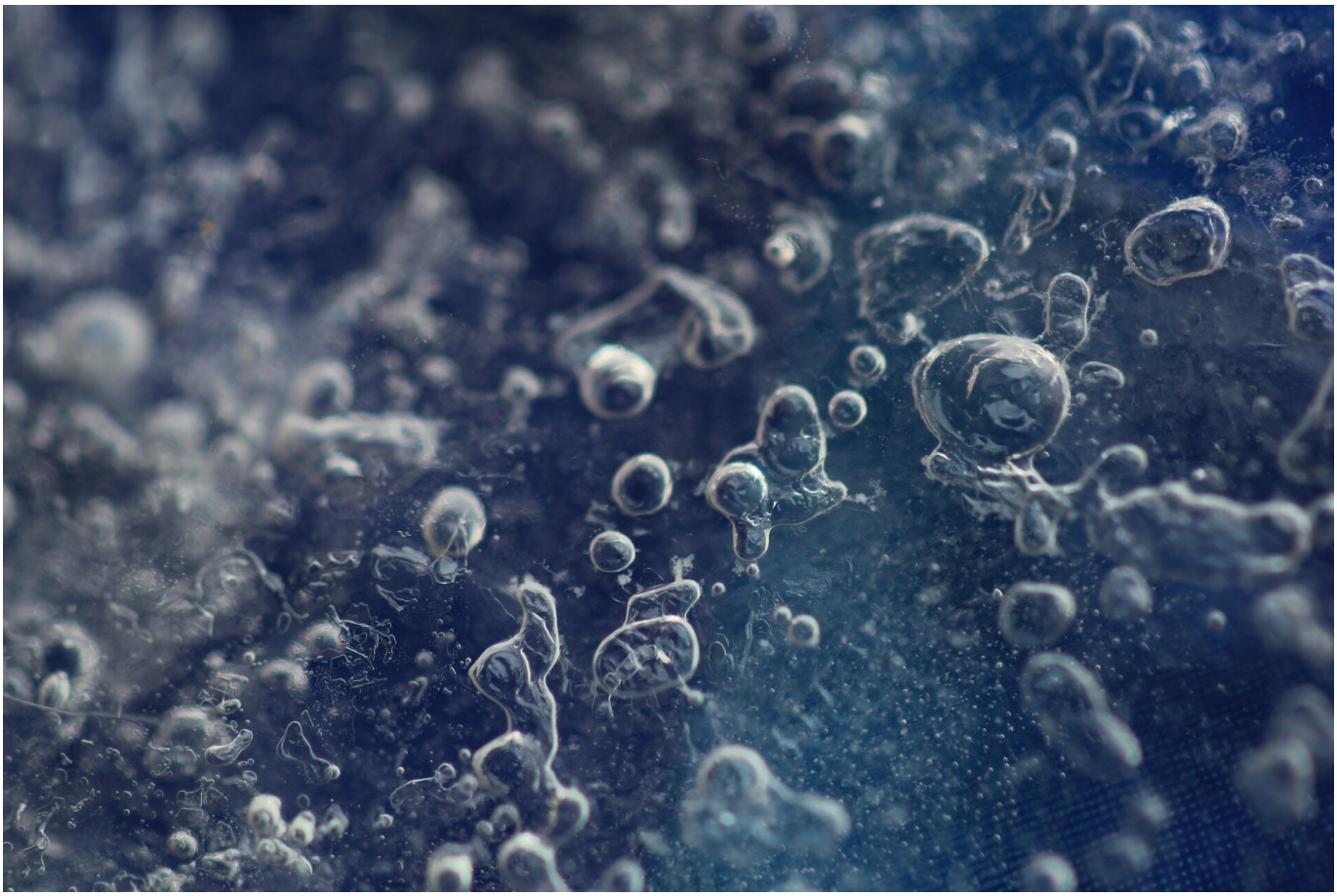


Sortowanie bąbelkowe

Źródło: Eliška Motisová, domena publiczna.

Jednym z najdłużej używanych algorytmów jest algorytm sortowania bąbelkowego. Wynika to prawdopodobnie z prostoty implementacji i przejrzystej składni. Jest to jeden z wielu algorytmów sortowania, o których możesz dowiedzieć się więcej z e-materiału: Wstęp do algorytmów sortowania.

Sortowanie bąbelkowe jest algorytmem liniowym bazującym na cyklicznym porównywaniu i zamianie kolejności par sąsiadujących ze sobą elementów. To algorytm stabilny, pracujący w miejscu. Oznacza to, że dla elementów o tej samej wartości zachowuje ich początkową kolejność i nie potrzebuje do swojego działania większej niż stała pamięci dodatkowej (poza pamięcią, w której przechowywane są dane wejściowe).

