

# Wykład 13: Algorytmy i złożoność obliczeniowa.

dr inż. Andrzej Stafiniak

*Wrocław 2023*



HR EXCELLENCE IN RESEARCH



Politechnika Wroclawska

# Algorytmy

➤ Czym jest **algorytm** ?

# Algorytmy

- **Algorytm** – „... skończony ciąg jasno zdefiniowanych czynności koniecznych do wykonania pewnego rodzaju zadań...” lub „...sposób postępowania prowadzący do rozwiązania problemu...” - [www.wikipedia.org](http://www.wikipedia.org)

# Cechy i elementy algorytmu

- **Skończoność i poprawność** – algorytm ma ostatecznie poprawnie rozwiązać postawiony problem.
- **Dane wejściowe** – zespół danych na których algorytm będzie operował.
- **Dane wyjściowe** – zespół danych, które zostaną przekazane jako rozwiązanie.
- **Kroki algorytmu** – zdefiniowane kolejne czynności prowadzące do rozwiązania problemu
- **Złożoność obliczeniowa** - zapotrzebowanie algorytmu na zasoby komputerowe takie jak np. czas lub pamięć potrzebne do realizacji rozwiązania zadania.

# Metody opisu algorytmów

➤ Jak zapisać **algorytm**?

# Metody opisu algorytmów

- Jak zapisać **algorytm**?
  - Opis słowny
  - Lista kroków
  - Schemat blokowy
  - Tablice decyzyjne
  - Drzewo algorytmu
  - Pseudokod
  - Język programowania

# Metody opisu algorytmów

## Algorytm Euklidesa

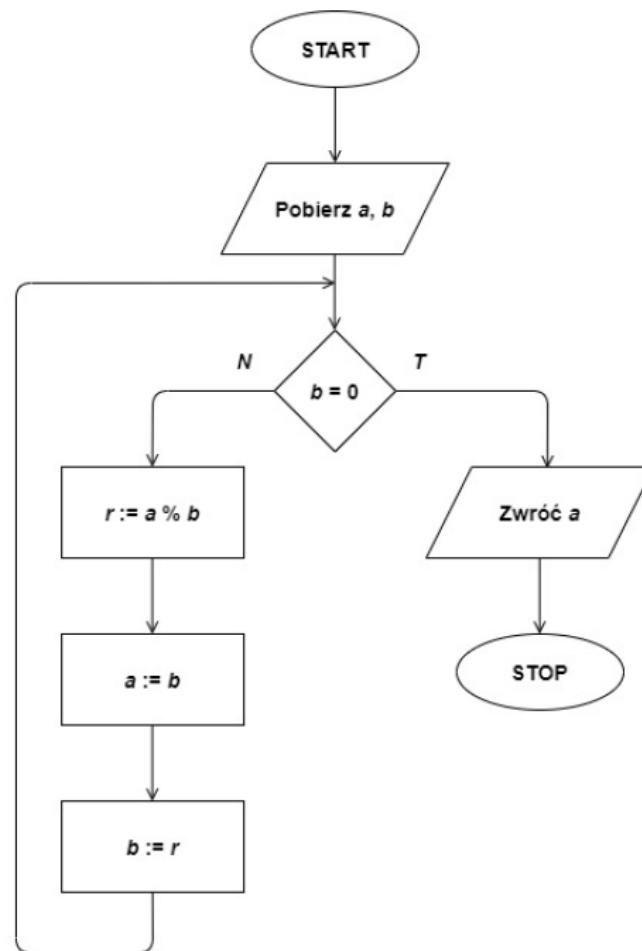
### Opis słowny:

1. jeżeli  $b$  jest równe 0, zwróć  $a$ , w przeciwnym razie:
2. oblicz  $r$  jako resztę z dzielenia  $a$  przez  $b$
3. podstaw  $a \leftarrow b$  oraz  $b \leftarrow r$
4. przejdź do punktu 1.

### Język programowania:

```
int gcd(int a, int b) {  
    return (b == 0) ? a : gcd(b, a % b);  
}
```

### Schemat blokowy:



# Złożoność obliczeniowa algorytmu

Różne sposoby realizacji algorytmu (różne definicje algorytmów), będą różnić się między sobą ponadto czasem wykonania programu, ilością wykorzystywanych zasobów (pamięci) oraz np. czytelnością kodu.

**Złożoność obliczeniowa** algorytmu (*computational complexity*) to zagadnienie opisujące pewną „**jakość**” algorytmu – jaki jest „wydajny”, liczba wykonanych operacji lub rozmiar wykorzystanych zasobów pamięci w zależności od danych wejściowych.

Czyli **złożoność obliczeniowa** może być przedstawiona w postaci np. funkcji rozmiaru danych wejściowych  $f(n)$  gdzie  $n$  – rozmiar danych wejściowych (np. w bitach).



# Złożoność obliczeniowa algorytmu

**Złożoność obliczeniowa** algorytmu (*computational complexity*) dzielimy na:

- **Złożoność pamięciową** (space complexity) – wyznaczana na podstawie ilości potrzebnej pamięci w celu rozwiązania problemu, dla  $n$  danych wejściowych. (wywołania rekurencyjne funkcji generują duże koszty pamięciowe)
- **Złożoność czasową** (time complexity) – jest to zapotrzebowanie algorytmu na czas, który liczymy za pomocą ilości operacji dominujących potrzebnych do rozwiązania problemu, dla  $n$  danych wejściowych.

