Sortowanie bąbelkowe - wersja nr 1
Bubble Sort

Algorytm

Algorytm **sortowania bąbelkowego** jest jednym z najstarszych algorytmów sortujących. Można go potraktować jako ulepszenie opisanego w poprzednim rozdziale algorytmu sortowania głupiego. Zasada działania opiera się na cyklicznym porównywaniu par sąsiadujących elementów i zamianie ich kolejności w przypadku niespełnienia kryterium porządkowego zbioru. Operację tę wykonujemy dotąd, aż cały zbiór zostanie posortowany.

Algorytm sortowania bąbelkowego przy porządkowaniu zbioru nieposortowanego ma klasę czasowej złożoności obliczeniowej równą $O(n^2)$. Sortowanie odbywa się w miejscu.

Przykład:

Jako przykład działania algorytmu sortowania bąbelkowego posortujemy przy jego pomocy 5-cio elementowy zbiór liczb $\{5\ 4\ 3\ 2\ 1\}$, który wstępnie jest posortowany w kierunku odwrotnym, co możemy uznać za przypadek najbardziej niekorzystny, ponieważ wymaga przestawienia wszystkich elementów.

Obieg	Zbiór	Opis operacji
1	**5 4** 3 2 1	Rozpoczynamy od pierwszej pary, która wymaga wymiany elementów
	4 **5 3** 2 1	Druga para też wymaga zamiany elementów
	4 3 **5 2** 1	Wymagana wymiana elementów
	4 3 2 **5 1**	Ostatnia para również wymaga wymiany elementów
	4 3 2 **1 5**	Stan po pierwszym obiegu. Zwróć uwagę, iż najstarszy element **(5)** znalazł się na końcu zbioru, a najmłodszy **(1)** przesunął się o jedną pozycję w lewo.

2	4 3 2 1 5	Para wymaga wymiany
	3 4 2 1 5	Para wymaga wymiany
	3 2 4 1 5	Para wymaga wymiany
	3 2 1 4 5	Elementy są w dobrej kolejności, zamiana nie jest konieczna.
	3 2 **1** 4 5	Stan po drugim obiegu. Zwróć uwagę, iż najmniejszy element **(1)** znów przesunął się o jedną pozycję w lewo. Z obserwacji tych można wywnioskować, iż po każdym obiegu najmniejszy element wędruje o jedną pozycję ku początkowi zbioru. Najstarszy element zajmuje natomiast swe miejsce końcowe.
3	3 2 1 4 5	Para wymaga wymiany
	2 3 1 4 5	Para wymaga wymiany
	2 1 3 4 5	Dobra kolejność
	2 1 3 4 5	Dobra kolejność
	2 1 **3 4 5**	Stan po trzecim obiegu. Wnioski te same.
4	2 1 3 4 5	Para wymaga wymiany
	1 2 3 4 5	Dobra kolejność
	1 2 3 4 5	Dobra kolejność
	1 2 3 4 5	Dobra kolejność
	1 2 **3 4 5**	Zbiór jest posortowany. Koniec

Posortowanie naszego zbioru wymaga 4 obiegów. Jest to oczywiste: w przypadku najbardziej niekorzystnym najmniejszy element znajduje się na samym końcu zbioru

wejściowego. Każdy obieg przesuwa go o jedną pozycję w kierunku początku zbioru. Takich przesunięć należy wykonać n - 1 (n - ilość elementów w zbiorze).

Algorytm sortowania bąbelkowego, w przeciwieństwie do algorytmu sortowania głupiego, nie przerywa porównywania par elementów po napotkaniu pary nie spełniającej założonego porządku. Po zamianie kolejności elementów sprawdzana jest kolejna para elementów sortowanego zbioru. Dzięki temu podejściu rośnie efektywność algorytmu oraz zmienia się klasa czasowej złożoności obliczeniowej z $O(n^3)$ na $O(n^2)$.

Uwaga:

Algorytm sortowania bąbelkowego jest uważany za bardzo zły algorytm sortujący. Można go stosować tylko dla niewielkiej liczby elementów w sortowanym zbiorze (do około 5000). Przy większych zbiorach czas sortowania może być zbyt długi.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in$ N

d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, $i, j \in$ N

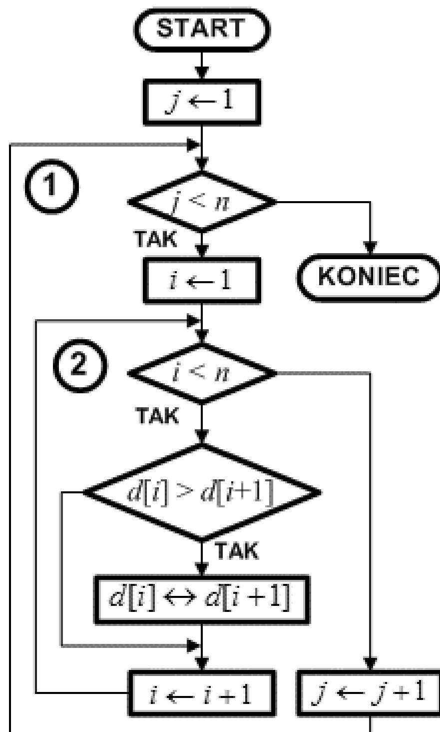
Lista kroków

K01: **Dla** *j* = 1,2,...,*n* - 1: **wykonuj** K02

K02: **Dla** *i* = 1,2,...,*n* - 1: **jeśli** d[*i*] > d[*i* + 1], **to** d[*i*] ↔ d[*i* + 1]

K03: **Zakończ**

Schemat blokowy



Sortowanie wykonywane jest w dwóch zagnieżdżonych pętlach. Pętla zewnętrzna nr 1 kontrolowana jest przez zmienną j. Wykonuje się ona n - 1 razy. Wewnątrz pętli nr 1 umieszczona jest pętla nr 2 sterowana przez zmienną i. Wykonuje się również n - 1 razy. W efekcie algorytm wykonuje w sumie:

$$T_1(n) = (n - 1)^2 = n^2 - 2n + 1$$

obiegów pętli wewnętrznej, po których zakończeniu zbiór zostanie posortowany.

