# PENYORTIRAN /PENGURUTAN

Dalam materi ini Anda akan berhadapan dengan berbagai teknik pengurutan dan algoritmanya yang digunakan untuk memanipulasi struktur data dan penyimpanannya. Metode pengurutan dapat diterapkan dengan cara yang berbeda - dengan pemilihan, metode penyisipan, atau dengan menggabungkan. Berbagai jenis dan bentuk metode pengurutan telah dieksplorasi dalam Kuliah ADP ini.

## Apa itu pengurutan?

Pengurutan mengacu pada operasi atau teknik menyusun dan menyusun ulang kumpulan data dalam beberapa urutan tertentu. Kumpulan rekaman yang disebut daftar di mana setiap catatan memiliki satu atau beberapa bidang. Bidang yang berisi nilai unik untuk setiap rekaman diilah sebagai *bidang* kunci. Misalnya, direktori nomor telepon dapat dianggap sebagai daftar di mana setiap rekaman memiliki tiga bidang - 'nama' orang tersebut, 'alamat' orang tersebut, dan 'nomor telepon' mereka. Menjadi nomor telepon yang unik dapat berfungsi sebagai kunci untuk menemukan catatan apa pun dalam daftar.

Pengurutan adalah operasi yang dilakukan untuk menyusun catatan tabel atau daftar dalam beberapa urutan sesuai dengan beberapa kriteria pemesanan tertentu. Pengurutan dilakukan sesuai dengan beberapa nilai kunci dari setiap rekaman.

Catatan diurutkan secara numerik atau alfanumerik. Rekaman kemudian disusun dalam urutan naik atau turun tergantung pada nilai numerik kunci. Berikut adalah contoh, di mana pengurutan daftar tanda yang diperoleh oleh siswa dalam subjek tertentu dari kelas.

## Kategori Pengurutan

Teknik pengurutan dapat dibagi menjadi dua kategori. Ini adalah:

- Pengurutan Internal
- Pengurutan Eksternal

**Pengurutan Internal**: Jika semua data yang akan diurutkan dapat disesuaikan pada satu waktu di memori utama, metode pengurutan internal sedang dilakukan.

**Pengurutan**Eksternal : Ketika data yang akan diurutkan tidak dapat diakomodasi dalam memori pada saat yang sama dan beberapa harus disimpan dalam memori tambahan seperti hard disk, disket, pita magnetik dll, maka metode pengurutan eksternal dilakukan.

## Kompleksitas Pengurutan Algoritma

Kompleksitas pengurutan algoritma menghitung waktu berjalan dari fungsi di mana 'n' jumlah item harus diurutkan. Pilihan metode pengurutan mana yang cocok untuk masalah tergantung pada beberapa konfigurasi dependensi untuk masalah yang berbeda. Yang paling penting dari pertimbangan ini adalah:

- Lamanya waktu yang dihabiskan oleh programmer dalam memprogram program pengurutan tertentu
- Jumlah waktu alat berat yang diperlukan untuk menjalankan program
- Jumlah memori yang diperlukan untuk menjalankan program

## Efisiensi Teknik Pengurutan

Untuk mendapatkan jumlah waktu yang diperlukan untuk mengurutkan array elemen 'n' dengan metode tertentu, pendekatan normal adalah menganalisis metode untuk menemukan jumlah perbandingan (atau pertukaran) yang diperlukan olehnya. Sebagian besar teknik pengurutan sensitif data, sehingga metrik untuk mereka tergantung pada urutan di mana mereka muncul dalam array input.

Berbagai teknik pengurutan dianalisis dalam berbagai kasus dan dinamai kasus-kasus ini sebagai berikut:

- Kasus terbaik
- Kasus terburuk
- Kasus rata-rata

Oleh karena itu, hasil dari kasus-kasus ini seringkali merupakan rumus yang memberikan waktu rata-rata yang diperlukan untuk jenis ukuran tertentu 'n.' Sebagian besar metode sortir memiliki persyaratan waktu yang berkisar dari O(nlog n) hingga $O(n^2)$.

# Jenis Teknik Pengurutan

- Bubble Sort
- Selection Sort
- Merge Sort
- Insertion Sort
- Quick Sort
- Heap Sort

# Kompleksitas Algoritma

Istilah kompleksitas algoritma mengukur berapa banyak langkah yang diperlukan oleh algoritma untuk menyelesaikan masalah yang diberikan. Ini mengevaluasi urutan jumlah operasi yang dijalankan oleh algoritma sebagai fungsi ukuran data input.

Untuk menilai kompleksitas, urutan (perkiraan) penghitungan operasi selalu dipertimbangkan alih-alih menghitung langkah-langkah yang tepat.

O(f) adalah notasi mewakili kompleksitas algoritma, yang juga disebut sebagai notasi Asimptomatik atau**"notasi Big O".** Di sini f sesuai dengan fungsi yang ukurannya sama dengan data input. Kompleksitas komputasi asimptomatik O(f) menentukan urutan sumber daya seperti waktu CPU, memori, dll.

Kompleksitas dapat ditemukan dalam bentuk apa pun seperti konstanta, logaritmik, linear, n*log(n), kuadrat, kubik, eksponensial, dll. Tidak lain adalah urutan konstan, logaritmik, linier dan sebagainya, jumlah langkah yang ditemui untuk penyelesaian algoritma tertentu. Untuk membuatnya lebih tepat, kita sering menyebut kompleksitas algoritma sebagai "waktu berjalan".

## Kompleksitas Standar Algoritma

- **Kompleksitas Konstan:**

  Ini memaksakan kompleksitas **O(1)**. Ini menjalani eksekusi sejumlah langkah konstan seperti 1, 5, 10, dll untuk memecahkan masalah tertentu. Jumlah operasi tidak independen dari ukuran data input.

