SORTING ALGORITHM

BUBBLE SORT

Bubble sort **is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. While simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes. Its average case and worst case is same O(n^2).**

SELECTION SORT

Selection sort **is a simple sorting algorithm that improves on the performance of bubble sort. It works by first finding the smallest element using a linear scan and swapping it into the first position in the list, then finding the second smallest element by scanning the remaining elements, and so on. Selection sort is unique compared to almost any other algorithm in that its running time is not affected by the prior ordering of the list: it performs the same number of operations because of its simple structure. Selection sort also requires only *n* swaps, and hence just Θ(*n*) memory writes, which is optimal for any sorting algorithm. Thus it can be very attractive if writes are the most expensive operation, but otherwise selection sort will usually be outperformed by insertion sort or the more complicated algorithms.**

INSERTION SORT

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists. This method is much more efficient than the bubble sort, though it has more constraints.

SHELL SORT

Shell sort **was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort. Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements (sets with less than 1000 or so elements). Another advantage of this algorithm is that it requires relatively small amounts of memory.**

MERGE SORT

Merge sort **takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (*i.e.* 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists.**

HEAPSORT

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O(n log n) time.

QUICKSORT

Quicksort **is a divide and conquer algorithm which relies on a *partition* operation: to partition an array, we choose an element, called a *pivot*, move all smaller elements before the pivot, and move all greater elements after it. This can be done efficiently in linear time and in-place. We then recursively sort the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log *n*) space usage, this makes quicksort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower (O(n^2)) performance, but if at each step we choose the *median* as the pivot then it works in O(*n* log *n*).**

BUCKET SORT

Bucket sort is a sorting algorithm that works by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. A variation of this method called the single buffered count sort is faster than the quick sort and takes about the same time to run on any set of data. More information is available at http://www.mcky.net/hsrto.htm

RADIX SORT

Radix sort **is an algorithm that sorts a list of fixed-size numbers of length *k* in O(*n* · *k*) time by treating them as bit strings. We first sort the list by the least significant bit while preserving their relative order using a stable sort. Then we sort them by the next bit, and so on from right to left, and the list will end up sorted. Most often, the counting sort algorithm is used to accomplish the bitwise sorting, since the number of values a bit can have is small.**

SORTING

Ascending Sorting

Descending Sorting

Contoh Algoritma Sorting :

Selection Sort

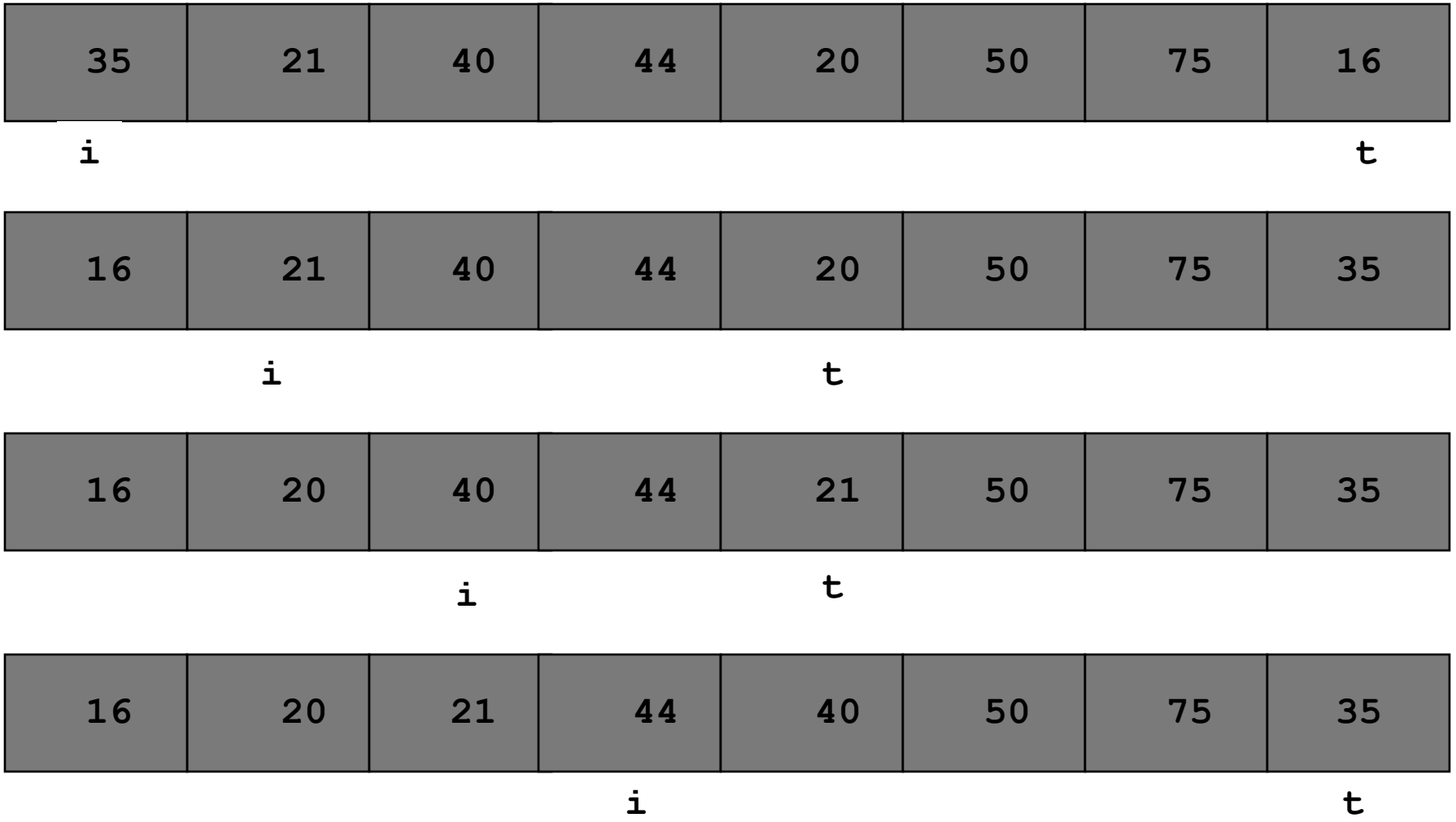
Bubble Sort

Shell Sort

SELECTION SORT

- Metode ini mulai dengan elemen pertama dan mencari pada seluruh array nilai yang terkecil
- Jika ada yang lebih kecil dari elemen pertama, akan ditukar
- Putaran kedua, akan dimulai dari elemen kedua, demikian seterusnya.
- Variabel `i` menyatakan tempat dimana elemen terkecil ditempatkan.
- Variabel `t` menyatakan elemen terkecil
- Data di dalam larik akan berubah-ubah

