COMPLEXITY ANALYSIS

PELATNAS 1 TOKI 2019

SUDAH BACA IOI SYLLABUS?

AL1. Basic algorithmic analysis

- ✓ Algorithm specification, precondition, postcondition, correctness, invariants
- ✓ Asymptotic analysis of upper complexity bounds (informally if possible)
- ✓ Big O notation
- ✓ Standard complexity classes: constant, logarithmic, linear, $\mathcal{O}(n \log n)$, quadratic, cubic, exponential, etc.
- ✓ Time and space tradeoffs in algorithms
- ✓ Empirical performance measurements.

IOI Syllabus: https://people.ksp.sk/~misof/ioi-syllabus/ioi-syllabus-2018.pdf

MANFAAT COMPLEXITY ANALYSIS

- Ada patokan untuk membandingkan algoritme.
- Estimasi *running time* pada saat kompetisi.
- Kebiasaan menganalisis dahulu suatu algoritme sebelum membuat program —> mengurangi waktu sia-sia akibat memprogram algoritme yang tidak sesuai dengan *constraint*.
- Menghindari *useless optimization*.

BAGAIMANA ANDA **MEMBANDINGKAN** SATU ATAU LEBIH ALGORITME YANG MENYELESAIKAN MASALAH SERUPA?

DUA HAL YANG SERING DIBANDINGKAN

- **Time:** berapa lama sebuah algoritme bekerja?
- **Space:** berapa banyak ukuran *memory* yang dibutuhkan oleh algoritme?

PERBANDINGAN EMPIRIS

- Membandingkan algoritme berdasarkan lamanya *runtime*.
- Asumsi: lingkungan komputer persis sama untuk setiap algoritme yang dijalankan (mis: bahasa pemrograman dan komputer).

Run time (nanoseconds)		$1.3\ N^3$	$10\ N^2$	$47\ N \log_2 N$	$48\ N$
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 trillion	76 trillion
	day	41,000	2.9 million	50 trillion	1,800 trillion
N multiplied by 10, time multiplied by		1,000	100	10+	10

ANALISIS KOMPLEKSITAS

- Membandingkan algoritme pada level konsep.
- Tidak berpengaruh pada:
  - Bahasa pemrograman yang digunakan.
  - Perangkat keras dan lingkungan yang digunakan.
- Mengukur banyaknya operasi (f(*n*))yang dilakukan berdasarkan ukuran *input* (*n*) yang diberikan.
  - Operasi: yang paling dominan dalam suatu algoritme (operasi matematika, perbandingan, jumlah kueri, dll).
  - Input: panjang list, dimensi matriks, dll
- Sangat berguna untuk memprediksi perilaku algoritme ketika **ukuran *input* sangat besar.**

UKURAN INPUT DAN OPERASI DASAR

Problem	**Ukuran Input**	**Operasi Dasar**
Mencari sebuah *key* di dalam *list*	Banyaknya nilai yang disimpan dalam *list*	Perbandingan
Perkalian dua matriks	Dimensi matriks	Perkalian dua bilangan
Mengecek keprimaan suatu bilangan bulat N	Jumlah digit biner dari N	Pembagian
Masalah graf umum	Jumlah *vertex* dan *edge*	Berapa kali suatu *vertex* dikunjungi atau suatu *edge* dilewati.

MENGHITUNG F(*N*)

- Diberikan potongan program berikut yang menerima masukan berupa *array* berukuran *N*.

```
1  int max = input[0];
2
3  for(int i = 0; i < n; i++){
4      if(angka[i] >= max)
5          max = angka[i];
6  }
```

- Jika operasi berikut dihitung sebagai 1 instruksi:
  - *Assignment*
  - Akses elemen *array*
  - *Perbandingan dua nilai*
  - Menambah nilai (*increment*)
  - Aritmatika Dasar (penjumlahan dan perkalian).
- **Berapa jumlah instruksi (f(*n*))yang dieksekusi?**

MENGHITUNG F(*N*)

```
1  int max = input[0];
2
3  for(int i = 0; i < n; i++){
4      if(angka[i] >= max)
5          max = angka[i];
6  }
```