Dokładne wyznaczenie **złożoności obliczeniowej** danego algorytmu jest dość trudne i czasochłonne. Dlatego wymagane są szacowania oraz założenia.

# Szacowanie złożoność algorytmów

Szacowanie:

- **od dołu** – zakładamy, że nasz algorytm ma złożoność nie mniejszą niż pewne klasa funkcji  $\Omega(f(n))$ , (ile zasobów zużyje dowolny algorytm rozwiązujący określony problem w najgorszym przypadku),
- **od góry** – zakładamy, że nasz algorytm ma złożoność nie większa niż pewna klasa funkcji  $O(f(n))$ , (ile zasobów w najgorszym przypadku wystarczy do przeprowadzenia obliczeń),
- **asymptotycznie** – szacujemy dokładnie poprzez klasę funkcji  $\Theta(f(n))$ .

Klasą funkcji będziemy nazywać pewne grupy funkcji, które będą w podobny sposób zależne od ilości przyjmowanych danych. Klasy te są przedstawiane w tz. notacjach omega  $\Omega(f(n))$ , duże O  $O(f(n))$ , teta  $\Theta(f(n))$ .

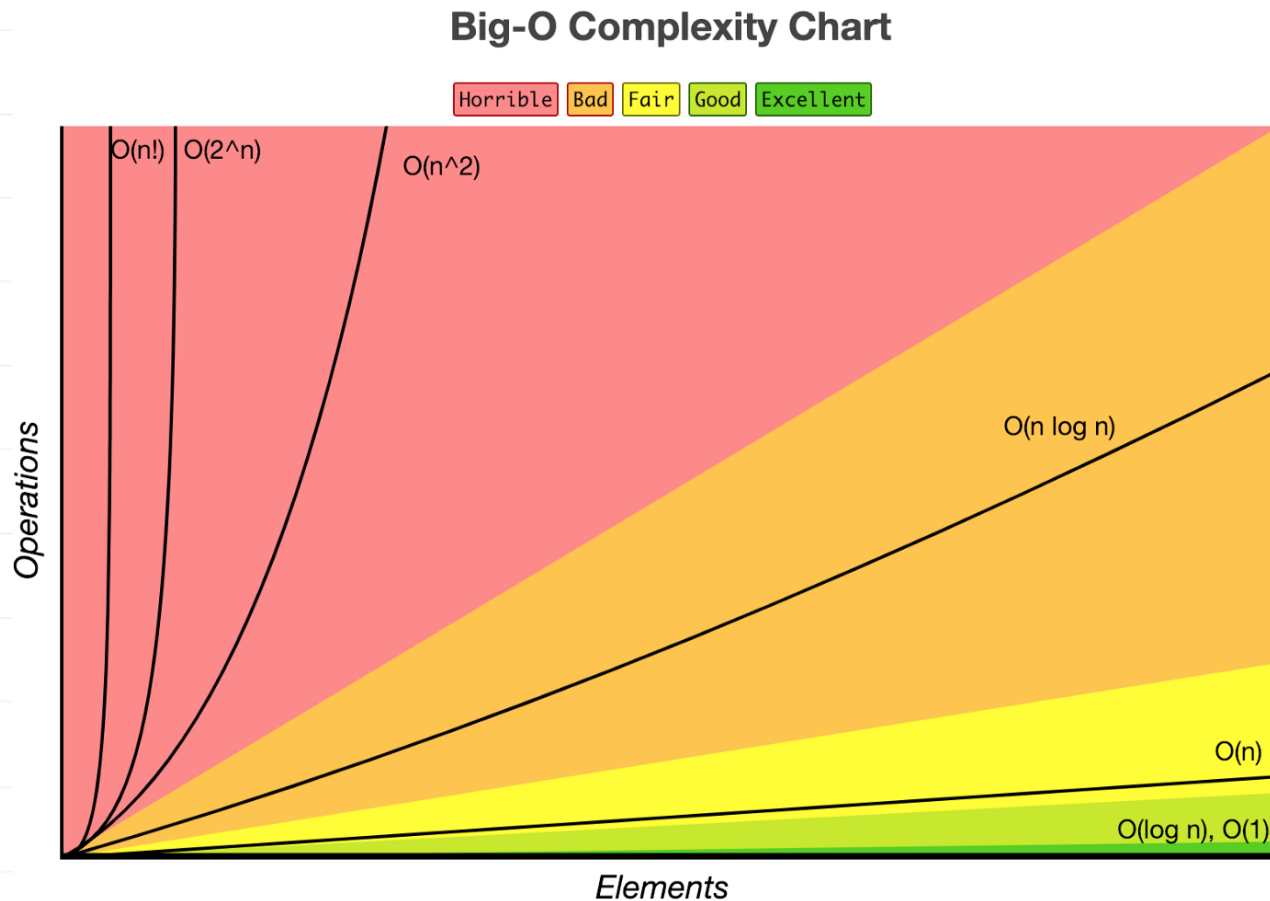
# Szacowanie złożoność algorytmów

Szacowanie:

- **od dołu** – zakładamy, że nasz algorytm ma złożoność nie mniejszą niż pewne klasa funkcji  $\Omega(f(n))$ , (ile zasobów zużyje dowolny algorytm rozwiązujący określony problem w najgorszym przypadku),
- **od góry** – zakładamy, że nasz algorytm ma złożoność nie większa niż pewna klasa funkcji  $O(f(n))$ , (ile zasobów z najgorszym przypadkiem wystarczy do przeprowadzenia obliczeń),
- **asymptotycznie** – szacujemy dokładnie poprzez klasę funkcji  $\Theta(f(n))$ .

