6.1 - Divide and Conquer (part 1)

[KOMS120403]

Desain dan Analisis Algoritma (2023/2024)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 6 (March 2024)

Daftar isi

- Prinsip algoritma "Divide-and-Conquer"
- Analisis kompleksitas waktu untuk Divide-and-Conquer
- Contoh algoritma "Divide-and-Conquer": Masalah MinMax
- "Divide-and-Conquer" berdasarkan pengurutan
  - Merge Sort
  - Insertion Sort
  - Quick Sort
  - Selection Sort

Bagian 1. Skema algoritma Divide and Conquer (DnC)

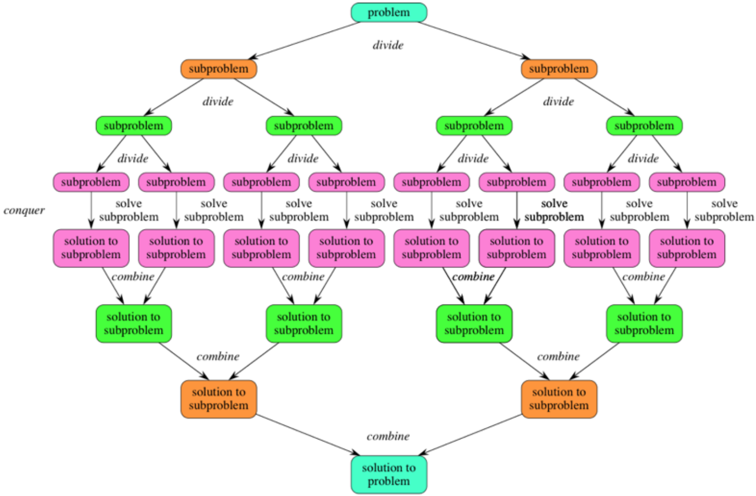
Prinsip dari algoritma **divide-and-conquer**

DIVIDE: memecah masalah menjadi dua atau lebih sub-masalah yang memiliki jenis yang sama atau serupa, hingga menjadi cukup sederhana untuk diselesaikan secara langsung. Idealnya, ukuran sub-masalah sama.

CONQUER: menyelesaikan setiap sub-masalah, secara langsung (jika ukurannya kecil) atau secara rekursif (jika ukurannya masih besar).

COMBINE: menggabungkan solusi untuk sub-masalah untuk menghasilkan solusi untuk masalah asli.

Prinsip dari algoritma divide-and-conquer



source: https://cdn.kastatic.org/ka-perseus-images/db9d172fc33b90e905c1213b8cce660c228bb99c.png

Contoh soal yang dapat diselesaikan dengan algoritma DnC

1. Merge sort
2. Quick sort
3. Masalah pasangan terpendek
4. Perkalian matriks
5. Algoritma Strassen
6. Algoritma Karatsuba untuk perkalian cepat
7. Perkalian dua polinomial

Divide-and-Conquer & Brute force

Studi kasus: jumlah array bilangan bulat

Permasalahan

Diberikan array yang berisi n bilangan bulat $a_0, a_1, \ldots, a_{n-1}$.
Temukan $a_0 + a_1 + \cdots + a_{n-1}$.

Penyelesaian dengan brute-force? tambahkan elemen secara berurutan (satu per satu)

Penyelesaian dengan Divide-and-Conquer:

- Jika $n = 1$, maka *return* a_0;
- Jika $n > 1$, maka lakukan hal berikut secara rekursif: bagi menjadi dua sub-array, lalu hitung jumlah dari setiap sub-array.

$$a_0 + a_1 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

Teknik manakah yang lebih efisien?

Divide-and-Conquer & Brute force

Studi kasus: jumlah array bilangan bulat

Permasalahan

Diberikan array yang berisi n bilangan bulat $a_0, a_1, \ldots, a_{n-1}$.
Temukan $a_0 + a_1 + \cdots + a_{n-1}$.

Penyelesaian dengan brute-force? tambahkan elemen secara berurutan (satu per satu)

Penyelesaian dengan Divide-and-Conquer:

- Jika $n = 1$, maka *return* a_0;
- Jika $n > 1$, maka lakukan hal berikut secara rekursif: bagi menjadi dua sub-array, lalu hitung jumlah dari setiap sub-array.

$$a_0 + a_1 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

Teknik manakah yang lebih efisien?

