Algoritma *Divide and Conquer*

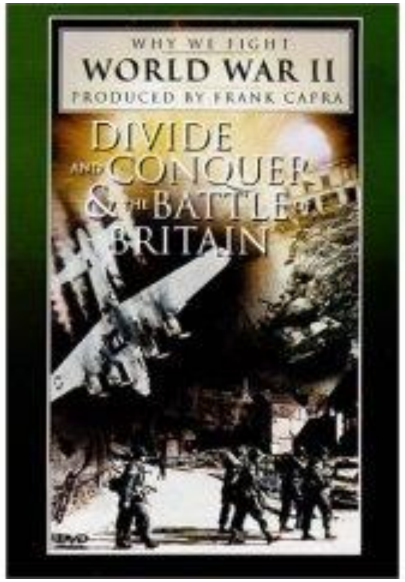
(Bagian 1)

Bahan Kuliah IF2211 Strategi Algoritma

Oleh: Rinaldi Munir

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika ITB
2024

- *Divide and Conquer* dulunya adalah strategi militer yang dikenal dengan nama *divide ut imperes.*



- Sekarang strategi tersebut menjadi strategi fundamental di dalam ilmu komputer dengan nama *Divide and Conquer*.

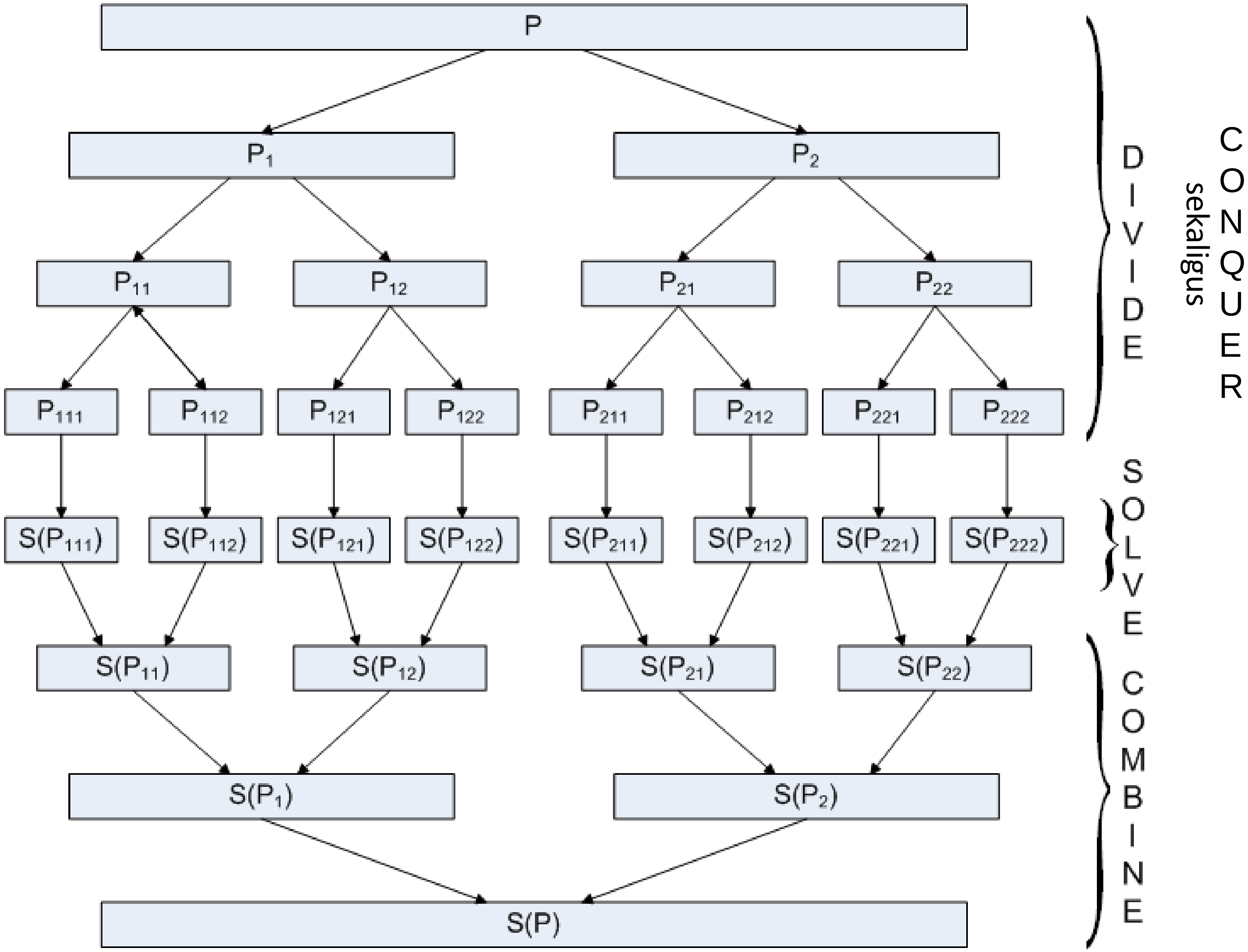

DIVIDE
ET IMPERA

Definisi Divide and Conquer

- *Divide*: membagi persoalan menjadi beberapa upa-persoalan yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya setiap upa-persoalan berukuran hampir sama),

- *Conquer*: menyelesaikan (*solve*) masing-masing upa-persoalan (secara langsung jika sudah berukuran kecil atau secara rekursif jika masih berukuran besar).

- *Combine:* menggabungkan solusi masing-masing upa-persoalan sehingga membentuk solusi persoalan semula.



Keterangan:

P = persoalan

S = solusi

- Obyek persoalan yang dibagi : masukan (*input*) atau *instances* persoalan yang berukuran *n* seperti:

  - tabel (larik),

  - matriks,

  - eksponen,

  - polinom,

  - dll, bergantung persoalannya.

