HACK PACK COP 4516

Hammad Usmani
Michelle Edwards
Andrew Mendez

Each topic contains:

1. Overview of topic
2. Implementation of algorithms that are within the topic

Topics Covered:

**IN ORDER OF APPEARANCE*

- Permutation Generation
- Combination Generation
- GCD
- LCM
- Dijkstra's Algorithm
- Bellman Ford's Algorithm
- Floyd-Warshall's Algorithm
- Topological Sort
- Kruskal's Algorithm
- Point in Polygon
- Line-Line Intersection
- Line-Plane Intersection
- Polygon Area
- Convex Hull
- Matrix Chain Multiplication DP
- Longest Common Subsequence DP
- Knapsack DP
- Network Flow

Permutation:

any reordering of the elements in a list L. All elements must be used and reordered

for a list of size N, there are N! permutations
*Generating all permutations of a list greater than 12 is unfeasible

Example: L={a,b,c} , Permutations: abc, acb , bac, bca, cba, cab

algorithm:

```
int[] perm;
public static void permute(int p, boolean[] choices, char[] L){

        //reach base case, permutation has been generated
        if(p==L.length){
        //do something to permutation
        return;
        }

        for(int i=0;i<L.length;i++){

        if(choices[i]==false){

        choices[i]=true;
        perm[p]=L[i];
        permute(p+1,choices,L);
        choices[i]=false;

        }


        }
        }
```

Combination:

A combination ,of size k, of a set L is any subset of L that has exactly k elements

For a list of N elements, there are $C_K^N = \frac{N!}{K!(N-K)!}$ of combinations of k elements.

The max size of combinations from a list of N elements occur when K=N/2;

Performance: combinations has a better run time than combinations, but K should be relatively small, like less than 10

Example: L={a,b,c} , generating all combinations of size 2, $C_2^3 = \frac{3!}{2!} = 3$

{a,b}, {a,c} , {b,c}

algorithm:

```
/*
 * Note, order does not matter, and similar to generating
 * subsets, except need to prune the amount of subsets you make
 */
public static void combination(int p, int size, int K, boolean[] choices, char[] L){

	//base case 1: no amount more of recursion
	//...will make combination
	if(L.length-p<K-size){
		return;
	}

	//base case 2: completed all choices to
	//...make combination
	if(size==K){

		//do something to combination

		return;
	}

	/*
	 * how to make binary decision
	 */


	// pick current choice
	choices[p]=true;
```

```
            //increase size of set, recurse
            combination(p+1,size+1,K,choices,L);
            //de-select current choice
            choices[p]=false;
            //recurse
            combination(p+1,size,K,choices,L);



    }
```

GCD:

The greatest common divisior of a and b is the greatest number that cleanly divides between a and b

if, g=GCD(a,b) , then g|a and g|b

Euclids Algorithm to find the GCD:

```
    public static int gcd(int a, int b){

            if(b==0) {
                    return a;
            }

            return gcd(b, a%b );

    }
```

LCM:

The least common multiple of a and b is the smallest number k that k is divisible by both a and b algorithm:

```
public static int lcm(int a,int b){

            return (a*b)/gcd(a,b);
    }
```

Dijkstra's Algorithm:

-Shortest Path

2d int Array Grid, 2d char Array arr(to hold maze or other abstract) Queue Q (Set for all nodes in graph

```
Arrays.fill(Grid, inf )          //Set every point (vertex) on that graph ,  to infinity or a really large #

Grid[Source] = 0                 //Distance from Initial position to itself is obv 0.
PriorityQueue Q:

while( !Q.isEmpty()   ) {      //While Q isn't  empty

        Node current  = queue.poll();
        //Get location/info off the neighbors of current

        For each valid neighbor of current
        if( Grid[ neighbor.row ][ neighbor.col ]  == 0 ){
                Grid[ neighbor.row ][ neighbor.col ] =  Grid[ current.row ][ current.col ] + 1;
                Q.offer( neighbor );
        }

        if(  Arr[ neighbor.row ][ neighbor.col ] == END )
                return Grid[ current.row ][ current.col ] + 1;


        return 0;                    //If no valid path exists

}
```