Divide and conquer vs Brute force

- DnC mungkin merupakan teknik desain algoritma umum yang paling terkenal.
- Tidak setiap algoritma divide-and-conquer lebih efisien daripada brute-force.
- Seringkali, waktu yang dibutuhkan untuk mengeksekusi algoritma DnC secara signifikan lebih kecil daripada menyelesaikan masalah dengan metode yang berbeda.
- Pendekatan DnC menghasilkan beberapa algoritma yang paling penting dan efisien dalam CS.

Skema Divide-and-Conquer

Algorithm 1 General scheme of divide-and-conquer

```
procedure DivideConquer(P: problem, n: integer)
    if n ≤ n_0 then                                   ▷ P is small enough
        Solve P
    else
        Divide to r sub-problems P_1, ..., P_r of size n_1, ..., n_r
        for each P_1, ..., P_r do
            DivideConquer(P_i, n_i)
        end for
        Combine the solutions of P_1, ..., P_r to solution of P
    end if
end procedure
```

Bagian 2. Analisis kompleksitas waktu Divide-and-Conquer

Analisis kompleksitas waktu Divide-and-Conquer

$$T(n) = \begin{cases} g(n), & n \leq n_0 \\ T(n_1) + T(n_2) + \cdots + T(n_r) + f(n), & n \geq n_0 \end{cases}$$

- $T(n)$: kompleksitas waktu masalah P (dengan ukuran n)
- $g(n)$: kompleksitas waktu untuk $\textsc{Solve}$ jika n kecil (mis. $n \leq n_0$)
- $T(n_1) + T(n_2) + \cdots + T(n_r)$: kompleksitas waktu untuk menyelesaikan setiap sub-masalah
- $f(n)$: kompleksitas waktu untuk $\textsc{Membagi}$ (*Divide*) masalah dan $\textsc{Menggabungkan}$ (*Conquer*) solusi dari setiap sub-masalah

Analisis kompleksitas waktu Divide-and-Conquer

Situasi ideal adalah ketika operasi DIVIDE *selalu menghasilkan dua sub-masalah dengan ukuran setengah dari masalah*.

```
procedure DIVIDECONQUER(P: problem, n: integer)
    if n ≤ n_0 then                                    ▷ P is small enough
        SOLVE P
    else
        DIVIDE to 2 sub-problems P_1, P_2 of size n/2
        DIVIDECONQUER(P_1, n/2)
        DIVIDECONQUER(P_2, n/2)
        COMBINE the solutions of P_1, P_2 to solution of P
    end if
end procedure
```

Analisis kompleksitas waktu Divide-and-Conquer

Jika instance selalu dapat dibagi menjadi dua sub-instance pada setiap langkah, maka:

$$T(n) = \begin{cases} g(n), & n \leq n_0 \\ 2T(n/2) + f(n), & n \geq n_0 \end{cases}$$

Secara lebih umum, jika instance selalu dibagi menjadi $b \geq 1$ instance dengan ukuran yang sama, dimana $a \geq 1$ instance perlu diselesaikan, maka kompleksitasnya diberikan oleh:

$$T(n) = aT(n/b) + f(n)$$

Dalam hal ini, nilai $T(n)$ bergantung pada nilai konstanta a dan b dan kecepatan pertumbuhan fungsi $f(n)$.

Bagian 3. Permasalahan MinMax: Contoh algoritma DnC

Permasalahan MinMax (1)

Permasalahan

Diberikan array A dari n bilangan bulat. Temukan nilai minimum dan maksimum array tersebut dengan <u>satu</u> algoritma.

Contoh:

4	10	21	11	23	3	42	34	1

$\min = 1$
$\max = 42$

Figure: Array bilangan bulat, dan nilai min & maks dari array

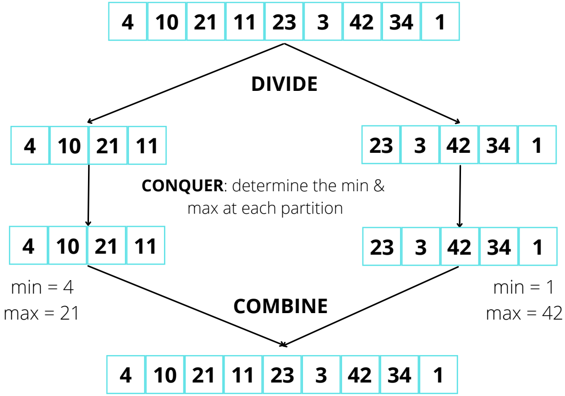
Permasalahan MinMax (2)