- **Kompleksitas Logaritma:**

  Ini memberlakukan kompleksitas **O(log(N))**. Ini menjalani eksekusi urutan langkah log(N). Untuk melakukan operasi pada elemen N, seringkali dibutuhkan basis logaritmik sebagai 2. Untuk N = 1.000.000, algoritma yang memiliki kompleksitas O(log(N)) akan mengalami 20 langkah (dengan presisi konstan). Di sini, basis logaritmik tidak memiliki konsekuensi yang diperlukan untuk urutan jumlah operasi, sehingga biasanya dihilangkan.

- **Kompleksitas Linear:**
  - Ini memaksakan kompleksitas **O(N)**. Ini mencakup jumlah langkah yang sama dengan jumlah total elemen untuk mengimplementasikan operasi pada elemen N. Misalnya, jika

ada 500 elemen, maka akan memakan waktu sekitar 500 langkah. Pada dasarnya, dalam kompleksitas linear, jumlah elemen secara linear tergantung pada jumlah langkah. Misalnya, jumlah langkah untuk elemen N bisa N/2 atau 3*N.

- Ini juga memberlakukan waktu jalan **O(n*log(n))**. Ini menjalani eksekusi order N*log(N) pada N jumlah elemen untuk memecahkan masalah yang diberikan. Untuk 1000 elemen yang diberikan, kompleksitas linier akan menjalankan 10.000 langkah untuk memecahkan masalah tertentu.

- **Kompleksitas Kuadrat:** Ini memaksakan kompleksitas **O($n^{2}$)**. Untuk ukuran data input N, ia mengalami urutan $N^2$ hitungan operasi pada N jumlah elemen untuk memecahkan masalah tertentu.
  Jika N = 100, itu akan bertahan 10.000 langkah. Dengan kata lain, setiap kali urutan operasi cenderung memiliki hubungan kuadrat dengan ukuran data input, itu menghasilkan kompleksitas kuadrat. Misalnya, untuk N jumlah elemen, langkah-langkah ditemukan dalam urutan 3 * $N^2$/ 2.

- **Kompleksitas Kubik:** Ini memaksakan kompleksitas **O($n^{3}$)**. Untuk ukuran data input N, ia menjalankan urutan $N^3$ langkah pada elemen N untuk memecahkan masalah tertentu.
  Misalnya, jika ada 100 elemen, itu akan mengeksekusi 1.000.000 langkah.

- **Kompleksitas Eksponensial:** Ini memaksakan kompleksitas **O($2^{n}$),O(N!), O($n^{k}$), ...**. Untuk elemen N, ia akan mengeksekusi urutan jumlah operasi yang secara eksponensial dapat diandalkan pada ukuran data input.
  Misalnya, jika N = 10, maka fungsi eksponensial $2^N$ akan menghasilkan 1024. Demikian pula, jika N = 20, itu akan menghasilkan 1048 576, dan jika N = 100, itu akan menghasilkan angka yang memiliki 30 digit. Fungsi eksponensial N! tumbuh lebih cepat; misalnya, jika N = 5 akan menghasilkan 120. Demikian juga, jika N = 10, itu akan menghasilkan 3.628.800 dan sebagainya.

Karena konstanta tidak memiliki efek yang signifikan pada urutan jumlah operasi, jadi lebih baik mengabaikannya. Dengan demikian, untuk mempertimbangkan algoritma menjadi linier dan sama efisiennya, ia harus menjalani N, N / 2 atau 3 * N hitungan operasi, masing-masing, pada jumlah elemen yang sama untuk memecahkan masalah tertentu.

# Komputasi Kompleksitas Waktu Algoritma Pengurutan Gelembung

## 1. Pengantar

Dalam Kuliah **ini, kita akan membahas algoritma pengurutan gelembung.** Kita akan menyajikan pseudocode algoritma dan menganalisis kompleksitas waktunya.

## 2. Algoritma

Pengurutan gelembung , juga dikenal sebagai pengurutan, adalah algoritma yang sangat sederhana untuk mengurutkan elemen dalam array. Pengurutan gelembung bekerja dengan terus bertukar elemen yang berdekatan jika muncul dalam urutan yang salah dalam daftar input asli. Proses pertukaran *swapp* ini berlanjut sampai kita mengurutkan daftar input.

Di bagian ini, kita akan membahas langkah-langkah pengurutan gelembung secara rinci.
Pertama-tama mari kita lihat pseudocode dari algoritma pengurutan gelembung:

**Algorithm 1:** Bubble sort

```
Data: Input array A[]   data input tidak terurut
Result: Sorted A[]   data output ter-urut naik /ascending
int i, j, k;   inisialisasi
N = length(A);   sisa elemen yg belum diurutkan
for j = 1 to N do   loop luar
    for i = 0 to N-1 do   loop dalam
        if A[i] > A[i+1] then   jika elemen kiri > elemen kanan
                                untuk hasil pengurutan acesnding
            temp = A[i];
            A[i] = A[i+1];      Swapp
            A[i+1] = temp;
        end
    end
end
```

Mari kita bahas langkah-langkah dan notasi yang digunakan dalam algoritma ini. Perhatikan bahwa tujuannya adalah $A[]$ untuk mengambil array input dan mengurutkan elemennya dalam urutan naik.

Kita mulai dengan elemen 0 pertama (indeks dalam array). $A[]$ Kemudian kita memeriksa apakah elemen $1^{A[]}$ berikutnya (indeks dalam) dalam array lebih besar dari elemen saat ini atau tidak. Jika elemen saat 0 ini $A[]$ (indeks dalam array) lebih $1^{A[]}$ besar maka elemen berikutnya (indeks masuk ), Kita akan menukarnya. Jika elemen saat ini lebih kecil maka elemen berikutnya, kita akan pindah ke elemen berikutnya dalam array. Dengan cara ini, kita akan memproses dan menyelesaikan swap untuk seluruh array. Ini adalah iterasi pertama.

