Algorytmy STL

Wykład 13

Algorytmy STL

- Algorytmy zdefiniowane są w pliku nagłówkowym **<algorithm>**
  - Znajduje się tam też kilka pomocniczych funkcji takich jak **min, max, swap**
- Algorytmy STL służące do przetwarzania numerycznego zdefiniowane są w pliku nagłówkowym **<numeric>**
- Podczas pracy z algorytmami często potrzebne są obiekty funkcyjne

Cechy ogólne

- Algorytmy przetwarzają co najmniej jeden zakres wyznaczony przez iteratory
  - Pierwszy zakres wyznaczony jest przez początek i koniec
    - Zakres zawsze jest półotwarty `[beg, end)`
  - Dla następnych na ogół wystarczy podać początek, bo liczba elementów przetwarzanych zdefiniowana jest przez zakres pierwszy
- Do funkcji wywołującej należy zapewnienie poprawności zakresów, tzn.
  - Koniec zakresu musi być osiągalny z początku
  - Zakresy muszą zawierać wystarczająca liczbę elementów
- Algorytmy działają w trybie nadpisywania, a nie wstawiania
  - Chyba, że użyjemy iteratorów wstawiających

Elastyczność

- Niektóre algorytmu pozwalają na przekazanie operacji zdefiniowanych przez programistę, które są przez nie wywoływane
  - Operacje mogą być zwykłymi funkcjami lub funktorami
    - Operacje numeryczne wykonujące np. iloczyn
  - Jeżeli funkcje zwracają wartość logiczną to nazywamy je predykatami (+dodatkowe warunki)
    - Np. do wyszukiwania używamy predykatów jednoargumentowych
    - Do sortowania predykatów dwuargumentowych
    - Predykat jednoargumentowy służący do określania, na których elementach wykonywać operację

Klasyfikacja algorytmów

- Algorytmy sklasyfikowano wg ich przeznaczenia
  - Np. czytające, modyfikujące, zmieniające kolejność
- Nazwa algorytmu i przyrostki
  - Na ogół nazwa intuicyjnie definiuje zadanie
  - Przyrostek **`_if`** – używanie jest gdy istnieją dwie postacie algorytmu, jedna nie wymaga podania funktora druga wymaga
    - **`find`** – szuka wartości, **`find_if`** – szuka elementu spełniającego podane kryterium
    - Nie jest tak zawsze (jeżeli funkcja wymaga istnienia funktora) np. **`min_element`**
  - Przyrostek **`_copy`** – wskazuje że elementy będą kopiowane do zakresu docelowego np. **`reverse`** i **`reverse_copy`**

Kategorie algorytmów

- Algorytmy niemodyfikujące (*nonmodifying algorithms*)
- Algorytmy modyfikujące (*modifying algorithms*)
- Algorytmy usuwające (*removing algorithms*)
- Algorytmy mutujące (*mutating algorithms*)
- Algorytmy sortujące (*sorting algorithms*)
- Algorytmy przeznaczone do zakresów posortowanych (*sorted range algorithms*)
- Algorytmy numeryczne (*numeric algorithms*)
- Niektóre algorytmu należą do kilku kategorii

Algorytmy niemodyfikujące

- Algorytmy te nie zmieniają kolejności, wartości przetwarzanych elementów, współpracują z iteratorami wejściowymi i postępującymi
  - Można je wywoływać dla wszystkich standardowych kontenerów
- Zestawienie wszystkich niemodyfikujących algorytmów
  - `for_each()`, `count()`, `count_if()`, `min_element()`, `max_element()`, `find()`, `find_if()`, `search_n()`, `search()`, `find_end()`, `find_first_of()`, `adjacent_find()`, `equal()`, `mismatch()`, `lexicographical_compare()`

`for_each()`

- Algorytm `for_each()` jest bardzo elastyczny, pozwala na dostęp, przetwarzanie i modyfikowanie każdego elementu na wiele różnych sposobów
  - `for_each(beg, end, op)`
  - Dla każdego elementy z przedziału `[beg, end)` wywołuje `op(elem)`
  - Zwraca kopię `op`, która jest ignorowana
  - Złożoność liniowa
- Przykład cpp_13.1

count(), count_if(), min_element(), max_element()

