05 - Algoritma Rekursif

[KOMS120403]

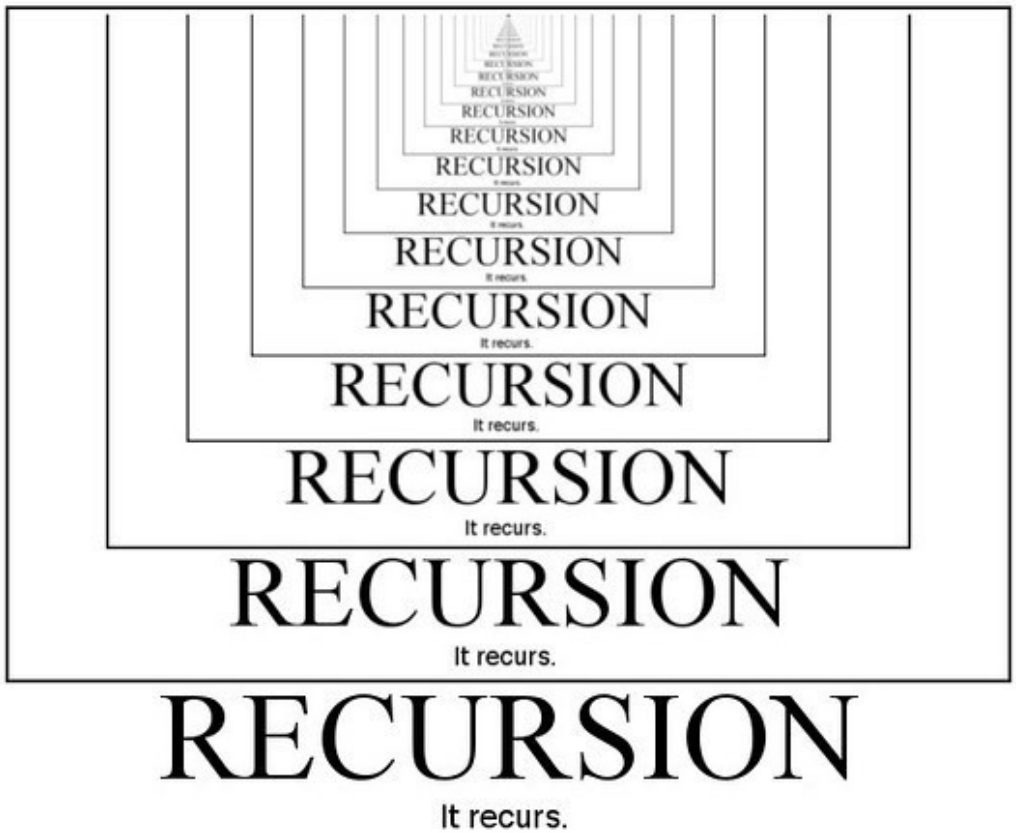
Desain dan Analisis Algoritma (2022/2023)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 5 (March 2023)

Daftar isi



Apa itu recursion atau recursive algorithm?

Bagian 1. Prinsip algoritma rekursif

Prinsip algoritma rekursif

Algoritma rekursif adalah sebuah algoritma yang menyebut dirinya sendiri dengan nilai input "lebih kecil (atau lebih sederhana)", dan yang memperoleh hasil untuk input saat ini dengan menerapkan operasi sederhana ke nilai yang dikembalikan untuk input yang lebih kecil (atau lebih sederhana).

Karakteristik algoritma rekursif:

1. Algoritma memanggil dirinya secara rekursif
2. Algoritma memiliki kasus dasar (*base case*)
3. Algoritma mengubah *state*-nya dan bergerak menuju *base-case*.

Base case adalah kondisi yang memungkinkan algoritma berhenti berulang: kasus dasar biasanya merupakan masalah yang cukup kecil untuk diselesaikan secara langsung.

Perubahan *state* berarti bahwa beberapa data yang digunakan algoritma diubah. Biasanya data yang mewakili masalah kita menjadi lebih kecil.

Rekursif dan Iterasi

Iterasi: Suatu fungsi yang mengulangi proses yang ditentukan sampai terdapat *stopping condition*. Ini biasanya dilakukan melalui perulangan, seperti perulangan for atau while dengan *counter* dan pernyataan komparatif yang membentuk kondisi yang akan gagal. Perulangan iterasi secara *infinite* (tak terbatas) terjadi ketika *stopping condition* tidak pernah terpenuhi.

Rekursi: Alih-alih menjalankan proses tertentu di dalam fungsi, fungsi tersebut memanggil dirinya berulang kali hingga kondisi tertentu terpenuhi (kondisi ini menjadi kasus dasar). Kasus dasar secara eksplisit dinyatakan untuk mengembalikan nilai tertentu ketika kondisi tertentu terpenuhi. Loop rekursif tak terbatas terjadi ketika fungsi tidak mengurangi inputnya dengan cara yang akan menyatu pada kasus dasar.

Bagian 2. Contoh sederhana algoritma rekursif

2.1. Komputasi faktorial

2.1 - Komputasi faktorial (1): Pernyataan masalah

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Rumusnya dapat dinyatakan secara rekursif:

$$n! = \begin{cases} n \times (n-1)!, & \text{if } n > 1 \\ 1, & n = 1 \end{cases}$$

2.1 - Komputasi faktorial (2): Pseudocode

Algorithm 1 Factorial of a number

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         return 1
4:     else
5:         temp = FACTORIAL(n − 1)
6:         return n ∗ temp
7:     end if
8: end procedure
```

- Apakah kasus dasar?

2.1 - Komputasi faktorial (2): Pseudocode

Algorithm 2 Factorial of a number

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         return 1
4:     else
5:         temp = FACTORIAL(n − 1)
6:         return n ∗ temp
7:     end if
8: end procedure
```

- Apakah kasus dasar? $n = 1$
- Deskripsikan *change-of-state*-nya!

