Podstawy Programowania
Semestr letni 2022/23
Materiały z laboratorium i zadania domowe

Przemysław Olbratowski

30 marca 2023

Slajdy z wykładu są dostępne w serwisie UBI. Informacje organizacyjne oraz formularz do uploadu prac domowych znajdują się na stronie `info.wsisiz.edu.pl/~olbratow`. Przy zadaniach domowych w nawiasach są podane terminy sprawdzeń.

5.2 Zadania domowe z działu Wektory (5, 26 kwietnia, 10 maja)

5.2.1 Backward: Wydruk w odwrotnej kolejności

Napisz program `backward`, który czyta ze standardowego wejścia liczby rzeczywiste do napotkania końca pliku i wypisuje je na standardowe wyjście w odwrotnej kolejności. Program załącza tylko pliki nagłówkowe `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 3.1 2.7 -0.5 0.1 4.3
Out: 4.3 0.1 -0.5 2.7 3.1
```

5.2.2 Count: Zliczanie elementów

Napisz funkcję `count`, która przyjmuje stałą referencję wektora liczb całkowitych oraz pojedynczą wartość całkowitą i zwraca liczbę wystąpień tej wartości w zadanym wektorze. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

```
int main() {
    const std::vector<int> vector {3, 7, -1, 12, -5, 7, 10};
    int result = count(vector , 7);
    std::cout << result << std::endl; }
```

Wykonanie

```
Out: 2
```

5.2.3 Count Sort: Sortowanie przez zliczanie

Wektor zawierający tylko liczby całkowite od zera włącznie do k wyłącznie najszybciej jest posortować metodą zliczania. Zliczamy, ile w sortowanym wektorze jest zer, jedynek i tak dalej. Następnie, sortując w kolejności niemalejącej, zaczynamy od początku wektora i wpisujemy do kolejnych komórek tyle zer, ile zliczyliśmy, potem tyle jedynek, ile zliczyliśmy, i tak dalej. Napisz funkcję `count_sort`, która przyjmuje modyfikującą referencję wektora liczb całkowitych oraz liczbę k i sortuje ten wektor metodą zliczania w kolejności niemalejącej. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

```
int main() {
    std::vector<int> vector {4, 2, 7, 5, 2, 1, 5};
    count_sort(vector, 10);
    for (int element: vector) {
        std::cout << element << " "; }
    std::cout << std::endl; }
```

Wykonanie

```
Out: 1 2 2 4 5 5 7
```

5.2.4 Find: Wyszukiwanie elementu - grupowo

Napisz funkcję `find`, która przyjmuje stałą referencję wektora liczb całkowitych oraz pojedynczą wartość całkowitą i zwraca indeks pierwszego wystąpienia tej wartości w wektorze albo długość wektora, jeżeli ta wartość w nim nie występuje. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

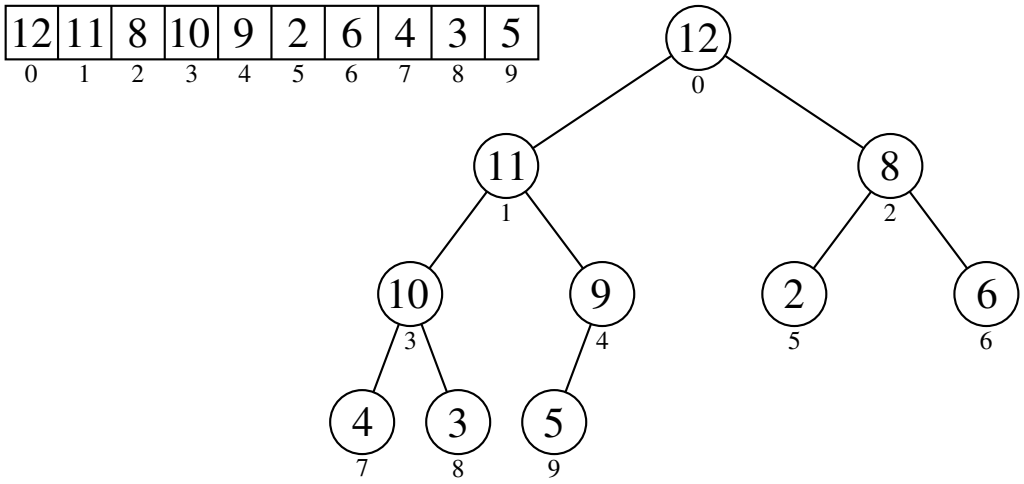
```
int main() {
    int result = find(std::vector<int> {3, -1, 7, 12, -5, 7, 10}, 7);
    std::cout << result << std::endl; }
```

Wykonanie