Jumlah iterasi yang diperlukan sama dengan jumlah elemen dalam array. Setelah menyelesaikan iterasi yang diperlukan, kita akan mendapatkan array diurutkan dalam urutan naik. Kita harus dipahami, bahwa gelembung dapat mengurutkan hal-hal baik dalam urutan naik dan turun .

Dalam pengurutan gelembung di atas, ada beberapa masalah. Dalam versi ini, Kita membandingkan semua pasang elemen untuk kemungkinan bertukar. Kita melanjutkan ini sampai Kita menyelesaikan iterasi yang diperlukan.

Sekarang mari kita asumsikan bahwa array input yang diberikan hampir atau sudah diurutkan. Sayangnya dengan pseudocode Kita saat ini, tidak ada indikator untuk menunjukkan bahwa array diurutkan, jadi Kita masih akan melalui semua iterasi. **Bisakah kita memperbaiki ini?** Ayo kita lihat.

Setelah setiap iterasi, mari kita lacak elemen yang kita tukar. Jika tidak ada swap, kita dapat mengasumsikan bahwa kita mengurutkan array input.

Dengan asumsi ini, Kita siap untuk menyajikan algoritma pengurutan gelembung yang ditingkatkan:

**Algorithm 2:** Improved Bubble Sort

```
Data: Input array A[]
Result: Sorted A[]
int i, j, k;
indicator = 1;
N = length(A);
for j = 1; j ≤ (N-1) ∧ indicator == 1; j++ do
    indicator = 0;
    for i = 1 to N-1; i++ do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
            indicator = 1;
        end
    end
end
```

Di sini, dalam algoritma ini, $indicator$Kita telah memperkenalkan variabel baru untuk melacak elemen yang ditukar. Juga, ini dapat menunjukkan apakah array yang diberikan sudah diurutkan atau tidak selama iterasi. Jadi pada titik tertentu selama iterasi, jika tidak ada swap yang terjadi untuk seluruh array, itu akan keluar dari lingkaran dan tidak akan ada lagi iterasi yang diperlukan.

# 3. Analisis Kompleksitas Waktu

## 3.1. Kompleksitas Waktu Pengurutan Gelembung Standar

Dalam kasus versi standar dari jenis gelembung, kita perlu melakukan $N$ iterasi. Dalam setiap iterasi, Kita melakukan perbandingan dan Kita melakukan pertukaran jika diperlukan. Mengingat array ukuran, $N$iterasi pertama $(N-1)$melakukan perbandingan. Iterasi kedua melakukan $(N-2)$perbandingan. Dengan cara ini, jumlah total perbandingan adalah:

$$(N-1)+(N-2)+(N-3)+.....+3+2+1=\frac{N(N-1)}{2}=\mathcal{O}(N^2)$$

Oleh karena **itu, dalam kasus rata-rata, kompleksitas waktu dari jenis gelembung standar akan** $\mathcal{O}(\mathbf{N^2})$.

Sekarang mari kita bicara tentang kasus terbaik dan kasus terburuk dalam jenis gelembung. Kasus terbaik adalah ketika array input sudah diurutkan. Dalam hal ini, Kita $N$memeriksa semua elemen untuk melihat apakah ada kebutuhan untuk swap. Kalau tidak ada tukar-tukar masih $N$kita lanjutkan dan selesaikan iterasi. Oleh karena **itu, dalam skenarioterbaik, kompleksitas waktu dari jenis gelembung standar akan** $\mathcal{O}(\mathbf{N^2})$.

Dalam kasus terburuk,array diurutkan secara terbalik. Jadi kita perlu $(N-1)$melakukan perbandingan dalam iterasi pertama, $(N-2)$dalam interaksi kedua, dan sebagainya. Oleh karena **itu, kompleksitas waktu dari jenis gelembung dalam kasus terburuk akan sama dengan kasus rata-rata dan kasus terbaik:** $\mathcal{O}(\mathbf{N^2})$.

## 3.2. Kompleksitas Waktu Dari Peningkatan Gelembung

Dalam kasus peningkatan jenis gelembung, kita perlu melakukan lebih sedikit swap dibandingkan dengan versi standar. Jika kita berbicara tentang kompleksitas waktu, **dalam rata-rata dan**

**kompleksitas waktu terburuk akan sama dengan yang standar:** $\mathcal{O}(\mathbf{N^2})$. Meskipun ada peningkatan efisiensi dan kinerja versi yang ditingkatkan dalam rata-rata dan kasus terburuk.
Dalam kasus terbaik, ketika array yang diberikan sudah diurutkan, pengurutan gelembung yang ditingkatkan mencapai kompleksitas waktu yang lebih baik dibandingkan dengan versi standar.
Dalam hal ini, diberikan array, Kita melintasi daftar mencari kemungkinan swap. Tetapi karena array sudah diurutkan, tidak akan ada swap. Di sini kita tidak akan melanjutkan iterasi lagi. Sebaliknya, kita akan keluar dari lingkaran dan algoritma berakhir. Dengan cara ini, kita tidak perlu menyelesaikan semua iterasi.

Jika array yang diberikan diurutkan, Kita melintasi array sekali. Jadi **kompleksitas waktu dalam kasus terbaik adalah** $\mathcal{O}(\mathbf{N})$.

# 4. Kelebihan dan Kekurangan

Bubble sort adalah algoritma penyortiran yang sangat sederhana untuk dipahami dan diimplementasikan. Karena kesederhanaannya, pengurutan gelembung digunakan untuk memperkenalkan algoritma penyortiran dalam ilmu komputer. Juga, ini memberikan basis yang baik untuk algoritma penyortiran populer lainnya. Ketika array input hampir diurutkan dan kita perlu menukar hanya beberapa elemen, maka pengurutan gelembung adalah pilihan yang baik. Ini juga merupakan algoritma penyortiran yang stabil dan algoritma pengisian poligon menggunakan konsep pengurutan gelembung.