SELECTION SORT



SELECTION SORT

16	20	21	35	40	50	75	44

i, t

16	20	21	35	40	50	75	44

i t

16	20	21	35	40	44	75	50

i t

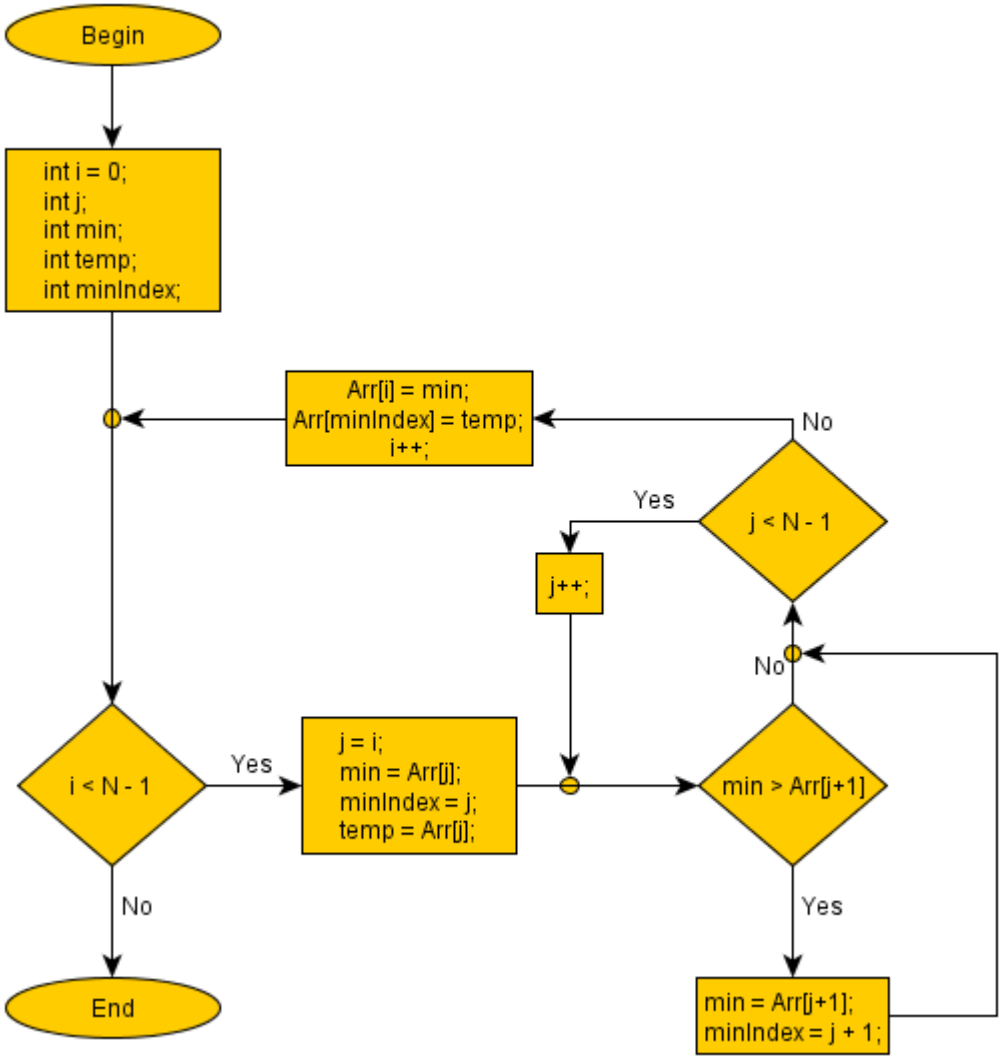
16	20	21	35	40	44	50	75

ALGORITMA SELECTION SORT

Jika `t` menyatakan elemen terkecil dari elemen ke `i` sampai dengan elemen ke-`n` dimana `n` menyatakan jumlah data yang akan diurutkan.

Algoritma untuk mencari elemen terkecil adalah: *(next slide)*

FLOWCHART SELECTION SORT



ALGORITMA SELECTION SORT

```
{cari elemen terkecil}
t := i {t adalah index elemen terkecil}
for j := i + 1 to n do
      if l[j] < l[t] then
            t := j
      eif
efor
{tukar elemen terkecil dengan elemen i}
      temp := l[j]
      l[j] := l[i]
      l[i] := temp
```

ALGORITMA SELECTION SORT

```
for i := 1 to n - 1 do
      {cari elemen terkecil}
      t := i {t adalah index elemen terkecil}
      for j := i + 1 to n do
            if l[j] < l[t] then
                  t := j
            eif
      efor
      {tukar elemen terkecil dengan elemen i}
            temp := l[j]
            l[j] := l[i]
            l[i] := temp
efor
```

ALGORITMA SELECTION SORT

SIMULASI

```
proc UrutPilih (l,n)
{mengurutkan membesar data di dalam larik l yang berelemen
sebanyak n dengan cara pemilihan}
for i := 1 to n - 1 do
        {cari elemen terkecil}
        t := i {t adalah index elemen terkecil}
        for j := i + 1 to n do
                if l[j] < l[t] then
                        t := j
                eif
        efor
        {tukar elemen terkecil dengan elemen i}
                temp := l[j]
                l[j] := l[i]
                l[i] := temp
efor
eproc
```