Algorytm sortowania bąbelkowego uznawany jest za rozwinięcie algorytmu sortowania naiwnego, które polega na badaniu kolejnych par elementów i zamianie ich miejscami, jeśli umieszczone są w złej kolejności, po czym cała operacja zaczyna się od początku. W tym e-materiale dowiemy się, na czym polega sortowanie bąbelkowe, jak je zaimplementować i jakie są jego złożoności.

Implementację algorytmu sortowania bąbelkowego w różnych językach programowania przedstawiamy w e-materiałach:

- Sortowanie bąbelkowe w języku C++,

- Sortowanie bąbelkowe w języku Java,
- Sortowanie bąbelkowe w języku Python.

Więcej zadań? Przejdź do: Sortowanie babelkowe – zadania maturalne.

> **Twoje cele**
>
> - Przeanalizujesz działanie algorytmu sortowania bąbelkowego.
> - Prześledzisz, w jaki sposób można usprawnić algorytm sortowania bąbelkowego.
> - Zbadasz złożoność czasową oraz pamięciową algorytmu sortowania bąbelkowego.
> - Posortujesz przykładowy ciąg liczbowy za pomocą sortowania bąbelkowego.

Przeczytaj

Sortowanie bąbelkowe

Sortowanie bąbelkowe to jeden ze sposobów sortowania serii danych, które można zaimplementować w dowolnym języku programowania. W kolejnych cyklach działania algorytmu największe wartości zostają przesunięte w prawo, na koniec serii danych. Operacja ta powtarza się, aż wszystkie wartości zostaną ułożone w zadanej kolejności, tzn. nierosnąco lub niemalejąco. Po przeanalizowaniu złożoności tego algorytmu dowiesz się, że prostota jego zapisu nie idzie jednak w parze z szybkością działania, dlatego najlepiej używać go do sortowania niewielkich zestawów danych.

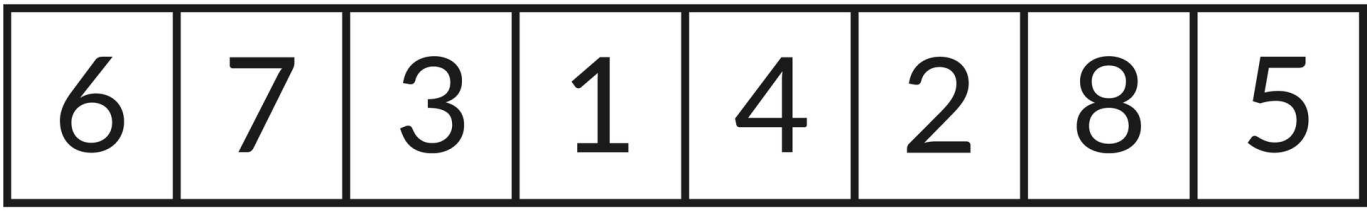
> **Przykład 1**
>
> Przedstaw działanie algorytmu sortowania bąbelkowego na nieuporządkowanym ciągu liczb. Dane powinny zostać posortowane niemalejąco. Ciąg do posortowania zapisany został w tablicy w następującej postaci: [6, 7, 3, 1, 4, 2, 8, 5].
>
> **Specyfikacja problemu:**
>
> *Dane*:
>
> - `n` – liczba elementów ciągu do posortowania (rozmiar tablicy)
> - `tablica[]` – tablica z nieposortowanym ciągiem wybranych liczb naturalnych
>
> *Wynik*:
>
> - `tablica[]` – ciąg liczb przechowywany w tablicy, posortowany niemalejąco

Przykład realizacji algorytmu

Nauczyciel informatyki poprosił chętnych o zgłoszenie się do udziału w konkursie programowania. Uczniowie wpisywali się na listę, podając swój numer z dziennika szkolnego. Lista zawierała następujące liczby: 6, 7, 3, 1, 4, 2, 8, 5.

Naszym zadaniem jest posortowanie listy chętnych rosnąco. Użyjemy do tego celu sortowania bąbelkowego.

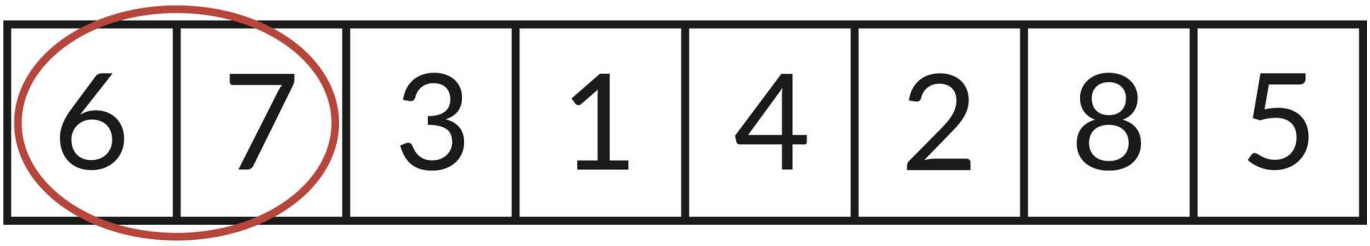
Zaczynamy od zapisania numerów w odpowiedniej strukturze danych, np. w tablicy lub liście.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Krok 1

Porównujemy ze sobą dwie pierwsze liczby – czyli 6 i 7.