Kerugian utama dari jenis gelembung adalah kompleksitas waktu. Ketika array input berisi sejumlah besar elemen, efisiensi pengurutan gelembung menurun drastis dan waktu rata-rata meningkat secara kuadrat. Kinerja gelembung semacam di perangkat keras CPU modern sangat buruk. Meskipun waktu berjalan dari jenis gelembung secara asimptotis setara dengan algoritma penyortiran populer lainnya seperti penyisipan, pengurutan gelembung melakukan sejumlah swap yang sangat tinggi di antara elemen.

# 5. Kesimpulan

Dalam Kuliah ini, kita telah membahas jenis gelembung. Kita telah menyajikan versi standar dan versi yang ditingkatkan dari algoritma pengurutan gelembung.
Selain itu, Kita telah menunjukkan analisis terperinci tentang kompleksitas waktu untuk kedua versi.

# Teknis Pengurutan Gelembung

## 1. Pendahuluan

Dalam artikel cepat ini, kita akan mengeksplorasi algoritma Bubble Sort secara rinci, berfokus pada implementasi Java.

Ini adalah salah satu algoritma penyortiran yang paling mudah; **ide intinya adalah untuk terus menukar elemen array yang** berdekatan jika mereka berada dalam urutan yang salah sampai koleksi diurutkan.

Item kecil "gelembung" ke bagian atas daftar saat Kita melakukan iterasi struktur data. Oleh karena itu, teknik ini dikenal sebagai jenis gelembung.

Karena penyortiran dilakukan dengan bertukar, kita dapat mengatakan itu melakukan penyortiran di tempat.

Selain **itu, jika dua elemen memiliki nilai yang sama, data yang dihasilkan akan mempertahankan urutannya -** yang membuatnya menjadi pengurutan yang stabil.

## 2. Metodologi

Seperti disebutkan sebelumnya, untuk mengurutkan array, Kita iterasi melaluinya sambil membandingkan elemen yang berdekatan, dan menukarnya jika perlu. Untuk array ukuran *n*,Kita melakukan *n-1* iterasi seperti itu.

Mari kita ambil contoh untuk memahami metodologi. Kita ingin mengurutkan array dalam urutan naik:

4 2 1 6 3 5

Kita memulai iterasi pertama dengan membandingkan 4 dan 2; Mereka jelas tidak dalam urutan yang tepat. Pertukaran akan menghasilkan:

**[2 4]** 1 6 3 5

Sekarang, mengulangi hal yang sama untuk 4 dan 1:

2 **[14]** 6 3 5

Kita terus melakukannya sampai akhir:

2 1 [**4 6]** 3 5

2 1 4 **[3 6]** 5

2 1 4 3 **[5 6]**

Seperti yang bisa kita lihat, pada akhir iterasi pertama, kita mendapat elemen terakhir di tempat yang sah. Sekarang, yang perlu kita lakukan adalah mengulangi prosedur yang sama dalam iterasi lebih lanjut. Kecuali, Kita mengecualikan elemen yang sudah diurutkan.

Dalam iterasi kedua, kita akan iterasi melalui seluruh array kecuali untuk elemen terakhir. Demikian pula, untuk iterasi ke-3, Kita menghilangkan 2 elemen terakhir. Secara umum, untuk iterasi k-th, Kita iterasi sampai indeks *n-k* (dikecualikan). Pada akhir *iterasi n-1*, kita akan mendapatkan array yang diurutkan.

Sekarang bahwa dalam diri Anda memahami teknik, mari kita menyelami implementasi.

# 3. Implementasi

Mari kita implementasikan pengurutan untuk contoh array yang kita bahas menggunakan pendekatan Java 8:

```
void bubbleSort(Integer[] arr) {
    int n = arr.length;
    IntStream.range(0, n - 1)
    .flatMap(i -> IntStream.range(1, n - i))
    .forEach(j -> {
        if (arr[j - 1] > arr[j]) {
            int temp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = temp;
            }
     });
}
```

Dan tes JUnit cepat untuk algoritma:

```
@Test
public void whenSortedWithBubbleSort_thenGetSortedArray() {
    Integer[] array = { 2, 1, 4, 6, 3, 5 };
    Integer[] sortedArray = { 1, 2, 3, 4, 5, 6 };
    BubbleSort bubbleSort = new BubbleSort();
    bubbleSort.bubbleSort(array);

    assertArrayEquals(array, sortedArray);
}
```

# 4. Kompleksitas dan Pengoptimalan

Seperti yang dapat kita **lihat, untuk kasus rata-rata dan terburuk, kompleksitas waktu adalah O(n^2)**.

Selain itu, **kompleksitas ruang**, bahkan dalam skenario terburuk, **adalah O(1) karena algoritma pengurutan gelembung tidak memerlukan memori tambahan** dan penyortiran terjadi dalam array asli.

Dengan menganalisis solusi dengan hati-hati, kita dapat **melihat bahwa jika tidak ada swap yang ditemukan dalam iterasi, kita tidak perlu iterasi lebih lanjut**.

Dalam kasus contoh yang dibahas sebelumnya, setelah iterasi ke-2, kita mendapatkan:

1 2 3 4 5 6

Dalam iterasi ketiga, kita tidak perlu menukar sepasang elemen yang berdekatan. Jadi kita bisa melewatkan semua iterasi yang tersisa.

Dalam kasus array yang diurutkan, bertukar tidak akan diperlukan dalam iterasi pertama itu sendiri - yang berarti kita dapat menghentikan eksekusi. Ini adalah skenario kasus terbaik dan **kompleksitas waktu algoritma adalah O(n)**.

