Pengurutan

Tim Olimpiade Komputer Indonesia

Pendahuluan

Melalui dokumen ini, kalian akan:

- Mempelajari konsep algoritma sederhana.
- Memahami berbagai algoritma pengurutan sederhana.
- Memahami keuntungan dan kerugian dari masing-masing algoritma.

Pendahuluan (lanj.)

- Pengurutan sering digunakan dalam pemrograman untuk membantu membuat data lebih mudah diolah.
- Terdapat berbagai macam cara untuk melakukan pengurutan, masing-masing dengan keuntungan dan kekurangannya.

Soal: Bebek Berbaris

Deskripsi:

- Sebelum masuk ke dalam kandang, para bebek akan berbaris terlebih dahulu.
- Seiring dengan berjalannya waktu, bebek-bebek tumbuh tinggi. Pertumbuhan ini berbeda-beda; ada bebek yang lebih tinggi dari bebek lainnya.
- Terdapat N ekor bebek, bebek ke-i memiliki tinggi sebesar h_i.
- Perbedaan tinggi ini menyebabkan barisan terlihat kurang rapi, sehingga Pak Dengklek ingin bebek-bebek berbaris dari yang paling pendek ke paling tinggi.
- Bantulah para bebek untuk mengurutkan barisan mereka!

Soal: Bebek Berbaris (lanj.)

Batasan:

- $1 \leq N \leq 1.000$
- $1 \leq h_i \leq 100.000$, untuk $1 \leq i \leq N$

Solusi

- Persoalan ini meminta kita melakukan pengurutan N bilangan dengan rentang datanya antara 1 sampai 100.000.
- Terdapat sejumlah algoritma pengurutan, yang akan dibahas pada bagian berikutnya.

Bagian 1

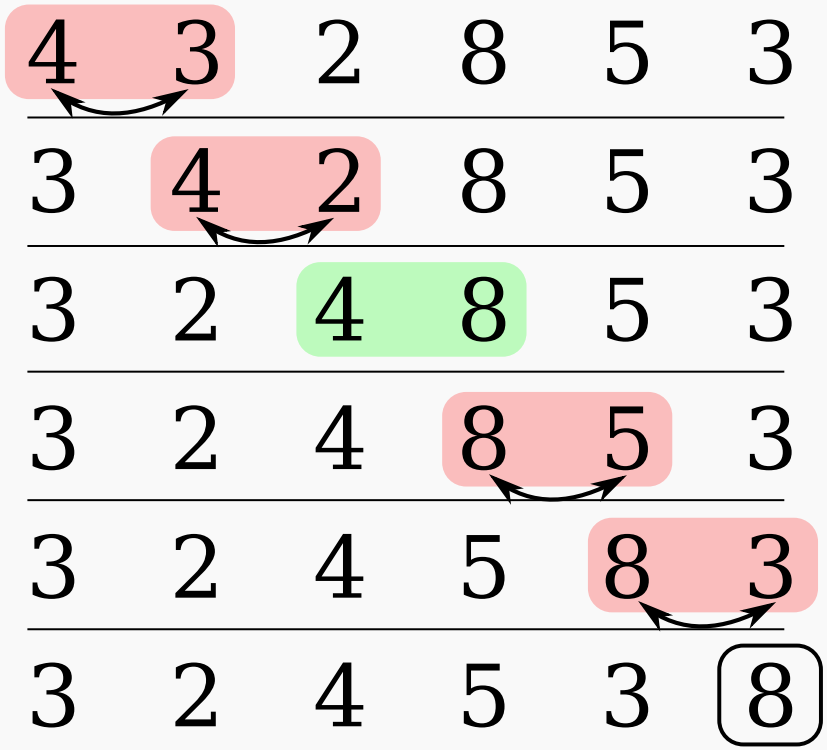
Bubble Sort

Ide Dasar

- Mulai dari elemen pertama, cek apakah elemen sesudahnya (yaitu elemen kedua) lebih kecil.
- Bila ya, artinya elemen pertama ini harus terletak sesudah elemen kedua. Untuk itu, lakukan penukaran.
- Bila tidak, tidak perlu lakukan penukaran.
- Lanjut periksa elemen kedua, ketiga, dan seterusnya.

Ide Dasar (lanj.)

- Proses ini mengakibatkan elemen dengan nilai terbesar pasti digiring ke posisi terakhir:

