Algorithmics	Student information	Date	Number of session
	UO: 296503	07/04/2025	Session6
	Surname: Mulet Alonso		
	Name: Sergio		

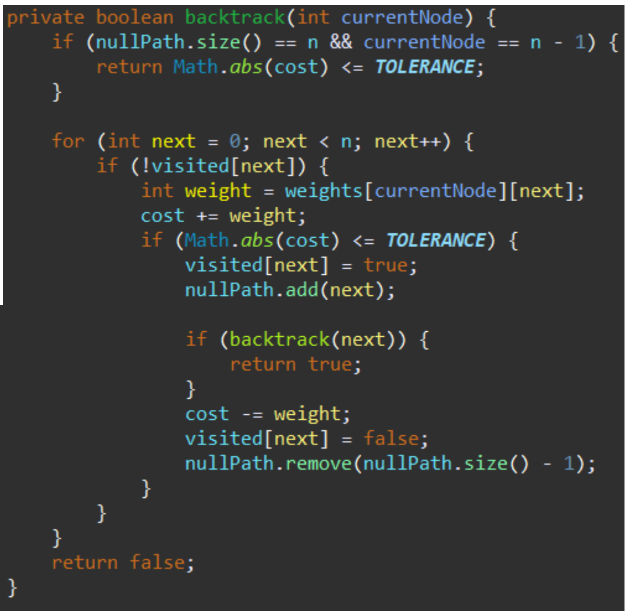
Escuela de Ingeniería Informática
Universidad de Oviedo

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Activity 1. NullPath

We can reuse some code of last practice for filling the weights making the appropriate changes.

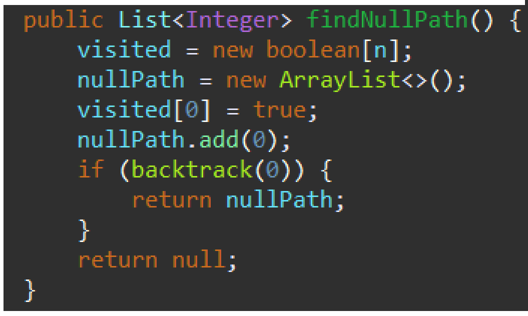
Then we have to implement the main algorithm to find the null path.



```java
private boolean backtrack(int currentNode) {
    if (nullPath.size() == n && currentNode == n - 1) {
        return Math.abs(cost) <= TOLERANCE;
    }

    for (int next = 0; next < n; next++) {
        if (!visited[next]) {
            int weight = weights[currentNode][next];
            cost += weight;
            if (Math.abs(cost) <= TOLERANCE) {
                visited[next] = true;
                nullPath.add(next);

                if (backtrack(next)) {
                    return true;
                }
                cost -= weight;
                visited[next] = false;
                nullPath.remove(nullPath.size() - 1);
            }
        }
    }
    return false;
}
```



```java
public List<Integer> findNullPath() {
    visited = new boolean[n];
    nullPath = new ArrayList<>();
    visited[0] = true;
    nullPath.add(0);
    if (backtrack(0)) {
        return nullPath;
    }
    return null;
}
```

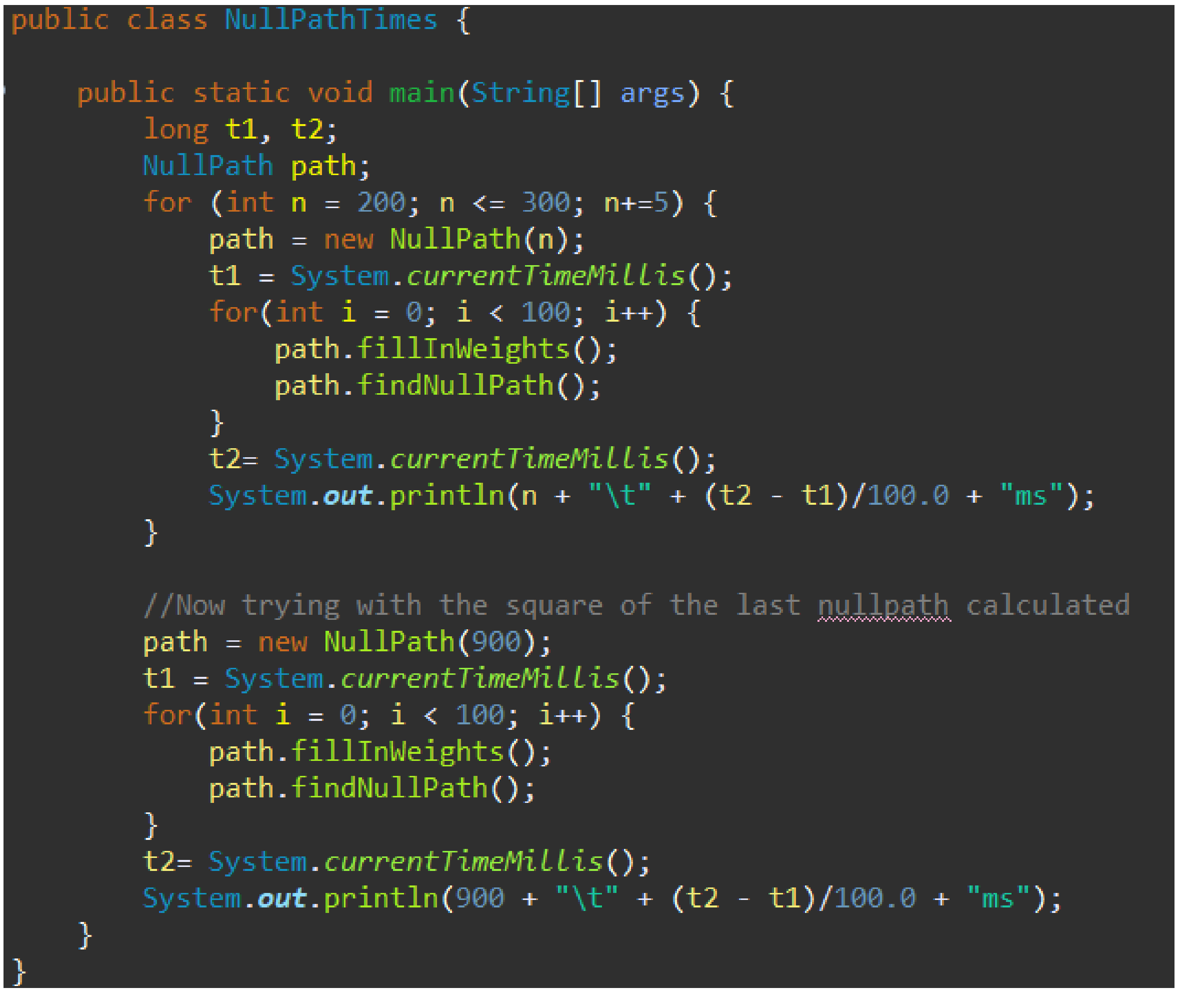
We always have to start with the first node and end with the last. Using a Boolean array to mark which nodes have been already visited is key for performance, so we are not continuously losing time on something that has been already calculated.