Sekarang, mari kita terapkan solusi yang dioptimalkan.

```
public void optimizedBubbleSort(Integer[] arr) {
    int i = 0, n = arr.length;
    boolean swapNeeded = true;
    while (i < n - 1 && swapNeeded) {
        swapNeeded = false;
        for (int j = 1; j < n - i; j++) {
            if (arr[j - 1] > arr[j]) {
                int temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
                swapNeeded = true;
            }
        }
        if(!swapNeeded) {
            break;
        }
        i++;
    }
}
```

Mari kita periksa output untuk algoritma yang dioptimalkan:

```
@Test
public void
  givenIntegerArray_whenSortedWithOptimizedBubbleSort_thenGetSortedArray() {
      Integer[] array = { 2, 1, 4, 6, 3, 5 };
      Integer[] sortedArray = { 1, 2, 3, 4, 5, 6 };
      BubbleSort bubbleSort = new BubbleSort();
      bubbleSort.optimizedBubbleSort(array);

      assertArrayEquals(array, sortedArray);
}
```

# 5. Kesimpulan

Dalam Kuliah ini, Kita melihat cara kerja Bubble Sort, dan implementasinya di Java. Kita juga melihat bagaimana hal itu dapat dioptimalkan. Untuk meringkas, ini adalah algoritma stabil di tempat, dengan kompleksitas waktu:

- Kasus Terburuk dan Rata-Rata: O(n*n), saat array berada dalam urutan terbalik
- Kasus terbaik: O(n), ketika array sudah diurutkan

Algoritma ini populer dalam grafik komputer, karena kemampuannya untuk mendeteksi beberapa kesalahan kecil dalam pengurutan. Misalnya, dalam array yang hampir diurutkan, hanya dua elemen yang perlu ditukar, untuk mendapatkan array yang sepenuhnya diurutkan. Pengurutan Gelembung dapat memperbaiki kesalahan tersebut (yaitu mengurutkan array ini) dalam waktu linear.

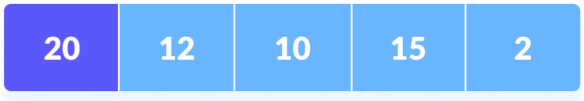
# Algoritma Pengurutan Pilihan

Dalam Kuliah ini, Anda akan mempelajari cara kerja pengurutan pilihan. Selain itu, Anda akan menemukan contoh yang berfungsi dari jenis pilihan di C, C++, Java dan Python.

Pengurutan pilihan adalah algoritma yang memilih elemen terkecil dari daftar yang tidak diurutkan di setiap iterasi dan menempatkan elemen tersebut di awal daftar yang tidak diurutkan.

## Bagaimana Cara Kerja Pengurutan Pilihan?

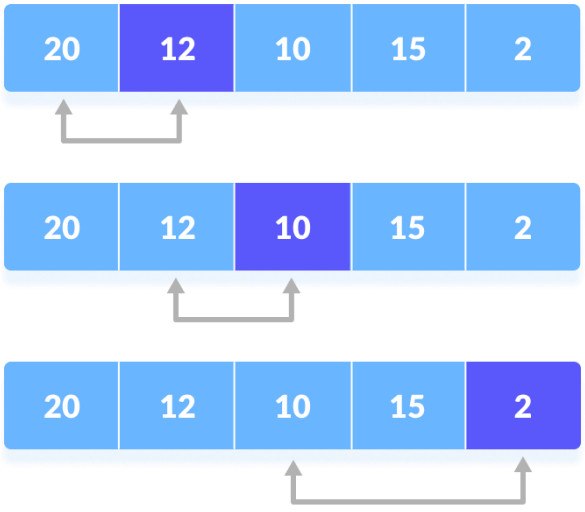
1. Atur elemen pertama sebagai **`minimum`.**




Pilih elemen pertama minimal

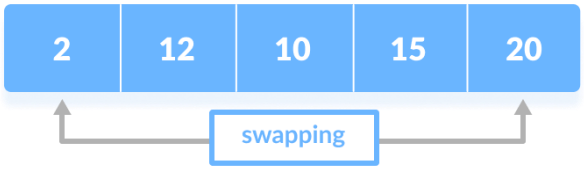
2. Bandingkan `minimum` dengan elemen kedua. Jika elemen kedua lebih kecil dari `minimum`, tetapkan elemen kedua sebagai **`minimum`.**

Bandingkan **`minimum`** dengan elemen ketiga. Sekali lagi, jika elemen ketiga lebih kecil, maka tetapkan **`minimum`** ke elemen ketiga jika tidak melakukan apa-apa. Proses berlangsung sampai elemen terakhir.



