WEEK 3

NEWTON SCHOOL

GCD

- **Greatest Common Divisor:** It is the highest number that completely divides two or more numbers. It is abbreviated for **GCD**. It is also known as the **Greatest Common Factor** (GCF) and the **Highest Common Factor** (HCF). It is used to simplify the fractions.
- GCD of two numbers doesn't change if smaller number is subtracted from a bigger number.
- How to Find the GCD
  - Write all the factors of each number.
  - Select the common factors.
  - Select the greatest number, as GCD.
- Eg: 12 and 8 GCD.

Factors of 12: 1, 2, 3, 4, 6, 12

Factors of 8: 1, 2, 4, 8

Common Factors: 1, 2, 4

Greatest Common Divisor: 4

TIME AND SPACE COMPLEXITY

- Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

TIME COMPLEXITY

- Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

  - Independent of the machine and its configuration, on which the algorithm is running on.
  - Shows a direct correlation with the number of inputs.
  - Can distinguish two algorithms clearly without ambiguity.

SPACE COMPLEXITY

- The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.
- *Auxiliary Space* is the extra space or temporary space used by an algorithm.
- *Space Complexity* of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

SEARCHING - LINEAR SEARCH

- Linear search is used to search a key element from multiple elements. Linear search is less used today because it is slower than binary search and hashing.
- 1: Traverse the array
- 2: Match the key element with array element
- 3: If key element is found, return the index position of the array element
- 4: If key element is not found, return -1

SEARCHING - BINARY SEARCH

- Binary search is used to search a key element from multiple elements present in **sorted order**. Binary search is faster than linear search.
- mid = l + (r - l) / 2;
- Compare key element with the middle element.
- If key matches with the middle element, we return the mid index.
- Else If key is greater than the mid element, then key can only lie in the right half subarray after the mid element. So we recur for the right half.
- Else (key is smaller) recur for the left half.

SEARCHING - TERNARY SEARCH

- Ternary is similar to binary search where we divide the array into two parts but in this algorithm, we divide the given array into three parts and determine which has the key (searched element).
- mid1 = l + (r-l)/3
- mid2 = r – (r-l)/3
- First, we compare the key with the element at mid1. If found equal, we return mid1.
- If not, then we compare the key with the element at mid2. If found equal, we return mid2.
- If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.
- If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.
- If not, then we recur to the second (middle) part.