- Tiap-tiap upa-persoalan memiliki karakteristik yang sama (*the same type*) dengan karakteristik persoalan semula namun berukuran lebih kecil

- sehingga metode *Divide and Conquer* lebih natural diungkapkan dalam skema rekursif.

Skema Umum Algoritma *Divide and Conquer*

```
procedure DIVIDEandCONQUER(input P : problem, n : integer)
{ Menyelesaikan persoalan P dengan algoritma divide and conquer
  Masukan: masukan persoalan P berukuran n
  Luaran: solusi dari persoalan semula    }
Deklarasi
     r : integer

Algoritma
   if  n ≤ n0 then  {ukuran persoalan P sudah cukup kecil }
        SOLVE  persoalan P yang berukuran n ini
  else
       DIVIDE  menjadi r upa-persoalan, P1, P2, …, Pr , yang masing-masing berukuran n1, n2, …, nr
       for  masing-masing P1, P2, …, Pr , do
             DIVIDEandCONQUER(Pi,  ni)
       endfor
       COMBINE  solusi dari P1, P2, …, Pr  menjadi solusi persoalan semula
  endif
```

Kompleksitas algoritma *divide and conquer*:
$$T(n) = \begin{cases} g(n) & , n \le n_0 \\ T(n_1) + T(n_2) \ldots + T(n_r) + f(n) & , n > n_0 \end{cases}$$

Penjelasan:

$$T(n) = \begin{cases} g(n) & , n \le \mathrm{n}_0 \\ T(n_1) + T(n_2) \ldots + T(n_r) + f(n) & , n > n_0 \end{cases}$$

- *T*(*n*) : kompleksitas waktu penyelesaian persoalan *P* yang berukuran *n*
- *g*(*n*) : kompleksitas waktu untuk SOLVE jika *n* sudah berukuran kecil
- $T(n_1) + T(n_2) \ldots + T(n_r)$: kompleksitas waktu untuk memproses setiap upa-persoalan
- *f*(*n*) : kompleksitas waktu untuk COMBINE solusi dari masing-masing upa-persoalan
- Tahap DIVIDE dapat dilakukan dalam O(1), sehingga tidak dimasukkan ke dalam formula

Jika pembagian selalu menghasilkan **dua** upa-persoalan yang berukuran sama:

```
procedure DIVIDEandCONQUER(input P : problem, n : integer)
{ Menyelesaikan persoalan dengan algoritma divide and conquer
  Masukan: masukan yang berukuran n
  Luaran: solusi dari persoalan semula
}
Deklarasi
    r : integer

Algoritma
  if  n ≤ n0 then  {ukuran persoalan sudah cukup kecil }
       SOLVE  persoalan P yang berukuran n ini
  else
      DIVIDE  menjadi 2 upa-persoalan, P1 dan P2, masing-masing berukuran n/2
      DIVIDEandCONQUER(P1, n/2)
      DIVIDEandCONQUER(P2, n/2)
      COMBINE solusi dari P1 dan P2
  endif
```

Kompleksitas algoritma *divide and conquer*: $T(n) = \begin{cases} g(n) & , n \le \mathrm{n}_0 \\ 2T(n/2) + f(n) & , n > n_0 \end{cases}$

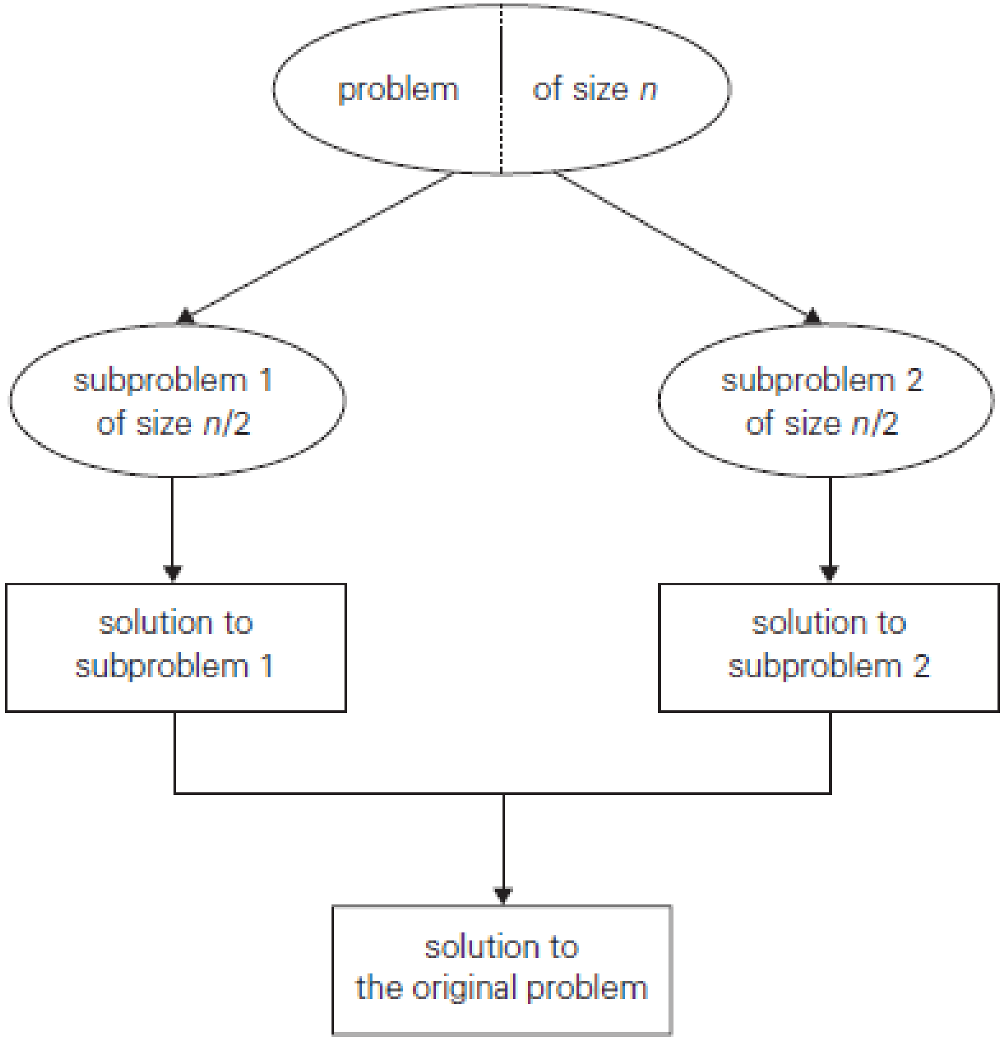


FIGURE 5.1 Divide-and-conquer technique (typical case).

