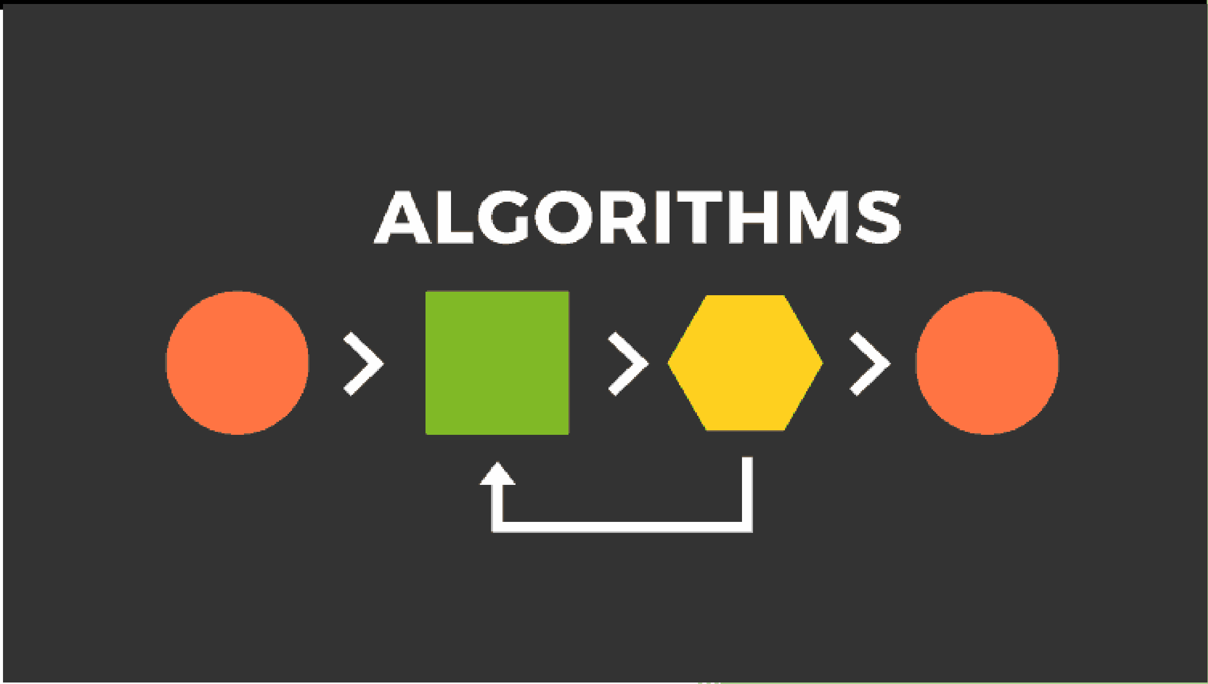
ALGORITHMS AND DATA STRUCTURES



Mariaa-K

Searching and Sorting Algorithms

Algorithm Performance Measurement

- To compare the performance of 2 search algorithms or 2 sorting algorithms, we compare based on : data movements (swaps / replacing one item in a list with another) and data comparisons (comparing one item in a list with either another item in the list or an item outside the list).
- It's sufficient to determine the approximate number of swaps and comparisons.
- It's often sufficient to know the order of magnitude of the number of swaps (i.e., is it in the 10's or 100's or 1,000,000's) and comparisons.
- For example, if one algorithm requires 100 swaps and another requires 50 swaps, we say that these two algorithms require the same number of swaps, since both of them are on the order of 100.
- We say that 50, 100, 500, 75, etc. are all on the order of 100, because all of them can be expressed as 100c, where c is some positive constant.
- For example, 50 = (100)(0.5) and 500 = (100)(5)

Search Algorithms

1. No assumption if list is initially sorted in order. (Linear Search)
2. Assuming list is initially sorted in order.(Binary Search)

Pseudo-code for linear search

```
for(each item in the list){
        compare the item you want with the current item
        if they match,
                save the index of the matching(current) item.
                break
}
return index of matching item or -1 if not found.
```

Implementation for linear search

- Without recursion:

```
public int linearSearch( int item, int[] list){
 int index=-1 //if index is still -1 at the end of this method, item wasn't found
 for(int i=0 ; i < list.length ; i++){
        if(list[i] == item){
                index=i;
                break;
        }
 }
 return index;
```

- With recursion:

```
private static int linearSearch ( int[] items, int target, int startIndex){
  if(startIndex == items.length)
      return -1;
  else if(items[startIndex] == target)
      return startIndex;
  else
      return linearSearch(items, target, startIndex+1);
}
```

Linear Search Performance

- When comparing performance, we look at three cases:
  - Best case: What is the fewest number of comparisons necessary to find an item? (The best case occurs when the search term is in the first slot in the array. The number of comparisons in this case is 1). O(1)
  - Worst case: What is the most number of comparisons necessary to find an item? (The worst case occurs when the search term is in the last slot in the array, or is not in the array. The number of comparisons in this case is equal to the size of the array). O(N)
  - Average case: On average, how many comparisons does it take to find an item in the list? (On average, the search term will be somewhere in the middle of the array. The number of comparisons in this case is approximately N/2). O(N)

Pseudo-code for Binary Search

```
set first=1, last=N, mid=N/2
while (item wanted not found and first < last) {
        compare wanted item to the item at mid
        if match
                save index
                 break
        else if wanted term is less than item at mid,
                set last = mid-1
        else
                 set first = mid+1
       set mid = (first+last)/2
}
return index of matching item, or -1 if not found
```

Implementation of binary search

- Without recursion

```
public int binarySearch (int item, int[] list) {
 int index = -1; //if index is still -1 at the end of this method, item wasn't found
 int low = 0;
 int high = list.length-1;
 int mid;
 while (high >= low) {
     mid = (high + low)/2;
     if (item < list[mid]) // value is in lower half, if at all
         high = mid - 1;
     else if (item > list[mid])  // value is in upper half, if at all
         low = mid + 1;
     else { // found it! break out of the loop
         index = mid;
         break;
     }
  }
 return index;
}
```

- With recursion

```
private static int binarySearch (int[] items , Comparable<Integer>
                                        target, int first, int last){
 if (first > last)
     return –1;
 else {
     int middle = (first + last) / 2;
     int compResult = target.compareTo(items[middle]);
     if (compResult == 0)
         return middle;
     else if (compResult < 0)
         return binarySearch(items, target, first, middle – 1);
     else
         return binarySearch(items, target, middle + 1, last);
}
```

Binary Search Performance

- Best case: O(1)
- Worst case: Search term is not in the list, or the search term is one item away from the middle of the list, or the search term is the first or last item in the list. O(logN)
- Average case: search term is anywhere else in the list. O(logN)

- Sorting Algorithms

1. Selection Sort

Mode of action: We put the smallest item at the start of the list, then the next smallest item at the second position in the list, and so on until the list is in order.

Pseudo-code for Selection Sort

```
for i=0 to N-2 { //N-2 is included
        set smallest = i
                for j=i+1 to N-1 {
                        compare list[j] to list[i]
                        if list[j[< list[i]
                        smallest = j
                }
                swap list[i] and list[smallest]
}
```

Implementation for Selection Sort

```
public void selectionSort(int[] list) {
  int minIndex; // index of current minimum item in array

   for(int i=0;  i < list.length-1 ; i++){
        minIndex=i;
        for(int j=i+1; j < list.length ; j++){
                if(list[j]<list[minIndex]){ // new minimum found, update
                                                minIndex
                        minIndex=j;
                }
        }
        swap(list,i,minIndex);
    }
}
```

Swapping:

```
public void swap(int[] list, int index1, int index2) {
  int temp = list[index1];
  list[index1] = list[index2];
  list[index2] = temp;
}
```

Selection Sort Performance

- Best case: List is already sorted. So number of swaps is 0. O(N^2)
- Worst case: The first item in the list is the largest, and the rest of the list is in order. In this case, we perform one swap on each pass through the algorithm, so the number of swaps is N. O(N^2)
- Average case: O(N^2) comparisons and N/2 swaps. O(N^2)

2. Bubble Sort

Mode of action: Starting from the beginning of the list, every adjacent pair is compared and their positions are swapped if they are not in the right order. After each iteration, one less element(the last one) is needed to be compared until there are no more elements left to be compared.

Pseudo-code for Bubble Sort

```
for i=0 to N-2{
        for j=0 to N-2-i{
                compare array[j] and array[j+1]
                if (array[j]>array[j+1])
                        swap array[j] and array[j+1]
```

Implementation of Bubble Sort

```
public static void bubbleSort(int[] array){
 for(int i=0; i < array.length-1 ; i++){
        for(int j=0 ; j < array.length-i-1 ; j++){
                if(array[j]>array[j+1])
                        swap(array, j , j+1)
        }
 }
```

Bubble Sort Performance

- Best case: List is already sorted. So number of swaps is 0. N-1 comparisons O(N^2)
- Worst case: List is in reverse order. O(N^2) swaps. O(N^2)
- Average case: Requires O(N^2) comparisons and O(N^2)swaps. O(N^2)

3. Insertion Sort

Mode of action: Going over the list, and when encountering a "not right order" of two elements, moving the greater element into place of the smaller and then inserting back the smaller infront of the greater.