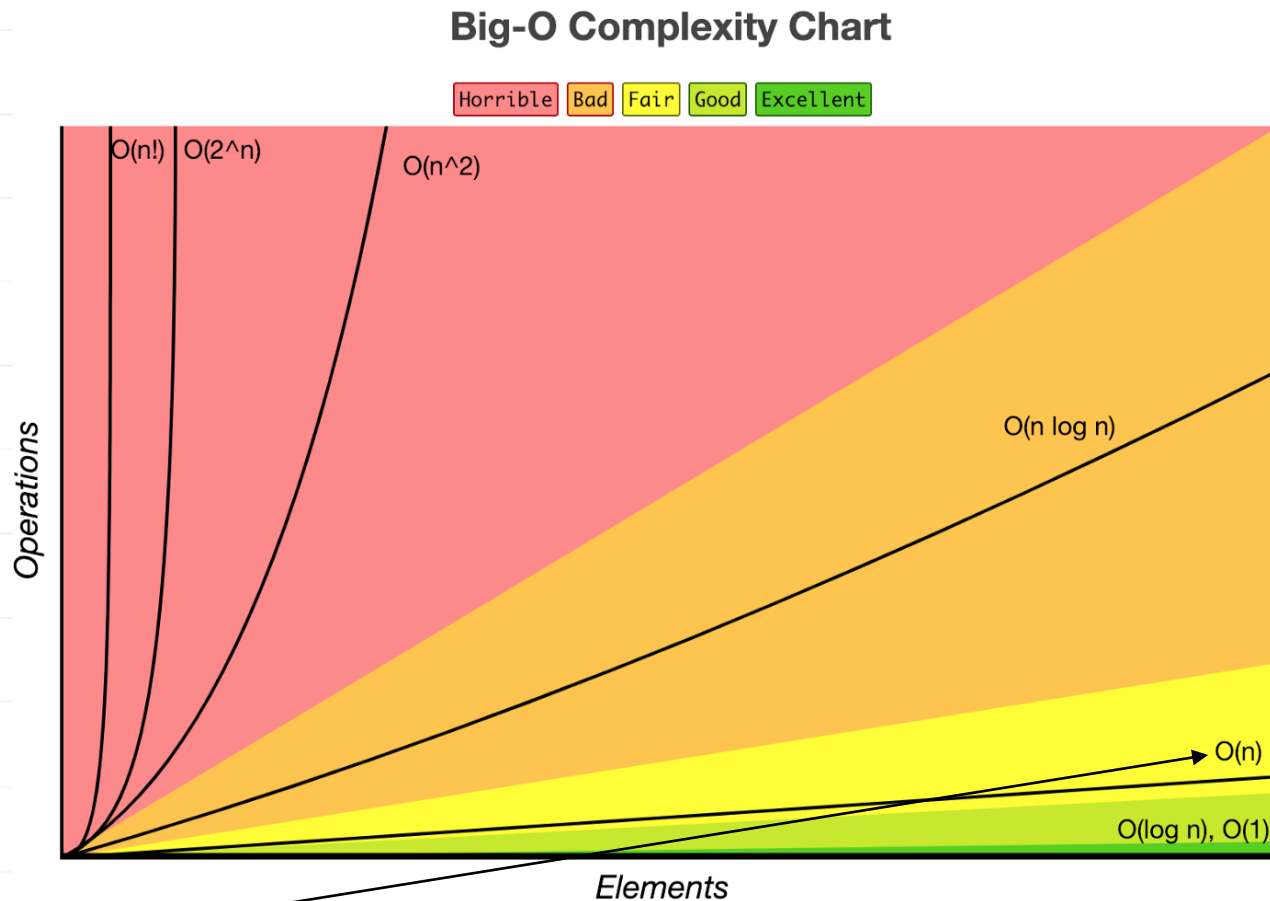
Najpopularniejszą metodą opisu złożoności obliczeniowej jest notacja **dużego O**, przedstawia pesymistyczne oszacowanie wartości **czasowej złożoności obliczeniowej** algorytmów.