2.1 - Komputasi faktorial (2): Pseudocode

Algorithm 3 Factorial of a number

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         return 1
4:     else
5:         temp = FACTORIAL(n − 1)
6:         return n ∗ temp
7:     end if
8: end procedure
```

- Apakah kasus dasar? $n = 1$
- Deskripsikan *change-of-state*-nya! n menurun
- Berapakah kompleksitasnya?

2.1 - Komputasi faktorial (2): Pseudocode

Algorithm 4 Factorial of a number

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         return 1
4:     else
5:         temp = FACTORIAL(n − 1)
6:         return n ∗ temp
7:     end if
8: end procedure
```

- Apakah kasus dasar? $n = 1$
- Deskripsikan *change-of-state*-nya! n menurun
- Berapakah kompleksitasnya? $\mathcal{O}(n)$

2.1 - Komputasi faktorial (3): Diagram

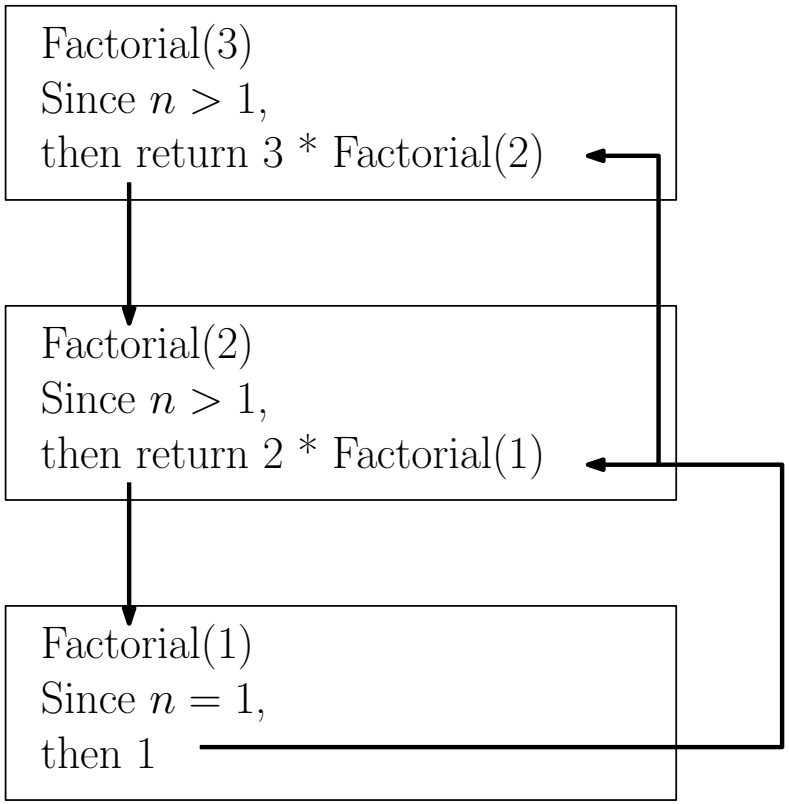


Figure: Ilustrasi algoritma rekursif FAKTORIAL dimana $n = 3$

2.1 - Komputasi faktorial (4): Pembuktian kebenaran dengan induksi

- Basis induksi: dari baris 1, kita melihat bahwa fungsi berfungsi dengan benar untuk $n = 1$.
- Hipotesis: misalkan fungsinya bekerja dengan benar untuk suatu input berukuran $n = m$, untuk beberapa $m \geq 1$.
- Tahap induksi: Kita buktikan bahwa itu juga berfungsi ketika dipanggil dengan $n = m + 1$. Berdasarkan hipotesis, kita tahu bahwa panggilan rekursif bekerja dengan benar untuk $n = m$ dan memberikan hasil $m!$.

  Selanjutnya, dikalikan dengan $n = m + 1$, sehingga menghasilkan $(m + 1)!$. Dan ini adalah nilai yang dikembalikan dengan benar oleh program.

2.2. Menemukan elemen maksimum dari array

2.2 - Menemukan elemen maksimum dari array (1)

Untuk menghitung maks elemen n untuk $n > 1$ secara rekursif:

- Hitung maks dari $n - 1$ elemen
- Bandingkan dengan elemen terakhir untuk menemukan seluruh maks

2.2 - Menemukan elemen maksimum dari array (1)

Untuk menghitung maks elemen n untuk $n > 1$ secara rekursif:

- Hitung maks dari $n - 1$ elemen
- Bandingkan dengan elemen terakhir untuk menemukan seluruh maks

Algorithm 6 Finding maximum of an array

```
1:  procedure MAX(A[0..n − 1], int n)
2:      if n = 1 then return A[0]
3:      else
4:          T = MAX(A, n − 1)
5:          if T < A[n − 1] then
6:              return A[n − 1]
7:          else
8:              return T
9:          end if
10:     end if
11: end procedure
```

2.2 - Menemukan elemen maksimum dari array (2)

Tugas:

- Hitunglah kompleksitas algoritma di atas!
- Periksalah kebenaran algoritma di atas!

2.3. Menghitung jumlah elemen pada array

2.3 - Menghitung jumlah elemen pada array (1)

Prermasalahan: Diberikan sebuah array dari n elemen $A[0..n-1]$. Kita ingin menghitung nilai dari: $S = \sum_{i=0}^{n-1} A[i]$

Algorithm 7 Sum of an array

```
1:  procedure SUM(A[0..n], int n)
2:      if n = 1 then  return A[0]
3:      else
4:          S = SUM(A, n − 1)
5:          S = S + A[n − 1]
6:          if T < A[n − 1] then
7:              return S
8:          end if
9:      end if
10: end procedure
```

2.3 - Menghitung jumlah elemen dalam array (2)

Tugas:

- Hitunglah kompleksitas algoritma di atas!
- Periksalah kebenaran algoritma di atas!

2.4. Recursive MAX

2.4. Recursive MAX, pendekatan kedua (1)

Permasalahan: Diberikan array A dari n elemen, kita bertujuan untuk menemukan elemen dengan nilai maksimum array.

Pendekatan:

1. Bagilah array menjadi dua bagian sub-array, yaitu sub-array Left dan sub-array Right.
2. Temukan maks dari setiap sub-array.
3. Bandingkan nilai maksimum array kiri dan array kanan.
4. Mengembalikan maksimum dari dua nilai.

2.4. Recursive MAX, 2nd Pendekatan (2)

Algorithm 8 Finding max of an array

```
1:  procedure FindMax(A[i..j], n)        ▷ i, j are respectively the index of start, end of A
2:      if n = 1 then  return A[S]
3:      end if
4:      m = ⌊(i+j)/2⌋
5:      T1 = FindMax(A[i..m], ⌊n/2⌋)      ▷ Recursive call the left sub-array
6:      T2 = FindMax(A[(m+1)..j], n − ⌊n/2⌋)   ▷ Rec. call right sub-array
7:      if T1 ≥ T2 then return T1          ▷ Compare the two max elements
8:      else return T2
9:      end if
10: end procedure
```

Remark. $\lfloor x \rfloor$ berarti bilangan bulat terbesar yang $\leq x$;
contoh: $\lfloor 3.5 \rfloor = 3$

2.4. Rekursif MAX, 2nd Pendekatan (3)

Analisis kompleksitas: Kasus khusus ketika $n = 2^k$

Misalkan $f(n)$: jumlah perbandingan kunci untuk menemukan maks dari n-array, dengan $n = 2^k$ untuk beberapa bilangan bulat positif k. Sehingga:

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

By repeated substitution:

$$\begin{aligned} f(n) &= 1 + 2f(n/2) \\ &= 1 + 2[1 + 2f(n/4)] = 1 + 2 + 2f(n/4) \\ &= 1 + 2 + 4 + 8f(n/4) \\ &\vdots \\ &= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k f(n/2^k) \\ &= 1 + 2 + 4 + \cdots + 2^{k-1} \\ &= 2^k - 1/(2-1) = 2^k - 1 \\ &= n - 1 \end{aligned}$$

2.4. Recursive MAX, 2nd Pendekatan (4)

$f(n)$: banyaknya perbandingan kunci untuk menemukan maksimum n-array, dengan $n = 2^k$ untuk beberapa $k \in \mathbb{Z}^+$.