- count() count_if() służą do zliczania elementów z zakresu [beg, end)
  - count(beg, end, val) - o wartości val
  - count_if(beg, end, op) - dla których jednoargumentowy predykat op(elem) zwraca prawdę
  - Złożoność liniowa
- min_element(), max_element() zwracają odpowiednio pozycję najmniejszego i największego elementu
  - min_element(beg, end) porównanie za pomocą <
  - max_element(beg, end, op) porównanie za pomocą predykatu dwuargumentowego op
  - Złożoność liniowa
- Przykład cpp_13.2

Wyszukiwanie elementów

- `find(beg, end, val)` i `find_if(beg, end, op)` - zwracają odpowiednio pozycję pierwszego elementu równego `val` lub dla którego predykat jednoargumentowy `op` jest prawdziwy
  - Do zakresów posortowanych stosujemy inne algorytmy
  - Złożoność liniowa
- `search_n(beg, end, n, val)` i `search_n(beg, end, n, val, op)` - zwracają odpowiednio pierwszą pozycję `n` kolejnych wystąpień wartości `val` lub dla których wywołanie `op(elem, val)` zwraca `true`
  - Złożoność liniowa
- Przykład cpp_13.3

Wyszukiwanie podzakresów

- `search(beg, end, srbeg, srend)` i `search(beg, end, srbeg, srend, op)` - zwracają odpowiednio pozycję pierwszego elementu podzakresu zgodnego z zakresem `[srbeg, srend)` w zakresie `[beg, end)`, dla którego elementy są równe, lub dla którego dla każdego elementu predykat dwuargumentowy `op(elem, srelem)` jest prawdziwy
  - Złożoność liniowa
  - W przypadku nie znalezienie odpowiedniego podzakresu zawracana jest wartość `end()`
- Podobnie działają dwa algorytmy `find_end()` z tą różnicą, że szukają ostatniego podzakresu
- Przykład cpp_13.4

Wyszukiwanie pierwszego z kilku możliwych elementów

- `find_first_of(beg, end, srbeg, srend)` i `find_first_of(beg, end, srbeg, srend, op)` - zwracają pierwszą pozycje elementu w zakresie `[beg, end)` , który również występuje w zakresie `[srbeg, srend)` lub predykat `op(elem, srelem)` zwraca prawdę
  - W przypadku nie znalezienia zwracana jest wartość `end()`
  - Złożoność liniowa
- Przykład cpp_13.5

Wyszukiwanie dwóch kolejnych równych elementów

- `adjacent_find(beg, end)` i `adjacent_find(beg, end, op)` - zwraca pozycję pierwszego elementu z zakresu `[beg, end)` o wartości równej następnemu elementowi lub dla którego `op(elem, nextElem)` zwraca prawdę
  - Jeżeli nie znajdzie to zwracane jest `end()`
  - Złożoność liniowa
- Przykład cpp_13.6

Porównywanie zakresów

- `equal(beg, end, cmpBeg)` i `equal(beg, end, cmpBeg, op)` - sprawdzają czy elementy z zakresu `[beg, end)` równe są elementom z zakresu rozpoczynającego się od `cmpBeg`, druga wersja sprawdza czy predykat `op(elem, cmpElem)` zwraca `true`
  - Zakres zaczynający się od `cmpElem` powinien posiadać wystarczającą liczbę elementów
  - Zwraca wartość logiczną
  - Złożoność liniowa
- Przykład cpp_13.7

Wyszukiwanie pierwszej różnicy

- `mismatch(beg, end, cmpBeg)` i `mismatch(beg, end, cmpBeg, op)` - zwraca pozycję pierwszych dwóch różnych elementów (parę iteratorów) z zakresu `[beg, end)` w zakresie rozpoczynającym się od `cmpBeg`, druga wersja zwraca parę jeśli predykat `op(elem, cmpElem)` zwraca `false`
  - Jeśli nie ma różnic to zwracana para zawiera `end()` oraz odpowiadającą jej wartość z drugiego ciągu
  - Jeśli `mismatch` zwróci `end()` to nie znaczy, że ciągi są równe
  - Złożoność liniowa
- Przykład cpp_13.8

Testowanie relacji mniejsze niż

- `lexicographical_compare(beg1, end1, beg2, end2)` i `lexicographical_compare(beg1, end1, beg2, end2, op)` - obie wersje sprawdzają czy elementy z zakresu `[beg1, end1)` są mniejsze od elementów `[beg2, end2)`. W pierwszym przypadku używany jest operator `<` a w drugim predykat `op(elem1, elem2)`
  - Zwraca wartość logiczną
  - Złożoność liniowa
- Przykład cpp_13.9