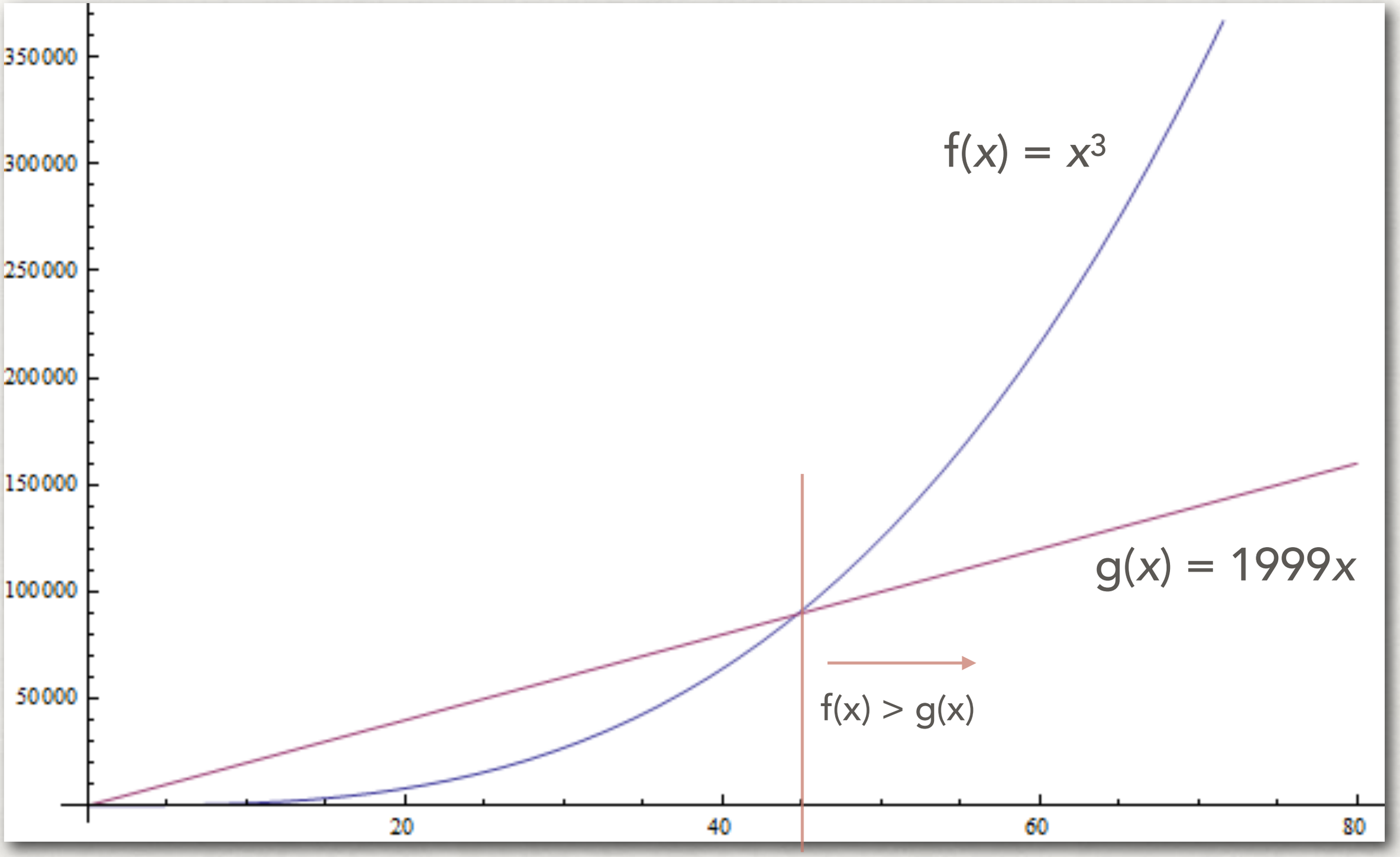
Instruksi pada bagian ini tidak jelas dilakukan berapa kali, bergantung pada hasil perbandingan yang dilakukan.