Analisis kompleksitas: Untuk bilangan bulat n

$$f(n) = \begin{cases} 0, & n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + f(n - \lfloor \frac{n}{2} \rfloor) + 1, & n \geq 2 \end{cases}$$

Buktikan bahwa:

Dengan induksi, diperoleh $f(n) = n - 1$. Bagaimana ini diperoleh?

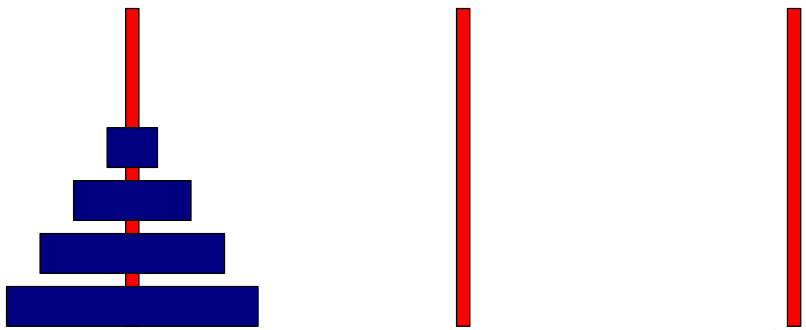
Bagian 3. Tower of Hanoi

Masalah Menara Hanoi (1): Problem statement

Permasalahan: ada tiga menara A, B, dan C. Awalnya, ada n disk dengan berbagai ukuran yang ditumpuk di menara A, diurutkan berdasarkan ukurannya, dengan disk terbesar di bawah dan yang terkecil di atas. Tujuannya adalah untuk memindahkan semua disk ke menara ke-2 dengan menjaga urutannya.

- Hanya satu cakram yang dapat dipindahkan pada satu waktu secara terbatas, dari puncak satu menara ke puncak menara lainnya.
- Cakram yang lebih besar tidak boleh diletakkan di atas cakram yang lebih kecil.

Cek `https://www.mathsisfun.com/games/towerofhanoi.html` untuk ilustrasi permasalahannya