INSERTION SORT

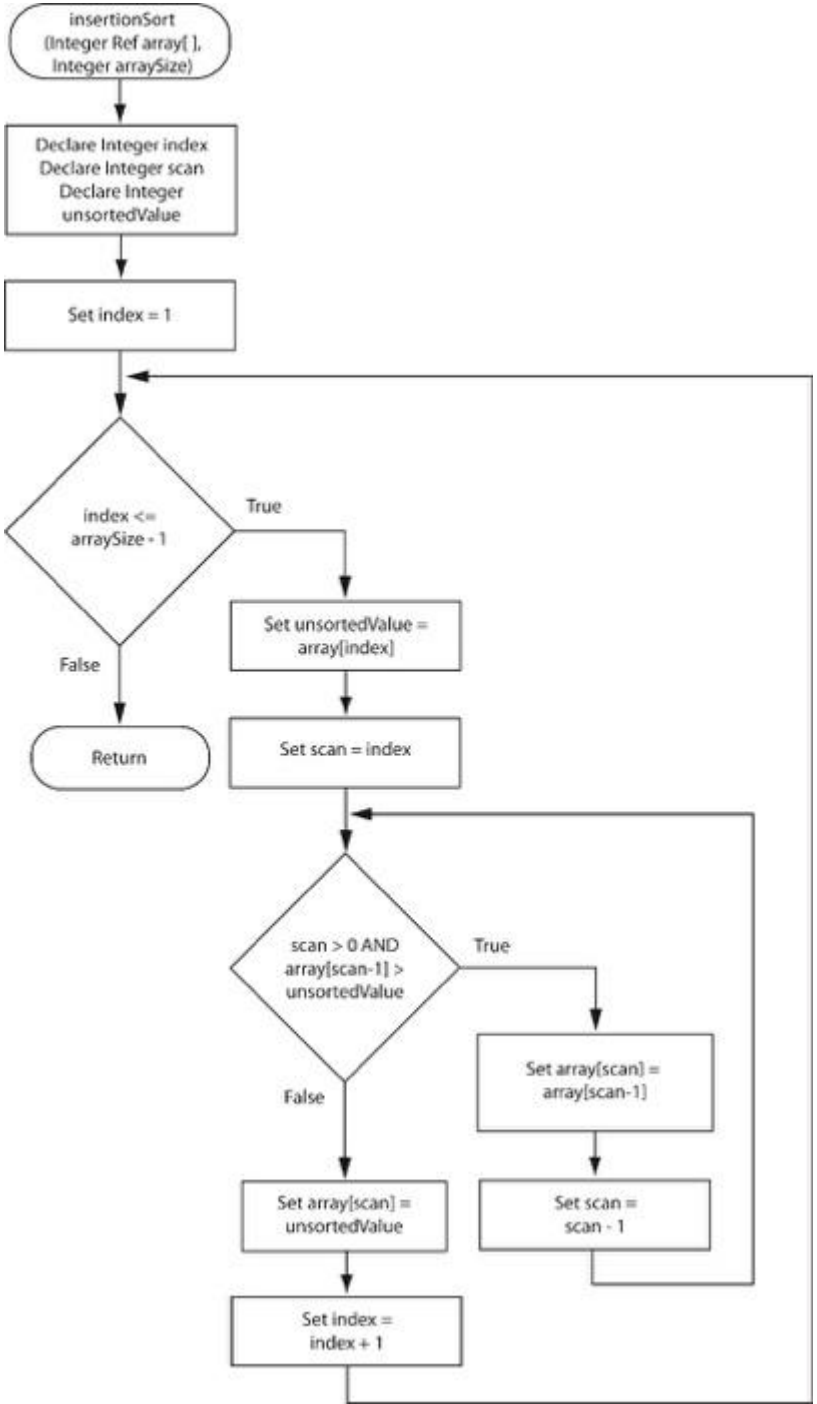
DENGAN CARA PENYISIPAN

Pengurutan dengan cara penyisipan :

Anggap elemen ke–`1` sampai dengan elemen ke `j` telah terurut

Elemen ke `i` adalah elemen yang akan disisipkan antara elemen yang telah terurut (`1` sampai `j`) pada posisi yang tepat, misalnya pada `p`

FLOWCHART INSERTION SORT



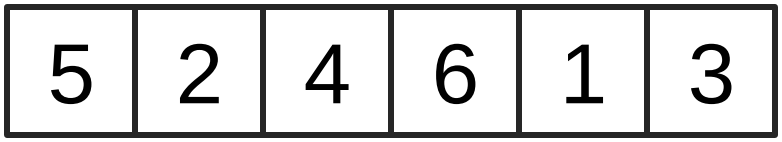
DENGAN CARA PENYISIPAN

Pencarian posisi yang tepat untuk `i` adalah sebagai berikut :

- Simpan `l[i]` pada suatu variabel temporer, temp.
- Geser elemen `j` ke bekas tempat `i` kemudian geser elemen `j - 1` ke `i - 1`, dan seterusnya sampai mencapai posisi yang tepat untuk `i`, yaitu `p`

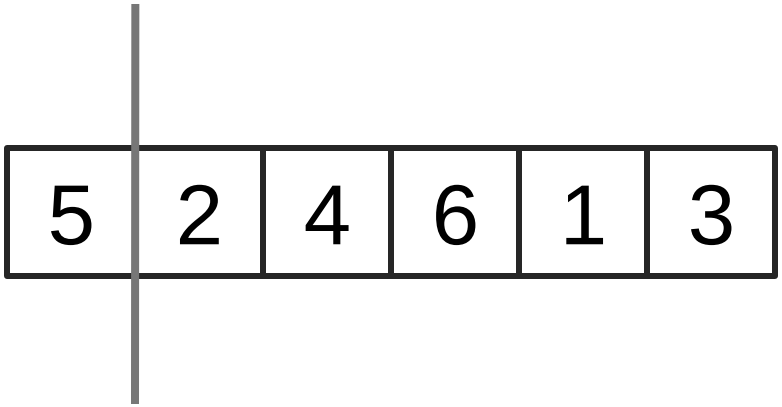
Simulasi insertion sort

CONTOH ALGORITMA



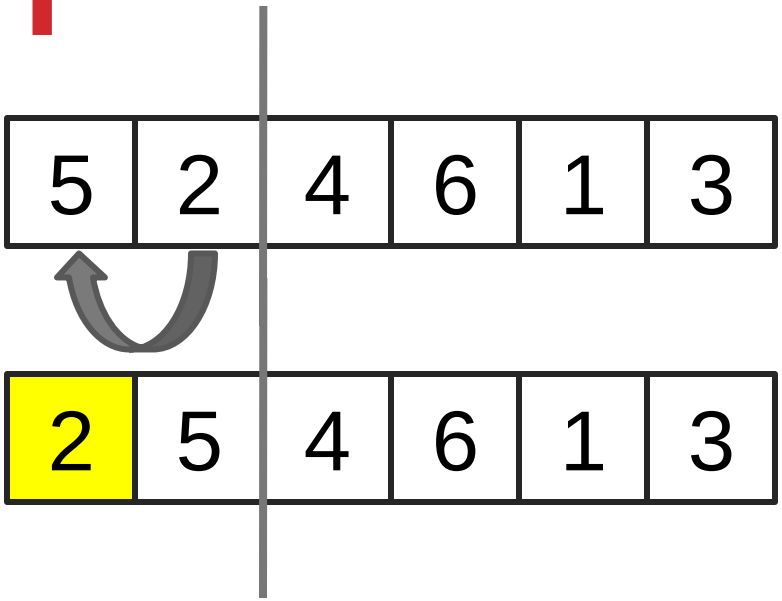
Data awal sebelum di urutkan, asumsi pertama anggaplah data indek ke- sudah benar

CONTOH ALGORITMA



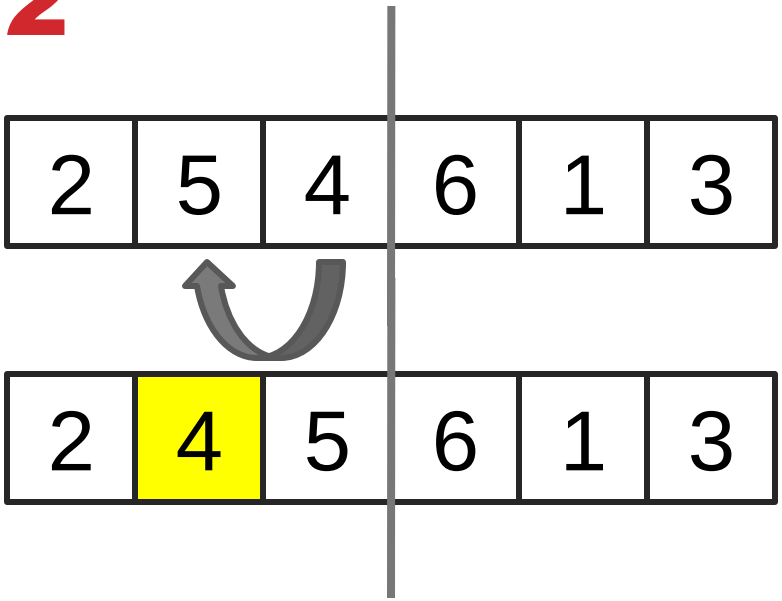
Data awal sebelum di urutkan, asumsi pertama anggaplah data indek ke-0 sudah benar