# Notacja $O()$ i klasy złożoności obliczeniowej



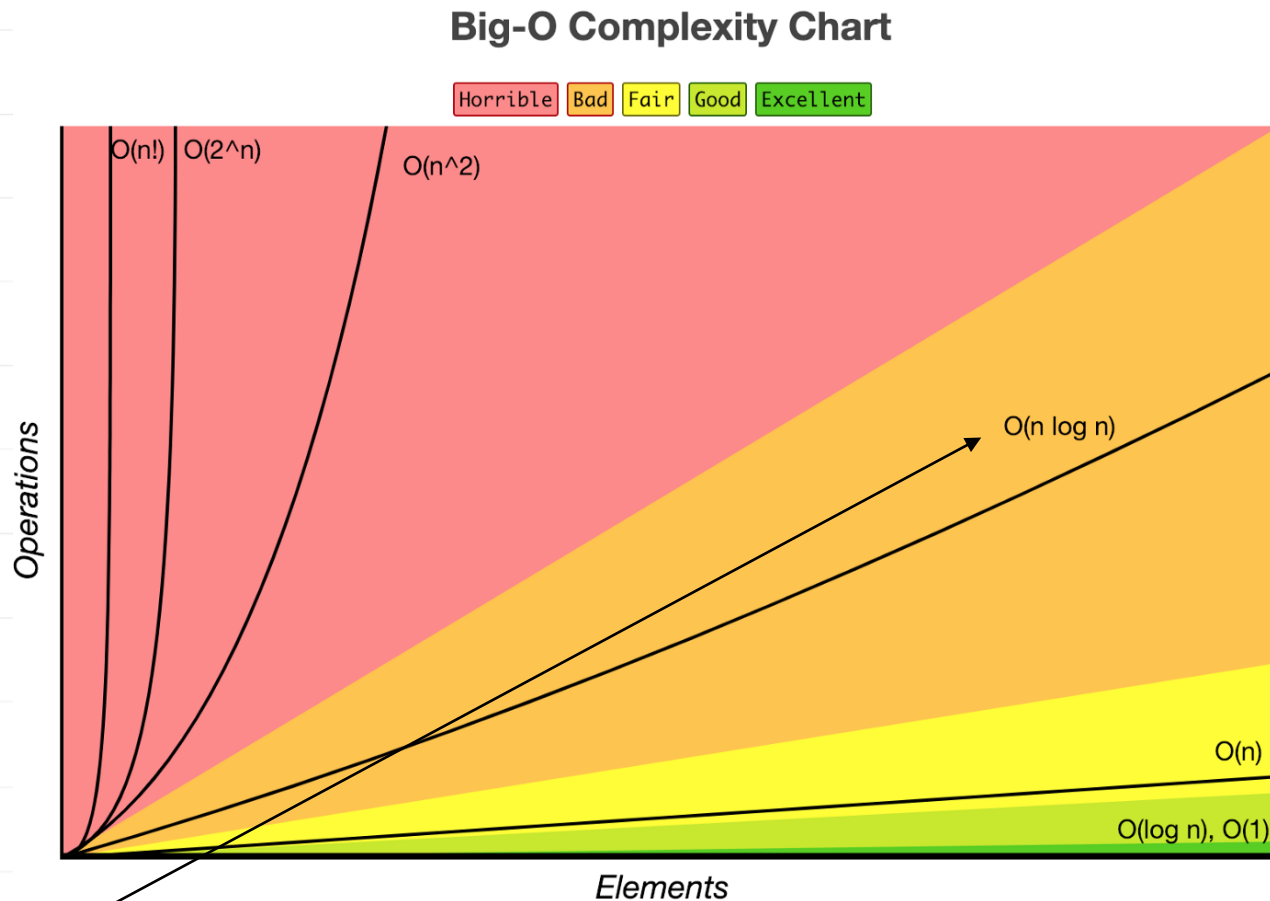
**$O(1)$  – złożoność stała** - czas wykonywania algorytmu nie zależy od ilości danych wejściowych, np. funkcje które mają tylko zwrócić pewną wartość.

# Notacja $O()$ i klasy złożoności obliczeniowej



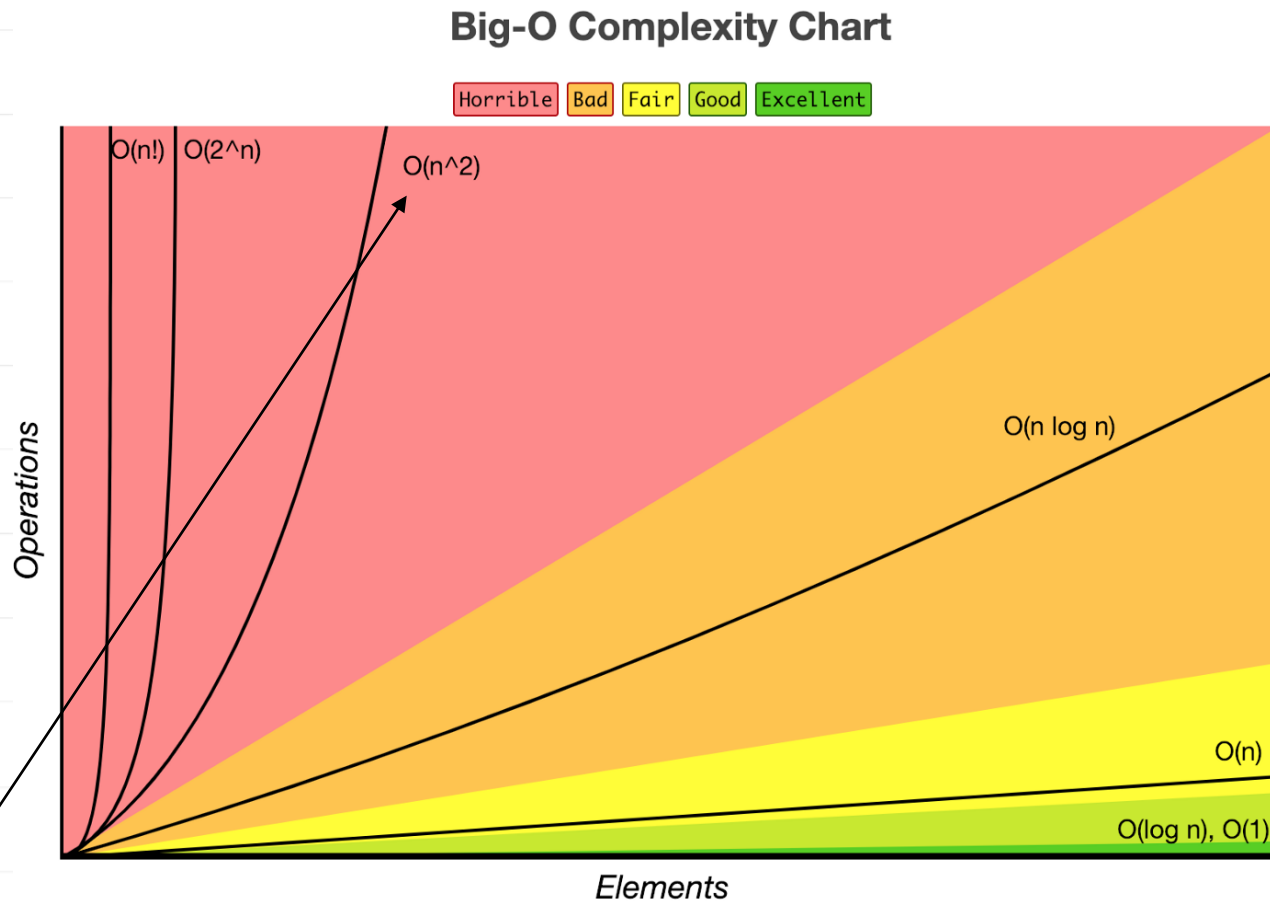
**$O(n)$  – złożoność liniowa** – czas wykonywania algorytmu jest wprost proporcjonalny do ilości danych wejściowych, np. funkcje, które wyszukują max/min w nieuporządkowanych ciągach