Sortowanie odbywa się wewnątrz pętli nr 2. Kolejno porównywany jest i-ty element z elementem następnym. Jeśli elementy te są w złej kolejności, to zostają zamienione miejscami. W tym miejscu jest najważniejsza różnica pomiędzy algorytmem sortowania bąbelkowego a algorytmem sortowania głupiego. Ten drugi w momencie napotkania elementów o złej kolejności zamienia je miejscami i rozpoczyna cały proces sortowania od początku. Algorytm sortowania bąbelkowego wymienia miejscami źle ułożone elementy sortowanego zbioru i przechodzi do następnej pary zwiększając indeks i o 1. Dzięki takiemu podejściu rośnie efektywność, co odzwierciedla klasa czasowej złożoności obliczeniowej:

Sortowanie głupie - $O(n^3)$; Sortowanie bąbelkowe - $O(n^2)$

Podsumowanie

Analizując wyniki obliczeń w arkuszu kalkulacyjnym otrzymanych czasów sortowania dla algorytmu sortowania bąbelkowego 1 wyciągamy następujące wnioski:

Cechy Algorytmu Sortowania Bąbelkowego wersja nr 1	
klasa złożoności obliczeniowej optymistyczna	$O(n^2)$
klasa złożoności obliczeniowej typowa	$O(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$O(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Klasy złożoności obliczeniowej szacujemy następująco:

- **optymistyczna** - dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoich miejscach) - na podstawie czasów t_{po}, t_{pp}, t_{pk}
- **typowa** - dla zbiorów o losowym rozkładzie elementów - na podstawie czasu t_{np}
- **pesymistyczna** - dla zbiorów posortowanych odwrotnie - na podstawie czasu t_{od}.

Własności algorytmu					
Algorytm	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe wersja nr 1	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
	$t_{po} \approx \frac{1}{5} t_{od}$		$t_{po} = t_{pp} = t_{pk}$		$t_{np} \approx \frac{2}{3} t_{od}$

1. Wszystkie badane przez nas czasy sortowania są proporcjonalne do kwadratu liczby elementów zbioru. Klasa czasowej złożoności obliczeniowej wynosi $O(n^2)$.
2. Czasy sortowania t_{po}, t_{pp} oraz t_{pk} są praktycznie takie same.
3. Czas sortowania zbioru nieuporządkowanego jest krótszy od czasu sortowania zbioru uporządkowanego odwrotnie.

Wzrost prędkości sortowania					
Algorytmy	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie głupie Sortowanie bąbelkowe 1	$\approx \frac{1}{n}$ źle	$\approx \frac{n}{3}$ 3 dobrze	$\approx \frac{1}{6}$ źle	$\approx \frac{1}{4}$ źle	$\approx \frac{n}{3}$ 3 dobrze

4. Porównując wzrost prędkości algorytmu sortowania bąbelkowego w stosunku do algorytmu sortowania głupiego zauważamy, iż korzyść następuje w przypadku sortowania zbioru uporządkowanego odwrotnie oraz zbioru nieuporządkowanego. Wzrost prędkości jest proporcjonalny liniowo do liczby elementów w sortowanym zbiorze. W pozostałych przypadkach algorytm sortowania głupiego jest dużo szybszy od tej wersji algorytmu.

Zadania dla ambitnych

1. Uzasadnij, iż wszystkie czasy dla tej wersji algorytmu są proporcjonalne do kwadratu liczby sortowanych elementów.
2. Dlaczego czasy t_{po}, t_{pp} oraz t_{pk} są takie same?
3. Uzasadnij wyniki otrzymane przy badaniu wzrostu prędkości algorytmu sortowania bąbelkowego w stosunku do algorytmu sortowania głupiego.

Sortowanie bąbelkowe - wersja nr 2
Bubble Sort

Algorytm

Podany w poprzednim rozdziale algorytm sortowania bąbelkowego można zoptymalizować pod względem czasu wykonania. Jeśli przyjrzymy się dokładnie obiegom wykonywanym w tym algorytmie, to zauważymy bardzo istotną rzecz:

> Po wykonaniu pełnego obiegu w algorytmie sortowania bąbelkowego najstarszy element wyznaczony przez przyjęty porządek zostaje umieszczony na swoim właściwym miejscu - na końcu zbioru.

Wniosek ten jest oczywisty. W każdej kolejnej parze porównywanych elementów element starszy przechodzi na drugą pozycję. W kolejnej parze jest on na pierwszej pozycji, a skoro jest najstarszym, to po porównaniu znów przejdzie na pozycję drugą itd. - jest jakby ciągnięty na koniec zbioru (jak bąbelek powietrza wypływający na powierzchnię wody).

Przykład:

Wykonamy jeden obieg sortujący dla zbioru pięcioelementowego

{ **9** 3 1 7 0 }. Elementem najstarszym jest pierwszy element - liczba 9.

Obieg	Zbiór	Opis operacji
1	**9** **3** 1 7 0	Para wymaga przestawienia elementów. Element najstarszy przejdzie na drugą pozycję w parze.
	3 **9** **1** 7 0	Konieczne przestawienie elementów. Element najstarszy znów trafi na pozycję drugą w parze.
	3 1 **9** **7** 0	Konieczne przestawienie elementów.
	3 1 7 **9** **0**	Ostatnia para również wymaga przestawienia elementów.

	3 1 7 0 9	Koniec obiegu. Najstarszy element znalazł się na końcu zbioru.

Co z tego wynika dla nas? Otóż po każdym obiegu na końcu zbioru tworzy się podzbiór uporządkowanych najstarszych elementów. Zatem w kolejnych obiegach możemy pomijać sprawdzanie ostatnich elementów - liczebność zbioru do posortowania z każdym obiegiem maleje o 1.

Przykład:

Dokończmy sortowania podanego powyżej zbioru uwzględniając podane przez nas fakty. Po pierwszym obiegu na końcu zbioru mamy umieszczony element najstarszy. W drugim obiegu będziemy zatem sortować zbiór 4 elementowy, w trzecim obiegu 3 elementowy i w obiegu ostatnim, czwartym - zbiór 2 elementowy.

Obieg	Zbiór	Opis operacji
2	3 1 7 0 9	Para wymaga przestawienia elementów.
	1 3 7 0 9	Dobra kolejność
	1 3 7 0 9	Konieczne przestawienie elementów.
	1 3 0 7 9	Koniec obiegu. Na końcu zbioru mamy 2 elementy uporządkowane.
3	1 3 0 7 9	Dobra kolejność
	1 3 0 7 9	Konieczne przestawienie elementów.
	1 0 3 7 9	Koniec obiegu. Na końcu zbioru mamy 3 elementy uporządkowane.
4	1 0 3 7 9	Konieczne przestawienie elementów.
	0 1 3 7 9	Koniec ostatniego obiegu - zbiór jest posortowany.