CONTOH ALGORITMA – STEP 1



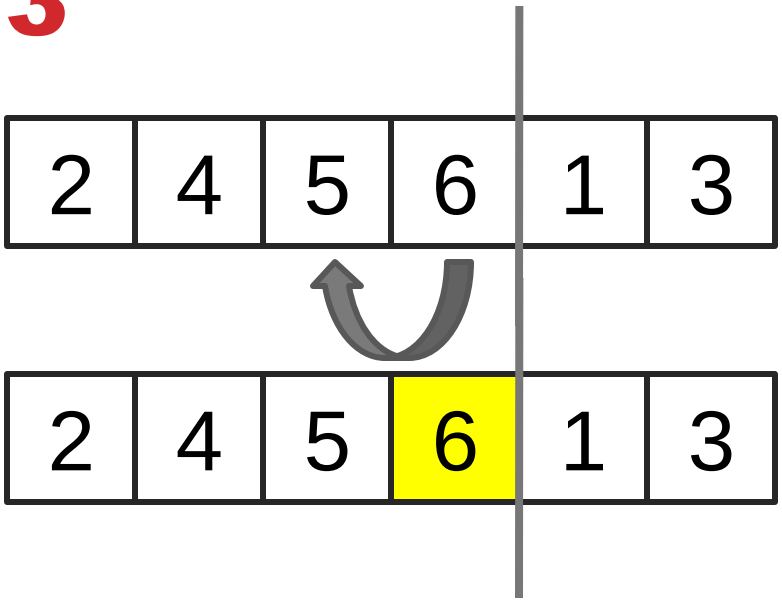
- **Bandingkan data index ke 0 dan ke 1**
- **Jika data index ke 1 lebih kecil, datanya ditukar**

CONTOH ALGORITMA – STEP 2



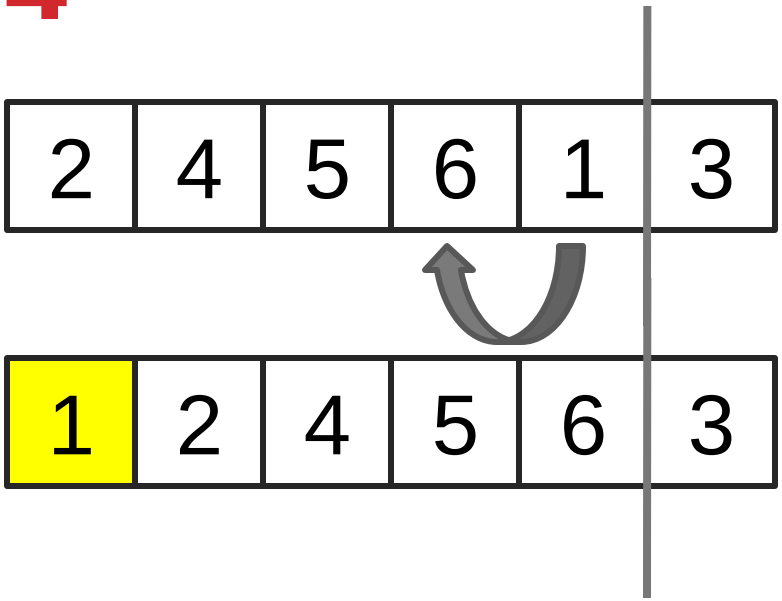
- **Bandingkan data index ke 2 dengan ke 1 dan 0**
- **Jika lebih kecil, selipkan pada posisi yang tepat**

CONTOH ALGORITMA – STEP 3



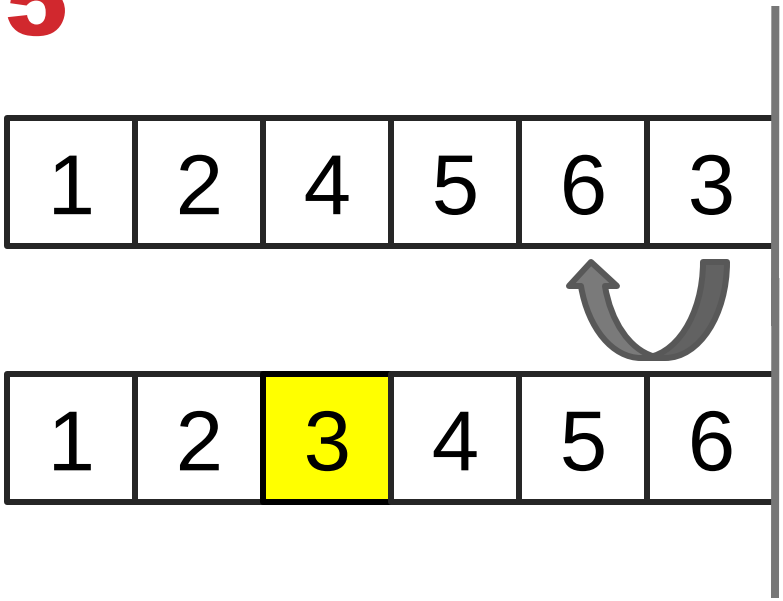
- **Bandingkan data index ke 3 dengan data didepannya**
- **Jika lebih kecil, selipkan pada posisi yang tepat. Jika tidak biarkan data tersebut**

CONTOH ALGORITMA – STEP 4



- **Bandingkan data index ke 4 dengan data didepannya**
- **Jika lebih kecil, selipkan pada posisi yang tepat.**
- **Data yang lain akan mundur posisinya, karena ada data yang maju.**