Pseudo-code for Insertion Sort

```
for i=1 to N-1{
   save value of array[i] in a temporary variable.
   j=i-1;
   while(j>=0 element at j is greater than the element at i){
                move element at j into place of element at i
                j--;
   }
   Insert back the element at i (stored in temp) in front of the element
at j
```

Implementation for Insertion Sort

```
public static void insertionSort(int[] array){
  for(int i=1 ; i < array.length ; i++){
     int temp=array[i]; //saving value of smaller item to be able to add it
                                    back
     int j=i-1;
     while(j>=0 && array[j] > temp){
           array[j+1]=array[j]; //moving larger item in place of smaller
           j--;
     }
     array[j+1]=temp; //inserting back the smaller item
   }
}
```

Insertion Sort Performance

- Best case: List is already sorted. O(N)
- Worst case: List is in reverse order. O(N^2) swaps. O(N^2)
- Average case: Requires O(N^2) comparisons and O(N^2)swaps. O(N^2)

4. Quick sort

Mode of action: Specify one element in the list as a “pivot” point. Then, go through all of the elements in the list, swapping items that are on the “wrong” side of the pivot. In other words, swap items that are smaller than the pivot but on the right side of the pivot with items that are larger than the pivot but on the left side of the pivot. Once you’ve done all possible swaps, move the pivot to wherever it belongs in the list. Now we can ignore the pivot, since it’s in position, and repeat the process for the two halves of the list (on each side of the pivot). We repeat this until all of the items in the list have been sorted.

Quick Sort

2 Methods

Partition Method
Assigns the pivot.
Does the necessary swapping.

QuickSort Method
Recursion of partition

How quicksorting actually works:

1- Pivot is specified (usually always first or always last element in array)
2- A reference is made to the lowest index (first element) and the highest index(last element excluding the pivot)
3- Swap the items that are smaller than pivot but on it's right side with items that are bigger than pivot but on its left side.
   (If the element to the left of the pivot is less than it, it's correct, so increment. Similarly, if the element on the right of the pivot is greater than it, it's correct, so decrement (increment to the left). If both left is greater than pivot(wrong) and the right is less than the pivot(wrong), swap them together). Repeat till array is over.
4- Once done, move pivot to appropriate place.
5- Now we can ignore pivot because it's in the correct position. Repeat the whole process for the two halves (partitions) of the array. (done with the recursive calling of quicksort method).
   In conclusion: The method partition will pick the last element in the array passed as parameter as pivot and will do the appropriate swapping. It will also place the pivot in the right place. The method quicksort will recursively repeat the partition method on smaller chunks of the array until it is completely sorted.

Implementation of Quick Sort

```
public static void quickSort(int[] array , int low, int high){
    if(low<high){
            int pi=partition(array, low, high);
            quickSort(array, low, pi-1); //( ignore pivot now) Before pi
            quickSort(array, pi+1, high);// After pi
    }
}
public static int partition(int [] array, int low, int high){
    int pivot = array[high];
    int i=low-1;

    for(int j=low ; j <= high-1 ; j++){  //looping over the array except the
pivot
            if(array[j] <= pivot){
                    i++;
                    swap(array, i, j);
            }
    }
    swap (array, i+1 , high); //swapping element at  i+1 with pivot
    return (i+1); //this is index of where pivot is now
}
```

Quick Sort Performance

• Best case: the partitions are evenly balanced as possible: their sizes either are equal or are within 1 of each other. O(NlogN)

• Worst case: List is arranged in descending or ascending order the pivot is always the largest or smallest item on each pass through the list. In this case, we do not split the list in half or nearly in half, so we do N comparisons over N passes, which means the worst case is closer to O(N^2). For the same reasons, the number of swaps can be as high as O(N^2)

•Average case: the pivot splits the list in half or nearly in half on each pass. Each pass through the algorithm requires N comparisons. The number of passes through the algorithm is approximately log2N, and thus the number of comparisons is O(NlogN)

5. Merge Sort

Mode of action: Merge sort starts by dividing the list to be sorted in half. Then, it divides each of these halves in half. The algorithm repeats until all of these “sublists” have exactly one element in them. At that point, each sublist is sorted. In the next phase of the algorithm, the sublists are gradually merged back together (hence the name), until we get our original list back — sorted, of course.

Merge Sort

2 Methods

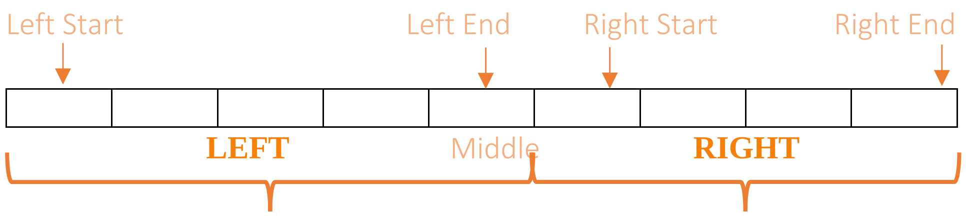
MergeHalves Method
Merges the 2 subarrays with the correct order.

MergeSort Method
Recursively calls itself on the 2 subarrays.

How mergesort actually works:

1- the array is divided into 2 subarrays until each subarray consists of 1 element only. (done by mergesort method)

2- the elements are combined (merged) together in the correct order (done by mergehalves method)



Implementation for Merge Sort

```
public static void mergeSort(int[] array,int leftStart, int rightEnd){
    if(leftStart >= rightEnd) // array is traversed completely, so it's over.
      return;
    int middle=(leftStart + rightEnd)/2;
    mergeSort(array, leftStart, middle);
    mergeSort(array, middle +1 , rightEnd);
    mergeHalves(array, leftStart, rightEnd);
  }
  public static void mergeHalves (int[] array , int leftStart, int rightEnd){
    int[] temp=new int[array.length];
    int leftEnd = (leftStart + rightEnd)/2;
    int rightStart = leftEnd + 1;
    int left=leftStart;
    int right=rightStart;
    int index=leftStart;

    while(left <= leftEnd && right <= rightEnd){
      if((array[left] <= array[right])){
        temp[index] = array[left];
        left++;
      } else {
        temp[index] = array[right];
        right++;
      }
      index++;
    }
    while(left <= leftEnd){ // for the remaining elements in the left
subarray
      temp[index]=array[left];
      index++;
      left++;
     }
    while(right <= rightEnd){ // for the remaining elements in the right
array
      temp[index]=array[right];
      index++;
      right++;
    }
    for(int i=leftStart ; i <= rightEnd ; i++){ //elements from temp to
array
      array[i]=temp[i];
    }
}
```

Merge Sort Performance

- Best case: List is already sorted. Number of comparisons per pass is (last+first)/2.O(NlogN)
- Worst case: Suppose the array in final step after sorting is {0,1,2,3,4,5,6,7} For worst case the array before this step must be {0,2,4,6,1,3,5,7} because here left subarray={0,2,4,6} and right subarray={1,3,5,7} will result in maximum comparisons (*Storing alternate elements in left and right subarray*) O(NlogN)
- Average case: O(NlogN)

6. Counting Sort

Mode of action: sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

How counting sort actually works

1- Get the minimum and maximum numbers in the array.

2- Range=max-min+1; this is the range of numbers in the array, starting from the minimum to the maximum

3- Create an array "count" that has the size of "range".

Create an array "output" that has the size of the input array.

4- Fill the count array according to the occurrences of each element in the array.

Input Arr

3	10	2	6	5

Min=2; Max= 10; Range=10-2+1=9

Index

2	3	4	5	6	7	8	9	10

Accordingly, the count array is:

1	1	0	1	1	0	0	0	1

Done by the first for loop in the code

5-Update the values of the count array, such that count[i]=count[i-1]

1	2	2	3	4	4	4	4	5

Done by the second for loop in the code

6- According to the "Index" and the latest count array, sort the array and fill the output array.