We check what would be the cost if we add the cost of the next path, if it is bigger than tolerance we simply go to the next one because it is not a valid path. If the cost is valid, we check backtracking for that next node, if we don't find the solution, we have to redo the changes, that is going back to previous cost and mark the node as not visited and remove it from the null path that is being calculated. The worst case would be finding the solution with the last nodes, because that would mean that we have spent a lot of time doing backtracking with previous ones for getting no solution.

	Student information	Date	Number of session
Algorithmics	UO: 296503	07/04/2025	Session6
	Surname: Mulet Alonso		
	Name: Sergio		

Activity 2. NullPathTimes

We simply create the class as we have been doing with other practices, in my case I have moved the creation of NullPath object before measuring time, just to not lose that non-significant time, anyway, keep in mind that the fulfillment of weights array is done while measuring time.



```java
public class NullPathTimes {

    public static void main(String[] args) {
        long t1, t2;
        NullPath path;
        for (int n = 200; n <= 300; n+=5) {
            path = new NullPath(n);
            t1 = System.currentTimeMillis();
            for(int i = 0; i < 100; i++) {
                path.fillInWeights();
                path.findNullPath();
            }
            t2= System.currentTimeMillis();
            System.out.println(n + "\t" + (t2 - t1)/100.0 + "ms");
        }

        //Now trying with the square of the last nullpath calculated
        path = new NullPath(900);
        t1 = System.currentTimeMillis();
        for(int i = 0; i < 100; i++) {
            path.fillInWeights();
            path.findNullPath();
        }
        t2= System.currentTimeMillis();
        System.out.println(900 + "\t" + (t2 - t1)/100.0 + "ms");
    }
}
```

At the end I also test for a value 3 times bigger than the last calculates, just for get a good reference when measuring times.

	Student information	Date	Number of session
Algorithmics	UO: 296503	07/04/2025	Session6
	Surname: Mulet Alonso		
	Name: Sergio		

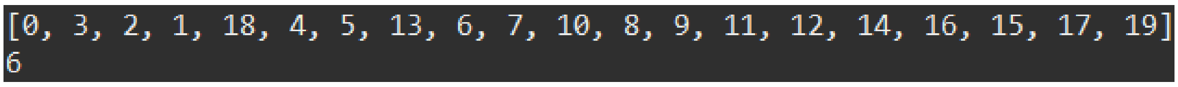
Activity 3. Average times obtained

n	t(ms)
200	12,17
205	12,78
210	13,35
215	14
220	14,74
…	…
300	28,29
900	251

Since the complexity of the backtracking should be O(n!) these times doesn't make much sense, but there is an explanation.

If you look at the times, the algorithm seems to follow quadratic complexity, as in the end you can se that by multiplying the time by 3, times got multiplied by 9.

This is due to the requirements of our problem, we have a tolerance equals to the maximum path value, and we are generating weights from [-99, -10] and from [10, 99] equally (being negative or positive have the same probability). This makes our problem much simpler, because most of the time we are getting a null path quite similar to the original graph, for instance this is the result for 20 nodes.



```
[0, 3, 2, 1, 18, 4, 5, 13, 6, 7, 10, 8, 9, 11, 12, 14, 16, 15, 17, 19]
6
```

If we work with a stricter tolerance and different maximum and minimum values, results would be different, and times would reflect much better the complexity.