Ford's Algorithm:

```
1d int Array dist[]          //Array to hold min distance from source node to End
1d edge Array edges[];   //Array to hold edges & their info,
                         //u start node
                         //v destination node
                         //w node
int start;

// Set all values in Grid to inf
Arrays.fill(Grid, inf );



dist[ start ] = 0;   //set start to 0
for( i : n - 1 )    //vertices/nodes - 1
        for( j : num of Edges )
```

```
if( dist[  edges[j].u ] + edges[j].weight < d[ edges[j].v )
        dist[ edges[j].v ] = dist[ edges[j].u ] + edges[j].weight;
```

Floyd's Algorithm:

-Using a 2d Array that will store the shortest distance between a pair of objects(nodes, etc..)
i.e grid[][]

```
for (int k = 0; k < grid.length; k++)
    for (int i = 0; i < grid.length; i++)
        for (int j = 0; j < grid.length; j++)
            if ( grid[i][k] + grid[k][j] < grid[i][j] ){

                    grid[i][j] = grid[i][k] + grid[k][j];

                }
```

TopoSort Algorithm:

```
int cnt = 0, flag = true;
        for (int i = 0; i < matrix.length; i++)
            for (int j = 0; j < matrix.length; j++)
                incoming[j] += (matrix[i][j] ? 1 : 0);


    Queue q = new Queue();

        // Any vertex with 0 incoming edges is ready to be visited, so add 2 queue
        for (int i = 0; i < matrix.length; i++)
            if (incoming[i] == 0)
                q.add(i);

        while ( !q.isEmpty )
        {
            // oull vertex from queue add to way array
            int node = q.remove();
```

```
                // Add each node into way array in toposort order
                way[cnt] = node;
                ++cnt;

                for (int i = 0; i < matrix.length; i++)
                        if (matrix[node][i] && --incoming[i] == 0)
                                q.add(i);
        }


        // If we pass out of the loop without including each vertex in graph there must b a
        cycle

        if (cnt != matrix.length)
                return false;

        // Vertices are from 1 to n
        x -= 1;
        y 1= 1;


        if(x == y)
                return false;  //Base Case

        if( matrix[y][x] == true)   //Backtracking Base case
                return false;


        for( int i = 0; i < matrix.length; i ++){
                if (matrix[i][x] == true){

                        //do backtrackign on incoming edges, if succesful set flag to true
                        flag = backtracking(i,y);

                        //if flag return false no toposort
                        if(  !flag )
                                return false;
                }
        }
        return true;
}
```

Kruskal's Algorithm:

```
Grid[ n ][ n ];                 //2d array to hold values
parent[ n ];

int u = 0;
int v = 0;
int min, numEdges,total;

//Initialize grid to infinitie

Arrays.fill( Grid, inf );
while (  numEdges  < Grid.size ){

        min = inf;
        for( i : Grid.size )
            for( j : Grid.size ){

        if( Grid[ i ][ j ] < min ) {
                min = Grid[ i ][ j ];
                u = i;
                v = j;
        }
}

while( parent[u] != 0 ){
        u = parent[u];
}
while( parent[v] != 0 ){
        v = parent[v];
}

if(  v != u ){
        //Edge was found
        numEdges+= 1;
        total          += 1;
        parent[ v ] = u;
}

Grid[ i ][ j ] = inf;
Grid[ j ][ i ] = inf;
}
//Total now holds the weight of the minimum spanning tre
```

Geometry :

Define a point, point in 3D, a 3D line, a 2D line, a plane, and a polygon:

```
static int MAX_POINTS=200;
	static double epsilon = 0.000001;
	static class Point{
		double x,y;

		public Point(){
			this.x=Double.POSITIVE_INFINITY;
			this.y=Double.POSITIVE_INFINITY;
		}
	}



	static class Point3D{
		double x,y,z;
		public Point3D(){
			this.x=Double.POSITIVE_INFINITY;
			this.y=Double.POSITIVE_INFINITY;
			this.z=Double.POSITIVE_INFINITY;




		}

	}

	static class Line3D{
		Point3D v;
		Point3D p;
		Double constant=0.0;

		public Line3D(Double a0,Double b0, Double c0,Double a, Double b, Double c, Double D){
			this.v=new Point3D();
			this.p=new Point3D();

			this.v.x=a;
			this.v.y=b;
			this.v.z=c;

			this.p.x=a0;
			this.p.y=b0;
			this.p.z=c0;
```

```
                this.constant=D;
            }
        }

        static class Plane{

            Point3D point;
            Point3D normal;
            Double constant=0.0;

            public Plane(Double a,Double b,Double c, Double nx, Double ny, Double nz, Double D){
                this.point=new Point3D();
                this.normal=new Point3D();
                this.point.x=a;
                this.point.y=b;
                this.point.z=c;

                this.normal.x=nx;
                this.normal.y=ny;
                this.normal.z=nz;

                this.constant=D;
            }

        }

static class Line2D{
            double x,y,c;
        }
```

