Algorytmy i struktury danych - Problem Sortowania

Marcin Żurowski

28 marca 2025

Problem sortowania

- Sortowanie polega na ustawianiu w zadanej kolejności elementów pewnego ciągu.
- Domyślnie przyjmujemy, że należy ustawić liczby zawarte w tablicy `A[1..n]` w porządku niemalejącym.
- Sortowanie w miejscu oznacza, że wszystkie oprócz stałej liczby elementów znajdują się cały czas w tej samej tablicy.
- Algorytm sortowania jest przyrostowy, jeśli po każdym głównym kroku tego algorytmu rośnie rozmiar posortowanej części tablicy.

Problem sortowania

- Sortowanie polega na ustawianiu w zadanej kolejności elementów pewnego ciągu.
- Domyślnie przyjmujemy, że należy ustawić liczby zawarte w tablicy `A[1..n]` w porządku niemalejącym.
- Sortowanie w miejscu oznacza, że wszystkie oprócz stałej liczby elementów znajdują się cały czas w tej samej tablicy.
- Algorytm sortowania jest przyrostowy, jeśli po każdym głównym kroku tego algorytmu rośnie rozmiar posortowanej części tablicy.

Problem sortowania

- Sortowanie polega na ustawianiu w zadanej kolejności elementów pewnego ciągu.
- Domyślnie przyjmujemy, że należy ustawić liczby zawarte w tablicy `A[1..n]` w porządku niemalejącym.
- Sortowanie w miejscu oznacza, że wszystkie oprócz stałej liczby elementów znajdują się cały czas w tej samej tablicy.
- Algorytm sortowania jest przyrostowy, jeśli po każdym głównym kroku tego algorytmu rośnie rozmiar posortowanej części tablicy.

Problem sortowania

- Sortowanie polega na ustawianiu w zadanej kolejności elementów pewnego ciągu.
- Domyślnie przyjmujemy, że należy ustawić liczby zawarte w tablicy `A[1..n]` w porządku niemalejącym.
- Sortowanie w miejscu oznacza, że wszystkie oprócz stałej liczby elementów znajdują się cały czas w tej samej tablicy.
- Algorytm sortowania jest przyrostowy, jeśli po każdym głównym kroku tego algorytmu rośnie rozmiar posortowanej części tablicy.

Sortowanie przez wstawianie (insertion sort)

- Mając posortowaną tablicę `A[1..j-1]`, wstawiamy element `A[j]` na właściwe miejsce w tej tablicy, otrzymując posortowaną tablicę `A[1..j]`
- powtarzamy powyższy proces dla $j = 2, \ldots, n$.
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie przez wstawianie (insertion sort)

- Mając posortowaną tablicę `A[1..j-1]`, wstawiamy element `A[j]` na właściwe miejsce w tej tablicy, otrzymując posortowaną tablicę `A[1..j]`
- powtarzamy powyższy proces dla $j = 2, \ldots, n$.
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie przez wstawianie (insertion sort)

- Mając posortowaną tablicę `A[1..j-1]`, wstawiamy element `A[j]` na właściwe miejsce w tej tablicy, otrzymując posortowaną tablicę `A[1..j]`
- powtarzamy powyższy proces dla $j = 2, \ldots, n$.
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie przez wstawianie (insertion sort)

- Mając posortowaną tablicę `A[1..j-1]`, wstawiamy element `A[j]` na właściwe miejsce w tej tablicy, otrzymując posortowaną tablicę `A[1..j]`
- powtarzamy powyższy proces dla $j = 2, \ldots, n$.
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie przez wstawianie

```
procedure SORT-W(A, n)
  int array A[1..n]
  int n, r
 for j = 2 to n
  r = A[j]
  i = j - 1
  while i > 0 and A[i]>r
   A[i + 1] = A[i]
   i = i - 1
  A[i + 1] = r
```

Sortowanie przez wstawianie

- W każdej iteracji pętli **for** wykonana jest pętla **while** oraz pewna stała liczba operacji.
- Liczba iteracji pętli **while** zależy od kolejności danych wejściowych.
- W każdej iteracji pętli **while** wykonana jest stałą liczba operacji.
- Niech t_j oznacza liczbę sprawdzeń warunku pętli **while** dla $j = 2, \ldots, n$.
- Złożoność algorytmu $\sum_{j=2}^{n}(\Theta(1) + t_j\Theta(1)) = \Theta(\sum_{j=2}^{n} t_j)$.

Sortowanie przez wstawianie

- W każdej iteracji pętli **for** wykonana jest pętla **while** oraz pewna stała liczba operacji.
- Liczba iteracji pętli **while** zależy od kolejności danych wejściowych.
- W każdej iteracji pętli **while** wykonana jest stałą liczba operacji.
- Niech t_j oznacza liczbę sprawdzeń warunku pętli **while** dla $j = 2, \ldots, n$.
- Złożoność algorytmu $\sum_{j=2}^{n}(\Theta(1) + t_j\Theta(1)) = \Theta(\sum_{j=2}^{n} t_j)$.

Sortowanie przez wstawianie

- W każdej iteracji pętli **for** wykonana jest pętla **while** oraz pewna stała liczba operacji.
- Liczba iteracji pętli **while** zależy od kolejności danych wejściowych.
- W każdej iteracji pętli **while** wykonana jest stałą liczba operacji.
- Niech t_j oznacza liczbę sprawdzeń warunku pętli **while** dla $j = 2, \ldots, n$.
- Złożoność algorytmu $\sum_{j=2}^{n}(\Theta(1) + t_j\Theta(1)) = \Theta(\sum_{j=2}^{n} t_j)$.

Sortowanie przez wstawianie

- W każdej iteracji pętli **for** wykonana jest pętla **while** oraz pewna stała liczba operacji.
- Liczba iteracji pętli **while** zależy od kolejności danych wejściowych.
- W każdej iteracji pętli **while** wykonana jest stałą liczba operacji.
- Niech t_j oznacza liczbę sprawdzeń warunku pętli **while** dla $j = 2, \ldots, n$.
- Złożoność algorytmu $\sum_{j=2}^{n}(\Theta(1) + t_j\Theta(1)) = \Theta(\sum_{j=2}^{n} t_j)$.