Algorytmy modyfikujące

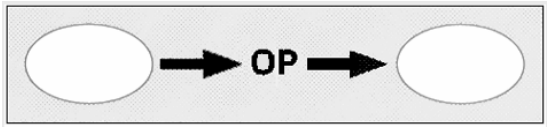
- Algorytmy takie mogą bezpośrednio lub przy kopiowaniu modyfikować wartości elementów
- Elementy kontenerów asocjacyjnych są stałe dla zagwarantowania posortowania kolejności
  - Nie można ich używać jako docelowych dla algorytmów modyfikujących
- Zestawienie wszystkich modyfikujących algorytmów
  - `for_each()` - argument musi być przesyłany przez referencję, `copy()`, `copy_backward()`, `transform()`, `merge()`, `swap_ranges()`, `fill()`, `fill_n()`, `generate()`, `generate_n()`, `replace()`, `replace_if()`, `replace_copy()`, `replace_copy_if()`

Kopiowanie

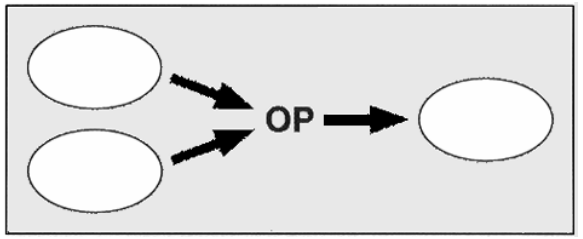
- **copy(srcBeg, srcEnd, destBeg)** i **copy_backward(srcBeg, srcEnd, destEnd)** - kopiują elementy z przedziału **[srcBeg, srcEnd)** na pozycje od **destBeg**, druga wersja wykonuje iterację odwrotną (**destEnd** jest końcem zakresu)
  - Pozycja **destBeg** i **destEnd** nie powinna należeć do **[srcBeg, srcEnd)**
  - **copy** operuje na iteratorach postępujących pozycja **destBeg** powinna być za **srcBeg** (kopiowanie od początku)
  - **copy_backward** operuje na iteratorach dwukierunkowych pozycja **destEnd** powinna być za **srcEnd** (kopiowanie od końca)
  - Należy zapewnić poprawność zakresów
  - Zwracają pozycję za ostatnim skopiowanym elementem
- Przykład cpp_13.10

Przekształcanie elementów

- `transform(sBeg, sEnd, dBeg, op)` - przekształca elementy źródłowe i umieszcza w zakresie docelowym
  - Wywołuje `op(elem)` przed kopiowaniem



- `transform(s1Beg, s1End, s2Beg, dBeg, op)` - wykonuje operacje na kombinacjach elementów z dwóch ciągów źródłowych i zapisuje w zakresie docelowym
  - Wywołuje `op(elem1, elem2)` przed kopiowaniem
  - Złożoność liniowa
- Pozycje źródłowe mogą być takie same jak docelowe
- Przykład cpp_13.11



Wymiana elementów

- **`swap_ranges(beg1, end1, beg2)`** - wymienia elementy z zakresu **`[beg1, end1)`** z odpowiadającymi im elementami na pozycjach zaczynających się od **`beg2`**, zwraca pozycję za ostatnim wymienionym elementem w drugim zakresie
  - Zakresy nie mogą się pokrywać
  - Złożoność liniowa
- Przykład cpp_13.12

Przypisywanie nowych wartości

- `fill(beg, end, val)`, `fill_n(beg, n, val)` - przypisuje wartości `val` odpowiednio w zakresie `[beg, end)` lub od `beg`, `n` elementom
- `generate(beg, end, op)`, `generate_n(beg, n, op)` - przypisuje wartości wywołanie funkcji `op()` odpowiednio w zakresie `[beg, end)` lub od `beg`, `n` elementom
- Złożoność liniowa
- Nic nie zwracają
- Przykład cpp_13.13

Zastępowanie elementów

- `replace(beg, end, val, newval)`, `replace_if(beg, end, op, newval)` - zastępuje odpowiednio elementy `val` lub elementy dla których `op(val)` jest prawdziwe elementami `newval` w zakresie `[beg, end)`
  - Nic nie zwracają
- `replace_copy(sbeg, send, dbeg, val, newval)`, `replace_copy_if(sbeg, send, dbeg, op, newval)` - jw. tylko przy kopiowaniu do zakresu zaczynającego się od `dbeg`
  - Zwracają pozycję za ostatnim przekopiowanym elementem w zakresie docelowym
- Złożoność liniowa
- Przykład cpp_13.14

