STL #3

ZŁOŻONOŚĆ OBLICZENIOWA, ITERATORY, ALGORYTMY

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

ZADANIA

Repo GH `coders-school/stl`

Zadania wykonywane podczas zajęć online nie wymagają ściągania repo. Pliki będą tworzone od zera.

KRÓTKIE PRZYPOMNIENIE

CO JUŻ WIEMY

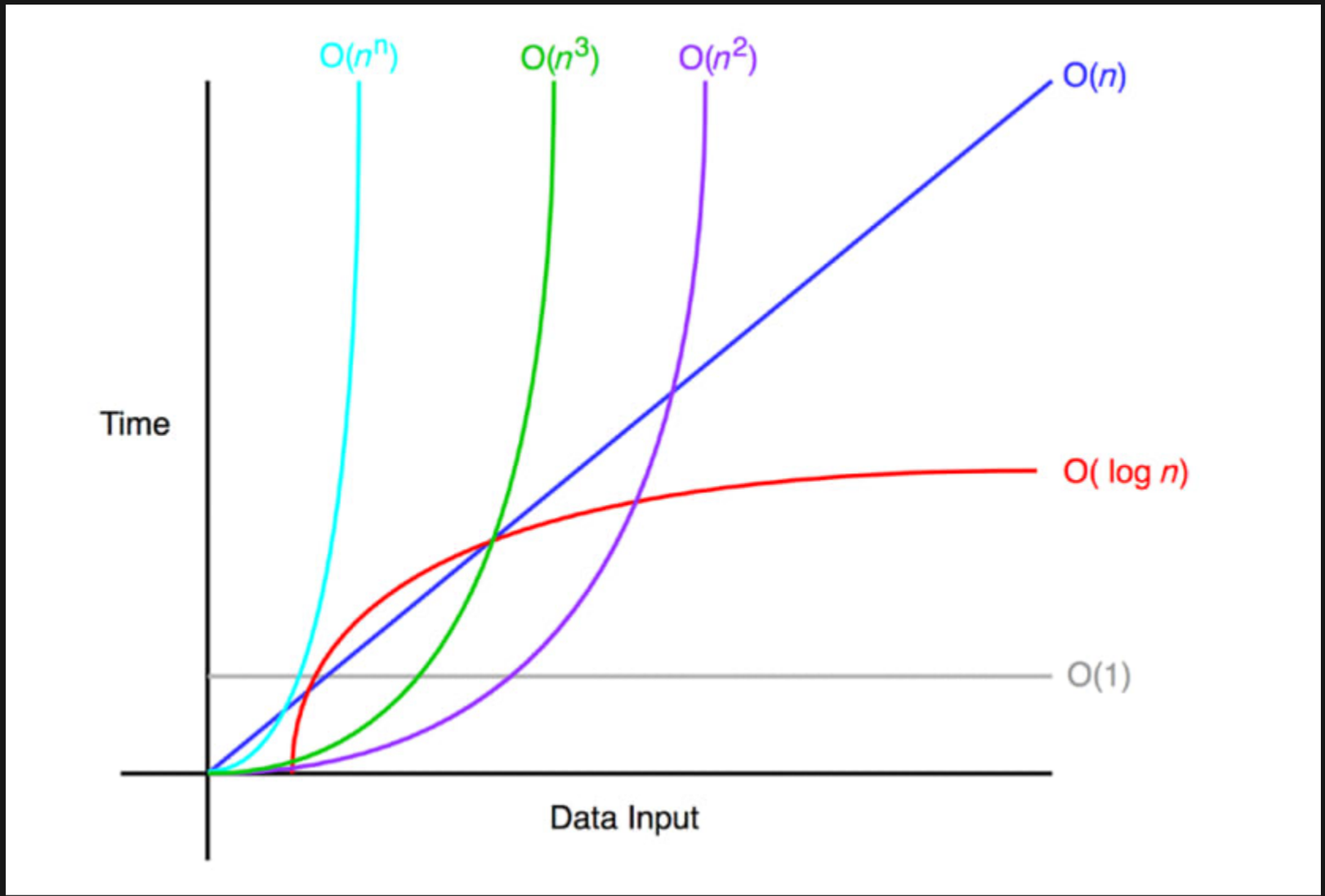
- co zapamiętaliście z poprzednich zajęć?
- co sprawiło największą trudność?
- co najłatwiej było wam zrozumieć?