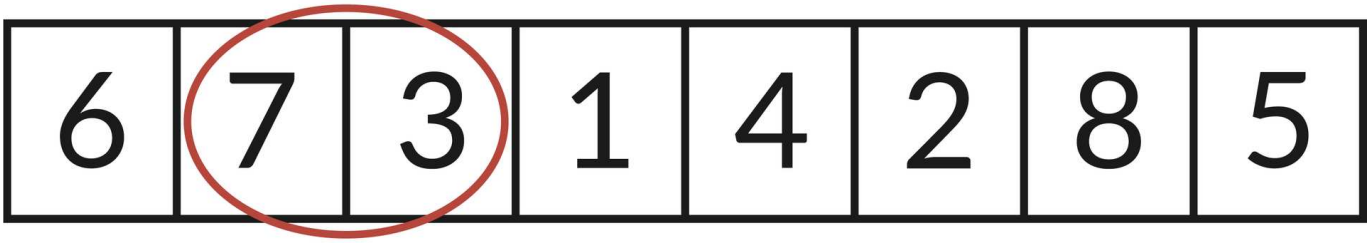


Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Liczba 6 jest mniejsza od 7, więc nie następuje zamiana.

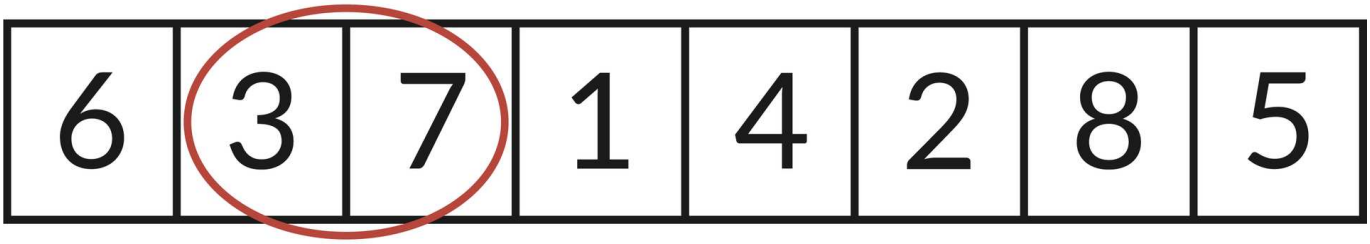
Krok 2

Teraz porównujemy liczby 7 i 3.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

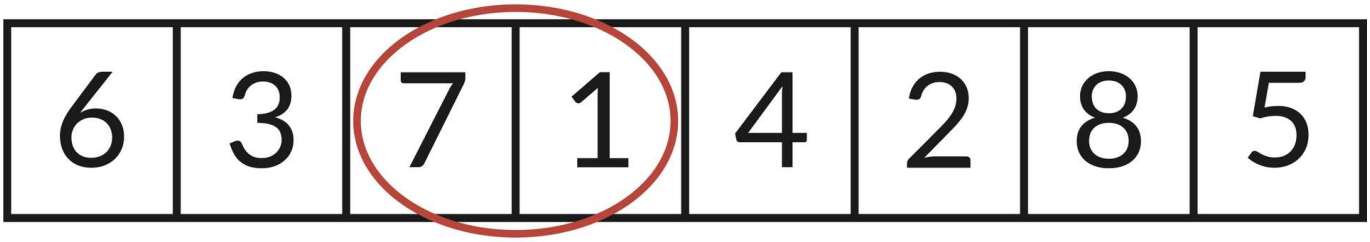
Liczba 7 jest większa od 3, więc następuje zamiana.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

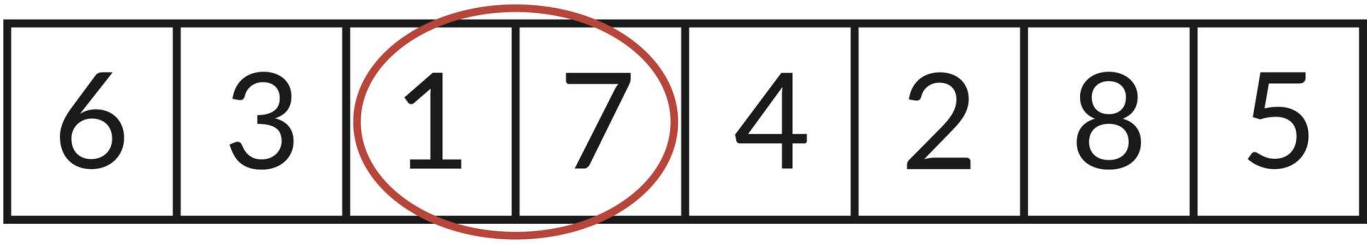
Krok 3

Następnie porównujemy liczby 7 i 1.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

