BASUDEV GODABARI DEGREE COLLEGE, KESAIBAHAL

Department of Computer Science

"Self Study Module"

Module Details:

- **Class – 6th Semester**
- **Subject Name: COMPUTER SCIENCE**
- **Paper Name : "Algorithm Design Techniques"**

UNIT-2: STRUCTURE

Searching and Sorting: Analysis of Linear Search, Binary Search, Merge Sort and Quick Sort, Heap Sort. Hashing: Hash functions, Hash table, Collision resolution: Chaining and Open Addressing (Linear probing, Quadratic probing, Double hashing)

Learning Objective

Sorting and searching algorithms are fundamental tools for analyzing and manipulating data. They can help you find, organize, and compare information in various contexts and scenarios. In this article, you will learn how to apply some common sorting and searching algorithms to real-world problems or scenarios.

You Can use the Following Learning Video link related to above topic:

https://youtu.be/YJeoQBevNVo?list=PLDzeHZWIZsTp4pb_WBRahP1tnipLuX9qM
https://youtu.be/UdO2NeHB46c?list=PLDzeHZWIZsTp4pb_WBRahP1tnipLuX9qM
https://youtu.be/zOhUavxlzw4?list=PLDzeHZWIZsTp4pb_WBRahP1tnipLuX9qM

You Can also use the following Books

Text books

1. Introduction to Algorithms, by Thomas H, Cormen, Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, PHI.

Reference books

1. Algorithm Desgin, by Jon Kleinberg, Eva Tardos.

https://www.pdfdrive.com/

Quick Sort

Quick sort is a comparison based sorting algorithm that follows the divide and conquer technique to sort the arrays. In quick sort, we usually use a **pivot (key)** element to compare and interchange the position of the element based on some condition. When a pivot element gets its fixed position in the array that shows the termination of comparison & interchange procedure. After that, the array divides into the two sub arrays. Where the first partition contains all those elements that are less than pivot (key) element and the other parts contains all those elements that are greater than pivot element. After that, it again selects a pivot element on each of the sub arrays and repeats the same process until all the elements in the arrays are sorted into an array.

Algorithm of Quick Sort

Partition (A, p, r)

```
1. X <- A[r]
2. I <- p-1
3. For j <- p to r -1
4. Do if A[j] <= x
5. Then I <- I + 1
6. Exchange A[i] <-> A[j]
7. Exchange A[I + 1] <--> A[r]
8. Return I + 1
```

Quicksort (A, p, r)

```
1. While (p < r)
2. Do q <- Partition (A, p, r)
3. R <- q-1
4. While (p < r)
5. Do q <- Partition (A, p, r)
6. P <- q + 1
```

Steps to sort an array using the quick sort algorithm

Suppose, we have an array X having the elements X[1], X[2], X[3],...., X[n] that are to be sort. Let's follow the below steps to sort an array using the quick sort.

Step 1: Set the first element of the array as the **pivot** or key element. Here, we assume pivot as X[0], **left** pointer is placed at the first element and the **last** index of the array element as **right**.

Step 2: Now we starts the scanning of the array elements from right side index, then

If X[key] is less than X[right] or if X[key] < X[Right],

1. Continuously decreases the right end pointer variable until it becomes equal to the key.
2. If X[key] > X[right], interchange the position of the key element to the X[right] element.
3. Set, key = right and increment the left index by 1.

Step 3: Now we again start the scanning of the element from left side and compare each element with the key element. X[key] > X[left] or X[key] is greater than X[left], then it performs the following actions:

1. Continuously compare the left element with the X[key] and increment the left index by 1 until key becomes equal to the left.
2. If X[key] < X[left], interchange the position of the X[key] with X[left] and go to step 2.

Step 4: Repeat Step 2 and 3 until the X[left] becomes equal to X[key]. So, we can say that if X[left] = X[key], it shows the termination of the procedures.

Step 5: After that, all the elements at the left side will be smaller than the key element and the rest element of the right side will be larger than the key element. Thus indicating the array needs to partitioned into two sub arrays.

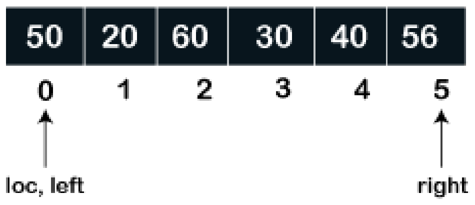
Step 6: Similarly, we need to repeatedly follow the above procedure to the sub arrays until the entire array becomes sorted.