AGENDA

- Złożoność obliczeniowa
- Iteratory
- Algorytmy

ZŁOŻONOŚĆ OBLICZENIOWA

ZŁOŻONOŚĆ OBLICZENIOWA



DEFINICJA

Złożoność obliczeniowa - oszacowanie czasu wykonania algorytmu. Mierzymy liczbę operacji, następnie szukamy funkcji opisującej liczbę operacji w zależności od danych wejściowych. Notacja O (dużego O) jest szacowaniem z góry. Ponieważ chcemy tylko przybliżyć wartość, pomijamy wszelkiego rodzaju stałe, które nie mają znaczenia dla dużych zbiorów danych wejściowych oznaczanych jako n. Zatem `O(2n + 5)`, `O(2n)` i `O(n)` są uznawane za złożoność obliczeniową `O(n)`.

ZŁOŻONOŚĆ O(1)

Jest to tzw. złożoność stała, która jest niezależna od liczby danych wejściowych. Przy obliczeniu sumy ciągu arytmetycznego (kod poniżej), nie iterujemy po wszystkich elementach tablicy, zatem czas wykonania jest stały i niezależny od wielkości tablicy.

```
int sum(std::vector<int> vec) {
    if (vec.empty()) {
        return 0;
    }

    return (vec.front() + vec.back()) * vec.size() / 2;
}

int main() {
    std::cout << sum({1, 2, 3, 4, 5, 6}) << std::endl;

    return 0;
}
```

Output: 21

ZŁOŻONOŚĆ O(logn)

Wyobraźmy sobie, że szukamy numeru telefonu naszego kolegi Andrzeja. Bierzemy książkę telefoniczną i otwieramy ją na środku i patrzymy, że wypada nam osoba o imieniu Kornelia. Wiemy, że Andrzej jest w pierwszej połowie książki adresowej, gdyż litera K jest dużo dalej w alfabecie niż litera A. Zatem znów otwieramy na środku pierwszej połowy i patrzymy, że widnieje tam imię Dominik. Więc powtarzamy nasze szukanie natrafiając w kolejnej części na Bartka, aż w końcu trafiamy na Andrzeja. Takie przeszukiwanie, w którym za każdym razem odrzucamy połowę zakresu jest właśnie zapisywane w notacji O(logn).

```
std::vector<int> vec{1, 2, 3, 4, 5, 6};
std::cout << std::boolalpha
          << std::binary_search(begin(vec), end(vec), 2) << std::endl
          << std::binary_search(begin(vec), end(vec), 0) << std::endl;
```

Output:

```
true
false
```

ZŁOŻONOŚĆ O(n)

Wyobraźmy sobie teraz sytuację, że w książce adresowej szukamy numeru Żanety. Jednak nie będziemy teraz przeszukiwać binarnie, tylko sprawdzimy ile zajmie nam to, gdy będziemy szukać osoba po osobie. Więc zaczynamy od litery A i 4 dni później znajdujemy w końcu numer Żanety ☺. Taka złożoność, gdzie sprawdzamy po kolei każdy element jest złożonością O(n).

ZŁOŻONOŚĆ O(n) - PRZYKŁAD

```cpp
constexpr size_t samples = 1'000'000'000;
constexpr size_t search_num = 987'654'321;
std::vector<int> vec(samples);
std::iota(begin(vec), end(vec), 0);

auto start = std::chrono::high_resolution_clock::now();
std::binary_search(begin(vec), end(vec), search_num);
auto stop = std::chrono::high_resolution_clock::now();
std::cout << "O(logn): " << std::chrono::duration_cast<std::chrono::nanoseconds>(stop - star

start = std::chrono::high_resolution_clock::now();
for (const auto el : vec) {
    if (el == search_num) {
        break;
    }
}
stop = std::chrono::high_resolution_clock::now();
std::cout << "O(n): " << std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start).
```

Example Output:

```
O(logn): 0 ns
O(n): 6'949'430'300 ns
```

ZŁOŻONOŚĆ O(nlog(n))

Jest to tzw. złożoność liniowo-logarytmiczna, której czas wykonania jest wprost proporcjonalny do iloczynu danych wejściowych i ich logarytmu. Wyobraźmy sobie teraz sytuację: próbujemy znaleźć numer Andrzeja w książce telefonicznej, ale nasz kolega zrobił nam psikus i pozamieniał strony. Teraz musimy ją posortować, zależy nam na czasie, więc chcemy to zrobić wydajnie. Dlatego wyrywamy kolejno strony z książki telefonicznej i wstawiamy je do nowej w zgodnej kolejności. Nie dość, że musimy zrobić taką operację dla n stron, to jeszcze musimy wstawiać je alfabetycznie, co zajmie nam `logn` czasu. Dlatego cały proces to `nlogn`.