Sumber: Levitin

Beberapa persoalan klasik yang diselesaikan dengan D&C

1. Persoalan MinMaks (mencari nilai minimum dan nilai maksimum)
2. Menghitung perpangkatan
3. Persoalan pengurutan (*sorting*) – *Mergesort* dan *Quicksort*
4. Mencari sepasang titik terdekat (*closest pair problem*)
5. *Convex Hull*
6. Perkalian matriks
7. Perkalian bilangan bulat besar
8. Perkalian dua buah polinom
9. *Skyline* problem
10. Dll

1. Persoalan *MinMaks*: Mencari Nilai Minimum dan Maksimum

Persoalan: Misalkan diberikan sebuah larik *A* yang berukuran *n* elemen dan sudah berisi nilai *integer*.

Carilah nilai minimum (min) dan nilai maksimum (max) sekaligus di dalam larik tersebut.

Contoh:

4	12	23	9	21	1	35	2	24

min = 1
max = 35

Penyelesaian dengan *algoritma brute force*

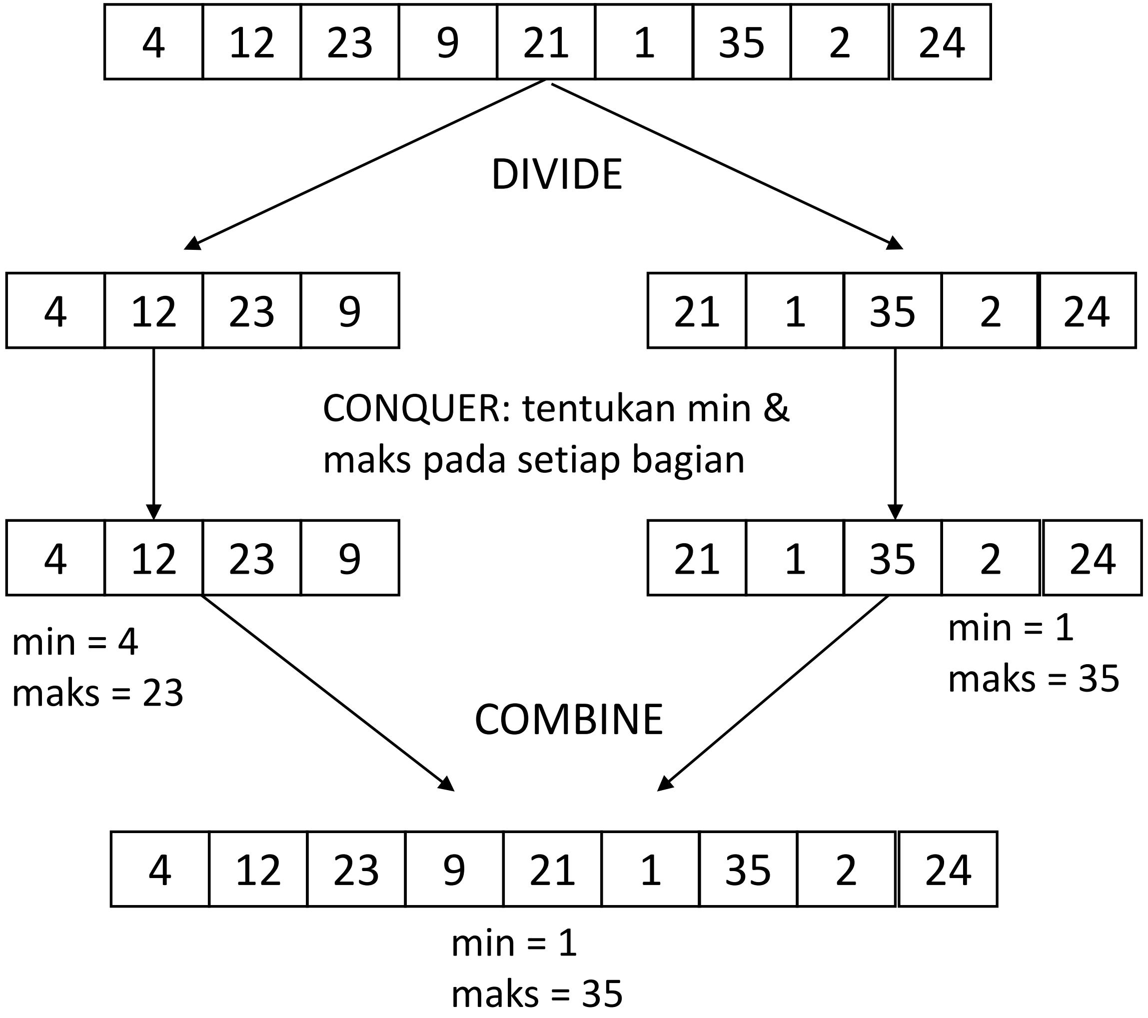
```
procedure MinMaks1(input A : Larik, n : integer,  output min, maks : integer)
{ Mencari nilai minimum dan maksimum di dalam larik a yang berukuran n elemen, secara brute force.
Masukan: larik a yang sudah terdefinisi elemen-elemennya
Luaran: nilai maksimum dan nilai minimum tabel
}
Deklarasi
   i : integer

Algoritma:
   min ← A[1]         { asumsikan elemen pertama sebagai nilai  minimum sementara}
   maks ← A[1]        {asumsikan elemen pertama sebagai nilai  maksimum sementara}
   for  i ← 2  to  n  do
      if  A[i] < min  then
         min ← A[i]
      endif
      if  A[i] > maks  then
        maks ← A[i]
      endif
   endfor
```