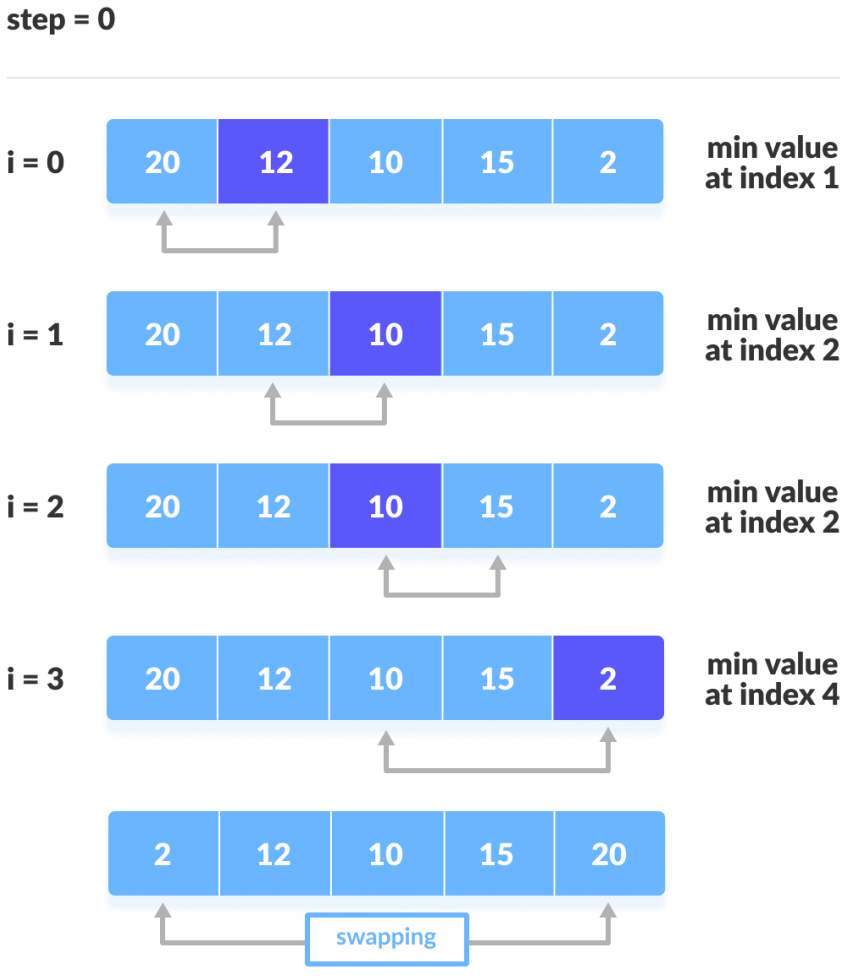

Bandingkan minimum dengan elemen yang tersisa

3. Setelah setiap iterasi, **`minimum`** ditempatkan di depan daftar yang tidak diurutkan.

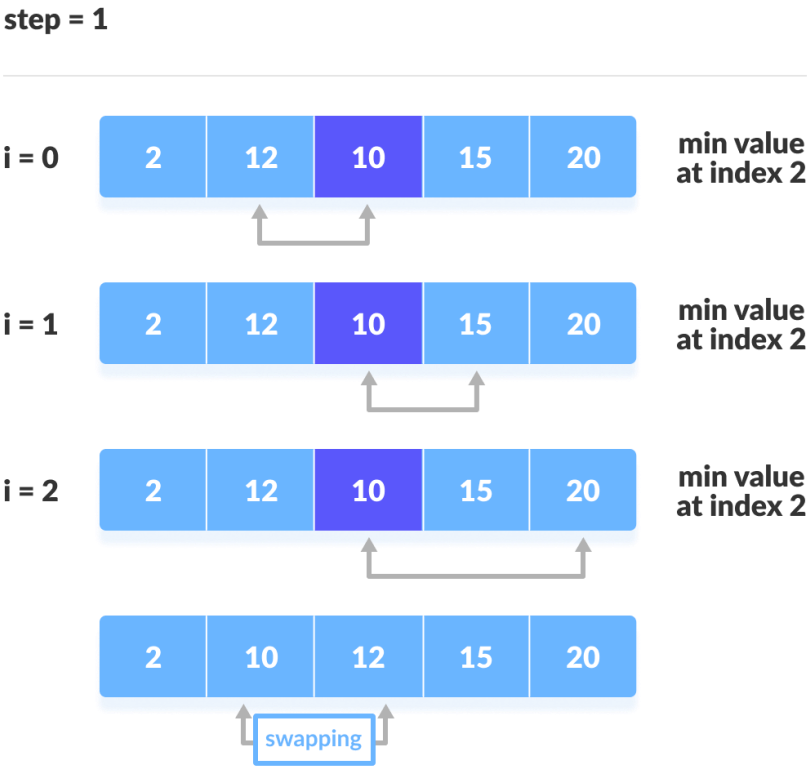



Tukar yang pertama dengan minimum

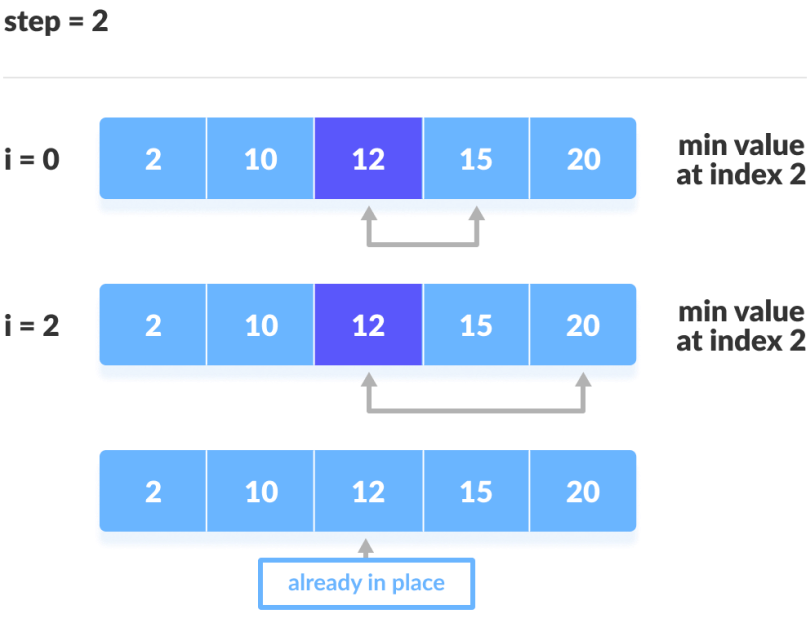
4. Untuk setiap iterasi, pengindeksan dimulai dari elemen pertama yang tidak diurutkan. Langkah 1 hingga 3 diulang sampai semua elemen ditempatkan pada posisi yang benar.