Algorithm 2 MinMax (brute-force)

```
procedure MINMAX1(A[0..n − 1]: array, n: integer)
    min ← A[0]                          ▷ Assign the first element as the minimum
    max ← A[0]                          ▷ Assign the first element as the maximum
    for i ← 1 to n − 1 do
        if A[i] < min then
            min ← A[i]
        end if
        if A[i] > max then max ← A[i]
        end if
    end for
end procedure
```

Permasalahan MinMax (3)

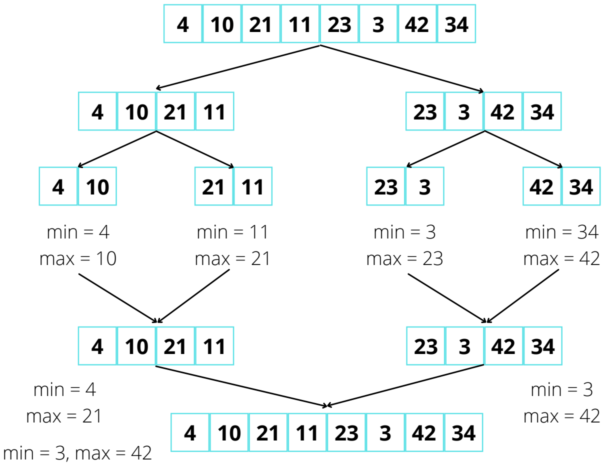
Skema algoritma Minmax dengan metode Divide-and-Conquer



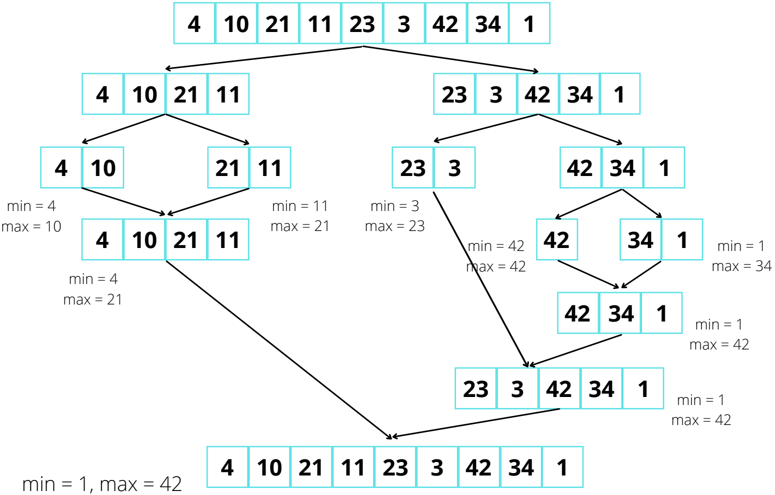
Algorithm 3 MinMax2 (DnC)

```
procedure MinMax2(input: A, i, j)
    if i = j then min ← A[i];  max ← A[i]
    else
        if i = j − 1 then                                   ▷ The array has size 2
            if A[i] < A[j] then min ← A[i];  max ← A[j]
            else min ← A[j];  max ← A[i]
            end if
        else
            k ← (i + j) div 2                               ▷ Divide the array in the middle (position k)
            min_1, max_1 = MinMax2(A, i, k)                 ▷ Suppose it returns min_1, max_1)
            min_2, max_2 = MinMax2(A, k + 1, j)             ▷ Suppose it returns min_2, max_2)
            if min_1 < min_2 then min ← min_1
            else min ← min_2
            end if
            if max_1 < max_2 then max ← max_2
            else max ← max_1
            end if
        end if
    end if
    return min, max
end procedure
```

Permasalahan MinMax (5): Contoh



Permasalahan MinMax (6): Contoh



Permasalahan MinMax (7): Kompleksitas waktu

Misal $T(n)$ menyatakan banyaknya perbandingan (*comparison*)

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 2 \cdot T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

Formula eksplisit:

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + 2 \\ &= 2 \cdot (2 \cdot T(n/4) + 2) + 2 = 4 \cdot T(n/4) + (4 + 2) \\ &= 4 \cdot (2 \cdot T(n/8) + 2) + 4 + 2 = 8 \cdot T(n/8) + (8 + 4 + 2) \\ &\vdots \\ &= 2^{k-1} \cdot 1 + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= n/2 + n - 2 \\ &= 3n/2 - 2 \ \in \mathcal{O}(n) \end{aligned}$$

Permasalahan MinMax (8): Kompleksitas waktu

- Brute force MinMax1: $T(n) = 2n - 2$
- DnC MinMax2: $T(n) = 3n/2 - 2$

$$3n/2 - 2 < 2n - 2 \Leftrightarrow \text{ for } n \geq 2$$

Permasalahan MinMax lebih efisien jika diselesaikan dengan menggunakan algoritma DnC. Namun secara asimptotis, kedua algoritma tidak berbeda jauh.

Bagian 4. Algoritma *sorting* berbasis DnC

Algoritma *sorting* berbasis DnC (1)

Review

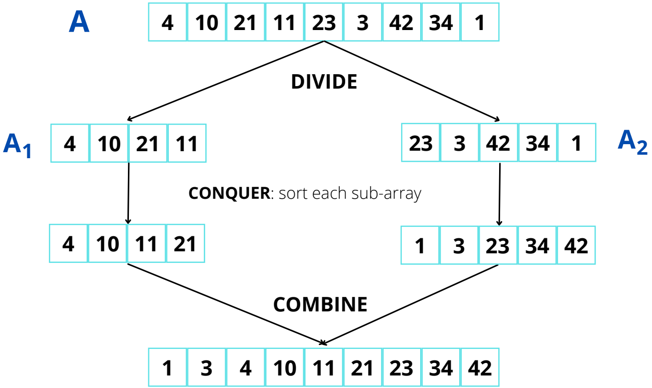
- Masalah *sorting*: Diberikan array $A[0..n-1]$ yang *ordorable* (yang dapat diurutkan). Array A dikatakan **terurut (sorted)** jika elemen dalam A diurutkan dalam urutan ascending atau descending.
- Ingatlah bahwa algoritma pengurutan berbasis brute force seperti *selection sort*, *bubble sort*, dan *insertion sort* memiliki kompleksitas waktu $\mathcal{O}(n^2)$.
- Bisakah kita menghasilkan algoritma pengurutan dengan kompleksitas waktu yang lebih baik menggunakan pendekatan DnC?

Algoritma *sorting* berbasis DnC (2)

Ide dari algoritma sorting berbasis DnC:

- Jika array memiliki ukuran $n = 1$, maka array tersebut **sudah terurut**.
- Jika array memiliki ukuran $n > 1$, maka bagilah array menjadi dua sub-array, lalu urutkan setiap sub-array.
- Gabungkan sub-array yang diurutkan menjadi larik yang diurutkan. Ini adalah hasil dari algoritma.

Algoritma *sorting* berbasis DnC (3): skema



Algorithm 4 Algoritma *sorting* berbasis DnC

procedure DNCSORT($A[0..n-1]$: array, n: integer)
  if size(A) $= 1$ **then**
    return A
  end if
  DIVIDE(A, A_1, A_2) yang masing-masing berukuran n_1 dan n_2. ▷ $n_2 = n - n_1$
  DNCSORT(A_1, n_1) ▷ $A_1 = A[0..n_1 - 1]$
  DNCSORT(A_2, n_2) ▷ $A_2 = A[n_1..n - 1]$
  COMBINE(A_1, A_2, A)
end procedure

- Prosedur untuk DIVIDE and COMBINE bergantung pada jenis permasalahannya.

Algoritma *sorting* berbasis DnC (4)

Dua pendekatan untuk algoritma DnC

1. Easy split/hard join
   - Langkah **Divide** mudah *secara komputasional*
   - Langkah **Combine** sulit *secara komputasional*
   - Contoh: *Merge Sort, Insertion Sort*
2. Hard split/easy join
   - Langkah **Divide** sulit *secara komputasional*
   - Langkah **Combine** mudah *secara komputasional*
   - Contoh: *Quick Sort, Selection Sort*

Algoritma *sorting* berbasis DnC (5)