Masalah Menara Hanoi (2): Ilustrasi

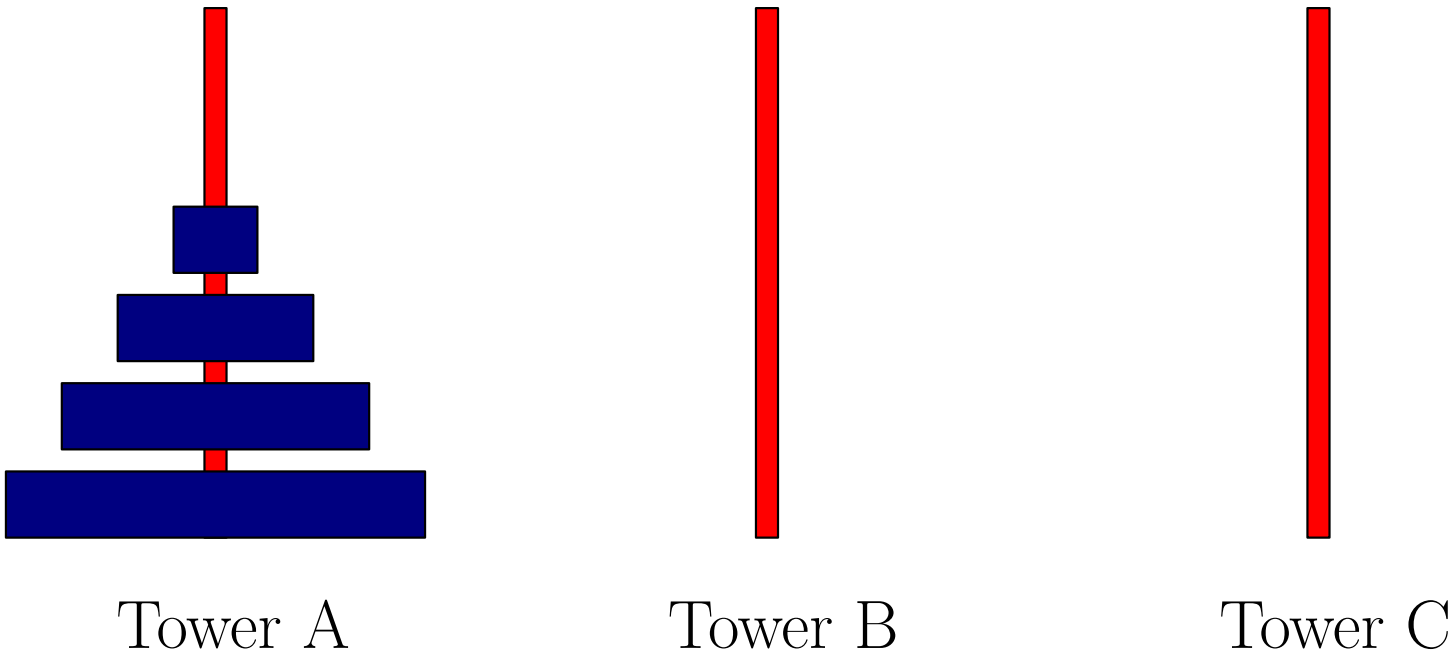


Figure: Konfigurasi awal dengan 4 disk di Tower A

Masalah Menara Hanoi (2): Ilustrasi

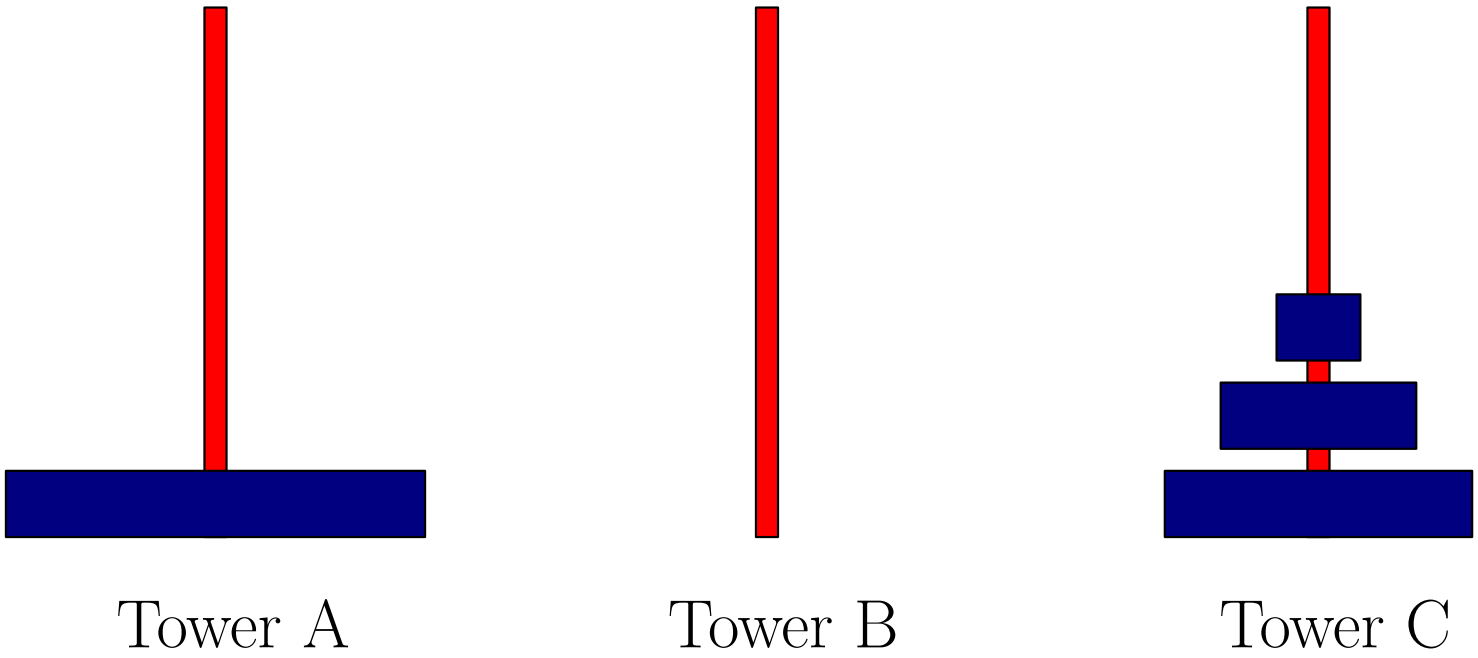


Figure: Setelah secara rekursif memindahkan 3 disk teratas dari Tower A ke Tower C