Let's see an example of quick sort.

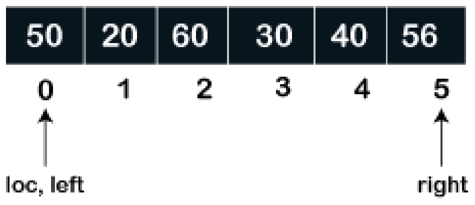
Example: Consider an array of 6 elements. Sort the array using the quick sort.

arr[] = {50, 20, 60, 30, 40, 56}

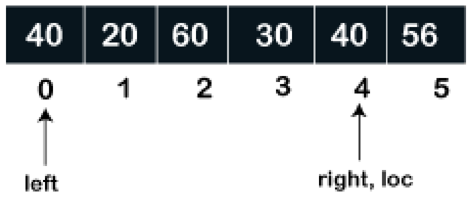
Here pivot element is 50 set loc & left is 0, right is 5



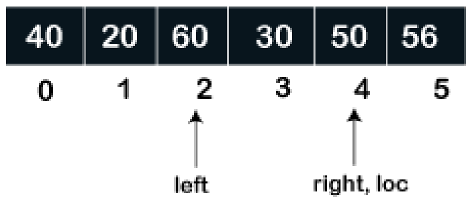
Scanning of the element from right and decrease the variable index,if a[loc] <a[right], we get



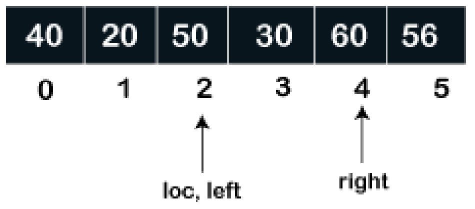
Here, a[loc]> aright, interchange the elements with pivot:



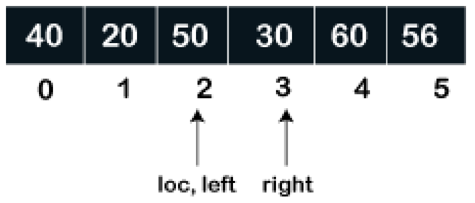
Start scanning of the elements from left and increment the left pointer, until a[loc]> a[left]:



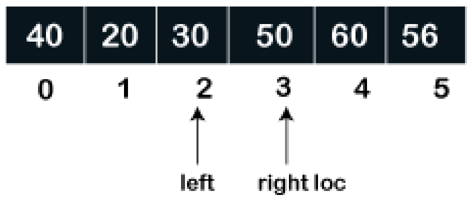
When a[loc] <a[left], interchange the element:



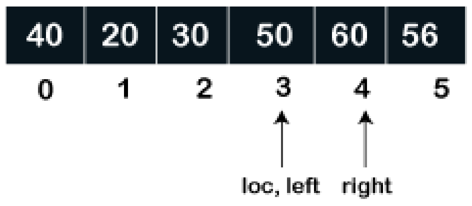
Start scanning of the element from right side:



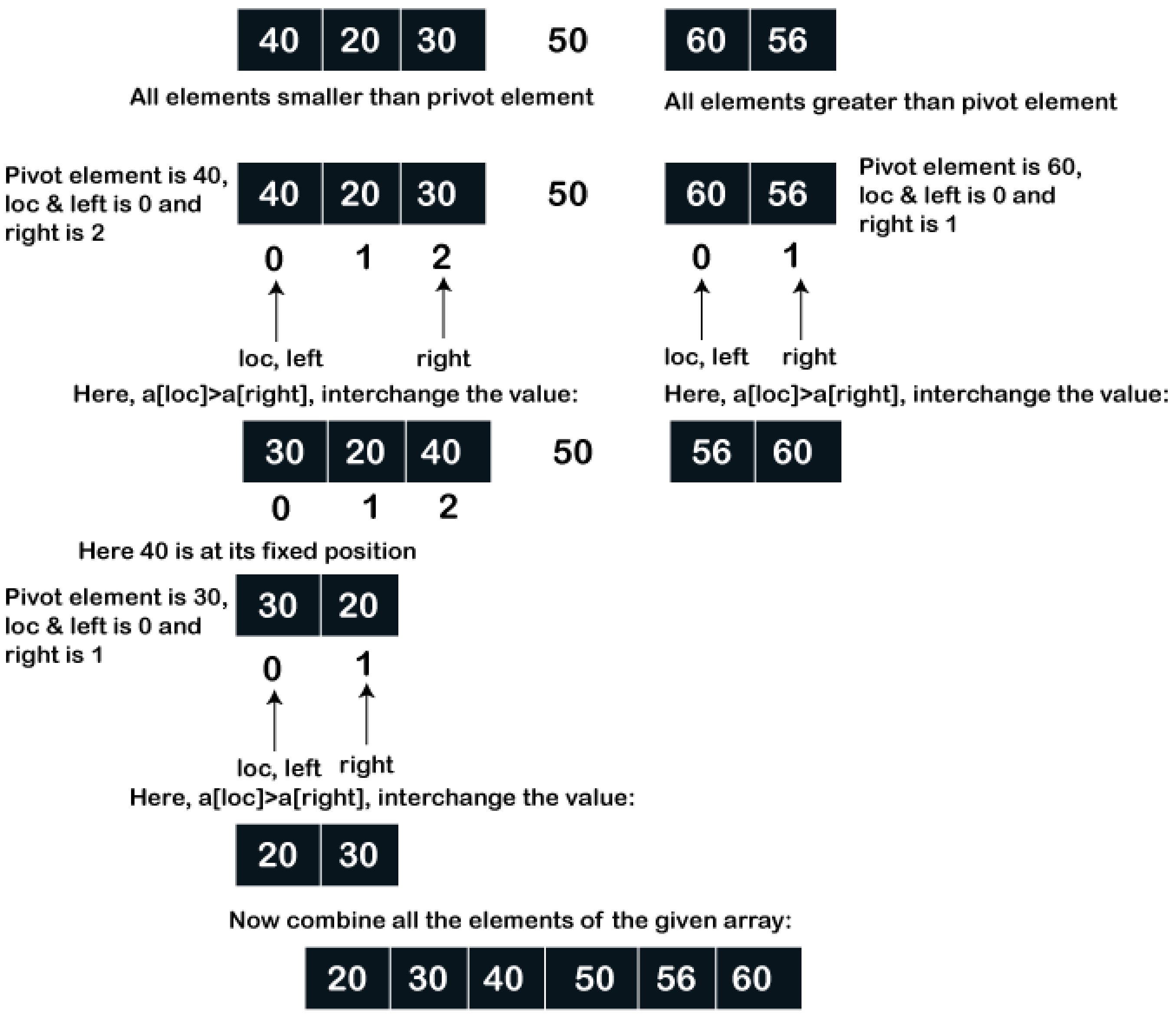
Here, a[loc]> a[right], interchage the element:



Now start Scanning of the element from left until a[loc] > a[left] then increment the left pointer:



In the above array, 50 is in its right place. So, we divided the elements that are less than pivot in one sub array and the elements that are larger than the pivot element in another sub array.



40 20 30 50 60 56
All elements smaller than privot element
All elements greater than pivot element
Pivot element is 40, loc & left is 0 and right is 2
Pivot element is 60, loc & left is 0 and right is 1
loc, left
right
Here, a[loc]>a[right], interchange the value:
30 20 40 50 56 60
Here 40 is at its fixed position
Pivot element is 30, loc & left is 0 and right is 1
20 30
Now combine all the elements of the given array:
20 30 40 50 56 60

Hence, we get the sorted array.

Let's implement the above logic in C program.

