

## STABILNE

(Nazwa | Złożoność czasowa | Złożoność pamięciowa)

---

```
// Insertion sort | n^2 | O(1) - stała |
void insertionSort (double t[], int n) {
    // bierzemy kolejny element
    for (int k = 1; k < n; k++) {
        // sprawdzamy ze wszystkimi poprzednimi
        for(int i = k; i > 0 && t[i-1] > t[i]; i--) {
            std::swap(t[i], t[i-1]);    // zamieniamy miejscami
            // warunek porównujący tj. ... i > 0 && t[i-1] > t[i] ...
        }
    }
}
```

```
// Bucket Sort | O(n^2) | O(n) |
void bucketSort(double t[], n) {
    vector<double> b[n];      // Tworzymy puste kubełki

    // Umieszczamy elementy tablicy w różnych kubełkach
    for(int i = 0; i < n; i++) {
        int index = n * t[i]; // Indeks w kubełku
        b[index].push_back(t[i]);
    }

    // Sortujemy każdy kubełek
    for(int i = 0; i < n; i++) {
        sort(b[i].begin(), b[i].end());      // insertionSort
    }

    // Konwersja wektor -> Array
    int index = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < b[i].size(); j++) {
            t[index++] = b[i][j];
        }
    }
}
```

```

// Merge sort | O(nlogn) | O(n) |
const N = 30000;
double b[N];

void merge(int n, int k, double t[], double b[]) {
    int i = 0, j = k, l = 0;

    while(i < k && j < n) {
        if(t[i] <= t[j]) {
            b[l++] = t[i++];
        } else {
            b[l++] = t[j++];
        }
    }
    while(i < k) { b[l++] = t[i++]; }
    for(i = 0; i < j; i++) { t[i] = b[i]; }
}

void mergeSort(double t[], int n) {
    if(n > 1) {
        int k = n/2;
        mergeSort(t, k);
        mergeSort(t+k, n-k);
        merge(n, k, t, b);
    }
}

// Counting sort | n^2 | O(1) - stała |
void countingSort (double t[], int n, int max) {
    int output[n];
    int i;
    int count[max] = {0};

    // zliczanie, liczba wystąpień liczb zapisywana
    for(int i = 0; i < n; i++) {
        count[int(t[i])]++;
    }
    // akumulacja, count[i] przechowuje pozycję liczby w output
    for(int i = 1; i < max; i++) {
        count[i] += count[i - 1];
    }
    // wpis do bufora, tworzenie końcowej tablicy
    for(int i = n-1; i >= 0; i--) {
        output[--count[int(t[i])]] = t[i];
    }
    // przypisanie, przeniesienie do tablicy
    for(int i = 0; i < n; i++) {
        t[i] = output[i];
    }
}

```

```

// Radix Sort | O(d(n+k)) | O(n+k) |
int maxValue(int t[], int n) {
    int m = t[0];
    for(int i = 1; i < n; i++) {
        if(t[i] > m) {
            m = t[i];
        }
    }
    return m;
}

void radixSort(int t[], int n) {
    int m = maxValue(t, n);

    for(int exp = 1; m/exp > 0; exp *= 10) {
        countingSort(t, n, exp);
    }
}

```

## NIESTABILNE

(Nazwa | Złożoność czasowa | Złożoność pamięciowa)

---

```

// Quick Sort | O(nlogn) Pesymistyczne: O(n^2) | O(logn) |
int partition (double t[], int n) {
    int k = -1;           // indeks pomocniczy od początku tablicy
    double x = t[n / 2]; // pivot

    // nieskończona pętla
    for(;;) {
        // przesuwamy n w lewo od końca, dopóki n-ty element jest większy od pivota
        while (t[--n] > x);
        // przesuwamy w prawo od początku, dopóki k-ty element jest mniejszy od pivota
        while (t[++k] < x);
        // zamieniamy n-ty i k-ty element miejscami, jeśli k < n
        if (k < n) {
            std::swap(t[k], t[n]);
        } else           // zwracamy indeks pivota
            return k;
    }
}

void quickSort (double t[], int n) {
    if (n > 1) {
        int k = partition (t, n); //podziel na dwie części
        quick_sort (t, k);      //posortuj lewa
        quick_sort (t+k, n-k);  //posortuj prawa
    }
}

```

```

// Heap Sort | O(nlogn) | O(1) |
void heapify(double t[], int n, int i) {
    int k;
    double x = t[i];
    while(true) {
        k = 2 * i + 2; // prawy syn x
        // prawy syn istnieje i jest większy od ojca i brata
        if( (k < n && t[k] > x && t[k] > t[k-1])
            || (--k < n && t[k] > x)) { //
            t[i] = t[k]; // podsuwamy syna w górę
            i = k; // przechodzimy w dół
        } else {
            t[i] = x; // zapis x w nowym miejscu
            return;
        }
    }
}

void heapSort(double t[], int n) {
    // Buildheap
    for(int i = n/2; i >= 0; i--) { heapify(t, n, i) }
    // Cleanheap
    while(--n) {
        std::swap(t[0], t[n]);
        heapify(t, n, 0);
    }
}

```

### Heapsort – indeksy lewego i prawego dziecka

Lewe	Prawe
2i+1	2i+2

### Złożoność build\_heap | O(n) | O(n) |

### Działanie procedury merge

Procedura merge polega na łączeniu dwóch zbiorów danych w jeden. Zakłada on, że obydwa otrzymane w argumencie zbiory są już posortowane. W każdym kroku rozpatrywane są dwa elementy, jeden z jednego zbioru, drugi z drugiego. Przechodząc po kolejnych elementach tych zbiorów, wybiera mniejszy z nich (w przypadku kolejności rosnącej) i przenosi go do zbioru wynikowego.

W przypadku, w którym któryś ze zbiorów zostaje pusty, pozostałe elementy z drugiego zbioru przenoszone zostają na koniec zbioru wynikowego.

### **Co to jest sortowanie kubełkowe (bucket\_sort)?**

Procedura bucket\_sort dzieli nasze dane na k podprzedziałów na podstawie określonych kryteriów, po czym sortuje te podprzedziały jakimś innym algorytmem sortującym, zazwyczaj insertion sort.

### **Uzasadnij, jaka jest jego złożoność średnia a jaka pesymistyczna?**

Złożoność średnia to  $O(n+k)$ , gdzie n to ilość danych, a k to ilość podzbiorów.

Złożoność pesymistyczna zachodzi w najgorszym przypadku, w którym powstaje tylko jeden kubełek, wtedy złożoność ta jest całkowicie uzależniona od złożoności algorytmu sortującego wykorzystywanego wewnątrz kubełka. W przypadku używania insertion\_sort złożoność ta wynosi  $O(n^2)$