Sortowanie przez wstawianie

- W każdej iteracji pętli **for** wykonana jest pętla **while** oraz pewna stała liczba operacji.
- Liczba iteracji pętli **while** zależy od kolejności danych wejściowych.
- W każdej iteracji pętli **while** wykonana jest stałą liczba operacji.
- Niech t_j oznacza liczbę sprawdzeń warunku pętli **while** dla $j = 2, \ldots, n$.
- Złożoność algorytmu $\sum_{j=2}^{n}(\Theta(1) + t_j\Theta(1)) = \Theta(\sum_{j=2}^{n} t_j)$.

Sortowanie przez wstawianie

- Złożoność optymistyczna:
  Dane wejściowe są posortowane niemalejąco
  $B(n) = \Theta(n)$.
- Złożoność pesymistyczna:
  Dane wejściowe są posortowane malejąco
  $W(n) = \Theta(n^2)$.

Sortowanie przez wstawianie

- Złożoność optymistyczna:
  Dane wejściowe są posortowane niemalejąco
  $B(n) = \Theta(n)$.
- Złożoność pesymistyczna:
  Dane wejściowe są posortowane malejąco
  $W(n) = \Theta(n^2)$.

Sortowanie bąbelkowe (bubble sort)

- Przeglądamy tablice `A[1..n]`, zamieniając miejscami dwie sąsiednie liczby, jeśli są ustanowione w niewłaściwej kolejności.
- Po przejściu całej tablicy jej największy element znajdzie się na ostatniej pozycji.
- Powtarzamy powyższy proces dla coraz krótszych podtablic: `A[1..n-1]`, `A[1..n-2]`,...,`A[1..2]`
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie bąbelkowe (bubble sort)

- Przeglądamy tablice `A[1..n]`, zamieniając miejscami dwie sąsiednie liczby, jeśli są ustanowione w niewłaściwej kolejności.
- Po przejściu całej tablicy jej największy element znajdzie się na ostatniej pozycji.
- Powtarzamy powyższy proces dla coraz krótszych podtablic: `A[1..n-1]`, `A[1..n-2]`,...,`A[1..2]`
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie bąbelkowe (bubble sort)

- Przeglądamy tablice `A[1..n]`, zamieniając miejscami dwie sąsiednie liczby, jeśli są ustanowione w niewłaściwej kolejności.
- Po przejściu całej tablicy jej największy element znajdzie się na ostatniej pozycji.
- Powtarzamy powyższy proces dla coraz krótszych podtablic: `A[1..n-1]`, `A[1..n-2]`,...,`A[1..2]`
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie bąbelkowe (bubble sort)

- Przeglądamy tablice `A[1..n]`, zamieniając miejscami dwie sąsiednie liczby, jeśli są ustanowione w niewłaściwej kolejności.
- Po przejściu całej tablicy jej największy element znajdzie się na ostatniej pozycji.
- Powtarzamy powyższy proces dla coraz krótszych podtablic: `A[1..n-1]`, `A[1..n-2]`,...,`A[1..2]`
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie bąbelkowe (bubble sort)

- Przeglądamy tablice `A[1..n]`, zamieniając miejscami dwie sąsiednie liczby, jeśli są ustanowione w niewłaściwej kolejności.
- Po przejściu całej tablicy jej największy element znajdzie się na ostatniej pozycji.
- Powtarzamy powyższy proces dla coraz krótszych podtablic: `A[1..n-1]`, `A[1..n-2]`,...,`A[1..2]`
- Sortowanie w miejscu.
- Metoda przyrostowa.

Sortowanie bąbelkowe

```
procedure SORT-B(A, n)
  int array A[1..n]
  int n, r
 for i = n - 1 downto 1
  for j = 1 to i
   if A[j] > A[j + 1]
    A[j] <-> A[j + 1]
```

Sortowanie bąbelkowe

- W każdej iteracji zewnętrznej pętli **for** wykonywana jest wewnętrzna pętla **for**.
- Liczba iteracji wewnętrznej pętli **for** wynosi i dla $i = n - 1, \ldots, 1$.
- W każdej iteracji wewnętrznej pętli **for** wykonana jest stała liczba operacji.
- Złożoność algorytmu: $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$ w przypadku optymistycznym i pesymistycznym.

Sortowanie bąbelkowe

- W każdej iteracji zewnętrznej pętli **for** wykonywana jest wewnętrzna pętla **for**.
- Liczba iteracji wewnętrznej pętli **for** wynosi i dla $i = n - 1, \ldots, 1$.
- W każdej iteracji wewnętrznej pętli **for** wykonana jest stała liczba operacji.
- Złożoność algorytmu: $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$ w przypadku optymistycznym i pesymistycznym.

Sortowanie bąbelkowe

- W każdej iteracji zewnętrznej pętli **for** wykonywana jest wewnętrzna pętla **for**.
- Liczba iteracji wewnętrznej pętli **for** wynosi i dla $i = n - 1, \ldots, 1$.
- W każdej iteracji wewnętrznej pętli **for** wykonana jest stała liczba operacji.
- Złożoność algorytmu: $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$ w przypadku optymistycznym i pesymistycznym.

Sortowanie bąbelkowe

- W każdej iteracji zewnętrznej pętli **for** wykonywana jest wewnętrzna pętla **for**.
- Liczba iteracji wewnętrznej pętli **for** wynosi i dla $i = n - 1, \ldots, 1$.
- W każdej iteracji wewnętrznej pętli **for** wykonana jest stała liczba operacji.
- Złożoność algorytmu: $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$ w przypadku optymistycznym i pesymistycznym.

Sortowanie przez scalanie (merge sort)

- Podzielić tablicę na dwie podtablice.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Scal posortowane podtablice w jedną posortowaną tablicę.
- Metoda dziel i zwyciężaj.
- Elementy nie są sortowane w miejscu.

Sortowanie przez scalanie (merge sort)

- Podzielić tablicę na dwie podtablice.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Scal posortowane podtablice w jedną posortowaną tablicę.
- Metoda dziel i zwyciężaj.
- Elementy nie są sortowane w miejscu.

Sortowanie przez scalanie (merge sort)

- Podzielić tablicę na dwie podtablice.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Scal posortowane podtablice w jedną posortowaną tablicę.
- Metoda dziel i zwyciężaj.
- Elementy nie są sortowane w miejscu.

