Arrays

An array is a group of instances of same datatype referred as a single element.

The elements are stored at contigoous memory locations.

The elements can be accessed using the position which is 0 based indexing.

Need of Array

While dealing with large number of instances it is difficult with a ~~single~~ normal variable ie, managing and retrieving.

To fulfill this need arrays comes into play,

Advantage

(i) Random accessing of data is allowed.
(ii) Sorting can be done efficiently.

Disadvantage.

(i) Size of array should be defined and it cannot be increased.

Array elements can be accessed in two ways:

(i) arr[i] where i = 0, 1, 2 ... so on.

(ii) i[arr] where i = 0, 1, 2 ... so on.

Arrays / Pointer

↳ Arrays are used to store element whereas Pointer stores the address of ~~elements~~ variables.

↳ size of operates gives size of array in case of arrays but it gives size of data type of pointers.

↳ Arrays can be initialized at declaration while pointers cannot

↳ Arrays are static while pointers are dynamic.

↳ Iteration :

arr[2] == *(arr + 2)

↓ ↓

arrays pointers

Dynamic Memory Allocation Using Pointes

It is used to allocate memory at run time rather than compile time

Sample Code:

```
int size;
cin >> size;

int *ptr;

ptr = new int[size];

for(int i=0; i<size; i++) {

  cout<< ptr[i];

}
```

Dynamic Memory Allocation.

Searching Algos

I. LINEAR SEARCH

(i) It is based on traversing the whole array to find the element.

(ii) Visiting each positions of array in order to search the element.

1st Approach : BASIC

```
void linearSearch ( int arr[], int num, int size ) {

    bool flag = false;

  for( int i=0; i< size ; i++) {
      if ( arr[i] == num ) {
          cout<<" found";
          flag = true;
          break;
        }
    }

    if (!flag)  cout<<" Not found;
}
```

2nd Apprach : Recursive

```
void linearSearch (int arr[], int num, int size) {

    if (size == 0) { return;
        cout << " Not found ";
        return;
    }

    if (arr[size-1] == num) {
        cout << " found " << arr[size-1];
        return;
    }

    int ans = linearSearch (arr, num, size-1);
    cout << "found " << ans;

}
```

Time Complexity ⟶ O(n)

II. Binary Search

(i) It is based on repeatedly dividing the array into two halfs and search in either of the two.

(ii) It works only when array is sorted.

(iii) No need to travel every element of array.

1st Approach : Iterative.

```
int binarySearch ( int arr[ ], int size, int key ).
{
    int low = 0;
    int high = size - 1;

    while ( low <= high ) {

        int mid = low + (high - low) / 2;

        if ( key == arr[mid] ) return mid;
        else if ( key > arr[mid] ) low = mid + 1;
        else if ( key < arr[mid] ) high = mid - 1;
    }

    cout << " Not found ";
    return 0;
}
```

2nd Approach : Recursive

```
int binarySearch( int arr[], int size, int key, int low,
int high) {
  if( low <= high) {
    int mid = low + (high - low) / 2;
    if( key == arr[mid]) return mid;
    if( key < arr[mid]) return binarySearch(arr, size, key,
    low, mid-1);
    if( key > arr[mid]) return binarySearch( arr, size, key,
    mid+1, high);
  }
  cout << " Not found ";
  return 0;
}
```

Time Complexity ⟶ O(log n)

III. JUMP SEARCH

(i) It is used for sorted arrays. Better than linear Search in terms of Time Complexity.

(ii) Jump should be of size $\sqrt{\text{size of array}}$

(iii) Performance linear Search < Jump Search < Binary Search

```
int JumpSearch( int arr[], int size, int k ) {

    int jump = √size  or  sqrt(size);
    int low = ;
    for(int i = 0; i < size; i += jump) {

    if( k == arr[i] )  return i;

    if( k > arr[i] )  low = i;

    if( k < arr[i] )  break;
}

for( int i = low; i < size; i++ ) {

    if( arr[i] == k )  return i;
}
```

return 0;

}

Steps involved in this algorithm:

(i) Consider the size of jump by $\sqrt{size\_of\_array}$.