ZŁOŻONOŚĆ O(nlog(n)) - PRZYKŁAD

```cpp
constexpr size_t samples = 1'000'000'000;
std::vector<int> vec(samples);
std::iota(begin(vec), end(vec), 0);
std::random_device rd;
std::mt19937 gen(rd());

//Here our colleague mixed up phone book.
std::shuffle(begin(vec), end(vec), gen);
auto start = std::chrono::high_resolution_clock::now();
std::sort(begin(vec), end(vec));
auto stop = std::chrono::high_resolution_clock::now();
std::cout << "O(nlogn): " << std::chrono::duration_cast<std::chrono::nanosecon
```

Possible output: `O(nlogn): 499'694'684'900 ns`

Previous slides output: `O(n): 6'949'430'300 ns`

ZŁOŻONOŚĆ O(n^x)

Jest to tzw. złożoność wielomianowa. Jej szczególnym i bardzo częstym przypadkiem jest złożoność kwadratowa - `O(n^2)`, której czas wykonania jest wprost proporcjonalny do kwadratu ilości danych wejściowych. Wyobraźmy sobie teraz inną sytuację. Udało nam się znaleźć numer Andrzeja i postanawiamy również zrobić psikus naszemu koledze, ale chcemy odwdzięczyć się z nawiązką. Dlatego drukujemy nową książkę telefoniczną, ale do każdego numeru dodajemy cyferkę '8' na początku numeru. Teraz nasz kolega nie dość, że musi poprawić każdą stronę `n` to jeszcze sprawdzić i poprawić każdy numer na podstawie oryginalnej książki. Taka złożoność obliczeniowa to `O(n^2)`. Przykładem złożoności `O(n^2)` jest popularne sortowanie bąbelkowe.

ZŁOŻONOŚĆ O(n^2) - PRZYKŁAD

```cpp
constexpr size_t samples = 1'000'000;
std::vector<int> vec(samples);
std::iota(begin(vec), end(vec), 0);
auto start = std::chrono::high_resolution_clock::now();
BubleSort(vec);
auto stop = std::chrono::high_resolution_clock::now();
std::cout << "O(n^2): " << std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start).count() << " ns\n"

constexpr size_t samples2 = 10'000'000;  // size is 10 times higher.
std::vector<int> vec2(samples2);
std::iota(begin(vec2), end(vec2), 0);
start = std::chrono::high_resolution_clock::noaw();
BubleSort(vec2);
stop = std::chrono::high_resolution_clock::now();
std::cout << "O(n^2): " << std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start).count() << " ns\n"

constexpr size_t samples3 = 100'000'000;  // size is 100 times higher.
std::vector<int> vec3(samples3);
std::iota(begin(vec3), end(vec3), 0);
start = std::chrono::high_resolution_clock::now();
BubleSort(vec3);
stop = std::chrono::high_resolution_clock::now();
std::cout << "O(n^2): " << std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start).count() << " ns\n"
```

Possible output:

```
O(n^2): 9'974'800 ns
O(n^2): 83'777'600 ns
O(n^2): 810'269'600 ns
```

ZŁOŻONOŚĆ O(x^n)

Jest to tzw. złożoność wykładnicza. Czas wykonania rośnie wykładniczo względem ilości danych wejściowych. Wyobraźmy sobie sytuację, w której nie dość, że książka zawiera błędy, które wcześniej celowo wprowadziliśmy, ale ktoś postanowił ją wydrukować w olbrzymim nakładzie i teraz musimy poprawić wszystkie książki, w których już czas poprawiania błędów wynosił n^2. Dla takiej kombinacji mówimy, że złożoność jest n^n. Czyli rośnie wykładniczo wraz ze wzrostem liczby książek (próbek). Przykładem może być algorytm przeszukiwania grafu DFS (deep-first search), jeżeli danymi wejściowymi będzie macierz. Ponieważ za każdym razem musimy przejść cały rząd, aż znajdziemy interesujący nas element, więc wykonamy n^n kroków. Rzadko spotykane, więc jest formą ciekawostki ☺

ZŁOŻONOŚĆ O(n!)

Jest to złożoność typu silnia, czas wykonania rośnie z szybkością silni względem ilości danych wejściowych. Przykładem problemu jest problem komiwojażera z grupy algorytmów grafowych. Należy znaleźć najkrótszą trasę rozpoczynając od miasta A przechodzącą jednokrotnie przez wszystkie pozostałe miasta i wracając do miasta A. Od wielu lat analitycy głowią się, jak poprawić ten algorytm. Wciąż mamy pole do popisu ☺. Nię będę tutaj wklejał kodu, ale zainteresowanych odsyłam do wyszukania sobie algorytmu komiwojażera.