Masalah Menara Hanoi (2): Ilustrasi

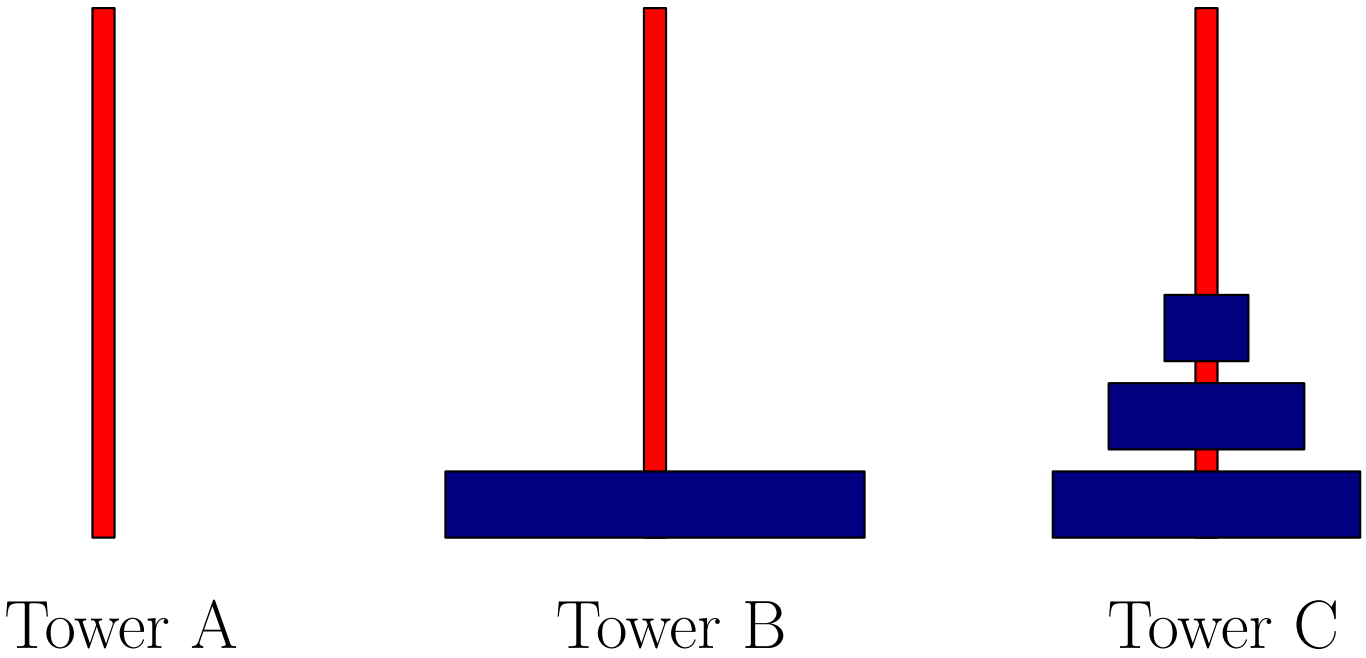


Figure: Setelah memindahkan cakram bawah dari Menara A ke Menara B

Masalah Menara Hanoi (2): Ilustrasi

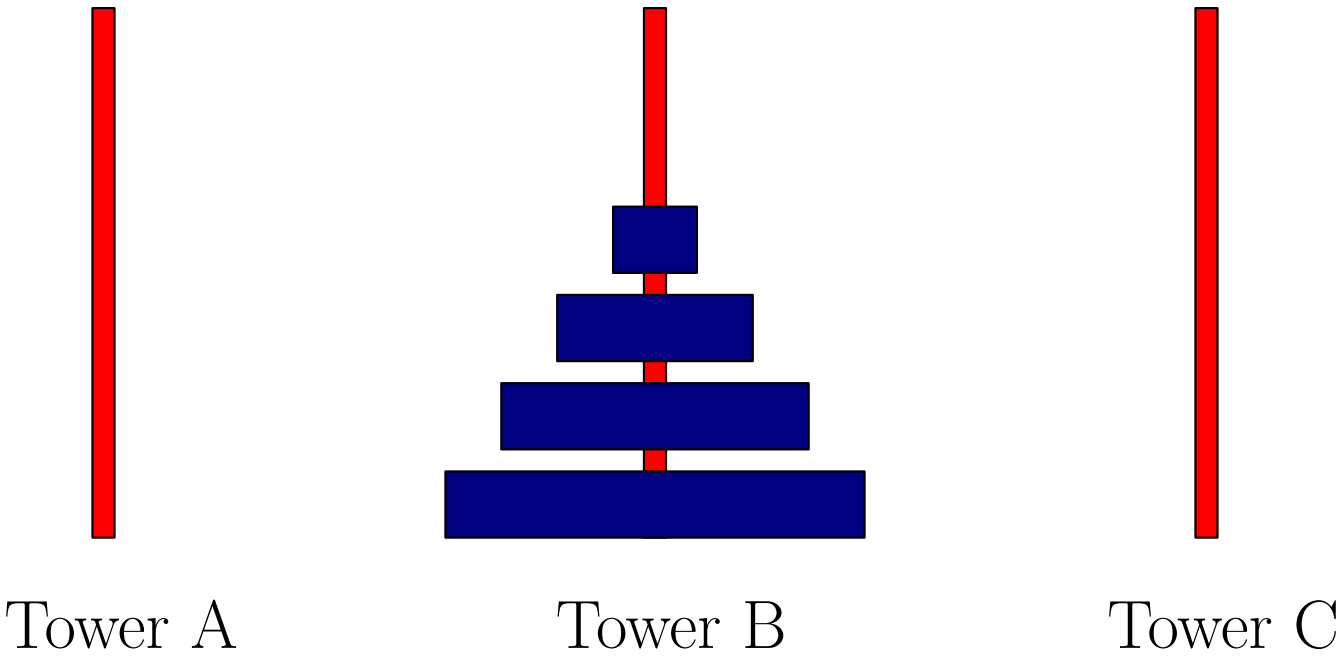


Figure: Setelah secara rekursif memindahkan 3 disk dari Tower C ke Tower B

Masalah Menara Hanoi (3): Pseudocode

Task: Tuliskan pseudocode untuk Masalah Menara Hanoi!

Masalah Menara Hanoi (3): Pseudocode

Task: Tuliskan pseudocode untuk Masalah Menara Hanoi!

Algorithm 10 Tower of Hanoi

```
1: procedure TOWERS(A, B, C, n)
2:     if n = 1 then
3:         MOVEONE(A, B)
4:         return
5:     end if
6:     TOWERS(A, C, B, n − 1)
7:     MOVEONE(A, B)
8:     TOWERS(C, B, A, n − 1)
9: end procedure
```

- TOWERS(A, B, C, n): memindahkan n disk dari A ke B, dengan A, B, C
- MOVEONE(A, B): memindahkan satu disk dari A ke B

Masalah Menara Hanoi (4): Proof of correctness

Pembuktian kebenaran: **gunakan induksi**

- Kasus dasar: Untuk $n = 1$, satu gerakan adalah dibuat dari A ke B. Jadi algoritma bekerja dengan benar untuk $n = 1$.
- Untuk $n \geq 2$ apa pun, misalkan algoritma bekerja dengan benar untuk $n - 1$.
- Kemudian, dengan hipotesis, panggilan rekursif baris 6 bekerja dengan benar dan memindahkan $n - 1$ disk teratas ke C, meninggalkan disk terbawah di menara A.
- Langkah selanjutnya, baris 7, memindahkan disk paling bawah ke B.
- Akhirnya, panggilan rekursif baris 8 bekerja dengan benar oleh hipotesis dan memindahkan kembali $n - 1$ disk dari C ke B.
- Dengan demikian, seluruh algoritma bekerja dengan benar untuk n.

3. Masalah Menara Hanoi (5): Analisis kompleksitas waktu

Persamaan perulangan untuk menganalisis kompleksitas waktu

Misalkan $f(n)$: jumlah gerakan tunggal untuk menyelesaikan masalah n disk

Sehingga diperoleh hubungan berikut:

$$f(n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + 2f(n-1), & n \geq 2 \end{cases}$$

Remark. Rumus di atas dikenal sebagai rumus rekursif; baca halaman ini atau tonton video ini untuk pemahaman lebih detail.

Untuk mendapatkan rumus eksplisit $f(n)$, kita harus menyelesaikan persamaan perulangan untuk $f(n)$.

Masalah Menara Hanoi (6): Analisis kompleksitas waktu

Metode 1: Substitusi berulang

Masalah Menara Hanoi (6): Analisis kompleksitas waktu

Metode 1: Substitusi berulang

$$\begin{aligned} f(n) &= 1 + 2 \cdot f(n-1) \\ &= 1 + 2 + 4 \cdot f(n-2) \\ &= 1 + 2 + 4 + 8 \cdot f(n-3) \\ &= \cdots \\ &= 1 + 2 + 2^2 + \cdots + 2^{n-1} \cdot f(1) \end{aligned}$$

Substitusikan *base case* $f(1) = 1$. Dengan rumus *deret geometris* (klik di sini untuk memeriksa rumus), diperoleh:

$$f(n) = \frac{2^n - 1}{2 - 1} = 2^n - 1$$

Masalah Menara Hanoi (7): Analisis kompleksitas waktu

Metode 2: Tebak solusinya dan buktikan dengan induksi

Masalah Menara Hanoi (7): Analisis kompleksitas waktu

Metode 2: Tebak solusinya dan buktikan dengan induksi

Misalkan tebakan kita adalah "$f(n)$ adalah fungsi eksponensial"

Tebakan: $f(n) = a \cdot 2^n + b$

Bukti induksi:

- Basis induksi: $n = 1$
  - $f(1) = 1$ (dari rekurens)
  - $f(1) = 2a + b$ (dari bentuk solusinya)

  Sehingga $2a + b = 1$
- Induksi: Misalkan solusinya benar untuk beberapa $n \geq 1$:

$$f(n) = a \cdot 2^n + b$$

Maka solusinya harus berlaku untuk $n + 1$, yaitu:

$$f(n+1) = a \cdot 2^{n+1} + b$$

Masalah Menara Hanoi (8): Analisis kompleksitas waktu

- Dari analisis rekurens, diperoleh

$$\begin{aligned} f(n+1) &= 2f(n)+1 \\ &= 2(a\cdot 2^n + b)+1 \\ &= a\cdot 2^{n+1} + (2b+1) \end{aligned}$$

- Dari kedua persamaan diperoleh:

$$a\cdot 2^{n+1} + b = a\cdot 2^{n+1} + (2b+1) \Leftrightarrow 2b+1 = b \Leftrightarrow b = -1$$

  Sehingga $2a+b=1 \Leftrightarrow a=1$. Jadi, $b=-1$.