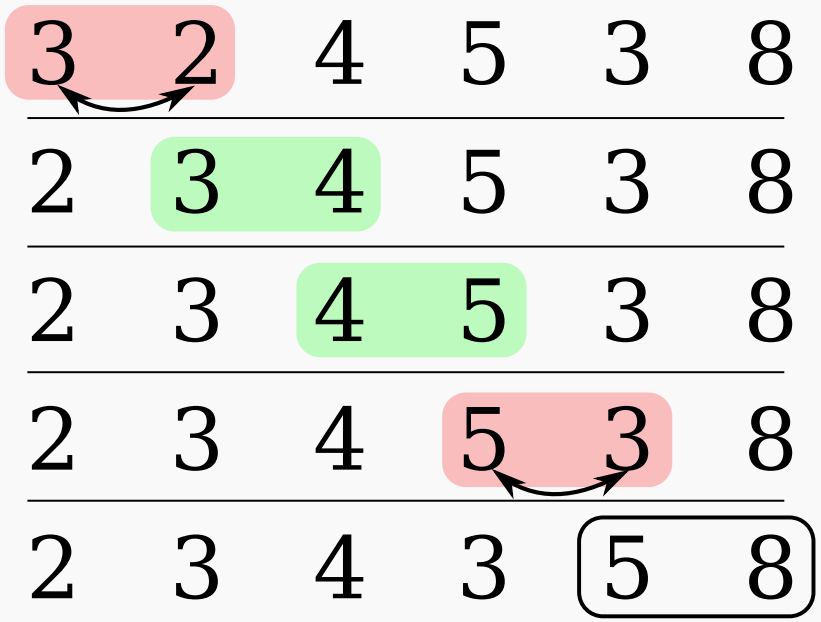


ditukar

tidak ditukar

Ide Dasar (lanj.)

- Bila proses ini dilakukan lagi, maka elemen kedua terbesar akan terletak di posisi kedua dari terakhir.
- Kali ini pemeriksaan cukup dilakukan sampai 1 elemen sebelum posisi terakhir, sebab elemen terakhir sudah pasti tidak akan berubah posisi:

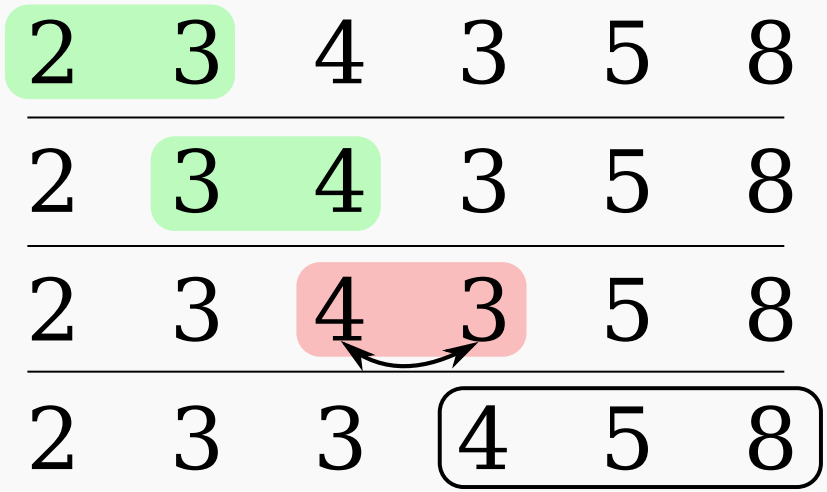


ditukar

tidak ditukar

Ide Dasar (lanj.)

- Demikian pula untuk eksekusi yang ketiga kalinya, yang kebetulan data sudah menjadi terurut:



ditukar

tidak ditukar

Ide Dasar (lanj.)

Pertanyaan

Jika eksekusi ke-i mengakibatkan i elemen terbesar terletak di i posisi terakhir, maka berapa kali eksekusi yang dibutuhkan sampai seluruh data dijamin terurut?

Analisis

- Dibutuhkan N kali eksekusi hingga seluruh data terurut.
- Dalam sekali eksekusi, dilakukan iterasi dari elemen pertama sampai elemen terakhir, yang kompleksitasnya berkisar antara $O(1)$ sampai $O(N)$, tergantung eksekusi ke berapa.
- Secara rata-rata, kompleksitasnya setiap eksekusi adalah $O(N/2)$, yang bisa ditulis $O(N)$.
- Total kompleksitas bubble sort adalah $O(N^2)$.

Contoh Kode

```
for i := 1 to N-1 do begin
  for j := 1 to N-i do begin
    if (h[j] > h[j+1]) then begin
      swap(h[j], h[j+1]); (* tukar h[j] dengan h[j+1] *)
    end;
  end;
end;
```

Catatan: fungsi **swap** tidak tersedia secara *default* pada Pascal, jadi harus Anda implementasikan sendiri.