Sortowanie przez scalanie (merge sort)

- Podzielić tablicę na dwie podtablice.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Scal posortowane podtablice w jedną posortowaną tablicę.
- Metoda dziel i zwyciężaj.
- Elementy nie są sortowane w miejscu.

Sortowanie przez scalanie (merge sort)

- Podzielić tablicę na dwie podtablice.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Scal posortowane podtablice w jedną posortowaną tablicę.
- Metoda dziel i zwyciężaj.
- Elementy nie są sortowane w miejscu.

Scalanie posortowanych tablic - idea

- Scalamy posortowane niemalejąco tablice `B[1..n]` o `C[1..m]`
- Aby znaleźć najmniejszy element (w obu tablicach łącznie), wystarczy porównać `B[1]` z `C[1]`. Wybrany element przepisujemy do tablicy wynikowej.
- W każdym kroku algorytmu wybieramy mniejszy z dwóch elementów: pierwszego jeszcze niewybranego z tablicy `B` i pierwszego niewybranego z tablicy `C`.
- Co zrobić, kiedy przepiszemy wszystkie elementy z którejś z tablic?
  przepisujemy wszystkie elementy z drugiej tablicy.
- Jeśli chcemy scalić części tablicy `A`, to najpierw przepisujem je do tablicy `B` i `C`.

Scalanie posortowanych tablic - idea

- Scalamy posortowane niemalejąco tablice `B[1..n]` o `C[1..m]`
- Aby znaleźć najmniejszy element (w obu tablicach łącznie), wystarczy porównać `B[1]` z `C[1]`. Wybrany element przepisujemy do tablicy wynikowej.
- W każdym kroku algorytmu wybieramy mniejszy z dwóch elementów: pierwszego jeszcze niewybranego z tablicy `B` i pierwszego niewybranego z tablicy `C`.
- Co zrobić, kiedy przepiszemy wszystkie elementy z którejś z tablic?
  przepisujemy wszystkie elementy z drugiej tablicy.
- Jeśli chcemy scalić części tablicy `A`, to najpierw przepisujem je do tablicy `B` i `C`.

Scalanie posortowanych tablic - idea

- Scalamy posortowane niemalejąco tablice `B[1..n]` o `C[1..m]`
- Aby znaleźć najmniejszy element (w obu tablicach łącznie), wystarczy porównać `B[1]` z `C[1]`. Wybrany element przepisujemy do tablicy wynikowej.
- W każdym kroku algorytmu wybieramy mniejszy z dwóch elementów: pierwszego jeszcze niewybranego z tablicy `B` i pierwszego niewybranego z tablicy `C`.
- Co zrobić, kiedy przepiszemy wszystkie elementy z którejś z tablic?
  przepisujemy wszystkie elementy z drugiej tablicy.
- Jeśli chcemy scalić części tablicy `A`, to najpierw przepisujem je do tablicy `B` i `C`.

Scalanie posortowanych tablic - idea

- Scalamy posortowane niemalejąco tablice `B[1..n]` o `C[1..m]`
- Aby znaleźć najmniejszy element (w obu tablicach łącznie), wystarczy porównać `B[1]` z `C[1]`. Wybrany element przepisujemy do tablicy wynikowej.
- W każdym kroku algorytmu wybieramy mniejszy z dwóch elementów: pierwszego jeszcze niewybranego z tablicy `B` i pierwszego niewybranego z tablicy `C`.
- Co zrobić, kiedy przepiszemy wszystkie elementy z którejś z tablic?
  przepisujemy wszystkie elementy z drugiej tablicy.
- Jeśli chcemy scalić części tablicy `A`, to najpierw przepisujem je do tablicy `B` i `C`.

Scalanie posortowanych tablic - idea

- Scalamy posortowane niemalejąco tablice `B[1..n]` o `C[1..m]`
- Aby znaleźć najmniejszy element (w obu tablicach łącznie), wystarczy porównać `B[1]` z `C[1]`. Wybrany element przepisujemy do tablicy wynikowej.
- W każdym kroku algorytmu wybieramy mniejszy z dwóch elementów: pierwszego jeszcze niewybranego z tablicy `B` i pierwszego niewybranego z tablicy `C`.
- Co zrobić, kiedy przepiszemy wszystkie elementy z którejś z tablic?
  przepisujemy wszystkie elementy z drugiej tablicy.
- Jeśli chcemy scalić części tablicy `A`, to najpierw przepisujem je do tablicy `B` i `C`.

Scalanie

```
procedure SCAL(A, p, q, r)
  int array A[1..n], array B[1..r]
  int p, q, r, i, j, k
 i = p
 j = q + 1
 k = p
 while (i <= q) and (j <= r)
  if A[i] <= A[j]
   B[k] = A[i]
   i = i + 1
  else
   B[k] = A[j]
   j = j + 1
  k = k + 1
```

Scalanie cd.

```
while i <= q
 B[k] = A[i]
 i = i + 1
 k = k + 1
while j <= r
 B[k] = A[j]
 j = j + 1
 k = k + 1
for i = p to r
 A[i] = B[i]
```

Sortowanie przez scalanie

```
procedure SORT-SCAL(A, p, r)
  int array A[1..n]
  int p, q, r
 if p < r
  q = (p + r) DIV 2
  SORT-SCAL(A, p, q)
  SORT-SCAL(A, q + 1, r)
  SCAL(A, p, q, r)
```

Sortowanie przez scalanie - złożoność

- Podział zadania na podproblemy (znalezienie środka tablicy): czas $\Theta(1)$.
- Rozwiązanie rekurencyjne dwóch podproblemów rozmiaru $\frac{n}{2}$: czas $2 \cdot T(\frac{n}{2})$.
- Połączenie rozwiązań podproblemów (scalanie): czas $\Theta(n)$.
- Otrzymujemy rekurencję $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalne $T(n) = \Theta(n \cdot \lg n)$.

Sortowanie przez scalanie - złożoność

- Podział zadania na podproblemy (znalezienie środka tablicy): czas $\Theta(1)$.
- Rozwiązanie rekurencyjne dwóch podproblemów rozmiaru $\frac{n}{2}$: czas $2 \cdot T(\frac{n}{2})$.
- Połączenie rozwiązań podproblemów (scalanie): czas $\Theta(n)$.
- Otrzymujemy rekurencję $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalne $T(n) = \Theta(n \cdot \lg n)$.