Line - Line Intersection:

```
/*
         * function checks if lines are parallel
         * NOTE: does not account for if lines are same
         */
        static boolean isLineParallel(Line2D u, Line2D v)
        {
            Double sameX = Math.abs(u.x-v.x);//checks if lines have same xConstant
            Double sameY = Math.abs(u.y-v.y);//checks if lines have same yConstant
            if ( (sameX<= epsilon) && ( sameY<= epsilon) ){
                return true;
            }
            else{
                return false;
            }
```

```
}

/*
 * function will check if they are same line if
 * ...lines are parallel and have same solution
 */
static boolean isSameLine(Line2D u, Line2D v){
boolean isParallel = isLineParallel(u,v);
Double constantComparison = Math.abs(u.c-v.c) ;//checks if lines have same constant

	if ( isParallel && (constantComparison <= epsilon)) {
		return true;
	}
	else{
		return false;
	}
}

/*
 * function to determine the intersection of two lines
 * NOTE: This is for a two dimensional line
 */
static Point intersection_point(Line2D u, Line2D v)
{
	Point p = new Point();
	//this condition checks if both lines are the same line
	if(isSameLine(u,v))
	{
		System.out.println("Lines are the same");
		p.x = p.y = 0.0;
		return p;
	}
	//if this condition is true, then lines are parallel
	//...and not the same
	if(isLineParallel(u,v))
	{
		System.out.print("Lines are parallel.");
		return p;
	}

	//NOTE: The point of intersection is computed
	//...by using Cramer's rule
	p.x = (v.y*u.c - u.y*v.c) / (v.x*u.y - u.x*v.y);
	if (Math.abs(u.y) > epsilon)
		p.y = - (u.x * (p.x) + v.c) / u.y;
	else
		p.y = - (u.x * (p.x) + v.c) / v.y;
	return p;
```

```
	}
```

Line - Plane Intersection:

```
//determine if line intersects plane

	/*
	 * Note, this assumes you already have line, if not,
	 * need to compute line from two points
	 *
	 * assumes you also have a point and a normal to define a plane
	 */
	public static boolean doesLineIntersectPlane(Line3D l, Plane q){

		//check if there is a solution to the equation

		//eqn: line: <a0,b0,c0>+ <d,e,f>

		//n=<a,b,c>

		//check if a*d+b*e+c*f=0
		Double check = q.normal.x*l.v.x+q.normal.y*l.v.y+q.normal.z*l.v.z;
		//System.out.println("Check= "+check);
		if(check!=0){
				//if not, then return soln:
			Double lambda = (q.constant-q.normal.x*l.p.x-q.normal.y*l.p.y-
q.normal.z*l.p.z)/check;

			System.out.println("There is an intersection at lamba: "+lambda);
			return true;
		}
		else{

			//System.out.println("There is no intersection");

			//pick any point on line and plug into plane eqn,
			//if it is satisfied, then done

			//else, check if line is on the plane

			//calculate new point from random lamda value, P0
			Double lambda=0.5;
			Point3D P0=new Point3D();
			P0.x=l.p.x+lambda*l.v.x;
			P0.y=l.p.y+lambda*l.v.y;
			P0.z=l.p.z+lambda*l.v.z;
```

```
                    //P is point from plane
                    //calculate vector from Point of plane and new point PP0
                    Point3D vec = new Point3D();
                    vec.x=P0.x-q.point.x;
                    vec.y=P0.y-q.point.y;
                    vec.z=P0.z-q.point.z;

                    //dot product PP0 vector to normal
                    Double dot=vec.x*q.normal.x+vec.y*q.normal.y+vec.z*q.normal.z;
                    System.out.println("Dot: "+dot);
                    if(dot<=epsilon){
                            //if zero, then point lies on plane
                            System.out.println("No intersection, Point lies on plane");

                            return false;
                    }



            //else, the line is not intersecting
                    else{
                            System.out.println("No intersection");
                    return false;
                    }

            }




    }
```