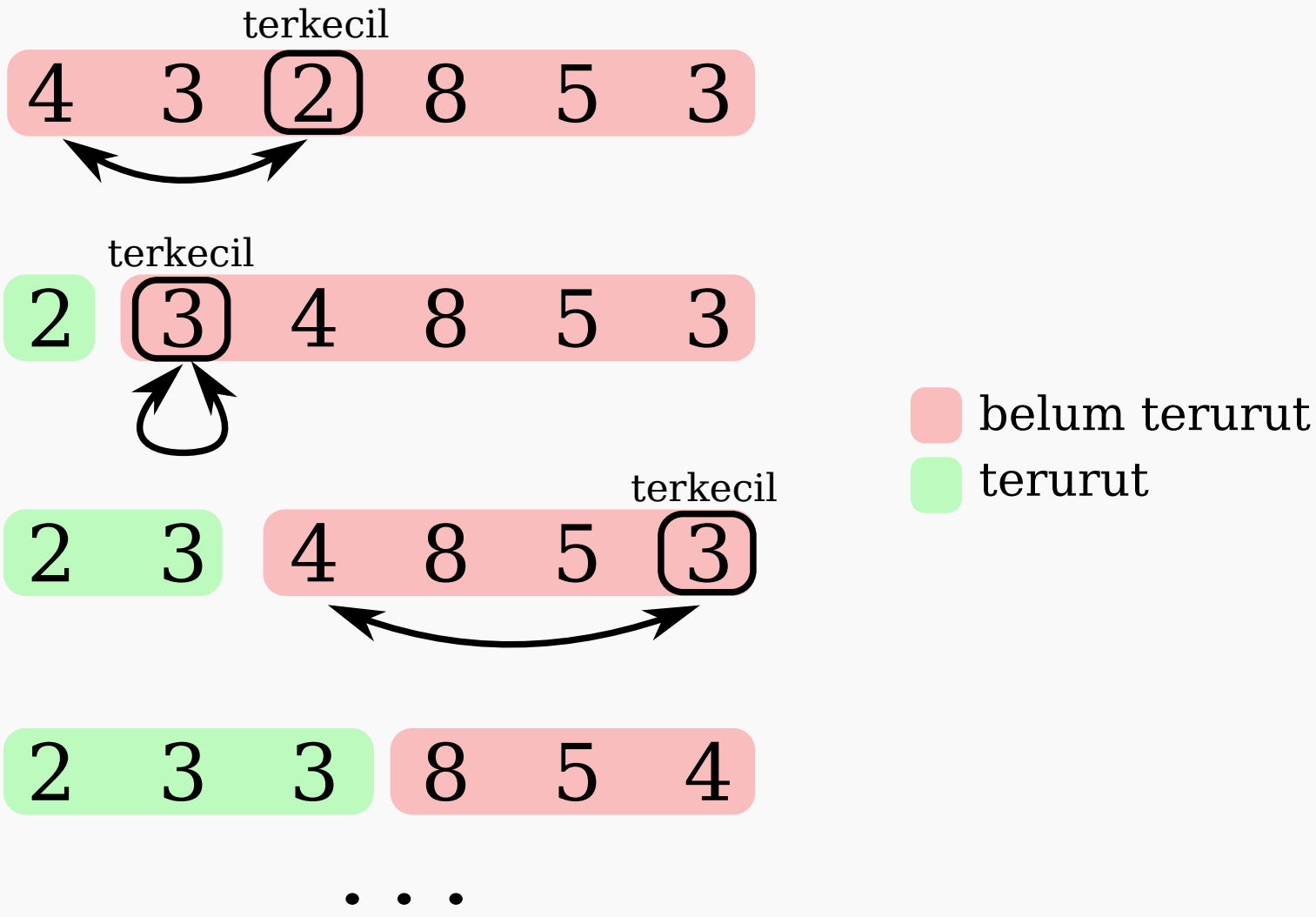
Bagian 2

Selection Sort

Ide Dasar

- Pilih elemen terkecil dari data, lalu pindahkan ke elemen pertama.
- Pilih elemen terkecil dari data yang tersisa, lalu pindahkan ke elemen kedua.
- Pilih elemen terkecil dari data yang tersisa, lalu pindahkan ke elemen ketiga.
- ... dan seterusnya sampai seluruh elemen terurut.

Ilustrasi Jalannya Algoritma



Analisis

- Pencarian elemen terkecil dapat dilakukan dengan *linear search*.
- Berhubung perlu dilakukan N kali *linear search*, maka kompleksitas selection sort adalah $O(N^2)$.

Contoh Kode

```
for i := 1 to N do begin
  (* pencarian indeks terkecil *)
  minIndex := i;
  for j := i+1 to N do begin
    if (h[j] < h[minIndex]) then begin
      minIndex := j;
    end;
  end;

  (* tukar *)
  swap(h[i], h[minIndex]);
end;
```

Kegunaan Khusus

- Cara kerja selection sort memungkinkan kita untuk melakukan *partial sort*.
- Jika kita hanya tertarik dengan K elemen terkecil, kita bisa melakukan proses seleksi dan menukar pada selection sort K kali.
- Dengan demikian, pencarian K elemen terkecil dapat dilakukan dalam $O(KN)$, cukup baik apabila K jauh lebih kecil dari N.

Bagian 3

Insertion Sort

Ide Dasar

- Anggap kita memiliki sebagian data yang terurut.
- Secara bertahap, sisipkan elemen baru ke dalam data yang sudah terurut.
- Penyisipan ini harus dilakukan sedemikian sehingga hasilnya tetap terurut.
- Misalnya saat ini data yang sudah terurut adalah [1, 2, 3, 8], lalu elemen yang akan disisipkan adalah 5, maka dihasilkan [1, 2, 3, 5, 8].