Sortowanie przez scalanie - złożoność

- Podział zadania na podproblemy (znalezienie środka tablicy): czas $\Theta(1)$.
- Rozwiązanie rekurencyjne dwóch podproblemów rozmiaru $\frac{n}{2}$: czas $2 \cdot T(\frac{n}{2})$.
- Połączenie rozwiązań podproblemów (scalanie): czas $\Theta(n)$.
- Otrzymujemy rekurencję $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalne $T(n) = \Theta(n \cdot \lg n)$.

Sortowanie przez scalanie - złożoność

- Podział zadania na podproblemy (znalezienie środka tablicy): czas $\Theta(1)$.
- Rozwiązanie rekurencyjne dwóch podproblemów rozmiaru $\frac{n}{2}$: czas $2 \cdot T(\frac{n}{2})$.
- Połączenie rozwiązań podproblemów (scalanie): czas $\Theta(n)$.
- Otrzymujemy rekurencję $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalne $T(n) = \Theta(n \cdot \lg n)$.

Sortowanie przez scalanie - złożoność

- Podział zadania na podproblemy (znalezienie środka tablicy): czas $\Theta(1)$.
- Rozwiązanie rekurencyjne dwóch podproblemów rozmiaru $\frac{n}{2}$: czas $2 \cdot T(\frac{n}{2})$.
- Połączenie rozwiązań podproblemów (scalanie): czas $\Theta(n)$.
- Otrzymujemy rekurencję $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalne $T(n) = \Theta(n \cdot \lg n)$.

Sortowanie szybkie (quicksort)

- Podziel tablicę `A[p..r]` na dwie podtablice `A[p..q - 1]` i `A[q + 1..r]`, takie że elementy tablicy `A[p..q - 1]` są nie większe niż **element rozdzielający** `A[q]`, a elementy tablicy `A[q + 1..r]` są nie mniejsze niż `A[q]`.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Metoda dziel i zwyciężaj.
- Sortowanie w miejscu.

Sortowanie szybkie (quicksort)

- Podziel tablicę `A[p..r]` na dwie podtablice `A[p..q - 1]` i `A[q + 1..r]`, takie że elementy tablicy `A[p..q - 1]` są nie większe niż **element rozdzielający** `A[q]`, a elementy tablicy `A[q + 1..r]` są nie mniejsze niż `A[q]`.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Metoda dziel i zwyciężaj.
- Sortowanie w miejscu.

Sortowanie szybkie (quicksort)

- Podziel tablicę `A[p..r]` na dwie podtablice `A[p..q - 1]` i `A[q + 1..r]`, takie że elementy tablicy `A[p..q - 1]` są nie większe niż **element rozdzielający** `A[q]`, a elementy tablicy `A[q + 1..r]` są nie mniejsze niż `A[q]`.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Metoda dziel i zwyciężaj.
- Sortowanie w miejscu.

Sortowanie szybkie (quicksort)

- Podziel tablicę `A[p..r]` na dwie podtablice `A[p..q - 1]` i `A[q + 1..r]`, takie że elementy tablicy `A[p..q - 1]` są nie większe niż **element rozdzielający** `A[q]`, a elementy tablicy `A[q + 1..r]` są nie mniejsze niż `A[q]`.
- Posortuj rekurencyjnie każdą z tych podtablic.
- Metoda dziel i zwyciężaj.
- Sortowanie w miejscu.

Sortowanie szybkie - pseudokod

```
procedure QUICK-SORT(A, p, r)
  int array A[1..n]
  int p, r
 if p < r
  q = PODZIAŁ(A,p,r)
  QUICK-SORT(A, p, q - 1)
  QUICK-SORT(A, q + 1, r)
```

Sortowanie szybkie - podział

```
procedure PODZIAL(A, p, r)
  int array A[1..n]
  int p, r, x, i
 x = A[r]
 i = p - 1
 for j = p to r - 1
  if A[j] ≤ x
   i = i + 1
   A[i] <-> A[j]
 A[i + 1] <-> A[r]
 return i + 1
```

Sortowanie szybkie - złożoność pesymistyczna

- W przypadku pesymistycznym podziały tablicy są skrajnie niezrównoważone. Jedna z podtablic ma długość 0, a druga `n - 1`.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $W(n) = W(n-1) + \Theta(n)$.
- Stosując metodę podstawiania, otrzymujemy $W(n) = \Theta(n^2)$.
- Przypadek pesymistyczny zachodzi nie tylko wtedy, gdy tablica wejściowa jest posortowana nierosnaco, ale także gdy jest posortowana niemalejąco.

Sortowanie szybkie - złożoność pesymistyczna

- W przypadku pesymistycznym podziały tablicy są skrajnie niezrównoważone. Jedna z podtablic ma długość 0, a druga `n - 1`.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $W(n) = W(n-1) + \Theta(n)$.
- Stosując metodę podstawiania, otrzymujemy $W(n) = \Theta(n^2)$.
- Przypadek pesymistyczny zachodzi nie tylko wtedy, gdy tablica wejściowa jest posortowana nierosnaco, ale także gdy jest posortowana niemalejąco.

Sortowanie szybkie - złożoność pesymistyczna

- W przypadku pesymistycznym podziały tablicy są skrajnie niezrównoważone. Jedna z podtablic ma długość 0, a druga `n - 1`.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $W(n) = W(n-1) + \Theta(n)$.
- Stosując metodę podstawiania, otrzymujemy $W(n) = \Theta(n^2)$.
- Przypadek pesymistyczny zachodzi nie tylko wtedy, gdy tablica wejściowa jest posortowana nierosnaco, ale także gdy jest posortowana niemalejąco.

Sortowanie szybkie - złożoność pesymistyczna

- W przypadku pesymistycznym podziały tablicy są skrajnie niezrównoważone. Jedna z podtablic ma długość 0, a druga `n - 1`.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $W(n) = W(n-1) + \Theta(n)$.
- Stosując metodę podstawiania, otrzymujemy $W(n) = \Theta(n^2)$.
- Przypadek pesymistyczny zachodzi nie tylko wtedy, gdy tablica wejściowa jest posortowana nierosnaco, ale także gdy jest posortowana niemalejąco.