CONTOH ALGORITMA – STEP 5

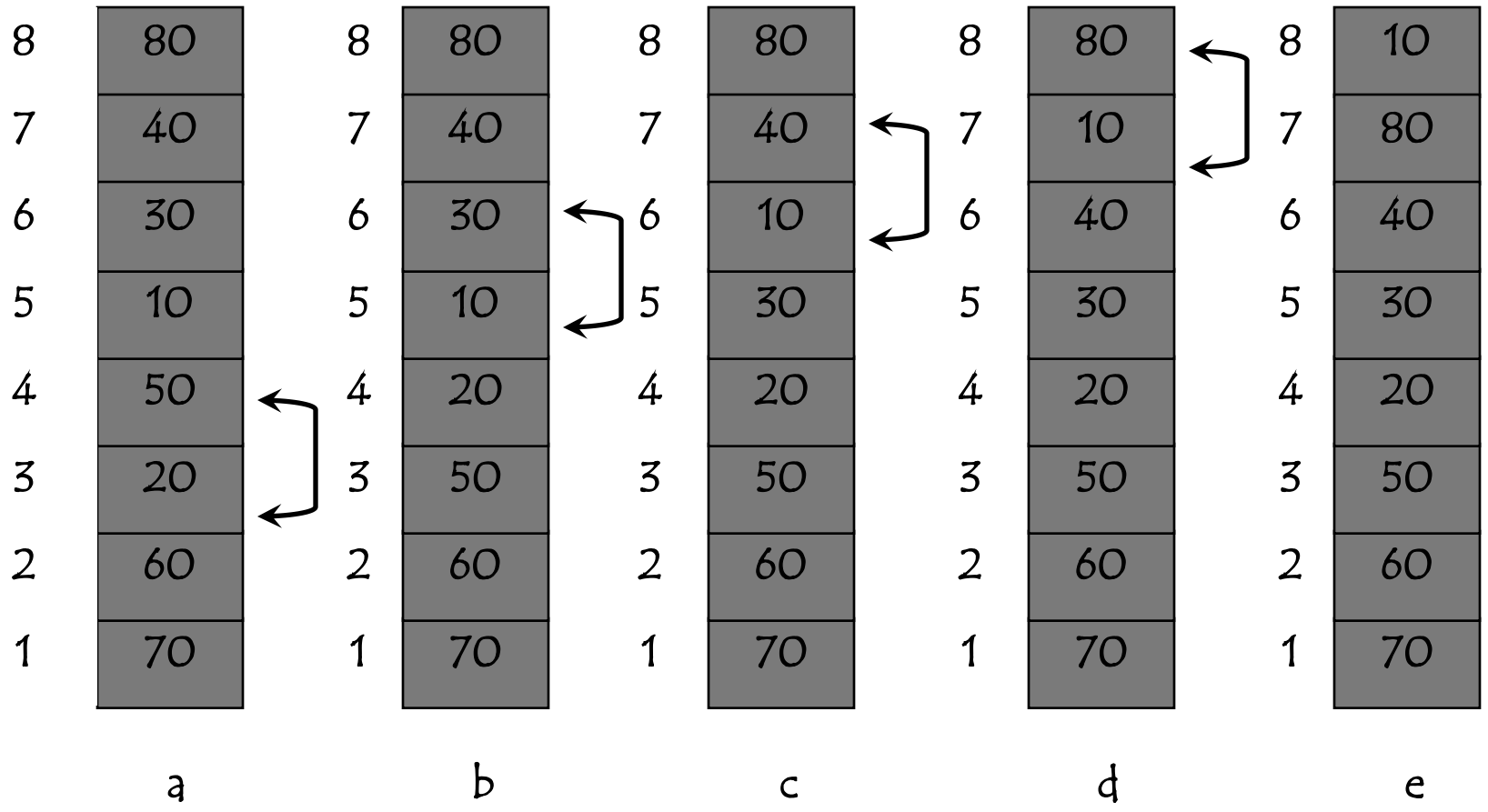


- **Bandingkan data index ke 5 dengan data didepannya**
- **Jika lebih kecil, selipkan pada posisi yang tepat.**
- **Data yang lain akan mundur posisinya, karena ada data yang maju.**

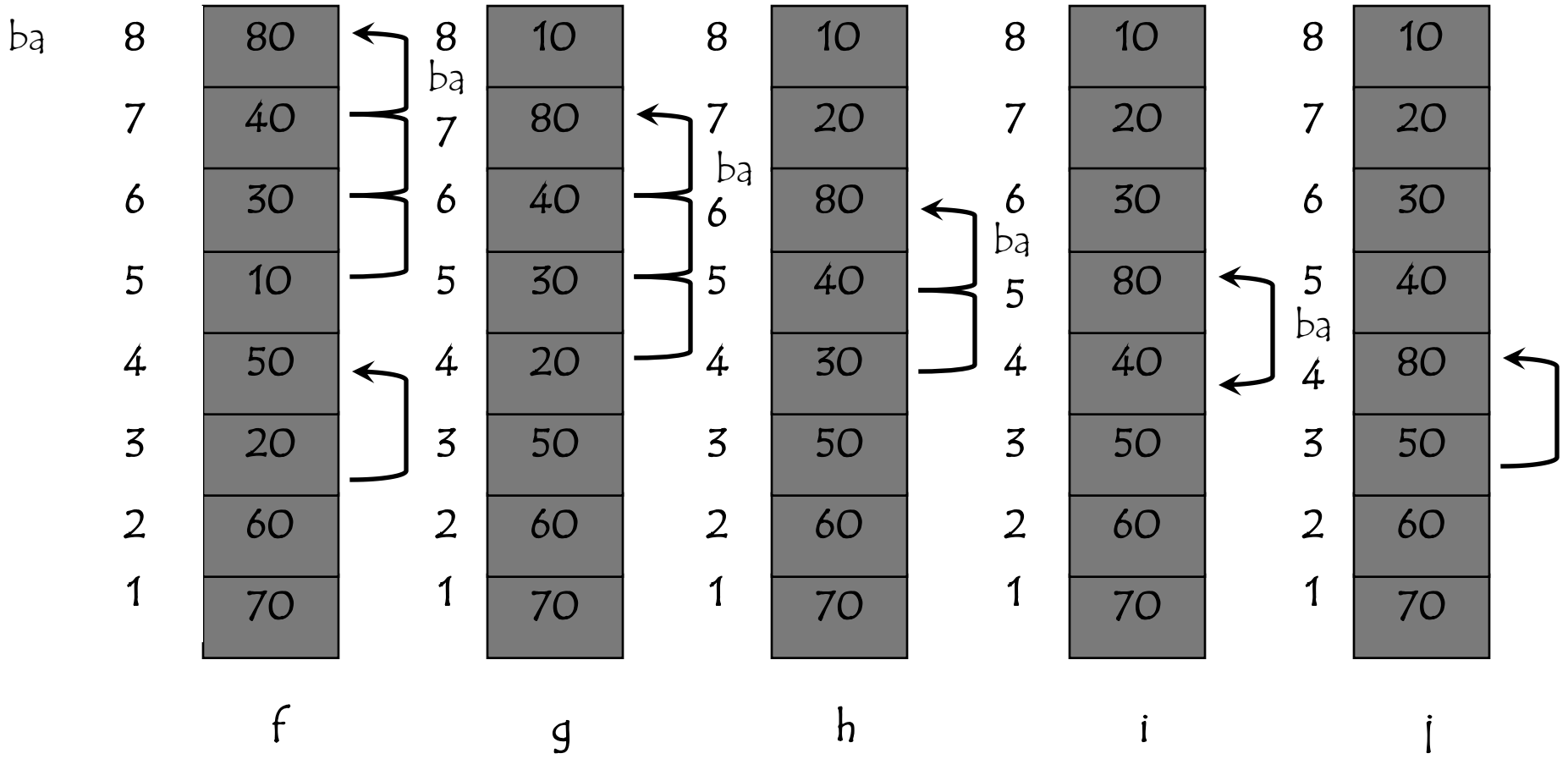
BUBBLE SORT

- Hampir sama dengan Selection Sort
- Cara pengurutan elemen yang paling sederhana
- Menggunakan metode pembandingan dan pertukaran
- Tiap putaran, elemen yang bersebelahan akan dibandingkan dan isinya akan ditukar jika nilainya tidak berurut