- *Worst-case scenario*: kondisi saat algoritme membutuhkan jumlah instruksi paling banyak (bisa juga berlaku untuk memori).
- Pada saat apa *worst case scenario* terjadi pada kasus di atas?
- Berapa (f(*n*)) ketika kondisi *worst case* terjadi? f(*n*) = 6*n* + 4
- Berapa (f(*n*)) ketika kondisi *best case* terjadi? f(*n*) = 4*n* + 4

SIFAT ASIMTOTIK

- f(n) sangat membantu kita memahami seberapa baik suatu algoritme. Namun, pada praktiknya, f(n) dapat disederhanakan.
- Cukup perhatian bagian yang bergantung pada ukuran *input*.
- Misal f(n) = $6n + 4$.
  - Cukup perhatikan $6n$ sehingga f(n) = $6n$.
- Bagaimana jika f(n) = $3n^2 + 6n + 12$?
  - Pilih *term* yang pertumbuhannya paling cepat: $3n^2$
- Konstanta pun bisa dihilangkan sehingga f(n) = $5n^3 + 1999n$ dapat ditulis sebagai f(n) = n^3.
  - f(n) = n^3 mendeskripsikan sifat asimtotik dari f(n) = $5n^3 + 1999n$.

SIFAT ASIMTOTIK



MENGHITUNG KOMPLEKSITAS

- Praktisnya, perhatikan bagian *loop*.
  - Jika tidak ada *loop*: f(*n*) = 1
  - Jika ada satu loop dari 1 … *n*: f(*n*) = *n*
  - Jika ada nested loop bersarang, masing-masing 1 … n: f(n) = *n***n**…**n*
  - Jika ada beberapa loop yang sekuensial: f(*n*) bergantung pada *loop* yang paling dominan.
- Bagaimana kalau bentuknya rekursif?
  - Subtitusi / pohon rekursi
  - Master Theorem

MENGHITUNG KOMPLEKSITAS

```
for (int i = 1; i <= 3*n; i++) {
    // code
}
```

```
for (int i = 1; i <= n+5; i++) {
    // code
}
```

```
for (int i = 1; i <= n; i += 2) {
    // code
}
```

MENGHITUNG KOMPLEKSITAS

```
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        // code
    }
}
```

MENGHITUNG KOMPLEKSITAS

```
for (int i = 1; i <= n; i++) {
    // code
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
for (int i = 1; i <= n; i++) {
    // code
}
```

