

Wyniki dla przykładów z zajęć:

```
▼ example1 = {Algorithm@796} "Algorithm{nodesAmount=7}"
  > nodesAmount = {Integer@857} 7
  > root = {Node@858} "Node{value=1}"
  ▼ maximalIndependentSet = {LinkedHashSet@859} size = 5
    > 0 = {Node@858} "Node{value=1}"
    > 1 = {Node@862} "Node{value=4}"
    > 2 = {Node@863} "Node{value=5}"
    > 3 = {Node@864} "Node{value=6}"
    > 4 = {Node@865} "Node{value=7}"
    > timeInMilliseconds = {Long@849} 1
  ▼ example2 = {Algorithm@797} "Algorithm{nodesAmount=13}"
    > nodesAmount = {Integer@846} 13
    > root = {Node@847} "Node{value=1}"
    ▼ maximalIndependentSet = {LinkedHashSet@848} size = 9
      > 0 = {Node@847} "Node{value=1}"
      > 1 = {Node@870} "Node{value=4}"
      > 2 = {Node@871} "Node{value=13}"
      > 3 = {Node@872} "Node{value=5}"
      > 4 = {Node@873} "Node{value=6}"
      > 5 = {Node@874} "Node{value=9}"
      > 6 = {Node@875} "Node{value=10}"
      > 7 = {Node@876} "Node{value=11}"
      > 8 = {Node@877} "Node{value=12}"
      > timeInMilliseconds = {Long@849} 1
```

Wyniki dla przykładów losowych:

```
▼ task1000 = {Algorithm@904} "Algorithm{nodesAmount=1000}"
  > nodesAmount = {Integer@908} 1000
  > root = {Node@909} "Node{value=-173689760}"
  > maximalIndependentSet = {LinkedHashSet@910} size = 539
  > timeInMilliseconds = {Long@911} 111
  ▼ task100000 = {Algorithm@913} "Algorithm{nodesAmount=100000}"
    > nodesAmount = {Integer@918} 100000
    > root = {Node@919} "Node{value=1380321272}"
    > maximalIndependentSet = {LinkedHashSet@920} size = 53576
    > timeInMilliseconds = {Long@921} 104813
```

Kod:

```
package model;

import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
import java.util.List;
import java.util.Objects;
import java.util.Set;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@AllArgsConstructor
@NoArgsConstructor
@Data
public class Node {

    private int value;
    private List<Node> children = Lists.newArrayList();
    private Set<Node> independentSet = Sets.newLinkedHashSet();
    private Integer independentSetSize = 0;

    public Node(
        int value
    ){
        this.value = value;
    }

    public Boolean addChild(
        Node child
    ){
        if (this.children == null) {
            this.children = Lists.newLinkedList();
        }
        this.getChildren().add(child);
        return Boolean.TRUE;
    }

    public Boolean addChildren(
        List<Node> children
    ){
        if (this.children == null) {
            this.children = Lists.newLinkedList();
        }
        this.getChildren().addAll(children);
        return Boolean.TRUE;
    }
}
```

```
public List<Node> getGrandChildren(  
    ) {  
    List<Node> grandChildren = Lists.newLinkedList();  
    for (Node child : this.getChildren()) {  
        grandChildren.addAll(  
            child.getChildren()  
        );  
    }  
    return grandChildren;  
}
```

```
public boolean addIndependentSetNode(  
    Node node  
){  
    if (this.independentSet == null) {  
        this.independentSet = Sets.newLinkedHashSet();  
    }  
    this.independentSet.add(node);  
    return Boolean.TRUE;  
}
```

```
public boolean addIndependentSetNodes(  
    Set<Node> nodes  
){  
    if (this.independentSet == null) {  
        this.independentSet = Sets.newLinkedHashSet();  
    }  
    this.independentSet.addAll(nodes);  
    return Boolean.TRUE;  
}
```

```
public boolean addIndependentSetNodes(  
    List<Node> nodes  
){  
    if (this.independentSet == null) {  
        this.independentSet = Sets.newLinkedHashSet();  
    }  
    this.independentSet.addAll(nodes);  
    return Boolean.TRUE;  
}
```

```
public Set<Node> getChildrenIndependentSet(  
    ) {  
    Set<Node> independentSet = Sets.newLinkedHashSet();  
    for (Node child : this.getChildren()) {  
        independentSet.addAll(  
            child.getIndependentSet()  
        );  
    }  
}
```

```

    return independentSet;
}

@Override
public boolean equals(
    Object o
){
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    Node node = (Node) o;
    return value == node.value;
}

@Override
public int hashCode() {
    return Objects.hash(value);
}

@Override
public String toString() {
    return "Node{" + "value=" + value + '}';
}
}