Polygon Area:

```
static double calcArea(Polygon poly){
            /*
             * note: when taking area, remember to take
             *...absolute value, result is signed
             *
             *assumed all points are all initalized to 0.0 0.0 if unused
             */

            double Area=0.0;

            for(int i=1;i<MAX_POINTS;i++){
                    Area+=.5*(poly.p[i].x-poly.p[i-1].x)*(poly.p[i].y+poly.p[i-1].y);
```

```
        }

        return Math.abs(Area);

    }
```

Convex Hull:

When finding convex hull:

1) sort points and remove any duplicate points, this will cause algorithm to break
    a) note: sort by y coordinates, then x coordinates
        i) first coordinate will be one with smallest x and smallest y
2) start with left-most and bottom most point
3) iterating through sorted list, form a vector from current point to previous point
4) from that vector, select the point with the vector that makes smallest positive angle with previous vector
    a) if several points are collinear, pick point farthest from the current point
5) when you loop back and are forming a vector with the initial point, the algorithm is complete

Point in Polygon:

```
    public static double computeAngle(Point u, Point v){

        //compute dot product
        double dot = u.x*v.x+u.y*v.y;
        //compute magnitudes and multiply them
        double magnitudes = Math.sqrt(u.x*u.x+u.y*u.y)+Math.sqrt(v.x*v.x+v.y*v.y);
        //take arcCos(dot/magns)
        if(magnitudes<=epsilon){
            return 0.0;
        }
        else{
        return Math.acos(dot/magnitudes);
        }



    }

//finding if a point is inside a polygon
    public static boolean isInsidePolygon(Polygon poly, Point q){

        double angle=0.0;
```

```
//loop through all points in polygon
for(int i=1;i<poly.p.length;i++){
//check if the point q is the current point of the polygon
if(q.x==poly.p[i-1].x&&q.y==poly.p[i-1].y){
        return false;
}
else if(q.x==poly.p[i].x&&q.y==poly.p[i].y){
        return false;
}

//need to check if point lies on edge of polygon

//else, compute vectors P(i-1)Q and P(i)Q and find angle in between them
else if(poly.p[i].x!=Double.POSITIVE_INFINITY){
Point vec1=new Point();
Point vec2=new Point();

vec1.x=q.x-poly.p[i-1].x;
vec1.y=q.y-poly.p[i-1].y;

vec2.x=q.x-poly.p[i].x;
vec2.y=q.y-poly.p[i].y;


double angleA=computeAngle(vec1,vec2);
System.out.println("AngleA: "+angleA);

if(angleA!=Double.NaN){
 angle+=angleA;
}
System.out.println("Angle: "+angle);
//in this case, point is outside polygon



}

}//end of loop

//take absolute value
angle=Math.abs(angle);

if(angle<=epsilon){
        return false;
}
//if true, then not inside polygon
```

```
            else if(angle<Math.PI){

                return false;
            }
            //if this case is true, then point lies on edge of polygon
            else if(angle==Math.PI){
                System.out.println("NOTE, point q on edge");
                return true;
            }

            //if case is true, then inside polygon
            else {
                return true;
            }


        }
```

Matrix Chain Multiplication:

-Matrix Chain Multiplication is an application of parenthesization. For our purposes, it seems it is interchangeable.

-Matrix Chain Multiplication can be parenthesized using the associative property. Matrix multiplication cannot be commutative (a x b = b x a) because the order matters.

-Simply put, the order of the occurrences matters. If there are three matrices parenthesized in the following way: A(BxC) the answer is not the same as B(AxC)

-The goal in Parenthesization is to minimize the number of operations.

1. **Let's get back to our example: We will show that the way we group matrices when multiplying A, B, C matters:**
    1. Let A be a 2x10 matrix
    2. Let B be a 10x50 matrix
    3. Let C be a 50x20 matrix
2. **Consider computing A(BC):**
    1. # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
    2. # multiplications for A(BC) = 2x10x20 = 400
    3. Total multiplications = 10000 + 400 = 10400.
3. **Consider computing (AB)C:**
    1. # multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix
    2. # multiplications for (AB)C = 2x50x20 = 2000,
    3. Total multiplications = 1000 + 2000 = 3000

Sub-problem Optimality Condition:

-Sounds really complicated :^)

-"If a particular parenthesization of the whole product is optimal, then any sub-parenthesization in that product is optimal as well.

-"Say What?