For instance, first element in input array=3.

go to the index “array” and find 3, then go to the new count array and find the same position as 3.(it is the first “2” in this case”) fill the output array at position 2 with 3 and decrement the count so that no 2 elements will be in the same position.(in case the array contains more than 1 occurrence of the same number)

Position	1	2	3	4	5
Output Array	2	3	5	6	10

Done by the third for loop in the code

Implementation of Counting Sort

```
public static void countingSort(int[] arr) {
    int max = getMaxValue(arr);
    int min = getMinValue(arr);
    int range = max - min + 1;
    int count[] = new int[range];
    int output[] = new int[arr.length];
    for (int i = 0; i < arr.length; i++) {
        count[arr[i] - min]++;
    }
    for (int i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }
    for (int i = arr.length - 1; i >= 0; i--) {
        output[count[arr[i] - min] - 1] = arr[i];
        count[arr[i] - min]--;
    }
    for (int i = 0; i < arr.length; i++) {
        arr[i] = output[i];
    }
}
```

Counting Sort Performance

The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most $k + 1$ times and therefore take $O(k)$ time. The other two for loops, and the initialization of the output array, each take $O(n)$ time. Therefore, the time for the whole algorithm is the sum of the times for these steps, $O(n + k)$.
Since we use array of size k+1 and n, the total space usage is also O(n+k)

- Best case: O(n+k)
- Worst case: O(n+k)
- Average case: O(n+k)

7. Bucket Sort

Mode of action: Create buckets of specific ranges(0-9,10-19…). Then, add corresponding elements to each bucket. Then sort content of each bucket using any sorting algorithm. Finally, transfer all elements to the array.

Implementation of Bucket Sort

```
private static void sortBuckets(int[] array){
   int bucketRange = 10;// range of each bucket: ex. 0-9, 10-19...
   int max = getMaxValue(array);
   int bucketCount = max/bucketRange + 1;
  //creating a list of buckets and each bucket has a list of elements, so
we create those lists.
  List<Integer>[] listBuckets = new List[bucketCount];
  for (int i = 0; i < bucketCount; i++){
          listBuckets[i] = new ArrayList<>();
  }
 //passing over the array and adding each element to the appropriate
  bucket
 for (int i = 0; i < array.length; i++){
      int element = array[i];
           int bucket = element / bucketRange; //to know which
                                                bucket to  add to
          listBuckets[bucket].add(element);
 }
 //sort each bucket using any sorting algorithm
 for (int i = 0; i < listBuckets.length; i++){
           Collections.sort(listBuckets[i]);
 }
 int index = 0;
 //first loop over the buckets and get the current bucket, then loop
over its elements and get the elements of this bucket and add them
to the array
 for (int i = 0; i < listBuckets.length; i++){
//   for every bucket empty it in array
          for (int j = 0; j < listBuckets[i].size(); j++){
               array[index] = listBuckets[i].get(j);
               index++;
        }
 }
}
```

Bucket Sort Performance

- Best case: O(n+k)
- Worst case: O(N^2)
- Average case: O(n+k)

When Bucket Sort is fast

Bucket sort's best case occurs when the data being sorted can be distributed between the buckets perfectly. If the values are sparsely allocated over the possible value range, a larger bucket size is better since the buckets will likely be more evenly distributed. An example of this is `[2303, 33, 1044]`, if buckets can only contain 5 different values then for this example 461 buckets would need to be initialized. A bucket size of 200-1000 would be much more reasonable.

The inverse of this is also true; a densely allocated array like `[103, 99, 119, 112, 111]` performs best when buckets are as small as possible.

Bucket sort is an ideal algorithm choice when:

- additional O(n + k) memory usage is not an issue
- Elements are expected to be fairly evenly distributed

When Bucket Sort is slow

Bucket sort performs at its worst, O(n^2), when all elements at allocated to the same bucket. Since individual buckets are sorted using another algorithm, if only a single bucket needs to be sorted, bucket sort will take on the complexity of the inner sorting algorithm.

This depends on the individual implementation though and can be mitigated. For example a bucket sort algorithm could be made to work with large bucket sizes by using insertion sort on small buckets (due to its low overhead), and merge sort or quicksort on larger buckets.

Note that the above implementation only works for positive numbers. Below is a manipulation of the algorithm that works on negative numbers as well

Implementation of Bubble Sort for negative numbers as well

```
public class BucketSort {

    public void sort(int[] array){
        List<Integer> negativeList = new ArrayList<>();
        List<Integer> positiveList = new ArrayList<>();

        for (int i = 0; i < array.length; i++){
//          if negative add to negative list else positive list
            if (array[i] < 0){
                negativeList.add(array[i]);
            }else{
                positiveList.add(array[i]);
            }
        }
//        array for each list
        int[] negative = new int[negativeList.size()];
        int[] positive = new int[positiveList.size()];

//        convert negative to positive values
        for (int i = 0; i < negative.length; i++)
            negative[i] = negativeList.get(i) * -1;
        for (int i = 0; i < positive.length; i++)
            positive[i] = positiveList.get(i);

//        sort negative bucket
        sortBuckets(negative);
//        sort positive bucket
        sortBuckets(positive);

        int index = 0;
//        loop in reverse over negative positive and fill them in array, changing sign
        for (int i = negative.length - 1; i >= 0; i--)
            array[index++] = negative[i] * -1;
//        loop in order over positive list
        for (int i = 0; i < positive.length; i++)
            array[index++] = positive[i];
    }
    private void sortBuckets(int[] array){
        if(array.length == 0)
                return;
        int bucketSize = 20;
        int max = Utils.findMax(array);//static method to find max or create your own
        int size = (int)Math.ceil(max/bucketSize) + 1;
        List<Integer>[] listBuckets = new List[size];
        for (int i = 0; i < size; i++){
            listBuckets[i] = new ArrayList<>();
        }
        for (int i = 0; i < array.length; i++){
            int element = array[i];
            int bucket = element / bucketSize;
            listBuckets[bucket].add(element);
        }

        for (int i = 0; i < listBuckets.length; i++){
            Collections.sort(listBuckets[i]);//or any other sorting algo
           //new QuickSort().sort(listBuckets[i]); works too
        }

        int index = 0;
//        loop over buckets
        for (int i = 0; i < listBuckets.length; i++){
//            empty bucket in array
            for (int j = 0; j < listBuckets[i].size(); j++){
                array[index] = listBuckets[i].get(j);
                index++;
            }
        }
    }
```

9. Radix Sort

Mode of action: sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

Implementation of Radix Sort

```
public void sort(int[] array) {
   int max = Utils.findMax(array);
    int exponent = 1;
    while(max > 0){
             countSort(array, exponent);
            max /= exponent;
           exponent *= 10;
   }
}
 private void countSort(int array[], int exponent) {
   int output[] = new int[array.length];
// store occurrence of each digit, at most 10 digits 0-9, so size 10
 int count[] = new int[10];
 int[] digitArray = new int[array.length];

//store current digits in digitArray instead of performing lengthy
operation every single time
for (int i = 0; i < array.length; i++){
           digitArray[i] = (array[i] / exponent) % 10;
 }
// get current digit from array by dividing by exponent and modulo 10
   for (int i = 0; i < array.length; i++)
           count[digitArray[i]]++;
              for (int i = 1; i < 10; i++)
                  count[i] += count[i - 1];

   for (int i = array.length - 1; i >= 0; i--) {
                  int countIndex = digitArray[i];
                  output[count[countIndex] - 1] = array[i];
                  count[countIndex]--;
  }

   for (int i = 0; i < array.length; i++)
                  array[i] = output[i];
    }
}
```

Radix Sort Performance

Radix sort complexity is $O(kn)$ for n keys which are integers of word size k.

• Best case: O(kn)
• Worst case: O(kn)
•Average case: O(kn)

Implementation of Radix Sort for negative numbers too

```
public class RadixSort {

    public void sort(int[] array) {

        List<Integer> negativeList = new ArrayList<>();
        List<Integer> positiveList = new ArrayList<>();

        for (int i = 0; i < array.length; i++){
//          if negative add to negative list else positive list
            if (array[i] < 0){
                negativeList.add(array[i]);
            }else{
                positiveList.add(array[i]);
            }
        }
//        array for each list
        int[] negative = new int[negativeList.size()];
        int[] positive = new int[positiveList.size()];

//        convert negative to positive values
        for (int i = 0; i < negative.length; i++)
            negative[i] = negativeList.get(i) * -1;
        for (int i = 0; i < positive.length; i++)
            positive[i] = positiveList.get(i);

        radixSort(negative);
        radixSort(positive);

        int index = 0;
        for (int i = negative.length - 1; i >= 0; i--){
            array[index] = negative[i] * -1;
            index++;
        }
        for (int i = 0; i < positive.length; i++){
            array[index] = positive[i];
            index++;
        }
    }

    private void radixSort(int[] array){
        int max = Utils.findMax(array);
        int exponent = 1;
        while(max > 0){
            countSort(array, exponent);
            max /= exponent;
            exponent *= 10;
        }
    }

    private void countSort(int array[], int exponent) {
        int output[] = new int[array.length];
//        store occurrence of each digit, at most 10 digits 0-9, so size 10
        int count[] = new int[10];
        int[] digitArray = new int[array.length];

//        store current digits in digitArray instead of performing lengthy operation every single time
        for (int i = 0; i < array.length; i++){
            digitArray[i] = (array[i] / exponent) % 10;
        }

//        get current digit from array by dividing by exponent and modulo 10
        for (int i = 0; i < array.length; i++)
            count[digitArray[i]]++;

        for (int i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for (int i = array.length - 1; i >= 0; i--) {
            int countIndex = digitArray[i];
            output[count[countIndex] - 1] = array[i];
            count[countIndex]--;
        }

        for (int i = 0; i < array.length; i++)
            array[i] = output[i];
    }
}
```

8. Heap Sort using Max

Mode of action: Create a binary tree representing the array. Then, start comparing the child nodes to its corresponding parent. Make sure that all parent nodes are >= its children. If not, swap them. Repeat this process and the array is now sorted.