Jumlah perbandingan elemen larik: $T(n) = (n - 1) + (n - 1) = 2n - 2 = O(n)$

Penyelesaian dengan *algoritma divide and conquer*

Ide dasar secara *divide and conquer*:



- Ukuran larik hasil pembagian dapat dibuat cukup kecil sehingga mencari minimum dan maksimum dapat diselesaikan (SOLVE) secara trivial.

- Dalam hal ini, ukuran “kecil” didefinisikan apabila larik hanya berukuran 1 elemen atau 2 elemen.

Prosedur MinMaks2(A, n, min, maks)

Algoritma:

1. Untuk kasus n = 1 atau n = 2,

   SOLVE: Jika n = 1, maka *min* = *maks* = *A*[*n*]

   Jika n = 2, maka bandingkan kedua elemen untuk menentukan *min* dan *maks*

2. Untuk kasus n > 2,

   (a) DIVIDE: Bagi dua larik *A* menjadi dua bagian yang sama, A1 dan A2, masing2 n/2 elemen

   (b) CONQUER:

   MinMaks2(A1, n/2, min1, maks1)

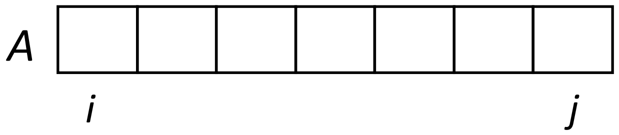
   MinMaks2(A2, n/2, min2, maks2)

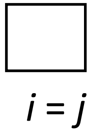
   (c) COMBINE:

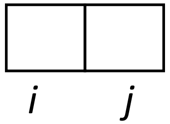
   if min1 < min2 then min $\leftarrow$ min1 else min $\leftarrow$ min2

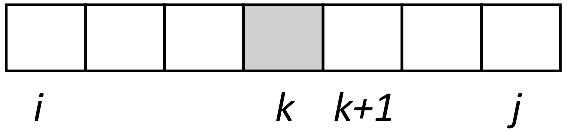
   if maks1 < maks2 then maks $\leftarrow$ maks2 else maks $\leftarrow$ maks1

```
procedure MinMaks2(input A : LarikInteger, i, j : integer,  output min, maks : integer)
{ Mencari nilai maksimum dan minimum di dalam larik A yang berukuran n elemen dengan algoritma divide and Conquer.
   Masukan: larik A yang sudah terdefinisi elemen-elemennya
   Luaran: nilai maksimum dan nilai minimum larik  }
Deklarasi
     min1, min2, maks1, maks2 : integer
 Algoritma:
     if  i = j  then                              { larik berukuran 1 elemen  }
         min ← A[i];  maks ← A[i]
     else
         if  (i = j – 1) then                     { larik berukuran 2 elemen  }
            if  A[i] < A[j]  then
                 min ← A[i];  maks ← A[j]
            else
                min ← A[j];  maks ← A[i]
            endif
         else                    { larik berukuran lebih dari 2 elemen }
             k ← (i + j) div 2            { bagidua larik pada posisi k }
            MinMaks2(A, i, k, min1, maks1)
            MinMaks2(A, k + 1, j, min2, maks2)
            if  min1 < min2  then  min ← min1  else  min ← min2 endif
            if  maks1 < maks2  then  maks ← maks2  else maks ← maks1  endif
        endif
     endif
```



