Algorytmy i struktury danych

Rekurencja

Aleksander Lamża
ZKSB · Instytut Informatyki
Uniwersytet Śląski w Katowicach

aleksander.lamza@us.edu.pl

Zawartość

- Rzut oka na podejście iteracyjne
- Pętla bez pętli
- Jak działa rekurencja?
- Kilka przykładów algorytmów rekurencyjnych

Rzut oka na podejście iteracyjne

Jeżeli mielibyście wielokrotnie wykonać pewne operacje, zastosujecie z pewnością **podejście iteracyjne**, czyli...

PĘTLE

```
while (powtorz) {
   cout << "Robię coś bardzo ważnego" << endl;
   ...
}
```

```
for (int i=0; i<100000; ++i) {
   cout << "Liczę... " << i << endl;
   ...
}
```

Pętla bez pętli

Załóżmy, że chcemy napisać kod odliczający od jakiejś wartości do zera.

Narzuca się zastosowanie pętli `for`:

```
for (int i=3; i>=0; --i) {
   cout << i << endl;
}
```

A czy da się to zrobić bez użycia jakiejkolwiek pętli?

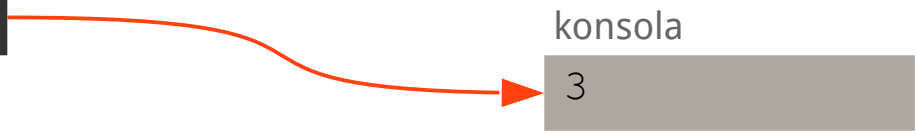
Pętla bez pętli

Możemy zacząć od zdefiniowania funkcji, która wyświetla przekazaną jej wartość:

```
void odliczaj(int x) {
   cout << x << endl;
}
```

Oto wynik jej działania:

```
odliczaj(3);
```



Nic w tym odkrywczego...

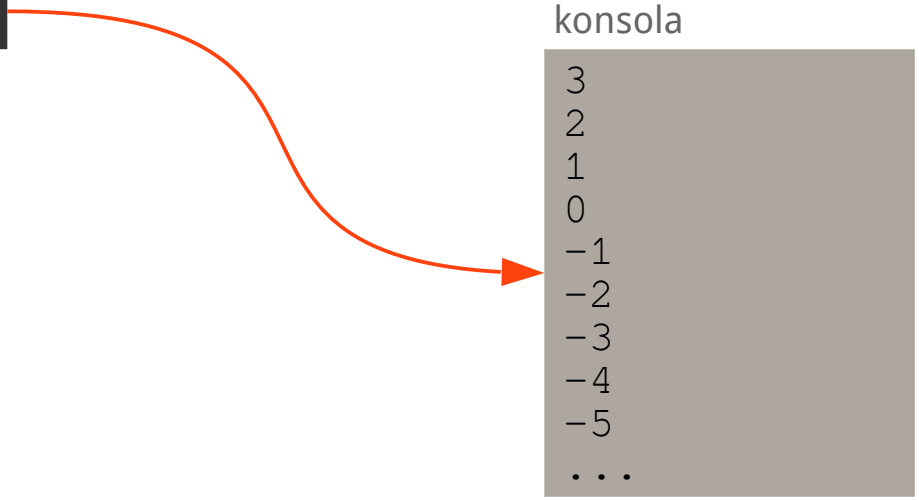
Pętla bez pętli

A co by się stało, gdyby funkcja `odliczaj()`, poza wyświetlaniem przekazanej wartości, **wywoływała samą siebie** z wartością o jeden mniejszą?

```
void odliczaj(int x) {
   cout << x << endl;
   odliczaj(x-1);
}
```

Po wywołaniu funkcji w tej postaci konsola zostanie „zalana” zmniejszającymi się wartościami:

```
odliczaj(3);
```



Trzeba to jakoś zatrzymać!
W tej chwili trzeba użyć Ctrl+C, ale...

Pętla bez pętli

Chcemy, by odliczanie zatrzymało się na zerze, więc...

