
















Błażej Piekos 273210

Wyniki:

```
▼  task10 = {Algorithm@805} "Algorithm{nodesAmount=10}"
  >  nodesAmount = {Integer@899} 10
  >  root = {Node@900} "Node{value=1670739772}"
  >  maximalIndependentSet = {LinkedHashSet@901} size = 5
  >  timeInMilliseconds = {Long@902} 0
▼  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.Sets;
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 Node left;
    private Node right;
    private Set<Node> independentSet = Sets.newLinkedHashSet();

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

    public void setLeft(Node left) {
        this.left = left;
    }

    public void setRight(
        Node right
    ){
        this.right = right;
    }

    public void addToIndependentSet(
        Node node
    ){
        if (
            node != null
        ){
            if (
                this.independentSet == null
            ){
                this.independentSet = Sets.newLinkedHashSet();
            }
            this.independentSet.addAll(
                node.getIndependentSet()
            );
        }
    }
}
```

```
}  
}
```

```
public void addToIndependentSet(  
    Set<Node> nodes  
) {  
    for (Node node : nodes) {  
        this.addToIndependentSet(  
            node  
        );  
    }  
}
```

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

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

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

```
}
```

```
package model;
```

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

```
@AllArgsConstructor  
@NoArgsConstructor
```

```

@Data
public class TreeBuilder {

    private int size;
    private Set<Integer> numbers = Sets.newLinkedHashSet();
    private Node root;

    public TreeBuilder(int size) {
        this.size = size;
        this.build();
    }

    /**
     * Budowanie drzewa na podstawie danych wejsciowych przypisanych do pól.
     */
    public void build(
    ) {
        while (this.numbers.size() != this.size) {
            Random random = new Random();
            this.numbers.add(
                random.nextInt()
            );
        }
        this.root = new Node(
            this.pickFromNumbers()
        );
        while (!this.numbers.isEmpty()) {
            this.fillNode(this.root);
        }
    }

    /**
     * Rekurencyjne wypelnianie poddrzew na zasadzie losowosci.
     */
    public void fillNode(
        Node node
    ) {
        if (this.numbers.isEmpty()) {
            return;
        }
        Random random = new Random();
        if (random.nextBoolean()) {
            if (node.getLeft() == null) {
                Node leftNode = new Node(
                    this.pickFromNumbers()
                );
                node.setLeft(leftNode);
            }
            this.fillNode(

```

```

        node.getLeft()
    );
}
if (this.numbers.isEmpty()) {
    return;
}
if (random.nextBoolean()) {
    if (node.getRight() == null) {
        Node rightNode = new Node(
            this.pickFromNumbers()
        );
        node.setRight(rightNode);
    }
    this.fillNode(
        node.getRight()
    );
}
}

/**
 * Pobieranie wartosci dla node'a z drzewa.
 */
public int pickFromNumbers(
) {
    int number = this.numbers
        .stream()
        .findFirst()
        .get();
    this.numbers.remove(
        Iterables.get(
            this.numbers,
            0
        )
    );
    return number;
}

}

package model;

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

@Data
public class Algorithm {

    private Integer nodesAmount;

```

```

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

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

public void run(
) {
    long start = System.currentTimeMillis();
    this.processCurrentRoot(
        this.root
    );
    this.timeInMilliseconds = System.currentTimeMillis() - start;
    this.maximalIndependentSet = this.root.getIndependentSet();
}

/**
 * Szukany jest niezależny ciąg dla aktualnego korzenia z iteracji (z argumentu metody
 * rekurencyjnej)
 */
public void processCurrentRoot(
    Node currentRoot
) {
    // Jeśli dany korzeń był już przeliczany to ta wartość zostanie przywołana
    if (
        currentRoot.getIndependentSet().size() != 0
    ) {
        return;
    }
    // Wyznaczany jest niezależny ciąg bez aktualnego korzenia dla iteracji
    Set<Node> setWithoutCurrentRoot = this.findSetWithoutRoot(
        currentRoot
    );
    // Wyznaczany jest niezależny ciąg z aktualnym korzeniem dla iteracji
    Set<Node> setWithCurrentRoot = this.findSetWithRoot(
        currentRoot
    );
    // Większy ciąg jest przypisywany do aktualnego korzenia z iteracji
    this.attachIndependentSetToRoot(
        currentRoot,
        setWithoutCurrentRoot,
        setWithCurrentRoot
    );
}

```

```

    );
}

/**
 * Wyznaczany jest niezależny ciąg z aktualnym korzeniem dla iteracji
 */
public Set<Node> findSetWithRoot(
    Node root
) {
    Set<Node> setWithCurrentRoot = Sets.newLinkedHashSet();
    setWithCurrentRoot.add(root);
    if (root.getLeft() != null) {
        if (root.getLeft().getLeft() != null) {
            this.processCurrentRoot(root.getLeft().getLeft());
            setWithCurrentRoot.addAll(root.getLeft().getLeft().getIndependentSet());
        }
        if (root.getLeft().getRight() != null) {
            this.processCurrentRoot(root.getLeft().getRight());
            setWithCurrentRoot.addAll(root.getLeft().getRight().getIndependentSet());
        }
    }
    if (root.getRight() != null) {
        if (root.getRight().getLeft() != null) {
            this.processCurrentRoot(root.getRight().getLeft());
            setWithCurrentRoot.addAll(root.getRight().getLeft().getIndependentSet());
        }
        if (root.getRight().getRight() != null) {
            this.processCurrentRoot(root.getRight().getRight());
            setWithCurrentRoot.addAll(root.getRight().getRight().getIndependentSet());
        }
    }
    return setWithCurrentRoot;
}

/**
 * Wyznaczany jest niezależny ciąg bez aktualnego korzenia dla iteracji
 */
public Set<Node> findSetWithoutRoot(
    Node root
) {
    Set<Node> setWithoutCurrentRoot = Sets.newLinkedHashSet();
    if (
        root.getLeft() != null
    ) {
        this.processCurrentRoot(
            root.getLeft()
        );
        setWithoutCurrentRoot.addAll(
            root.getLeft().getIndependentSet()

```

```

    );
}
if (
    root.getRight() != null
){
    this.processCurrentRoot(
        root.getRight()
    );
    setWithoutCurrentRoot.addAll(
        root.getRight().getIndependentSet()
    );
}
setWithoutCurrentRoot.removeAll(
    Collections.singleton(null)
);
return setWithoutCurrentRoot;
}

/**
 * Większy ciąg jest przypisywany do aktualnego korzenia z iteracji
 */
public void attachIndependentSetToRoot(
    Node root,
    Set<Node> setWithoutCurrentRoot,
    Set<Node> setWithCurrentRoot
){
    if (
        setWithoutCurrentRoot.size() < setWithCurrentRoot.size()
    ){
        root.addToIndependentSet(
            setWithCurrentRoot
        );
        root.getIndependentSet().add(
            root
        );
    } else {
        root.addToIndependentSet(
            setWithoutCurrentRoot
        );
    }
}
}
}

```