W porównaniu do tabelki z poprzedniego rozdziału nawet wzrokowo możemy zauważyć istotne zmniejszenie ilości niezbędnych operacji.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, $i, j \in N$

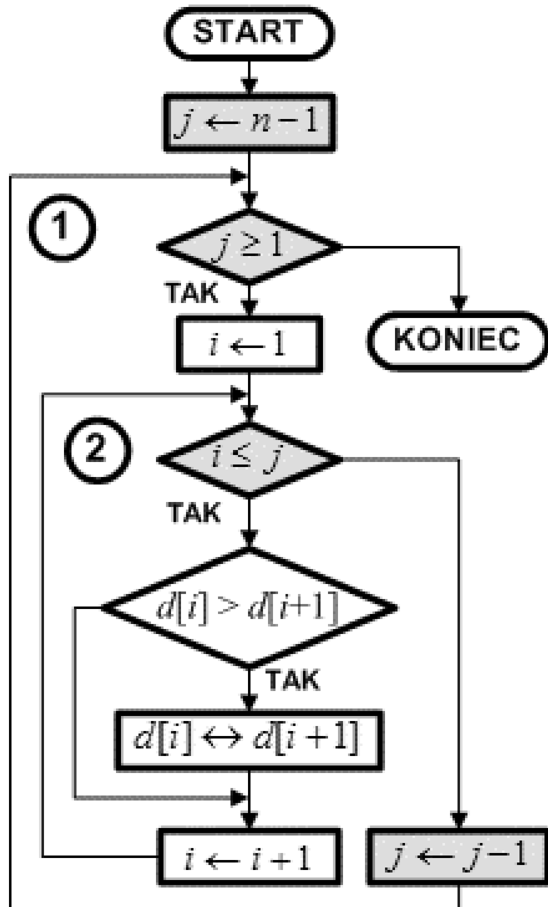
Lista kroków

K01: **Dla** *j* = *n* - 1, *n* - 2, ..., 1: **wykonuj** K02

K02: **Dla** *i* = 1, 2, ..., *j*: **jeśli** d[*i*] > d[*i* + 1], **to** d[*i*] ↔ d[*i* + 1]

K03: **Zakończ**

Schemat blokowy



Zmiany w stosunku do poprzedniej wersji zaznaczyliśmy na schemacie blokowym innym kolorem elementów. Są one następujące:

- pętla zewnętrzna nr 1 zlicza obiegi wstecz, tzn. pierwszy obieg ma numer n-1. Dzięki takiemu podejściu w zmiennej j mamy zawsze numer ostatniego elementu, do którego ma dojść pętla wewnętrzna nr 2. Ta zmiana wymaga również odwrotnej iteracji zmiennej j.

- pętla wewnętrzna sprawdza w warunku kontynuacji, czy wykonała j obiegów, a nie jak poprzednio n-1 obiegów. Dzięki temu po każdym obiegu pętli nr 1 (zewnętrznej) pętla nr 2 będzie wykonywać o jeden obieg mniej.

Pozostała część algorytmu nie jest zmieniona - w pętli wewnętrznej nr 2 sprawdzamy, czy element $d[i]$ jest w złej kolejności z elementem $d[i+1]$. Sprawdzany warunek spowoduje posortowanie zbioru rosnąco. Przy sortowaniu malejącym zmieniamy relację większości na relację mniejszości. Jeśli warunek jest spełniony, zamieniamy miejscami element $d[i]$ z elementem $d[i+1]$, po czym kontynuujemy pętlę nr 2 zwiększając o 1 indeks i. Po każdym zakończeniu pętli nr 2 indeks j jest zmniejszany o 1.

Ilość obiegów pętli wewnętrznej wynosi:

$$T_2(n) = (n\text{-}1) + (n\text{-}2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$$

Otrzymane wyrażenie ma wciąż kwadratową klasę złożoności obliczeniowej, jednakże $T_2(n) < T_1(n)$ dla $n > 1$. Osiągnęliśmy zatem większą efektywność działania dzięki wprowadzonym zmianom.

Podsumowanie

Analizując wyniki obliczeń w arkuszu kalkulacyjnym otrzymanych czasów sortowania dla algorytmu sortowania bąbelkowego 2 wyciągamy następujące wnioski:

Cechy Algorytmu Sortowania Bąbelkowego wersja nr 2	
klasa złożoności obliczeniowej optymistyczna	$O(n^2)$
klasa złożoności obliczeniowej typowa	$O(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$O(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Klasy złożoności obliczeniowej szacujemy następująco:

- **optymistyczna** - dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoich miejscach) - na podstawie czasów t_{po}, t_{pp}, t_{pk}
- **typowa** - dla zbiorów o losowym rozkładzie elementów - na podstawie czasu t_{np}
- **pesymistyczna** - dla zbiorów posortowanych odwrotnie - na podstawie czasu t_{od}.

Własności algorytmu					
Algorytm	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$

wersja nr 2	$t_{po} \approx \frac{1}{10} t_{od}$	$t_{po}=t_{pp}=t_{pk}$	$t_{np} \approx \frac{3}{5} t_{od}$

1. Wszystkie czasy sortowania są proporcjonalne do kwadratu liczby elementów w sortowanych zbiorach, zatem klasa czasowej złożoności obliczeniowej jest równa $O(n^2)$.
2. Czasy sortowania t_{po}, t_{pp} oraz t_{pk} są praktycznie takie same.

Wzrost prędkości sortowania					
Algorytmy	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe 1 Sortowanie bąbelkowe 2	$\approx \frac{13}{6}$	$\approx \frac{11}{10}$	≈ 2	≈ 2	$\approx \frac{7}{6}$
	dobrze	niewiele	dobrze	dobrze	niewiele

3. Analizując wzrost prędkości algorytmu sortowania bąbelkowego w wersji 2 w stosunku do wersji 1 zauważamy, iż wprowadzona zmiana faktycznie zwiększyła ogólną prędkość działania. Jednakże korzyść w typowym przypadku zbioru nieuporządkowanego jest stosunkowo niewielka. Największy wzrost prędkości notujemy przy sortowaniu zbiorów częściowo uporządkowanych.

Zadania dla ambitnych

1. Jakiego typu operacje są wykonywane w algorytmie sortowania bąbelkowego?
2. Na jakich operacjach zyskaliśmy w tej wersji algorytmu?
3. Które operacje dominują przy wyznaczaniu czasów t_{po}, t_{pp} oraz t_{pk}?
4. Uzasadnij wyniki otrzymane przy badaniu wzrostu prędkości algorytmu sortowania bąbelkowego w wersji 2 w stosunku do wersji 1.

Sortowanie bąbelkowe - wersja nr 3
Bubble Sort

Algorytm

Algorytm **sortowania bąbelkowego** wykonuje dwa rodzaje operacji:

- test bez zamiany miejsc elementów
- test ze zamianą miejsc elementów.

Pierwsza z tych operacji nie sortuje zbioru, jest więc **operacją pustą**. Druga operacja dokonuje faktycznej zmiany porządku elementów, jest zatem **operacją sortującą**.

Ze względu na przyjęty sposób sortowania algorytm bąbelkowy zawsze musi wykonać tyle samo operacji sortujących. Tego nie możemy zmienić. Jednakże możemy wpłynąć na eliminację operacji pustych. W ten sposób usprawnimy działanie algorytmu.

Jeśli dokładnie przyjrzałeś się wersjom 1 i 2, które prezentowaliśmy w poprzednich rozdziałach, to powinieneś dokonać następujących spostrzeżeń:

1. Wersja pierwsza jest najmniej optymalną wersją algorytmu bąbelkowego. Wykonywane są wszystkie możliwe operacje sortujące i puste.
2. Wersja druga redukuje ilość operacji pustych poprzez ograniczanie liczby obiegów pętli wewnętrznej (sortującej).