Sortowanie szybkie - złożoność pesymistyczna

- W przypadku pesymistycznym podziały tablicy są skrajnie niezrównoważone. Jedna z podtablic ma długość 0, a druga `n - 1`.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $W(n) = W(n-1) + \Theta(n)$.
- Stosując metodę podstawiania, otrzymujemy $W(n) = \Theta(n^2)$.
- Przypadek pesymistyczny zachodzi nie tylko wtedy, gdy tablica wejściowa jest posortowana nierosnaco, ale także gdy jest posortowana niemalejąco.

Sortowanie szybkie - złożoność optymistyczna

- W przypadku optymistycznym podziały tablicy są idealnie zrównoważone. Długość każdej podtablicy nie przekracza $\frac{n}{2}$.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $B(n) = 2 \cdot B(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalnej otrzymujemy $B(n) = \Theta(n \cdot \lg n)$.
- Można wykazać, że złożoność w przypadku średnim również wynosi $\Theta(n \cdot \lg n)$.

Sortowanie szybkie - złożoność optymistyczna

- W przypadku optymistycznym podziały tablicy są idealnie zrównoważone. Długość każdej podtablicy nie przekracza $\frac{n}{2}$.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $B(n) = 2 \cdot B(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalnej otrzymujemy $B(n) = \Theta(n \cdot \lg n)$.
- Można wykazać, że złożoność w przypadku średnim również wynosi $\Theta(n \cdot \lg n)$.

Sortowanie szybkie - złożoność optymistyczna

- W przypadku optymistycznym podziały tablicy są idealnie zrównoważone. Długość każdej podtablicy nie przekracza $\frac{n}{2}$.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $B(n) = 2 \cdot B(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalnej otrzymujemy $B(n) = \Theta(n \cdot \lg n)$.
- Można wykazać, że złożoność w przypadku średnim również wynosi $\Theta(n \cdot \lg n)$.

Sortowanie szybkie - złożoność optymistyczna

- W przypadku optymistycznym podziały tablicy są idealnie zrównoważone. Długość każdej podtablicy nie przekracza $\frac{n}{2}$.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $B(n) = 2 \cdot B(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalnej otrzymujemy $B(n) = \Theta(n \cdot \lg n)$.
- Można wykazać, że złożoność w przypadku średnim również wynosi $\Theta(n \cdot \lg n)$.

Sortowanie szybkie - złożoność optymistyczna

- W przypadku optymistycznym podziały tablicy są idealnie zrównoważone. Długość każdej podtablicy nie przekracza $\frac{n}{2}$.
- Funkcja `PODZIAŁ` działa w czasie $\Theta(n)$.
- Zatem $B(n) = 2 \cdot B(\frac{n}{2}) + \Theta(n)$.
- Z twierdzenia o rekurencji uniwersalnej otrzymujemy $B(n) = \Theta(n \cdot \lg n)$.
- Można wykazać, że złożoność w przypadku średnim również wynosi $\Theta(n \cdot \lg n)$.

Sortowanie szybkie - randomizowana wersja

Zastępujemy funkcję PODZIAŁ następująca funkcją: procedure

```
LOSOWY-PODZIAŁ(A, p, r)
  int array A[1..n]
  int p, r, x, i
 i = RANDOM(p, r)
 A[r] <-> A[i]
 return PODZIAŁ(A, p, r)
```

Oczekiwany czas działania algorytmu dla dowolnej tablicy zawierającej różne elementy wynosi $\Theta(n \cdot \lg n)$.

Sortowanie przez zliczanie (counting sort)

- Założenie: Sortujemy tablicę `A[1..n]` zawierającą liczby całkowite z zakresu od 0 do `k`.
- W pomocniczej tablicy `C[0..k]` dla każdego elementu tablicy `A` obliczmy, ile jest elementów nie większych od niego.
- Na podstawie tej informacji obliczamy pozycję, którą dany element powinien zająć w posortowanej tablicy. Uwaga: równe elementy nie mogą trafić na tę samą pozycję.
- Elementy nie są sortowane w miejscu. Wynik sortowania zapisujemy w tablicy `B[1..n]`.

Sortowanie przez zliczanie (counting sort)

- Założenie: Sortujemy tablicę `A[1..n]` zawierającą liczby całkowite z zakresu od `0` do `k`.
- W pomocniczej tablicy `C[0..k]` dla każdego elementu tablicy `A` obliczmy, ile jest elementów nie większych od niego.
- Na podstawie tej informacji obliczamy pozycję, którą dany element powinien zająć w posortowanej tablicy. Uwaga: równe elementy nie mogą trafić na tę samą pozycję.
- Elementy nie są sortowane w miejscu. Wynik sortowania zapisujemy w tablicy `B[1..n]`.

Sortowanie przez zliczanie (counting sort)

- Założenie: Sortujemy tablicę `A[1..n]` zawierającą liczby całkowite z zakresu od `0` do `k`.
- W pomocniczej tablicy `C[0..k]` dla każdego elementu tablicy `A` obliczmy, ile jest elementów nie większych od niego.
- Na podstawie tej informacji obliczamy pozycję, którą dany element powinien zająć w posortowanej tablicy. Uwaga: równe elementy nie mogą trafić na tę samą pozycję.
- Elementy nie są sortowane w miejscu. Wynik sortowania zapisujemy w tablicy `B[1..n]`.

Sortowanie przez zliczanie (counting sort)

- Założenie: Sortujemy tablicę `A[1..n]` zawierającą liczby całkowite z zakresu od `0` do `k`.
- W pomocniczej tablicy `C[0..k]` dla każdego elementu tablicy `A` obliczmy, ile jest elementów nie większych od niego.
- Na podstawie tej informacji obliczamy pozycję, którą dany element powinien zająć w posortowanej tablicy. Uwaga: równe elementy nie mogą trafić na tę samą pozycję.
- Elementy nie są sortowane w miejscu. Wynik sortowania zapisujemy w tablicy `B[1..n]`.