ZŁOŻONOŚĆ O(n * n!)

Jeden z najgorszych scenariuszy jaki możemy wykonać dla algorytmu. Wyobraźmy sobie sytuację, że nasz kolega postanowił pokazać nam, że nie warto z nim zadzierać i skarży się waszemu przełożonemu, że namieszaliście w książce telefonicznej. Teraz za karę musicie napisać program do robota układającego książki na półkach waszego kolegi w kolejności alfabetycznej. Jednak Wy postanawiacie zrobić mu kolejny (najgorszy) psikus, i piszecie robota, który będzie układał książki losowo, a następnie sprawdzał, czy może udało się je ułożyć poprawnie a jak nie, to ponownie je ściągał i znów układał na nowo. W ten sposób robot będzie układał książki kilka tygodni lub miesięcy, lecz w końcu mu się to uda ☺

W ten sposób napisaliśmy idealnie nieoptymalny algorytm sortowania o złożoności `O(n * n)!`. Przykładem takiego sortowania jest bogosort.

ZŁOŻONOŚĆ O(n * !n) PRZYKŁAD

```cpp
std::random_device rd;
std::mt19937 generator(rd());

void BogoSort(std::vector<int>& vec) {
    while (!std::is_sorted(vec.begin(), vec.end())) {
        std::shuffle(vec.begin(), vec.end(), generator);
    }
}

int main() {
    constexpr size_t samples = 10;  // Only 10 elements! Try it for 100 :)
    std::vector<int> vec(samples);
    std::iota(begin(vec), end(vec), 0);
    std::shuffle(vec.begin(), vec.end(), generator);
    for (int i = 0; i < 5; ++i) {
        auto start = std::chrono::high_resolution_clock::now();
        BogoSort(vec);
        auto stop = std::chrono::high_resolution_clock::now();
        std::cout << "O(n * n!): " << std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start).count() << " ns
    }

    return 0;
}
```

Possible output:

```
O(n * n!): 35'938'300 ns
O(n * n!): 85'772'000 ns
O(n * n!): 899'885'600 ns
O(n * n!): 2'603'326'600 ns
O(n * n!): 145'608'700 ns
```