Algorytmy usuwające

- Algorytmy usuwające są specjalną postacią algorytmów modyfikujących
  - Mogą usuwać elementy z pojedynczego zakresu lub przy jednoczesnym kopiowaniu do innego
  - Nie można używać kontenerów asocjacyjnych jako docelowych dla algorytmów usuwających
  - Nie mogą zmieniać liczby elementów
  - Usuwają tylko logicznie tzn. nadpisują elementy usuwane następnymi nieusuniętymi
  - Zwracają nowy logiczny koniec kontenera
- Zestawienie wszystkich usuwających algorytmów
  - **`remove()`, `remove_if()`, `remove_copy()`, `remove_copy_if()`, `unique()`, `unique_copy()`**

Usuwanie wartości

- `remove(beg, end, val)`, `remove_if(beg, end, op)` – usuwają elementy z zakresu `[beg, end)` odpowiednio o wartości `val` lub dla których `op(val)` zwraca prawdę
  - Zwracają nowy logiczny koniec zmodyfikowanego ciągu
  - Złożoność liniowa
- `remove_copy(beg, end, dbeg, val)`, `remove_copy_if(beg, end, dbeg, op)` – jw. tylko usuwają elementy kopiowane do zakresu zaczynającego się od `dbeg` (kombinacja algorytmów `copy + remove` oraz `copy + romove_if`)
- Przykład cpp_13.15

Usuwanie kolejnych powtórzeń

- `unique(beg, end)`, `unique(beg, end, op)` - odpowiednio usuwa elementy z zakresu `[beg, end)` powtarzające się lub takie dla których `op(elm, e)` zwraca prawdę (usuwa wszystkie element po `e` dla których `op` zwraca prawdę)
  - Zwracają nowy logiczny koniec zmodyfikowanego ciągu
  - Złożoność liniowa
- `unique_copy(beg, end, dbeg)`, `unique_copy(beg, end, dbeg, op)` - jw. tylko przy kopiowaniu (kombinacja algorytmów `copy+unique`)
- Przykład cpp_13.16

Algorytmy mutujące

- Algorytmy mutujące to takie, które zmieniają kolejność, a nie wartości elementów
  - W stosunku do kontenerów asocjacyjnych uwaga jak dla algorytmów modyfikujących
- Zestawienie wszystkich mutujących algorytmów
  - `reverse()`, `reverse_copy()`, `rotate()`, `rotate_copy()`, `next_permutation()`, `prev_permutation()`, `random_shuffle()`, `partition()`, `stable_partition()`

Odwracanie kolejności

- **`reverse(beg, end)`** - odwraca kolejność elementów w zakresie **`[beg, end)`**
- **`reverse_copy(beg, end, dbeg)`** - odwraca kolejność elementów przy kopiowaniu do zakresu zaczynającego się od **`dbeg`**
  - Złożoność liniowa
- Przykład cpp_13.17

Przesunięcie cykliczne elementów

- **`rotate(beg, newbeg, end)`** - przesuwa elementy cyklicznie tak, aby nowym pierwszym elementem był **`newbeg`**
  - Złożoność liniowa
- **`rotate_copy(beg, newbeg, end, desbeg)`** - jw. tylko kopiuje do zakresu zaczynającego się od **`desbeg`**
  - Zakres źródłowy i docelowy nie powinny się pokrywać
- Przykład cpp_13.18

Permutacje elementów

- `next_permutation(beg, end)`, `prev_permutation(beg, end)` - zwracają odpowiednio następną oraz poprzednią permutację elementów z zakresu `[beg, end)`
  - Zwracają wartość logiczną `false` gdy elementy są posortowane odpowiednio rosnąco i malejąco
    - Dzięki temu można wykonać pełną pętlę permutacji
  - Złożoność liniowa
- `next_permutation(beg, end, op)`, `prev_permutation(beg, end, op)` - jw. tylko predykat `op` użyty jako kryterium sortowania
- Przykład cpp_13.19