(ii) Run the loop with jumping of size as determined.

(iii) If the element found return the index.

(iv) If the element (current) is less than the key, then move the low iterator to it.

(v) If the element (current) is greater, break the loop.

(vi) Now the low iterator points to the position where the searching should start.

(vii) Run the loop again and search for the key.

Time Complexity $\longrightarrow O(\sqrt{n})$

IV. INTERPOLATION SEARCH (Formula Based)

(i) It is an improvement of Binary Search.

(ii) It finds the position instead of mid point. The position is derived based on key to be searched.

(iii) According to the position's value the element is compared

(iv) To find the position, A formula is needed:

$$pos = low + \frac{(key - arr[low])}{arr[high] - arr[low]} * (high - low)$$

where low & high is calculated as per Binary Search Algo.

Code:

```
void interSort ( int arr[], int n, int key ) {
int low = 0, high = n-1;
```

```
while (low <= high) {
int pos = low + (((double) (high - low) / (arr[high] - arr[lo
                )) * (key - arr[low]));

if (key == arr[pos]) { cout << "found"; return; }

if (key > arr[pos]) low = pos+1;
else high = pos-1;
}
}
```

V. Interpolation Vs Binary Search

(i) It works of sorted & uniformly distributed array and unlike binary search it does not find the middle element on every iteration.

(ii) Despite of that, it finds a position which is most likely close to the element or key we are searching for.

(iii) Most of the cases it takes $O(\log(\log(n)))$ time complexity.

Linear Vs Binary Search

(i) In case of linear Search, the array need not to be sorted unlike binary Search.

(ii) Time complexity of Binary Search is $O(\log n)$ which is better than linear Search $O(n)$.

(iii) Binary Search is pretty fast compared to linear search.

SORTING Algo

1. Selection Sort :

(i) It is based on chosing a position and placing the smallest value by comparing its value to other ahead values.

(ii) For every iteration, the smallest element is swaped with the current position's element.

(iii) While reaching the second last element of array, it gets already sorted.

Code :

```
void selectionSort ( int arr[], int size ) {

  int min ;

  for ( int i = 0 ; i < size - 1 ; i++ ) {

    min = i ;
```

```
for( int j = i+1; j< size ; j++ ) {
    if( arr[j] < arr[min] ) {
        min = j;
    }
}
if( min != i ) {
    int temp = arr[min];
    arr[min] = arr[i];
    arr[i] = temp;
}
```

Time Complexity ⟶ $O(n^2)$

II. Insertion Sort

(i) In this method, the ahead element is compared with the left subarray.

(ii) On every iteration, the left subarray gets sorted such that at last the whole array gets sorted.

(iii) It is sufficient for small data values.

Code:

```
void insertionSort (int arr[], int size) {

int j = 0, key;

for(int i=1; i<size; i++) {

  key = arr[i];
  j = i-1;

while( j>=0 && arr[j] > key ) {

    arr[j+1] = arr[j];
    j = j-1;
}
```

```
    arr[j+1] = key;
  }
}
```

Time Complexity ⟶ $O(n^2)$

III. Bubble Sort

(i) In this sorting, every element is compared to its next element if found smaller the swaping takes place.

Code:

```
void bubbleSort(int arr[], int size){
  for(int i=0; i<size; i++){
    for(int j=i+1; j<(size-i-1); j++){
      if(arr[j] < arr[j+1]){
        swap(arr[j], arr[j+1]);
      }
    }
  }
}
```

Time Complexity ⟶ $O(n^2)$

IV. Optimized Bubble Sort.

(i) It basically checks after every iteration the array is sorted or not.

(ii) If it get already sorted, the loop breaks and answer is showed.