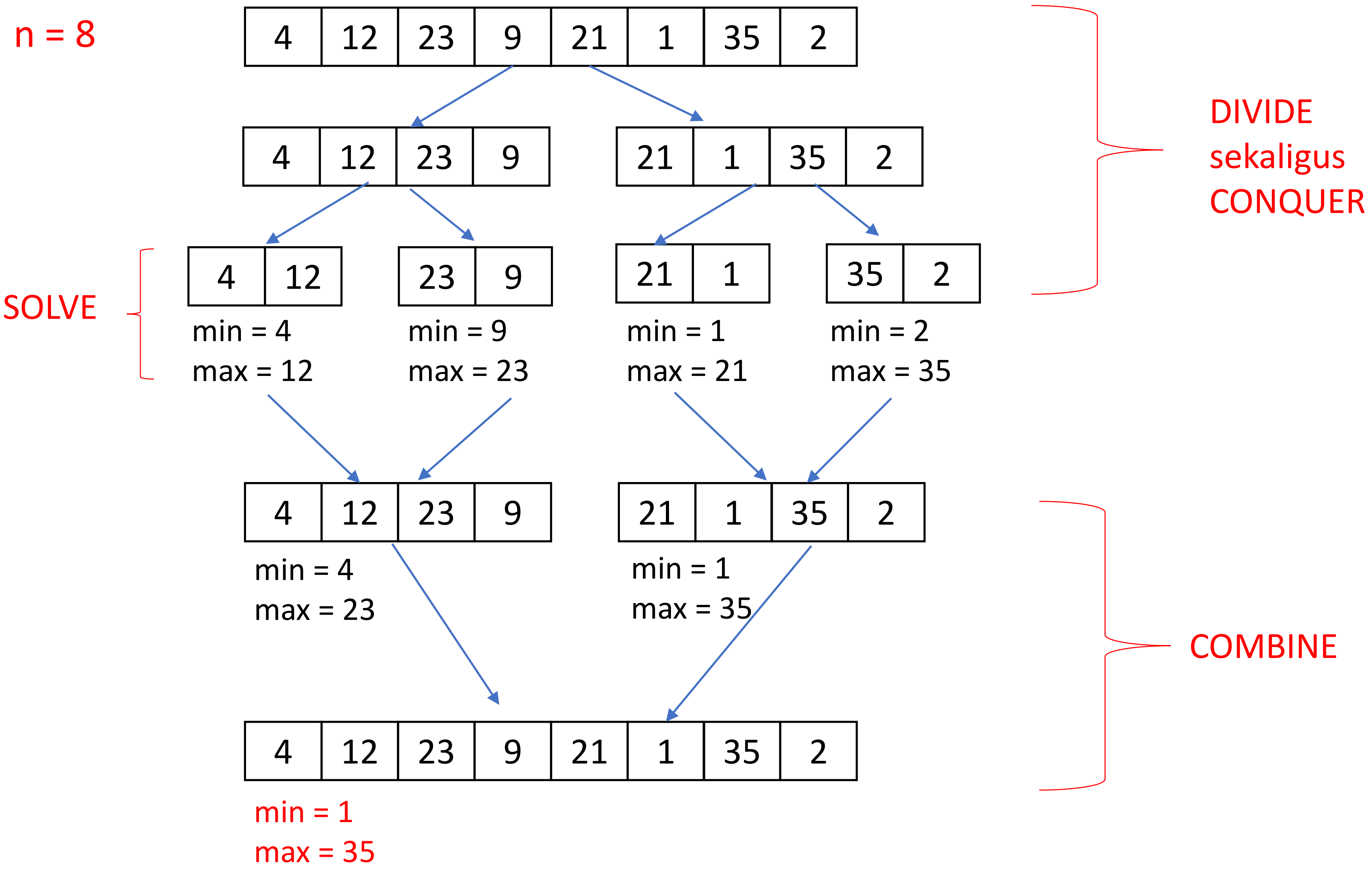






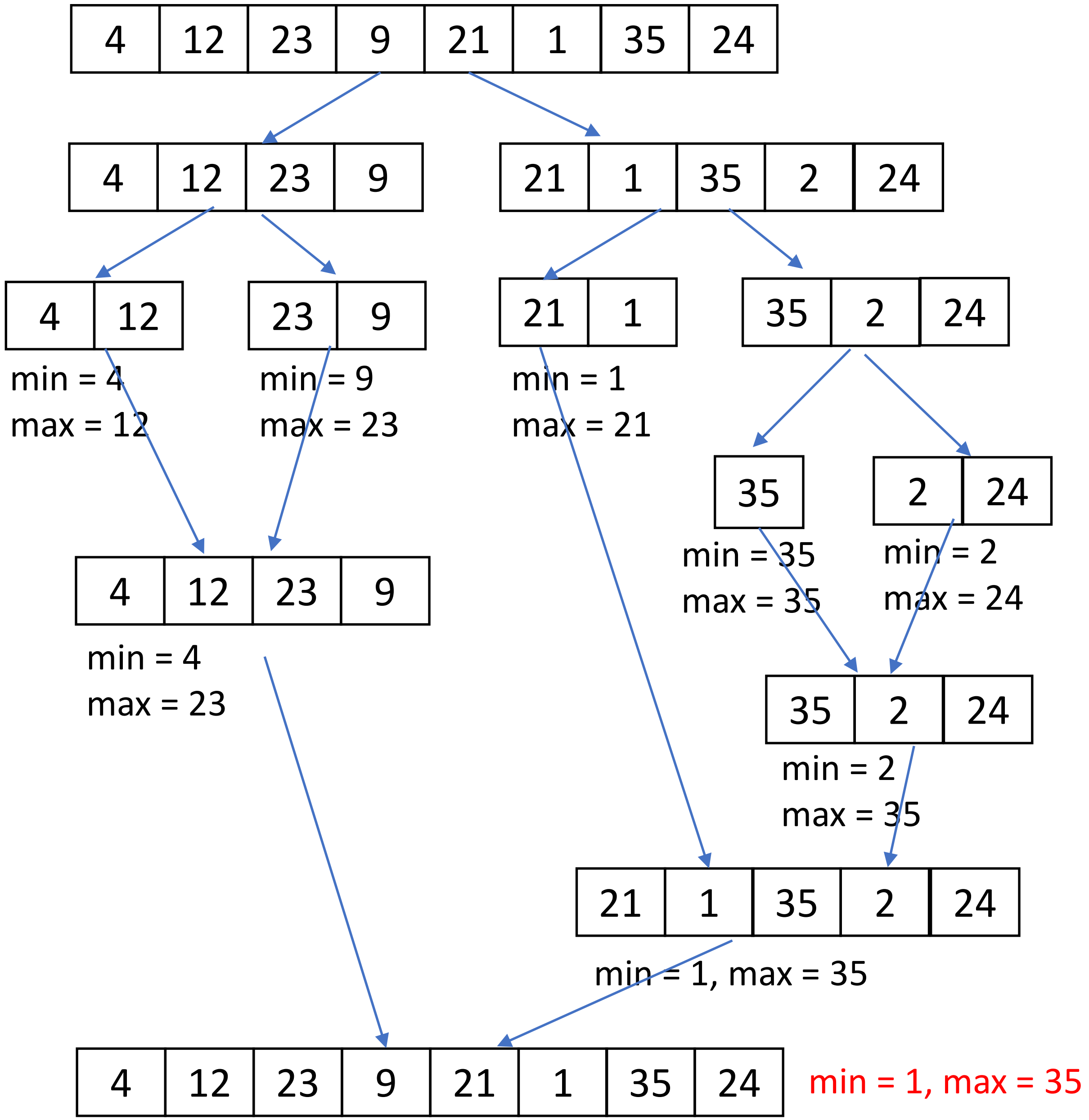
Pemanggilan pertama kali: *MinMaks2(A*, 1, *n*, *min*, *maks*)