1. If (A (B ((CD) (EF)))) is optimal
2. Then (B ((CD) (EF))) is optimal as well"

Programming Solution:

- The solution as I interpret it requires us to test every possible permutation of our Matrix Chain.
- Then count the number of calculations (as we did in our example) for each permutation
- Then return the lowest calculation.
- LongForm explanation:
    - "•In essence, there is exactly one value of k for which we should "split" our work into two separate cases so that we get an optimal result.
    - •Here is a list of the cases to choose from:
    - •(A0) · (A1· Ak+2· ... An-1)
    - •(A0· A1) · (A2· Ak+2· ... An-1)
    - •(A0· A1·A2) · (A3· Ak+2· ... An-1)
    - •...
    - •(A0· A1· ... An-3) · (An-2 · An-1)
    - •(A0· A1· ... An-2) · (An-1)"

Pseudocode Algorithm:

```
Initialize N[i][i] = 0, and all other entries in N to infinity
for i=1 to n-1 do the following
        for j=0 to n-1-i do
                for k=j to j+i-1
                        if (N[j][j+i-1] > N[j][k]+N[k+1][j+i-1]+djdk+1di+j)
                                N[j][j+i-1]= N[j][k]+N[k+1][j+i-1]+djdk+1di+j
```

Java Algorithm:

```
int MatrixChainMultiplication(int sizes[], int dimension){
        //initialize table[dimension][dimension] and all other entries to zero
        //this is our base case. the table will help find the optimal number of calculations
        int table = new int[dimension][dimension];
        for (int i = 0; i < dimension; i++){
                for (int j = 0; j < dimension; j++){
                        tabe[i][j] = 0;
                }
        }

        //calculate every element in the helper table. the last element in the table is our optimized number
        //our i is going to be the chain length
        for (int i = 2; i < dimension; i++){
                for (int j = 0; j < dimension - i - 1; j++){
                        for (int k = j; k < j + i + 1; k++){
                                if (table[j][j + i + 1] > table[j][k] + table[k + 1][j + i -1] + sizes[i - 1] *
sizes[j] * size[k])
                                table[j][j + i - 1]=table[j][k] + table[k + 1][j + i -1] + sizes[i - 1] *
sizes[j] * size[k];
                        }
                }
        }

        return table[1][dimension - 1];
}
```

Knapsack Dynamic Programming

The classic Knapsack problem originates from a comical and otherwise devious situation outlined as follows:

You are a successful thief with a mathematical inclination. Being in such a high risk occupation you thought it was necessary to optimize the value you receive from every job. As a thief, your accessory of choice is a backpack to store all the collected loot. The problem with optimizing the value of every successful robbery is the weight of your backpack. Although there are potentially many valuable items within a location you are only able to carry items that you can carry.

Given the dimensions of items with their corresponding value (that you determine on the spot through loyal watching of The Price is Right) find the optimal selection of items that will maximize the value of a robbery.

The constraint in our situation is weight. In other fields the problem is much more complex. We deal with a one dimensional physical trait, weight, while a more practical question would be multi dimensional. This would include height, width, length, and possibly weight.

Algorithm:

1. We will be using a table to keep track of our computations. Each next element of the table will be dependent on the previous. It may be easier to understand using a recurrence relation.
2. We will place the numerical value (how much an item would be if we pawned it) for the x value
3. We will place the constraint value (our weight or the price in the code below) for the y value
4. Iterate through every element starting with the first y value while filling in the x values
5. For each iteration apply the following to the table / matrix element:
    a. if the price of the current item (item at x) is less than or equal to our x value (in chronological order) then:
        i. the element equals whichever of the following is the most:
            1. constraint of the last element plus the corresponding element
            2. the previous element
    b. otherwise make it equal to the previous element
6. The answer is the last element in the matrix

Java Code;

```
public static int knapsack(int[] time, int[] price, int purse){
        int n = price.length;
        int[][] matrix = new int[n + 1][purse + 1];

        //setup x coordinates
        for (int i = 0; i <= purse; i++){
                matrix[0][i] = 0;
        }
        //setup y coordinates
        for (int i = 0; i <= price.length; i++){
                matrix[i][0] = 0;
        }

        //fill out the correct matrix
```

```
        for (int i = 1; i <= n; i++){
                for (int j = 1; j <= purse; j++){
                        if (price[i - 1] <= j){
                                matrix[i][j] = Math.max(time[i - 1] + matrix[i - 1][j - price[i - 1]], matrix[i -
1][j]);
                        }
                        else{
                                matrix[i][j] = matrix[i - 1][j];
                        }
                }
        }
        return matrix[n][purse];
  }
```

Longest Common Subsequence Dynamic Programming

The problem is straightforward and simple to understand with some attention to detail. For our purposes the problem is as states:

Given two strings find the length of the longest common subsequence.