# Notacja $O()$ i klasy złożoności obliczeniowej



$O(n \log n)$  – złożoność liniowo-logarytmiczna – sortowanie merge sort

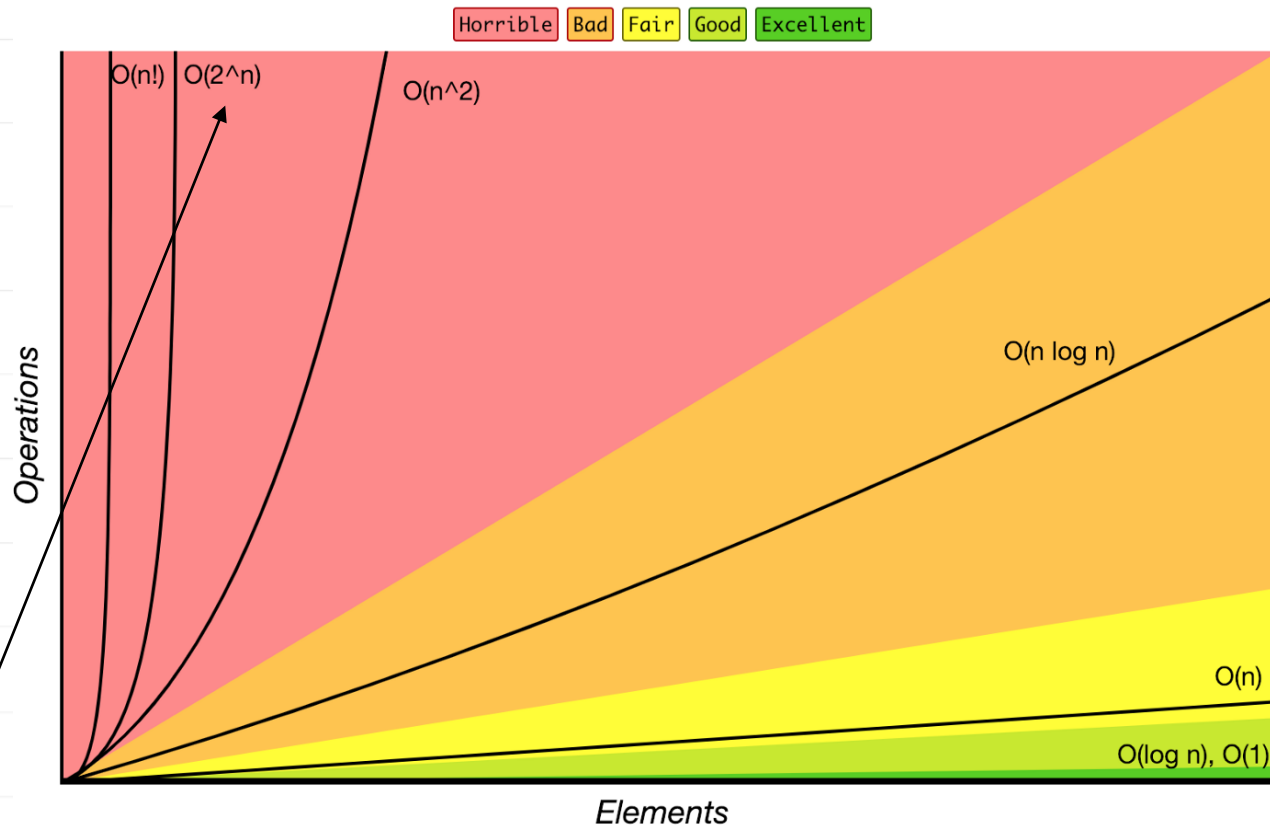
# Notacja $O()$ i klasy złożoności obliczeniowej



