# Projekt - Skarbonki

Maciej Wójcik

2023

---

## Spis Treści

# 1. Polecenie zadania

Smok Bajtazar ma N skarbonek. Każdą skarbonkę można otworzyć jej kluczem lub rozbić młotkiem. Bajtazar powrzucał klucze do pewnych skarbonek, pamięta przy tym który do której. Bajtazar zamierza kupić samochód i potrzebuje dostać się do wszystkich skarbonek. Chce jednak zniszczyć jak najmniej z nich. Pomóż Bajtazarowi ustalić ile skarbonek musi rozbić.

## Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia liczbę skarbonek i rozmieszczenie odpowiadających im kluczy
- obliczy minimalną liczbę skarbonek, które trzeba rozbić, aby dostać się do wszystkich skarbonek
- wypisze wynik na standardowe wyjście.

## Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita N (1 <= N <= 1.000.000) - tyle skarbonek posiada smok. Skarbonki (jak również odpowiadające im klucze) są ponumerowane od 1 do N. Dalej na wejściu mamy N wierszy: w i+1-szym wierszu zapisana jest jedna liczba całkowita - numer skarbonki, w której znajduje się i-ty klucz.

## Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą - minimalną liczbę skarbonek, które trzeba rozbić, aby dostać się do wszystkich skarbonek.

## Przykład

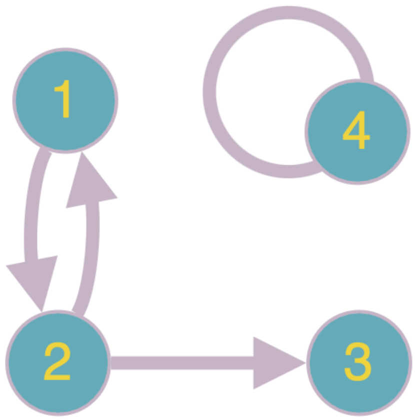
Dla danych wejściowych:
4
2
1
2
4
Poprawnym wynikiem jest:
2
W powyższym przykładzie wystarczy rozbić skarbonki numer 1 i 4

## 2. Opis zadania

Nasz problem sprowadza się do kilku aspektów:

- Od skarbonki w i+1-szym wierszu prowadzimy krawędź do i-tej skarbonki.

Czyli w przykładzie z polecenia:




4 - ilość skarbonek
2 -> 1
1 -> 2
2 -> 3
4 -> 4

- W tym przykładzie, żeby dotrzeć do wszystkich skarbonek musimy rozbić dwie skarbonki:

• Musi to być na pewno odosobniona skarbonka 4, ponieważ w żadnej innej skarbonce nie ma do niej klucza
• Musi to być któraś ze skarbonek 1 lub 2, żeby dostać się do pozostałych skarbonek
• Nie może to być skarbonka 3, ponieważ w niej nie ma żadnego klucza

- Do jednej skarbonki jest tylko jeden klucz. To znaczy, że do wierzchołka może wchodzić maksymalnie jedna krawędź.

- W zadaniu musimy znaleźć minimalną ilość skarbonek, które musimy rozbić, tak aby dostać się do wszystkich skarbonek.

W interpretacji grafowej szukamy najmniejszej ilości wierzchołków dzięki którym przejdziemy przez cały graf.

- Problem można uprościć sprowadzając go do stworzenia drzew, których korzeniem będzie rozbita skarbonka. Naszym rozwiązaniem będzie wyliczenie ile takich drzew powstanie.

- Skoro nie interesuje nas która skarbonka zostanie rozbita, a która otworzona problem sprowadza się do wyliczenia ilości słabo spójnych składowych grafu.

- Ważną zmianą, którą musimy wprowadzić, jest branie pod uwagę grafu, jako graf nieskierowany.

Jeśli nie interesuje nas które skarbonki zostaną rozbite, a tylko ich najmniejsza liczba, to w grafie interesuje nas czy jest przejście z jednego wierzchołka na drugi, a nie kierunek w którym będzie to przejście.