Contoh

Diberikan array $A = [4, 12, 3, 9, 1, 21, 5, 1]$

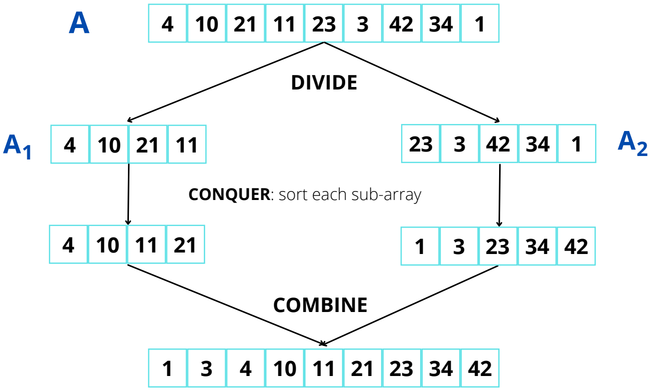
1. **Easy split/hard join:** A dibagi berdasarkan **posisi elemennya**
   - *Divide:* $A_1 = [4, 12, 3, 9]$ and $A_2 = [1, 21, 5, 2]$
   - *Sort:* $A_1 = [3, 4, 9, 12]$ and $A_2 = [1, 2, 5, 21]$
   - *Combine:* $A = [1, 2, 3, 4, 5, 9, 12, 21]$

2. **Hard split/easy join:** A dibagi berdasarkan **nilai elemennya**
   - *Divide:* $A_1 = [4, 2, 3, 1]$ and $A_2 = [9, 21, 5, 12]$
   - *Sort:* $A_1 = [1, 2, 3, 4]$ and $A_2 = [5, 9, 12, 21]$
   - *Combine:* $A = [1, 2, 3, 4, 5, 9, 12, 21]$

Bagian 5. Merge Sort

Merge Sort (1)

Ide dasar:



Merge Sort (2)

Algoritma:

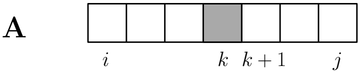
Input: array A, integer n
Output: array A terurut (sorted)

1. If $n = 1$, then A terurut.
2. If $n > 1$, then:
   - **Divide:** pisahkan A menjadi dua bagian, masing-masing berukuran $\lfloor n/2 \rfloor$ dan $\lceil n/2 \rceil$;
   - **Conquer:** secara rekursif, implementasikan $\textsc{MergeSort}$ di setiap sub-array;
   - **Merge:** gabungkan sub-array yang diurutkan ke dalam array A yang sudah diurutkan.

Merge Sort (3)

Algorithm 5 Merge Sort

```
procedure MergeSort(A: ordorable array, i, j: integer)  ▷ i: starting index, j: last index, initialization: i = 0, j = n − 1 (i.e. the whole array A)
    if i = j then                                       ▷ length(A) = 1
        return A[i]
    end if
    k ← (i + j) div 2                                   ▷ Divide the array into two
    MergeSort(A, i, k)                                  ▷ Sort the sub-array A[i..k]
    MergeSort(A, k + 1, j)                              ▷ Sort the sub-array A[k + 1..j]
    Merge(A, i, k, j)                                   ▷ Merge sorted A[i..k] and A[k + 1..j] into the sortedA[i..j]
end procedure
```



Algorithm 6 "Merge" in MERGESORT

procedure MERGE(A, i, k, j) ▷ *$A[i..k]$ and $A[k+1..j]$ are sorted (ascending)*
output: Array $A[i..j]$ sorted (ascending)
declaration
    B : temporary array to store the merged values
end declaration

$p \leftarrow i;\ \ q \leftarrow k+1;\ \ r \leftarrow i$
while $p \leq k$ **and** $q \leq j$ **do** ▷ *while the left-array and the right-array are not finished*
    if $A[p] \leq A[q]$ **then**
        $B[r] \leftarrow A[p]$ ▷ *B is a temporary array to store the merged array; assign $A[p]$ (of left array) to B*
        $p \leftarrow p+1$
    else
        $B[r] \leftarrow A[q]$ ▷ *Assign $A[q]$ (of right array) to B*
        $q \leftarrow q+1$
    end if
    $r \leftarrow r+1$
end while ▷ *At this point, $p > k$ or $q > j$*

```
1:  while p ≤ k do                ▷ If the left-array is not finished, copy the rest of left-array A to B (if any)
2:      B[r] ← A[p]
3:      p ← p + 1
4:      r ← r + 1
5:  end while
6:  while q ≤ j do                ▷ If the right-array is not finished, copy the rest of right-array A to B (if any)
7:      B[r] ← A[q]
8:      q ← q + 1
9:      r ← r + 1
10: end while
11: for r ← i to j do                                      ▷ Assign back all elements of B to A
12:     A[r] ← B[r]
13: end for
14: return A                                                ▷ A is in ascending order
15: end procedure
```

Catatan. penomoran baris kode lanjutan dari slide sebelumnya: 17, 18, 19, ...