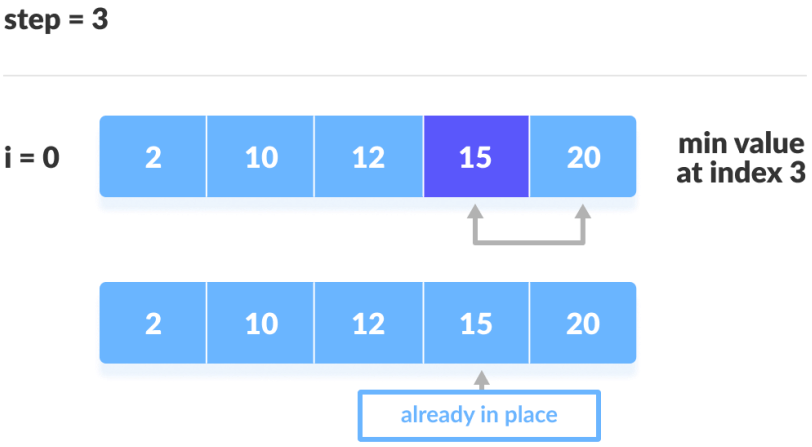

Iterasi pertama




Iterasi kedua




Iterasi ketiga




Iterasi keempat

# Algoritma Pengurutan Pilihan

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

# Kompleksitas

| Siklus | Jumlah Perbandingan |
|---|---|
| **Ke-1** | (n-1) |
| **Ke-2** | (n-2) |
| **Ke-3** | (n-3) |
| **...** | ... |
| **Terakhir** | 1 |

Jumlah perbandingan:

$$(n - 1) + (n - 2) + (n - 3) + ..... + 1 = n(n - 1) / 2$$

hampir sama dengan $\mathbf{n^2}$.

**Kompleksitas = O ($n^2$)**

Juga, kita dapat menganalisis kompleksitas hanya dengan mengamati jumlah loop.

Ada 2 loop sehingga kompleksitasnya **`n*n = n`$^2$.**

**Kompleksitas Waktu:**

- **Kompleksitas Kasus Terburuk: O($n^2$)**

  Jika kita ingin mengurutkan dalam urutan naik dan array dalam urutan menurun maka, kasus terburuk terjadi.

- **Kompleksitas Kasus Terbaik**: **O($n^2$)**

  Ini terjadi ketika array sudah diurutkan

- **Kompleksitas Kasus Rata-rata**: **O($n^2$)**

  Ini terjadi ketika elemen array berada dalam urutan jumbled (baik naik maupun turun).

Kompleksitas waktu dari jenis pemilihan sama dalam semua kasus. Di setiap langkah, Anda harus menemukan elemen minimum dan meletakkannya di tempat yang tepat. Elemen minimum tidak diketahui sampai akhir array tidak tercapai.

**Kompleksitas Ruang:**

Kompleksitas ruang **`O(1)`** karena variabel tambahan **`temp`** digunakan.

# Pertimbangan Aplikasi Pengurutan Pilihan

*Pengurutan pilihan digunakan ketika:*

- Jumlah data yang akan diurutkan sedikit
- Biaya /usaha/overhead pertukaran data /swapp tidak masalah
- memeriksa semua elemen adalah wajib
- biaya penulisan data ke memori yang penting seperti dalam memori flash (jumlah tulisan/swap `O(n)` dibandingkan dengan **`O(n²)`** semacam gelembung)

# Algoritma Pengusisipan

Dalam Kuliah ini, Anda akan mempelajari cara kerja pengurutan. Selain itu, Anda akan menemukan contoh yang berfungsi dari pengurutan pengurutan Java.

Penyisipan berfungsi sama seperti kita mengurutkan kartu di tangan kita dalam permainan kartu. Kita berasumsi bahwa kartu pertama sudah diurutkan kemudian, Kita memilih kartu yang tidak diurutkan. Jika kartu yang tidak diurutkan lebih besar dari kartu di tangan, kartu ditempatkan di sebelah kanan sebaliknya, di sebelah kiri. Dengan cara yang sama, kartu lain yang tidak diurutkan diambil dan diletakkan di tempat yang tepat.

Pendekatan serupa digunakan oleh pengukupan.
Penyisipan adalah algoritma pengurutan yang menempatkan elemen yang tidak diurutkan di tempat yang cocok di setiap iterasi.

## Bagaimana Cara Kerja Pengurutan Penyisipan?

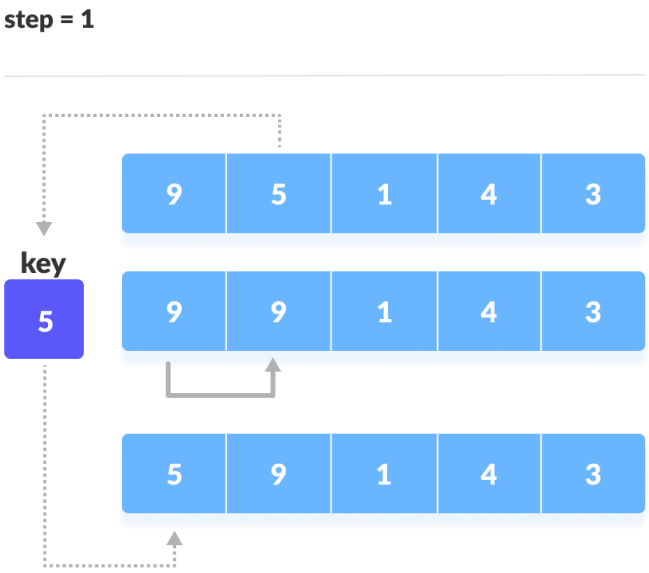
Misalkan kita perlu mengurutkan array berikut.