- Sehingga $f(n) = a\cdot 2^n + b = 2^n - 1$.

Bagian 4. Binary Search

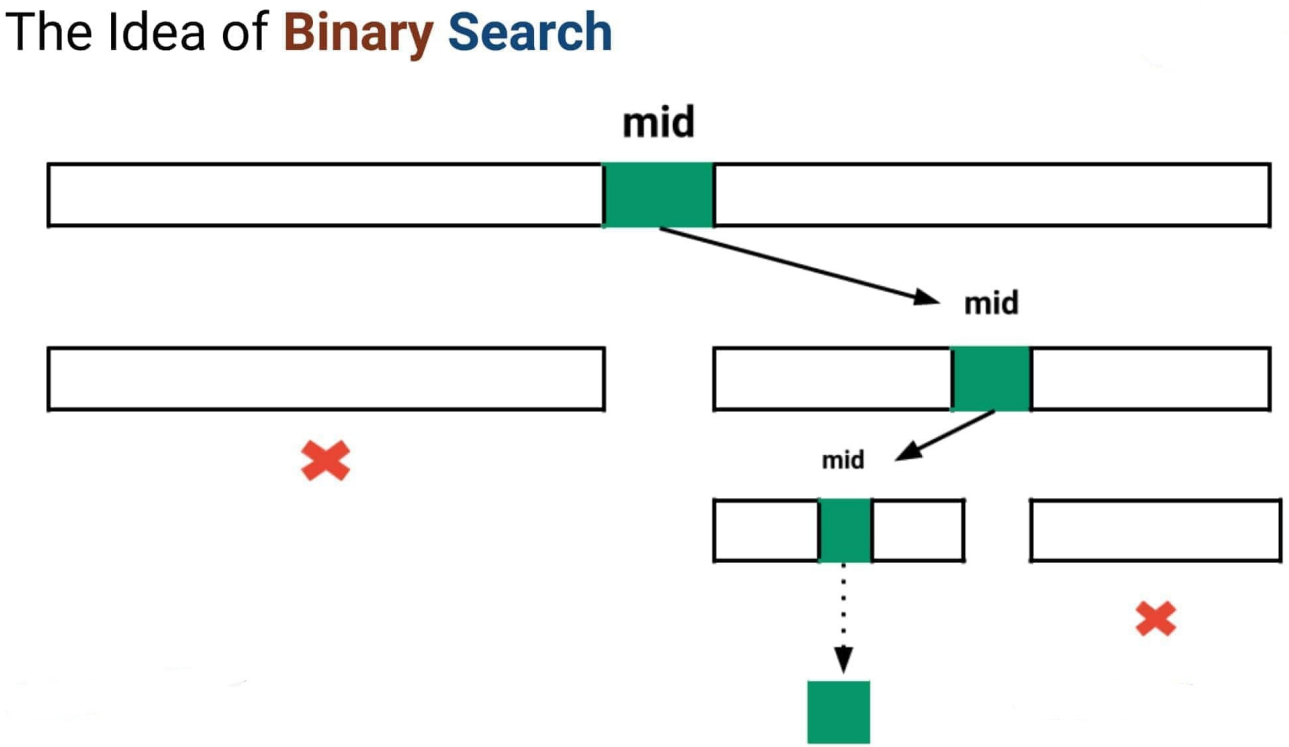
4. Algoritma Binary Search (1): Prinsip

Permasalahan: Diberikan array *sorted* $A[0..n-1]$ dan kunci pencarian KEY. Algoritma melakukan langkah-langkah berikut:

- Jika $KEY = A[m]$, maka kembalikan m
- Jika $KEY < A[m]$, maka cari secara rekursif di bagian kiri array
- Jika $KEY > A[m]$, maka cari secara rekursif di bagian kanan array

Di setiap langkah, ukuran pencarian *berkurang setengahnya*.

Binary Search (2): Diagram



source: `https://www.enjoyalgorithms.com/blog/binary-search-algorithm`

Binary Search (3): Pseudocode

Algorithm 11 Algoritma Binary Search

```
1:  procedure BinSearch(A, i, j, KEY)
2:      if i > j then
3:          return −1                          ▷ Base case is reached but KEY is not found
4:      end if
5:      m = ⌊(i+j)/2⌋                          ▷ Choose the pivot
6:      if KEY = A[m] then
7:          return m                           ▷ KEY is found at index m
8:      else
9:          if KEY < A[m] then                 ▷ The KEY is located on the Left sub-array
10:             return BinSearch(A, i, m − 1, KEY)   ▷ Rec-call left part
11:         else
12:             return BinSearch(A, m + 1, j, KEY)   ▷ Rec-call right part
13:         end if
14:     end if
15: end procedure
```

Binary Search (4): Analisis kompleksitas waktu

Misalkan $f(n)$ adalah banyaknya perbandingan.

Analisis kompleksitas: kasus khusus ketika $n = 2^k$

$$f(n) = \begin{cases} 1, & n = 1 \\ 1 + f(n/2), & n \geq 2 \end{cases}$$

Dengan substitusi berulang:

$$\begin{aligned} f(n) &= 1 + f(n/2) \\ &= 1 + 1 + f(n/4) \\ &= 1 + 1 + 1 + f(n/8) \\ &\vdots \\ &= k + f(n/2^k) \\ &= k + f(1) \\ &= k + 1 \\ &= \log n + 1 \end{aligned}$$

Binary Search (5): Analisis kompleksitas waktu

Analisis kompleksitas: untuk sebarang nilai n

$$f(n) = \begin{cases} 1, & n = 1 \\ 1 + f(\lfloor \frac{n}{2} \rfloor), & n \geq 2 \end{cases}$$

Dengan induksi, diperoleh: $f(n) = \lfloor \log n \rfloor + 1$

Sebagai latihan, coba tuliskan secara runut pembuktian dengan induksi tersebut.

Binary Search (6): Bukti induktif untuk kompleksitas

- Basis induksi: $n = 1$:
  Dari perulangan, $f(1) = 1$, dan solusi yang diklaim $f(1) = \lfloor \log 1 \rfloor + 1 = 1$.
  Jadi basis induksi benar.

- Pembuktian hipotesis: Misalkan rumus tersebut benar untuk semua nilai yang lebih kecil.

$$f(m) = \lfloor \log m \rfloor, \quad \forall m < n$$

  Setiap bilangan bulat n dapat dinyatakan sebagai:

$$2^{k-1} \leq \lfloor n/2 \rfloor < 2^k$$

  untuk beberapa bilangan bulat k.

  Jadi, $\lfloor \log \lfloor n/2 \rfloor \rfloor = k - 1$.