BUBBLE SORT

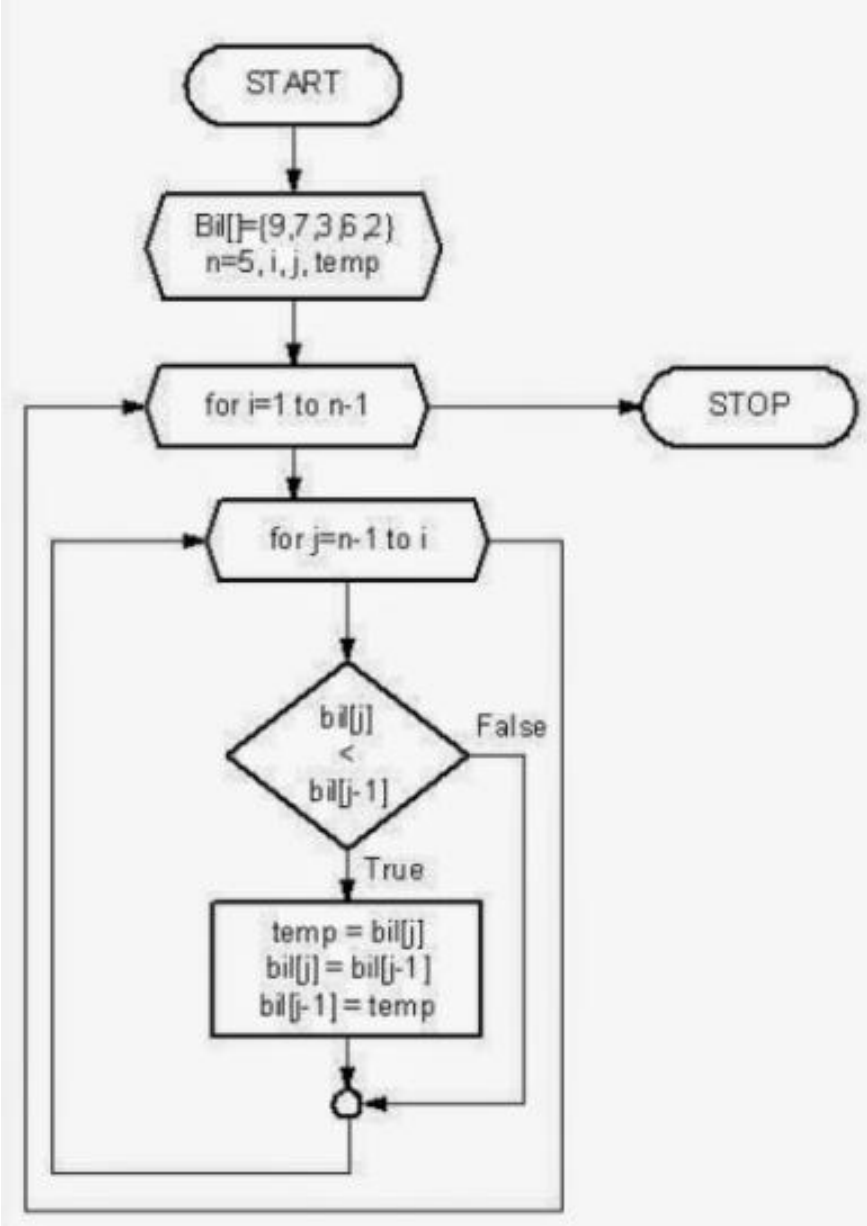

8 80 7 40 6 30 5 10 4 50 3 20 2 60 1 70
a
8 80 7 40 6 30 5 10 4 20 3 50 2 60 1 70
b
8 80 7 40 6 10 5 30 4 20 3 50 2 60 1 70
c
8 80 7 10 6 40 5 30 4 20 3 50 2 60 1 70
d
8 10 7 80 6 40 5 30 4 20 3 50 2 60 1 70
e

BUBBLE SORT



ba : batas atas pengapungan

FLOWCHART BUBBLE SORT



ALGORITMA BUBBLE SORT

```
ba := n
i := 1 {mulai dari bawah}
if l[i] < l[i+1] then {jika elemen itu lebih ringan dari pada elemen di atasnya}
    {tukarkan}
     temp := l[i]
     l[i] := l[i+1]
     l[i+1] := temp
eif
```

ALGORITMA BUBBLE SORT

Proses pemeriksaan dan penukaran seperti itu harus dilakukan mulai dari bawah sampai ke batas atas pengapungan.

Jadi, `i` harus digerakkan mulai dari `1` sampai dengan `ba-1`; algoritmanya menjadi:

ALGORITMA BUBBLE SORT

```
ba := n
i := 1 {mulai dari bawah}
while i < ba do
      if l[i] < l[i+1] then {jika elemen itu
lebih ringan      dari pada elemen di atasnya}
      {tukarkan}
            temp := l[i]
            l[i] := l[i+1]
            l[i+1] := temp
      eif
      i := i + 1
ewhile
```

ALGORITMA BUBBLE SORT

Berikutnya, batas atas pengapungan `ba` dikurangi dengan `1`

Proses pengapungan yang sama dilakukan mulai dari bawah sampai ke `ba`

Dilakukan sampai dengan nilai `ba` menjadi `2`

Algoritmanya menjadi:

ALGORITMA BUBBLE SORT

```
ba := n
while ba > 1 do
      i := 1 {mulai dari bawah}
      while i < ba do
            if l[i] < l[i+1] then {tukarkan}
                  temp := l[i]
                  l[i] := l[i+1]
                  l[i+1] := temp
            eif
            i := i + 1
            ewhile
      ba := ba - 1
ewhile
```

ALGORITMA BUBBLE SORT

Bila algoritma dituliskan dengan procedur `UrutApung`, dengan argumen `l` dan `n`, maka :

```
proc UrutApung(l,n)
ba := n
while ba > 1 do
  i := 1 {mulai dari bawah}
  while i < ba do
       if l[i] < l[i+1] then {tukarkan}
            temp := l[i]
            l[i] := l[i+1]
            l[i+1] := temp
       eif
       i := i + 1
   ewhile
   ba := ba - 1
 ewhile
eproc
```