```
Out: 2
```

5.2.5 Heap Sort: Sortowanie kopcowe

Zupełny kopiec binarny to rodzaj drzewa przedstawiony na rysunku. Każdy węzeł jest mniejszy lub równy swojemu rodzicowi. Węzły dodajemy do kopca w kolejności wyznaczonej małymi cyframi, które są jednocześnie numerami węzłów. Kopiec wygodnie jest przechowywać w wektorze, jak zaznaczono na rysunku. Wartość każdego węzła przechowujemy w komórce wektora o indeksie równym numerowi węzła.



Kopcowe sortowanie wektora w kolejności niemalejącej odbywa się w dwóch etapach. W pierwszym wyjmujemy z wektora kolejne liczby od lewej do prawej i budujemy z nich kopiec. Każdą liczbę dodajemy początkowo jako ostatni węzeł kopca według opisanej kolejności. Może się jednak zdarzyć, że tak dodany węzeł jest większy od swojego rodzica. W takiej sytuacji zamieniamy go z rodzicem miejscami. Czynność tę powtarzamy aż przestawiany węzeł okaże się mniejszy lub równy nowemu rodzicowi bądź zawędruje na samą górę kopca stając się jego nowym korzeniem. W drugim etapie wyjmujemy z kopca kolejne liczby od największej do najmniejszej i wpisujemy je do wektora od końca. Największym węzłem kopca jest oczywiście jego korzeń, więc jego wyjmujemy najpierw. Następnie odbudowujemy naruszoną w ten sposób strukturę kopca. Na miejsce korzenia wstawiamy ostatni węzeł kopca. Prawdopobnie okaże się on mniejszy od swoich nowych dzieci. W takiej sytuacji zamieniamy go miejscami z większym dzieckiem. Czynność tę powtarzamy aż przestawiany węzeł okaże się mniejszy lub równy nowegu rodzicowi. Wszystkie te operacje wykonujemy bezpośrednio w sortowanym wektorze, który jednocześnie przechowuje kopiec. W pierwszym etapie kopiec rozbudowuje się od lewej przejmując kolejne elementy wektora, zaś w drugim kurczy się pozostawiając po prawej elementy posortowane. Napisz program `heap_sort`, który czyta ze standardowego wejścia liczby całkowite do napotkania końca pliku i sortuje je kopcowo w kolejności niemalejącej. Program wypisuje na standardowe wyjście w kolejnych liniach sortowane liczby po każdej zamianie. Program załącza tylko pliki nagłówkowe `iostream`, `utility` i `vector`.

Przykładowe wykonanie

```
 In: 3 7 -1 12 -5 7 10
```

```
Out: 7 3 -1 12 -5 7 10
Out: 7 12 -1 3 -5 7 10
Out: 12 7 -1 3 -5 7 10
Out: 12 7 7 3 -5 -1 10
Out: 12 7 10 3 -5 -1 7

Out: 7 7 10 3 -5 -1 12
Out: 10 7 7 3 -5 -1 12
Out: -1 7 7 3 -5 10 12
Out: 7 -1 7 3 -5 10 12
Out: 7 3 7 -1 -5 10 12
Out: -5 3 7 -1 7 10 12
Out: 7 3 -5 -1 7 10 12
Out: -1 3 -5 7 7 10 12
Out: 3 -1 -5 7 7 10 12
Out: -5 -1 3 7 7 10 12
Out: -1 -5 3 7 7 10 12
Out: -5 -1 3 7 7 10 12
```

5.2.6 Intersection: Część wspólna zbiorów

Napisz funkcję `intersection`, któa przyjmuje stałe referencje dwóch uporządkowanych rosnąco wektorów liczb całkowitych i zwraca uporządkowany rosnąco wektor liczb zawartych w obu tych wektorach jednocześnie. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