How heap sort using max actually works:
1-Build heap
2-Build max heap
3-Delete root node
4-Put the last node of the heap into position
5-Repeat from step 2 till all nodes are covered

Implementation of Heap Sort using max/using min

```
class MaxHeap{
   int heapsize;

   public MaxHeap(int[] array){
      heapsize = array.length;
   }

  public void HeapSort(int[] array){
      buildMaxHeap(array); //max heap Is built so we need to delete the root
            node and put the last node in the root node's position (step3-step4)
      for(int i=array.length-1; i >= 0;i-- ){

         int tmp = array[0];
         array[0] = array[i];
         array[i] = tmp;
         heapsize--;

         maxHeapify(array, 0); //step5(repeating from step2 till all nodes are
                                covered)
      }
   }
   public void buildMaxHeap(int[] array){ //step 1
      for(int i=(array.length/2) - 1; i >= 0; i--){ //only half the array because half are
actually parent nodes, plus, we have left=2*i+1, if we loop over all we would
have out of bounds
         maxHeapify(array, i); //after building the heap, we need to create the max
      }                         heap (done by maxHeapify)
    }
```

```
//step 2
  public void maxHeapify(int[] array,int i){
    int left = (2*i)+1;
    int right = left + 1;
    int max;                                        int min

    if(left < heapsize){ //if left child exists
      if(array[left] > array[i]){ //child>parent     if(array[left] < array[i])
        max = left;
      }else
        max = i; // if child is not less than parent then for now, it's the maximum

    }else
      max = i;

    if(right < heapsize){ //if right child exists
      if(array[right] > array[max]){                 if(array[right] < array[min])
        max = right;
      }
    }
    if(max != i){ //if parent is not the max, so it has a child bigger than it and we need to swap
      int tmp  = array[max];
      array[max] = array[i];
      array[i] = tmp;
      maxHeapify(array, max);
    }
  }
}
```

<u>Heap Sort Performance</u>

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

The height of a complete binary tree containing n elements is log(n)
To fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.

In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of log(n) comparisons and swaps.
During the build_max_heap stage, we do that for n/2 elements so the worst case complexity of the build_heap step is n/2*log(n) ~ nlogn.

During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes logn worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this n times, the heap_sort step is also nlogn.
Also since the build_max_heap and heap_sort steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of nlogn.

In short, you can build your heap in O(n). Then you pop elements off, one at a time, each taking O(log n) time. This takes O(n log n) time total.

Graphs

1. A graph is a non-linear data structure.
2. A graph G is an ordered pair of of a set V of vertices and a set E of edges
   G = (V , E) *The order of a graph is usually the number elements of V
3. A subgraph of G is a graph G' = (V', E') where V' is a subset of V and E' is a subset of E
4. A proper subgraph of G is a subgraph G' of G such that: G' $\neq$ G and G' $\neq \varphi$
5. An induced subgraph of G is a subgraph G'=(V',E') such that: If e belongs to E joins two vertices of V', then e belongs to E'

Vertices and Edges

- 2 types of edges: a) directed and b)undirected

a) Directed edges (digraph): connection is one way

A symmetric digraph is a directed graph in which every directed edge has a reverse edge: (If there is an edge from a to b then there is also an edge from b to a)

b) Undirected edges: connection is two way

-A graph is called a self loop or a self edge if it contains only one vertex.(if both endpoints are the same(Can be directed or undirected).

- An edge is called a multiedge or parallel edge if it occurs more than once in a graph(can be directed or undirected).
- A graph is called simple graph if it doesn't contain self loops or multiedges.
- Two vertices a and b are adjacent if there is an edge connecting them.
- Vertices u and v are neighbors if they are adjacent. So we say v is a neighbor of u and u is a neighbor of v.
- Edges connecting a vertex v to its neighbors are said to be incident on v.
- Given a number n of vertices (no single edge or multiedge), the number of edges is:

a) For directed: $0 <= |E| <= n(n-1)$

b) For undirected: $0 <= |E| <= n(n-1)/2$

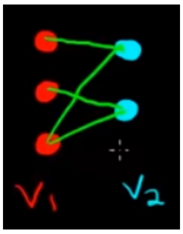
- A vertex is reachable from another vertex if there is a path between them
- Degree is the number of edges connected to a vertex
- The minimum degree of a vertex in a graph G is called the minimum degree of G, denoted $\delta(G)$.
- The maximum degree of G is the maximum among all the vertex degrees in G. It is denoted $\Delta(G)$.
- The neighborhood, NG(v), of a vertex v in G is the set of vertices that are adjacent to it. Denoted by N(v)
- The vertex connectivity of a graph G is the minimum number of vertices whose removal disconnects G
- The maximal connected subgraphs of a graph are called the connected components of the graph
- Edge connectivity is defined similarly

Dense and Scarce graphs

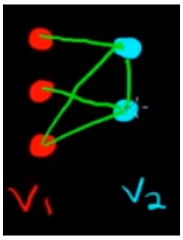
- A graph is called Dense if it has too many edges(stored in adjacency matrix)
- A graph is called Scarce if it has too few edges (stored in adjacency list)

Families of Graphs

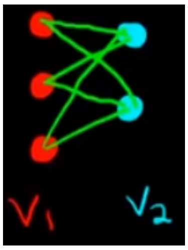
- A complete graph is an undirected graph such that any two vertices are adjacent(edge between every pair of vertices). It's denoted by Kn having n vertices.
- The number of edges in a complete graph on n vertices is n(n-1)/2 (by the Handshaking Lemma)
- An empty graph is a graph that has vertices but no edges
- A bipartite graph is a graph whose vertex set can be partitioned into 2 sets v1 and v2 such that every edge uv∈E has u ∈ v1 and u∈v2
- A complete bipartite graph is a graph which has every possible edge between the two sets of vertices denoted by Kn,m (n representing number of v1 and m number of v2)



Bipartite



Not bipartite



Complete bipartite(K3,2)

Paths and cycles

- A path is a sequence of vertices where each adjacent pair is connected by an edge. Denoted by Pn. Number of edges=number of vertices-1.
- A simple path is a path in which no vertices (and thus no edges) are repeated
- A closed walk (cycle) is a path that starts and ends at the same vertex and the length of the walk(number of edges) must be >0. Denoted by Cn. Number of edges=number of vertices
- An acyclic graph is a graph that has no cycles. A tree is an undirected acyclic graph

Handshaking Lemma

The sum of degrees in a graph is twice the number of edges

Connected graphs

- A strongly connected graph is a graph in which there's a path from one vertex to any other vertex. If it's an undirected graph, we call it connected. If it is a directed graph, we call it strongly connected.
- An undirected graph is 2-vertex-connected (or just biconnected) if the removal of any single vertex of G results in a connected graph (so the graph cannot be disconnected by removing a single vertex).
- An undirected graph G is t-vertex connected if removal of any collection of t-1 vertices of G results in a connected graph (so the graph cannot be disconnected by removing t-1 vertices).