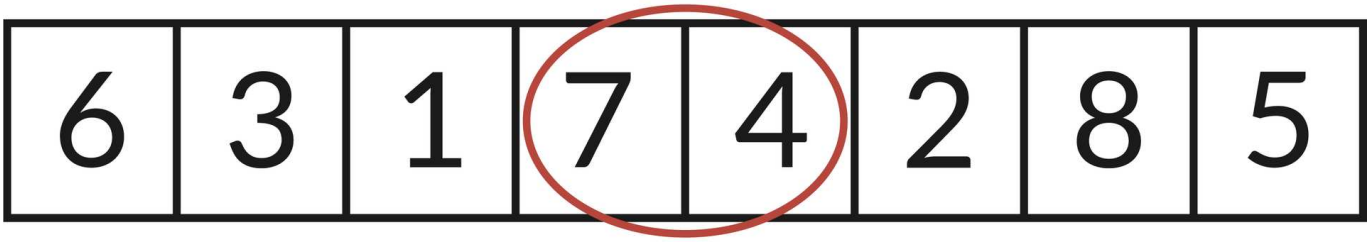
Liczba 7 jest większa od 1, więc następuje zamiana.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

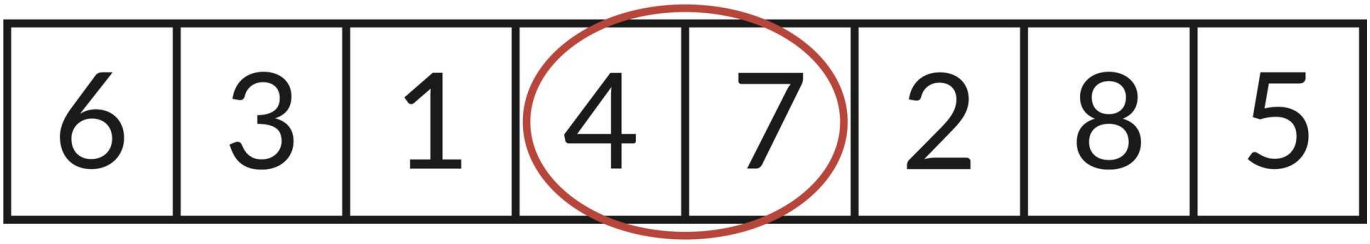
Krok 4

Dalej porównujemy liczby 7 i 4.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

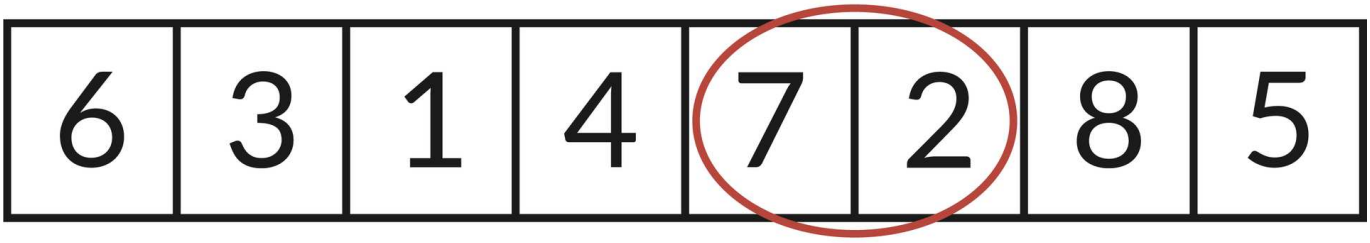
Liczba 7 jest większa od 4, więc następuje zamiana.



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Krok 5

Teraz porównujemy liczby 7 i 2.

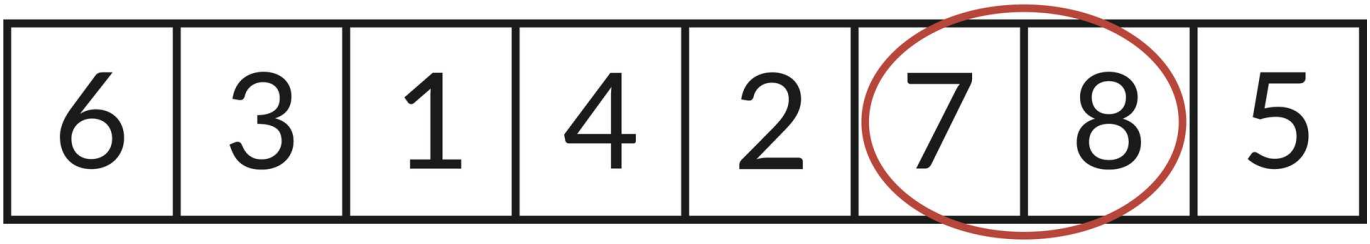


Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Liczba 7 jest większa od 2, więc następuje zamiana.