Array awal

1. Elemen pertama dalam array diasumsikan untuk diurutkan. Ambil elemen kedua dan simpan secara terpisah di **`key`**.

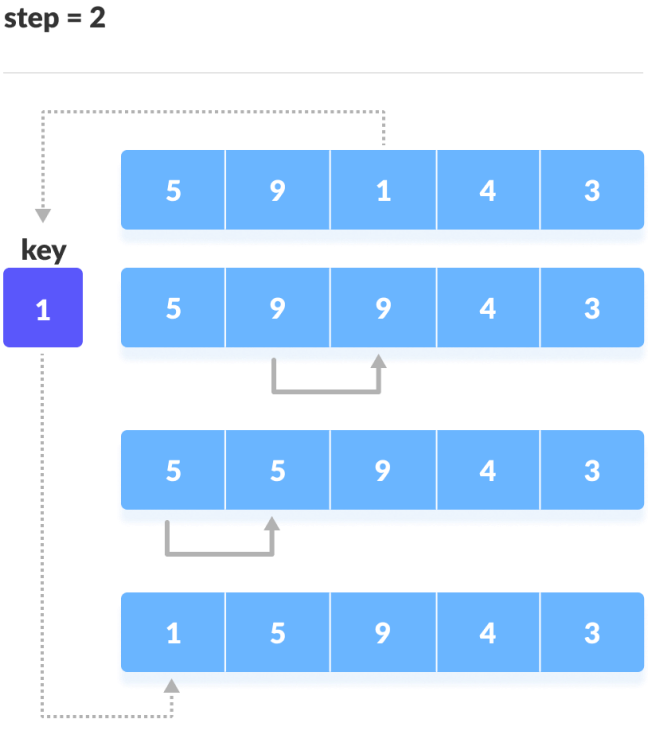
Bandingkan **`key`** dengan elemen pertama. Jika elemen pertama lebih besar dari **`key`**

Kemudian *key* ditempatkan di depan elemen pertama.



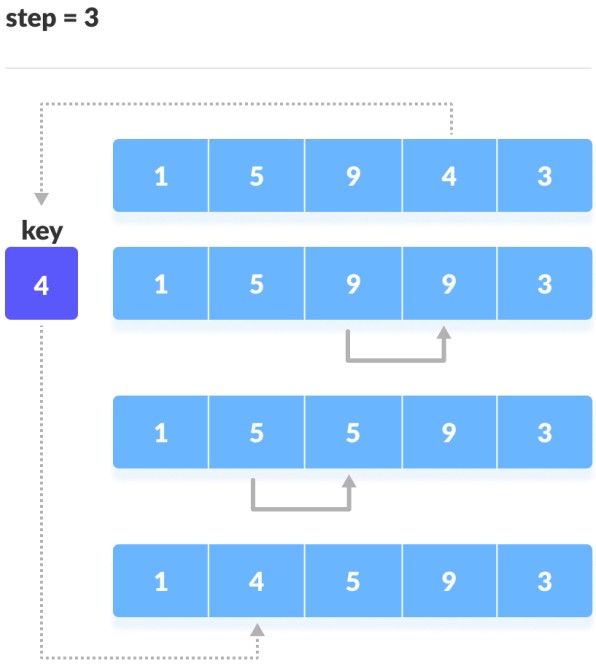

Jika elemen pertama lebih besar dari kunci, maka kunci ditempatkan di depan elemen pertama.

2. Sekarang, dua elemen pertama diurutkan.Ambil elemen ketiga dan bandingkan dengan elemen di sebelah kirinya. Menempatkannya tepat di belakang elemen yang lebih kecil dari itu. Jika tidak ada elemen yang lebih kecil dari itu, maka letakkan di awal array.

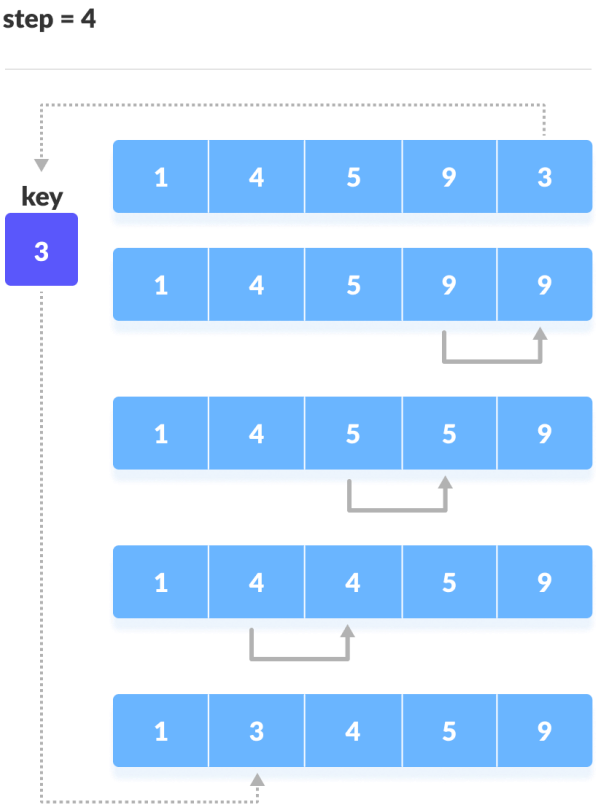



Tempat 1 di awal

3. Demikian pula, tempatkan setiap elemen yang tidak terseort pada posisi yang benar.




Tempat 4 di belakang 1




Tempatkan 3 di belakang 1 dan array diurutkan

# Algoritma Pengusisipan

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

# Kompleksitas

### Kompleksitas Waktu

- **Kompleksitas Kasus Terburuk:** $O(n^2)$ Misalkan, array berada dalam urutan naik, dan Anda ingin mengurutkannya dalam urutan menurun. Dalam hal ini, kompleksitas kasus terburuk terjadi.Setiap elemen harus dibandingkan dengan masing-masing elemen lain sehingga, untuk setiap elemen ke-n, $(n-1)$ jumlah perbandingan dibuat.Dengan demikian, jumlah total perbandingan $= n*(n-1) \sim n^2$

- **Kompleksitas Kasus Terbaik:** $O(n)$ Ketika array sudah diurutkan, loop luar berjalan untuk $n$ berapa kali sementara loop dalam tidak berjalan sama sekali. Jadi, hanya ada $n$ jumlah perbandingan. Dengan demikian, kompleksitasnya linier.
-
- **Kompleksitas Kasus Rata-rata:** $O(n^2)$ Ini terjadi ketika elemen array berada dalam urutan jumbled (tidak naik atau turun).

### Kompleksitas Ruang

Kompleksitas ruang $O(1)$ karena variabel tambahan **key** digunakan.

# Aplikasi Pengurutan

Pengurutan digunakan ketika:

- array memiliki sejumlah kecil elemen
- hanya ada beberapa elemen yang tersisa untuk diurutkan