  Berdasarkan fungsi rekursif:

$$f(\lfloor n/2 \rfloor) = \lfloor \log \lfloor n/2 \rfloor \rfloor + 1 = (k-1) + 1 = k = \lfloor \log n \rfloor$$

  Maka:

$$f(n) = f(\lfloor n/2 \rfloor) + 1 = k + 1 = \lfloor \log n \rfloor + 1$$

Contoh lanjut: Recursive powering

Recursive powering (1): Deskripsi masalah

Permasalahan: Diberikan X dan bilangan bulat n. Kita ingin menghitung X^n.

Algorithm 12 Recursive powering (*brute force*)

```
1: procedure POWER1(X, n)
2:     T = X
3:     for i = 2 to n do
4:         T = T * X
5:     end for
6: end procedure
```

Kompleksitas $\mathcal{O}(n)$. Selidikilah mengapa?

Recursive powering (2): Penyelesaian

Ide: $X^{16} = ((((X^2)^2)^2)^2)^2$

Diberikan $n = 2^k$, kita dapat mencari kuadrat-nya secara berulang.

Algorithm 13 Improvement brute force

```
1: procedure POWER2(X, n = 2^k)
2:     T = X
3:     for i = 2 to k do
4:         T = T * T
5:     end for
6: end procedure
```

Kompleksitas: $\mathcal{O}(\log n)$. Selidikilah mengapa?

Recursive powering (3): Penyelesaian

Perumuman untuk sebarang nilai n: Hitunglah X^n untuk $n \in \mathbb{Z}^+$

- Hitung $X^2 = X * X$
- Hitung $X^3 = X^2 * X$
- Hitung $X^6 = X^3 * X^3$
- Hitung $X^{12} = X^6 * X^6$
- Hitung $X^{13} = X^{12} * X$

Recursive powering (4): Penyelesaian

Ide dasar: Bagi n dengan 2, $n = n/2 + n/2$. Jadi

$$X^n = X^{(n/2+n/2)} = X^{n/2} \cdot X^{n/2}$$

Masalahnya adalah $n/2$ tidak selalu bilangan bulat. Jadi kita harus menerapkan sedikit modifikasi:

- Untuk $n = 0$, lalu $X^n = 1$
- Untuk $n > 0$, maka:
  - Jika n adalah *genap*, maka $X^n = X^{n/2} \cdot X^{n/2}$
  - Jika n adalah *ganjil*, maka $X^n = X^{\lfloor n/2 \rfloor} \cdot X^{\lfloor n/2 \rfloor} \cdot X$

Recursive powering (5): Pseudocode

Algorithm 14 Recursive powering

```
1:  procedure POWER3(X, n)
2:      if n = 1 then
3:          return X
4:      end if
5:      T = POWER3(X, ⌊n/2⌋)
6:      T = T * T                                   ▷ T = T^⌊n/2⌋ * T^⌈n/2⌉
7:      if n mod 2 = 1 then                         ▷ skipped
8:          T = T * X
9:          return T
10:     end if
11: end procedure
```

Kompleksitas: ?

Recursive powering (6): Contoh penerapan

Contoh: Hitung 3^{16}

$$
\begin{aligned}
3^{16} = 3^8 \cdot 3^8 &= (3^8)^2 \\
&= ((3^4)^2)^2 \\
&= (((3^2)^2)^2)^2 \\
&= ((((3^1)^2)^2)^2)^2 \\
&= ((((3^0) \cdot 3)^2)^2)^2)^2 \\
&= (((1 \cdot 3)^2)^2)^2)^2 \\
&= (((3)^2)^2)^2)^2 \\
&= (((9)^2)^2)^2 \\
&= ((81)^2)^2 \\
&= (6561)^2 \\
&= 43,046,721
\end{aligned}
$$

Recursive powering (7): Kebenaran algoritma

Algorithm 14 Power by multiplications

```
1:  procedure POWER3(X, n)
2:      if n = 1 then
3:          return X
4:      end if
5:      T = POWER(X, ⌊n/2⌋)
6:      T = T * T
7:      if n mod 2 = 1 then
8:          T = T * X
9:          return T
10:     end if
11: end procedure
```

Misalkan $n = 2m + r$, dimana $r \in \{0, 1\}$.

- Algoritma melakukan panggilan rekursif untuk menghitung $T = X^m$.
- Kuadratkan T untuk mendapatkan $T = X^{2m}$. Jika $r = 0$, maka *return*.
- Jika tidak, ketika $r = 1$, algoritma mengalikan T dengan X, untuk menghasilkan $T = X^{2m+1}$.

Recursive powering (8): Analisis kompleksitas waktu

Misalkan $f(n)$: jumlah kasus terburuk dari langkah perkalian untuk menghitung X^n.

- Panggilan rekursif mengambil perkalian $f(\lfloor \frac{n}{2} \rfloor)$.
- Kemudian diikuti dengan satu perkalian lagi. Dalam kasus terburuk, ketika n ganjil, satu perkalian tambahan dibutuhkan.

Jadi,

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + 2, & \text{if } n \geq 2, n \text{ ganjil} \\ f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2, n \text{ genap} \end{cases}$$

Tunjukkan bahwa $f(n) = 2\lfloor \log n \rfloor$.

Recursive powering (8): Analisis kompleksitas waktu

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + 2, & \text{if } n \geq 2, n \text{ odd} \\ f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2, n \text{ even} \end{cases}$$

Dua kasus terakhir memiliki perbedaan kecil. Jadi kita dapat mengaproksimasi fungsi di atas dengan fungsi berikut untuk menyederhanakan perhitungan:

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + 2, & \text{if } n \geq 2 \end{cases}$$

Recursive powering (9): Pembuktian induktif

Tunjukkan bahwa $f(n) = 2\lfloor \log n \rfloor$.

- Basis induksi ($n = 1$): Dari pengulangan, $f(1) = 0$, dan dari rumus, $f(1) = 2\lfloor \log 1 \rfloor = 0$. Correct.
- Pembuktian induktif: Misalkan rumusnya benar untuk semua nilai yang lebih kecil.

  $$f(m) = 2\lfloor \log m \rfloor, \quad \forall m < n$$

  Setiap bilangan bulat n dapat dinyatakan sebagai (untuk beberapa bilangan bulat k):

  $$2^k \leq n < 2^{k+1}$$

  Jadi, $\lfloor \log n \rfloor = k$, and $\lfloor \frac{\log n}{2} \rfloor = k - 1$. Dengan fungsi rekursif:

  $$f(n) = f(\lfloor \frac{n}{2} \rfloor) + 2 = 2(k-1) + 2 = 2k = 2\lfloor \log n \rfloor$$

Remark. Pendekatan ini memberikan kompleksitas yang lebih baik dibandingkan *brute-force* ($\mathcal{O}(n)$).