Graph Representations:

➢ **Adjacency Matrix**

- An unweighted graph G on n vertices can be represented by a n×n matrix G where G[i,j] =
  - 1 if there is an edge from i to j
  - 0 otherwise.
- If the graph is a network(weighted), then G[i,j] is a numeric value equal to the weight of the edge from i to j.
- In a network, the value assigned to a weight where there is no edge is usually "infinity."
- For an undirected graph, this matrix would be symmetric because A[i][j]=A[j][i]. in fact, to see all the edges in the graph, we would only have to go through one of the halves.
- Good when a graph is dense (number of edges is close to v^2)
- Bad when a graph is scarce because we would have a lot of storage of 0's which is redundant.
- Time cost for most common operations is O(|V|) and not O(|E|) which can be as high as V^2
- Finding adjacent nodes: O(V)
- Finding if two nodes are connected: O(1)

```
public class AdjacencyMatrix {
    public static void main(String[] args) throws FileNotFoundException {
        int vertex;
        int[][]matrix;
        Scanner scan=new Scanner(new File("input.txt"));
        vertex=scan.nextInt();
        matrix=new int[vertex][vertex];
        while(scan.hasNext()){
            int source=scan.nextInt();
            int dest=scan.nextInt();
            matrix[source][dest]=1;
            matrix[dest][source]=1;
        }
        for(int i=0; i < vertex ; i++){
            for(int j=0 ; j < vertex ; j++){
                System.out.print(matrix[i][j]+" ");
            }
            System.out.println();
        }
        for (int i = 0; i < vertex; i++) {
            System.out.print("Vertex " + i + " is connected to:");
            for (int j = 0; j <vertex ; j++) {
                if(matrix[i][j]==1){
                    System.out.print(j + " ");
                }
            }
            System.out.println();
        }
    }
}
```

setUpGraph

Adjacency List

- A graph of order n is often represented as an n-item array where each item is a pointer to a linked list
- For a network, each item in the linked list is a pair, the vertex name and the weight as in G[3] > (1, 5) > (4, 3) > (6, 2) whereby the edge from 3 to 1 has a weight of 5, and the edge from 3 to 4 has a weight of 3, etc
- Good for sparce graphs (it doesn't save redundant 0's)
- Space Complexity: O(E)
- Finding adjacent nodes: O(V)
- Finding if two nodes are connected: O(V) (linear search) / O(logV)(binary search)

```
public class AdjacencyList {
    public static void main(String[] args) throws FileNotFoundException {
        int vertex;
        LinkedList<Integer> array[]; // array of linkedlists
        Scanner scan=new Scanner(new File("input.txt"));
        vertex=scan.nextInt();
        array=new LinkedList[vertex];
        for(int i=0 ; i < vertex ; i++){
            array[i]=new LinkedList<>();
        }
        while(scan.hasNext()){
            int source=scan.nextInt();
            int destination=scan.nextInt();
            array[source].add(destination); //add edge
            array[destination].add(source); //for undirected graph
        }
        for (int i = 0; i <vertex ; i++) {
            if(array[i].size()>0) {
                System.out.print("Vertex " + i + " is connected to: ");
                for (int j = 0; j < array[i].size(); j++) {
                    System.out.print(array[i].get(j) + " ");
                }
                System.out.println();
                }
            }
    }
}
```

Graph Traverals

There are two common traversals:

- depth-first traversal/search
- breadth-first traversal/search:

❖ Properties for both DFS and BFS:

- Can be applied for directed and undirected graphs
- In undirected graph, only difference in execution is that since every edge is undirected, it will be considered twice, once from each endpoint
- In a program, the vertex is picked arbitrarily, without regard to its place in the graph. Traversal is not completed unless all vertices are covered.
- In directed graphs, if we reach a dead end and we have vertices before it that are uncovered, we restart DFS/BFS at that vertex. We can restart as many times as needed to cover all vertices.
- In undirected graphs, restarts only happen if the graph is unconnected. If we start at a vertex from island A, we will cover vertices from that island and have to restart at a vertex from island B to traverse it. By this way, we can know how many islands there are (by the number of times we need to restart DFS/BFS on it)
- Traversal does 2 things repeatedly: 1) it visits a vertex 2) from the vertex it checks along every edge to see if that neighbor has been visited or not
- If the graph is directed and has e edges and n vertices, the neighbor checks=e and the total time is n+e
- If the graph is undirected and has e edges and n vertices, the neighbor checks=2e and the total time is n+2e.
- Time complexity for both directed and undirected is O(n+e)

Depth-First Search(DFS)

Mode of action: DFS starts at some vertex in the graph and proceeds across a sequence of edges and goes as deep as it can until it hits a dead end at which point it starts backtracking until it encounters a vertex off of which it can take another path down the graph.

- First visit all nodes reachable from node s (ie visit neighbors of s and their neighbors)
- Then visit all (unvisited) nodes that are neighbors of s
- Repeat until all nodes are visited

Implementation of DFS(iterative and recursive)

```
private static void iterativeDFS(int[][] matrix){
    boolean visited[] = new boolean[matrix.length];
    Stack<Integer> stack = new Stack<>();
    int vertex = 0;
    // Mark the current node as visited and enqueue it
    visited[vertex]=true;
    stack.push(node);

    while (stack.size() != 0) {
        vertex = stack.pop();
        System.out.print(vertex + " ");
        for (int i = matrix[vertex].length - 1; i >= 0; i--) {
            if (matrix[vertex][i] == 1) {
                if (!visited[i]) {
                    visited[i] = true;
                    stack.push(i);
                }
            }
        }
    }
}

private static void recursiveDFS(int[][] matrix, int vertex, boolean[] visited) {
    visited[vertex] = true;
    System.out.print(vertex + " ");
    for (int i = 0; i < matrix[vertex].length; i++){
        if (matrix[vertex][i] == 1){
            if (!visited[i]){
                recursiveDFS(matrix, i, visited);
            }
        }
    }
}
```

Breadth-First Search(BFS)

Mode of action: BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

- First move horizontally and visit all the nodes of the current layer
- Move to the next layer

Implementation of BFS(iterative and recursive)

```
private static void iterativeBFS(int[][] matrix){
    boolean visited[] = new boolean[graph.length];
    LinkedList<Integer> queue = new LinkedList<>();
    int vertex = 0;
    visited[vertex]=true;
    queue.add(vertex);

    while (queue.size() != 0) {
        vertex = queue.poll();
        System.out.print(vertex + " ");
        for (int i = 0; i < matrix[vertex].length; i++) {
            if (matrix[vertex][i] == 1) {
                if (!visited[i]) {
                    visited[i] = true;
                    queue.add(i);
                }
            }
        }
    }
}

private static void recursiveBFS(int[][] matrix, LinkedList<Integer> queue, boolean[] visited){
    if (queue.isEmpty())
        return;
    int vertex = queue.poll();
    visited[vertex] = true;
    System.out.print(vertex + " ");
    for (int i = 0; i < matrix[vertex].length; i++){
        if (matrix[vertex][i] == 1){
            if (!visited[i]){
                queue.add(i);
                visited[i] = true;
            }
        }
```

- Is the graph connected?

Mode of action: Traverse through the graph by using either DFS or BFS (implementation below is with BFS) and check if all the vertices present in the graph are visited.

```
private static boolean areAllVisited(boolean[] visited){
    for (int i = 0; i < visited.length; i++){
        if (!visited[i])
            return false;
    }
    return true;
}
private static boolean isConnected(int[][] graph){
    boolean[] visited = new boolean[graph.length];

    LinkedList<Integer> queue = new LinkedList<>();
    queue.add(0);

    while (queue.size() != 0){
        int node = queue.poll();
        for (int i = 0; i < graph[node].length; i++){
            if (graph[node][i] == 1){
                if (!visited[i]){
                    visited[i] = true;
                    queue.add(i);
                }
            }
        }
    }
    return areAllVisited(visited);
}
```

- <u>How many components(distinct graphs) does the graph have?</u>

  Mode of action: start count of components from 0. While not all the nodes are visited, increment components. Then, save the node that's not visited and apply BFS/DFS on it. Once done, return count of components.

```
    private static int connectedComponents(int[][] graph){
        boolean[] visited = new boolean[graph.length];
        int components = 0;
//        as long as not all nodes are visited
        while (!areAllVisited(visited)){
//            new component has been found
            components++;
            int node = 0;
//            select first unvisited node
            for (int i = 0; i< visited.length; i++){
                if (!visited[i]){
                    node = i;
                    break;
                }
            }
            LinkedList<Integer> queue = new LinkedList<>();
            queue.add(node);
            visited[node] = true;
            while (queue.size() != 0){
                node = queue.poll();
                for (int i = 0; i < graph[node].length; i++){
                    if (graph[node][i] == 1){
                        if (!visited[i]){
                            visited[i] = true;
                            queue.add(i);
                        }
                    }
                }
            }
        }
        return components;
    }
```

__What is the largest connected component (count of nodes of the component that has the most nodes)__

```
private static int connectedComponents(int[][] graph){
    boolean[] visited = new boolean[graph.length];
    int max=Integer.MIN_VALUE;
    int components = 0;

    while (!areAllVisited(visited)){
        int count=0;
        components++;
        int node = 0;

        for (int i = 0; i< visited.length; i++){
            if (!visited[i]){
                node = i;
                break;
            }
        }
        LinkedList<Integer> queue = new LinkedList<>();
        queue.add(node);
        visited[node] = true;
        while (queue.size() != 0){
            node = queue.poll();
            count++;
            for (int i = 0; i < graph[node].length; i++){
                if (graph[node][i] == 1){
                    if (!visited[i]){
                        visited[i] = true;
                        queue.add(i);
                    }
                }
            }
        }
        if(count > max)
            max=count;
    }
    return max;
```

Declare a variable

Initialize count to 0. This is the count of nodes in each component. Notice that for each component, count is restarted at 0

Once the node is polled from the queue, increment the count of nodes

At this point, we have the count of the nodes in the component, so we update the max value if it's greater than current max

- <u>Does the graph have a cycle?</u>

  A graph has a cycle if there's a path that leads to a **visited** vertex from a **non parent** one.

