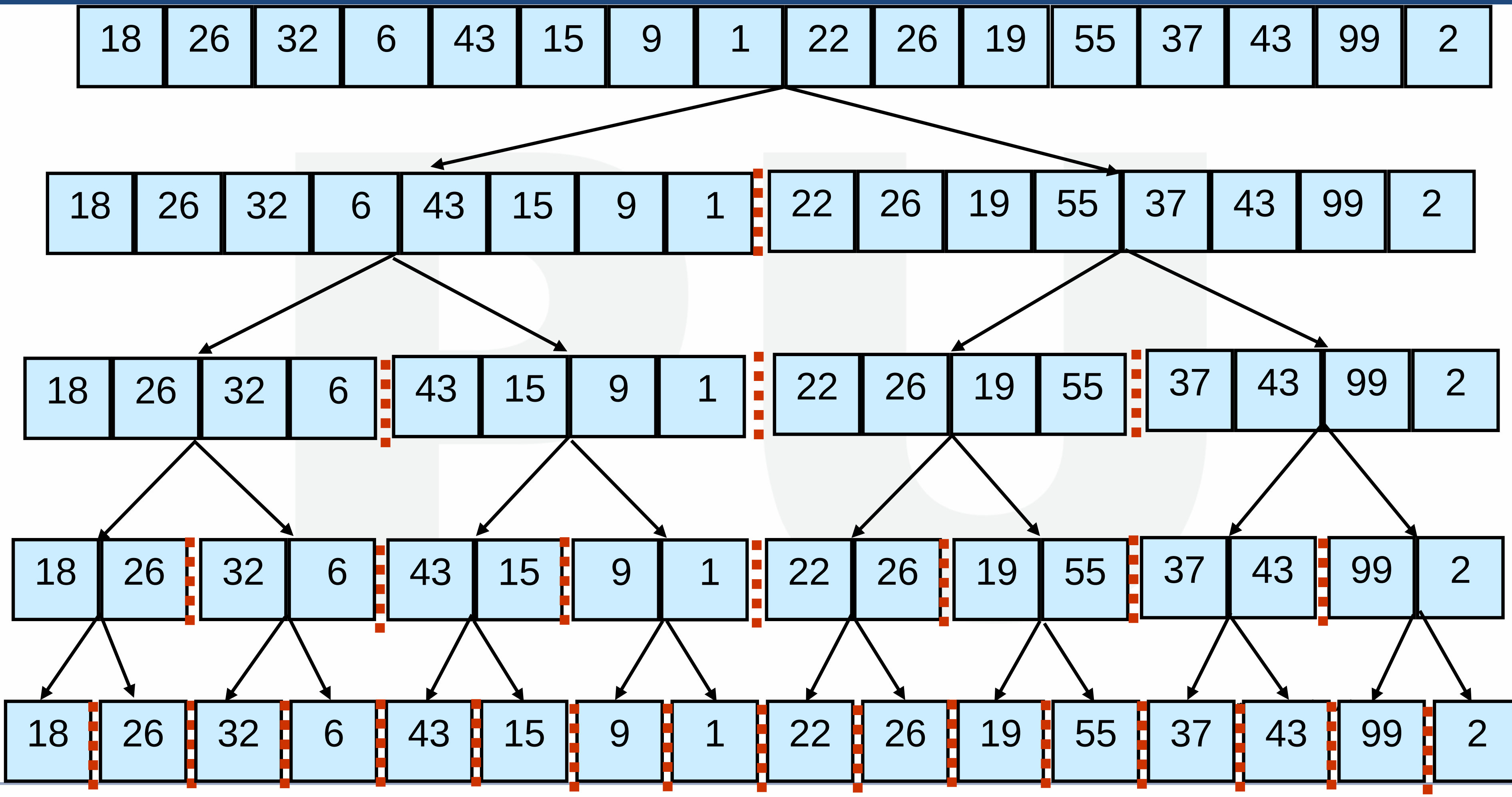


Image source : Youtube video

Merge Sort – Example



Parul® University

DIGITAL LEARNING CONTENT

Merge Sort – Example

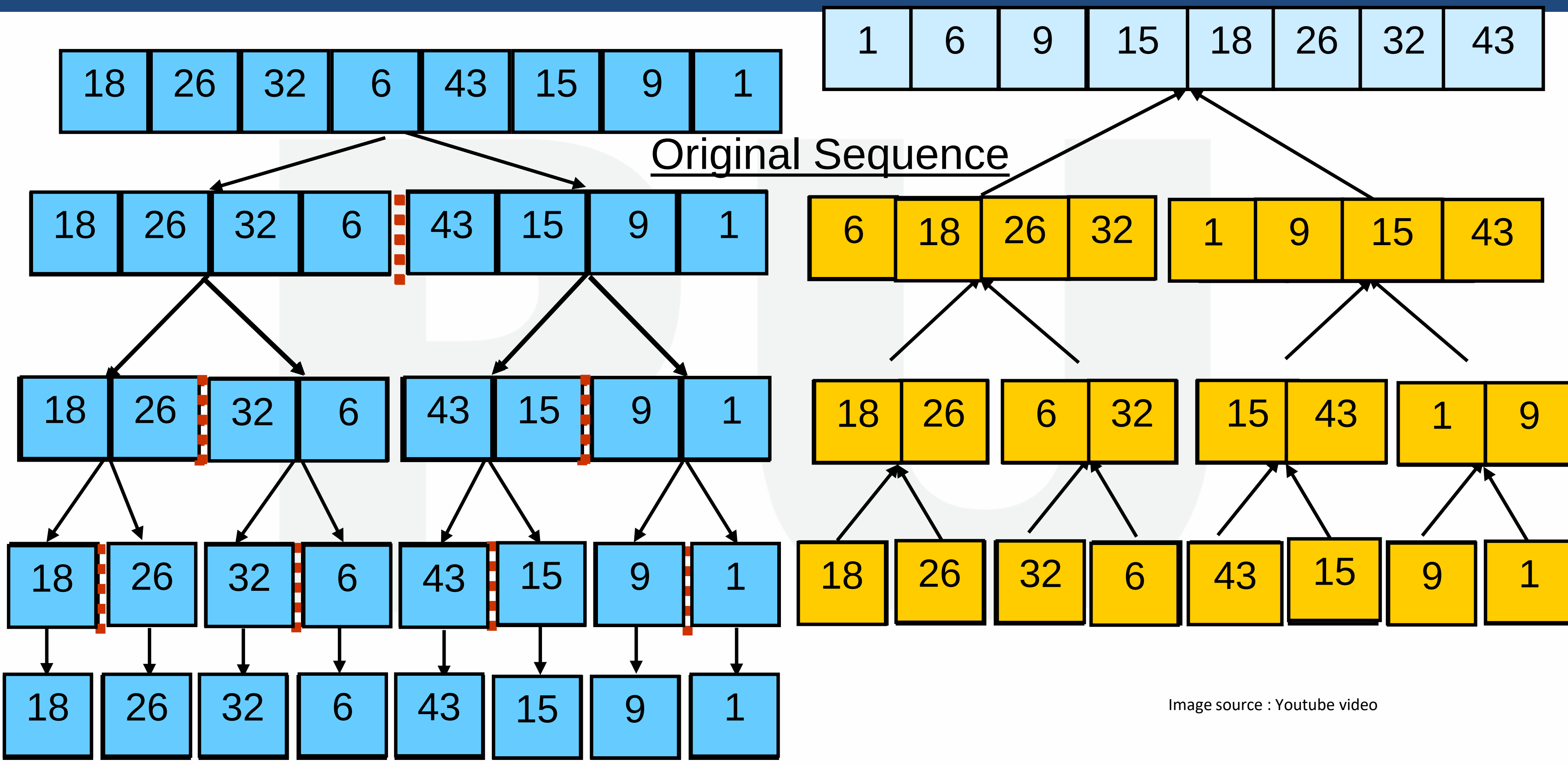


Image source : Youtube video

Divide-and-conquer Technique

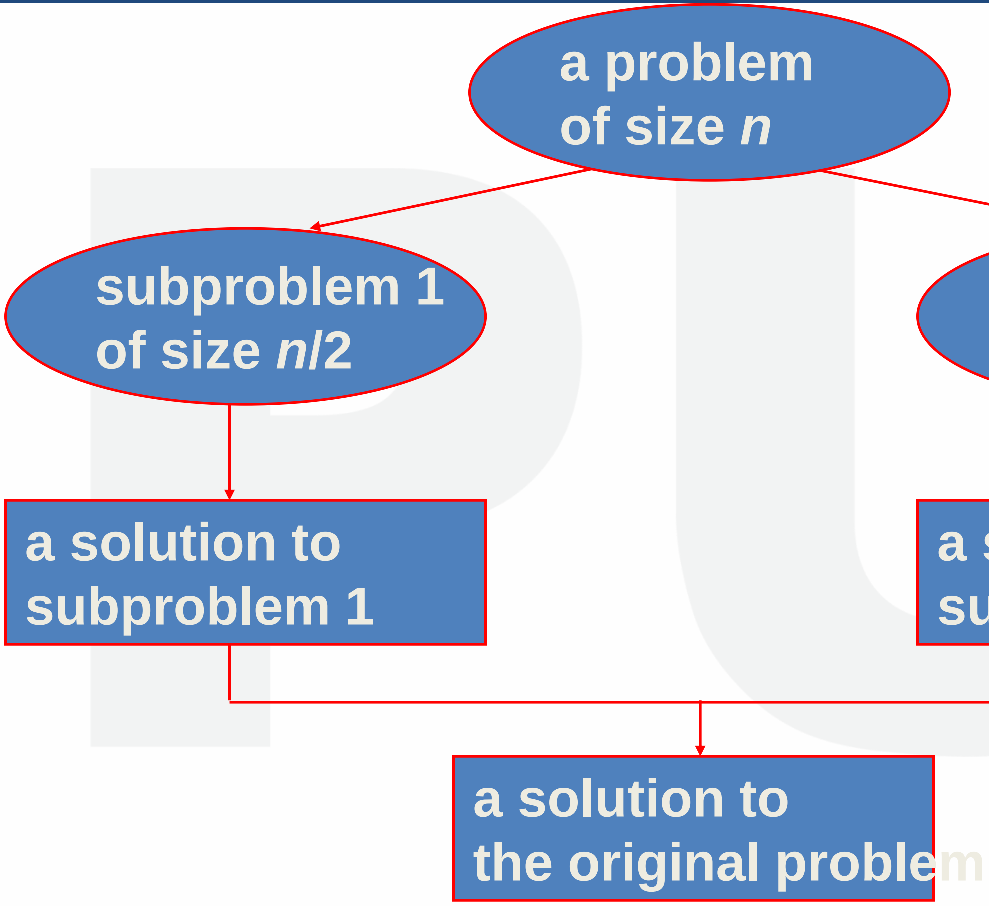

a problem of size *n*
subproblem 1 of size *n*/2