Quick.c

```
#include <stdio.h>
int division_of_array(int x[], int st, int lt); // declaration of functions
void quick_sort(int x[], int st, int lt);
void main()
{
   int i;
   int ar[6] = {50, 20, 60, 30, 40, 56}; // given array elements
   quick_sort(ar, 0, 5);  // it contains array, starting index and last index as an parameters.
   print(" Sorted Array is : \n");
   for(i=0; i<6; i++)
   {
      printf("Array = %d\t", ar[i]);
   }
   int division_of_array(int x[], int st, int lt)
   {
      int left, right, temp, key, flag;
      key = left = st; // initially these variables are in same place.
```

```
            right = lt;
             flag = 0;
            while(flag != 1)
             {
                while((x[key] <= x[right]) && (key != right))
                 right--;
                if(key == right) // if pivot element is equal to the last index.
                 flag = 1;
                else if(x[key] > x[right]) // if pivot is greater than last index.
                 {
                   temp = x[key];
                    x[key] = x[right];
                   x[right] = temp;
                    key = right;
                }
                 if (flag != 1)
                {        /* Repeat the while loop until pivot is greater than left and equal to the left. */
                    while((x[key] >= x[left]) && (key != left))
                   left++;
                    if(key == left)
                   flag = 1;
                    else if(x[key] < x[left])
                   {
                         temp = x[key];
                       x[key] = x[left];
                         x[left] = temp;
                       key = left;
                    }
                }
             }
            return key;
         }
     }void quick_sort(int x[], int st, int lt)
     {
         int key;
          if(st<lt) // left index value is greater than right index.
         {
             key = division_of_array(x, st, lt);
            quick_sort(x, st, key-1);
             quick_sort(x, key+1, lt);
         }

     }
```

Output:

```
Sorted Array is:
Array = 50  20  60  30  40  56
```

Merge sort

Merge sort is a most important sorting techniques that work on the divide and conquer strategies. It is the most popular sorting techniques used to sort data that is externally available in a file. The merge sort algorithm divides the given array into two halves (N/2). And then, it recursively divides the set of two halves array elements into the single or individual elements or we can say that until no more division can take place. After that, it compares the corresponding element to sort the element and finally, all sub elements are combined to form the final sorted elements.

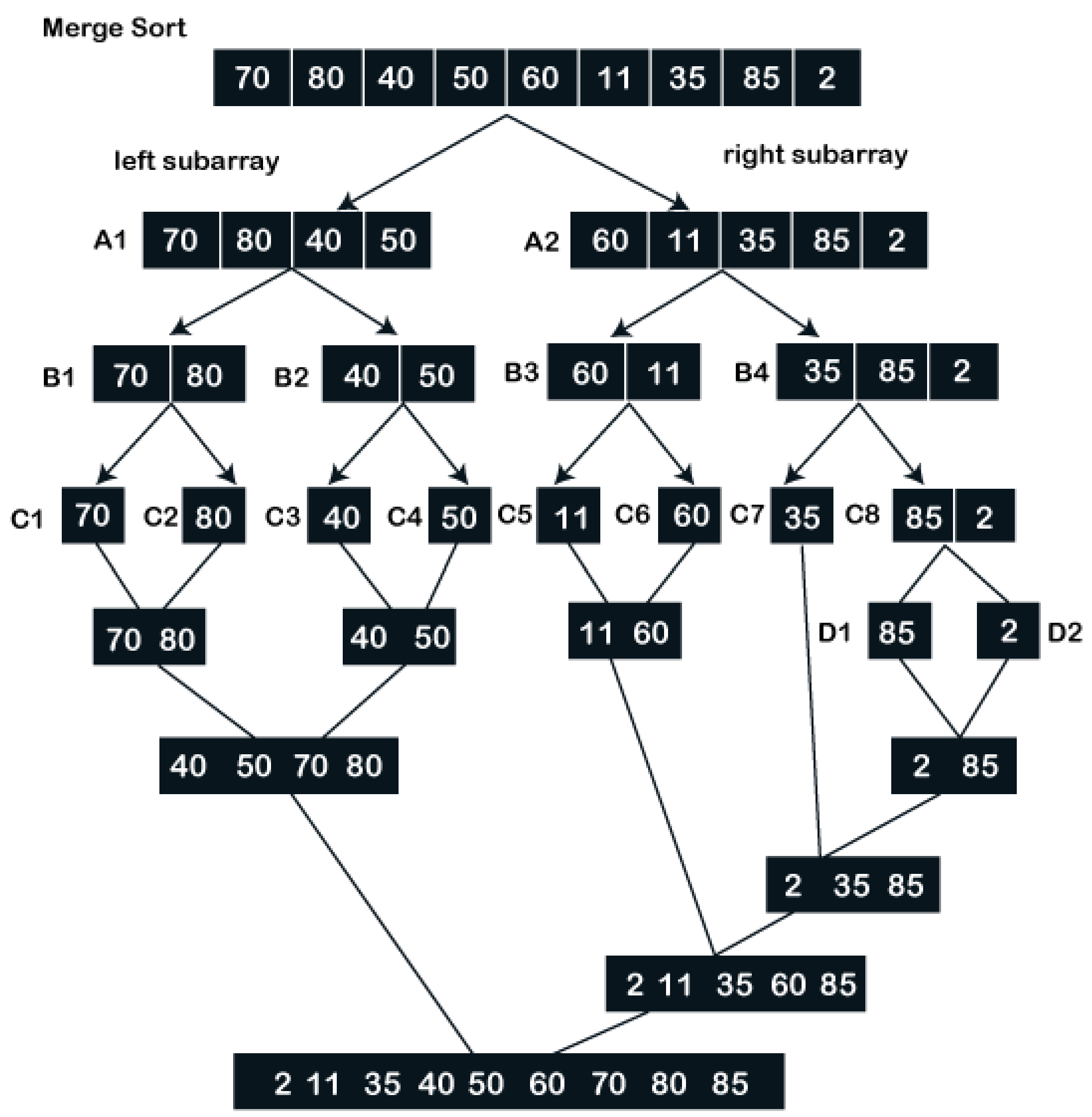
Steps to sort an array using the Merge sort algorithm