Contoh 1: Mencari nilai minimum dan maksimum di dalam larik berikut (kasus n =2^k)



Contoh 2: Mencari nilai minimum dan maksimum di dalam larik berikut (kasus $n \neq 2^k$)

n = 9



Kompleksitas waktu algoritma *MinMaks2*, dihitung dari jumlah operasi perbandingan elemen-elemen larik:

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

Penyelesaian:

Asumsi: $n = 2^k$, dengan k bilangan bulat positif, maka

$T(n) = 2T(n/2) + 2$

$= 2(2T(n/4) + 2) + 2 = 4T(n/4) + 4 + 2$

$= 4T(2T(n/8) + 2) + 4 + 2 = 8T(n/8) + 8 + 4 + 2$

$= ...$

$= 2^{k-1}\, T(2) + \sum_{i=1}^{k-1} 2^i$

$= 2^{k-1} \cdot 1 + 2^k - 2$

$= n/2 + n - 2$

$= 3n/2 - 2$

$= O(n)$

Bandingkan:

- *MinMaks1* secara *brute force* : $T(n) = 2n - 2$
- *MinMaks2* secara *divide and conquer*: $T(n) = 3n/2 - 2$

Perhatikan bahwa $3n/2 - 2 < 2n - 2$ untuk $n \geq 2$.

- Kesimpulan: persoalan *MinMaks* lebih sangkil diselesaikan dengan algoritma *Divide and Conquer*.

- Moral dari contoh ini adalah bahwa algoritma *divide and conquer* dapat membantu kita menghasilkan algoritma yang sangkil.

2. Perpangkatan a^n

- Misalkan $a \in R$ dan n adalah bilangan bulat tidak negatif, maka perpangkatan a^n didefinisikan sebagai berikut:

$$a^n = \begin{cases} 1, & n = 0 \\ a \times a \times \cdots \times a, & n > 0 \end{cases}$$

Bagaimana algoritma menghitung perpangkatan a^n secara *brute force* dan secara *divide and conquer*?

Penyelesaian dengan *algoritma brute force*

```
function  Exp1(a : real, n : integer)→ real
{ Menghitung a^n, a > 0 dan n bilangan bulat tak-negatif   }

Deklarasi
    k  : integer
    hasil : real

Algoritma:
   hasil ← 1
   for  k ← 1 to  n do
        hasil ← hasil * a
  endfor

  return  hasil
```

Kompleksitas algoritma, dihitung dari jumlah operasi perkalian: $T(n) = n = O(n)$

Penyelesaian dengan algoritma *Divide and Conquer*

Ide dasar: bagi dua pangkat n menjadi $n = n/2 + n/2$

$$a^n = a^{(n/2 + n/2)} = a^{n/2} \times a^{n/2}$$

Contoh: $25^{12} = 25^6 \times 25^6$

Algoritma *divide and conquer* untuk menghitung a^n:

1. Untuk kasus $n = 0$, maka $a^n = 1$.

2. Untuk kasus $n > 0$, bedakan menjadi dua kasus lagi:

   (i) jika n genap, maka $a^n = a^{n/2} \cdot a^{n/2}$

   (ii) jika n ganjil, maka $a^n = a^{n/2} \cdot a^{n/2} \cdot a$

Contoh 3. Menghitung 3^{16} dengan metode *Divide and Conquer*:

$$
\begin{aligned}
3^{16} = 3^8 \cdot 3^8 &= (3^8)^2 \\
&= ((3^4)^2)^2 \\
&= (((3^2)^2)^2)^2 \\
&= ((((3^1)^2)^2)^2)^2 \\
&= ((((3^0)^2 \cdot 3)^2)^2)^2)^2 \\
&= ((((1)^2 \cdot 3)^2)^2)^2)^2 \\
&= ((((3)^2)^2)^2)^2 \\
&= (((9)^2)^2)^2 \\
&= (81)\ ^2)^2 \\
&= (6561)^2 \\
&= 43046721
\end{aligned}
$$

$\rightarrow$ Hanya membutuhkan enam operasi perkalian
(operasi perpangkatan dua = perkalian)

Pseudo-code menghitung a^n dengan *divide and conquer*:

```
function Exp2(a : real, n : integer) → real
{ mengembalikan nilai a^n, dihitung dengan metode Divide and Conquer }

Algoritma:
  if  n = 0  then
    return  1
  else
    if  odd(n)  then    { kasus n ganjil }
         return  Exp2(a, n div 2) * Exp2(a, n div 2) * a       {a^n = a^(n/2) · a^(n/2) · a}
    else                  { kasus  n genap }
         return  Exp2(a, n div 2) * Exp2(a, n div 2)           {a^n = a^(n/2) · a^(n/2) }
    endif
 endif
```

Fungsi *Exp2* tidak sangkil, sebab terdapat dua kali pemanggilan rekursif untuk nilai parameter yang sama → *Exp2*(*a*, *n* **div** 2) * *Exp2*(*a*, *n* **div** 2)

Perbaikan: simpan hasil *Exp2*(*a*, *n* **div** 2) di dalam sebuah peubah (misalkan *x*), lalu gunakan *x* untuk menghitung a^n pada kasus *n* genap dan *n* ganjil.

```
function  Exp3(a : real, n : integer) → real
{ mengembalikan nilai a^n, dihitung dengan metode Divide and Conquer }

Deklarasi
    x : real
Algoritma:
   if  n = 0  then
      return  1
   else
      x ← Exp3(a, n div 2)
      if  odd(n)  then    { kasus n ganjil }
            return  x * x * a
      else                 { kasus  n genap }
            return  x * x
      endif
  endif
```

- Kompleksitas algoritma *Exp3* dihitung dari jumlah operasi perkalian:

$$T(n) = \begin{cases} 0, & n = 0 \\ T\left(\frac{n}{2}\right) + 2, & n > 0 \text{ dan } n \text{ ganjil} \qquad x * x * a \\ T\left(\frac{n}{2}\right) + 1, & n > 0 \text{ dan } n \text{ genap} \qquad x * x \end{cases}$$

- Dalam menghitung $T(n)$ ini ada sedikit kesulitan, yaitu nilai n mungkin ganjil atau genap, sehingga penyelesaian relasi rekurens menjadi lebih rumit.