Możliwa jest dalsza redukcja operacji pustych, jeśli będziemy sprawdzać, czy w pętli wewnętrznej były przestawiane elementy (czyli czy wykonano operacje sortujące). Jeśli nie, to zbiór jest już posortowany (dlaczego?) i możemy zakończyć pracę algorytmu.

Teraz rośnie trudność wyznaczenia **czasowej złożoności obliczeniowej**, ponieważ ilość faktycznie wykonywanych operacji porównań i przestawień zależy od rozkładu elementów w sortowanym zbiorze. Zadanie komplikuje dodatkowo fakt, iż operacja pusta jest zwykle wykonywana kilkakrotnie szybciej od operacji sortującej. Na pewno można powiedzieć, iż dla zbioru posortowanego algorytm wykona tylko $n - 1$ operacji pustych, zatem w przypadku najbardziej optymistycznym czasowa złożoność obliczeniowa redukuje się do klasy $O(n)$ - liniowej. W przypadku najbardziej niekorzystnym algorytm wykona wszystkie operacje puste i sortujące, zatem będzie posiadał klasę czasowej złożoności obliczeniowej $O(n^2)$.

Przykład:

Posortujmy zbiór { 3 1 0 7 9 } zgodnie z wprowadzoną modyfikacją.

Obieg	Zbiór	Opis operacji
1	**3 1** 0 7 9	Konieczne przestawienie elementów
	1 **3 0** 7 9	Konieczne przestawienie elementów
	1 0 3 7 9	Elementy w dobrej kolejności
	1 0 3 7 9	Elementy w dobrej kolejności
	1 0 3 7 9	Koniec pierwszego obiegu. Ponieważ były przestawienia elementów, sortowanie kontynuujemy
2	**1 0** 3 7 9	Konieczne przestawienie elementów
	0 1 3 7 9	Elementy w dobrej kolejności
	0 1 3 7 9	Elementy w dobrej kolejności
	0 1 3 7 9	Koniec drugiego obiegu. Było przestawienie elementów, zatem sortowanie kontynuujemy
3	0 1 3 7 9	Elementy w dobrej kolejności
	0 1 3 7 9	Elementy w dobrej kolejności
	0 1 3 7 9	Koniec trzeciego obiegu. Nie było przestawień elementów, kończymy sortowanie. Wykonaliśmy o 1 obieg sortujący mniej.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, $i, j \in N$

p - znacznik zamiany miejsc elementów w zbiorze. $p \in N$

Lista kroków

K01: **Dla** *j* = *n* - 1, *n* - 2, ..., 1: **wykonuj** K02...K04

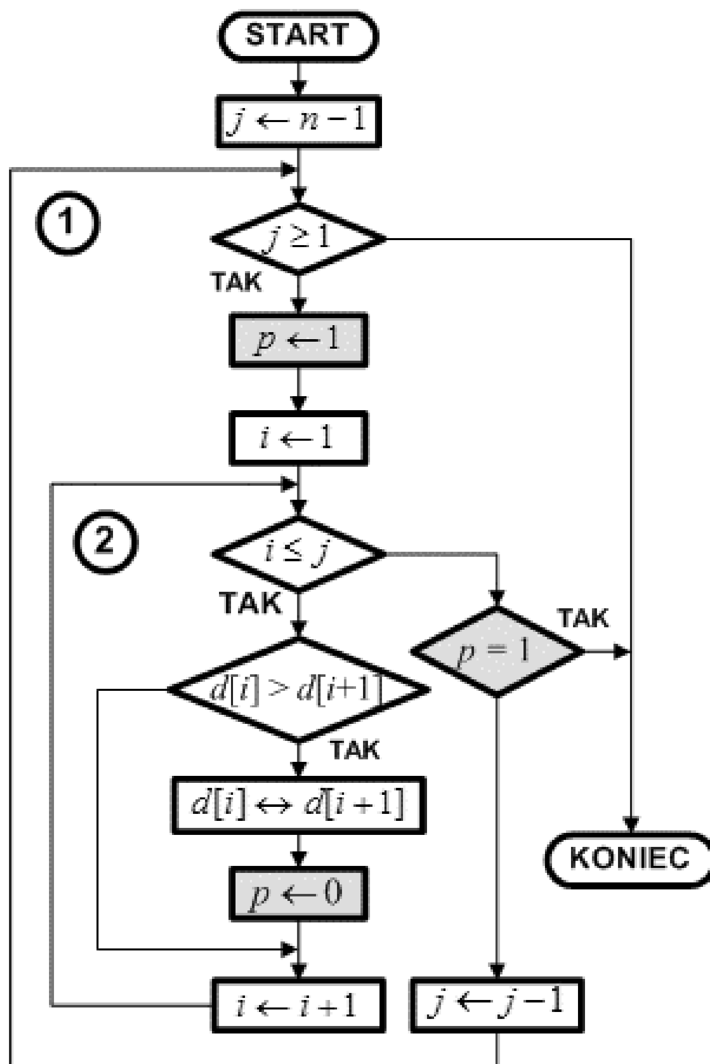
K02: $p \leftarrow 1$

K03: **Dla** *i* = 1, 2, ..., j: **jeśli** d[*i*] > d[*i* + 1], **to** d[*i*] ↔ d[*i* + 1]; $p \leftarrow 0$

K04: **Jeśli** *p* = 1, **to zakończ**

K05: **Zakończ**

Schemat blokowy



Wprowadzona do algorytmu sortowania bąbelkowego modyfikacja ma na celu wykrycie posortowania zbioru. Zmiany zaznaczyliśmy blokami o odmiennym kolorze.

Zbiór będzie posortowany, jeśli po wykonaniu wewnętrznego obiegu sortującego nie wystąpi ani jedno przestawienie elementów porządkowanego zbioru.

Przed wejściem do pętli sortującej nr 2 ustawiamy zmienną pomocniczą p. Jeśli w pętli zajdzie potrzeba przestawienia elementów, to zmienna p jest zerowana. Po wykonaniu pętli sortującej sprawdzamy, czy zmienna p jest ustawiona. Jeśli tak, to przestawienie elementów nie wystąpiło, zatem kończymy algorytm. W przeciwnym razie wykonujemy kolejny obieg pętli nr 1.

Podsumowanie

Analizując wyniki obliczeń w arkuszu kalkulacyjnym otrzymanych czasów sortowania dla algorytmu sortowania bąbelkowego 3 wyciągamy następujące wnioski:

Cechy Algorytmu Sortowania Bąbelkowego wersja nr 3	
klasa złożoności obliczeniowej optymistyczna	$O(n)$ -$O(n^2)$
klasa złożoności obliczeniowej typowa	$O(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$O(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Klasy złożoności obliczeniowej szacujemy następująco:

- **optymistyczna** - dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoich miejscach) - na podstawie czasów t_{po}, t_{pp}, t_{pk}
- **typowa** - dla zbiorów o losowym rozkładzie elementów - na podstawie czasu t_{np}
- **pesymistyczna** - dla zbiorów posortowanych odwrotnie - na podstawie czasu t_{od}.