```
int main() {
    const std::vector<int> vector1 {-7, 2, 3, 7, 15, 18, 23},
                           vector2 {-8, 3, 5, 8, 15, 23, 30};
    std::vector<int> result = intersection(vector1, vector2);
    for (int element: result) {
        std::cout << element << " "; }
    std::cout << std::endl; }
```

Wykonanie

```
Out: 3 15 23
```

5.2.7 Longest Slope: Najdłuższy przedział niemalejący

Napisz program `longest_slope`, który czyta ze standardowego wejścia liczby całkowite do napotkania końca pliku i wypisuje na standardowe wyjście najdłuższy niemalejący podciąg tych liczb. Jeżeli takich podciągów jest kilka, program wypisuje najwcześniej się zaczynający. Zaproponuj algorytm o złożoności liniowej. Program załącza tylko pliki nagłówkowe `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 10 10 2 4 7 13 8 2 0 17 7 0 1 12 18 18 17 6 5 19
Out: 0 1 12 18 18
```

5.2.8 Maximum Sum: Przedział o największej sumie elementów

Napisz program `maximum_sum`, który czyta ze standardowego wejścia liczby całkowite do napotkania końca pliku i wypisuje na standardowe wyjście podciąg tych liczb o największej dodatniej sumie elementów. Jeżeli takich podciągów jest kilka, program wypisuje najkrótszy i najwcześniej się zaczynający. Jeżeli taki podciąg nie istnieje, program nic nie wypisuje. Zaproponuj algorytm o złożoności liniowej. Program załącza tylko pliki nagłówkowe `iostream` i `vector`.

Przykładowe wykonanie

```
 In: -5 0 -3 -2 -3 -3 -3 0 0 -2 -2 3 4 -1 4 5 -4 1 3 -2
Out: 3 4 -1 4 5
```

5.2.9 Min Element: Element najmniejszy

Napisz funkcję `min_element`, która przyjmuje stałą referencję wektora liczb całkowitych i zwraca indeks najmniejszego elementu. Jeżeli takich elementów jest kilka, funkcja zwraca indeks pierwszego z nich. Jeżeli taki element nie istnieje, funkcja zwraca długość wektora. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

```
int main() {
    const std::vector<int> vector {9, -7, 5, 1, 12, 3, -7};
    int result = min_element(vector);
    std::cout << result << std::endl; }
```

Wykonanie

```
Out: 1
```

5.2.10 Partial Sum: Sumy częściowe

Napisz funkcję `partial_sum`, która przyjmuje modyfikującą referencję wektora liczb rzeczywistych i do każdego jego elementu dodaje sumę wszystkich go poprzedzających. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

```
int main() {
    std::vector<double> vector {3.1, 2.7, -0.5, 0.1, 4.3};
    partial_sum(vector);
    for (double element: vector) {
        std::cout << element << " "; }
    std::cout << std::endl; }
```

Wykonanie

```
Out: 3.1 5.8 5.3 5.4 9.7
```

5.2.11 Relay: Przesiadka

Kierowca planuje podróż z miasta początkowego przez kilka pośrednich do miasta końcowego. Trasa jest na tyle długa, że wymaga po drodze jednego noclegu. Kierowca chce tak wybrać miasto na nocleg, aby kilometraże pierwszego i drugiego dnia podróży jak najmniej się różniły. Napisz program `relay`, który czyta ze standardowego wejścia odległości między kolejnymi miastami do napotkania końca pliku i wypisuje na standardowe wyjście numer miasta, w którym wypada nocleg, przy czym miasto początkowe ma numer zero. Program załącza tylko pliki nagłówkowe `cmath`, `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 15.2 23.1 2.5 7.3 11 5.3
Out: 2
```

5.2.12 Rotate: Cykliczne przesunięcie wektora

Cykliczne przesunięcie wektora o jedną pozycję w lewo polega na przesunięciu elementów o indeksach większych od zera o jedną pozycję w lewo i przestawieniu oryginalnego elementu o indeksie zero na koniec. Napisz funkcję `rotate`, która przyjmuje modyfikującą referencję wektora liczb rzeczywistych oraz nieujemną liczbę całkowitą n i przesuwa elementy wektora cyklicznie o n pozycji w lewo. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Zaproponuj algorytm lepszy od n-krotnego przesunięcia o jedną pozycję. Funkcja korzysta tylko z plików nagłówkowych `utility` i `vector`.

Przykładowy program