1. Suppose we have a given array, then first we need to divide the array into sub array. Each sub array can store 5 elements.
2. Here we gave the first sub array name as A1 and divide into next two subarray as B1 and B2.
3. Similarly, the right sub array name as A2 and divide it into next two sub array as B3 and B4.
4. This process is repeated continuously until the sub array is divided into a single element and no more partitions may be possible.
5. After that, compare each element with the corresponding one and then start the process of merging to arrange each element in such a way that they are placed in ascending order.
6. The merging process continues until all the elements are merged in ascending order.

Let's see an example of merge sort.

Example: Consider an array of 9 elements. Sort the array using the merge sort.

arr[] = {70, 80, 40, 50, 60, 11, 35, 85, 2}



Hence, we get the sorted array using the merge sort.

Let's implement the above logic in a C program.

Merge.c

```
#include <stdio.h>
//#include <conio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int end)
{
    int i, j, k;
    int a1 = m - l + 1;
    int a2 = end - m;

    // create temp subarray
    int sub1[a1], sub2[a2];

    // Store data to temp subarray subArr1[] and subArr2[]
    for (i = 0; i < a1; i++)
        sub1[i] = arr[l + i];
    for (j = 0; j < a2; j++)
        sub2[j] = arr[m + 1 + j];

    // Merge the temp array into the original array arr[]
    i = 0;
    j = 0;
    k = l;
    while (i < a1 && j < a2)
    {
        if (sub1[i] <= sub2[j])
        {
            arr[k] = sub1[i];
            i++;
        }
        else
        {
            arr[k] = sub2[j];
            j++;
        }
        k++;
    }

    // copry the rest element of subArr1[], if some element is left;
while (i < a1)
{
    arr[k] = sub1[i];
    i++;
    k++;
}

    // copry the rest element of subArr2[], if some element is left;
while (j < a2)
{
    arr[k] = sub2[j];
    j++;
    k++;
    }
}

/* st represents the first index and end represents the right index of the subarray of arr[] to be sorted
using merge sort. */
void mergeSort(int arr[], int l, int end)
{
    if (l < end)
    {
        int m = l + (end - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, end);

        merge(arr, l, m, end);
    }
}

// Function to print the merge sort.
void mergePrint(int Ar[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d \t", Ar[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 70, 80, 40, 50, 60, 11, 35, 85, 2 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf(" Predefined Array is \n");
    mergePrint(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\n Sorted array using the Merge Sort algorithm \n");
    mergePrint(arr, arr_size);
    return 0;
}
```

Output:

```
Predefined Array is
70      80      40      50      60      11      35      85      2

 Sorted array using the Merge Sort algorithm
2       11      35      40      50      60      70      80      85
```