Krok 6

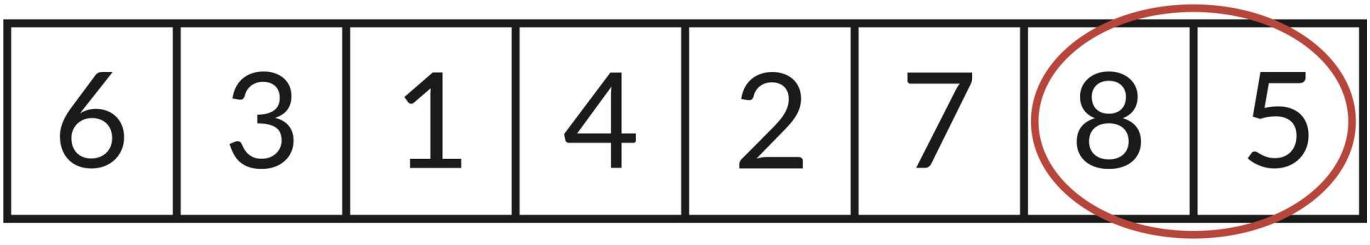
Następnie porównujemy liczby 7 i 8.



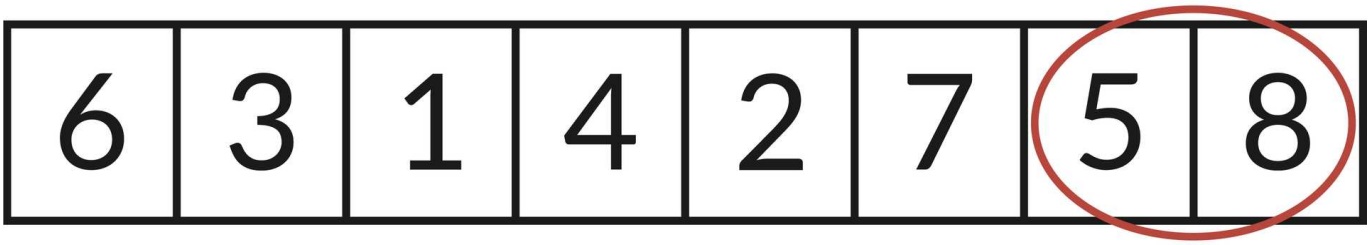
Liczba 7 jest mniejsza od 8, więc nie następuje zamiana.

Krok 7

Na koniec porównujemy dwie ostatnie liczby: 8 i 5.



Liczba 8 jest większa od 5, więc następuje zamiana.



Po zakończeniu ostatniego kroku liczba 8, czyli największa liczba w rozważanym ciągu, znajdzie się na ostatnim miejscu w tablicy, a więc na swoim miejscu docelowym. Ponieważ pozostałe elementy tablicy (poza liczbą 8) wciąż nie są posortowane, należy wrócić do początku tablicy i ponownie wykonywać po kolei wszystkie porównania.

W każdym kolejnym cyklu przeglądany ciąg będzie krótszy, dlatego w poszczególnych cyklach zostanie wykonane $n - i$ porównań, gdzie:

- n – rozmiar tablicy, która ma zostać posortowana,
- i – kolejny numer wykonywanego cyklu.

Po każdym cyklu algorytmu największy element obecnie analizowanego ciągu zostanie umieszczony na końcu, co oznacza, że w każdym przebiegu kolejna liczba znajdzie się na prawidłowej pozycji (elementy ciągu nierozważane w trakcie obiegu są już posortowane). Algorytm powinien wykonywać się do momentu, gdy wszystkie wartości zostaną posortowane.

Poniżej zostały przedstawione dwie wersje algorytmu sortowania bąbelkowego zapisane przy użyciu pseudokodu: podstawowa (nieuwzględniająca wcześniejszego zakończenia wykonywania algorytmu) i usprawniona (mogąca zakończyć się wcześniej w przypadku posortowania ciągu).