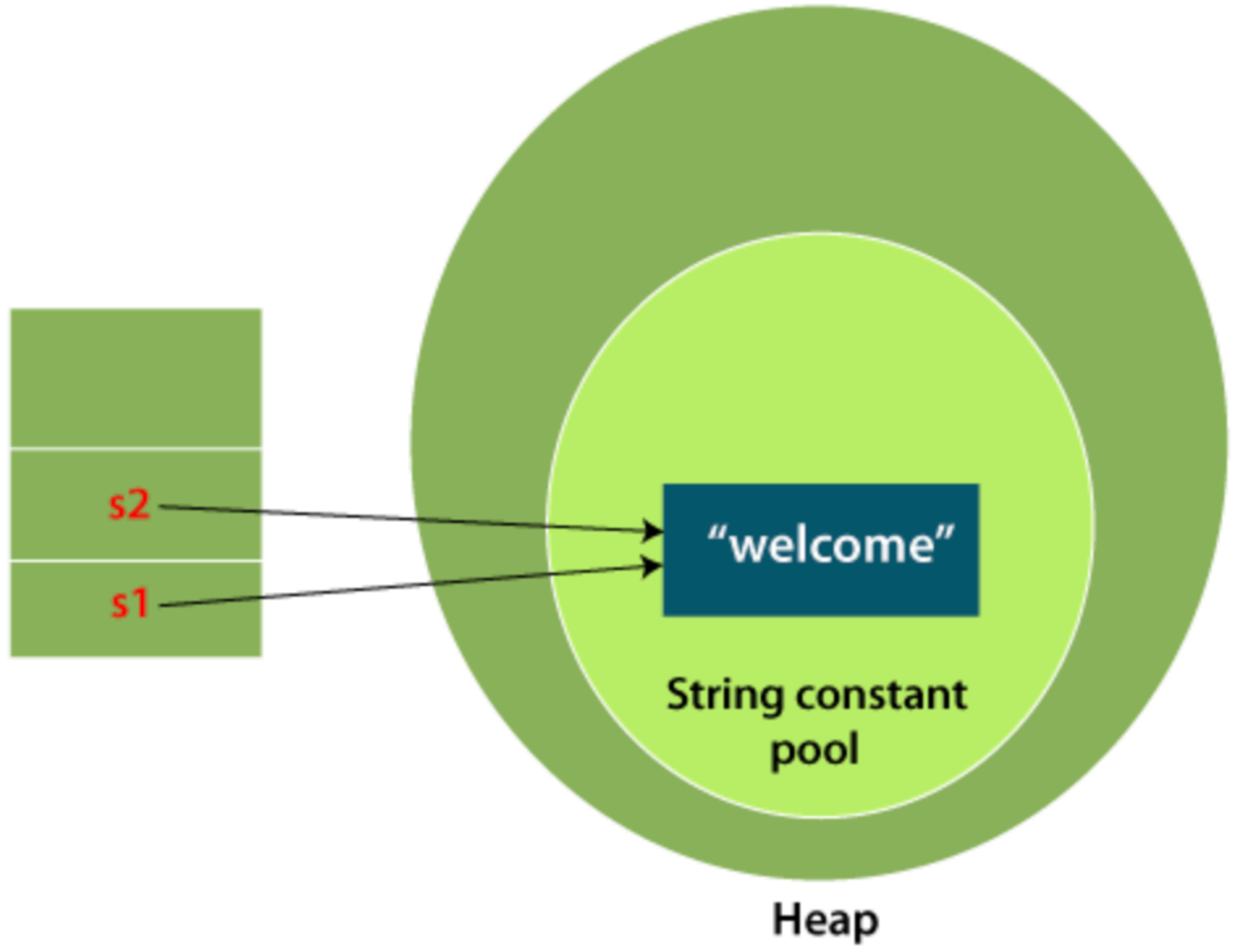
STRINGS

- String is basically an object that represents sequence of char values. An array of characters works same as Java string.
- Int arr[]={1,2,3,4};
- String arr[]={“ayoob”,”tarun”,”prajakta”,”i love my india”};
- **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};   String s=**new** String(ch);   - same as
- String s="javatpoint";
- **Java String** class provides a lot of methods to perform operations on strings such as concat(), equals(), length(), etc.

STRINGS

- The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created.
- For mutable strings, you can use StringBuilder classes.
- There are two ways to create String object:

1. By string literal, which is memory efficient : String s="welcome"; Each time you create a string literal, the JVM checks the "string constant pool" first which is present in Heap. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
2. By new keyword : String s=**new** String(“Welcome"); In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).


s2
s1
“welcome”
String constant pool
Heap

ARRAY OF STRINGS

```
1. String[] strAr1=new String[] {"Ani", "Sam", "Joe"}; //inline initialization
2. String[] strAr2 = {"Ani", "Sam", " Joe"};
3. String[] strAr3= new String[3]; //Initialization after declaration with specific size
   strAr3[0]= "Ani";
   strAr3[1]= "Sam";
   strAr3[2]= "Joe";
```

```
String[] strAr = {"Ani", "Sam", "Joe"};
for (int i=0; i<StrAr.length; i++)
{
System.out.println(strAr[i]);
}
for ( String str: strAr)
{
Sytem.out.println(str);
}
```

SORTING

- Software Engineer deals with real life problems to make our life easy.
- The problem here is to arrange the elements of a list in a certain order.
- Sorting is one of the important categories of algorithms, sometimes significantly reduces the complexity of a problem.
- It can be used to reduce the search complexity.

SORTING - BUBBLE SORT

- In each pass , maximum goes to last.
- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- **void** bubbleSort(**int** arr[])

```
{
    int n = arr.length;
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
            {
                // swap arr[j+1] and arr[j]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
```

SORTING - BUBBLE SORT - OPTIMIZED APPROACH

```
static void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    boolean swapped;
    for (i = 0; i < n - 1; i++)
    {
        swapped = false;
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
```

SORTING - INSERTION SORT

- Starting from second element, select each element and while going backward put that in right position.
- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- **void** sort(**int** arr[])

```
    {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;

 /* Move elements 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;
        }
    }
```

SORTING - MERGE SORT

- Divide and Conquer Approach to sort.
- Refer another ppt.

SORTING - QUICK SORT

- Divide and Conquer Approach to sort.
- It picks an element as pivot and partitions the given array around the picked pivot.
- There are many different versions of quickSort that pick pivot in different ways.
- The key process in quickSort is partition().
- Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

SORTING - QUICK SORT

- static void quickSort(int[] arr, int low, int high)

```
{
    if (low < high)
    {

        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

SORTING - QUICK SORT

- **static int** partition(**int**[] arr, **int** low, **int** high)

```
{
    int pivot = arr[high];

    // Index of smaller element and
    // indicates the right position
    // of pivot found so far
    int i = (low - 1);

    for(int j = low; j <= high - 1; j++)
    {

        // If current element is smaller
        // than the pivot
        if (arr[j] < pivot)
        {

            // Increment index of
            // smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}
```

- This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot

QUESTIONS

THANKS