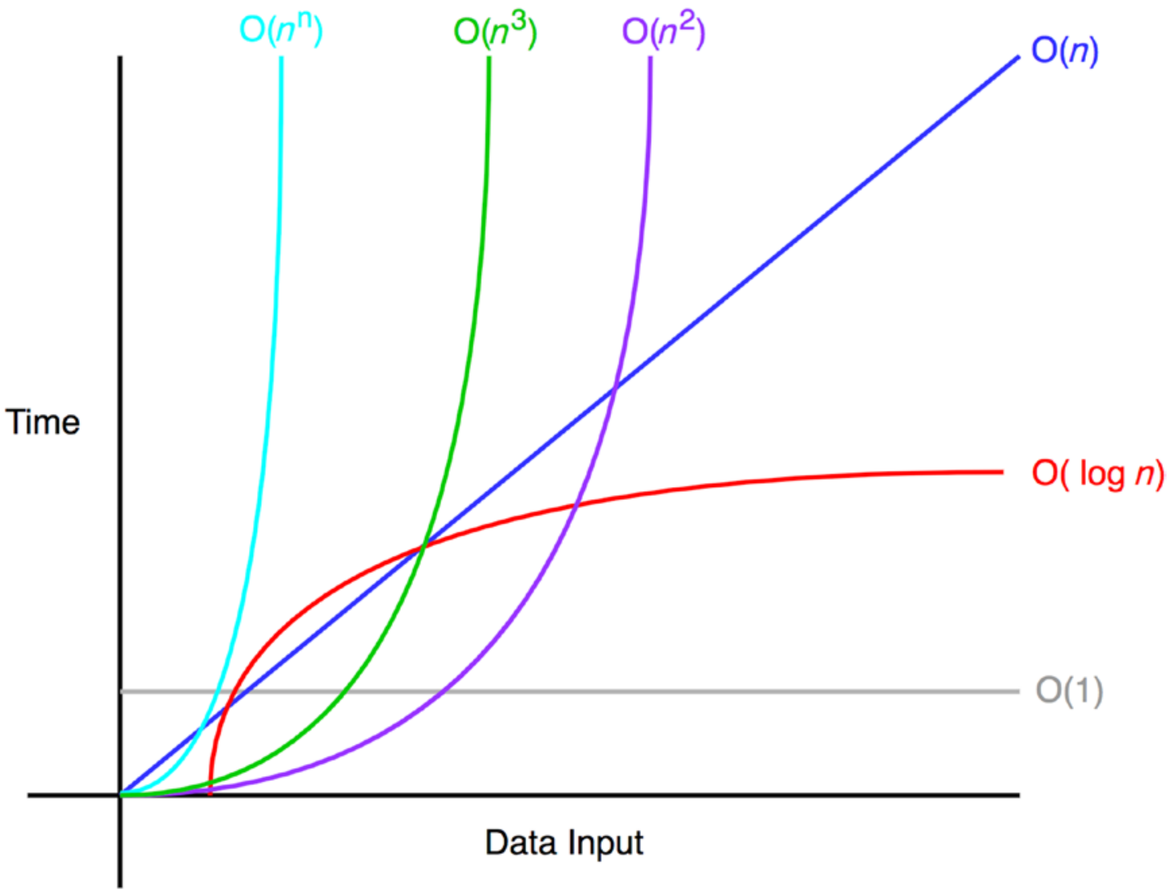
Notacja Big-O – analizowanie skuteczności algorytmów (analiza złożoności czasowej i algorytmicznej przestrzeni).

Notacja oblicza złożoność algorytmu w najgorszym przypadku, czyli co się stanie, gdy n (liczba wejść) zbliży się do nieskończoności.



O(1) nie zmienia się w odniesieniu do przestrzeni wejściowej (np. wyszukiwanie indeksu w tablicy poprzez indeks).

O(n) dotyczy algorytmów, które muszą wykonać n operacji, np. zwracanie liczb od 0 do n-1 :

```
for(let i=0; i<n; i++){
    console.log(i);
}
```

O(*n*2) – dotyczy funkcji wykonującej się w czasie do kwadratu, czyli:

```
for(let i=0; i<n; i++){
   console.log( i );
 for (let j=0; j<n; j++){
        console.log( j );
 }
```

Ostatecznie przykładem algorytmicznej złożoności czasowej jest zwracanie elementów, które są potęgą 2 między 2 a n, np.:

2,4,8,16,32,64

ZASADA WSPÓŁCZYNNIKA: ” Pozbądź się stałych ”

Ignorujemy wszelkie stałe niezwiązane z rozmiarem wejściowym (non-input-size-related). Współczynniki są nieistotne przy dużych rozmiarach wejściowych.

```
function a(n){
        let count =0;
        for (let i=0; i<n; i++){
                count += 1;
        }
        return count;
}
```

Ten blok kodu zawiera f(n) = n, więc jest to funkcja O(n) w złożoności czasowej:

```
function a(n){
        let count =0;
        for (let i=0; i<5*n; i++){
        count += 1;
         }
return count;
 }
```

Ten blok kodu zawiera f(n)=5n. Oby dwa przykłady mają notację O(n).

