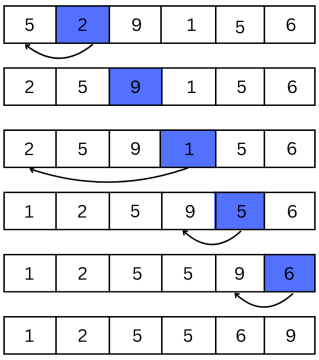
Algorytmy i Struktury Danych

Wojciech Typer

Algorithm 1 Insertion Sort

```
1:  procedure InsertionSort(A, n)
2:      for i = 1 to n − 1 do
3:          key = A[i]
4:          j = i − 1
5:          while j ≥ 0 and A[j] > key do
6:              A[j + 1] = A[j]
7:              j = j − 1
8:          end while
9:          A[j + 1] = key
10:     end for
11: end procedure
```

Złożoność czasowa: $O(n^2)$
Best case: w najlepszym przypadku złożoność czasowa będzie wynosić $O(n)$
Złożoność pamięciowa: $O(1)$



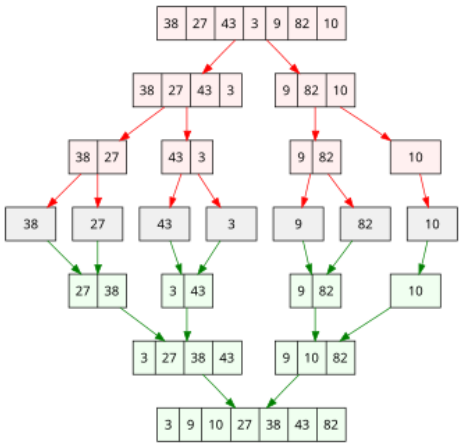
Algorithm 2 Merge Sort

```
1: procedure MergeSort(A, 1, n)
2:     if |A[1..n]| == 1 then
3:         return A[1..n]
4:     else
5:         B = MergeSort(A, 1, ⌊n/2⌋)
6:         C = MergeSort(A, ⌊n/2⌋, n)
7:         return Merge(B, C)
8:     end if
9: end procedure
```

Algorithm 3 Merge

```
1: procedure Merge(X[1..k], Y[1..n])
2:     if X = ∅ then
3:         return Y
4:     else if Y = ∅ then
5:         return X
6:     else if X[1] ≤ Y[1] then
7:         return [X[1]] × Merge(X[2..k], Y[1..n])
8:     else
9:         return [Y[1]] × Merge(X[1..k], Y[2..n])
10:    end if
11: end procedure
```

Złożoność czesowa Merge Sort: $O(n \log n)$
Złożoność pamięciowa Merge Sort: $O(n)$



Istnieje również iteracyjna wersja algorytmu Merge, sort, która została przedstawiona poniżej w postaci pseudokodu.

Algorithm 4 IterativeMergeSort

```
1: procedure IterativeMergeSort(A[1..n])
2:     for size = 1 to n − 1 by size × 2 do
3:         for left = 0 to n − 1 by 2 × size do
4:             mid ← min(left + size − 1, n − 1)
5:             right ← min(left + 2 × size − 1, n − 1)
6:             Merge(A, left, mid, right)
7:         end for
8:     end for
9: end procedure
```

Złożoność czasowa Iterative Merge Sort: $O(n \log n)$ - dzieje się tak, ponieważ size jest podwajany o 2 w każdej iteracji, więc potrzebujemy około $\log_2 n$ iteracji, a w każdej z nich wykonujemy $O(n)$ operacji.

Złożoność pamięciowa Iterative Merge Sort: $O(n)$