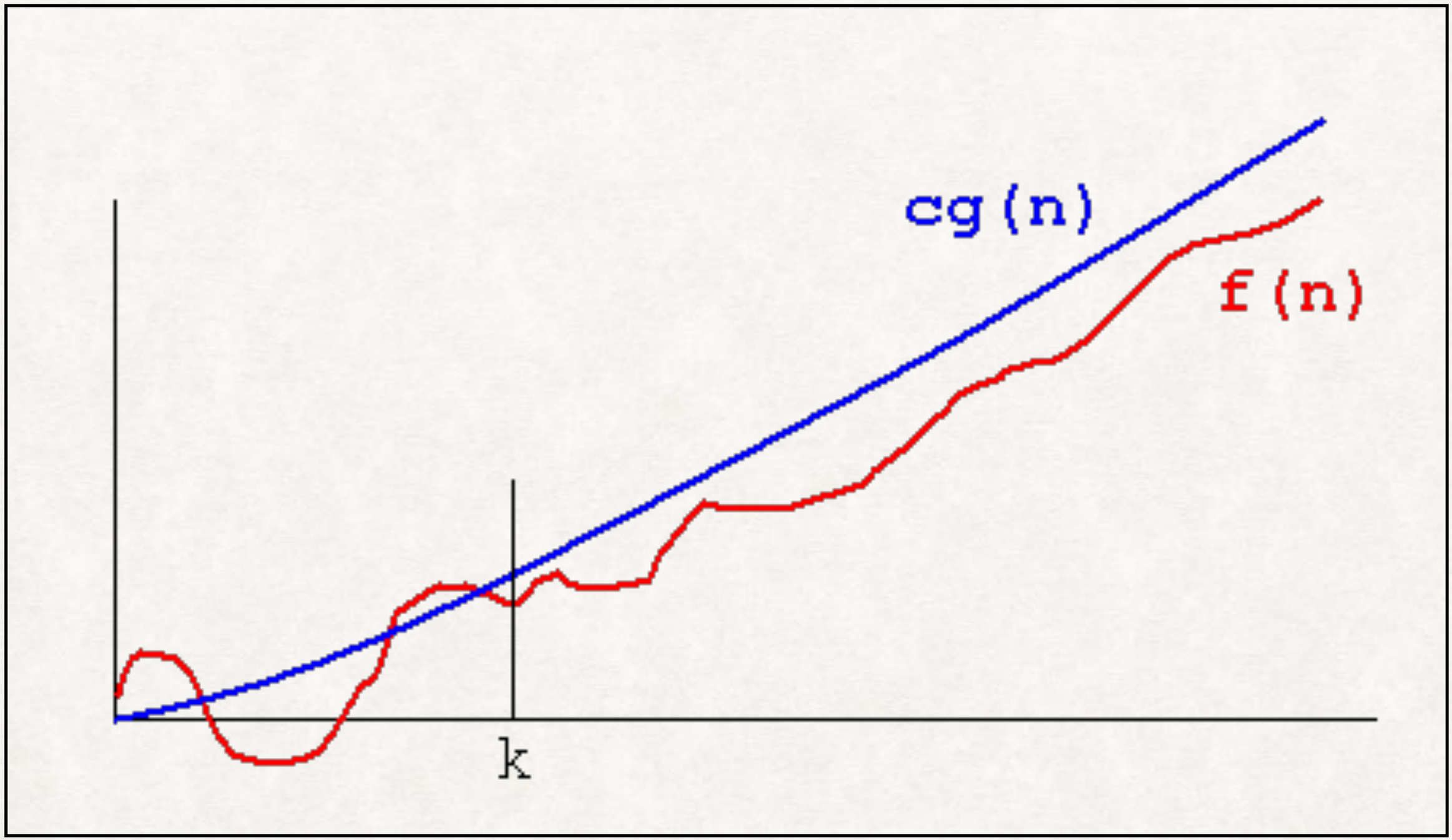
MENGHITUNG KOMPLEKSITAS

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // code
    }
}
```

BIG-O

- Menentukan f(*n*) yang spesifik pada suatu algoritme bisa menjadi sangat sulit, terutama pada algoritme yang kompleks.
- Namun kita bisa menyebut "f(*n*) dari suatu algoritme tidak akan pernah melebihi suatu batas atas (*upper bound*)" dengan lebih mudah.
- Asumsi: ukuran *input* besar.

BIG-O (FORMAL)



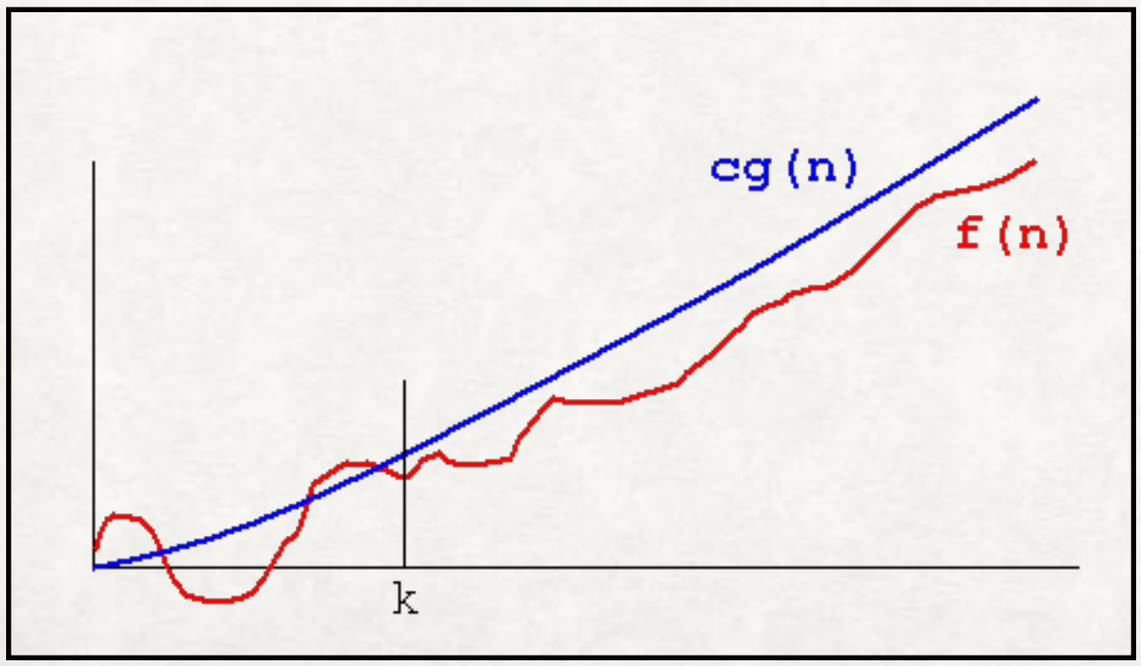
f(n) = O(g(n)): terdapat konstanta positif c dan k sehingga
0 ≤ f(n) ≤ cg(n) untuk n ≥ k

MISAL: SELECTION SORT