Q&A

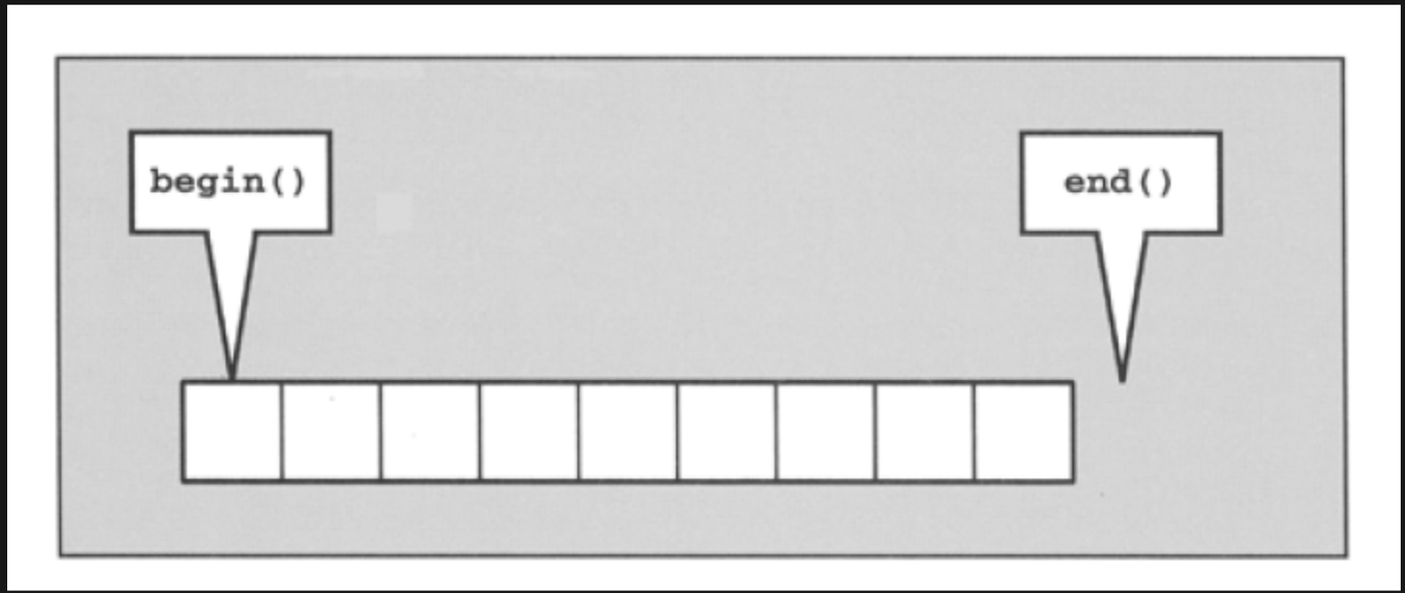
ITERATORY

KRÓTKA POWTÓRKA #1

Iterator jest to obiekt, który wskazuje na dany element w kontenerze. W zależności od typu możemy na nim wykonywać różne operacje, np: inkrementować go `operator++`, dekrementować `operator--` lub wykonywać operacje typu `it += 6`. W celu odwołania się do wskazywanego elementu przez iterator używamy `operator*` czyli dereferencji (jak na zwykłych wskaźnikach).

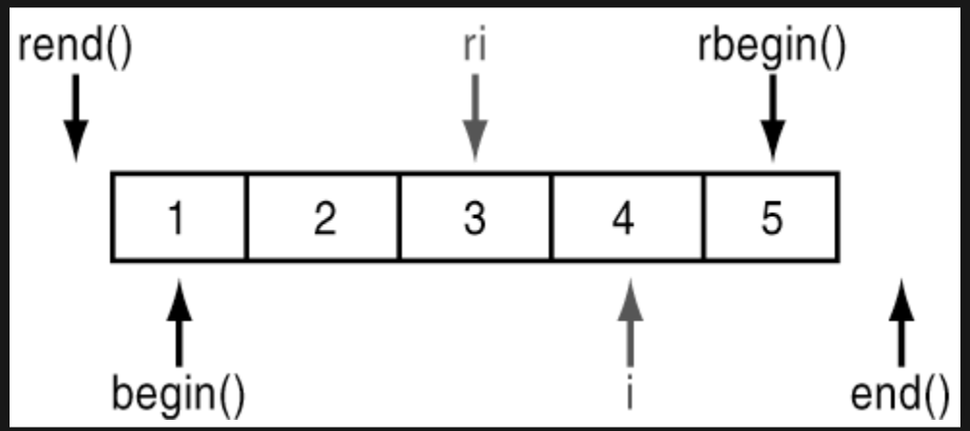
KRÓTKA POWTÓRKA #2

Każdy kontener ma 2 końce. Na jeden z nich wskazuje `begin()`, a na drugi `end()`.



KRÓTKA POWTÓRKA #3

Dla niektórych kontenerów możemy także pobrać odwrotny iterator (ang. reverse iterator) umożliwiający nam przejście wstecz przez zakres.