  Mode of action: As long as the node is not the parent node, is a neighbor of the parent node, we check if it's visited or if any of the neighbors have a cycle.

```
private static boolean hasCycle(int[][] graph, boolean[]visited, int node ,int parent){
    visited[node] = true;
    for (int i = 0; i < graph[node].length; i++){
        if (i != parent && graph[node][i] == 1){ //if not parent and is a neighbor
            if (visited[i] || hasCycle(graph, visited, i, node))
                return true;
        }
    }
    return false;
}
```

- <u>Is the graph a tree?</u>

  A graph is a tree if it is connected and does not have a cycle.

```
public static boolean isTree(int s){
        if(isConnected(s) && !hasCycle(s))
                return true;
        return false;
}
```

- <u>Is the graph strongly connected?</u>

  A directed graph is said to be strongly connected if every vertex is reachable from every other vertex.

  Mode of action: perform BFS or DFS starting from every vertex in the graph. If each BFS/DFS visits every other vertex in the graph, the graph is strongly connected

```
private static boolean isStronglyConnected(int[][]graph, int vertices){
    for(int i=0; i < vertices ; i++){//do for every vertex
        boolean visited[]=new boolean[vertices];//stores vertex is visited or not
        recursiveDFS(graph, i , visited);//start DFS from the first vertex
        for(int j=0; j < visited.length ; j++){ //if DFS traversal doesn't visit all
            if(!visited[j])                      vertices, then the graph is not strongly
                return false;                    connnected
        }
    }
    return true;
}
```

- <u>Is the graph Bipartite?</u>

Mode of action: start at node 0, coloring it 1(that is, it belongs to group 1). Then, start traversing the graph using BFS(DFS cannot be used here, because in bipartite, we need to check the neighbors!). Then, color the **unvisited** neighbors of this node with the opposite color(that is, they belong to group 2). If the neighbor is **visited**, and has the same color as the node, the graph is not bipartite.

<u>Color codes:</u>

- Color 0: indicates that the node is unvisited (by default, all array elements are 0. thus, if color remained unchanged (stayed 0) then it means the node is unvisited.
- Color 1: indicates that the node is of color 1.
- Color 2: indicates that the node is of color 2.

```
private static boolean isBipartite(int[][] graph){
    int[] colors = new int[graph.length];//color array of size number of nodes in graph
    colors[0] = 1;//start by marking the first node(node 0) in graph with color 1

    LinkedList<Integer> queue = new LinkedList<>();
    queue.add(0);
                                                         // traverse the graph using BFS
    while (queue.size() != 0){
        int node = queue.poll();
        for (int i = 0; i < graph[node].length; i++){
            if (graph[node][i] == 1 && colors[i] == 0){//If it is a neighbor and is not visited
                if (colors[node] == 1)   //If the node is colored 1, then its
                    colors[i] = 2;       neighbor must be colored 2
                else
                    colors[i] = 1;  //If the node is colored 2, then its neighbor must be colored 1
                queue.add(i);
            }else if (graph[node][i] == 1 && colors[i] == colors[node]){ //If it's a neighbor
                return false;                                              and has same
            }                                                              color as node,
        }                                                                  return false
    }

    return true;
}
```

Dijsktra's Algorithm (shortest path)

- Dijkstra algorithm is a greedy algorithm.
- It finds a shortest path tree for a weighted undirected graph.
- This means it finds a shortest paths between nodes in a graph, which may represent, for example, road networks
- For a given source node in the graph, the algorithm finds the shortest path between source node and every other node.
- This algorithm also used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.
- Dijkstra's algorithm is very similar to Prim's algorithm. In Prim's algorithm we create minimum spanning tree (MST) and in Dijkstra algorithm we create *shortest path tree (SPT)* from the given source
- Performance using Adjacency Matrix: O(V^2)

Steps:

1- Start with empty SPT[]. This will be used to keep track of visited vertices.
2- Assign a distance to all vertices (using distance[] array) and initialize all distances with infinity except the source vertex. This will be used to keep track of distance vertices from the source vertex.
Distance from source vertex to source vertex is 0
3-Repeat the below steps till all vertices are covered:

a) Pick a vertex U that's not in SPT[] and has min distance.
b) Add vertex U to SPT[]
c) Loop over all adjacent vertices V.
d) For adjacent vertex V, if V is not in SPT[] and distance[V] > distance[U] + edge u-v weight then update distance[V]= distance[u] + edge u-v weight

```
public class DijkstraAdjacencyMatrix {

    private static int vertices;
    private static int[][] matrix;

    public static void main(String[] args) throws FileNotFoundException {
        Scanner scan = new Scanner(new File("input.txt"));
        vertices = scan.nextInt();
        matrix = new int[vertices][vertices];
        while (scan.hasNext()) {
            int source = scan.nextInt();
            int destination = scan.nextInt();
            int weight = scan.nextInt();

            matrix[source][destination] = weight;
            matrix[destination][source] = weight;
        }
        dijkstra_GetMinDistances(0);
    }

    private static int getMinimumVertex(boolean[] mst, int[] distance) {
        int minKey = Integer.MAX_VALUE;
        int vertex = -1;
        for (int i = 0; i < vertices; i++) {
            if (mst[i] == false && minKey > distance[i]) {//step 3a// pick vertex that's not in spt
                minKey = distance[i];                      and has min distance
                vertex = i;
            }
        }
        return vertex;
    }

    private static void dijkstra_GetMinDistances(int sourceVertex) {
        boolean[] spt = new boolean[vertices]; //step 1//
        int[] distance = new int[vertices];
        int INFINITY = Integer.MAX_VALUE;

        for (int i = 0; i < vertices; i++) {
            distance[i] = INFINITY;                //Initialize all distances to infinity   //step 2//
        }
        distance[sourceVertex] = 0;//start from vertex 0
        for (int i = 0; i < vertices; i++) {
            int vertex_U = getMinimumVertex(spt, distance);//get vertex with minimum distance //step 3a//
            spt[vertex_U] = true;//include this vertex in spt //step 3b//
            for (int vertex_V = 0; vertex_V < vertices; vertex_V++) { //step 3c//
                if (matrix[vertex_U][vertex_V] > 0) { //check of edge between vertex_U and vertex_V
//check if vertex_V is already in
spt and if distance is not infinity
                    if (spt[vertex_V] == false && matrix[vertex_U][vertex_V] != INFINITY) {
                        int newKey = matrix[vertex_U][vertex_V] + distance[vertex_U];   //step 3d//
                        if (newKey < distance[vertex_V])
                            distance[vertex_V] = newKey;
                    }
                }
            }
        }
        printDijkstra(sourceVertex, distance);
    }
    private static void printDijkstra(int sourceVertex, int[] distance) {
        System.out.println("Dijkstra Algorithm: (Adjacency Matrix)");
        for (int i = 0; i < vertices; i++) {
            System.out.println("Source Vertex: " + sourceVertex + " to vertex " + +i +
                    " distance: " + distance[i]);
        }
    }
}
```

Prim's Algorithm

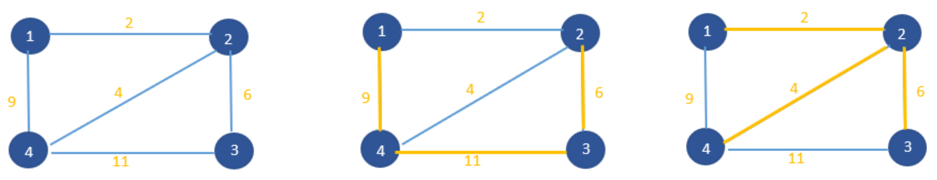
Definitions:

1- Spanning tree: given a connected and undirected graph, a spanning tree of that graph is a tree that connects all the vertices together.

2- Minimum spanning tree: the spanning tree of the graph whose sum of weights of edges is minimum.

Note: a graph can have more than 1 minimum spanning tree.

Connected undirected graph | Spanning Tree | Minimum Spanning tree



Cost=26 | Cost=12

- Prim's algorithm is a greedy algorithm.
- It finds a minimum spanning tree for a weighted undirected graph.
- This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.
- The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.
- Performance using Adjacency Matrix: O(V^2)

Steps:

1- Start with the empty spanning tree.

2- Maintain a set mst[] to keep track to vertices included in minimum spanning tree.

3- Assign a key value to all the vertices, (say distance []) and initialize all the keys with +∞ (Infinity) except the first vertex. (We will start with this vertex, for which key will be 0).