```
1  void selectionSort(int arr[], int n)
2  {
3      int i, j, min_idx;
4
5      for (i = 0; i < n-1; i++)
6      {
7          // Cari elemen terkecil pada array
8          min_idx = i;
9          for (j = i+1; j < n; j++)
10           if (arr[j] < arr[min_idx])
11             min_idx = j;
12
13         // Pindahkan elemen terkecil ke awal
14         swap(&arr[min_idx], &arr[i]);
15     }
16 }
```

BIG-O

- Selection sort = linear search sebanyak *n* kali.
  - f(*n*) = 1 + 2 + … + *n*-1 + *n* = *n* (*n* + 1) / 2 = ½n^2 + ½n
- f(*n*) = O(n^2)
  - Misal c = 5 maka f(n) ≤ 5 ketika n ≥ 1/9
  - c = 5 dan k = 1/9



f(*n*) = O(g(*n*)): terdapat konstanta positif c dan k sehingga 0 ≤ f(n) ≤ cg(n) untuk n ≥ k

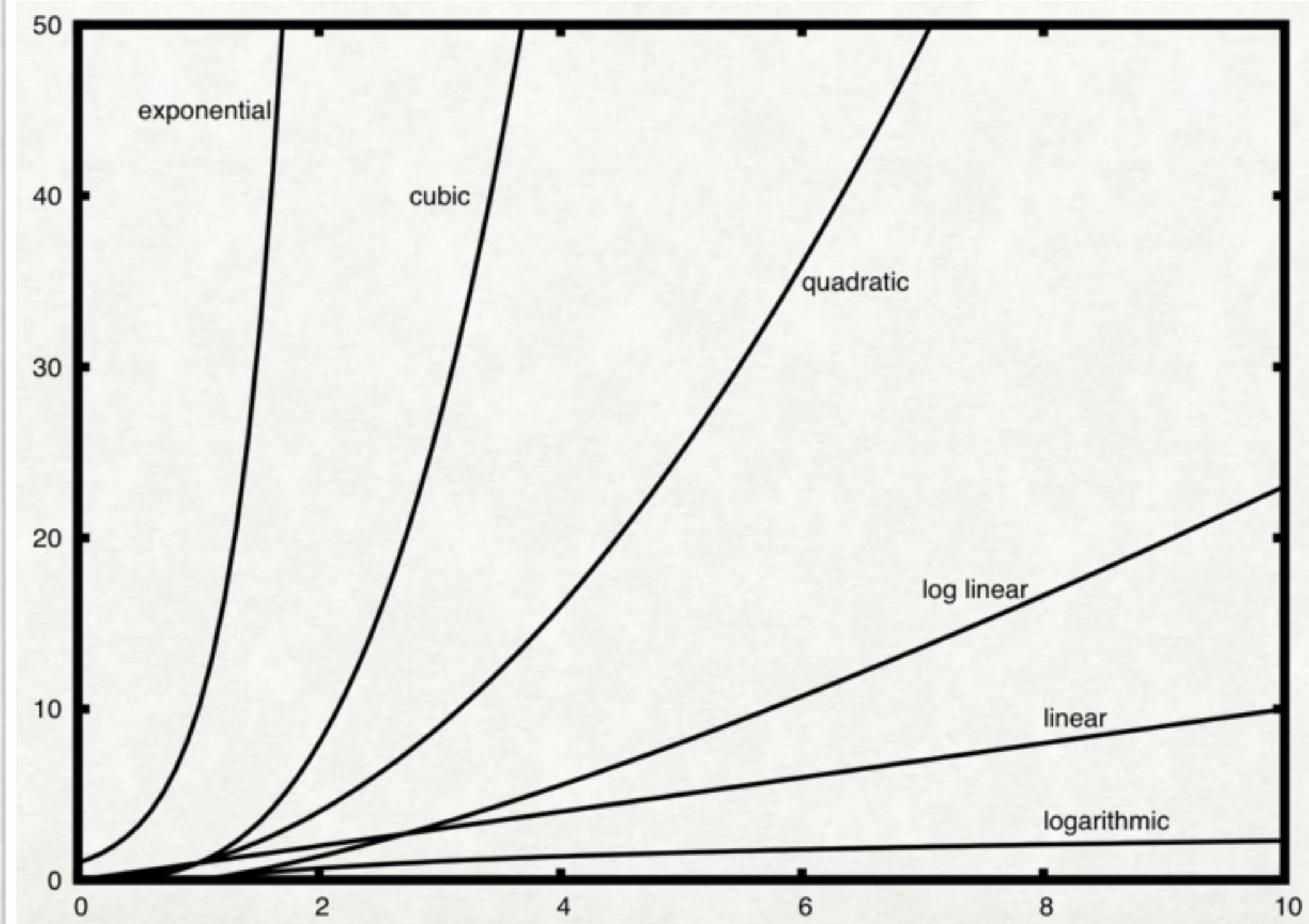
BIG-OH DARI FUNGSI BERIKUT?

$f(n)$= 689	
$f(n) = 689n$	
$f(n) = 6n + 89$	
$f(n) = 6n^8 + 9n$	
$f(n) = 6^n + 8n^9$	

BIG-OH DARI FUNGSI BERIKUT?

$f(n)$= 689	O(1)
$f(n)$ = 689n	O(n)