subproblem 2 of size *n*/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

Image source : Youtube video

Merge Sort Algorithm

- Divide: Partition ‘n’ elements array into two sub lists with n/2 elements each
- Conquer: Sort sub list1 and sub list2
- Combine: Merge sub list1 and sub list2

Image source : Youtube video

Partitioning Array

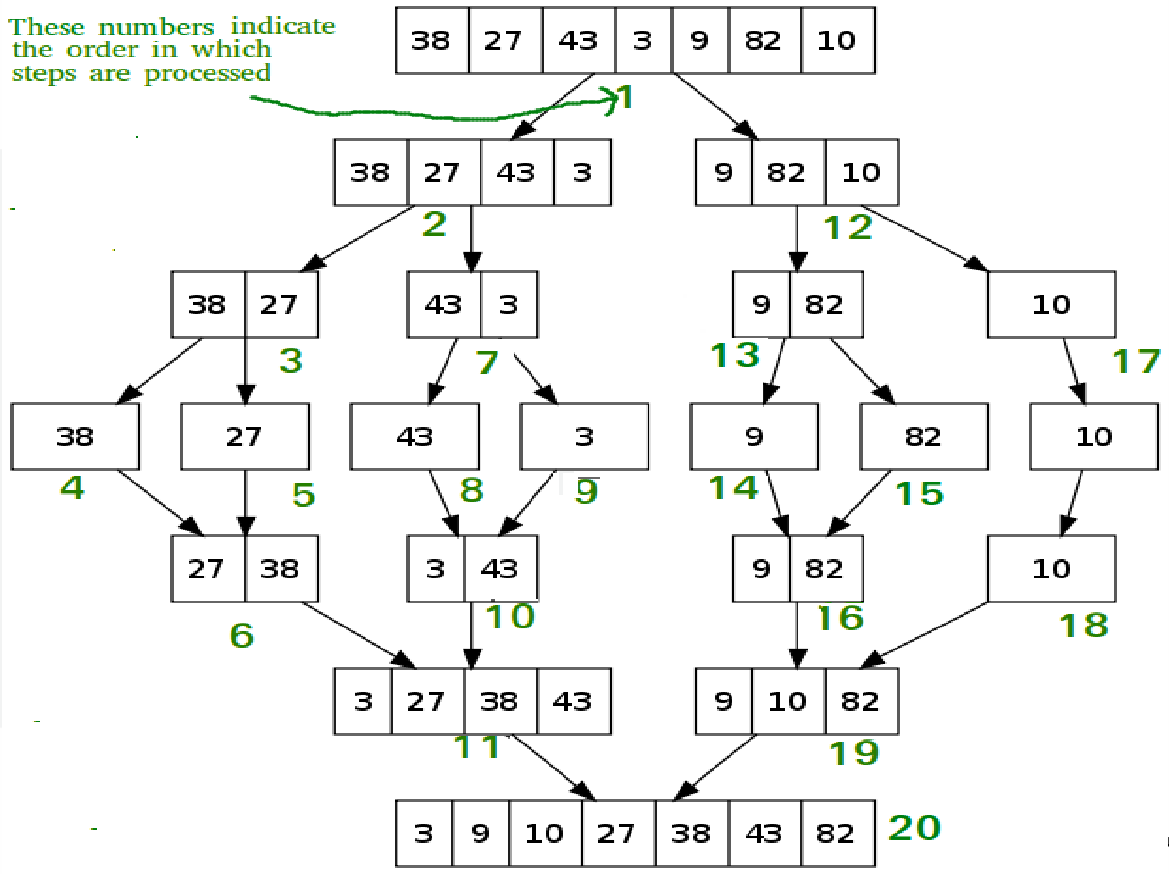
- Given a pivot, partition the elements of the array such that the resulting array consists of:
    1. One sub-array that contains elements >= pivot
    2. Another sub-array that contains elements < pivot

- The sub-arrays are stored in the original data array.

- Partitioning loops through, swapping elements below/above pivot.