Code:

```
void bubbleSort ( int arr[], int size ) {
    bool flag;
for ( int i = 0; i < size; i++ ) {
        flag = false;
    for ( int j = 0; j < size - i - 1; j++ ) {
        if ( arr[j] > arr[j+1] ) {
            flag = true;
            swap ( arr[j], arr[j+1] );
        }
    }
    if ( flag == false ) break; // Optimization
}
```

I. Recursive Bubble Sort

Code:

```
void bubbleSort( int arr[], int size ) {

    if ( size == 1 ) return;

    bool flag = false;

    for( int i=0; i< size-1; i++ ) {

        if( arr[i] > arr[i+1] ) {
            flag = true;
            swap( arr[i], arr[i+1] );

        }
}

    if ( flag == false ) return;

    bubbleSort ( arr, size-1 );

}
```

VII. Recursive Insertion Sort

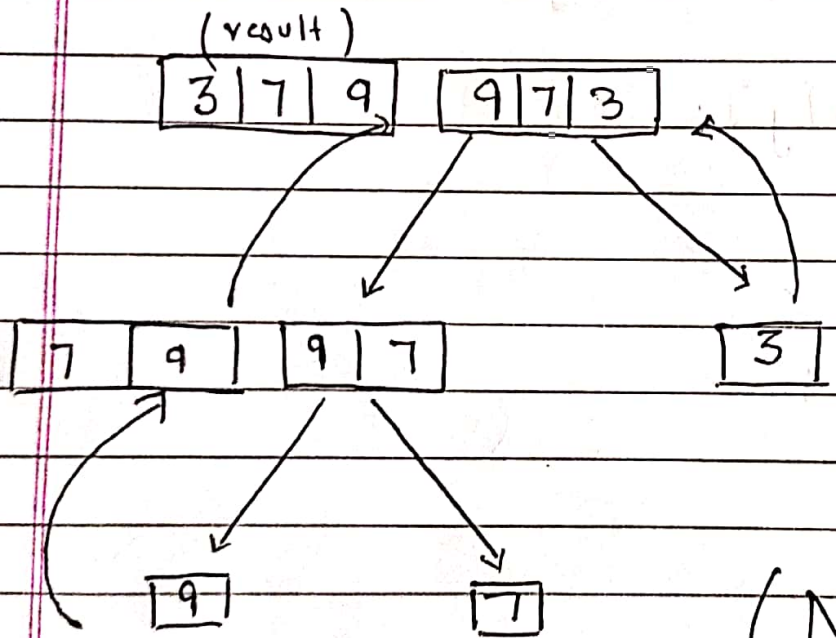
```
void insertionSort( int arr[], int size ) {
    if( n <= 1 ) return;
    insertionSort( arr, n-1 );      // Recursion
    int last = arr[n-1];            // last element Approach
    int j = n-2;
    while( j>=0 && arr[j] > last ) {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

VIII. Merge Sort

(i) It is based on Divide and Conquer strategy.

(ii) On every recursive call, divide the array into two halfs.

(iii) While returning from recursive calls, the two halves merges such that they are arranged in a sorted order.



(MERGE Sort)

Code :

```
void merge ( int arr[], int l , int m , int r ) {

int i = l;
int j = m+1;
int k = l;

int temp[5];