...powinniśmy dopuszczać do wywołania `odliczaj(x-1)` tylko wtedy, gdy `x` jest większe od zera:

```
void odliczaj(int x) {
   cout << x << endl;
   if (x > 0) {
      odliczaj(x-1);
   }
}
```

Oto wynik:

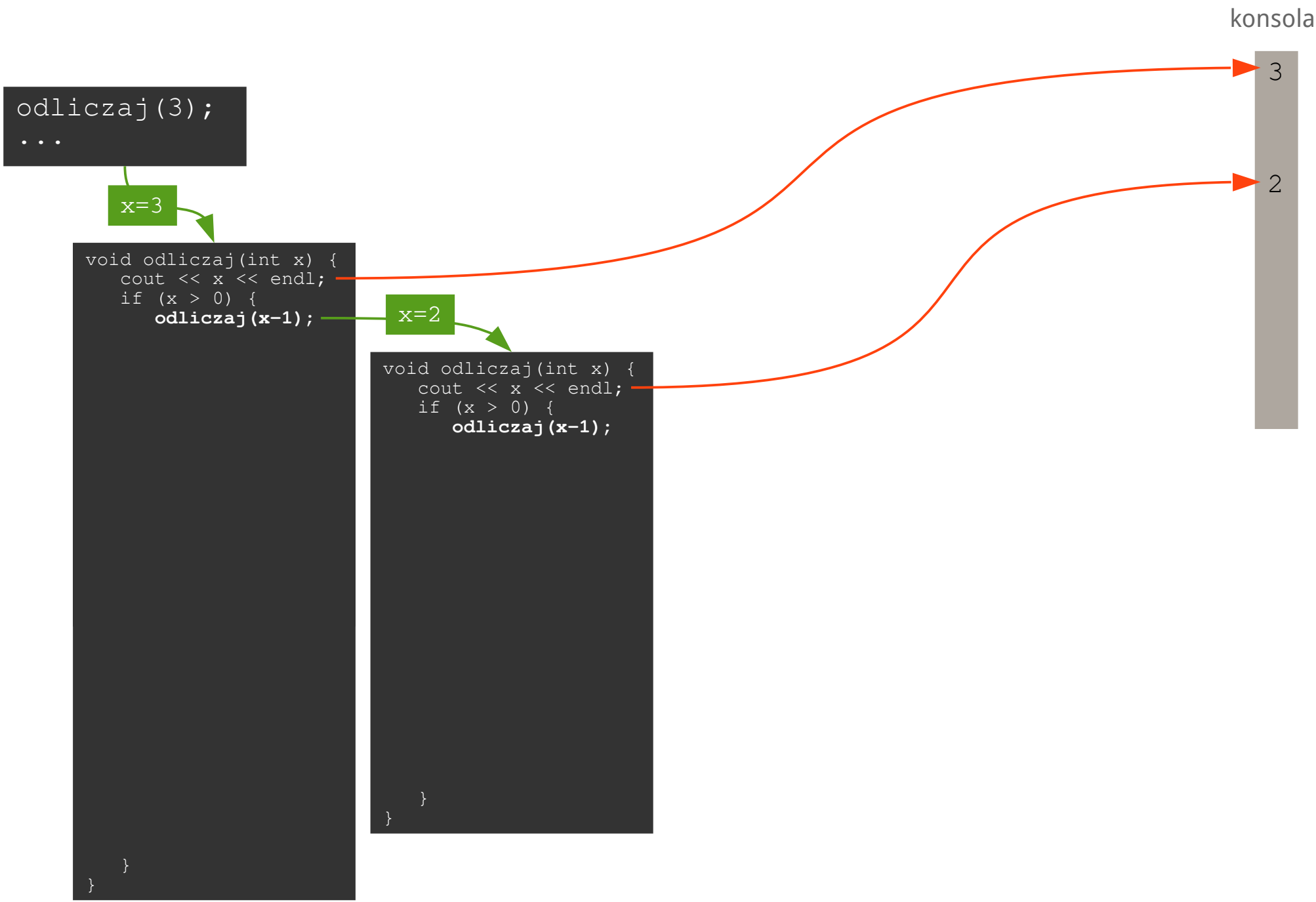
```
odliczaj(3);
```

To, co tu widzicie, to najprostszy przykład **rekurencji**.