Image source : Youtube video

Merge Sort



ıbe video

Partitioning Array

```
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
 /* Driver program to test above functions */
```

```
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printf("Given array is \n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Image source : Youtube video

Code

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;

   /* create temp arrays */
   int L[n1], R[n2];

   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];

   /* Merge the temp arrays back into arr[l..r]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
 while (i < n1 && j < n2)
   {
      if (L[i] <= R[j])
      {
         arr[k] = L[i];
         i++;
      }
      else
      {
         arr[k] = R[j];
         j++;
      }
      k++;
   }
```

Image source : Youtube video

Parul® University

DIGITAL LEARNING CONTENT

Partitioning Array

```
    /* Copy the remaining elements of L[], if there are any
*/
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
     /* Copy the remaining elements of R[], if there are
any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
      /* l is for left index and r is right index of the sub-
array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

Image source : Youtube video

Time Complexity Analysis

- Worst case: O(nlog2n)
- Average case: O(nlog2n)
- Best case: O(nlog2n)

Image source : Youtube video

What is Search?

- Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

- Types of Search
    1) Linear Search
    2) Binary Search

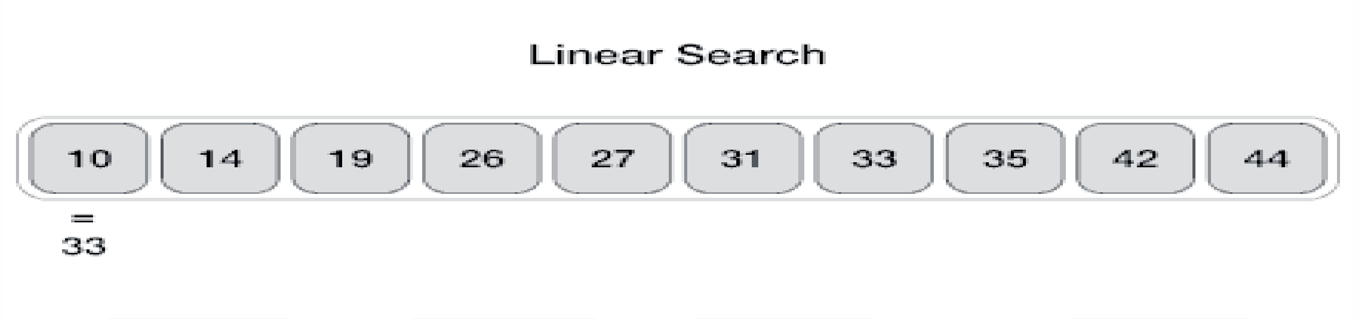
\

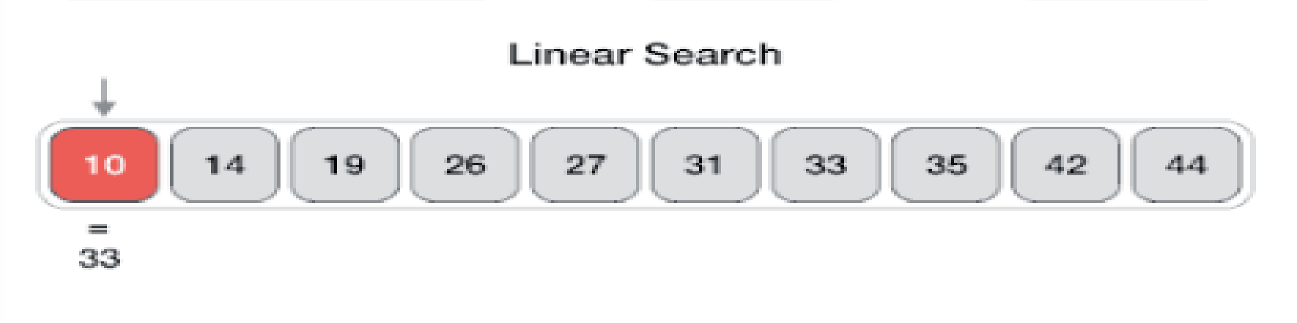
\

Linear Search

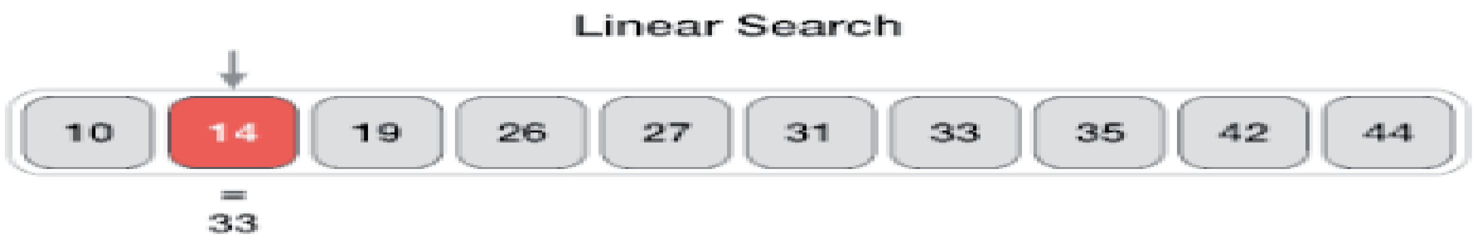
- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Linear Search





Linear Search



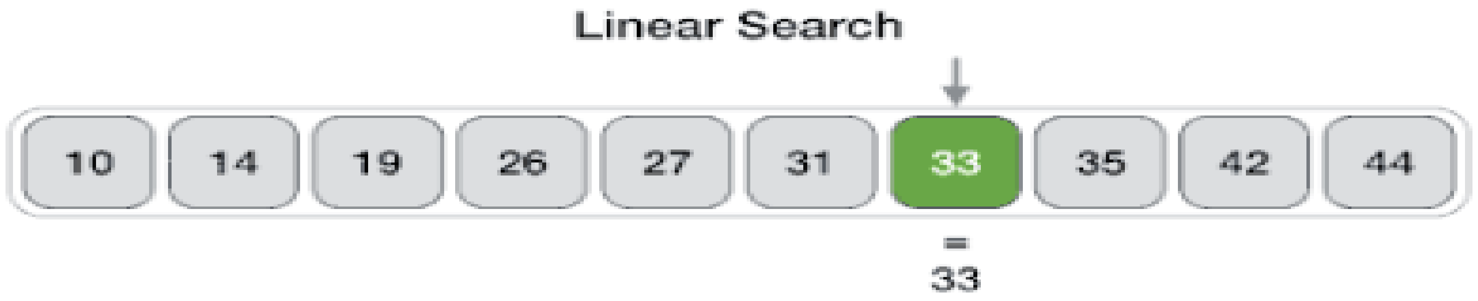


Image source : Youtube video

Algorithm Steps