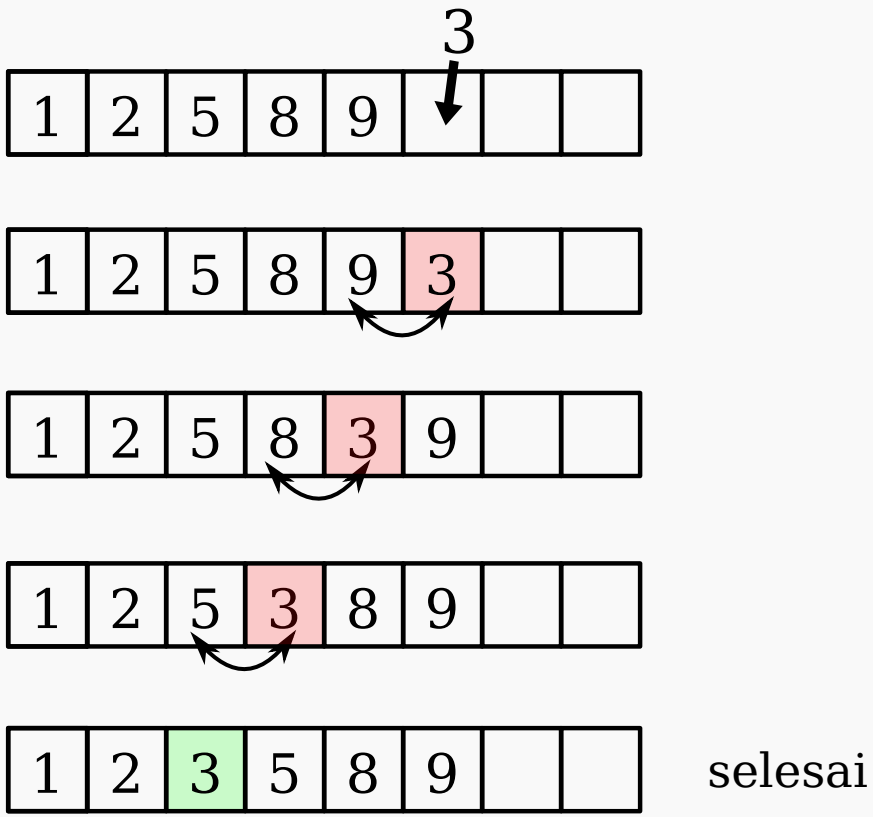
Jalannya Algoritma

Prosesnya dapat digambarkan sebagai berikut:

Data Asal	Data Terurut
[4, 3, 2, 8, 5, 3]	[]
[3, 2, 8, 5, 3]	[4]
[2, 8, 5, 3]	[3, 4]
[8, 5, 3]	[2, 3, 4]
[5, 3]	[2, 3, 4, 8]
[3]	[2, 3, 4, 5, 8]
[]	[2, 3, 3, 4, 5, 8]

Proses Penyisipan (insertion)

- Strategi yang dapat digunakan adalah meletakkan angka yang hendak disisipkan pada bagian paling belakang, lalu digiring mundur sampai posisinya tepat.
- Misalnya pada kasus menyisipkan angka 3 pada data [1, 2, 5, 8, 9]:



Analisis

- Untuk mengurutkan data, diperlukan N kali penyisipan.
- Setiap menyisipkan, dilakukan penggiringan yang kompleksitasnya:
  - Pada kasus terbaik $O(1)$, ketika angka yang dimasukkan merupakan angka terbesar pada data saat ini.
  - Pada kasus terburuk $O(N)$, yaitu ketika angka yang dimasukkan merupakan angka terkecil pada data saat ini.
  - Pada kasus rata-rata, kompleksitasnya $O(N/2)$, atau bisa ditulis $O(N)$.

Analisis (lanj.)

- Berdasarkan observasi tersebut, insertion sort dapat bekerja sangat cepat ketika datanya sudah hampir terurut.
- Pada kasus terbaik, insertion sort bekerja dalam $O(N)$, yaitu ketika data sudah terurut.
- Pada kasus terburuk, kompleksitasnya $O(N^2)$.
- Secara rata-rata, kompleksitasnya adalah $O(N^2)$.

Contoh Kode

```
for i := 1 to N do begin
  pos := i;

  (* selama belum tepat, giring ke belakang *)
  while ((pos > 1) and (h[pos] < h[pos-1])) do begin
    swap(h[pos], h[pos-1]);
    pos := pos - 1;
  end;
end;
```

Kegunaan Lain

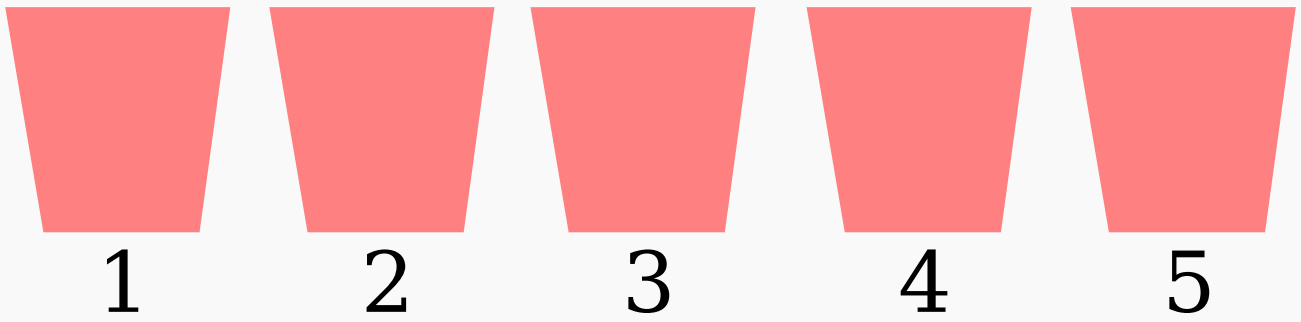
- Strategi *insertion* pada algoritma ini dapat digunakan untuk menambahkan sebuah elemen pada data yang sudah terurut.
- Ketimbang mengurutkan kembali seluruh elemen, cukup lakukan strategi *insertion* yang secara rata-rata bekerja dalam $O(N)$.