Tasowanie elementów

- `random_shuffle(beg, end)`, `random_shuffle(beg, end, op)` - pierwsza tasuje kolejność elementów w zakresie `[beg, end)` wykorzystując generator liczby pseudolosowych o rozkładzie równomiernym, a druga wykorzystuje `op(max)`, która powinna zwracać liczbę losową z przedziału `[0, max)`
  - Złożoność liniowa
  - `op` jest niestałą referencją – nie można podać obiektu tymczasowego i zwykłej funkcji
- Przykład cpp_13.20

Przenoszenie elementów na początek

- `partition(beg, end, op)`, `stable_partition(beg, end, op)` - oba algorytmu przenoszą wszystkie elementy na początek dla których `op(element)` zwraca prawdę. Wersja „stable" pozostawia niezmienioną względną kolejność przenoszonych elementów
  - Zwracają pierwszą pozycję dla której `op(elem)` jest fałszywe
  - Złożoność liniowa
- Przykład cpp_13.21

Algorytmy sortujące

- Algorytmy sortujące są specjalnym rodzajem algorytmów mutujących
  - Sortowanie jest jednak bardziej skomplikowane
  - Wymagają iteratorów dostępu swobodnego
- Zestawienie wszystkich sortujących algorytmów
  - `sort()`, `stable_sort()`, `partial_sort()`, `partial_sort_copy()`, `nth_element()`, `partition()`, `stable_partition()`, `make_heap()`, `push_heap()`, `pop_heap()`, `sort_heap()`

Sortowanie wszystkich elementów

- `sort(beg, end)`, `sort(beg, end, op)` - sortują zakres `[beg, end)` odpowiednio za pomocą operatora `<` i predykatu `op(elem1, elem2)`
  - Złożoność przeciętnie N*log(N)
- `stable_sort(beg, end)`, `stable_sort(beg, end, op)` - jw. ale zachowują wzajemne położenie elementów równych
- Przykład cpp_13.22

Sortowanie częściowe

- `partial_sort(beg, sortEnd, end)`, `partial_sort(beg, sortEnd, end, op)` - sortują zakres `[beg, end)` tak żeby zakres `[beg, sortEnd)` zawierał posortowane elementy zgodnie z operatorem `<` lub gdy predykat `op` zwraca prawdę
  - Złożoność między liniową a N*log(N)
  - Nic nie zwracają
- `partial_sort_copy(beg, end, dbeg, dend)`, `partial_sort_copy(beg, end, dbeg, dend, op)` - jw. tylko kopiują elementy do zakresu `[dbeg, dend)`
  - Zwracają pozycję w zakresie docelowym za ostatnim przekopiowanym elementem
  - Kombinacja algorytmów `copy+partial_sort`
- Przykład cpp_13.23

Sortowanie wg n-tego elementu

- `nth_element(beg, nth, end)`, `nth_element(beg, nth, end, op)` - sortują zakres `[beg, end)` w taki sposób aby prawidłowy element znalazł się na n-tej pozycji, wszystkie elementy przed nim są mniejsze bądź równe, a za nim są elementy większe bądź równe
  - Złożoność przeciętnie liniowa
- Przykład cpp_13.24

Algorytmy stogowe

- Stóg (sterta) można traktować jako drzewo binarne, które zaimplementowano w postaci kolekcji sekwencyjnej
  - Pierwszy element jest największy
  - Elementy można dodawać i usuwać w czasie log
  - Bardzo dobrze nadaje się do tworzenia kolejki priorytetowej
- `make_heap(beg, end)` i `make_heap(beg, end, op)` - tworzy stóg w zakresie `[beg, end)`
- `push_heap(beg, end)` i `push_heap(beg, end, op)` - dodają ostatni element znajdujący się na pozycji przed `end` do stogu
- `pop_heap(beg, end)` i `pop_heap(beg, end, op)` - przenoszą pierwszy element na pozycję przed `end` i z pozostały tworzą stóg
- `sort_heap(beg, end)` i `sort_heap(beg, end, op)` - sortują stóg
- Przykład cpp_13.25

Algorytmy przeznaczone dla zakresów posortowanych

- Algorytmy przeznaczone dla zakresów posortowanych wymagają, zakresy na których operują były posortowane zgodnie z ich kryterium sortowania
- Zestawienie wszystkich takich algorytmów
  - `binary_search()`, `includes()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `merge()`, `set_union()`, `set_intersection()`, `set_difference()`, `set_symmetric_difference()`, `inplace_merge()`