Własności algorytmu					
Algorytm	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe wersja nr 3	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
	$t_{po} << t_{od}$		$t_{pp} << t_{pk}$		$t_{np} \approx \frac{3t_{od}}{5}$

1. Wprowadzona zmiana wpłynęła na zmianę klasy czasowej złożoności obliczeniowej przy sortowaniu zbioru uporządkowanego oraz przy sortowaniu zbioru uporządkowanego z losowym elementem na początku. W obu przypadkach notujemy proporcjonalność czasu sortowania do liczby

elementów w zbiorze, zatem klasa czasowej złożoności obliczeniowej wynosi $O(n)$.

2. Nie zmieniła się natomiast klasa czasowej złożoności obliczeniowej przy sortowaniu zbioru uporządkowanego odwrotnie, przy sortowaniu zbioru uporządkowanego z losowym elementem na końcu oraz przy sortowaniu zbioru nieuporządkowanego. Dlatego w przypadku ogólnym klasa czasowej złożoności obliczeniowej wynosi $O(n^2)$.

Wzrost prędkości sortowania					
Algorytmy	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe 2 Sortowanie bąbelkowe 3	$\approx \frac{2}{5}n$	≈ 1	$\approx 0{,}07n$	$\approx \frac{3}{2}$	≈ 1
	dobrze	**brak**	**dobrze**	**dobrze**	**brak**

3. Analizując wzrost prędkości algorytmu sortowania bąbelkowego w wersji 3 w stosunku do wersji 2 zauważamy, iż wprowadzona zmiana nie zwiększyła prędkości działania algorytmu dla przypadku zbioru posortowanego odwrotnie oraz dla zbioru nieuporządkowanego. Największy zysk (zmniejszenie klasy czasowej złożoności obliczeniowej) występuje przy sortowaniu zbiorów częściowo uporządkowanych.

Zadania dla ambitnych

1. Uzasadnij zmianę klasy czasowej złożoności obliczeniowej z $O(n^2)$ dla operacji sortowania zbioru uporządkowanego w wersjach nr 1 i 2 algorytmu sortowania bąbelkowego na klasę $O(n)$ w wersji nr 3.
2. Dlaczego nie uzyskaliśmy istotnego wzrostu prędkości pracy algorytmu dla przypadku zbioru posortowanego odwrotnie oraz zbioru nieuporządkowanego.
3. Zbadaj wzrost prędkości algorytmu sortowania bąbelkowego w wersji nr 3 w stosunku do wersji nr 2. Wyciągnij odpowiednie wnioski.

Sortowanie bąbelkowe - wersja nr 4
Bubble Sort

Algorytm

Czy algorytm **sortowania bąbelkowego** można jeszcze ulepszyć? Tak, ale zaczynamy już osiągać kres jego możliwości, ponieważ ulepszenia polegają jedynie na redukcji **operacji pustych**. Wykorzystamy informację o miejscu wystąpienia **zamiany elementów** (czyli o miejscu wykonania operacji sortującej).

Jeśli w **obiegu sortującym** wystąpi pierwsza zamiana na pozycji i-tej, to w kolejnym obiegu będziemy rozpoczynali sortowanie od pozycji o jeden mniejszej (chyba, że pozycja *i*-ta była pierwszą pozycją w zbiorze). Dlaczego? Odpowiedź jest prosta. Zamiana spowodowała, iż młodszy element znalazł się na pozycji i-tej. Ponieważ w obiegu sortującym młodszy element zawsze przesuwa się o 1 pozycję w kierunku początku zbioru, to nie ma sensu sprawdzanie pozycji od 1 do i-2, ponieważ w poprzednim obiegu zostały one już sprawdzone, nie wystąpiła na nich zamiana elementów, zatem elementy na pozycjach od 1 do i-2 są chwilowo w dobrej kolejności względem siebie. Nie mamy tylko pewności co do pozycji i-1-szej oraz i-tej, ponieważ ostatnia zamiana elementów umieściła na i-tej pozycji młodszy element, który być może należy wymienić z elementem na pozycji wcześniejszej, czyli i-1. W ten sposób określimy początkową pozycję, od której rozpoczniemy sortowanie elementów w następnym obiegu sortującym.

Ostatnia zamiana elementów wyznaczy pozycję końcową dla następnego obiegu. Wiemy, iż w każdym obiegu sortującym najstarszy element jest zawsze umieszczany na swojej docelowej pozycji. Jeśli ostatnia zamiana elementów wystąpiła na pozycji i-tej, to w następnym obiegu porównywanie elementów zakończymy na pozycji o 1 mniejszej - w ten sposób nie będziemy sprawdzać już najstarszego elementu z poprzedniego obiegu.

Sortowanie prowadzimy dotąd, aż w obiegu sortującym nie wystąpi ani jedna zamiana elementów.

Teoretycznie powinno to zoptymalizować algorytm, ponieważ są sortowane tylko niezbędne fragmenty zbioru - pomijamy obszary posortowane, które tworzą się na końcu i na początku zbioru. Oczywiście zysk nie będzie oszałamiający w przypadku zbioru nieuporządkowanego lub posortowanego odwrotnie (może się zdarzyć, iż ewentualne korzyści czasowe będą mniejsze od czasu wykonywania dodatkowych operacji). Jednakże dla zbiorów w dużym stopniu uporządkowanych możemy uzyskać całkiem rozsądny algorytm sortujący prawie w czasie liniowym O(n).

Przykład:

Według opisanej powyżej metody posortujmy zbiór { 0 1 2 3 **5** **4** 7 9 }. W zbiorze tym są tylko dwa elementy nieuporządkowane - 5 i 4.

Obieg	Zbiór	Opis operacji
1	0 1 2 3 5 4 7 9	Pierwszy obieg, rozpoczynamy od pierwszej pozycji.
	0 1 2 3 5 4 7 9	Elementy w dobrej kolejności. Zamiana miejsc nie występuje.
	0 1 2 3 5 4 7 9	
	0 1 2 3 5 4 7 9	
	0 1 2 3 5 4 7 9	Elementy w złej kolejności. Zapamiętujemy pozycję wymiany. Jest to jednocześnie pierwsza i ostatnia wymiana elementów w tym obiegu sortującym.
	0 1 2 3 4 5 7 9	Elementy w dobrej kolejności. Zamiana miejsc nie występuje.
	0 1 2 3 4 5 7 9	
	0 1 2 3 4 5 7 9	Koniec pierwszego obiegu. Zbiór jest już uporządkowany, ale ponieważ była zamiana elementów, algorytm dla pewności musi wykonać jeszcze jeden obieg sortujący.
2	0 1 2 3 4 5 7 9	Sortowanie rozpoczynamy od pozycji o 1 mniejszej od tej, na której wystąpiła w poprzednim obiegu wymiana elementów. Elementy są w dobrej kolejności. Dalszych sprawdzeń nie wykonujemy - kończymy na pozycji o 1 mniejszej, niż pozycja ostatniej zamiany w poprzednim obiegu.
	0 1 2 3 4 5 7 9	Koniec, zbiór jest posortowany