Bagian 4

Counting Sort

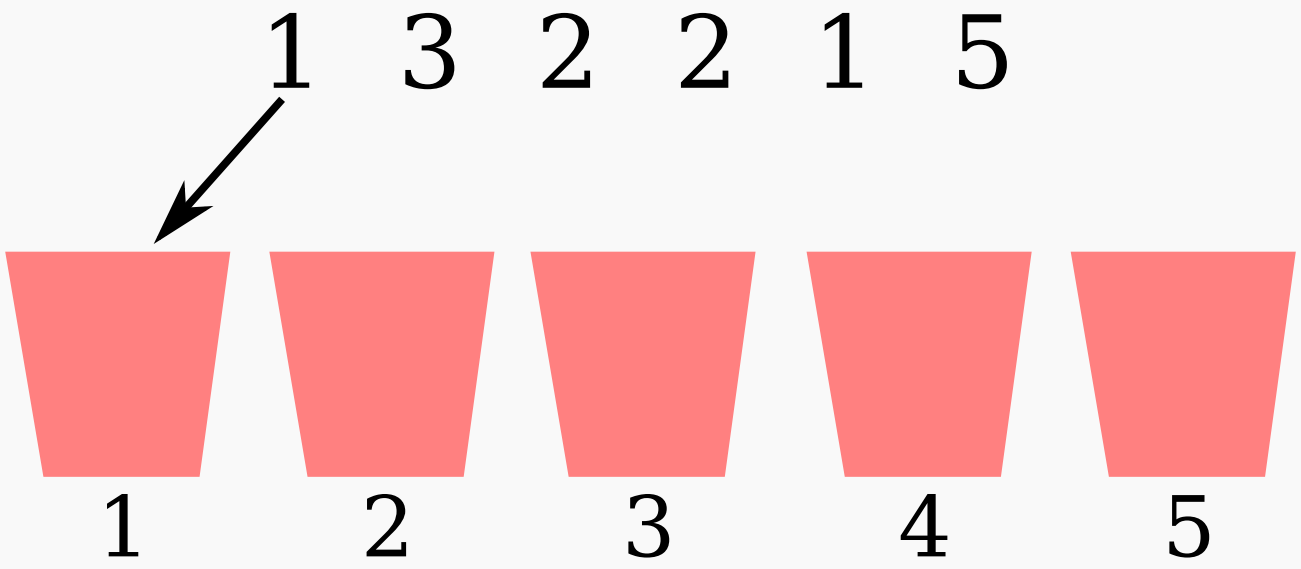
Ide Dasar

- Misalkan kita memiliki M ember.
- Setiap ember dinomori dengan sebuah angka, yaitu mulai dari 1 sampai dengan M.
- Sebagai contoh, anggap $M = 5$.



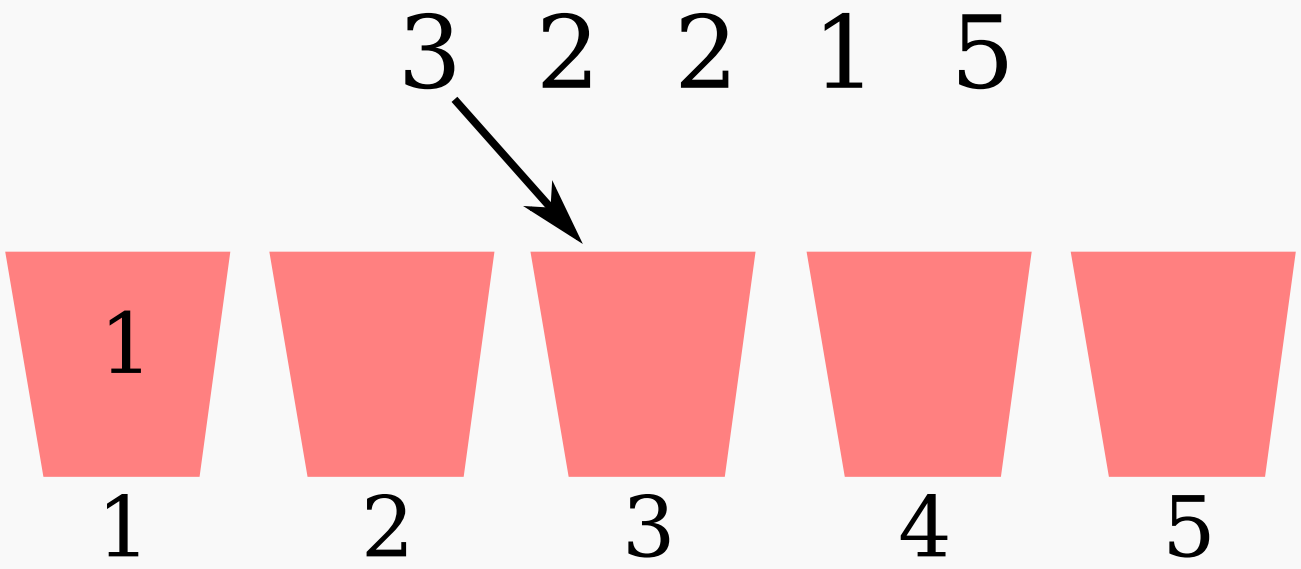
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