Bubble sort simulation

SHELL SORT

Penemu : Donald Shell

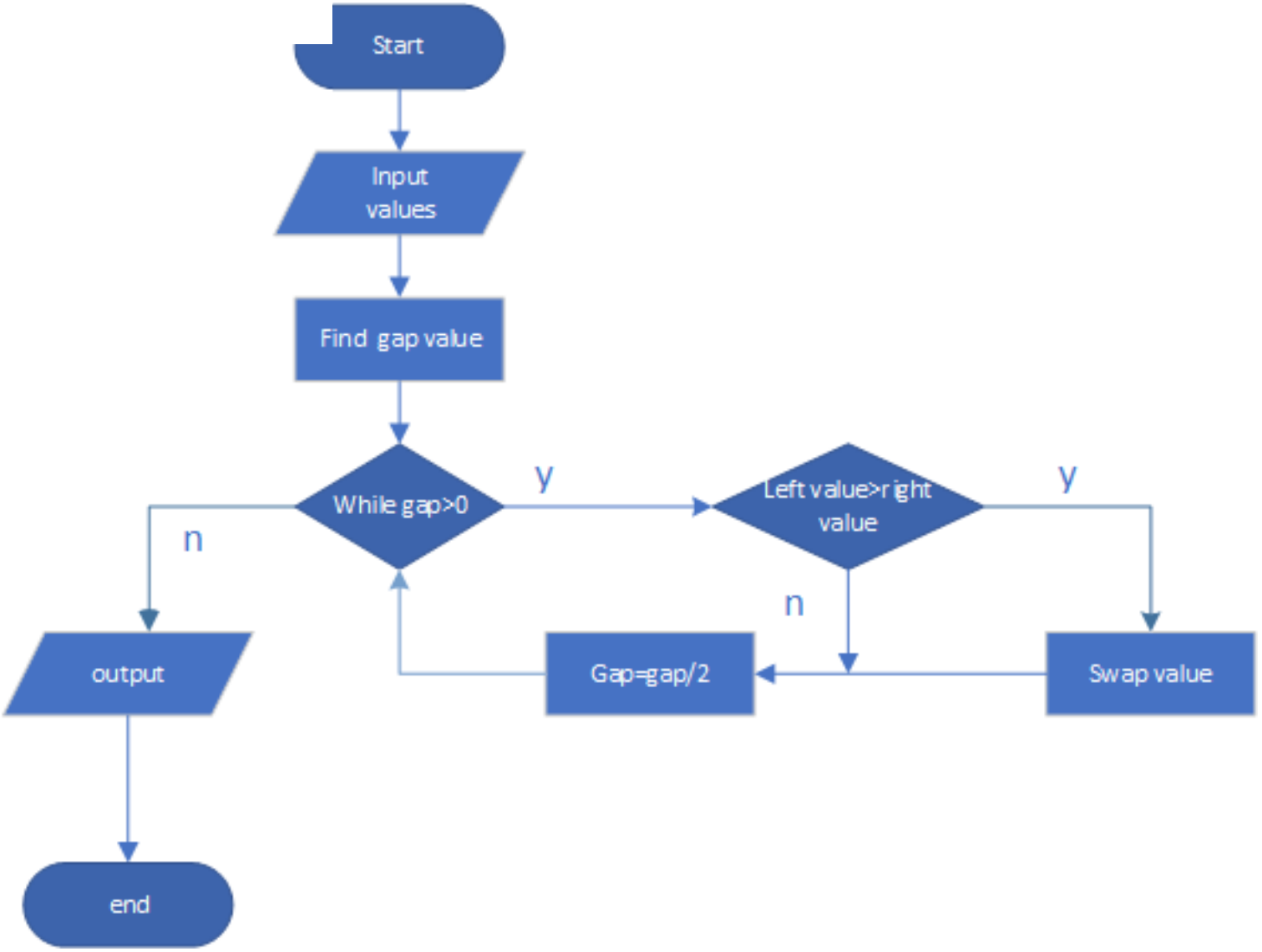
Metode perbandingan dan pertukaran

Perbandingan dimulai dari separuh array yang akan disortir dengan separuh bagian yang lain.

Contoh :

- Jika terdapat 100 elemen, diperbandingkan elemen 1 dan elemen 51, elemen 2 dan elemen 52 dst. Selanjutnya algoritma akan membandingkan elemen 1 dan elemen 26, elemen 2 dan elemen 27 dst.

FLOWCHART SHELL SORT



ALGORITMA SHELL SORT

```
Banyak = N
range = banyak / 2
while range <> 0
   counter = 1
   target = banyak-range
        while
counter<=target
          kiri = counter
          selesai = false
          while
selesai=false
              kanan =
kiri+range
    if item (kiri)>= item
       (kanan)
       selesai = true
       else
       swap item (kiri) dan
          item (kanan)
       if kiri > range
          kiri = kiri - range
       else
          selesai = true
       end_if
    end_if
    end_while
    counter = counter + 1
    end_while
   range = range / 2
end_while
```

PENJELASAN ALGORITMA

Program akan dijalankan jika `range<>0` terpenuhi

Sebelum masuk putaran ditentukan range dan target

Pada putaran ke-`1`, `range = banyak/2`

Tiap putaran dimulai dari `counter=1` sampai dengan `counter=target`

PENJELASAN ALGORITMA

Pada tiap counter dilakukan proses : `kiri = counter` dan selanjutnya,

`item(kiri)` dibandingkan dengan `item(kanan)` dimana : `kanan = kiri + range`

Jika `item(kiri) >= item(kanan)` maka proses selesai dan dilanjutkan counter atau mungkin putaran berikutnya

PENJELASAN ALGORITMA

Jika `item(kiri) < item(kanan)` maka terjadi pertukaran, selanjutnya :

jika `item kiri < range` maka proses selesai dan dilanjutkan counter berikutnya

jika `kiri > range` maka `kiri = kiri – range` dan proses dimulai dari awal pembandingan `item(kiri)` dan `item(kanan)` lagi

Jika semua `counter` pada suatu putaran telah selesai maka range akan dihitung kembali yaitu : `range = range/2`.

Jika `range <> 0` maka program akan dijalankan sampai `range = 0` berarti data telah terurut

Shell Sort

Goto simulation

Elemen ke	1	2	3	4	5	6	7	8	9	10
Nilai awal	14	6	23	18	7	47	2	83	16	38
Untuk Range 5										
Setelah putaran ke-1	47	6	23	18	7	14	2	83	16	38
Setelah putaran ke-2	47	6	23	18	7	14	2	83	16	38
Setelah putaran ke-3	47	6	83	18	7	14	2	23	16	38
Setelah putaran ke-4	47	6	83	18	7	14	2	23	16	38
Setelah putaran ke-5	47	6	83	18	38	14	2	23	16	7
Untuk Range 2										
Setelah putaran ke-6	83	6	47	18	38	14	2	23	16	7
Setelah putaran ke-7	83	18	47	6	38	14	2	23	16	7
Setelah putaran ke-8	83	18	47	6	38	14	2	23	16	7
Setelah putaran ke-9	83	18	47	14	38	6	2	23	16	7
Setelah putaran ke-10	83	18	47	14	38	6	2	23	16	7
Setelah putaran ke-11	83	23	47	18	38	14	2	6	16	7
Setelah putaran ke-12	83	23	47	18	38	14	16	6	2	7
Setelah putaran ke-13	83	23	47	18	38	14	16	7	2	6
Untuk Range 1										
Setelah putaran ke-14	83	23	47	18	38	14	16	7	2	6
Setelah putaran ke-15	83	47	23	18	38	14	16	7	2	6
Setelah putaran ke-16	83	47	23	18	38	14	16	7	2	6
Setelah putaran ke-17	83	47	38	23	18	14	16	7	2	6
Setelah putaran ke-18	83	47	38	23	18	14	16	7	2	6
Setelah putaran ke-19	83	47	38	23	18	16	14	7	2	6
Setelah putaran ke-20	83	47	38	23	18	16	14	7	2	6
Setelah putaran ke-21	83	47	38	23	18	16	14	7	2	6
Setelah putaran ke-22	83	47	38	23	18	16	14	7	6	2