Chociaż podany przykład jest troszeczkę tendencyjny, to jednak pokazuje wyraźnie, iż zoptymalizowany algorytm sortowania bąbelkowego może bardzo szybko posortować zbiory prawie uporządkowane.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i - zmienna sterująca pętli, $i \in N$

p_{min} - dolna granica pozycji sortowanych elementów, $p_{min} \in N$

p_{max} - górna granica pozycji sortowanych elementów, $p_{max} \in N$

p - numer pozycji zamiany elementów, $p \in N$

Lista kroków

K01: $p_{min} \leftarrow 1$; $p_{max} \leftarrow n - 1$

K02 $p \leftarrow 0$

K03: **Dla** $i = p_{min}, ..., p_{max}$: **wykonuj** K04...K07

K04: **Jeśli** $d[i] \leq d[i + 1]$, **to następny obieg pętli** K03

K05: $d[i] \leftrightarrow d[i + 1]$

K06: **Jeśli** $p = 0$, **to** $p_{min} \leftarrow i$

K07: $p \leftarrow i$

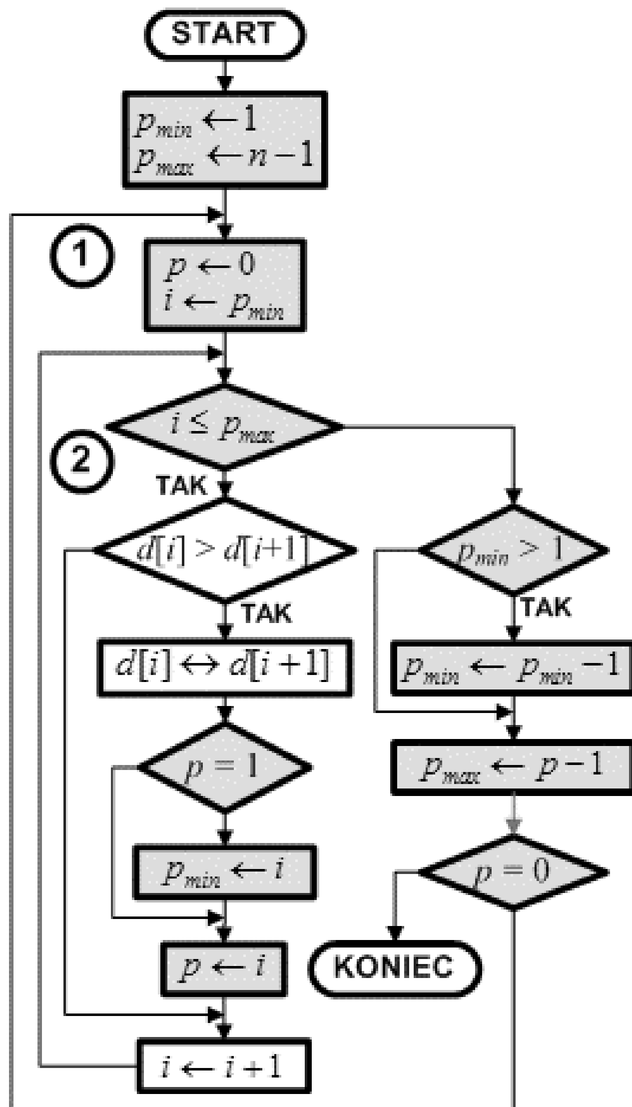
K08: **Jeśli** $p_{min} > 1$, **to** $p_{min} \leftarrow p_{min} - 1$

K09: $p_{max} \leftarrow p - 1$

K10: **Jeśli** $p > 0$, **to idź do** K02

K11: **Zakończ**

Schemat blokowy



Tym razem wprowadzonych zmian do algorytmu sortowania bąbelkowego jest dużo, zatem opiszemy cały algorytm od początku.

Zmienna p_{min} przechowuje numer pozycji, od której rozpoczyna się sortowanie zbioru. W pierwszym obiegu sortującym rozpoczynamy od pozycji nr 1. Zmienna p_{max} przechowuje numer ostatniej pozycji do sortowania. Pierwszy obieg sortujący kończymy na pozycji n-1, czyli na przedostatniej.

Pętla numer 1 wykonywana jest dotąd, aż w wewnętrznej pętli nr 2 nie wystąpi żadna zamiana elementów. Zmienna p pełni w tej wersji algorytmu nieco inną rolę niż poprzednio. Mianowicie będzie przechowywała numer

pozycji, na której algorytm ostatnio dokonał wymiany elementów. Na początku wpisujemy do p wartość 0, która nie oznacza żadnej pozycji w zbiorze. Zatem jeśli ta wartość zostanie zachowana, uzyskamy pewność, iż zbiór jest posortowany, ponieważ nie dokonano wymiany elementów.

Wewnętrzną pętlę sortującą rozpoczynamy od pozycji p_{min}. W pętli sprawdzamy kolejność elementu i-tego z elementem następnym. Jeśli kolejność jest zła, wymieniamy miejscami te dwa elementy. Po wymianie sprawdzamy, czy jest to pierwsza wymiana - zmienna p ma wtedy wartość 0. Jeśli tak, to numer pozycji, na której dokonano wymiany umieszczamy w p_{min}. Numer ten zapamiętujemy również w zmiennej p. Zwróć uwagę, iż dzięki takiemu podejściu p zawsze będzie przechowywało numer pozycji ostatniej wymiany - jest to zasada zwana "ostatni zwycięża".

Po sprawdzeniu elementów przechodzimy do następnej pozycji zwiększając i o 1 i kontynuujemy pętlę, aż do przekroczenia pozycji p_{max}. Wtedy pętla wewnętrzna zakończy się.

Jeśli w pętli nr 2 była dokonana zamiana elementów, to p_{min} zawiera numer pozycji pierwszej zamiany. Jeśli nie jest to pierwsza pozycja w zbiorze, p_{min} zmniejszamy o 1, aby pętla sortująca rozpoczynała od pozycji poprzedniej w stosunku do pozycji pierwszej zamiany elementów.

Pozycję ostatnią zawsze ustalamy o 1 mniejszą od numeru pozycji końcowej zamiany elementów.

Na koniec sprawdzamy, czy faktycznie doszło do zamiany elementów. Jeśli tak, to p jest większe od 0, gdyż zawiera numer pozycji w zbiorze, na której algorytm wymienił miejscami elementy. W takim przypadku pętlę nr 1 rozpoczynamy od początku. W przeciwnym razie kończymy, zbiór jest uporządkowany.