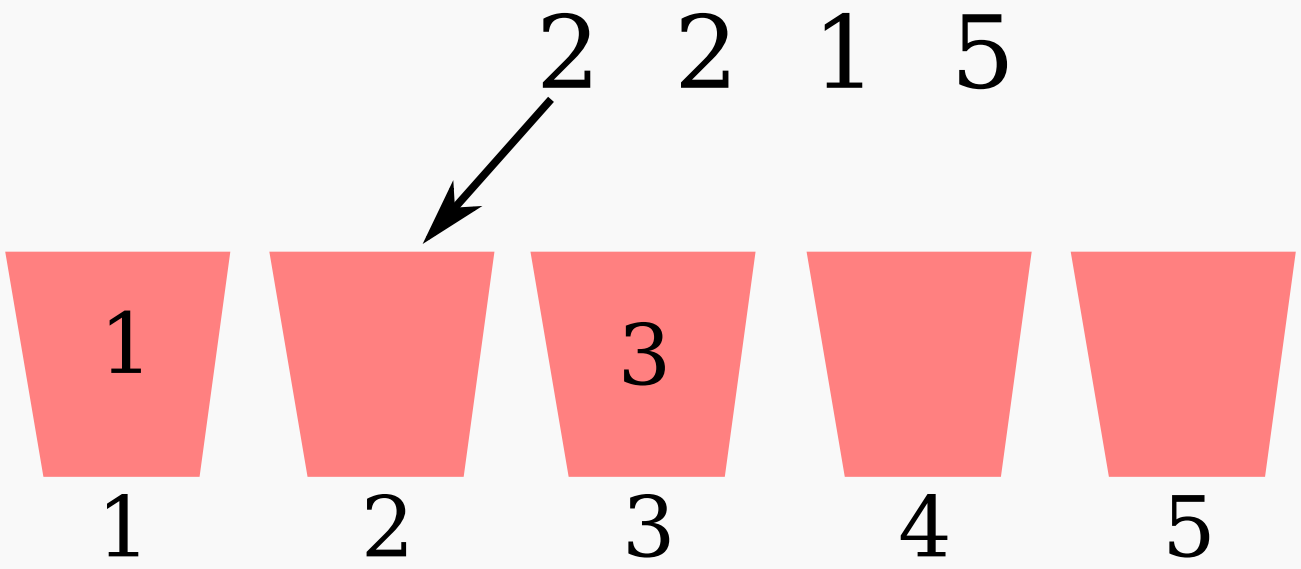
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



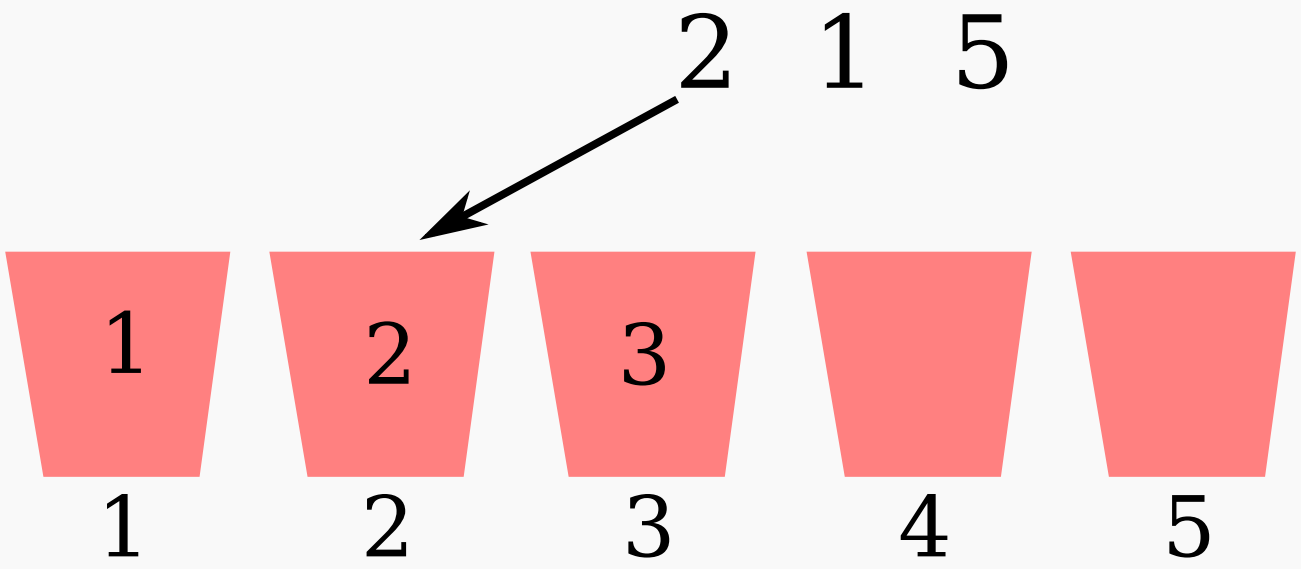
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