Dzieje się tak, ponieważ jeśli n jest bliskie nieskończoności pozostałe operacje wcześniej są bez znaczenia, dlatego stałe są pomijane w notacji.

Poniższy kod demonstruje inną funkcję z czasem liniowym, ale z dodatkową operacją:

```
function a(n){
        let count =0;
         for (let i=0; i<n; i++){
        count+=1;
        }
        count+=3;
        return count;
}
```

Na końcu bloku kodu jest f(n) = n + 1. Tu jest +1 z ostatniej operacji (Count += 3). Wciąż jest to notacja O(n). Dzieje się tak, ponieważ operacja nr.1 nie jest zależna od wejścia n. Będzie to nieistotne, gdy będzie się zbliżać do nieskończoności.

REGUŁA SUMY: ” Dodaj notacje Big-O w górę ”

Zasada sumy oznacza, że algorytm główny posiada w sobie dwa algorytmy. Notacja Big-O głównego algorytmu jest sumą notacji tych wewnątrz niego.

Przy tym należy pamiętać o poprzedniej regule współczynnika.

Poniższy blok kodu ukazuje funkcję z dwiema pętlami, których czas złożoności należy rozpatrywać niezależnie, a następnie zsumować:

```
function a(n){
        let count =0;
        for (let i=0; i<n; i++){
                count+=1;
        }
        for (let i=0; i<5*n; i++){
        count+=1;
        }
        return count;
}
```

W tym przykładzie wiersz nr.4 ma f(n) = n, a wiersz nr.7 ma f(n) = 5n. Daje to 6n. Jednak przy zastosowaniu zasady współczynnika ostateczny wynik to O(n) = n.

Reguła Produktu: ” Mnożenie notacji Big-O ”

Określa w jaki sposób można pomnożyć notacje Big-O.

Poniższy przykład demonstruje funkcję z dwiema zagnieżdżonymi pętlami, gdzie ma zastosowanie reguła produktu:

```
1    function (n){
2        let count =0;
3        for (let i=0;i<n;i++){
4            count+=1;
5            for (let i=0; i<5*n; i++){
6                count+=1;
7            }
8        }
9        return count;
10   }
```

W tym przykładzie f(n) = 5n * n, ponieważ linia nr.7 jest wykonywana 5n razy, co daje łącznie n iteracji. Dlatego daje to łącznie 5*n*2 operacji.

Reguła Wielomianu: ” Big-O do potęgi k ”

Reguła wielomianu mówi, że złożoność wielomianu w czasie ma notację Big-O tego samego wielomianu. Matematycznie wygląda to w następujący sposób:

Jeśli f(*n*) jest wielomianem stopnia k, to f(*n*) jest O(*n*k).
Poniższy blok kodu ma tylko jedną pętlę for o złożoności czasu kwadratowego

```
function a(n){
    let count =0;
    for (let i=0; i<n*n; i++){
    count+=1;
    }
    return count;
}
```

Blok kodu określa f(n) = n ^ 2, ponieważ linia nr.4 wykonuje n * n iteracji.

REKURENCJA

Technika programowania umożliwiająca wywołanie funkcji przez samą siebie. Metoda rozwiązywania problemów. Która polega na rozbijaniu dużego problemu na coraz to mniejsze części (podproblemy). Robimy to aż do momentu, w którym otrzymamy problem tak prosty, że jesteśmy sobie w stanie z nim poradzić.

Przypadek rekurencyjny – w nim funkcja wywołuje samą siebie z argumentem, który upraszcza problem i przybliża nas do rozwiązania.

Przypadek bazowy – problem staje się błahy i jesteśmy w stanie uzyskać jego rozwiązanie. Kończymy proces wywołania funkcji przez samą siebie.

CEL: dobrze nadają się przy przeszukiwaniu u sortowaniu stuktur danych.

NIEBEZPIECZEŃSTWO: Błędnie zdefiniowany przypadek bazowy – funkcja będzie się chciała wykonywać w nieskończoność, aż do momentu, gdy zabraknie dostępnej pamięci (stack overflow).

PRZYKŁAD: Przypuszczenie Collatza

Przypuszczenie dotyczące dodatnich liczb całkowitych i doprowadzenia ich wartości do 1.

- Jeżeli n to 1, zatrzymaj się
- Jeżeli n jest parzyste, powtórz działanie dla n/2
- Jeżeli n jest nieparzyste, powtórz działanie dla 3n + 1

Algorytm zwraca liczbę kroków potrzebnych do doprowadzenia liczby do 1.

```
let coll = n => {
  if (n === 1) {
    return 0;
  }

  return n % 2 === 0 ? coll(n / 2) + 1 : coll(3 * n + 1) + 1;
}

console.log(`${coll(21)}`);

```