Podsumowanie

Analizując wyniki obliczeń w arkuszu kalkulacyjnym otrzymanych czasów sortowania dla algorytmu sortowania bąbelkowego 4 wyciągamy następujące wnioski:

Cechy Algorytmu Sortowania Bąbelkowego wersja nr 4

klasa złożoności obliczeniowej optymistyczna	$O(n)$
klasa złożoności obliczeniowej typowa	$O(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$O(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Klasy złożoności obliczeniowej szacujemy następująco:

- **optymistyczna** - dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoich miejscach) - na podstawie czasów t_{po}, t_{pp}, t_{pk}
- **typowa** - dla zbiorów o losowym rozkładzie elementów - na podstawie czasu t_{np}
- **pesymistyczna** - dla zbiorów posortowanych odwrotnie - na podstawie czasu t_{od}.

Własności algorytmu					
Algorytm	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe wersja nr 4	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n^2)$
	$t_{po} << t_{od}$		$t_{pp} \approx \frac{5}{3} t_{pk}$		$t_{np} \approx \frac{3}{5} t_{od}$

1. Wprowadzona zmiana wpłynęła na zmianę klasy czasowej złożoności obliczeniowej przy sortowaniu zbioru uporządkowanego z losowym elementem na końcu z $O(n^2)$ na $O(n)$.
2. Nie zmieniła się natomiast klasa czasowej złożoności obliczeniowej przy sortowaniu zbioru uporządkowanego odwrotnie i przy sortowaniu zbioru nieuporządkowanego. Dlatego w przypadku ogólnym klasa czasowej złożoności obliczeniowej wynosi $O(n^2)$.
3. Z wyników testu wyciągamy wniosek, iż ta wersja algorytmu sortowania bąbelkowego jest najlepszą ze wszystkich przedstawionych tutaj wersji. Dlatego dalsze algorytmy sortujące będziemy porównywali do wyników zoptymalizowanego algorytmu sortowania bąbelkowego.

Wzrost prędkości sortowania					
Algorytmy	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}

Sortowanie bąbelkowe 3	≈ 1	≈ 1	≈ 1	$\approx 0{,}08n$	≈ 1
Sortowanie bąbelkowe 4	**brak**	**brak**	**brak**	**dobrze**	**brak**

3. Analizując wzrost prędkości algorytmu sortowania bąbelkowego w wersji 4 w stosunku do wersji 3 zauważamy, iż wprowadzona zmiana nie zwiększyła w istotny sposób prędkości działania algorytmu w przypadku ogólnym. Zysk jest tylko przy sortowaniu zbiorów w dużym stopniu uporządkowanych. W takich przypadkach zoptymalizowany algorytm sortowania bąbelkowego posiada liniową czasową złożoność obliczeniową.

Zadania dla ambitnych

1. Uzasadnij powód zmiany klasy czasowej złożoności obliczeniowej z $O(n^2)$ na $O(n)$ w tej wersji algorytmu dla przypadku sortowania zbiorów w znacznym stopniu uporządkowanych.
2. Dlaczego w pozostałych przypadkach wzrost prędkości działania algorytmu jest niewielki?

Dwukierunkowe sortowanie bąbelkowe
Bidirectional Bubble Sort
Cocktail Sort

Algorytm

Dwukierunkowe sortowanie bąbelkowe oparte jest na spostrzeżeniu, iż każdy obieg wewnętrznej **pętli sortującej** umieszcza na właściwym miejscu **element najstarszy**, a **elementy młodsze** przesuwa o 1 pozycję w kierunku początku zbioru. Jeśli pętla ta zostanie wykonana w kierunku odwrotnym, to wtedy najmłodszy element znajdzie się na swoim właściwym miejscu, a elementy starszy przesuną się o jedną pozycję w kierunku końca zbioru. Połączmy te dwie pętle sortując wewnętrznie naprzemiennie w kierunku normalnym i odwrotnym, a otrzymamy algorytm dwukierunkowego sortowania bąbelkowego.

Wykonanie pętli sortującej w normalnym kierunku ustali maksymalną pozycję w zbiorze, od której powinna rozpocząć sortowanie pętla odwrotna. Ta z kolei ustali minimalną pozycję w zbiorze, od której powinna rozpocząć sortowanie pętla normalna w następnym obiegu pętli zewnętrznej. Sortowanie możemy zakończyć, jeśli nie wystąpiła potrzeba zamiany elementów w żadnej z tych dwóch pętli.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do n.

Dane wyjściowe

d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do n.

Zmienne pomocnicze

i - zmienna sterująca pętli, $i \in N$

p_{min} - dolna granica pozycji sortowanych elementów, $p_{min} \in N$

p_{max} - górna granica pozycji sortowanych elementów, $p_{max} \in N$

p - numer pozycji zamiany elementów, $p \in N$

Lista kroków

Operacja sortująca

K01: **Jeśli** d[i] ≤ d[i + 1], **to zakończ operację sortującą**

K02: d[i] ↔ d[i + 1]

K03: $p \leftarrow i$

K04: **Zakończ operację sortującą**

Algorytm główny

K01: $p_{min} \leftarrow 1$; $p_{max} \leftarrow n - 1$

K02: $p \leftarrow 0$

K03: **Dla** $i = p_{min}, p_{min} + 1, ..., p_{max}$: **wykonuj operację sortującą**

K04: **Jeśli** $p = 0$, **to zakończ**

K05: $p_{max} \leftarrow p - 1$; $p \leftarrow 0$

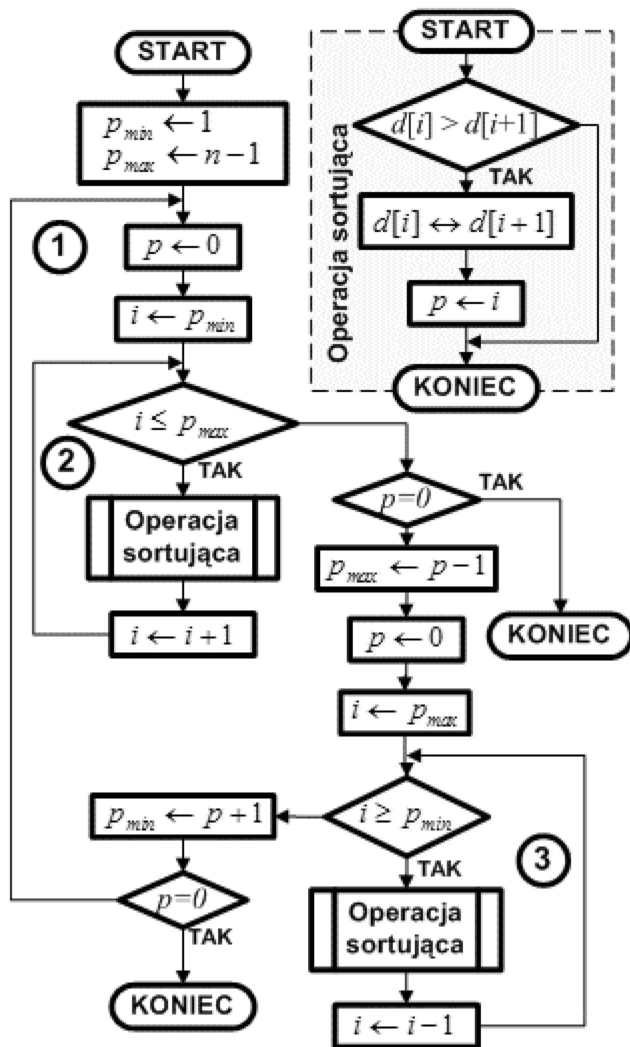
K06: **Dla** $i = p_{max}, p_{max} - 1, ..., p_{min}$: **wykonuj operację sortującą**