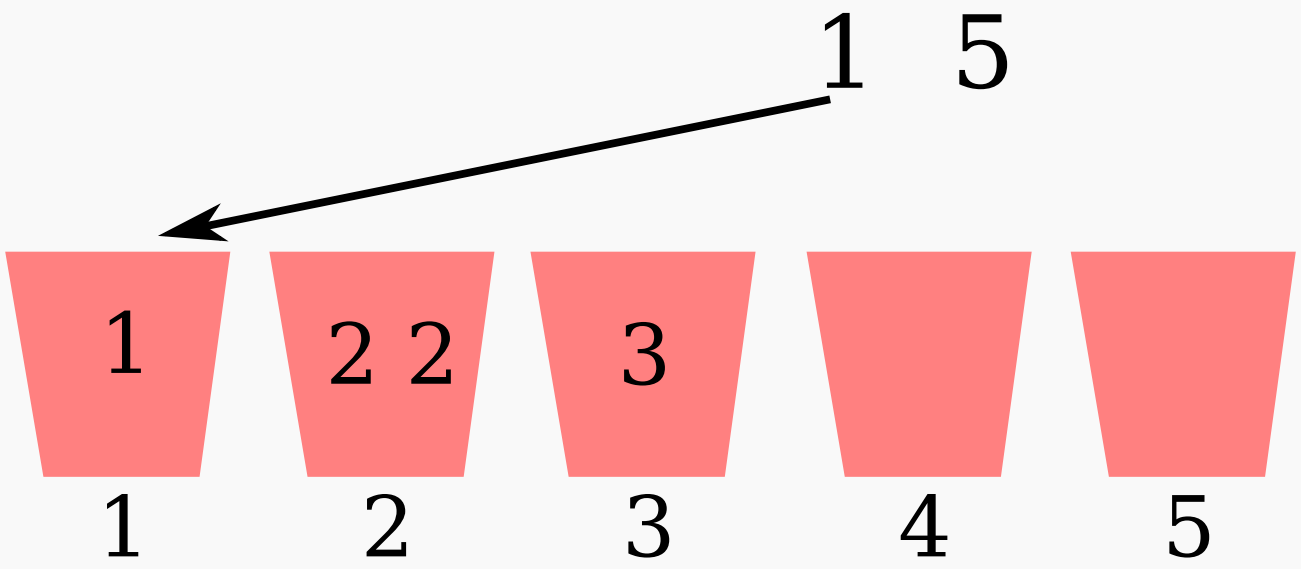
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



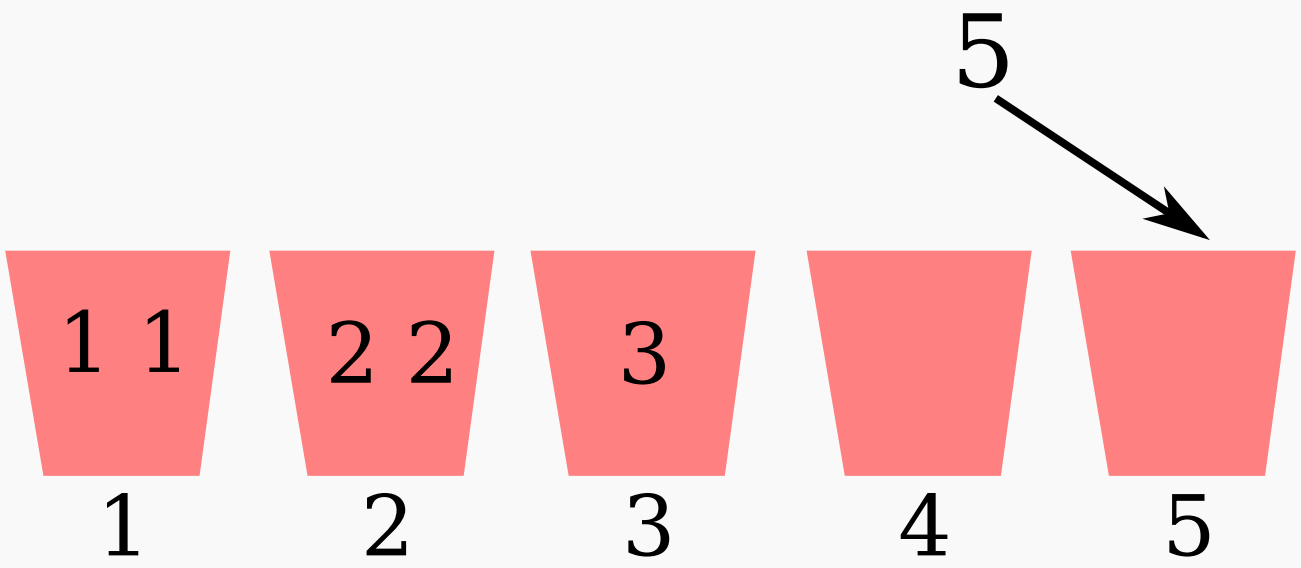
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



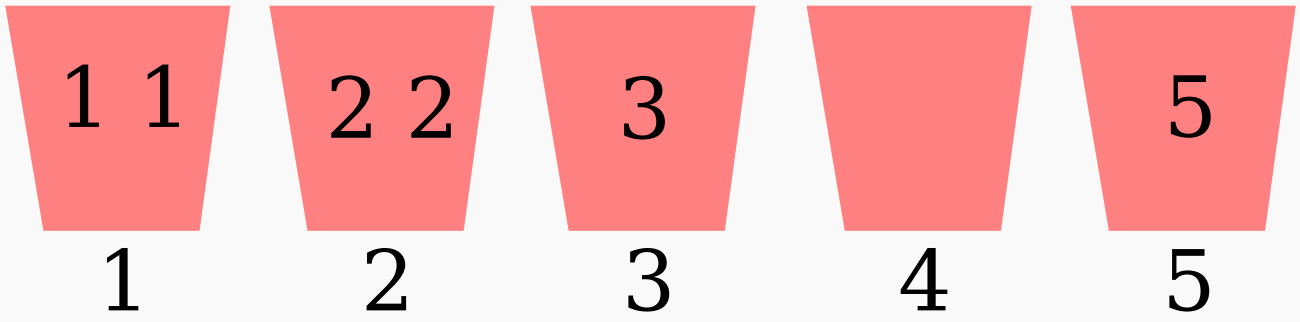
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