```
Linear Search ( Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Image source : Google

Linear Search

```
procedure linear_search (list, value)
    for each item in the list
        if match item == value
            return the item's location
        end if
    end for
end procedure
```

Binary Search

- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer.
- For this algorithm to work properly, the data collection should be in the sorted form.

Binary Search

- Binary search looks for a particular item by comparing the middle most item of the collection.
- If a match occurs, then the index of item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the subarray reduces to zero.

Binary Search

- For a binary search to work, it is mandatory for the target array to be sorted.
- We shall learn the process of binary search with a pictorial example.
- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

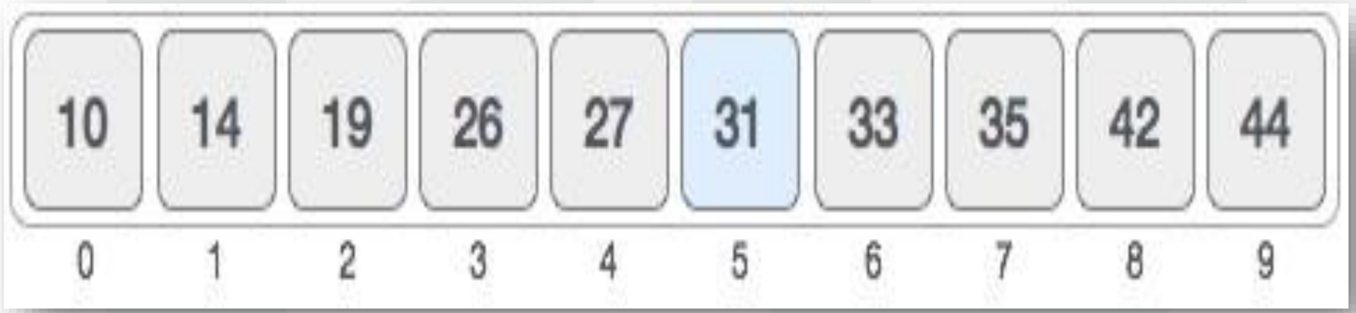


Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Binary Search

First, we shall determine half of the array by using this formula

mid = low + high / 2

Here it is,

(0 + 9) / 2 = 4

(integer value of 4.5).

So, 4 is the mid of the array.