- Namun, perbedaan ini dianggap cukup kecil sehingga dapat kita abaikan. Sebagai implikasinya, kita membuat asumsi penghampiran bahwa untuk n genap atau ganjil, jumlah operasi perkalian relatif sama.

- Sehingga, kompleksitas algoritma *Exp3* menjadi:

$$T(n) = \begin{cases} 0, & n = 0 \\ T\left(\frac{n}{2}\right) + 1, & n > 0 \end{cases}$$

- Asumsikan n adalah perpangkatan dari 2, atau $n = 2^k$, maka

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ &= 1 + (1 + T(n/4) = 2 + T(n/4) \\ &= 2 + (1 + T(n/8) = 3 + T(n/8) \\ &= \ldots \\ &= k + T(n/2^k) \end{aligned}$$

Karena $n = 2^k$ maka $k = {}^2\log n$, sehingga

$$\begin{aligned} &= k + T(n/2^k) = {}^2\log n + T(1) \\ &= {}^2\log n + (1 + T(0)) = {}^2\log n + 1 + 0 \\ &= {}^2\log n + 1 = O({}^2\log n) \end{aligned}$$

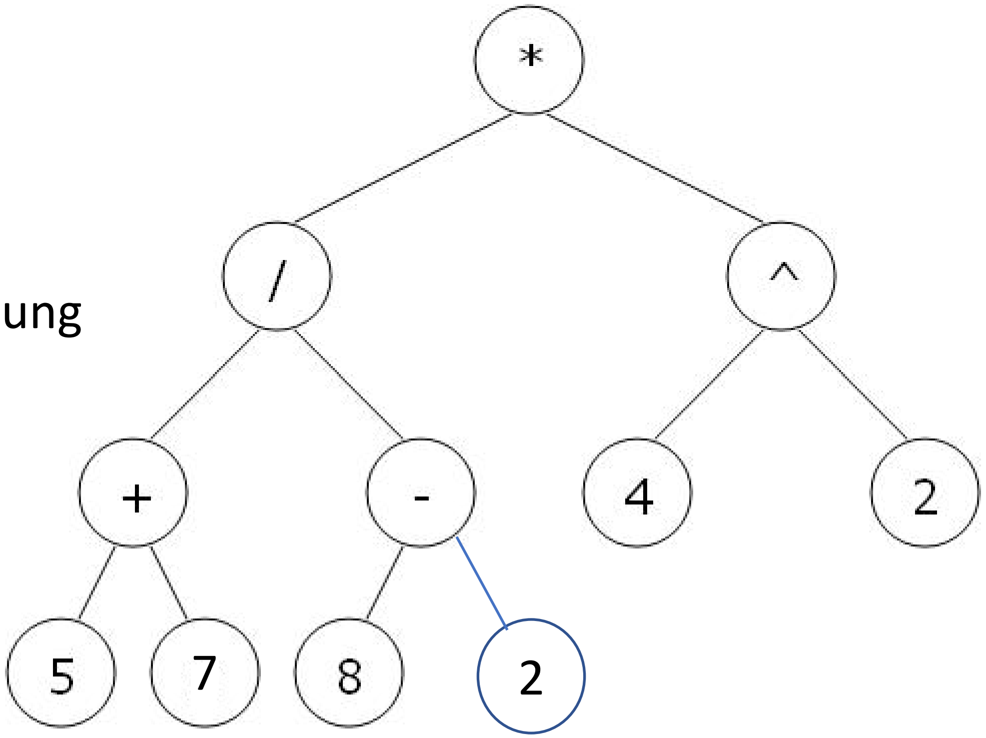
$\rightarrow$ lebih baik daripada algoritma *brute force*!

3. Mengevaluasi Pohon Ekspresi

- Di dalam *compiler* bahasa pemrograman, ekspresi aritmetika direpresentasikan dalam pohon biner yaitu pohon ekspresi (*expression tree*)