**$O(n^2)$  – złożoność kwadratowa** – czas wykonywania algorytmu jest uzależniony od kwadratu danych wejściowych, np. sortowanie bąbelkowe

# Notacja $O()$ i klasy złożoności obliczeniowej

Big-O Complexity Chart

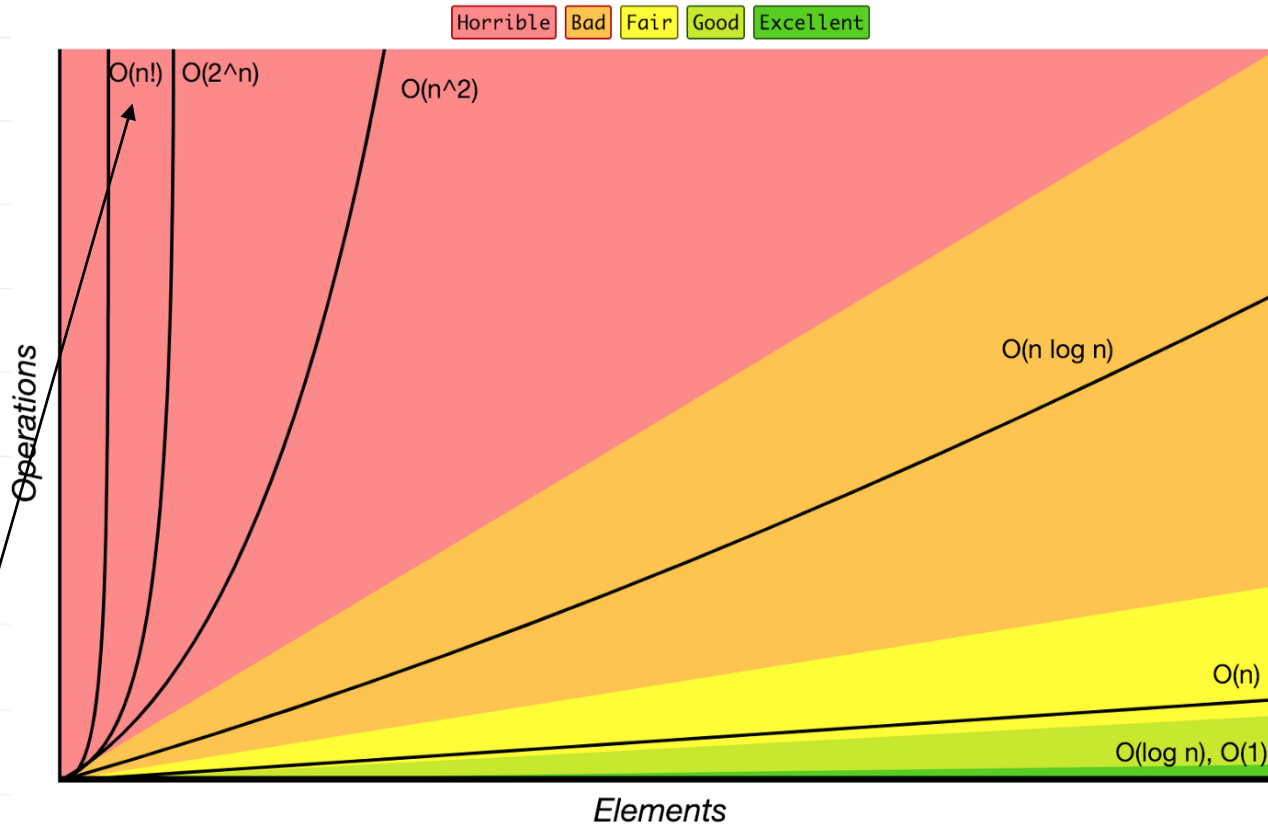


$O(2^n)$  – złożoność wykładnicza - np. ciąg Fibbonaciego



# Notacja $O()$ i klasy złożoności obliczeniowej

Big-O Complexity Chart



$O(n!)$  – złożoność typu silnia – algorytmy grafowe

# Algorytmy

## Algorytm Euklidesa - złożoność $O(\log n)$

```
int gcd (int a, int b) {  
    int tmp;  
    while (b != 0) {  
        tmp = a % b;  
        a = b;  
        b = tmp;  
    }  
    return a;  
}
```

```
int gcd (int a, int b) {  
    int tmp;  
    while (a != b) {  
        if (a > b)  
            a -= b;  
        else  
            b -= a;  
    }  
    return a;  
}
```

```
int gcd (int a, int b) {  
    return b ? gcd(b, a % b) : a;  
}
```

```
int gcd (int a, int b) {  
    if (b == 0)  
        return a;  
    return gcd(b, a % b);  
}
```

# Algorytmy

Algorytm przeszukiwania, wartość max/min - złożoność  $O(n)$

```
int search (int tab[], int size, int x) {  
    int count = 0;  
    for(int i=0; i<size; ++i){  
        if(tab[i]==x)  
            count++;  
    }  
    return count;  
}
```

```
int max (int tab[], int size) {  
    int temp = tab[0];  
    for(int i=1; i<size; ++i){  
        if(tab[i]>temp)  
            temp = tab[i];  
    }  
    return temp;  
}
```