Merge Sort (4): Contoh prosedur MERGE

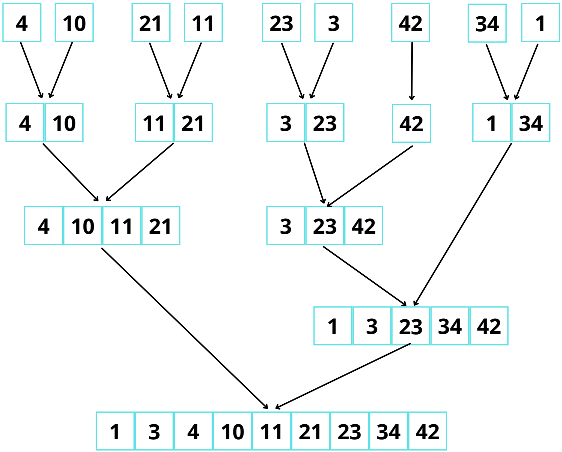


Figure: Contoh prosedur MERGE

Merge Sort (5): Contoh prosedur MERGESORT

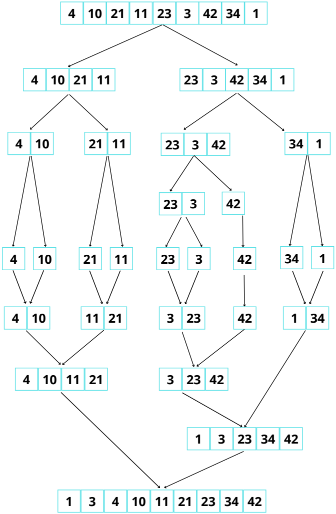


Figure: Example of MERGESORT procedure

Merge Sort (4): Kompleksitas waktu

Menghitung kompleksitas waktu dari Merge Sort mirip dengan menghitung kompleksitas waktu dari algoritma rekursif lainnya.

- Kompleksitas algoritma Merge Sort diukur dari banyaknya perbandingan elemen dalam array yang dinotasikan dengan $T(n)$.
- Banyaknya perbandingan memiliki kompleksitas $\mathcal{O}(n)$, atau cn untuk suatu konstanta c.
  (Dalam hal ini, banyaknya perbandingan tidak dapat dihitung secara presisi, karena prosedur MERGE *melibatkan banyak operasi.)*
- Jadi, $T(n) = 2T(n/2) + cn$, untuk suatu konstanta c
- Dengan demikian:

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + cn, & n > 1 \end{cases}$$

Merge Sort (4): Kompleksitas waktu

- Fungsi eksplisit dapat dihitung dengan mengganti fungsi secara iteratif. Untuk penyederhanaan, mari kita selidiki kasus khusus, yakni ketika $n = 2^k$ untuk suatu bilangan bulat k.

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn) + 3cn \\ &= 4(2T(n/8) + cn) + 3cn \\ &\vdots \\ &= 2^k T(n/2^k) + kcn \end{aligned}$$

Karena $n = 2^k$, maka $k = \log_2 n$. Sehingga:

$$T(n) = n \cdot T(1) + cn \cdot \log_2 n = 0 + cn \cdot \log_2 n \in \mathcal{O}(n \log n)$$

- Hal ini menunjukkan bahwa Merge Sort memiliki kompleksitas yang lebih baik ($\mathcal{O}(n \log n)$) dibandingkan dengan algoritma pengurutan berbasis brute-force ($\mathcal{O}(n^2)$).

Bagian 6. Recursive Insertion Sort

Kasus khusus Merge Sort

Insertion sort (1): Prinsip dasar

- Algoritma ini merupakan easy split/hard join-sorting.
- Kita telah membahas versi iteratif dari algoritma Insertion Sort. Kita juga dapat melihatnya dengan cara rekursif (yang merupakan kasus khusus dari Merge Sort).
- Array dibagi menjadi dua sub-array, di mana sub-array pertama hanya terdiri dari satu elemen, dan sub-array kedua terdiri dari $n-1$ elemen.