while ( i<=m && j<=r ) {

   if ( arr[i] <= arr[j] ) {

temp[k] = arr[i];
    i++; k++;
} else {

temp[k] = arr[j];
    j++, k++;
}
}

while ( i<=m ) {
temp arr[k] = arr[i];
    i++, k++;
}
```

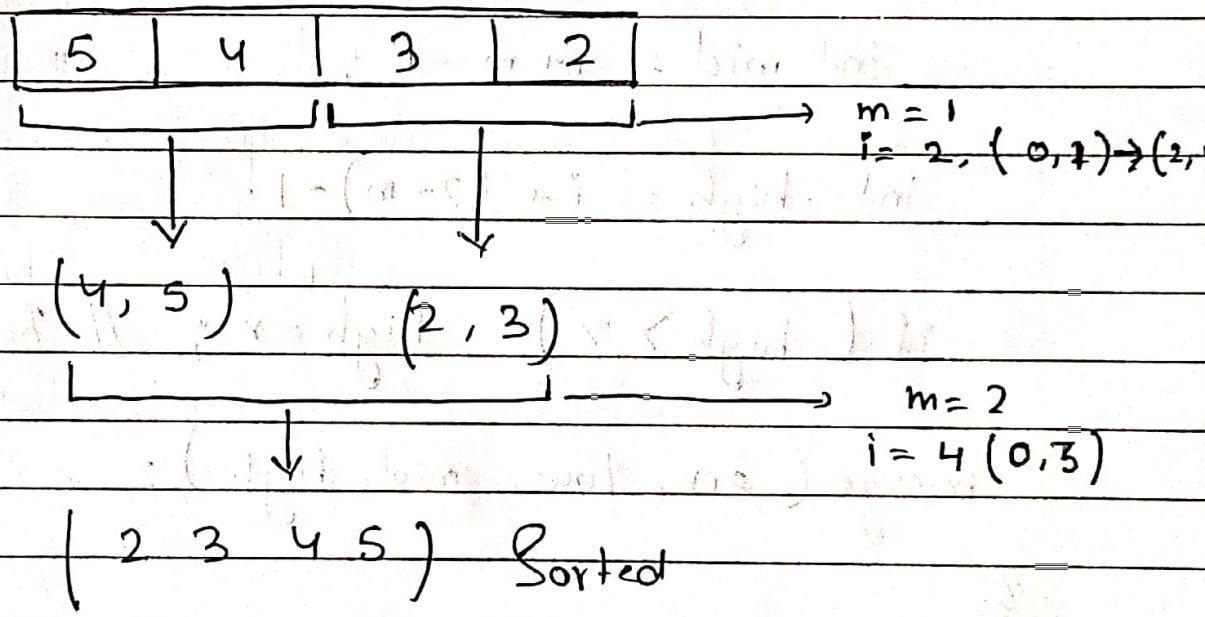
```
while ( j<=r ) {
temp[k] = arr[j];
  j++, k++;
}

for ( int i=l ; i<=r ; i++ ) {
    arr[i] = temp[i];
}
}

void mergeSort ( int arr[], int l , int r ) {
if ( l<r ) {
int mid = (l+r)/2 ;
mergeSort ( arr, l, mid ) ;
mergeSort ( arr, mid+1, r ) ;
merge ( arr, l, mid, r ) ;
}
```

Time Complexity → O(n log(n))

IX. 2-Way Merge Sort (Iterative Approach)

(i) It can be achieved by dividing the array by 1, 2, 4, 8 .... so on.

(ii) On each division perform merging in order to get the sorted array at last.



(iii) Merging function is same as of Recursive Approach.

(iv) On every window size, the inside iterator jumps accordingly such as by 2 (first time), by 4 (second time), by 8 (third time) and so on.

Code:

```
void mergeSort(int arr[], int l, int r) {
  for(m=1; m<5; m=m*2) {
    for(i=l; i<5; i+=m*2) {
      int low = i;
      int mid = i+m-1;
      int high = i+(2*m)-1;
      if(high > r) high = r; // Boundary condition
      merge(arr, low, mid, high);
    }
```

Time Complexity ⟶ $O(n \log(n))$

I. Quick Sort

(i) In this way of sorting, we have to take a concept of a pivot.

(ii) A pivot element can be selected as first or last element of array.

(iii) Our aim is to place the pivot at the centre of array such that left subarray contains smaller value than pivot and right subarray contains larger values.

(iv) After that we perform sorting to both the sides in order to get the result.

(v) The time complexity of both merge and quick sort is same but the space complexity of quick sort is better than merge.

Space Complexity → O(log(n))

Code:

```
int partition ( int arr[], int start, int end ) {

    int pivot = arr[end];
    int pIndex = start;

    for ( int i=start ; i<end; i++ ) {

        if ( arr[i] < pivot) {

            swap ( arr[i], arr[pIndex] );

        }
}

    swap ( arr[end], arr[pIndex] );

    return pIndex;

}


void quickSort ( int arr[], int start , int end ) {
    if ( start < end ) {
    int p= partition ( arr, start, end);
    quickSort ( arr, start, p-1);
    quickSort (arr, p+1, end);
}
```

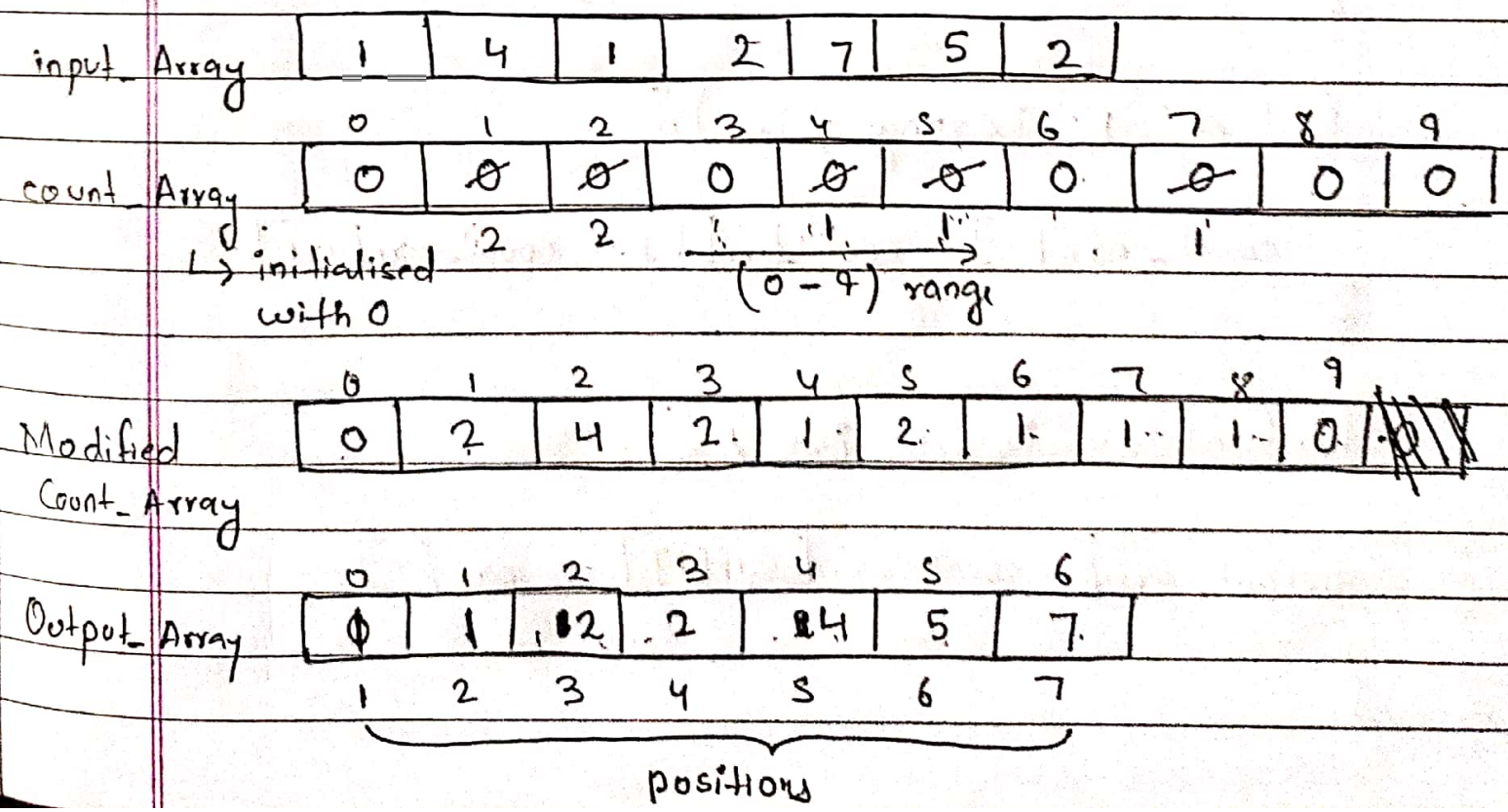
XII Counting Sort

(i) It is a sequence or occurance based sorting Algorithm.

(ii) To achieve this, an auxilary array is used to count the occurance of each element of array

(iii) An output array is also used for the purpose of storing the result.

(iv) The number of occurances of each element represent the position's range in the output array and the index of each occurance represents the actual element or value in the output array.



Code :