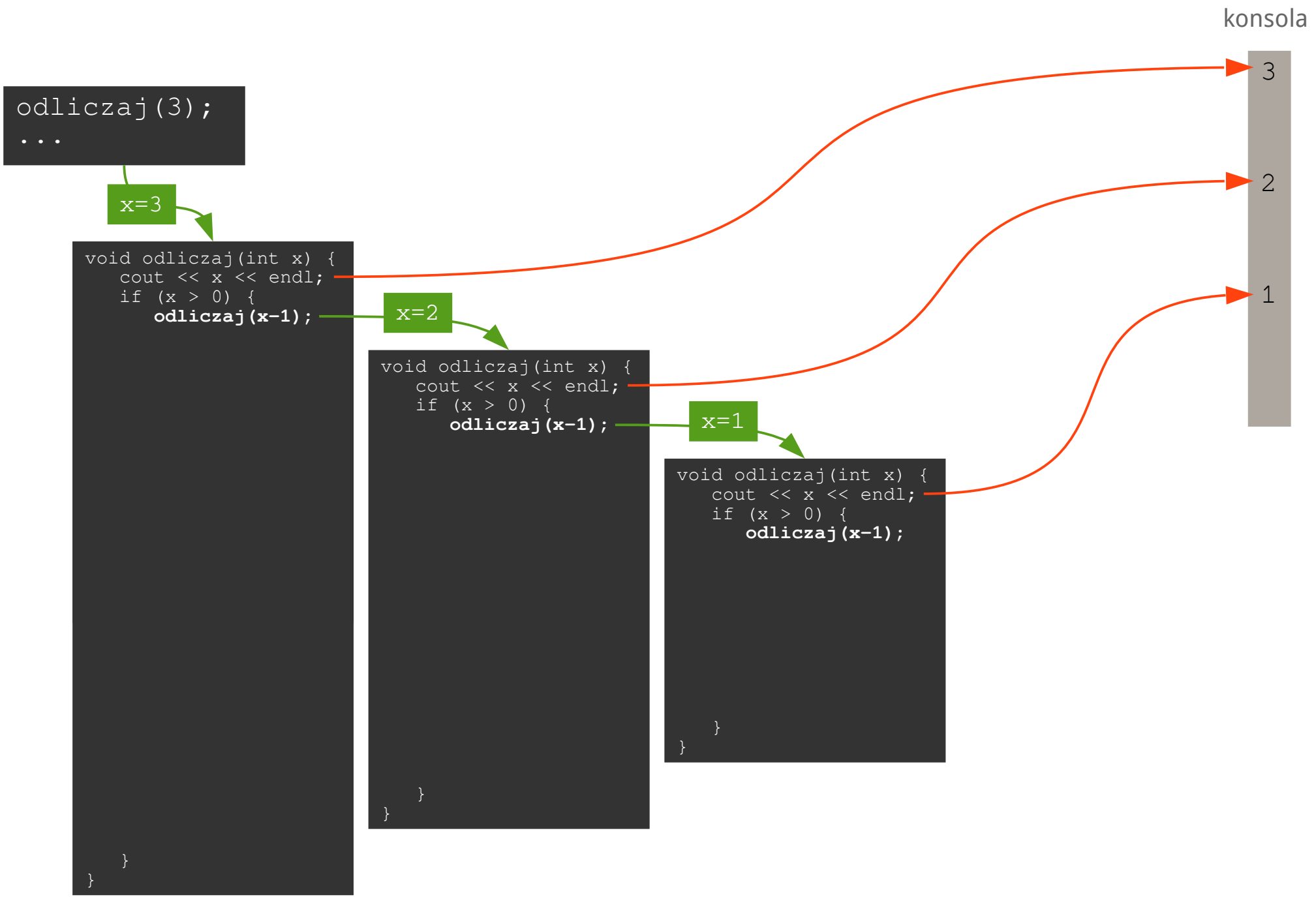
Przeanalizujmy, krok po kroku, działanie tego kodu.

Jak działa rekurencja?