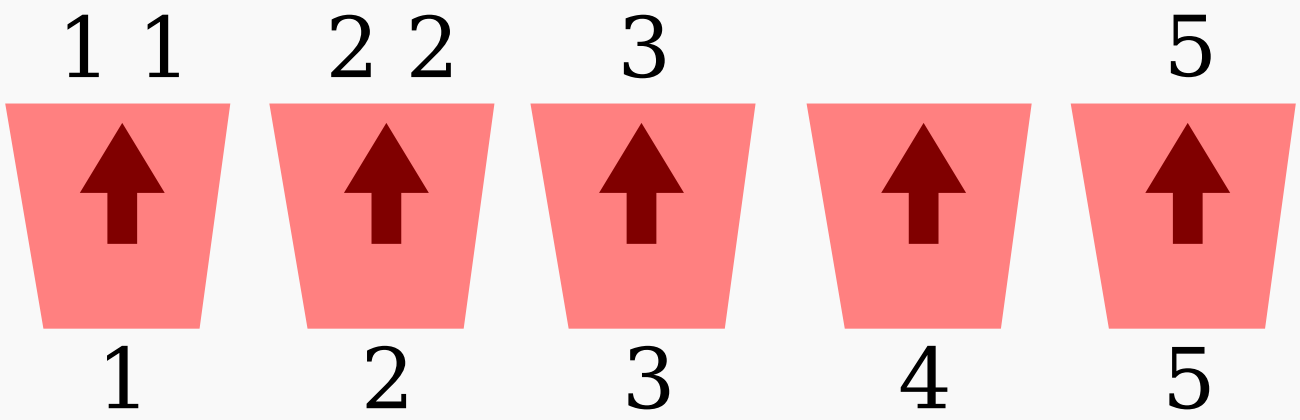
Ide Dasar (lanj.)

- Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.



Ide Dasar (lanj.)

- Setelah seluruh elemen dimasukkan ke ember yang bersesuaian, keluarkan isi ember-ember mulai dari ember 1 sampai M secara berurutan.



Ide Dasar (lanj.)

- Kini didapatkan data yang telah terurut.

1 1 2 2 3 5

Implementasi

- Ide ini dapat diwujudkan dengan menyiapkan tabel frekuensi yang berperan sebagai "ember".
- Untuk setiap nilai elemen yang mungkin, catat frekuensi kemunculannya.
- Terakhir, iterasi tabel frekuensi dari elemen terkecil sampai elemen terbesar.

Contoh Kode

- Tabel frekuensi dapat diimplementasikan dengan array sederhana.

```
var
  ftable: array[1..100000] of longint;
  ...

begin
  ...

  (* catat frekuensinya *)
  for i := 1 to N do begin
    x := h[i];
    ftable[x] := ftable[x] + 1;
  end;
```

Contoh Kode (lanj.)

```
  (* tuang kembali ke h[] *)
  index := 1;
  for i := 1 to 100000 do begin
    for j := 1 to ftable[i] do begin
      h[index] := i; (* timpa h[] dengan data terurut *)
      index := index + 1;
    end;
  end;

  ...
end.
```

Analisis

- Dapat diperhatikan bahwa kompleksitas counting sort adalah $O(N + M)$, dengan M adalah rentang nilai data.
- Jika M tidak terlalu besar, maka counting sort dapat bekerja dengan sangat cepat.
- Lebih tepatnya, counting sort merupakan opsi yang sangat baik jika datanya memiliki rentang yang kecil, misalnya data tentang usia penduduk yang rentangnya hanya [0, 125].
- Bandingkan dengan algoritma pengurutan lain yang kompleksitasnya $O(N^2)$!

Kekurangan

- Karena perlu membuat tabel frekuensi, maka counting sort hanya dapat digunakan ketika rentang nilai datanya kecil, misalnya $\leq 10^7$.
- Selain itu, algoritma ini hanya dapat mengurutkan data diskret. Data seperti bilangan pecahan tidak dapat diurutkan secara tepat.

Pengembangan Counting Sort

- Dengan adanya keterbatasan ini, counting sort dikembangkan menjadi radix sort.
- Pembelajaran tentang radix sort akan dilakukan pada kesempatan yang lain.
- Bila Anda tertarik, Anda dapat mempelajarinya **di sini**.

Rangkuman

Algoritma	Kompleksitas	Keterangan
Bubble Sort	$O(N^2)$	-
Selection Sort	$O(N^2)$	Dapat digunakan untuk *partial sort* dalam $O(KN)$
Insertion Sort	$O(N^2)$	Sangat cepat jika data hampir terurut, kasus terbaiknya $O(N)$
Counting Sort	$O(N + M)$	Cepat hanya untuk data dengan rentang yang kecil

Catatan

- Terdapat algoritma pengurutan yang lebih efisien, misalnya Quicksort dan Merge Sort.
- Algoritma pengurutan lanjut akan dipelajari pada kesempatan yang lain.