Insertion sort (2): Pseudocode

Algorithm 7 Recursive Insertion Sort

```
procedure InsertionSort(A: ordorable array, i, j: integers)
    output: A in ascending order

    if i < j then                                          ▷ size(A) > 1
        k ← i                          ▷ A is split at position i (initialize as i = 0
        InsertionSort(A, i, k)                        ▷ sort the sub-array A[i..k]
        InsertionSort(A, k + 1, j)                ▷ sort the sub-array A[k + 1..j]
        Merge(A, i, k, j)     ▷ merge the sub-array A[i..k] and A[k + 1..j] into A[i..j]
    end if
end procedure
```

Insertion sort (3): Pseudocode

Catatan. Karena sub-array kiri berukuran 1, maka kita dapat menghapus prosedur InsertionSort untuk sub-array kiri.

Algorithm 8 Insertion Sort

```
procedure InsertionSort(A: ordorable array, i, j: integers)
    output: A in ascending order
    initialization: i ← 0, j ← n − 1
    if i < j then                          ▷ size(A) > 1
        k ← i                              ▷ A is split at position i (initialize as i = 0
        InsertionSort(A, k + 1, j)         ▷ sort the sub-array A[k + 1..j]
        Merge(A, i, k, j)                  ▷ merge the sub-array A[i] and A[k + 1..j] into A[i..j]
    end if
end procedure
```

Catatan. Prosedur Merge dapat diganti dengan 'Insertion method' yang digunakan pada versi rekursif.

Insertion sort (4): Contoh

Contoh: Misalkan kita ingin mengurutkan array $A = [4, 10, 21, 11, 23, 3, 42, 34, 1]$.

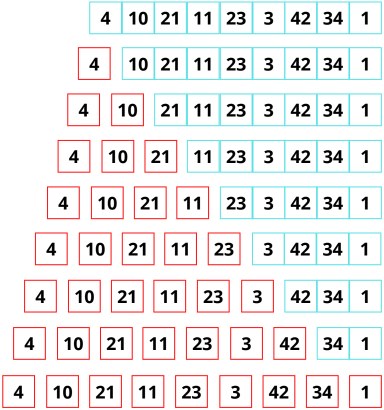


Figure: Tahap 'Divide' and 'Conquer'

Insertion sort (5): Contoh

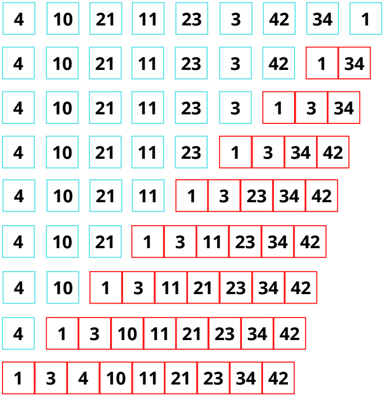


Figure: Penerapan prosedur MERGE

Insertion sort (6): Kompleksitas waktu

Fungsi rekursif untuk menghitung kompleksitas waktu:

$$T(n) = \begin{cases} a, & n = 1 \\ T(n-1) + cn, & n > 1 \end{cases}$$

Rumus eksplisit diperoleh dengan substitusi rekursif:

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= (T(n-2) + c(n-1)) + cn = T(n-2) + (cn + c(n-1)) \\ &= (T(n-3) + c(n-2)) + (cn + c(n-1)) = T(n-3) + \\ &\quad (cn + c(n-1) + c(n-2)) \\ &\vdots \\ &= cn + c(n-1) + c(n-2) + \cdots + 2c + a \\ &= c\left(\frac{1}{2} \cdot (n-1)(n+2)\right) \\ &= \frac{cn^2}{2} + \frac{cn}{2} + (a - c) \\ &= \mathcal{O}(n^2) \quad \text{(sama dengan versi iteratif-nya)} \end{aligned}$$

Quick Sort

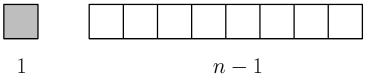
Click here

Bagain 7. Recursive Selection Sort

Kasus khusus dari Quick Sort

Selection sort (1): Prinsip dasar

- Ini adalah hard split/easy join-sorting.
- Kita telah mempelajari versi iteratif dari algoritma Selection Sort. Kita juga dapat melihatnya secara rekursif, sebagai kasus khusus dari Quick Sort.
- Array dibagi menjadi dua sub-array, di mana sub-array pertama hanya terdiri dari satu elemen, dan sub-array kedua terdiri dari $n - 1$ elemen.