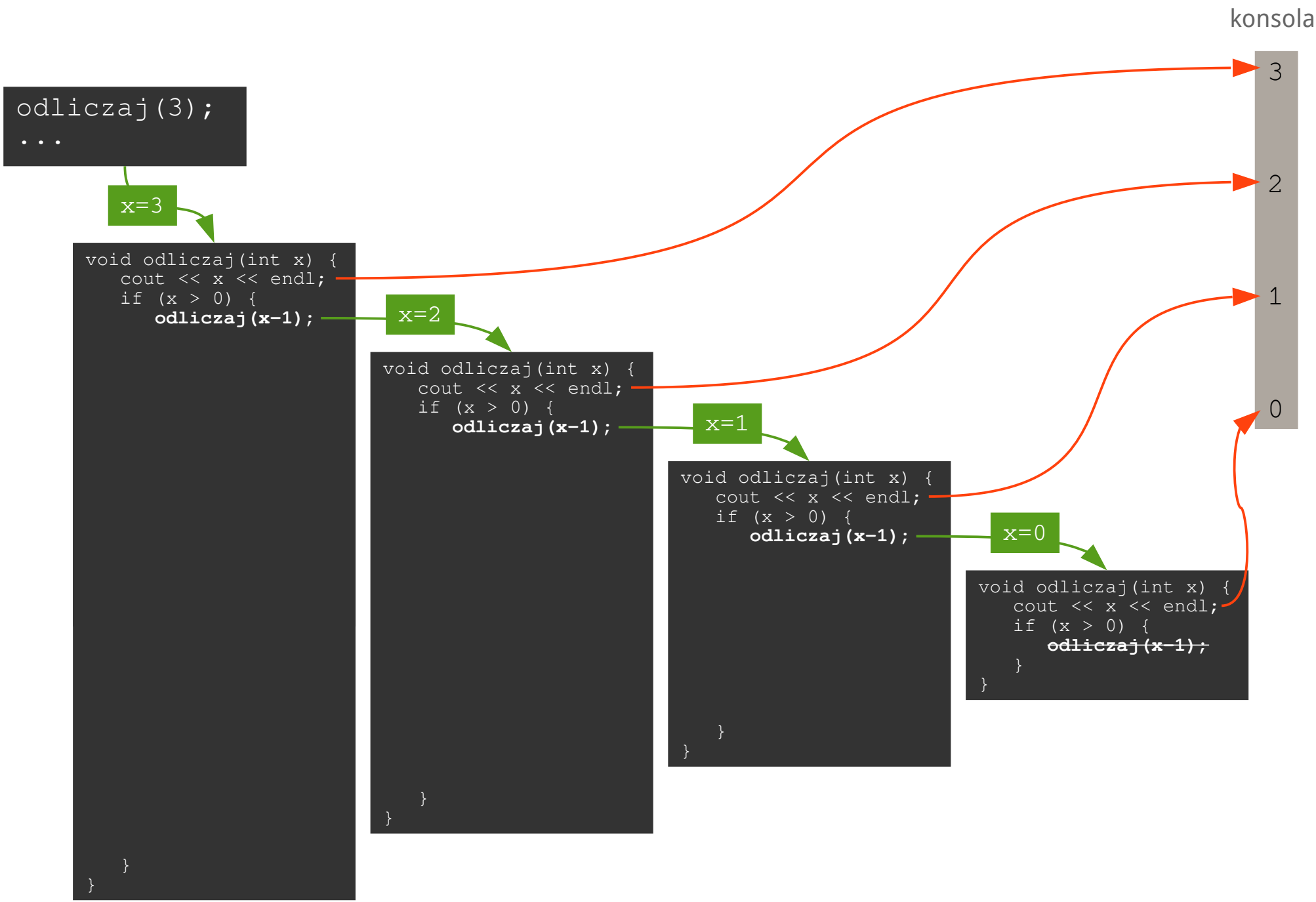
Jak działa rekurencja?