```
void countSort (int arr, int size, int range) {

int output_arr[size];
int count_arr[range];

for (int i=0; i<range; i++) {

  count_arr[i] = 0;

}

for (int i=0; i<size; i++) {

   ++ count_arr[arr[i]];

}

for (int i=1; i<range; i++) {

   count_arr[i] = count_arr[i] + count_arr[i-1];

}

for (int i=0; i<size; i++) {

output_arr[--count_arr[arr[i]] = arr[i];

}
```

```
for ( int i= 0; i< size; i++ ) {
    arr[i] = output_arr[i];
}
```

Time Complexity ⟶ $O(n+k)$ (range)

XIII. Radix Sort

(i) It is based on counting sort but it is implemented on individual digits from least significant digit to most significant.

(ii) Counting sort only works where range is between 0 to 9.

(iii) For large number, radix sort is used

(iv) In this we first find the maximum number in the given array.

(v) According to the number of digits present in that max number, the count sort is called.

(vi) In every count sort, the sorting is based on different digits i.e from least to most significant.

(vi) On the basis of digit, the temporary results are calculated.

(vii) At last, the final result is calculated.

Code:

```
void countingSort(int arr[], int n, int d){

    int output_arr[n];
    int count_arr[10] = {0};

    for(int i=0; i<n; i++){

        count_arr[(arr[i]/d)%10]++;

    }

    for(int i=1; i<10; i++){

        count_arr[i] += count_arr[i-1];

    }
```

```
for( int i= n-1; i>=0; i--) {
output_arr [ count_arr [(arr[i]/d) %10 ] -1 ] = arr[i];
count_arr[(arr[i]/d) %10] --;
}

for ( int i= 0; i<n; i++ ) {
  arr[i] = output_arr[i];
}

Void radixSort ( int arr[], int n ) {
int m = maxNum ( arr, n );
for ( int d = 1; m/d >0 ; d *= 10 ) {
  countingSort ( arr, n, d );
}
```

Time complexity → O d(n+k)

(d → no. of digits)

IX. Bucket Sort

(i) It is used when the elements are distributed uniformly over a range.

Like floating point numbers 0.0 to 1.0,

0.897, 0.656, 0.1234 .... so on.

Code:

```
void bucketSort( int arr[], int n ) {

vector<float> b[n];

for( int i=0; i<n; i++ ) {

  int bi = arr[i] * n;   // index
  b[bi].push_back( arr[i] );
}

for (int i=0; i<n; i++ ) {

  sort( b[i].begin(), b[i].end() );
}
  int index = 0;
```

```
for( int i=0; i<n; i++ ) {
    for( int j=0; j< b[i].size(); j++ ) {
        arr[index++] = b[i][j];
    }
}
```

X. Shell Sort

(i) Shell sort is a variation of bubble sort and Insertion sort.

(ii) It can be used on behalf of insertion sort when there is greater array

(iii) It reduces the number of operation compared to insertion sort.

(iv) It can be implemented by taking a gap or we can say window.

(v) At every iteration two elements are compared at both ends of window

(vi) Swapping takes place if larger value is at left.

Code:

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

Time Complexity → $O(n^2)$