$f(n)$ = 6n + 89	O(n)
$f(n)$ = 6n^8 + 9n	O(n^8)
$f(n)$ = 6^n + 8n^9	O(6^n)

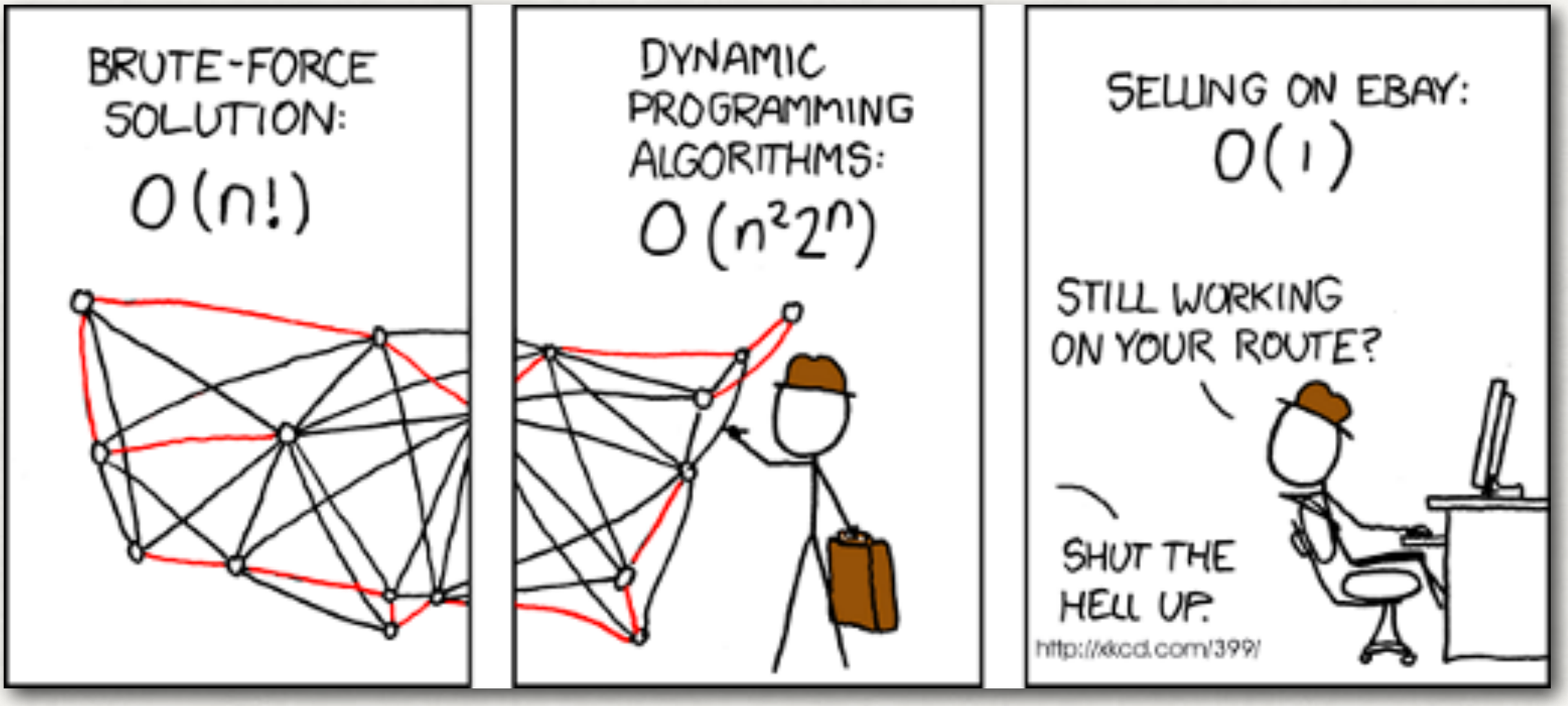
STANDARD COMPLEXITY CLASSES



STANDARD COMPLEXITY CLASSES

Big-O	**Kelas**	**Contoh**
O(1)	Konstan	Menghitung jumlah *n* bilangan bulat bertama.
O(log n)	Logaritmik	Binary search.
O(n)	Linear	Membaca *n* buah masukan. *Linear search*.
O(n^2)	Kuadratik	Bubble Sort.
O(n^3), O(n^4),...	Polinomial	Perkalian matriks
O(2^n)	Eksponensial	Seluruh kemungkinan n-bitset.

STANDARD COMPLEXITY CLASSES



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

PERKIRAKAN WAKTU EKSEKUSI PROGRAM

- Komputer modern dapat mengeksekusi 10^7 - 10^8 operasi dasar.

Ukuran Input (n)	Algoritme Terburuk untuk AC
$10^{10\text{^}8} < n$	O(1)
$10^6 < n < 10^{10\text{^}8}$	O(log n)
$10^4 < n < 10^6$	O(n) atau O(n log n)
$500 < n < 10^4$	O(n^2)
$100 < n < 500$	O(n^3)
$25 < n < 100$	O(n^4)
$12 < n < 25$	O(2^n)
$1 < n < 12$	O($n!$)