Pseudokod algorytmu sortowania bąbelkowego

Proces sortowania metodą bąbelkową można zapisać za pomocą następującego pseudokodu:

```
Sortowanie_babelkowe(tablica, n)
    Dla i = 0, 1, 2, ... n - 2, wykonuj
        Dla j = 0, 1, 2, ... n - i - 2, wykonuj
            Jeżeli tablica[j] > tablica[j + 1], to
                pomocnicza ← tablica[j + 1]
                tablica[j + 1] ← tablica[j]
                tablica[j] ← pomocnicza
```

- Pętla zewnętrzna odpowiada za wykonanie kolejnych cykli sortowania. Pętla ta zostanie wykonana $n - 1$ razy, gdzie n to liczba elementów sortowanego ciągu. W ostatnim cyklu sortowania elementy ciągu będą już posortowane.
- W pętli wewnętrznej porównywane są kolejne wartości, począwszy od lewej strony (mniejszych indeksów tablicy). Pętla ta zaczyna się od indeksu 0 (pierwszego elementu sortowanej tablicy) i kończy na indeksie $n - i - 2$ (przedostatnim elemencie nieposortowanej jeszcze części tablicy). Jest to spowodowane tym, że porównujemy obecny i następny element tablicy. W ograniczeniu pętli odejmujemy 2 od wartości $n - i$, żeby pętla wewnętrzna mogła się wykonać we wszystkich iteracjach pętli zewnętrznej (dla $j = n - i - 2$ zawartość również się wykona, w wyniku czego otrzymamy $n - i - 1$ wykonań kodu zawartego w pętli wewnętrznej, na obieg pętli zewnętrznej). Następnie w pętli wewnętrznej porównywane są kolejne pary

elementów. Jeśli poprzedni element jest większy od następującego po nim, zostaje wykonana zamiana.

Ważne!

Algorytm można usprawnić, dodając do niego flagę, która będzie informowała, czy w danym cyklu dokonała się choć jedna zamiana. Jeżeli flaga poinformuje nas, że nie została dokonana żadna zmiana, będzie to sygnał, że ciąg jest już posortowany i możemy przerwać dalsze wykonywanie algorytmu. Przekłada się to na skrócenie jego czasu działania.

```
Sortowanie_babelkowe(tablica, n)
  Dla i = 0, 1, 2, ... n - 2, wykonuj
    flaga ← fałsz
    Dla j = 0, 1, 2, ... n - i - 2, wykonuj
        Jeżeli tablica[j] > tablica[j + 1], to
            pomocnicza ← tablica[j + 1]
            tablica[j + 1] ← tablica[j]
            tablica[j] ← pomocnicza
            flaga ← prawda
    Jeżeli flaga = fałsz
        zakończ wykonywanie pętli
```

Do przedstawionego wcześniej algorytmu dodaliśmy flagę, która na początku każdego obiegu pętli zewnętrznej przyjmuje wartość `fałsz`. Ustawiając taką wartość flagi, zakładamy, że ciąg jest już posortowany, a następnie sprawdzamy prawdziwość tego stwierdzenia. Jeżeli okaże się, że elementy ciągu nie są posortowane i istnieje potrzeba ich zamiany miejscami, zmieniamy wartość flagi na `prawda`. Sygnalizujemy w ten sposób, że w tym obiegu pętli zewnętrznej została zmieniona kolejność elementów ciągu. Jeżeli po przejściu całej pętli wewnętrznej domyślna wartość flagi nie zmieniła się, oznacza to, że elementy ciągu są posortowane i możemy zakończyć wykonywanie algorytmu.

Dla zainteresowanych

Omówione zastosowanie sortowania bąbelkowego bazujące na porównywaniu wybranego elementu z elementem występującym za nim nie jest jedyną możliwą implementacją tego algorytmu.

Innym rozwiązaniem jest porównanie w obiegu pętli wewnętrznej wybranego elementu z elementem występującym przed nim. Wymaga to jednak wprowadzenia pewnych zmian w stworzonym wcześniej kodzie.

Jedną z wymaganych modyfikacji jest zmiana indeksów, na których operuje pętla wewnętrzna. Ponieważ porównujemy element pod indeksem wyznaczonym przez

iterator pętli, to powinna się ona zaczynać od wartości 1, by wskazywać na drugi z sortowanych elementów (żeby możliwe było porównanie go z pierwszym).

Kolejną modyfikacją jest zmiana indeksów elementów we fragmencie kodu odpowiedzialnym za porównanie i zamianę elementów miejscami. W tej implementacji algorytmu sortowania bąbelkowego porównujemy element wyznaczony przez iterator pętli z elementem go poprzedzającym. Jeśli warunek zostanie spełniony, zamieniamy ze sobą porównywane elementy, używając elementu pomocniczego w sposób analogiczny do zastosowanego we wcześniejszej implementacji. Reszta mechanizmów algorytmu pozostaje bez zmian.

```
Sortowanie_babelkowe(tablica, n)
  Dla i = 0, 1, 2, ... n - 2, wykonuj
    flaga ← fałsz
    Dla j = 1, 2, 3 ... n - i - 1, wykonuj
        Jeżeli tablica[j - 1] > tablica[j], to
            pomocnicza ← tablica[j - 1]
            tablica[j - 1] ← tablica[j]
            tablica[j] ← pomocnicza
            flaga ← prawda
    Jeżeli flaga = fałsz
        zakończ wykonywanie pętli
```

Złożoność czasowa i pamięciowa algorytmu

W rozważanym algorytmie operacją dominującą (elementarną) jest porównanie wartości sąsiednich elementów.