Bagian 5. *Redundancy* pada algoritma rekursif

Contoh 1: Recursive powering (1)

Algorithm 14 Power by multiplications

```
1:  procedure POWER3(X, n)
2:      if n = 1 then
3:          return X
4:      end if
5:      T = POWER(X, ⌊n/2⌋)
6:      T = T * T
7:      if n mod 2 = 1 then
8:          T = T * X
9:          return T
10:     end if
11: end procedure
```

Apakah perlu untuk menyimpan $\textsc{Power}(X, \lfloor \frac{n}{2} \rfloor)$ dalam beberapa variabel T?

Contoh 1: Recursive powering (2)

Misalkan bahwa $n = 2^k$ untuk beberapa nilai k.

Algorithm 15 Recursive powering

```
1: procedure POWER4(X, n)
2:     if n = 1 then
3:         return X
4:     end if
5:     return POWER(X, ⌊n/2⌋) * POWER(X, ⌊n/2⌋)
6: end procedure
```

- Apakah algoritmanya benar?
- Bagaimana kompleksitasnya?

Contoh 1: Recursive powering (3)

Algoritmanya benar.

Jumlah panggilan rekursif:

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2 \end{cases}$$

Dengan induksi, kita dapat membuktikan bahwa $f(n) = n - 1$ (lebih buruk secara asimtotik dari algoritma sebelumnya).

Apa yang dapat Anda simpulkan?

Contoh 1: Recursive powering (3)

Algoritmanya benar.

Jumlah panggilan rekursif:

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2 \end{cases}$$

Dengan induksi, kita dapat membuktikan bahwa $f(n) = n - 1$ (lebih buruk secara asimtotik dari algoritma sebelumnya).

Apa yang dapat Anda simpulkan?
POWER4 juga tidak efisien, karena kita melakukan dua pemanggilan rekursif untuk fungsi yang sama $f(\lfloor \frac{n}{2} \rfloor)$

Contoh 2: Barisan Fibonacci (1)

Barisan Fibonacci didefinisikan sebagai berikut.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n \geq 3 \end{cases}$$

Barisan Fibonacci: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Contoh 2: Barisan Fibonacci (1)

Barisan Fibonacci didefinisikan sebagai berikut.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n \geq 3 \end{cases}$$

Barisan Fibonacci: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Konstruksilah sebuah algoritma untuk menghitung deret Fibonacci!

- Dengan algoritma naif (brute force), kita dapat mencapai kompleksitas $\mathcal{O}(n)$. Bagaimana?

Contoh 2: Barisan Fibonacci (1)

Barisan Fibonacci didefinisikan sebagai berikut.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n \geq 3 \end{cases}$$

Barisan Fibonacci: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Konstruksilah sebuah algoritma untuk menghitung deret Fibonacci!

- Dengan algoritma naif (brute force), kita dapat mencapai kompleksitas $\mathcal{O}(n)$. Bagaimana?

  Dengan perulangan (metode berulang); kita menambahkan nomor satu per satu.

Contoh 2: Barisan Fibonacci (1)

Barisan Fibonacci didefinisikan sebagai berikut.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n \geq 3 \end{cases}$$

Barisan Fibonacci: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Konstruksilah sebuah algoritma untuk menghitung deret Fibonacci!

- Dengan algoritma naif (brute force), kita dapat mencapai kompleksitas $\mathcal{O}(n)$. Bagaimana?

  Dengan perulangan (metode berulang); kita menambahkan nomor satu per satu.

- Buat algoritma rekursif!

Contoh 2: Barisan Fibonacci (2)

Algorithm 16 Barisan Fibonacci

1: **procedure** $\textsc{Fib}(n)$
2: **if** $n \leq 2$ **then** **return** 1
3: **end if**
4: **return** $(\textsc{Fib}(n-1) + \textsc{Fib}(n-2))$
5: **end procedure**

Contoh 2: Barisan Fibonacci (2)

Algorithm 17 Barisan Fibonacci

```
1: procedure FIB(n)
2:     if n ≤ 2 then return 1
3:     end if
4:     return (FIB(n − 1) + FIB(n − 2))
5: end procedure
```

Program ini membuat panggilan rekursif dengan banyak perhitungan yang tumpang tindih, menyebabkan inefisiensi yang sangat besar.

Contoh 2: Barisan Fibonacci (2)

Algorithm 18 Barisan Fibonacci

```
1: procedure FIB(n)
2:     if n ≤ 2 then return 1
3:     end if
4:     return (FIB(n − 1) + FIB(n − 2))
5: end procedure
```

Program ini membuat panggilan rekursif dengan banyak perhitungan yang tumpang tindih, menyebabkan inefisiensi yang sangat besar.

Kompleksitas:

$$T(n) = \begin{cases} 0, & n = 1 \\ 0, & n = 2 \\ T(n-1) + T(n-2) + 1, & n \geq 3 \end{cases}$$

Tunjukkan bahwa: fungsi eksplisit $T(n) \geq (1.618)^{n-2}$.

Bagian 6. Kelebihan & kekurangan algoritma rekursif

Kelebihan & kekurangan algoritma rekursif (1)

Kelebihan

- Rekursi memberikan kejelasan dan mengurangi waktu yang dibutuhkan untuk menulis dan men-debug kode (karena mengurangi panjang kode).
- Bermanfaat pada penyelesaian masalah yang secara alami bersifat rekursif, misalnya Masalah Menara Hanoi.
- Rekursi dapat mengurangi kompleksitas waktu (*terkadang kontra-intuitif*).
- Mengurangi pemanggilan fungsi yang tidak perlu.
- Sangat berguna saat menerapkan solusi yang sama.

Kelebihan dan kekurangan algoritma rekursif (2)

Kekurangan

- Fungsi rekursif umumnya lebih lambat daripada fungsi non-rekursif.
- Mungkin memerlukan banyak ruang memori untuk menyimpan "hasil antara" pada proses rekursi.
- Cenderung sulit untuk menganalisis atau memahami kode.
- Tidak lebih efisien dari segi kompleksitas ruang dan waktu (bisa lambat).
- Komputer mungkin kehabisan memori jika panggilan rekursif tidak diperiksa dengan benar.

Rangkuman...

What have we learned today?

1. Peninjauan kembali algoritma brute force
2. Memahami konsep algoritma rekursif
3. Beberapa contoh algoritma rekursif
4. Persamaan perulangan untuk menganalisis kompleksitas waktu
5. Redundansi dalam rekursi $\rightarrow$ jadi, berhati-hatilah saat menuliskan kode
6. Algoritma Binary Search

end of slide...