# Algorytmy

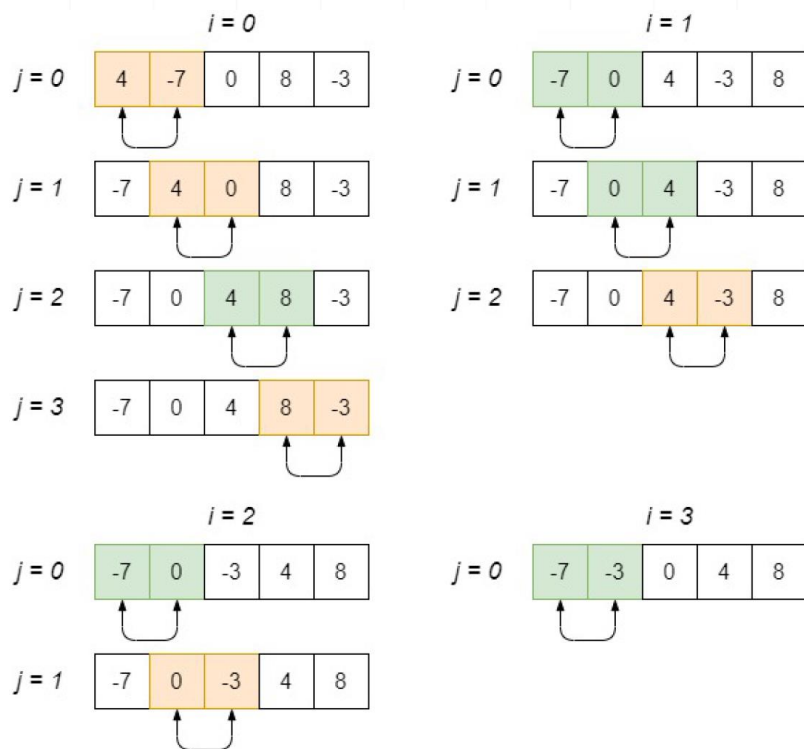
Algorytm **sortowania bąbelkowego** (bubble sort) - złożoność  $O(n^2)$

```
int * bubbleSort (int * ptr, int n){
    int buff =0;
    for(int i=0; i<n; ++i){
        for (int j=i+1; j<n; ++j){
            if (*(ptr+j)<*(ptr+i)){
                buff = *(ptr + i);
                *(ptr+i)=*(ptr+j);
                *(ptr+j) = buff;
            }
        }
    }
    return ptr;
}
```

```
int * bubbleSort (int * ptr, int n){
    int buff =0;
    for(int i=0; i<n; ++i){
        for (int j=0; j<n-i-1; ++j){
            if (ptr[j] > ptr[j+1]){
                buff = ptr[j];
                ptr[j] = ptr[j + 1];
                ptr[j+1] = buff;
            }
        }
    }
    return ptr;
}
```

# Algorytmy

Algorytm sortowania bąbelkowego (bubble sort) - złożoność  $O(n^2)$



```
int * bubbleSort (int * ptr, int n){
    int buff =0;
    for(int i=0; i<n; ++i){
        for (int j=0; j<(n-i-1); ++j){
            if (ptr[j] > ptr[j+1]){
                buff = ptr[j];
                ptr[j] = ptr[j + 1];
                ptr[j+1] = buff;
            }
        }
    }
    return ptr;
}
```