CEK KONSTRAIN SOAL

Contoh Keluaran

16

Batasan

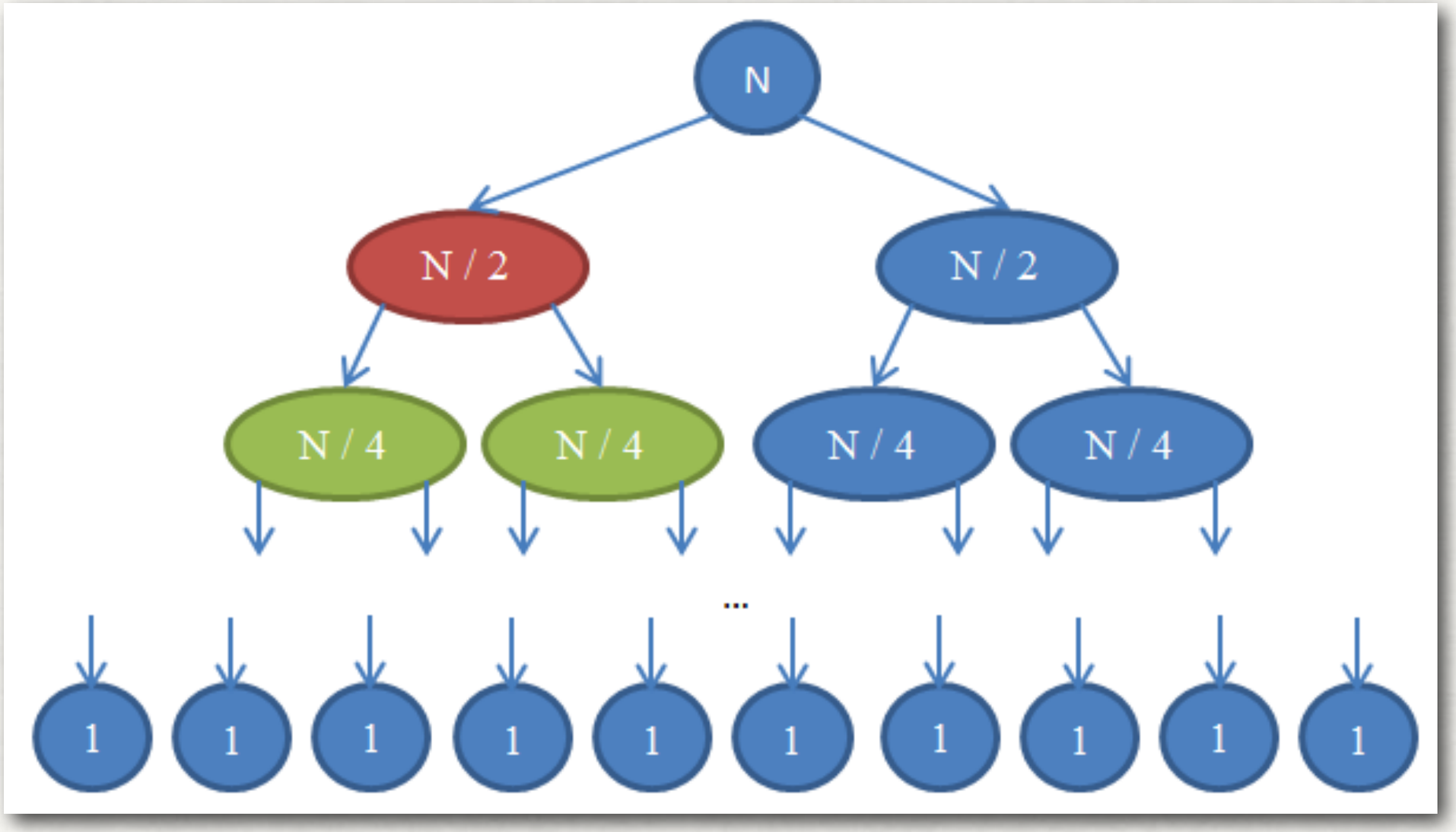
- $1 \leq N \leq 1.000.000$
- $0 \leq A_i \leq 1.000.000.000$

MERGE SORT

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

- Apa kompleksitas dari Merge Sort?

RECURSION TREE (MERGE SORT)



n

n

n

n

log n

MASTER THEOREM

- Diberikan relasi *recurrence* dalam bentuk berikut:

$$T(n) = aT(n/b) + f(n) \text{ where } a >= 1 \text{ and } b > 1$$

- *n*: ukuran *input*.
- *a*: jumlah subproblem
- *n/b*: ukuran setiap subproblem (dianggap sama)
- *f*(*n*): pekerjaan yang dilakukan di luar rekursif (misal: cost *divide* dan *merge*).

1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$

3.If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

MASTER THEOREM

$$T(n) = aT(n/b) + f(n) \text{ where } a >= 1 \text{ and } b > 1$$

- Kasus I: f(n) < $n^{\log_b a}$ —> T(n) = **Θ**($n^{\log_b a}$)