Catatan. Metode ini mengikuti versi SELECTIONSORT Levitin (dengan mencari elemen min). Di versi lain (jika kita mencari elemen max), sub-array kanan berukuran satu dan sub-array kiri berukuran $n - 1$.

Selection sort (2): Pseudocode

Catatan. Karena sub-array kiri berukuran 1, maka kita tidak perlu memanggil INSERTIONSORT secara rekursif untuk sub-array *kiri*.

Algorithm 9 Recursive Selection Sort

```
procedure SELECTIONSORT(A: ordorable array, i, j: integers)
    input: array A[i..j]
    output: A[i..j] in ascending order

    initialization: i ← 0, j ← n − 1
    if i < j then                                        ▷ size(A) > 1
        PARTITION(A, i, j)        ▷ Partition the array into sub-arrays of size 1 and n − 1
        SELECTIONSORT(A, i + 1, j)                       ▷ Sort only the right sub-array
    end if
end procedure
```

Selection sort (3): Pseudocode

Catatan. Karena sub-array kiri berukuran 1, maka kita tidak perlu memanggil INSERTIONSORT secara rekursif untuk sub-array *kiri*.

Algorithm 10 Partition procedure

```
procedure PARTITION(A: ordorable array, i, j: integers)  ▷ Partition A[i..j] by looking for the minimum element and assign it to A[i]
    idxMin ← i
    for k ← i + 1 do to j
        if A[k] < A[idxMin] then
            idxMin ← k
        end if
    end for
    SWAP(A[i], A[idxMin])  ▷ Exchange A[i] and A[idxMin]
end procedure
```

Selection sort (4): Contoh

Misalkan kita ingin mengurutkan array: $A = [4, 10, 21, 11, 23, 3, 42, 34, 1]$

4	10	21	11	23	3	42	34	1
1	10	21	11	23	3	42	34	4
1	3	21	11	23	10	42	34	4
1	3	4	11	23	10	42	34	21
1	3	4	10	23	11	42	34	21
1	3	4	10	11	23	42	34	21
1	3	4	10	11	21	42	34	23
1	3	4	10	11	21	23	34	42
1	3	4	10	11	21	23	34	42
1	3	4	10	11	21	23	34	42
1	3	4	10	11	21	23	34	42

- X Unsorted
- X Sorted
- X Current left sub-array

Selection sort (4): Kompleksitas waktu

Fungsi rekursif dari kompleksitas waktunya:

$$T(n) = \begin{cases} a, & n = 1 \\ T(n-1) + cn, & n > 1 \end{cases}$$

Formula eksplisit diperoleh dengan metode substitusi (*sperti pada Insertion Sort*):

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= (T(n-2) + c(n-1)) + cn = T(n-2) + (cn + c(n-1)) \\ &= (T(n-3) + c(n-2)) + (cn + c(n-1)) = T(n-3) + \\ &\quad (cn + c(n-1) + c(n-2)) \\ &\vdots \\ &= cn + c(n-1) + c(n-2) + \cdots + 2c + a \\ &= c\left(\frac{1}{2} \cdot (n-1)(n+2)\right) \\ &= \frac{cn^2}{2} + \frac{cn}{2} + (a - c) \\ &= \mathcal{O}(n^2) \quad \text{(sama dengan versi iteratif-nya)} \end{aligned}$$

Kesimpulan

Apa yang dapat kita simpulkan dari keempat algoritma sorting tersebut?

Memisahkan array menjadi dua **balanced** array (masing-masing berukuran $n/2$) akan menghasilkan kinerja algoritma terbaik (dalam kasus Merge Sort dan Quick Sort, yaitu $\mathcal{O}(n \log n)$).

Sementara pemisahan yang tidak seimbang (**unbalanced**) (menjadi 1 elemen dan $n - 1$ elemen) menghasilkan kinerja algoritma yang buruk (dalam kasus Insertion sort dan Selection sort, yaitu $\mathcal{O}(n^2)$).

to be continued...