Quick Sort vs. Merge Sort

S.N.	Parameter	Quick Sort	Merge Sort

1.	**Definition**	It is a quick sort algorithm that arranges the given elements into ascending order by comparing and interchanging the position of the elements.	It is a merge sort algorithm that arranges the given sets of elements in ascending order using the divide and conquer technique, and then compare with corresponding elements to sort the array.
2.	**Principle**	It works on divide and conquer techniques.	It works on divide and conquer techniques.
3.	**Partition of elements**	In quick sort, the array can be divide into any ratio.	Merge sort partition an array into two sub array (N/2).
4.	**Efficiency**	It is more efficient and work faster in smaller size array, as compared to the merge sort.	It is more efficient and work faster in larger data sets or array, as compare to the quick sort.
5	**Sorting method**	It is an internal sorting method that sort the array or data available on main memory.	It is an external sorting method that sort the array or data sets available on external file.
6	**Time complexity**	Its worst time complexity is O (n2).	Whereas, it's worst time complexity is O (n log n).
7	**Preferred**	It is a sorting algorithm that is applicable for large unsorted arrays.	Whereas, the merge sort algorithm that is preferred to sort the linked lists.
8	**Stability**	Quick sort is an unstable sort algorithm. But we can made it stable by using some changes in programming code.	Merge sort is a stable sort algorithm that contains two equal elements with same values in sorted output.
9	**Requires Space**	It does not require any additional space to perform the quick sort.	It requires the additional space as temporary array to merge two sub arrays.
10.	**Functionality**	Compare each element with the pivot until all elements are arranged in ascending order.	Whereas, the merge sort splits the array into two parts (N/2) and it continuously divides the array until an element is left.

How do you apply sorting and searching algorithms to real-world problems or scenarios?

Sorting and searching algorithms are fundamental tools for analyzing and manipulating data. They can help you find, organize, and compare information in various contexts and scenarios. In this article, you will learn how to apply some common sorting and searching algorithms to real-world problems or scenarios.

Sorting algorithms

Sorting algorithms are methods of arranging data in a certain order, such as ascending, descending, alphabetical, or numerical. Sorting algorithms can have different advantages and disadvantages depending on the size, type, and distribution of the data. Some examples of sorting algorithms are bubble sort, insertion sort, merge sort, quick sort, and heap sort. Sorting algorithms can help you sort data for easier access, analysis, or presentation. For example, you can use a sorting algorithm to sort a list of names alphabetically, a list of numbers from smallest to largest, or a list of products by price or popularity.

Searching algorithms

Searching algorithms are methods of finding a specific element or value in a collection of data, such as an array, a list, a tree, or a graph. Searching algorithms can have different approaches and complexities depending on the structure, size, and order of the data. Some examples of searching algorithms are linear search, binary search, depth-first search, breadth-first search, and hash table search. Searching algorithms can help you locate data for retrieval, modification, or verification. For example, you can use a searching algorithm to find a name in a phone book, a number in a sorted array, a path in a maze, or a keyword in a document.

Sorting and searching in real-world scenarios

Sorting and searching algorithms can be applied to various real-world scenarios that involve data analysis and manipulation. For instance, in e-commerce, these algorithms can help customers find and compare products based on criteria such as price, rating, category, or availability. Quick sort and binary search algorithms can be used for this purpose. In social media, sorting and searching algorithms can help users find and interact with content and people based on their preferences, interests, or connections. Merge sort and hash table search algorithms can be used for this purpose. In education, sorting and searching algorithms can help students and teachers organize and access information and resources based on their needs, goals, or levels. Insertion sort and depth-first search algorithms can be used for this purpose. In health care, sorting and searching algorithms can help doctors and patients diagnose and treat diseases and conditions based on their symptoms, tests, or treatments. Heap sort and breadth-first search algorithms can be used for this purpose.

Benefits of sorting and searching algorithms

Sorting and searching algorithms can provide various benefits for data analysis and manipulation, such as improved efficiency, accuracy, and flexibility. These algorithms can reduce the time and space complexity of data processing, eliminate errors and inconsistencies in data, and adapt to different data types, structures, and orders. Additionally, sorting and searching algorithms can inspire new ideas and insights for data exploration and innovation, enabling data-driven decision making and problem solving.

Challenges of sorting and searching algorithms

Sorting and searching algorithms can present some challenges for data analysis and manipulation. These algorithms can involve complex logic, mathematics, and programming, and require a deep understanding of data structures and algorithms concepts and principles. Additionally, there are trade-offs between time and space complexity, stability and instability, and simplicity and sophistication that must be carefully evaluated and selected. Furthermore, sorting and searching algorithms can have some limitations and drawbacks in terms of data quality, security, and privacy, and thus require proper handling and protection of data and information.