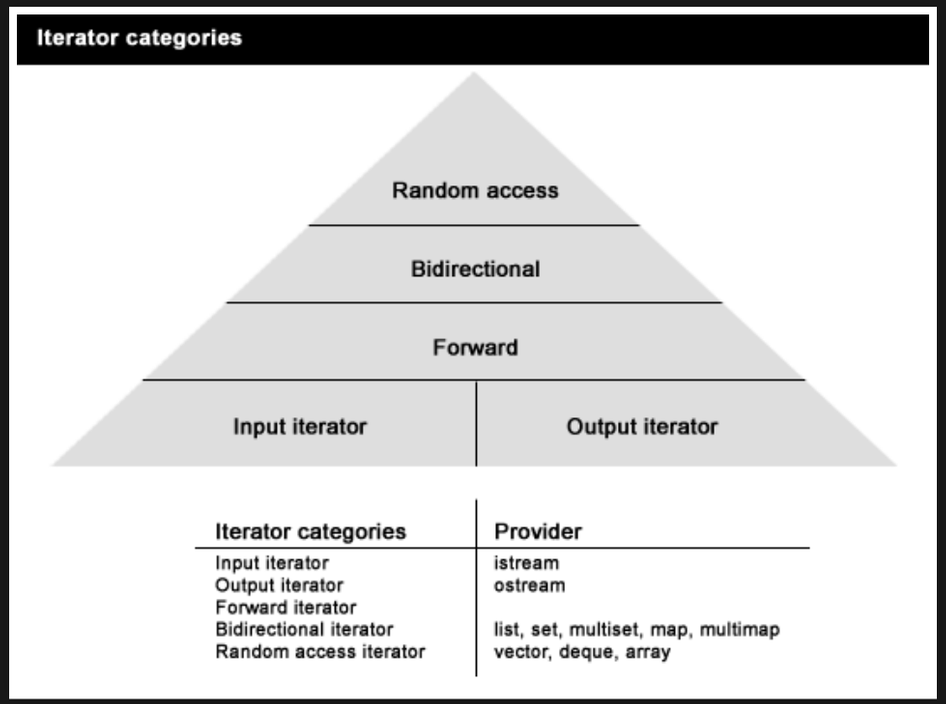


KRÓTKA POWTÓRKA #4

Jeżeli nie chcemy modyfikować danych wskazywanych przez iterator zastosujemy opcję z przedrostkiem `c` pochodzącym od słowa `constant`.

- `cbegin()`
- `cend()`
- `crbegin()`
- `crend()`

HIERARHIA ITERATORÓW



Pytanie, co powinno się znaleźć w miejscu `forward_iterator`?

- `std::forward_list`
- `std::unordered_set`
- `std::unordered_map`

"NAJBIEDNIEJSZE" ITERATORY, CZYLI INPUT I OUTPUT

Input iterator: pochodzi np. ze strumienia `std::istream`, czyli znanego nam `std::cin`. Raz wczytane dane znikają, nie możemy ich ponownie odczytać. Mamy możliwość tylko jednorazowego przejścia przez dane. Innym słowem jak tylko odczytamy jakąś daną nasz operator od razu jest inkrementowany.

```
int a;
int b;
std::cin >> a >> b;
```

Output iterator: pochodzi np. ze strumienia `std::ostream`, czyli znanego nam `std::cout`. Raz wypisane dane znikają, nie możemy ich ponownie wyświetlić, musimy ponownie podać dane.

```
int a;
int b;
std::cin >> a >> b;
std::cout << a << ' ' << b;
```

PYTANIE

Jakie mamy dostępne operatory dla input iterator, a jakie dla output iterator?

Input iterator:

- `operator++`
- `operator*`
- `operator->`
- `operator==`
- `operator!=`

Output iterator:

- `operator()++`
- `operator*`

FORWARD ITERATOR

Jest to iterator, który umożliwia nam wielokrotne przejście danego zakresu w jedną stronę (w przód).

```
std::forward_list<int> list {1, 2, 3, 4, 5};
for (auto it = list.begin() ; it != list.end() ; ++it) {
    std::cout << *it << '\n';
}
```

Output: 1 2 3 4 5

Pytanie: Jakie mamy dostępne operatory dla tego iteratora?

- operator++
- operator*
- operator->
- operator==
- operator!=

BIDIRECTIONAL ITERATOR

Jest to iterator, który umożliwia nam wielokrotne przejście danego zakresu w obie strony (w przód i tył).

```
std::list<int> list{1, 2, 3, 4, 5};
for (auto it = list.begin(); it != list.end(); ++it) {
    std::cout << *it << ' ';
}
auto last = std::prev(list.end());
for (auto it = last; it != std::prev(list.begin()); --it) {
    std::cout << *it << ' ';
}
```

Output: `1 2 3 4 5 5 4 3 2 1`

Pytanie: Jakie mamy dostępne operatory dla tego iteratora?

- `operator++`
- `operator*`
- `operator->`
- `operator==`
- `operator!=`
- `operator--`

RANDOM ACCESS ITERATOR

Jest to iterator, który umożliwia nam wielokrotne przejście danego zakresu w obie strony (w przód i tył), a także dostęp do dowolnego obiektu.

```
std::vec<int> vec{1, 2, 3, 4, 5};
for (auto it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << ' ';
}
auto last = std::prev(vec.end());
for (auto it = last; it != std::prev(vec.begin()); --it) {
    std::cout << *it << ' ';
}
std::cout <<  vec[3];
```

Output: 1 2 3 4 5 5 4 3 2 1 3

RANDOM ACCESS ITERATOR

Pytanie: Jakie mamy dostępne operatory dla tego iteratora?

- `operator++`
- `operator*`
- `operator->`
- `operator==`
- `operator!=`
- `operator--`
- `operator<`
- `operator<=`
- `operator>`
- `operator>=`
- `operator+`
- `operator-`
- `operator[]`

CIEKAWOSTKA

W C++ 17 wprowadzono jeszcze typ: `ContiguousIterator`. Zawiera on wszystkie cechy Random Access iterator oraz zapewnia, że wszystkie dane są ułożone w jednym miejscu w pamięci.