Złożoność czasowa przypadku pesymistycznego

Przypadek pesymistyczny to najgorszy z punktu widzenia programu zestaw danych, czyli taki, dla którego program wykona najwięcej porównań.

W przypadku pesymistycznym sortować będziemy tablicę uporządkowaną odwrotnie, czyli np. posortujemy niemalejąco tablicę zawierająca liczby uporządkowane nierosnąco.
W takiej sytuacji wykonamy najwięcej zamian i maksymalną liczbę porównań.

Wykona się więc $n-1$ cykli po $n-i$ porównań, gdzie i to numer cyklu.

Przybliżona liczba operacji porównania (operacji dominujących) wykonanych przez algorytm będzie wynosić: n^2. Zatem złożoność czasowa wyniesie: $O\left(\mathrm{n}^2\right)$.

Złożoność czasowa przypadku optymistycznego

Przypadek optymistyczny to najlepszy z punktu widzenia programu zestaw danych, czyli taki, dla którego program wykona najmniej porównań.

Jeżeli poprawimy algorytm tak, żeby kończył działanie, gdy w ostatnim przebiegu (iteracji pętli zewnętrznej) nie została wykonana żadna zamiana, to będziemy w stanie ograniczyć liczbę wykonywanych operacji elementarnych. Rozwiązanie to pozwoli usprawnić działanie algorytmu, a tym samym zaoszczędzić czas poprzez ograniczenie liczby iteracji. Przykładowo, przy rozważaniu przypadku optymistycznego (czyli od razu posortowanej tablicy) będziemy mogli zakończyć działanie algorytmu po pierwszym obiegu zewnętrznej pętli. Najczęściej rozwiązanie to implementowane jest w postaci zmiennej pełniącej rolę flagi, która jest podnoszona przy dokonywaniu zamiany. Jeżeli pod koniec obiegu pętli zewnętrznej flaga nie jest podniesiona, oznacza to, że nie została dokonana żadna zamiana i można zakończyć działanie algorytmu.

Wykona się więc 1 cykl wykonujący $n-1$ porównań (liczba elementów sortowanych zmniejszona o 1, czyli liczba kolejnych par elementów sortowanych). Jeśli przy sortowaniu tablicy, na której przykładzie omówiliśmy algorytm, zaistniałby przypadek optymistyczny, liczba porównań wynosiłaby 7.

Zatem przybliżona liczba operacji porównania (operacji dominujących) wykonana przez algorytm będzie wynosić: n, czyli jego złożoność czasowa to: $O(\mathrm{n})$.

Można zauważyć, że w tej sytuacji algorytm pesymistyczny wykona się n razy wolniej niż optymistyczny.

Bez flagi zaprezentowany algorytm będzie miał złożoność czasową kwadratową, czyli $O\left(\mathrm{n}^2\right)$. Wynika to z faktu, że algorytm, mimo że nie dokonuje zamian i tak zrealizuje wszystkie cykle. Wykona się więc $n-1$ cykli po $n-i$ porównań.

Złożoność pamięciowa algorytmu

Sortowanie bąbelkowe wykonywane jest w jednej tablicy poprzez przestawianie jej elementów, więc potrzebuje tylko tyle miejsca w pamięci, ile zajmą elementy tablicy. Algorytm ten nie zajmuje dodatkowej pamięci, zatem złożoność pamięciowa tego algorytmu jest stała i zależna od liczby sortowanych elementów. Taka złożoność pamięciowa oznaczana jest symbolem $O(1)$. Jest to tzw. algorytm sortowania w miejscu.

Słownik

> **złożoność czasowa**
>
> czas, jaki zajmuje rozwiązanie danego problemu; zazwyczaj dla jego określenia podaje się liczbę operacji dominujących (elementarnych) potrzebnych do wykonania algorytmu;

zależy od ilości wprowadzonych danych wejściowych

złożoność pamięciowa

ilość pamięci potrzebnej do wykonania danego algorytmu; zwyczajowo określona w zależności od rozmiaru danych wejściowych lub innych parametrów mogących mieć wpływ na działanie algorytmu; mierzona jest w liczbie komórek pamięci komputera zajmowanej przez program; jest to jedna z miar wykorzystywanych do określania użyteczności i funkcjonalności algorytmu

Schemat interaktywny

Polecenie 1
Za pomocą schematu blokowego stwórz program, który posortuje niemalejąco ciąg liczb całkowitych o podanej długości i zawartości z użyciem algorytmu sortowania bąbelkowego.

Przetestuj działanie swojego programu dla ciągu zawierającego 5 elementów zapisanych w tablicy: [5, 2, 4, 1, 3].