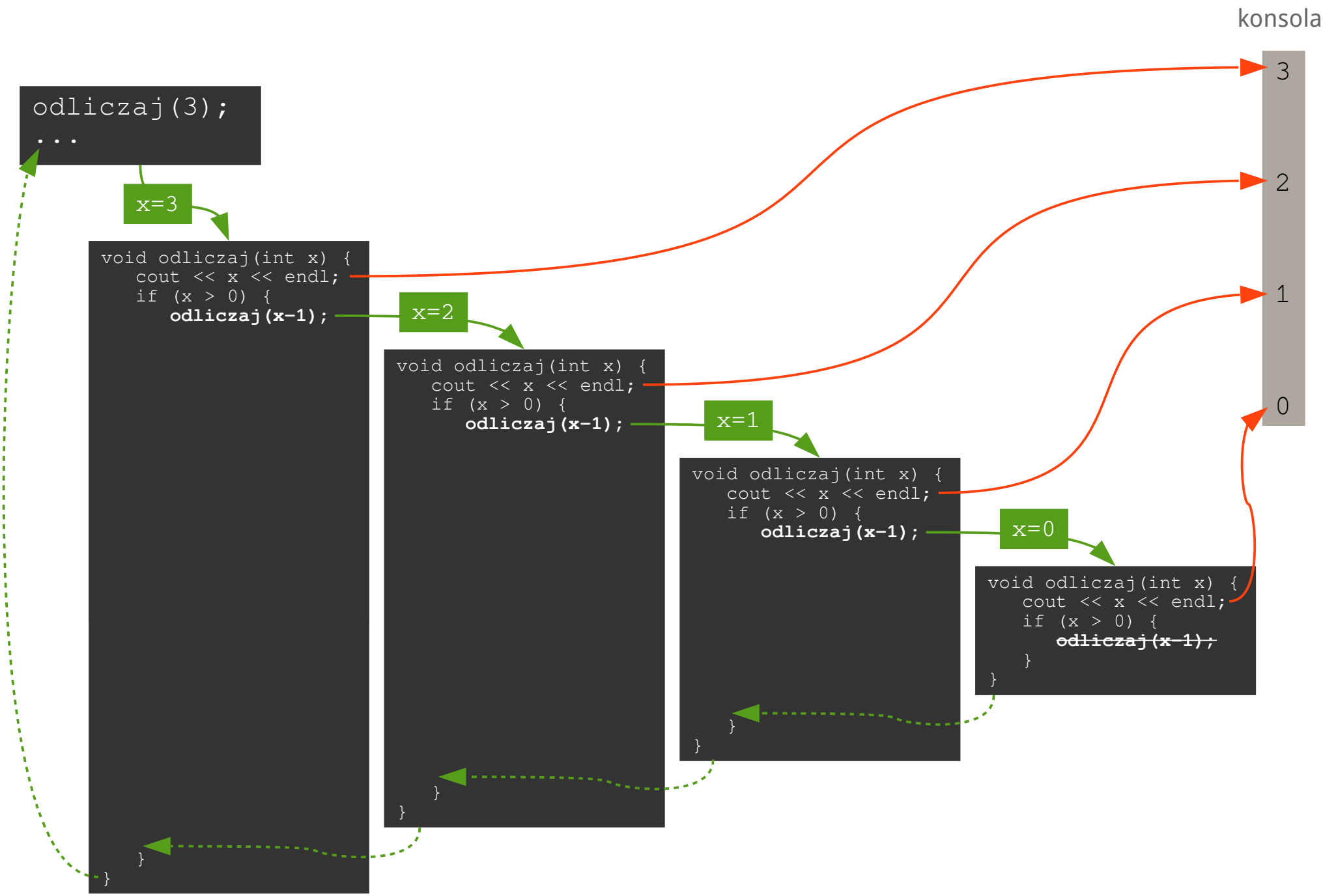
Jak działa rekurencja?



Jak działa rekurencja?



Jak działa rekurencja?



Przykłady algorytmów rekurencyjnych

Klasycznym przykładem ilustrującym rekurencję jest algorytm **obliczania silni**.

Definicję silni można przedstawić w następujący sposób:

0! = 1
n! = n * (n-1)! dla $n \in \mathbb{N}, n \geqslant 1$

Implementacja rekurencyjnej wersji algorytmu może wyglądać tak:

```
unsigned int silniaRek(unsigned int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * silniaRek(n-1);
    }
}
```

Z kolei wersja iteracyjna wygląda tak:

```
unsigned int silniaIt(unsigned int n) {
    unsigned int wynik = 1;
    while (n > 0) {
        wynik = wynik * n--;
    }
    return wynik;
}
```

Przykłady algorytmów rekurencyjnych

A teraz coś mniej matematycznego – **odwracanie wpisanego łańcucha znaków**.

Ma to działać tak: jeżeli wpiszę w konsoli `abc` i wcisnę Enter, ma wyświetlić `cba`.

Pojedyncze znaki odczytuje się za pomocą metody `cin.get(char)`, a wyświetla za pomocą metody `cout.put(char)`.

```
void odwroc() {
   char znak;
   cin.get(znak);
   if (znak != '\n') {
      odwroc();
      cout.put(znak);
   }
}
```

Przykłady algorytmów rekurencyjnych

Są sytuacje, gdy podejście rekurencyjne nie jest optymalne.

Za przykład może posłużyć wyznaczanie **liczb Fibonacciego**.

Definicja ciągu Fibonacciego: jeżeli dwie pierwsze liczby to 0 i 1, to każda liczba z ciągu jest sumą swoich dwóch poprzedników:

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2) dla $n \in \mathbb{N}, n \geqslant 2$

Implementacja tej funkcji jest bardzo prosta:

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

Funkcja jest jednak niewydajna (zwróćcie uwagę na podwójne wywołanie rekurencyjne).

Przykłady algorytmów rekurencyjnych

A jak by wyglądała wersja iteracyjna?

```
unsigned int fibIt(unsigned int n) {
    if (n < 2) {
        return n;
    } else {
        unsigned int n_1 = 1,
                     n_2 = 0,
                     wynik;
        for (int i=2; i<=n; ++i) {
            wynik = n_1 + n_2;
            n_2 = n_1;
            n_1 = wynik;
        }
        return wynik;
    }
}
```

Funkcja jest trochę bardziej skomplikowana, ale wykonuje się o wiele szybciej.

Waszym zadaniem będzie porównanie obu algorytmów.