# Algorytmy

Algorytm **sortowania przez scalanie** (merge sort) - złożoność  $O(n \log n)$

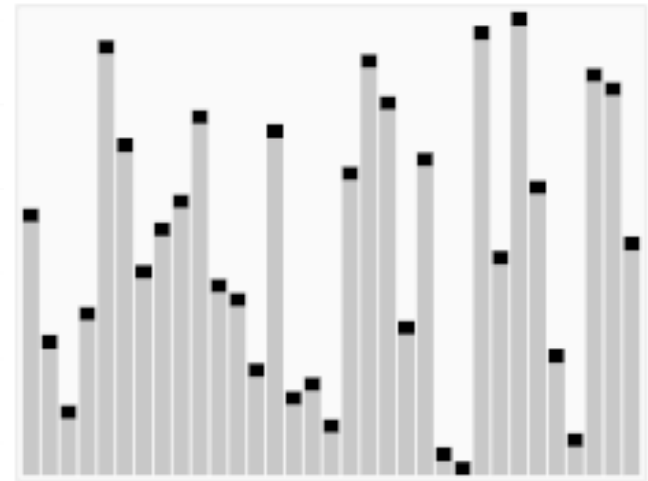
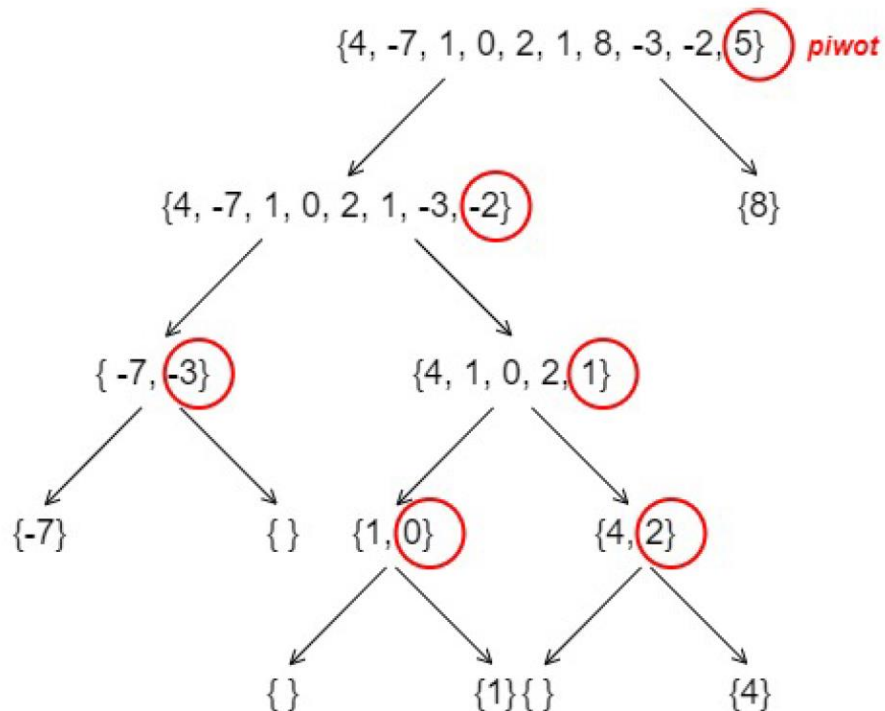
Algorytm wykorzystuje metodę **dziel i zwyciężaj** (metoda projektowania algorytmów, podział złożonego problemu na mniejsze łatwe do rozwiązania problemy, następnie łączenie otrzymanych wyników w ostateczne rozwiązanie)

6 5 3 1 8 7 2 4

# Algorytmy

Algorytm **szybkiego sortowania** (quick sort) - złożoność  $O(n \log n)$  (pesymistycznie  $n^2$ )

Algorytm wykorzystuje metodę **dziel i zwyciężaj**.

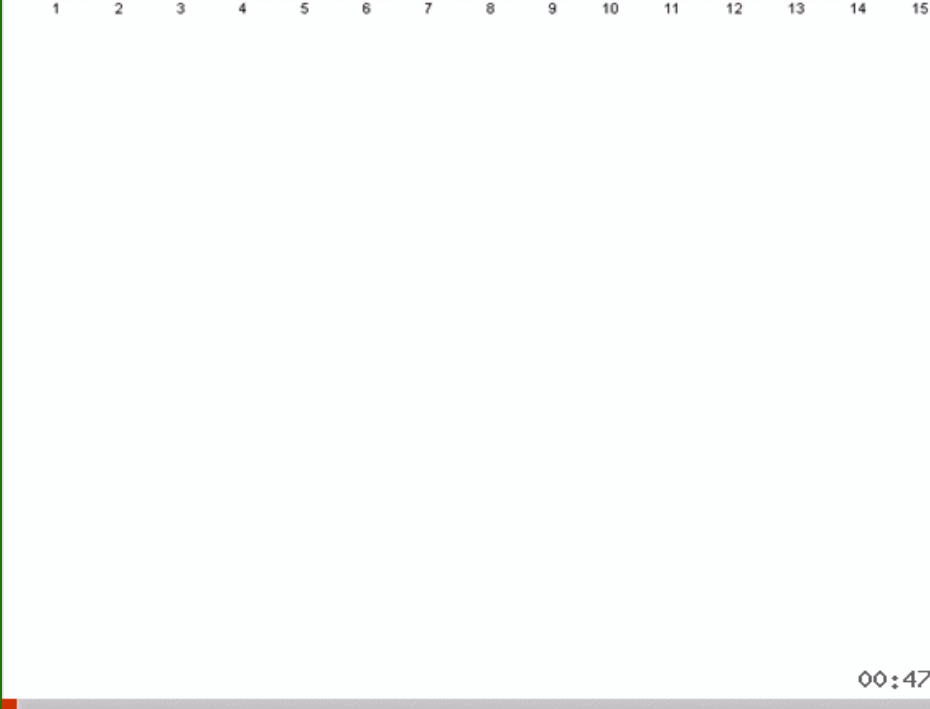


# Algorytmy

Algorytm **sortowania kubełkowego** (bucket sort) - złożoność  $O(n)$  dla w miarę jednostajnie rozłożonych danych oraz  $O(n^2)$  dla pesymistycznego przypadku.

Trzeba znać zakres zbioru

65	62	71	45	32	45	97	30	29	89	95	10	86	39	65
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



00:47



# Algorytmy

## Algorytm wyznaczenia elementu ciągu Fibonacciego

Zapis rekurencyjny - złożoność  $O(2^n)$

```
int fibRecur(int n){  
    if(n < 2)  
        return n;  
    else  
        return fibRecur(n-2) +  
                fibRecur(n-1);  
}
```



Krótki, szybki zapis, **ale !?**

Zapis iteracyjny - złożoność  $O(n)$

```
int fibIter(int n){  
    if(n < 2)  
        return n;  
    else{  
        int tmp;  
        int current = 1;  
        int last = 0;  
        for(i = 2, i <= n, i++){  
            tmp = current;  
            current = current + last;  
            last = tmp;  
        }  
        return current;  
    }  
}
```