Q&A

ALGORYTMY

TYPY ALGORYTMÓW

Podaj, jakie znasz algorytmy:

- Niemodyfikujące sekwencji?
- Modyfikujące sekwencje?
- Partycjonujące?
- Sortujące?
- Przeszukujące binarnie?
- Operujące na kopcu (ang. Heap)?
- Min/max?
- Porównujące?
- Dokonujące permutacji?
- Numeryczne?

QUIZ

Korzystając z cppreference odpowiedz na pytania:

- Jaką złożoność ma `std::sort()`?
- Jaki algorytm użyjesz dla połączenia 2 kontenerów naprzemiennie?
- Jaki algorytm użyjesz, aby zsumować wszystkie elementy w `std::vector`?
- Jaki algorytm użyjesz, aby pomnożyć 2 kontenery ze sobą?
- Jaki algorytm użyjesz, aby usunąć liczby parzyste?
- Jaki algorytm użyjesz, aby wypełnić strukturę wartościami od 0 do n?
- Jaki algorytm użyjesz, aby zawsze na 1. miejscu (0 index) znajdował się największy element?
- Jaki algorytm użyjesz, by wyszukać czy dana liczba znajduje się w posortowanym kontenerze?
- Jaki algorytm użyjesz, aby podzielić kontener na 2 zakresy, zawierające mniejsze i większe wartości niż 10?
- Jaki algorytm użyjesz, aby zmienić kolejność 2 elementów w kontenerze?

STD::ROTATE

Link do wideo

ZADANIE 1

Napisz funkcję, która przyjmie `std::vector<int>&`, zmienną `int value`, oraz zmienną `int new_pos`. Funkcja powinna odnaleźć `value` w `std::vector<int>` i jeżeli ją znajdzie wstawić ją na nowe miejsce `new_pos`, odpowiednio przesuwając resztę elementów by nie zaburzyć ich sekwencji. Następnie zwróć `true`, jeżeli proces się udał, lub `false`, jeżeli nie dało się zmienić kolejności (np. nie istnieje wartość).

```
bool ChangePos(std::vector<int>& vec, int value, int new_pos)
```

ROZWIĄZANIE 1

```
bool ChangePos(std::vector<int>& vec, int value, int new_pos) {
    if (new_pos >= vec.size()) {
        return false;
    }

    auto begin = vec.begin();
    auto end = vec.end();
    auto it = std::find(begin, end, value);
    if (it == end) {
        return false;
    }

    auto item_pos = std::distance(begin, it);
    if (item_pos > new_pos) {
        std::rotate(begin + new_pos, begin + item_pos, begin + item_pos + 1);
    } else {
        std::rotate(begin + item_pos, begin + item_pos + 1, begin + new_pos +
    }

    return true;
}
```

STD::STABLE_PARTITION

Przykład wideo

ZADANIE 2

Napisz funkcję, która przyjmie dwa `std::vector<int>` oraz zmienną `int`. Pierwszy `std::vector<int>` zawiera kontener na którym operujemy, drugi zawiera wartości jakie chcemy przenieść, a wartość `int` to numer indexu, na który chcemy przenieść wartości.

```
void ChangePos(std::vector<int>& vec, const std::vector<int>& values, int position)
```

ROZWIĄZANIE 2

```cpp
void ChangePos(std::vector<int>& vec, const std::vector<int>& values, int new_pos) {
    auto pred {[&](auto& el){
        return (std::find(values.begin(), values.end(), el) == values.end());
    }};

    auto middle = vec.begin() + new_pos + values.size() - 1;
    std::stable_partition(vec.begin(), middle, pred);
    std::stable_partition(middle, vec.end(), [&](auto& el){ return !pred(el); });
}

void PrintVec(const std::vector<int>& vec) {
    for (const auto el : vec) {
        std::cout << el << ' ';
    }
    std::cout << '\n';
}

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    PrintVec(vec);
    ChangePos(vec, {1, 3, 9}, 4);
    PrintVec(vec);
```

ZADANIE 3

Napisz funkcję Gather, która przyjmie `std::vector<char>` oraz spowoduje, że wszystkie wystąpienia * pojawią się w środku `std::vector<char>`.