## 3. Pseudokod

Do wyliczenia słabo spójnych składowych użyjemy algorytmu Depth First Search. Ilość wywołań tego algorytmu jest równa z ilością rozbitych skarbonek

```
rozbijacz:
    ilość_rozbitych_skarbonek = 0
    oznacz wszystkie wierzchołki jako nieodwiedzone
    znajdź nieodwiedzony wierzchołek x:
        DFS1(x, tablica_odwiedzonych)
        ilość_rozbitych_skarbonek++
zwróć ilość_rozbitych_skarbonek
```

```
DFS1(vertex, tablica_odwiedzonych):
    oznacz wierzchołek vertex jako odwiedzony
    dla każdego wierzchołka v będącego sąsiadem wierzchołka vertex:
        jeśli wierzchołek v nie został wcześniej odwiedzony:
            DFS1(v, tablica_odwiedzonych)
```

## 4. Złożoność algorytmu

**Złożoność czasowa:**

Ze względu na interpretację iteratora po krawędziach wychodzących, który sprawdza wszystkie krawędzie (nawet te nie istniejące) (Mamy n krawędzi i n wierzchołków) wynosi:

$$O(N^2)$$

**Złożoność pamięciowa:**

Użyty jest wektor odpowiadający za przechowywanie informacji, który wierzchołek został już odwiedzony. Dodatkowa złożoność z tym związana wynosi:

$$O(N)$$

N - ilość wierzchołków

## 5. Złożoność struktur danych

**Złożoność czasowa:**

Utworzenie grafu (utworzenie macierzy n x n i utworzenie n wierzchołków):

$$O(N^2) + O(N) = O(N^2 + N) = O(N^2)$$

Dodawanie krawędzi:

$$O(1)$$

Iterator po wierzchołkach wchodzących i wychodzących (ma do przejścia odpowiednio całą kolumnę o wielkości N lub cały wiersz o wielkości N):

$$O(N)$$

Iterator po wszystkich krawędziach:

$$O(N^2)$$

**Złożoność pamięciowa:**

Przechowywanie macierzy sąsiedztwa $n \ x \ n$:

$$O(N^2)$$

Przechowywanie wierzchołków:

$$O(N)$$

Całkowita złożoność:

$$O(N^2) + O(N) = O(N^2 + N) = O(N^2)$$

## 6. Użyte struktury

- **Klasa Vertex** - obiektom tej klasy przypisywane są indywidualne numery (numery wierzchołków)

```
class Vertex {
private:
    int number;
public:
    int weight;
    std::string label;
    Vertex(int n) {
        number = n;
    }
    int Number() const {
        return number;
    }
};
```

- **Klasa Edge** - obiekty tej klasy tworzone są z dwoma wierzchołkami V0 i V1

```
class Edge {
protected:
    Vertex* v0;
    Vertex* v1;

public:
    int weight;
    std::string label;
    Edge(Vertex* V0, Vertex* V1) {
        v0 = V0;
        v1 = V1;
    }
    Vertex* V0() {
        return v0;
    }
    Vertex* V1() {
        return v1;
    }
    Vertex* Mate(Vertex* v) {
        return v == v0 ? v1 : v0;
    }
};
```

- **Klasa Iterator** - służąca do przemieszczania się po grafie

Posiada ona:

- IsDone() - zwraca prawdę jeśli iterator dojdzie do końca
- Operator * - zwraca wartość z iteratora
- Operator ++ - przesuwa iterator do przodu

```
template <typename T>
class Iterator
{
public:
    Iterator(){;}
    virtual ~Iterator(){}
    virtual bool IsDone() const = 0;
    virtual T& operator*() = 0;
    virtual void operator++() = 0;
};
```

- **GraphAsMatrix** - zawiera całą implementację grafu jako macierzy sąsiedztwa

Posiada ona:

- Wektor składający się ze wszystkich wierzchołków
- Macierz sąsiedztwa
- Flagę IsDirected
- Ilość wierzchołków
- Ilość krawędzi

```
class GraphAsMatrix
{
private:
    std::vector<Vertex*> vertices;
    std::vector<std::vector<Edge*> > adjacencyMatrix;
    bool isDirected;
    int numberOfVertices = 9;
    int numberOfEdges = 0;
```

# 7. Instrukcja uruchomienia

Aby uruchomić należy wpisać:

c++ main.cpp

Aby uruchomić, należy wpisać:

./a.out