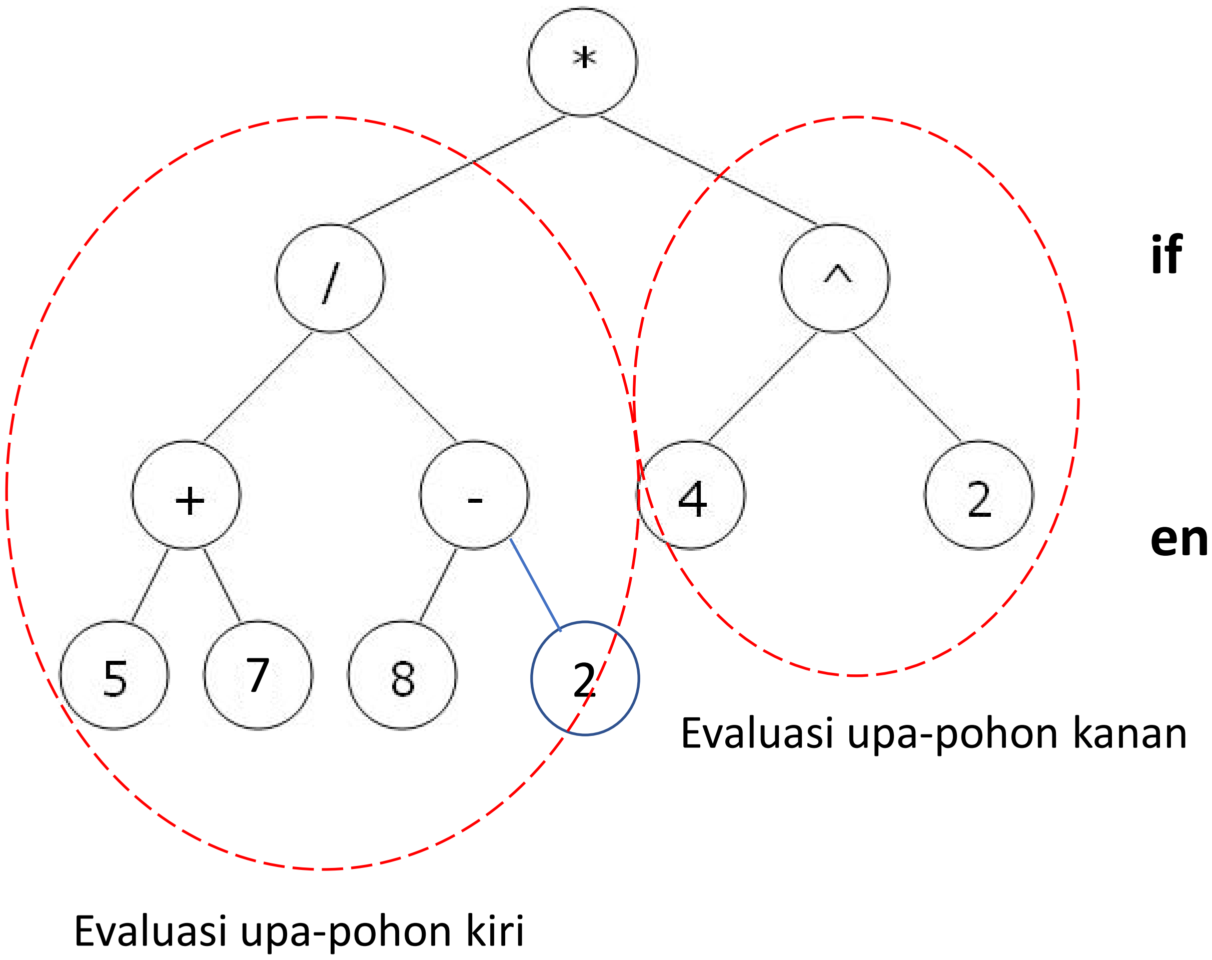
Contoh: (5 + 7) / (8 - 2) * (4 ^ 2)

- Mengevaluasi pohon ekspresi artinya menghitung nilai ekspresi aritmetika yang dinyatakannya.
  Contoh: (5 + 7) / (8 – 2) * (4 ^ 2) = 32



Pohon ekspresi

- Algoritma *divide and conquer*:



if pohon tidak kosong **then**
    nilai1 ← Evaluasi(upa-pohon kiri)
    nilai2 ← Evaluasi(upa-pohon kanan)
    Gabungkan nilai1 dan nilai2 dengan operatornya
end

- Misalkan pohon ekspresi direpresentasikan dengan senarai berkait (*linked list*).

  Simpul daun → *operand*, contoh: 4, –2, 0, dst

  Simpul dalam → operator, contoh: +, -, *, /, ^

  Struktur setiap simpul:

left	info	right

  info: *operand* atau operator

  Pada simpul daun → left = NIL dan right = NIL

- Algoritma *divide and conquer*:

if simpul adalah daun **then**
    return info
else
    secara rekursif evaluasi upa-pohon kiri dan return nilainya
    secara rekursif evaluasi upa-pohon kanan dan return nilainya
    gabungkan kedua nilai tersebut sesuai dengan operator dan return nilainya
endif

- Algoritma evaluasi pohon ekspresi dalam bentuk prosedur:

```
procedure EvaluasiPohon(input T : Pohon, output nilai : real)
{ Mengevaluasi  pohon ekspresi T
   Masukan: Pohon ekspresi T, asumsik T tidak kosong
   Luaran: nilai berisi hasil evaluasi ekspresi
 }
Deklarasi
      nilai1, nilai2 : real
Algoritma:
    if   left(T) = NIL and right(T) = NIL  { simpul daun}
         nilai ← info(T)
     else   { simpul dalam }
           EvaluasiPohon(left(T), nilai1);
           EvaluasiPohon(right(T), nilai2);
          case info(T) of
              "+"  : nilai ← nilai1 + nilai2
               "-"  : nilai ← nilai1 – nilai2
               "*" : nilai ← nilai1 * nilai2
                "/" : nilai ← nilai1  / nilai2         {dengan syarat nilai2 ≠ 0 }
               "^" : nilai ← nilai1  ^  nilai2       {dengan syarat nilai1 ≠ 0  dan nilai2 ≠ 0  }
             end
 end
```

- Algoritma evaluasi pohon ekspresi dalam bentuk fungsi:

```
function EvaluasiPohon(T : Pohon) → real
{ mengevaluasi  pohon ekspresi T }
Deklarasi
      nilai1, nilai2 : real
Algoritma:
    if   left(T) = NIL and right(T) = NIL  { simpul daun}
          return  info(T)
     else   { simpul dalam }
         case info(T) of
                  “+” :  return EvaluasiPohon(left(T), nilai1) + EvaluasiPohon(right(T), nilai2);
                  “-” :  return EvaluasiPohon(left(T), nilai1) –  EvaluasiPohon(right(T), nilai2);
                  “*” :  return EvaluasiPohon(left(T), nilai1) * EvaluasiPohon(right(T), nilai2);
                  “/” :  return EvaluasiPohon(left(T), nilai1) / EvaluasiPohon(right(T), nilai2);     {nilai2 ≠ 0 }
                  “^” :  return EvaluasiPohon(left(T), nilai1) ^ EvaluasiPohon(right(T), nilai2);   {nilai1 ≠ 0 dan nilai2 ≠ 0 }
          end
 end
```

BERSAMBUNG