```

```

package model;

import com.google.common.collect.Sets;
import java.util.Set;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@AllArgsConstructor
@NoArgsConstructor
@Data
public class Algorithm {

    private Integer nodesAmount;
    private Node root;
    private Set<Node> maximalIndependentSet = Sets.newHashSet();
    private Long timeInMilliseconds = 0L;

    public Algorithm(
        Node root,
        Integer nodesAmount
    ) {
        this.root = root;
        this.nodesAmount = nodesAmount;
        this.run();
    }

    /**
     * Metoda uruchamiająca algorytm, zbierającą wyniki i mierzącą czas.
     */
    public int run() {
        long start = System.currentTimeMillis();
        Integer maximalIndependentSetSize = this.processNode(this.root);
        this.timeInMilliseconds = System.currentTimeMillis() - start;
        this.maximalIndependentSet = this.root.getIndependentSet();
        return maximalIndependentSetSize;
    }

    /**
     * Główna metoda algorytmu
     */
    public Integer processNode(
        Node node
    ) {
        // Jeśli węzeł został już przetworzony zwróć jego ciąg
        if (!node.getIndependentSet().isEmpty()) {
            return node.getIndependentSet().size();
        }
        // Jeśli węzeł nie ma dzieci to należy do ciągu
    }

```

```

    if (node.getChildren().isEmpty()) {
        node.getIndependentSet().add(node);
        return node.getIndependentSet().size();
    }
    // Szukanie ciągu
    int setSize = this.findSet(node);
    return setSize;
}

/**
 * Szukanie ciągu dla danego węzła
 */
public Integer findSet(
    Node node
) {
    // Znajdujemy długość ciągu gdy przynależą do niego dzieci aktualnie przetwarzanego ciągu
    Integer childSetSize = this.findChildSet(node);
    // Znajdujemy długość ciągu gdy przynależą do niego wnukowie aktualnie przetwarzanego ciągu
    Integer grandChildSetSize = this.findGrandChildSet(node);
    // Sprawdzamy, który ciąg jest dłuższy
    Integer result = Math.max(childSetSize, grandChildSetSize);
    // Przypisujemy wyniki aktualnie przetwarzanego węzła
    if (grandChildSetSize > childSetSize) {
        node.addIndependentSetNode(node);
    }
    node.addIndependentSetNodes(
        node.getChildrenIndependentSet()
    );
    node.setIndependentSetSize(result);
    return result;
}

/**
 * Szukanie ciągu w wersji z dziećmi
 */
public Integer findChildSet(
    Node parent
) {
    // Początkowa wartość ciągu w wersji dzieci
    Integer childSetSize = 0;
    // Przetwarzanie dzieci
    for (Node child : parent.getChildren()) {
        childSetSize = childSetSize + this.processNode(child);
    }
    return childSetSize;
}

/**
 * Szukanie ciągu w wersji z wnukami

```

```

*/
public Integer findGrandChildSet(
    Node parent
){
    // Początkowa wartość ciągu w wersji wnukowie (1, bo aktualnie przetwarzany węzeł)
    Integer grandchildrenSetSize = 1;
    // Przetwarzanie wnuków
    for (Node grandChild : parent.getGrandChildren()) {
        grandchildrenSetSize = grandchildrenSetSize + this.processNode(grandChild);
    }
    return grandchildrenSetSize;
}

@Override
public String toString() {
    return "Algorithm{" + "nodesAmount=" + this.nodesAmount + '}';
}
}

```