Väldigt förenklat består vår algoritm av en while-loop som innehåller 2 for-loopar, Tidskomlexiteten för Lab2a blir alltså $n * (n + n) = O(n^2)$.

```
while(resultPoly.length > k*2){
//some code
    for (int i = 1; i < resultPoly.length/2 - 1; i++){
        //some code
    }

    for( int i = 0; i < resultPoly.length; i++ ){
        //some code
    }
}
```

För Lab2b har vi istället 2 loopar, inräknat komplexiteten för operationerna på queue får vi $n * \log(n) + n * (\log(n) + n) = O(n^2)$.

```
for (int i = 1; i < poly.length / 2 - 1; i++) {
        //some code
        queue.add(prevListNode);
        //some code
    }

    while (queue.size() > k - 2) {
        DLList.Node nodeToRemove = queue.poll();
        //some code
        for( int i = 0; i < 2; i++){
                //some code
                queue.remove(nodesToRecalc[i]);

        }
        //some code
    }
    //some code
}
```

Javas LinkedList ger ej förbättrad tidskomplexitet eftersom att ta bort ett element från en LinkedList har linjär komplexitet. Poll i en priorityQueue har logaritmisk komplexitet och eftersom vår implementation ger oss tillgång till nodens grannar kan vi updatera deras värde på konstant tid. Tyvärr måste vi fortfarande ta bort element vilket har linjär komplexitet så det blir ingen skillnad i komplexitet.