Wyszukiwanie elementów

- `binary_search(beg, end, val)`, `binary_search(beg, end, val, op)` - obie wersje sprawdzają czy w posortowanym zakresie `[beg, end)` występuje wartość `val`
  - Zwraca wartość logiczną
  - Złożoność logarytmiczna
- `includes(beg, end, sbeg, send)`, `includes(beg, end, sbeg, send, op)` - jw. tylko sprawdzają czy wszystkie elementy z posortowanego zakresu `[sbeg, send)` znajdują się w zakresie `[beg, end)`
- Przykład cpp_13.26

Wyszukiwanie pozycji

- `lower_bound(beg, end, val)`, `lower_bound(beg, end, val, op)` - zwracają pozycję pierwszego elementu o wartości `>= val`
- `upper_bound(beg, end, val)`, `upper_bound(beg, end, val, op)` - zwracają pozycję pierwszego elementu o wartości `> val`
- `equal_range(beg, end, val)`, `equal_range(beg, end, val, op)` - zwracają parę odpowiadającą wywołaniu `lower_bound` i `upper_bound`
- Złożoność logarytmiczna
- Przy nie znalezieniu elementu zwracają `end()`
- Przykład cpp_13.27

Scalanie elementów

- `merge(beg1, end1, beg2, end2, destBeg)`, `merge(beg1, end1, beg2, end2, destBeg, op)` - obie postacie algorytmu scalają posortowane zakresy `[beg1, end1)` oraz `[beg2, end2)` tak, aby zakres docelowy zaczynający się od `destBeg` zawierał wszystkie elementy
  - Zakresy źródłowe nie są modyfikowane
  - Zakres docelowy nie powinien się pokrywać ze źródłowymi
  - Zwracają pozycję za ostatnim przekopiowanym elementem
  - Złożoność liniowa

Suma posortowanych zbiorów

- `set_union(beg1, end1, beg2, end2, destBeg)`, `set_union (beg1, end1, beg2, end2, destBeg, op)` - obie postacie algorytmu scalają posortowane zakresy `[beg1, end1)` oraz `[beg2, end2)` tak, aby zakres docelowy zaczynający się od `destBeg` zawierał wszystkie elementy, które występują w pierwszym lub w drugim lub w obu zakresach
  - Zakresy źródłowe nie są modyfikowane
  - Zakres docelowy nie powinien się pokrywać ze źródłowymi
  - Zwracają pozycję za ostatnim przekopiowanym elementem
  - Złożoność liniowa

Iloczyn posortowanych zbiorów

- `set_intersection(beg1, end1, beg2, end2, destBeg)`, `set_intersection(beg1, end1, beg2, end2, destBeg, op)` - obie postacie algorytmu scalają posortowane zakresy `[beg1, end1)` oraz `[beg2, end2)` tak, aby zakres docelowy zaczynający się od `destBeg` zawierał wszystkie elementy występujące jednocześnie w obu zakresach
  - Zakresy źródłowe nie są modyfikowane
  - Zakres docelowy nie powinien się pokrywać ze źródłowymi
  - Zwracają pozycję za ostatnim przekopiowanym elementem
  - Złożoność liniowa

Różnica posortowanych zbiorów

- `set_difference(beg1, end1, beg2, end2, destBeg)`, `set_difference(beg1, end1, beg2, end2, destBeg, op)` - obie postacie algorytmu scalają posortowane zakresy `[beg1, end1)` oraz `[beg2, end2)` tak, aby zakres docelowy zaczynający się od `destBeg` zawierał wszystkie elementy występujące w zakresie pierwszym i jednocześnie nie występujące w drugim zakresie
  - Zakresy źródłowe nie są modyfikowane
  - Zakres docelowy nie powinien się pokrywać ze źródłowymi
  - Zwracają pozycję za ostatnim przekopiowanym elementem
  - Złożoność liniowa

Różnica symetryczna posortowanych zbiorów

- `set_symmetric_difference(beg1, end1, beg2, end2, destBeg)`, `set_symmetric_difference(beg1, end1, beg2, end2, destBeg, op)` - obie postacie algorytmu scalają posortowane zakresy `[beg1, end1)` oraz `[beg2, end2)` tak, aby zakres docelowy zaczynający się od `destBeg` zawierał wszystkie elementy występujące albo w zakresie pierwszym albo drugim, lecz nie jednocześnie w obu
  - Zakresy źródłowe nie są modyfikowane
  - Zakres docelowy nie powinien się pokrywać ze źródłowymi
  - Zwracają pozycję za ostatnim przekopiowanym elementem
  - Złożoność liniowa
- Przykład cpp_13.28 i cpp_13.29