Sortowanie przez zliczanie - pseudokod

```
procedure COUNTING-SORT(A, B)
  int array A[1..n]
  int array B[1..n]
  int array C[0..k]
 for i = 0 to k
  C[i] = 0
 for j = 1 to n
  C[A[j]] = C[A[j]] + 1
 for i = 1 to k
  C[i] = C[i] + C[i - 1]
 for j = n downto 1
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

Sortowanie przez zliczanie - złożoność i stabilność

- Czas działania sortowania przez zliczanie wynosi $\Theta(n + k)$.
- Zatem jeśli $k = O(n)$, to sortowanie przez zliczanie działa w czasie $\Theta(n)$.
- Sortowanie przez zliczanie jest **stabilne**: równe elementy zachowują taką samą kolejność, jaką miały w wejściowej tablicy.
- Stabilność sortowania może być przydatna, kiedy z sortowanymi elementami ziązane są dodatkowe dane.

Sortowanie przez zliczanie - złożoność i stabilność

- Czas działania sortowania przez zliczanie wynosi $\Theta(n + k)$.
- Zatem jeśli $k = O(n)$, to sortowanie przez zliczanie działa w czasie $\Theta(n)$.
- Sortowanie przez zliczanie jest **stabilne**: równe elementy zachowują taką samą kolejność, jaką miały w wejściowej tablicy.
- Stabilność sortowania może być przydatna, kiedy z sortowanymi elementami ziązane są dodatkowe dane.

Sortowanie przez zliczanie - złożoność i stabilność

- Czas działania sortowania przez zliczanie wynosi $\Theta(n + k)$.
- Zatem jeśli $k = O(n)$, to sortowanie przez zliczanie działa w czasie $\Theta(n)$.
- Sortowanie przez zliczanie jest **stabilne**: równe elementy zachowują taką samą kolejność, jaką miały w wejściowej tablicy.
- Stabilność sortowania może być przydatna, kiedy z sortowanymi elementami ziązane są dodatkowe dane.

Sortowanie przez zliczanie - złożoność i stabilność

- Czas działania sortowania przez zliczanie wynosi $\Theta(n + k)$.
- Zatem jeśli $k = O(n)$, to sortowanie przez zliczanie działa w czasie $\Theta(n)$.
- Sortowanie przez zliczanie jest **stabilne**: równe elementy zachowują taką samą kolejność, jaką miały w wejściowej tablicy.
- Stabilność sortowania może być przydatna, kiedy z sortowanymi elementami ziązane są dodatkowe dane.

Poprawność algorytmów

- Algorytm `A` jest poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych algorytm `A` zatrzymuje się i daje poprawny wynik.
- Algorytm `A` jest częściowo poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych, jeżeli algorytm `A` zatrzymuje się, to daje poprawny wynik.
- Zdanie logiczne `p` nazywamy niezmiennikiem pętli `L`, jesli `p` jest prawdziwe przed rozpoczęciem wykonania `1` oraz `p` jest prawdzie po każdej iteracji tej pętli (lub, równoważnie, przed każdą iteracją).
- Pojęcie niezmiennika pętli jest użyteczne w dowodzeniu poprawności algorytmów.

Poprawność algorytmów

- Algorytm `A` jest poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych algorytm `A` zatrzymuje się i daje poprawny wynik.
- Algorytm `A` jest częściowo poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych, jeżeli algorytm `A` zatrzymuje się, to daje poprawny wynik.
- Zdanie logiczne `p` nazywamy niezmiennikiem pętli `L`, jesli `p` jest prawdziwe przed rozpoczęciem wykonania `1` oraz `p` jest prawdzie po każdej iteracji tej pętli (lub, równoważnie, przed każdą iteracją).
- Pojęcie niezmiennika pętli jest użyteczne w dowodzeniu poprawności algorytmów.

Poprawność algorytmów

- Algorytm `A` jest poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych algorytm `A` zatrzymuje się i daje poprawny wynik.
- Algorytm `A` jest częściowo poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych, jeżeli algorytm `A` zatrzymuje się, to daje poprawny wynik.
- Zdanie logiczne `p` nazywamy niezmiennikiem pętli `L`, jesli `p` jest prawdziwe przed rozpoczęciem wykonania `l` oraz `p` jest prawdzie po każdej iteracji tej pętli (lub, równoważnie, przed każdą iteracją).
- Pojęcie niezmiennika pętli jest użyteczne w dowodzeniu poprawności algorytmów.

Poprawność algorytmów

- Algorytm `A` jest poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych algorytm `A` zatrzymuje się i daje poprawny wynik.
- Algorytm `A` jest częściowo poprawny $\Leftrightarrow$ dla każdych poprawnych danych wejściowych, jeżeli algorytm `A` zatrzymuje się, to daje poprawny wynik.
- Zdanie logiczne `p` nazywamy niezmiennikiem pętli `L`, jesli `p` jest prawdziwe przed rozpoczęciem wykonania `l` oraz `p` jest prawdzie po każdej iteracji tej pętli (lub, równoważnie, przed każdą iteracją).
- Pojęcie niezmiennika pętli jest użyteczne w dowodzeniu poprawności algorytmów.

Przykład 1: poprawność sortowania przez wstawianie.