Input:
```
std::vector<char> vec {'*', ' ', '@', '*', '#', '@', '^', '*', '(', ')', ' ', '*'};
```

Output:
```
@ # @ * * * * ^ ( )
```

ROZWIĄZANIE 3

```
void Gather(std::vector<char>& vec) {
    auto pred {[&](auto& el){
        return el != '*';
    }};

    auto middle = vec.begin() + vec.size() / 2;
    std::stable_partition(vec.begin(), middle, pred);
    std::stable_partition(middle, vec.end(), [&](auto& el){ return !pred(el); });
}

void PrintVec(const std::vector<char>& vec) {
    for (const auto el : vec) {
        std::cout << el << ' ';
    }
    std::cout << '\n';
}

int main() {
    std::vector<char> vec {'*', '$', '@', '*', '#', '@', '^', '*', '(', ')', '$', '*'};
    PrintVec(vec);
    Gather(vec);
    PrintVec(vec);
```

ZADANIE 4

Napisz funkcję `GetVec(size_t count)`, która zwróci `std::vector<int>` z wartościami od `10` do `10 + n`, inkrementując je co 1. Następnie napisz drugą funkcję `int Multiply(std::vector<int> vec)`, która zwróci wartość równą iloczynowi każdego elementu `std::vector<int>`.

Input: `GetVec(7)`

Output: `{10, 11, 12, 13, 14, 15, 16}`

Input: `Multiply(vec)`

Output: `57657600`

ROZWIĄZANIE 4

```
std::vector<int> GetVec(size_t count) {
    std::vector<int> vec(count);
    std::iota(begin(vec), end(vec), 10);
    return vec;
}

int Multiply(const std::vector<int>& vec) {
    return std::accumulate(begin(vec), end(vec), 1, std::multiplies<int>());
}

int main() {
    std::cout << Multiply(GetVec(7)) << '\n';

    return 0;
}
```

Q&A

ZADANIE DOMOWE

ZADANIE 1

Wykorzystując `std::inner_product`, napisz program, który obliczy średnią arytmetyczną dwóch `std::vector<int>`.

Input: `{1, 2, 3, 4}` `{1, 2, 3, 4}`

Output: `2.5`

ZADANIE 1 #2

Następnie wykorzystując `std::inner_product`, napisz funkcję, która obliczy odległość między 2 punktami dla n wymiarów.

Metrykę euklidesową w przestrzeni definiujemy:

$$\mathbb{R}^n$$

$$d_e(\mathbf{x}, \mathbf{y}) = \sqrt{(y_1 - x_1)^2 + \cdots + (y_n - x_n)^2}$$

Input: `{7, 4, 3}` `{17, 6, 2}`

Output: `10.247`

ZADANIE 2

Napisz program `advanced_calculator`. Program ten powinien posiadać pętle główną, która będzie przyjmować dane od użytkownika. Wszystkie komendy kalkulatora powinny być przechowywane w mapie, która ma klucz `char` w postaci znaku odwołującego się do komendy (np. `+` -> dodaj , `%` -> modulo), a jej wartością powinna być funkcja `std::function<>` będąca wraperem na wyrażenia lambda dokonujące określonej kalkulacji. Program powinien także zwracać odpowiedni error code, jeżeli użytkownik poda złe dane np. dzielenie przez 0 lub spróbuje dodać `ala + 5`.

Input: `5 + 5` -> operacja dodawania dwóch liczb 5 i 5 -> output: `10`.

Input: `5 ^ 2` -> operacje potęgowania -> output: `25`.

Input `125 $ 3` -> operacja pierwiastka (sqrt za długie), pierwiastek sześcienny z 125 -> output: 5.

FUNKCJE KALKULATORA

- Dodawanie, mnożenie, dzielenie, odejmowanie, (+, *, /, -),
- Modulo (%),
- Obliczanie silni (!),
- Podnoszenie liczby do potęgi (^),
- Obliczanie pierwiastka ($).

ERROR CODE

- Ok,
- `BadCharacter` -> Jeżeli użytkownik poda znak inny niż liczbę.
- `BadFormat` -> Jeżeli użytkownik poda zły format komendy np. + 5 4, powinno być 4 + 5.
- `DivideBy0` -> dzielenie przez 0.
- `SqrtOfNegativeNumber` -> pierwiastek z liczby ujemnej.
- `ModuleOfNonIntegerValue` -> Próba obliczenia % na liczbie niecałkowitej.

FUNKCJA, KTÓRA BĘDZIE TESTOWANA

- `ErrorCode process(std::string input, double* out)`
- Funkcja ta powinna w pętli przyjmować kolejne dane od użytkownika oraz dokonywać odpowiedniej kalkulacji.
- Jeżeli dane są poprawne, zwróci `ErrorCode:Ok`, a w zmiennej `out` zapisze dane.
- Jeżeli wystąpi któryś z błędów, funkcja go zwróci, a w `out` nie zapisze nic.

CODERS SCHOOL