Scalanie sąsiednich posortowanych zakresów

- `inplace_merge(beg1, end1beg2, end2)`, `inplace_merge(beg1, end1beg2, end2, op)` – obie postacie algorytmu scalają posortowane zakresy `[beg1, end1beg2)` oraz `[end1beg2, end2)` tak, aby zakres `[beg1, end2)` zawierał wszystkie elementy w postaci posortowanego zakresu sumarycznego
  - Nic nie zwraca
  - Złożoność liniowa
- Przykład cpp_13.30

Algorytmy numeryczne

- Algorytmy te wykonują różne operacje na kombinacjach elementów numerycznych
  - Algorytmy te dają większe możliwości niżby to wynikało z ich nazwy
- Algorytmy te wymagają dołączenia nagłówka `<numeric>`
- Zestawienie wszystkich numerycznych algorytmów
  - `accumulate(), inner_product(), adjacent_difference(), partial_sum()`

Obliczanie wartości

- `accumulate(beg, end, val)` - zwraca sumę wszystkich elementów z `[beg, end)` dodając wartość `val`
  - `wynik = val + a1 + a2 + a3 ... (val=val+elem)`
  - Złożoność liniowa
- `accumulate(beg, end, val, op)` - zwraca wynik wywołania operacji `op(val, elem)` dla wszystkich elementów z `[beg, end)`
  - `wynik = val op a1 op a2... (val=op(val, elem))`
  - Jeżeli zbiór jest pusty zwraca `val`
- Przykład cpp_13.31

Iloczyn skalarny

- `inner_product(beg1, end1, beg2, val)` - zwraca iloczyn skalarny wartości `val` oraz wszystkich elementów z zakresu `[beg1, end1)` w kombinacji z elementami z zakresu rozpoczynającego się od `beg2`
  - `wynik = val + (a1*b1) + (a2*b2)...`
    `(val = val + elem1*elem2)`
- `inner_product(beg1, end1, beg2, val, op1, op2)` - zwraca wynik operacji op1 wobec wartości `val` oraz `op2` wszystkich elementów z zakresu `[beg1, end1)` w kombinacji z elementami z zakresu rozpoczynającego się od `beg2`
  - `wynik = val op1 (a1 op2 b1) op1 (a2 op2 b2)...`
    `(val = op1(val, op(elem1, elem2)))`
  - Jeśli zbiór jest pusty zwracana jest `val`
  - Złożoność liniowa
- Przykład cpp_13.32

Przekształcanie wartości względnych i bezwzględnych

- `partial_sum(beg, end, dbeg)` – oblicza sumę częściową dla każdego elementu z zakresu `[beg, end)` i wynik zapisuje do zakresu zaczynającego się od `dbeg`
  - `a1, a1+a2, a1+a2+a3...`
  - Zwraca pozycję za ostatnią zapisaną wartością w zakresie docelowym
  - Zakres źródłowy i docelowy mogą być równe
  - Złożoność liniowa
- `partial_sum(beg, end, dbeg, op)` – jw. tylko nie sumuje, a wywołuje operacje `op`
  - `a1, a1 op a2, a1 op a2 op a3,...`
- `adjacent_difference(beg, end, dbeg)` – odwraca działanie `partial_sum(beg, end, dbeg)`
  - `a1, a2-a1, a3-a2, a4-a3,...`
- `adjacent_difference(beg, end, dbeg, op)` – wywołuje operacje `op` wobec każdego elementu z zakresu `[beg, end)` i jego poprzednika
  - `a1, a2 op a1, a3 op a2, a4 op a3,...`
- Przykład cpp_13.33

Algorytmy uwagi

- Algorytmy pracują na zakresie (-ach) zdefiniowanych przez iteratory
  - Jeżeli używamy iteratorów zwykłych to konsekwentnie wszędzie i jeżeli otrzymujemy rezultat w postaci iteratora też jest on zwykły
  - Podobnie jeżeli używamy iteratorów odwrotnych – wtedy wszystkie użyte iteratory w algorytmie muszą być iteratmorami odwrotnymi i jeżeli algorytm zwraca pozycję to w postaci iteratora odwrotnego
    - Można zawsze skonwertować do zwykłego metodą `base()`
- Algorytmy również mogą pracować na zwykłych tablicach
  - Przykład cpp_13.34