```
procedure SORT-W(A, n)
  int array A[1..n]
  int n, r
 for j = 2 to n
  r = A[j]
  i = j - 1
  while i > 0 and A[i] > r
   A[i + 1] = A[i]
   i = i - 1
  A[i + 1] = r
```

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli fragment tablicy `A[1..j-1]` składa się z elementów znajdujących się pierwotnie w `A[1..j-1]`, ale posortowanych niemalejąco.
- **Inicjalizacja**: na początku pierwszej iteracji pętli **for** mamy `j = 2`. Fragment tablicy `A[1..j - 1]` składa się z jednego elementu `A[1]`, który od początku znajdował się na tej pozycji, i oczywiście jest posortowany.

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli fragment tablicy `A[1..j-1]` składa się z elementów znajdujących się pierwotnie w `A[1..j-1]`, ale posortowanych niemalejąco.
- **Inicjalizacja**: na początku pierwszej iteracji pętli **for** mamy `j = 2`. Fragment tablicy `A[1..j - 1]` składa się z jednego elementu `A[1]`, który od początku znajdował się na tej pozycji, i oczywiście jest posortowany.

Poprawność algorytmów

- **Utrzymanie**: nieformalnie, pojedyncza iteracja pętli **for** polega na przesuwaniu elementów `A[j - 1]`, `A[j - 2]`, `A[j - 3]` itd. o jedną pozycję w prawo aż do znalezienia właściwego miejsca na `A[j]`. Zatem jeśli niezmiennik pętli był prawdziwy przed wykonaniem tej iteracji, to po jej wykonaniu fragment tablicy `A[1..j` zawiera posortowane elementy znajdujące się pierwotnie w tej podtablicy.
- Uwaga! Formalny dowód utrzymania niezmiennika pętli wymagałby sformułowania i udowodnienia niezmiennika pętli **while**.

Poprawność algorytmów

- **Utrzymanie**: nieformalnie, pojedyncza iteracja pętli **for** polega na przesuwaniu elementów `A[j - 1]`, `A[j - 2]`, `A[j - 3]` itd. o jedną pozycję w prawo aż do znalezienia właściwego miejsca na `A[j]`. Zatem jeśli niezmiennik pętli był prawdziwy przed wykonaniem tej iteracji, to po jej wykonaniu fragment tablicy `A[1..j` zawiera posortowane elementy znajdujące się pierwotnie w tej podtablicy.
- Uwaga! Formalny dowód utrzymania niezmiennika pętli wymagałby sformułowania i udowodnienia niezmiennika pętli **while**.

Poprawność algorytmów

- **Zakończenie**: pętli **for** kończy się, kiedy `j` osiąga wartość `n + 1`. Wstawiając `j = n + 1` w sformułowaniu niezmiennika pętli, wnioskujemy, że fragment `A[1..n]`, czyli cała tablica, zawiera znajdujące się w niej początkowo elementy, posortowane niemalejąco.
- Za pomocą niemziennika pętli pokazaliśmy, że jeśli algorytm sortowania się zakończy, to zwróci prawidłowy wynik. Ponadto pętla **for** ma dokładnie `n - 1` iteracji, a zawarta w niej pętli **while** wykona co najwyżej `j` iteracji dla każdej wartości `j` przyjmowanej w pętli **for**. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm sortowania przez wstawianie jest poprawny.

Poprawność algorytmów

- **Zakończenie**: pętli **for** kończy się, kiedy `j` osiąga wartość `n + 1`. Wstawiając `j = n + 1` w sformułowaniu niezmiennika pętli, wnioskujemy, że fragment `A[1..n]`, czyli cała tablica, zawiera znajdujące się w niej początkowo elementy, posortowane niemalejąco.
- Za pomocą niemziennika pętli pokazaliśmy, że jeśli algorytm sortowania się zakończy, to zwróci prawidłowy wynik. Ponadto pętla **for** ma dokładnie `n - 1` iteracji, a zawarta w niej pętli **while** wykona co najwyżej `j` iteracji dla każdej wartości `j` przyjmowanej w pętli **for**. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm sortowania przez wstawianie jest poprawny.

Poprawność algorytmów

- **Zakończenie**: pętli **for** kończy się, kiedy `j` osiąga wartość `n + 1`. Wstawiając `j = n + 1` w sformułowaniu niezmiennika pętli, wnioskujemy, że fragment `A[1..n]`, czyli cała tablica, zawiera znajdujące się w niej początkowo elementy, posortowane niemalejąco.
- Za pomocą niemziennika pętli pokazaliśmy, że jeśli algorytm sortowania się zakończy, to zwróci prawidłowy wynik. Ponadto pętla **for** ma dokładnie `n - 1` iteracji, a zawarta w niej pętli **while** wykona co najwyżej `j` iteracji dla każdej wartości `j` przyjmowanej w pętli **for**. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm sortowania przez wstawianie jest poprawny.

Przykład 2: poprawność funkcji PODZIAŁ wykorzystywanej w sortowaniu szybkim.