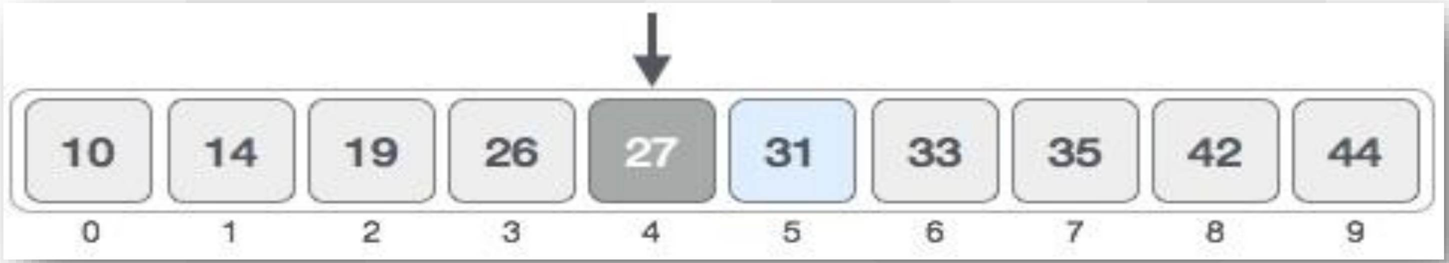


Image source : Google

Binary Search

Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

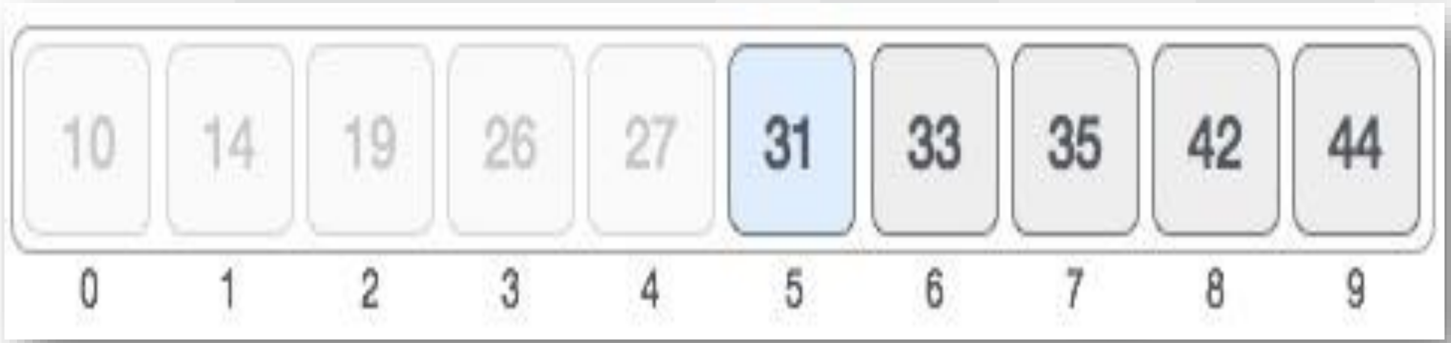


Image source : Google

Binary Search

- We change our low to mid + 1 and find the new mid value again.
- low = mid + 1
- mid = low + high / 2
- Our new mid is 7 now. We compare the value stored at location 7 with our target value 31./ 2

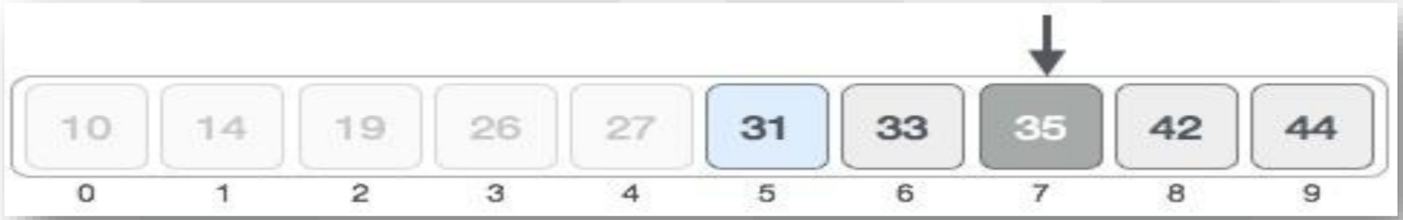
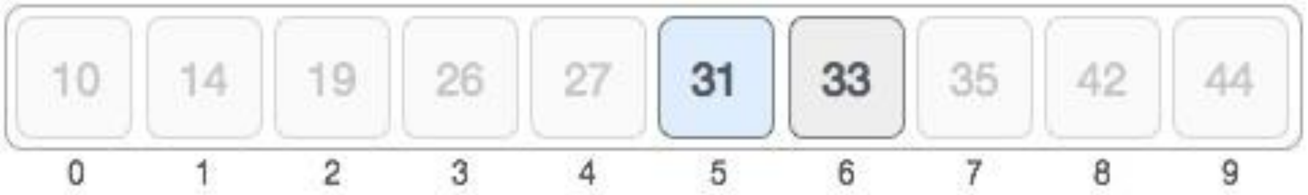


Image source : Google

Binary Search

- The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



- Hence, we calculate the mid again. This time it is 5.

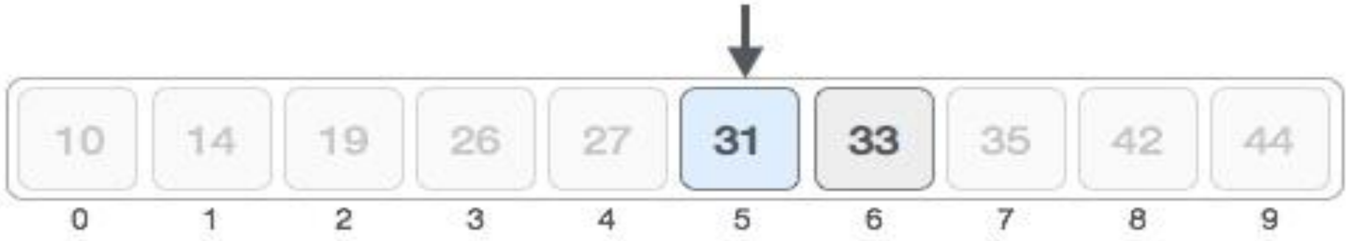
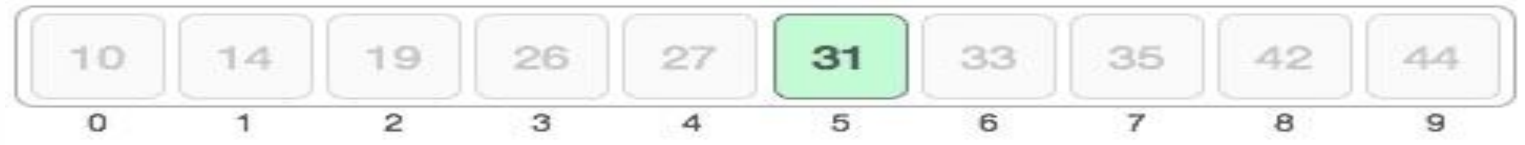


Image source : Google

Binary Search

- We compare the value stored at location 5 with our target value. We find that it is a match.



- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Image source : Google

Binary Search Algorithm

```
int binarySearch(int low,int high,int key)
 {
 while(low<=high)
{ int mid=(low+high)/2;
if(a[mid]<key)
{
 low=mid+1;
}
 else if(a[mid]>key)
{
high=mid-1;
}
else
{
 return mid;
}
}
return -1; //key not found
}
```

Image source : Google

File Organization

- A file organization refers to the organization of the data of a file into records, blocks and access structures - This includes the way the records and blocks are placed on the storage medium and interlinked.

OR

- File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record.
- In simple terms, Storing the files in certain order is called file Organization.

\

\

File Organization

- File organization ensures that records are available for processing.
- For example, if we want to retrieve employee records in alphabetical order of name. Sorting the file by employee name is a good file organization. However, if we want to retrieve all employees whose marks are in a certain range, a file is ordered by employee name would not be a good file organization.

Types of Graph - Undirected Graph

The objectives of computer based file organization:

- Ease of file creation and maintenance
- Efficient means of storing and retrieving information.

The various file organization methods are:

1. Sequential access file organization.
2. Direct or random access file organization.
3. Index sequential access file organization.

Operations on files

1) Read Operation: Meant To Read the information which is Stored into the Files.
2) Write Operation: For inserting some new Contents into a File.
3) Rename or Change the Name of File.
4) Copy the File from one Location to another.
5) Sorting or Arrange the Contents of File.
6) Move or Cut the File from One Place to Another.
7) Delete a File
8) Execute Means / Run Means File Display Output.

Image source : Youtube video

File Organization- selection criteria

The selection of a particular method depends on:

- Type of application.
- Method of processing.
- Size of the file.
- File inquiry capabilities.
- File volatility.
- The response time.

Image source : Google

Sequential access method

- Here the records are arranged in the ascending or descending order or chronological order of a key field which may be numeric or both.
- Since the records are ordered by a key field, there is no storage location identification.
- Here, to have an access to a particular record, each record must be examined until we get the desired record.
- Sequential files are normally created and stored on magnetic tape using batch processing method.
- In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.

Sequential access method cont.

Advantages:

- Simple to understand.
- Easy to maintain and organize
- Loading a record requires only the record key.
- Relatively inexpensive I/O media and devices can be used.
- Easy to reconstruct the files.

Image source : Google

Sequential access method cont.

Disadvantages:

- Entire file must be processed, to get specific information.
- Sequential file is time consuming process.
- It has high data redundancy.
- Random searching is not possible.
- Transactions must be stored and placed in sequence prior to processing.
- Data redundancy is high, as same data can be stored at different places with different keys.
- Impossible to handle random enquiries.
- The proportion of file records to be processed is high.

Direct access files organization

- (Random or relative organization)
- Files in his type are stored in direct access storage devices such as magnetic disk (DASD), using an identifying key.
- The identifying key relates to its actual storage position in the file. The computer can directly locate the key to find the desired record without having to search through any other record first.
- Here the records are stored randomly, hence the name random file.
- This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.

Image source : Google

Direct access files organization cont.

Advantages:

- Records can be immediately accessed for updation.
- Several files can be simultaneously updated during transaction processing.
- Transaction need not be sorted.
- Existing records can be amended or modified.
- Very easy to handle random enquiries.
- Most suitable for interactive online applications.

Direct access files organization cont.

Disadvantages:

- Data may be accidentally erased or over written unless special precautions are taken.
- Risk of loss of accuracy and breach of security.
- Direct access file does not provide back up facility.
- It is expensive.
- Less efficient use of storage space.
- Expensive hardware and software are required.
- High complexity in programming.

Image source : Google

DIGITAL LEARNING CONTENT

Indexed sequential access organization

- Here the records are stored sequentially on a direct access device i.e. magnetic disk and the data is accessible randomly and sequentially. It covers the positive aspects of both sequential and direct access files.
- The type of file organization is suitable for both batch processing and online processing.
- Here, the records are organized in sequence for efficient processing of large batch jobs but an index is also used to speed up access to the records.
- Indexing permit access to selected records without searching the entire file.

Image source : Google

Indexed sequential access organization

Advantages:

- In indexed sequential access file, sequential file and random file access is possible.
- It accesses the records very fast if the index table is properly organized.
- The records can be inserted in the middle of the file.
- It provides quick access for sequential and direct processing.
- It reduces the degree of the sequential search.

Indexed sequential access organization

Disadvantages:

- Slow retrieval, when compared to other methods.
- Indexed sequential access file requires unique index and periodic reorganization.
- Indexed sequential access file takes longer time to search the index for the data access or retrieval.
- It requires more storage space.
- It is expensive because it requires special software.
- It is less efficient in the use of storage space as compared to other file organizations.