4- Key value in step 3 will be used in making decision that which next vertex and edge will be included in the mst[]. We will **pick the vertex which is not included in mst[] and has the minimum key**. So at the beginning the first vertex will be picked first.

5- Repeat the following steps until all vertices are processed

a- Pick the vertex ***u*** which is not in mst[] and has minimum key.

b- Add vertex ***u*** to mst[].

c- Loop over all the adjacent vertices of ***u***

d- For adjacent vertex ***v***, if ***v*** is not in mst[] and ***edge u-v weight*** is less than the key of vertex u, ***key[u]*** then update the key[u]= edge u-v weight

```java
public class Prims {
    private static int vertices;
    private static int[][] matrix;

    public static void main(String[] args) throws FileNotFoundException {
        Scanner scan = new Scanner(new File("input.txt"));
        vertices = scan.nextInt();
        matrix = new int[vertices][vertices];
        while (scan.hasNext()) {
            int source = scan.nextInt();
            int destination = scan.nextInt();
            int weight = scan.nextInt();

            matrix[source][destination] = weight;
            matrix[destination][source] = weight;
        }
        primMST();
    }

    private static int getMinimumVertex(boolean[] mst, int[] distance) {
        int minKey = Integer.MAX_VALUE;
        int vertex = -1;
        for (int i = 0; i < vertices; i++) {
            if (!mst[i] && minKey > distance[i]) {
                minKey = distance[i];
                vertex = i;
            }
        }
        return vertex;
    }
    private static void primMST(){
        boolean[] mst = new boolean[vertices];     //step 1-step2//
        int[] parent=new int[vertices];
        int[] weight=new int[vertices];
        int [] distance = new int[vertices];
        //Initialize all the keys to infinity
        for (int i = 0; i <vertices ; i++) {
            distance[i] = Integer.MAX_VALUE;                //step3-step4//
        }
        //start from the vertex 0
        distance[0] = 0;
        parent[0]=-1;

        //create MST
        for (int i = 0; i <vertices ; i++) {

            int vertex_U = getMinimumVertex(mst, distance);   //step 5a //
            mst[vertex_U] = true;    //step 5b//


            for (int vertex_V = 0; vertex_V <vertices ; vertex_V++) {

                if(matrix[vertex_U][vertex_V]>0){
                    //check if this vertex 'vertex_V' is already in mst. If not, check if distance needs update or not
                    if(!mst[vertex_V] && matrix[vertex_U][vertex_V] < distance[vertex_V]){    //step 5c //
                        //update distance parent and weight
                        distance[vertex_V] = matrix[vertex_U][vertex_V];
                        parent[vertex_V] = vertex_U;
                        weight[vertex_V] = distance[vertex_V];
                    }
                }
            }
        }
        printMST(parent, weight);
    }

      public static void printMST(int[] parent, int[] weight){
          int total_min_weight = 0;
          System.out.println("Minimum Spanning Tree: ");
          for (int i = 1; i <vertices ; i++) {
              System.out.println("Edge: " + i + " - " + parent[i] +
                      " key: " + weight[i]);
              total_min_weight += weight[i];
          }
          System.out.println("Total minimum key: " + total_min_weight);
      }

  }
```

- Kruskal's Algorithm

Mode of action:

1. Sort all edges in increasing order of weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. repeat step 2 until there are (V-1) edges in the spanning tree

Prim's Algorithm for finding MST	**Kruskal's Algorithm for finding MST**
Begins with Node	Begins with edge
Move from one node to another. *Prim's Algorithm will grow a solution from a random vertex by adding the next cheapest vertex to the existing tree.*	Select the next edge in increasing order *Kruskal's Algorithm will grow a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest*
Restricted on connected graph.	Works on both connected and disconnected graph
Faster for dense graphs	Faster for sparse graphs

Questions and Answers

1- Will both Prim's Algorithm and Kruskal's Algorithm always produce the same Minimum Spanning Tree (MST) for any given graph?

Answer: 2 cases:

Case 1: when all the edge weights are distinct, both the algorithms always produce the same MST having the same cost.

Case 2: when all the edge weights are not distinct, different MSTs could be produced by both the algorithms but the cost of the resulting MSTs from both the algorithms would always be same.

2- Which algorithm is preferred- Prim's Algorithm or Kruskal's Algorithm?

Answer:

- Kruskal's Algorithm is preferred when the graph is sparse i.e when there are less number of edges in the graph like $E = O(V)$ or when the edges are already sorted or can be sorted in linear time.

- Prim's Algorithm is preferred when the graph is dense i.e. when there are large number of edges in the graph like $E = O(V^2)$ because we do not have to pay much attention to the cycles by adding an edge as we primarily deal with the vertices in Prim's Algorithm.

```java
public class Kruskal {

    /**
     * A helper class to hold variables for an edge in a weighted graph. An edge is defined by a source, destination,
     * and weight. The class implements @Comparable interface in which edges are compared by the weight.
     * NOTE: We CAN use 3 separate arrays, but implementation would be very tedious, this is much easier especially
     * if we want to do some basic manipulation
     */
    private static class Edge implements Comparable<Edge>{
        int source;
        int destination;
        int weight;

        public Edge(int source, int destination, int weight) {
            this.source = source;
            this.destination = destination;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge edge) {
            if (weight < edge.weight)
                return -1;
            if (weight == edge.weight)
                return 0;
            return 1;
        }

        @Override
        public String toString() {
            return weight + " " + source + " " + destination;
        }
    }
    public static void main(String[] args) throws FileNotFoundException {
        List<Edge> listEdges = new ArrayList<>();
        Scanner scan = new Scanner(new File("graph.txt"));
        int vertices = scan.nextInt();
        int edges = scan.nextInt();
        for (int i = 0; i < edges; i++){
            int source = scan.nextInt();
            int destination = scan.nextInt();
            int weight = scan.nextInt();
            Edge e = new Edge(source, destination, weight);
            listEdges.add(e);
        }
        Collections.sort(listEdges); //sort edges in ascending order
        int[][] matrix = new int[vertices][vertices];
        int edgeIndex = 0;
        int edgeCount = 0;
        int totalWeight = 0;
        while (edgeCount < vertices - 1){//number of edges in a tree is number of vertices-
            boolean[] visited = new boolean[vertices];
            Edge e = listEdges.get(edgeIndex);
            matrix[e.source][e.destination] = 1;
            matrix[e.destination][e.source] = 1;
            if (hasCycle(matrix, visited, e.source, e.source)){ //check if the added edge would form a cycle
                matrix[e.source][e.destination] = 0;  //if a cycle was found, remove that edge from the matrix
                matrix[e.destination][e.source] = 0;
            }else{
                //otherwise, add this weight to the total weight and increment count of edges
                totalWeight += e.weight;
                edgeCount++;
            }
            edgeIndex++;//done with this edge. Need to go to the next edge and repeat process
        }
        System.out.println(totalWeight);
        printMatrix(matrix);
    }

    private static void printMatrix(int[][] matrix){
        for (int i = 0; i < matrix.length; i++){
            for (int j = 0; j < matrix.length; j++){
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }

    private static boolean hasCycle(int[][] graph, boolean[]visited, int node ,int parent){
        visited[node] = true;
        for (int i = 0; i < graph.length; i++){
            if (i != parent && graph[node][i] == 1){
                if (visited[i] || hasCycle(graph, visited, i, node))
                    return true;
            }
        }
        return false;
    }
}
```

Kruskal's Algorithm Time Complexity Analysis

Sorting ⟶ O(ElogE) ⟶ O(V^2logV^2) ⟶ O(2V^2logV) ⟶ O(V^2logV)
⟶ O(ElogV)

+ while loop and hasCycle method ⟶ O(V^2)

Since O(ElogV)>O(V^2), time complexity of the algorithm implemented above is O(ElogV).

However, if we used a linear sort(counting sort, bucket sort etc), O(V^2)>O(1), time complexity would be O(V^2).

- K-Coloring

  A graph is k-colorable if we can color it with k colors such that no two adjacent nodes share the same color.