```
procedure PODZIAL(A, p, r)
  int array A[1..n]
  int p, r, x, i
 x = A[r]
 i = p - 1
 for j = p to r - 1
  if A[j] ≤ x
   i = i + 1
   A[i] <-> A[j]
 A[i + 1] <-> A[r]
 return i + 1
```

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli dla każdego indeksu `k` w tablicy `A[p..r]` zachodzą zależności:
  1. Jeśli $p \leq k \leq i$, to $A[k] \leq x$.
  2. Jeśli $i+1 \leq k \leq j-1$, to $A[k] > x$.
  3. Jeśli $k = r$, to $A[k] = x$.
- **Inicjalizacja**: przed pierwszą iteracją pętli **for** mamy $i = p-1$ oraz $j = p$. Nie istnieje więc takie k, że $p \leq k \leq i$ lub $i+1 \leq k \leq j-1$. Dla $k = r$ mamy $A[k] = x$ dzięki wykonaniu pierwszej instrukcji algorytmu.

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli dla każdego indeksu `k` w tablicy `A[p..r]` zachodzą zależności:
  1. Jeśli $p \leq k \leq i$, to $A[k] \leq x$.
  2. Jeśli $i+1 \leq k \leq j-1$, to $A[k] > x$.
  3. Jeśli $k = r$, to $A[k] = x$.
- **Inicjalizacja**: przed pierwszą iteracją pętli **for** mamy $i = p-1$ oraz $j = p$. Nie istnieje więc takie k, że $p \leq k \leq i$ lub $i+1 \leq k \leq j-1$. Dla $k = r$ mamy $A[k] = x$ dzięki wykonaniu pierwszej instrukcji algorytmu.

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli dla każdego indeksu `k` w tablicy `A[p..r]` zachodzą zależności:
  1. Jeśli $p \leq k \leq i$, to $A[k] \leq x$.
  2. Jeśli $i+1 \leq k \leq j-1$, to $A[k] > x$.
  3. Jeśli $k = r$, to $A[k] = x$.
- **Inicjalizacja**: przed pierwszą iteracją pętli **for** mamy $i = p-1$ oraz $j = p$. Nie istnieje więc takie k, że $p \leq k \leq i$ lub $i+1 \leq k \leq j-1$. Dla $k = r$ mamy $A[k] = x$ dzięki wykonaniu pierwszej instrukcji algorytmu.

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli dla każdego indeksu `k` w tablicy `A[p..r]` zachodzą zależności:
  1. Jeśli $p \leq k \leq i$, to $A[k] \leq x$.
  2. Jeśli $i+1 \leq k \leq j-1$, to $A[k] > x$.
  3. Jeśli $k = r$, to $A[k] = x$.
- **Inicjalizacja**: przed pierwszą iteracją pętli **for** mamy $i = p-1$ oraz $j = p$. Nie istnieje więc takie k, że $p \leq k \leq i$ lub $i+1 \leq k \leq j-1$. Dla $k = r$ mamy $A[k] = x$ dzięki wykonaniu pierwszej instrukcji algorytmu.

Poprawność algorytmów

- **Niezmiennik pętli for**: na początku każdej iteracji tej pętli dla każdego indeksu `k` w tablicy `A[p..r]` zachodzą zależności:
  1. Jeśli $p \leq k \leq i$, to $A[k] \leq x$.
  2. Jeśli $i+1 \leq k \leq j-1$, to $A[k] > x$.
  3. Jeśli $k = r$, to $A[k] = x$.
- **Inicjalizacja**: przed pierwszą iteracją pętli **for** mamy $i = p-1$ oraz $j = p$. Nie istnieje więc takie k, że $p \leq k \leq i$ lub $i+1 \leq k \leq j-1$. Dla $k = r$ mamy $A[k] = x$ dzięki wykonaniu pierwszej instrukcji algorytmu.

Poprawność algorytmów

- **Utrzymanie**: rozważmy dwa przypadki:
  1. $A[j] > x$. Wówczas jedynym działaniem wykonanym w pętli jest zwiększenie j. Po tym zwiększeniu dla $k = j - 1$ jest spełniony warunek $A[k] > x$. Warunki dla pozostałych elementów pozostają bez zmian, więc są spełnione.
  2. $A[j] \leq x$. Wówczas zwiększana jest wartość i, elementy $A[i]$ i $A[j]$ są zamieniane, a następnie zwiększa się j. Dla $k = i$ mamy $A[k] \leq x$ ze względu na dokonaną zamianę. Podobnie dla $k = j - 1$ mamy $A[k] > x$, ponieważ wstawiony na tę pozycję element pochodził z obszaru wartości większych od x.

Poprawność algorytmów

- **Utrzymanie**: rozważmy dwa przypadki:
  1. $A[j] > x$. Wówczas jedynym działaniem wykonanym w pętli jest zwiększenie j. Po tym zwiększeniu dla $k = j - 1$ jest spełniony warunek $A[k] > x$. Warunki dla pozostałych elementów pozostają bez zmian, więc są spełnione.
  2. $A[j] \leq x$. Wówczas zwiększana jest wartość i, elementy $A[i]$ i $A[j]$ są zamieniane, a następnie zwiększa się j. Dla $k = i$ mamy $A[k] \leq x$ ze względu na dokonaną zamianę. Podobnie dla $k = j - 1$ mamy $A[k] > x$, ponieważ wstawiony na tę pozycję element pochodził z obszaru wartości większych od x.

Poprawność algorytmów

- **Utrzymanie**: rozważmy dwa przypadki:
  1. $A[j] > x$. Wówczas jedynym działaniem wykonanym w pętli jest zwiększenie j. Po tym zwiększeniu dla $k = j - 1$ jest spełniony warunek $A[k] > x$. Warunki dla pozostałych elementów pozostają bez zmian, więc są spełnione.
  2. $A[j] \leq x$. Wówczas zwiększana jest wartość i, elementy $A[i]$ i $A[j]$ są zamieniane, a następnie zwiększa się j. Dla $k = i$ mamy $A[k] \leq x$ ze względu na dokonaną zamianę. Podobnie dla $k = j - 1$ mamy $A[k] > x$, ponieważ wstawiony na tę pozycję element pochodził z obszaru wartości większych od x.

Poprawność algorytmów

- **Zakończenie**: pętla **for** kończy się, kiedy j osiąga wartość r. Tablica została więc podzielona na trzy obszary: dla $p \leq k \leq i$ mamy $A[k] \leq x$, dla $i+1 \leq k \leq r-1$ mamy $A[k] > x$, oraz $A[r] = x$.
- Ostatnie dwie instrukcje powodują przestawienie elementu x na miejsce pomiędzy częściami tablicy zawierającymi elementy nie większe od x i elementy większe od x, a następnie zwrócenie pozycji elementu x. Otrzymujemy zatem poprawny podział tablicy.
- Pętla **for** wykonuje dokładnie $r - p$ iteracji. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm `PODZIAŁ` jest poprawny.

Poprawność algorytmów

- **Zakończenie**: pętla **for** kończy się, kiedy j osiąga wartość r. Tablica została więc podzielona na trzy obszary: dla $p \leq k \leq i$ mamy $A[k] \leq x$, dla $i+1 \leq k \leq r-1$ mamy $A[k] > x$, oraz $A[r] = x$.
- Ostatnie dwie instrukcje powodują przestawienie elementu x na miejsce pomiędzy częściami tablicy zawierającymi elementy nie większe od x i elementy większe od x, a następnie zwrócenie pozycji elementu x. Otrzymujemy zatem poprawny podział tablicy.
- Pętla **for** wykonuje dokładnie $r - p$ iteracji. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm `PODZIAŁ` jest poprawny.

Poprawność algorytmów

- **Zakończenie**: pętla **for** kończy się, kiedy j osiąga wartość r. Tablica została więc podzielona na trzy obszary: dla $p \leq k \leq i$ mamy $A[k] \leq x$, dla $i+1 \leq k \leq r-1$ mamy $A[k] > x$, oraz $A[r] = x$.
- Ostatnie dwie instrukcje powodują przestawienie elementu x na miejsce pomiędzy częściami tablicy zawierającymi elementy nie większe od x i elementy większe od x, a następnie zwrócenie pozycji elementu x. Otrzymujemy zatem poprawny podział tablicy.
- Pętla **for** wykonuje dokładnie $r - p$ iteracji. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm PODZIAŁ jest poprawny.

Poprawność algorytmów

- **Zakończenie**: pętla **for** kończy się, kiedy j osiąga wartość r. Tablica została więc podzielona na trzy obszary: dla $p \leq k \leq i$ mamy $A[k] \leq x$, dla $i+1 \leq k \leq r-1$ mamy $A[k] > x$, oraz $A[r] = x$.
- Ostatnie dwie instrukcje powodują przestawienie elementu x na miejsce pomiędzy częściami tablicy zawierającymi elementy nie większe od x i elementy większe od x, a następnie zwrócenie pozycji elementu x. Otrzymujemy zatem poprawny podział tablicy.
- Pętla **for** wykonuje dokładnie $r - p$ iteracji. Zatem algorytm zawsze się zatrzymuje.
- **Wniosek**: algorytm `PODZIAŁ` jest poprawny.