QUICK SHORT

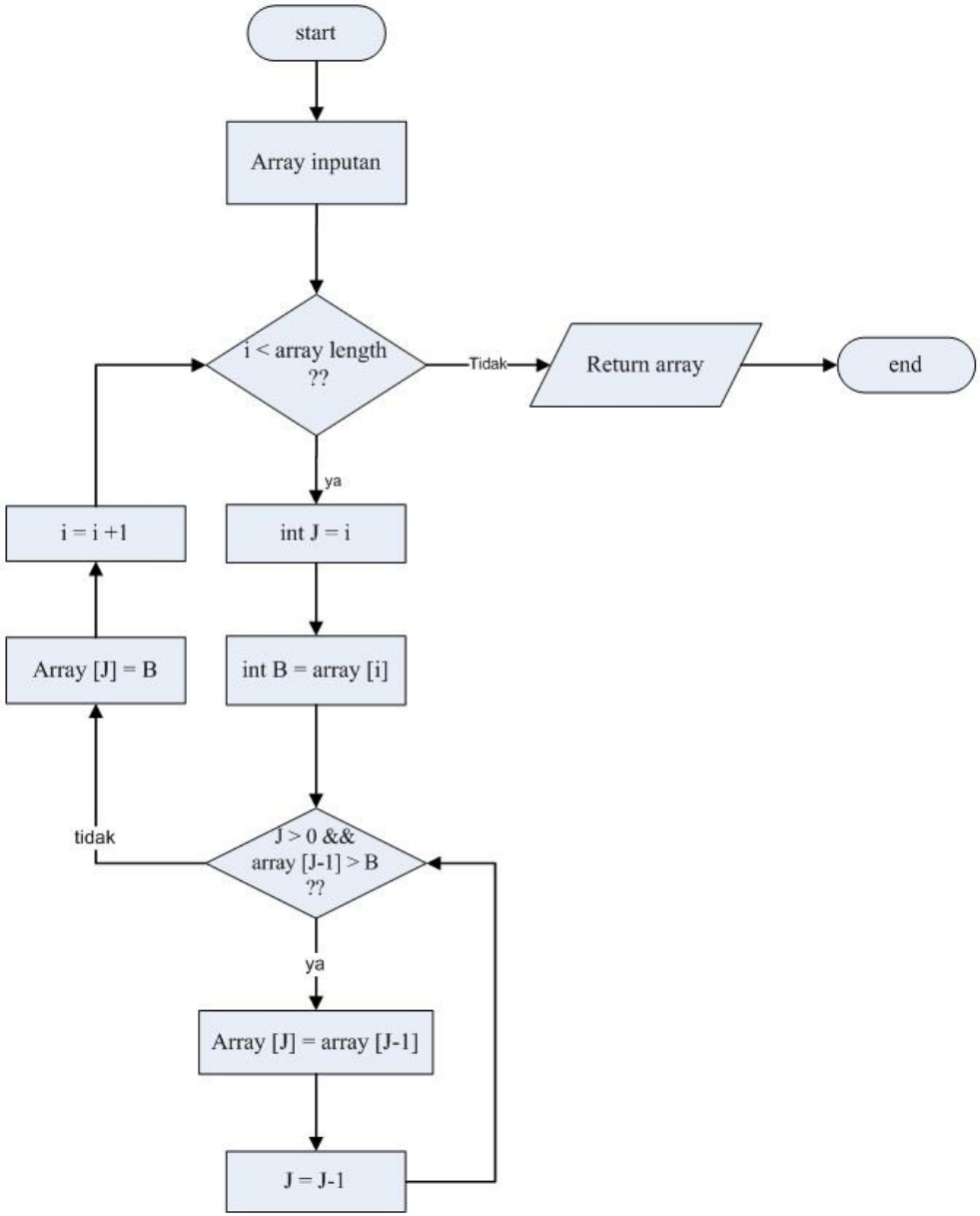
Merupakan membandingkan suatu elemen (disebut juga pivot) dengan elemen yang lain dan menyusunnya sedemikian rupa sehingga elemen-elemen lainnya yang lebih kecil daripada pivot tersebut terletak disebelah kirinya dan elemen-elemen lain yang lebih besar daripada pivot terletak disebelah kanannya.

Dengan demikian telah terbentuk dua sublist, yang terletak di sebelah kiri dan kanan dari pivot.

Lalu pada sublist kiri dan sublist kanan anggap sebuah list baru dan kerjakan proses yang sama seperti sebelumnya.

Demikian seterusnya sampai tidak terdapat sublist lagi. Sehingga didalamnya terjadi sebuah proses rekursif.

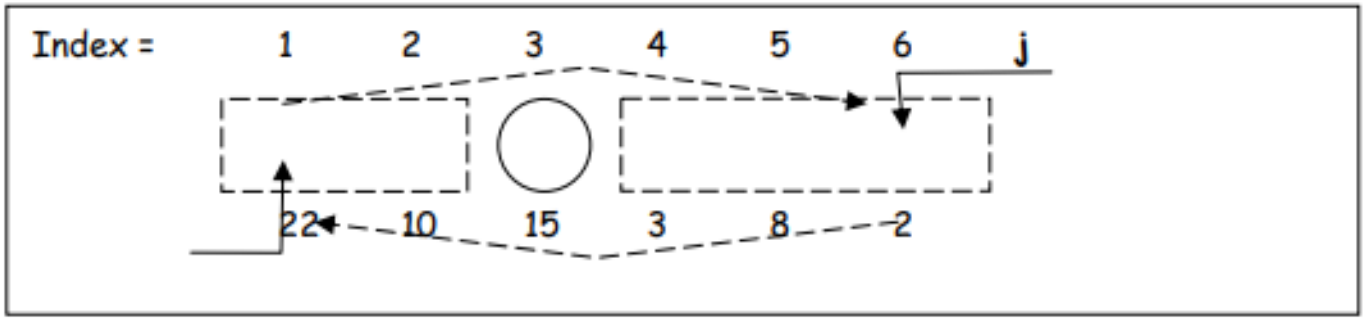
FLOWCHART ALGORITMA



PENJELASAN ALGORITMA

Proses :

1. **Bilangan yang didalam kurung merupakan pivot**
2. **Persegi panjang yang digambarkan dengan garis terputu s2 menunjukan sublist**
3. **i bergerak dari sudut kiri ke kanan sampai mendapatkan nilai yang >= pivot**
4. **j bergerak dari sudut kanan ke kiri sampai menemukan ni lai yang < pivot**

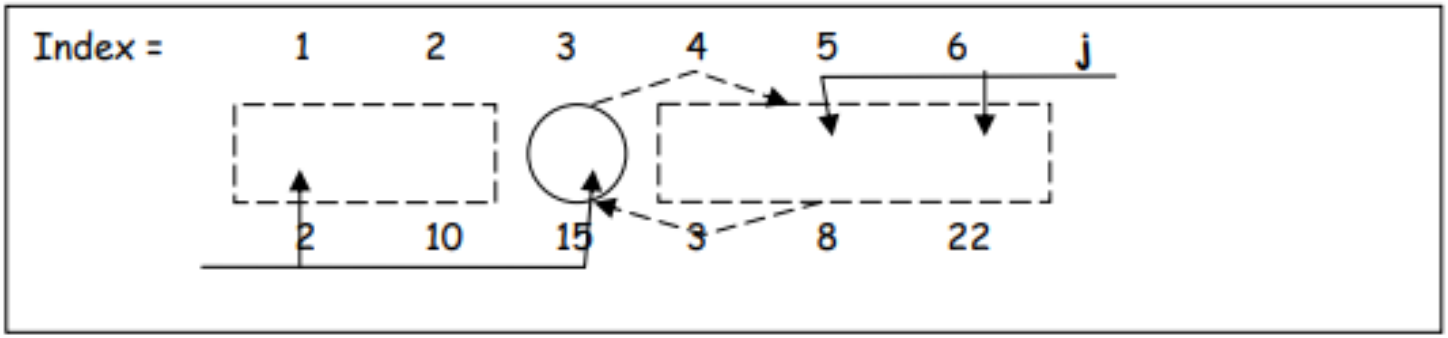


PENJELASAN ALGORITMA

Proses :

5. **berhenti pada index ke-1 karena langsung mendapatk an nilai yang > dari pivot(15).**
6. **berhenti pada index ke-6 karena juga langsung mend apatkan nilai yang < dari pivot.**
7. **Karena i< j maka data yang ditunjuk oleh I ditukar dengan data yang ditunjuk oleh j sehingga urutan menjadi :**

2 10 15 3 8 22



PENJELASAN ALGORITMA

Proses :

8. **i berhenti pada index ke-3 (pivot) karena tidak menemukan bilangan yang > dari pivot.**
9. **j berhenti pada index ke-5 menunjuk pada nilai yang < dari pivot.**
10. **karena i<j maka data yang ditunjuk oleh I (pivot) ditukar dengan data yang ditunjuk oleh j sehingga menjadi :**

2 10 8 3 15 22

11. **Proses yang sama seperti sebelumnya dilakukan terhadap 2 buah sublist yang baru. Sehingga menjadi :**

2 3 8 10 15 22 6.