For our purposes we do not necessarily need to find the longest common subsequence but merely the length of it. Consider the following example:

- Given the 2 strings:
    - ABCDGH
    - AEDFHR
- Find the longest subsequence of letters. Remember to note that a subsequence does not have to have every letter adjacent to the last but they have to be in chronological order
- Based on the above, the length of the longest common subsequence is 3, or ADH

Algorithm:

Our dynamic programming algorithm will again use a table to break the problem down into smaller subproblems. Each element of the table will be dependent on the last and the solution is housed in the last element.

Java Code:

```
public int longestCommonSubsequence(String _a, String _b){
        char[] a = _a.toCharArray();
        char[] b = _b.toCharArray();

        int[] table = new table[a.length + 1][b.length + 1];

        //calculate table
        for (int i = 0; i < a.length; i++){
                for (int j = 0; j < b.length; j++){
                        if (a[i] == b[j]){
                                table[i + 1][j + 1] = table[i][j] + 1;
                        }
                        else{
                                table[i + 1][j + 1] = Math.max(table[i + 1][j], table[i][j + 1]);
                        }
```

```
                }
        }

        return table[a.length][b.length];
}
```

Network Flow Problem

An easy way to visualize the problem is through the exciting field of plumbing and its nuances. For our exercise imagine that you are a plumber. Maybe an eccentric italian with a large mustache and an insatiable thirst for princesses in distress? But unlike the nefarious nature of the small italian plumber in a red uniform you are tasked with an actual plumbing problem.

Given a network of pipes find the most sewage water that can flow from a source to a sink. Your task is made more difficult because your boss ordered different types of pipes because they were on sale. The different pipes have different flow capacities. This means some pipes may be used to their maximum flow while others may not.

In terms of algorithm analysis the network of pipes is a graph and each pipe can be represented as a node within the graph. The flow capacity can be interpreted as edge weights between pipes.

We will now define an augmented path. An augmented path is a valid path from a source to a sink. The residual capacity is the unused capacity of a path. If we combine the two concepts we have a residual graph that tells us any possible additional flow.

To solve our problem, we will be using what is called the Ford Fulkerson Algorithm:

In the simplest terms the Ford Fulkerson Algorithm (or FF algorithm for short) begins by setting the initial flow to zero. Then the algorithm will look at all possible augmented paths and add it to our repository of flows. In our case, we will use Breadth First Search to look at our outcomes. The algorithm then returns the augmented path with the most flow. Although this is not the fastest algorithm it is able to be understood easily and implemented.

```
import java.util.*;

public class flow{
   static boolean[] visited;

   public static void main(String args[]){

   }

public static int maxFlow(int source, int sink, int nodes[], int[][]graph, int numPipes){
        //uses the ford fulkerson algorithm
        int max = 0;

        int[][] residGraph = new int[numPipes + 1][numPipes + 1];
```

```
        //construct our initial residual graph
        for (int i = 1; i < numPipes; i++){
                for (int j = 1; j < numPipes; j++){
                        residGraph[i][j] = graph[i][j];
                }
        }

        while(BFS(source, sink)){
                int flow = Integer.MAX_VALUE, index1, index2;

                index1 = sink;
                while( index1 != source ){
                        index2 = nodes[index1];
                        flow = Math.min(flow, residGraph[index2][index1]);

                        index1 = nodes[index1];
                }

                index1 = sink;
                while (index1 != source){
                        index2 = nodes[index1];

                        //update our residual flows
                        residGraph[index2][index1] -= flow;
                        residGraph[index1][index2] += flow;
                }

                max += flow;
        }

        return max;
}

public static int BFS(int source, int sink, int nodes[], int[][]graph, int numPipes){
        boolean found = false;

        //setup our BFS specifically for the ff algorithm
        for (int i = 1; i <= numPipes; i++){
                nodes[i] = Integer.MIN_VALUE;
                visited[i] = false;
        }

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

        while (!queue.isEmpty()){
                //dequeue
                Integer current = queue.remove();
```

```
                int count = 1;

                //add all the connections into the queue
                while(count <= numPipes){
                        if (!visited[count] && graph[current][count]){
                                nodes[count] = current;
                                queue.add(count);
                                visited[count] = true;
                        }

                        count++;
                }
        }

        //merely checks to see if it is an augmented path
        return visited[sink];
    }
}
```