K07: $p_{min} \leftarrow p + 1$

K08: **Jeśli** $p > 0$, **to idź do** K02

K09: **Zakończ**

Schemat blokowy



W algorytmie wydzieliliśmy powtarzający się fragment operacji i nazwaliśmy go operacją sortującą. Porównuje ona dwa kolejne elementy zbioru i zamienia je miejscami, jeśli są w złej kolejności. Po zamianie do zmiennej p trafia indeks pierwszego z elementów pary. Podany warunek sprawdza uporządkowanie rosnące. Jeśli chcemy posortować zbiór malejąco, relację większości należy zastąpić relacją mniejszości.

W algorytmie występują trzy pętle. Pętla nr 1 jest pętlą warunkową i obejmuje dwie pętle wewnętrzne nr 2 i nr 3. Pętla ta wykonywana jest dotąd, aż w sortowanym zbiorze nie wystąpi w trakcie sortowania ani jedna zamiana miejsc elementów.

Pętla nr 2 jest pętlą sortującą w górę. Pętla nr 3 sortuje w dół.

Na początku algorytmu ustalamy dwie granice sortowania:

- dolną w p_{min}

- górną w p_{max}.

Granice te określają indeksy elementów zbioru, które będą przeglądały pętle sortujące nr 2 i nr 3. Początkowo granice są tak ustawione, iż obejmują cały sortowany zbiór.

Na początku pętli nr 1 zerujemy zmienną p. Zmienna ta będzie zapamiętywać pozycję ostatniej zamiany elementów. Jeśli po przejściu pętli sortującej nr 2 lub nr 3 zmienna p wciąż zawiera wartość 0, to nie wystąpiła żadna zamiana elementów. Zbiór jest wtedy uporządkowany i kończymy algorytm.

Pierwszą pętlę sortującą wykonujemy kolejno dla indeksów od p_{min} do p_{max}. Po zakończeniu pętli sprawdzamy, czy p jest równe 0. Jeśli tak, kończymy algorytm. W przeciwnym razie p zawiera pozycję ostatniej zamiany elementów. Ponieważ pętla nr 2 ustala pozycję najstarszego elementu, to elementy o indeksach od p do n są już właściwie uporządkowane. Dlatego dla kolejnego obiegu pętli przyjmujemy p_{max} o 1 mniejsze niż p.

Przed rozpoczęciem drugiej pętli zerujemy p. Jest to dosyć ważne. Załóżmy, iż zbiór został już uporządkowany w poprzedniej pętli nr 2, lecz wystąpiła tam zamiana elementów. Jeśli nie wyzerujemy p, to następna pętla sortująca nie zmieni zawartości tej zmiennej i p zachowa wartość większą od 0. Zatem pętla główna nr 1 nie zakończy się i algorytm wykona niepotrzebnie jeszcze jeden pusty obieg. Niby nic, a jednak...

Pętla nr 3 sortuje w kierunku odwrotnym od p_{max} do p_{min}. Po jej zakończeniu p zawiera indeks ostatniej zamiany elementów. Podobnie jak poprzednio zbiór jest uporządkowany od elementu nr 1 do p. Zatem dla kolejnych obiegów przyjmujemy p_{min} o 1 większe od p. Jeśli p jest równe 0, kończymy algorytm. W przeciwnym razie kontynuujemy pętlę nr 1.

Podsumowanie

Analizując wyniki obliczeń w arkuszu kalkulacyjnym otrzymanych czasów sortowania dla algorytmu dwukierunkowego sortowania bąbelkowego wyciągamy następujące wnioski:

Cechy Algorytmu Dwukierunkowego Sortowania Bąbelkowego	
klasa złożoności obliczeniowej optymistyczna	$O(n)$
klasa złożoności obliczeniowej typowa	$O(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$O(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Klasy złożoności obliczeniowej szacujemy następująco:

- **optymistyczna** - dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoich miejscach) - na podstawie czasów t_{po}, t_{pp}, t_{pk}
- **typowa** - dla zbiorów o losowym rozkładzie elementów - na podstawie czasu t_{np}
- **pesymistyczna** - dla zbiorów posortowanych odwrotnie - na podstawie czasu t_{od}.

Własności algorytmu					
Algorytm	**t_{po}**	**t_{od}**	**t_{pp}**	**t_{pk}**	**t_{np}**
Dwukierunkowe sortowanie bąbelkowe	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n^2)$
	$t_{po} << t_{od}$		$t_{pp} \approx \frac{9t_p}{8_k}$		$t_{np} \approx \frac{3t_o}{5_d}$

1. W przypadku zbioru posortowanego odwrotnie oraz zbioru nieuporządkowanego algorytm wykonuje się w czasie proporcjonalnym do kwadratu liczby elementów, zatem jego typowa czasowa złożoność obliczeniowa jest klasy $O(n^2)$.
2. Zbiór nieuporządkowany sortowany jest szybciej od zbioru uporządkowanego odwrotnie. Wnioskujemy zatem, iż zbiór uporządkowany odwrotnie jest najtrudniejszym zadaniem dla tego algorytmu.
3. Przy sortowaniu zbiorów w znacznym stopniu posortowanych klasa czasowej złożoności obliczeniowej redukuje się do $O(n)$, jest zatem bardzo korzystna.
4. W porównaniu do zoptymalizowanego algorytmu bąbelkowego w wersji 4 algorytm dwukierunkowego sortowania bąbelkowego nie wykazuje specjalnie nowych własności. Obserwujemy jedynie wyrównanie czasów sortowania zbioru uporządkowanego z jednym elementem losowym na początku i na końcu.

5.

Wzrost prędkości sortowania					
Algorytmy	t_{po}	t_{od}	t_{pp}	t_{pk}	t_{np}
Sortowanie bąbelkowe 4 Dwukierunkowe sortowanie bąbelkowe	≈ 1	$\approx \frac{7}{6}$	≈ 1	$\approx \frac{7}{10}$	$\approx \frac{6}{5}$
	brak	lepiej	brak	gorzej	lepiej

5. Jeśli porównamy prędkości obu algorytmów, to obecna wersja radzi sobie nieco lepiej w przypadku zbiorów uporządkowanych odwrotnie oraz zbiorów nieuporządkowanych. W pozostałych sytuacjach jest bez zmian, lub nawet gorzej.

Zadania dla ambitnych

1. Uzasadnij wyrównanie czasów t_{pp} i t_{pk} w algorytmie dwukierunkowego sortowania bąbelkowego.