Hash Tables

- A hash table is a data structure that supports the following operations:

1. insert(k) - puts key k into the hash table
2. search(k) - searches for key k in the hash table
3. remove(k) - removes key k from the hash table

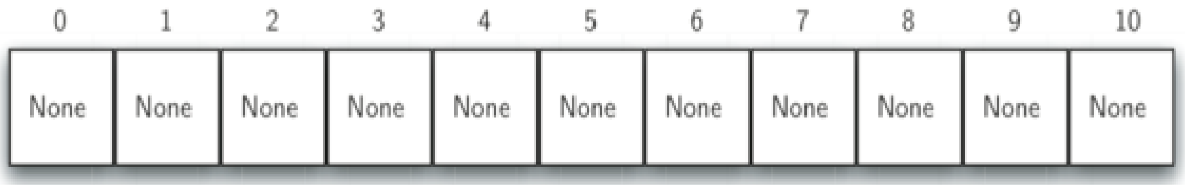
- In a well formed hash table, each of these operations take on average O(1) time, making hash tables a very useful data structure.

\

\

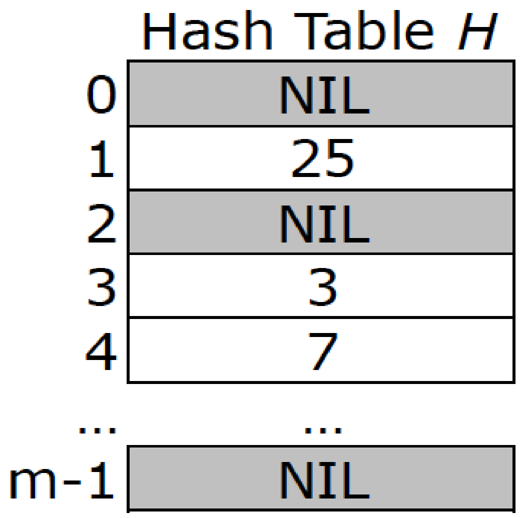
Hash Tables

- A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special Python value None. Figure 4 shows a hash table of size m=11m=11. In other words, there are m slots in the table, named 0 through 10.
- The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1.



Hash Tables

You can think of a hash table as a list of m slots. Inserting a key puts it in one of the slots in the hash table, deleting a key removes it from the slot it was inserted in, and searching a key looks in the slot the key would have been inserted into to see if it is indeed there. Empty slots are designated with a NIL value. The big question is figuring out which slot should a key k be inserted into in order to maintain the O(1) runtime of these operations.



Hash Functions

- Consider a function h(k) that maps the universe U of keys (specific to the hash table, keys could be integers, strings, etc. depending on the hash table) to some index 0 to m.
- We call this function a hash function. When inserting, searching, or deleting a key k, the hash table hashes k and looks at the h(k)th slot to add, look for, or remove the key.

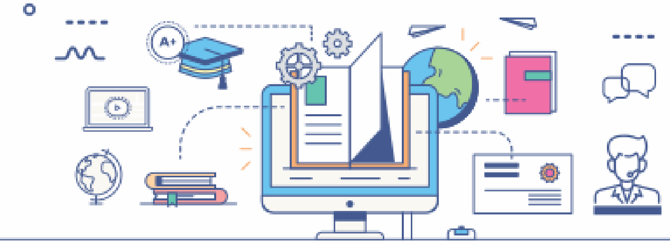
Image source : Youtube video

Hash Functions

A good hash function

- satisfies (approximately) the assumption of simple uniform hashing: each key is equally
- likely to hash to any of the m slots. The hash function shouldn't bias towards particular slots
- does not hash similar keys to the same slot (e.g. compiler's symbol table shouldn't hash
- variables i and j to the same slot since they are used in conjunction a lot)
- is quick to calculate, should have O(1) runtime
- is deterministic. h(k) should always return the same value for a given k

Image source : Google



Hash Functions

Example 1: Division method

- The division method is one way to create hash functions. The functions take the form

  $h(k) = k \bmod m$ (1)

- Since we're taking a value mod m, h(k) does indeed map the universe of keys to a slot in the hash table. It's important to note that if we're using this method to create hash functions, m should not be a power of 2. If m = 2p, then the h(k) only looks at the p lower bits of k, completely ignoring the rest of the bits in k. A good choice for m with the division method is a prime number

Image source : Google

Parul® University

Hash Functions

Example 2: Multiplication method

- The multiplication method is another way to create hash functions. The functions take the form

$$h(k) = bm(kA \text{ mod } 1)c \quad (2)$$

- where $0 < A < 1$ and (kA mod 1) refers to the fractional part of kA. Since $0 <$ (kA mod 1) <1, the range of $h(k)$ is from 0 to m. The advantage of the multiplication method is it works equally well with any size m. A should be chosen carefully.

Image source : Google

Hash Functions

Example 3: Folding method

- The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value.
- For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+0143+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

Image source : Google

Collision Resolution

- When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.
- One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as open addressing in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called linear probing.

Image source : Google

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in