Specyfikacja problemu:

Dane:

- n – liczba naturalna; liczba elementów ciągu do posortowania
- `tablica[]` – tablica elementów do posortowania; tablica liczb całkowitych

Wynik:

- elementy tablicy `tablica[]` posortowane niemalejąco

Polecenie 2
Zapisz program z polecenia 1 w wybranym języku programowania.

Polecenie 3
Dodaj do napisanego przez siebie programu komentarze, aby nawet osoba, która nie potrafi programować, mogła go zrozumieć.

Polecenie 4
Porównaj program zapisany w formie schematu blokowego z tym, który zaprezentowany został w sekcji „Przeczytaj”. Przeanalizuj różnice.

Sprawdź się

Pokaż ćwiczenia:

- Ćwiczenie 1
- Ćwiczenie 2
- Ćwiczenie 3
- Ćwiczenie 4
- Ćwiczenie 5
- Ćwiczenie 6
- Ćwiczenie 7
- Ćwiczenie 8
- Ćwiczenie 9

Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Sortowanie bąbelkowe

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres podstawowy

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres podstawowy. Uczeń:

1) planuje kolejne kroki rozwiązywania problemu, z uwzględnieniem podstawowych etapów myślenia komputacyjnego (określenie problemu, definicja modeli i pojęć, znalezienie rozwiązania, zaprogramowanie i testowanie rozwiązania).

2) stosuje przy rozwiązywaniu problemów z różnych dziedzin algorytmy poznane w szkole podstawowej oraz algorytmy:

c) porządkowania ciągu liczb: przez wstawianie i metodą bąbelkową,

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres podstawowy. Uczeń:

1) projektuje i programuje rozwiązania problemów z różnych dziedzin, stosuje przy tym: instrukcje wejścia/wyjścia, wyrażenia arytmetyczne i logiczne, instrukcje

warunkowe, instrukcje iteracyjne, funkcje z parametrami i bez parametrów, testuje poprawność programów dla różnych danych; w szczególności programuje algorytmy z punktu I.2);

2) do realizacji rozwiązań problemów prawidłowo dobiera środowiska informatyczne, aplikacje oraz zasoby, wykorzystuje również elementy robotyki;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Przeanalizujesz działanie algorytmu sortowania bąbelkowego.
- Prześledzisz, w jaki sposób można usprawnić algorytm sortowania bąbelkowego.
- Zbadasz złożoność czasową oraz pamięciową algorytmu sortowania bąbelkowego.
- Posortujesz przykładowy ciąg liczbowy za pomocą sortowania bąbelkowego.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Sortowanie bąbelkowe". Nauczyciel prosi uczniów o zapoznanie się z treściami w sekcji „Przeczytaj".

Faza wstępna:

1. Nauczyciel wyświetla uczniom temat, wskazuje cele zajęć oraz ustala z uczestnikami zajęć kryteria sukcesu.
2. Prowadzący prosi uczniów, aby zgłaszali swoje propozycje pytań do tematu. Jedna osoba może zapisywać je na tablicy. Gdy uczniowie wyczerpią swoje pomysły, a pozostały jakieś ważne kwestie do poruszenia, nauczyciel je dopowiada.

Faza realizacyjna:

1. **Praca z tekstem.** Jeżeli przygotowanie uczniów do lekcji jest niewystarczające, nauczyciel prosi o indywidualne zapoznanie się z treścią zawartą w sekcji „Przeczytaj". Każdy uczestnik zajęć podczas cichego czytania wynotowuje najważniejsze kwestie poruszane w tekście.
2. **Praca z multimedium.** Uczniowie pracują w parach. Analizują treść polecenia 1 z sekcji „Schemat interaktywny". Wybrana grupa omawia rozwiązanie na forum klasy. Następnie uczniowie implementują algorytm w wybranym języku programowania, nauczyciel sprawdza poprawność wykonania zadania.
3. **Ćwiczenie umiejętności.** Uczniowie wykonują indywidualnie ćwiczenia nr 1-5 następnie dzielą się wynikami swojej pracy z innymi uczniami.

Faza podsumowująca:

1. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności, omawia ewentualne problemy podczas rozwiązywania ćwiczeń.
2. Nauczyciel wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie". W kontekście ich realizacji podsumowuje przebieg zajęć, a także wskazuje mocne i słabe strony pracy uczniów.

Praca domowa:

1. Uczniowie wykonują ćwiczenia 6-8 z sekcji „Sprawdź się".
2. Uczniowie wykonują polecenie 3 z sekcji „Schemat interaktywny".

Wskazówki metodyczne:

- Uczniowie mogą wykorzystać treści w sekcjach: „Przeczytaj", „Schemat interaktywny", „Sprawdź się" jako materiał do lekcji powtórkowej.