  Mode of action:

  1- Color first vertex with first color

  2- Do the following for the remaining V-1 vertices: Consider the currently picker vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

K-Coloring

3 Methods

isColorable

Checks if the vertex can be colored ,i.e. if it was colored before, return false

basically loop over neighbors of this node and check if there's a node that has same color, return false(it's not valid) else return true

selectV

Select a vertex that is not colored

Coloring

Colors the graph

```
public class KColoring {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner scan = new Scanner(new File("input.txt"));
        int vertices = scan.nextInt();
        int [][] matrix = new int[vertices][vertices];
        while (scan.hasNext()) {
            int source = scan.nextInt();
            int destination = scan.nextInt();

            matrix[source][destination] = 1;
            matrix[destination][source] = 1;
        }

        int[] color = new int[vertices];
        int numColor = 2;
        System.out.println(Coloring(matrix, color, numColor));
    }
                                  //v is the current node. withColor is the color of this current node v
    private static boolean isColorable(int[][] matrix, int color[], int v, int withColor) {
        for (int i = 0; i < matrix.length; i++) {
            if(matrix[v][i]==1){
                if (color[i] == withColor)
                    return false;
            }
        }
        return true;
    }
                                  //the color array has length of nodes. So color[0]
    private static int selectV(int[] color) { represents color at node 0 etc
        for (int i = 0; i < color.length; i++) {
            if (color[i] == 0)
                return i;//if is not colored return its index
        }
        return -1;//return -1 if all vertices are covered
    }                                                       //numcolors is k, how many
                                                            colors I wanna color the
                                                            graph with.
    private static boolean Coloring(int[][] matrix, int[] color, int numColors) {
        int v = selectV(color);//First we need to select a vertex and color it
        if (v == -1) //if its -1 which means that my graph is colored
            return true;

        for (int i = 1; i <= numColors; i++) {//we start from 1 because 0 means not colored
            if (isColorable(matrix, color, v, i)) { //if i can color this vertex v with i, do it.
                color[v] = i;

                if (Coloring(matrix, color, numColors)) //keep coloring the graph. If all the graph was
                    return true;                         colored, return true.
                color[v] = 0; //if it didn't work,  need to backtrack,  need to try a new color so  set it to
            }                 0 and (with for loop i++),  give it a new color and repeat.
        }
        return false;//not all graph was colored so return false
    }
}
```

- <u>Vertex Cover</u>

A vertex cover of a graph is a subset of vertices which can cover every edge. An edge is covered if one of its endpoints is chosen.

<u>Steps:</u>

1- Find a vertex v with the maximum degree.

2- Add v to the solution and remove v and all its indecent edges from the graph.

If this max vertex covered all the graph, then cover is found.

If it doesn't, add its neighbors to the cover.

3- Repeat until all edges are covered.

Vertex Cover

6 Methods

<u>maxDegree</u>
takes degree array.
Returns the node/index that has max degree

<u>copyMatrix</u>
takes 2 matrices.
copies elements from one matrix to the other

<u>isEdgeless</u>
takes degree array
checks if all degrees > 0 if they are, this means that it has edges

<u>edgeDelete</u>
takes matrix, degree, source node, destination node
if src and dest are connected, disconnect them. When an edge is removed, degree decreases, so update degree by decrementing it.

<u>hasVC</u>
takes matrix, degree array and int k
checks if given a certain int k, we can find a vertex cover

<u>addToCover</u>
takes matrix, degree array and vertex
adds the vertex to cover
**when we add vertex to cover, we remove all of its edges*
Thus, this method checks if there's an edge between the nodes, if so, it deletes it by calling edgeDelete

```java
public class VertexCover {
    public static void main(String[] args) throws Exception {
        Scanner scan = new Scanner(new File("input.txt"));
        int k = 5; //size of vertex cover
        int nodes = scan.nextInt();
        int edges = scan.nextInt();
        int[][] matrix = new int[nodes][nodes];
        int degree[] = new int[nodes];
        for (int i = 0; i < edges; i++) {
            int source = scan.nextInt();
            int destination = scan.nextInt();

            degree[source]++;
            degree[destination]++;

            matrix[source][destination] = 1;
            matrix[destination][source] = 1;
        }

        if (hasVC(matrix, degree, k)) {
            System.out.println("YES:");
            for (int i = 0; i < degree.length; i++)
                if (degree[i] == -1)
                    System.out.print(i + " ");
            System.out.println();
        } else System.out.println("No");
    }

    public static int maxDegree(int[] degree) {
        int index = 0;
        for (int i = 1; i < degree.length; i++)
            if (degree[i] > degree[index])
                index = i;
        return index;
    }
    public static boolean edgeDelete(int[][] matrix, int[] degree, int source, int destination) {
        if (matrix[source][destination] == 1) {
            matrix[source][destination] = 0;
            matrix[destination][source] = 0;
            degree[source]--; //when edge is removed, degree decreases
            degree[destination]--;
            return true;
        }
        return false;
    }

    private static void copyMatrix(int[][] from, int[][] to) {
        if (from.length != to.length || from[0].length != to.length) //if size not equal stop
            return;
        for (int i = 0; i < from.length; i++)
            System.arraycopy(from[i], 0, to[i], 0, from.length);
    }

    public static boolean isEdgeless(int degree[]) {
        for (int i = 0; i < degree.length; i++) {
            if (degree[i] > 0) {
                return false;
            }
        }
        return true;
    }

    public static void addToCover(int[][] matrix, int[] degree, int v) {
        for (int i = 0; i < matrix.length; i++) {
            if (matrix[v][i] == 1) {
                edgeDelete(matrix, degree, v, i);
            }
        }
        degree[v] = -1; //to mark that this vertex belongs to the set of covers
    }
```

*Continuation

```
public static boolean hasVC(int[][] matrix, int[] degree, int k) {
    int[] tempdeg = new int[degree.length];
    int[][] tempMatrix = new int[matrix.length][matrix.length];
    copyMatrix(matrix, tempMatrix);

    int vertex = maxDegree(degree); //pick vertex with max degree
    if (degree[vertex] < 2) { //degree=0 or degree=1 (2 nodes 1 edge)
        while (!isEdgeless(degree)) {
            vertex = maxDegree(degree);
            addToCover(matrix, degree, vertex);
            k--;
        }
    }
    if (k < 0)
        return false;
    if (isEdgeless(degree))//I was able to selected k vertices and is edgeless
        return true;
    if (k == 0) //k=0 but it's not edgeless
        return false;

    System.arraycopy(degree, 0, tempdeg, 0, degree.length);
    addToCover(tempMatrix, tempdeg, vertex);
    if (hasVC(tempMatrix, tempdeg, k - 1)) {
        System.arraycopy(tempdeg, 0, degree, 0, degree.length);
        copyMatrix(tempMatrix, matrix);
        return true;
    }
    if (degree[vertex] > k) {
        return false;
    }
    for (int i = 0; i < matrix.length; i++) {//if it's not vc, i have to put its neighbors in the vc
        if (matrix[vertex][i] == 1) { //check if neighbor, add to cover
            addToCover(matrix, degree, i);
            k--; //vertex added so decrement it
        }
    }
    return hasVC(matrix, degree, k);
}
}
```

- <u>N queen problem</u>

  It's the problem of placing N queens on an NxN chessboard such that no 2 queens can attack eachother. Thus, no 2 queens should share the same row, column or diagonal.

N-Queen problem

2 Methods

<u>isValid</u>

checks if a queen can be in put this specific row &column.

8 for loops since the queen can move in 8 directions: Left, Right, Top, Down, Left-diagonal-up, Right-diagonal-up, Left-diagonal-down, Right -diagonal-down. Each for loop is for each direction check

<u>solve</u>

Checks if it's possible to place a queen by calling isValid method. If so, a queen is added.

```
public class NQueens{

	public static void main (String[] args) {
		int n=8;
		int A[][]=new int[n][n];
		System.out.println(solve(A,0,n));
		for(int i=0;i<n;i++){
			for(int j=0;j<n;j++){
				System.out.print(A[i][j]+" ");
			}
			System.out.println();
		}
	}

	public static boolean solve(int A[][],int col,int queens){
		if(queens==0)
			return true;

		for(int row=0;row<A.length;row++){//loop over the rows of the 2d-array
			if(isValid(A,row,col)){//check if it's possible to put a queen
				A[row][col]=1;//if possible, place queen

				if(solve(A,col+1,queens-1))//if all done
					return true;

				A[row][col]=0; //if queen couldn't be placed, backtrack
			}
		}

		return false;
	}

	public static boolean isValid(int A[][],int row,int col){
		for(int j=col;j<A.length;j++)//checks right cells of my current position
			if(A[row][j]==1)
				return false;

		for(int i=row;i<A.length;i++)//checks down cells of my current  position
			if(A[i][col]==1)
				return false;

		for(int j=col;j>=0;j--)//check left cells of my current  position
			if(A[row][j]==1)
				return false;

		for(int i=row;i>=0;i--)//checks top cells of my current  position
			if(A[i][col]==1)
				return false;

		for(int j=col,i=row;i<A.length && j<A.length;i++,j++)//checks diagonal direction right-down
			if(A[i][j]==1)
				return false;

		for(int j=col,i=row;i>=0 && j>=0;i--,j--)//checks diagonal side left-up
			if(A[i][j]==1)
				return false;

		for(int j=col,i=row;i<A.length && j>=0;i++,j--)//checks diagonal side right-up
			if(A[i][j]==1)
				return false;

		for(int j=col,i=row;i>=0 && j<A.length;i--,j++)//checks diagonal side left-down
			if(A[i][j]==1)
				return false;

		return true;
	}
}
```