```
int main() {
    std::vector<double> vector {2.5, 0, 12, 3,7, -2.8, 0.1};
    rotate(vector, 2);
    for (double element: vector) {
        std::cout << element << " "; }
    std::cout << std::endl; }
```

Wykonanie

```
Out: 12 3 7 -2.8 0.1 2.5 0
```

5.2.13 Selection Sort: Sortowanie przez wybór - indywidualnie

Sortowanie wektora w kolejności niemalejącej przez wybór przebiega następująco. Znajdujemy w wektorze pierwszy element najmniejszy i zamieniamy go z pierwszym elementem wektora. Następnie powtarzamy te czynności dla wektora bez pierwszego elementu i tak dalej. Napisz program `selection_sort`, który czyta ze standardowego wejścia liczby całkowite do napotkania końca pliku i sortuje je niemalejąco przez wybór. Program wypisuje na standardowe wyjście w kolejnych liniach sortowane liczby po każdej zamianie. Program załącza tylko pliki nagłówkowe `iostream`, `utility` i `vector`.

Przykładowe wykonanie

```
 In: 3 7 -1 12 -5 7 10
Out: -5 7 -1 12 3 7 10
Out: -5 -1 7 12 3 7 10
Out: -5 -1 3 12 7 7 10
Out: -5 -1 3 7 12 7 10
Out: -5 -1 3 7 7 12 10
Out: -5 -1 3 7 7 10 12
Out: -5 -1 3 7 7 10 12
```

5.2.14 Shuffle: Tasowanie elementów

Napisz funkcję `shuffle`, która przyjmuje modyfikującą referencję wektora liczb rzeczywistych i losowo przestawia jego elementy. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z plików nagłówkowych `cstdlib`, `utility` i `vector`.

Przykładowy program

```
int main() {
    std::srand(std::time(nullptr));
    std::vector<double> vector {7.2, -1, 12.3, 3, 10.5};
    shuffle(vector);
    for (double element: vector) {
        std::cout << element << " "; }
    std::cout << std::endl; }
```

Wykonanie

```
Out: 10.5 12.3 7.2 3 -1
```

5.2.15 Triangles: Liczba trójkątów

Napisz program `triangles`, który czyta ze standardowego wejścia podane w kolejności niemalejącej długości odcinków aż do napotkania końca pliku i wypisuje na standardowe wyjście liczbę trójkątów, które można z nich zbudować. Każdego odcinka można użyć w jednym trójkącie tylko raz a dwa odcinki o jednakowych długościach uważamy za różne. Zaproponuj algorytm o złożoności kwadratowej. Program załącza tylko pliki nagłówkowe `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 1 2.5 3 3.7
Out: 3
```

5.2.16 Upper Bound: Wyszukiwanie binarne

Chcąc znaleźć daną wartość w wektorze posortowanym niemalejąco można zastosować wyszukiwanie binarne. Rozważamy najpierw element leżący w połowie wektora. Jeżeli jest on większy od poszukiwanej wartości, to na pewno leży ona w lewej połowie wektora, więc do niej ograniczamy dalsze poszukiwania. W przeciwnym razie prowadzimy je tylko w prawej połowie. Wybraną połowę znów dzielimy na pół i tak dalej. Napisz funkcję `upper_bound`, która przyjmuje stałą referencję posortowanego niemalejąco wektora liczb rzeczywistych oraz pojedynczą wartość rzeczywistą i zwraca indeks pierwszego elementu większego od podanej wartości. Jeżeli taki element nie istnieje, funkcja zwraca długość wektora. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja korzysta tylko z pliku nagłówkowego `vector`.

Przykładowy program

```
int main() {
    const std::vector<double> vector {-4.3, -1.2, 0.1, 8.2, 11.8};
    int result = upper_bound(vector , 7.6);
    std::cout << result << std::endl; }
```

Wykonanie

```
Out: 3
```