- Kasus II: f(n) > $n^{\log_b a}$ —> T(n) = **Θ**(f(*n*))
- Kasus III: f(n) = $n^{\log_b a}$ —> T(n) = **Θ**($n^{\log_b a}$ log n)
- Hitung kompleksitas dari relasi *recurrence* berikut?
  - $T(n) = 8T(n/2) + n^2$
  - $T(n) = 2T(n/2) + n^2$
  - $T(n) = 4T(n/2) + n^2$

MASTER THEOREM

T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

- Kasus I: f(n) < $n^{\log_b a}$ —> T(n) = **Θ**($n^{\log_b a}$)
- Kasus II: f(n) > $n^{\log_b a}$ —> T(n) = **Θ**(f(*n*))
- Kasus III: f(n) = $n^{\log_b a}$ —> T(n) = **Θ**($n^{\log_b a}$ log n)
- Hitung kompleksitas dari relasi *recurrence* berikut?
  - T(*n*) = 8T(*n*/2) + n^2 —> **Θ**(n^3)
  - T(*n*) = 2T(*n*/2) + n^2 —> **Θ**(n^2)
  - T(*n*) = 4T(*n*/2) + n^2 —> **Θ**(n^2 log *n*)

SPACE VS TIME TRADEOFFS

- Suatu algoritme yang dapat berjalan lebih cepat karena diberikan memori yang lebih besar.

- *Input enhancement:*
  - Sorting by counting
  - Horspool's and Boyer-Moore untuk String Matching

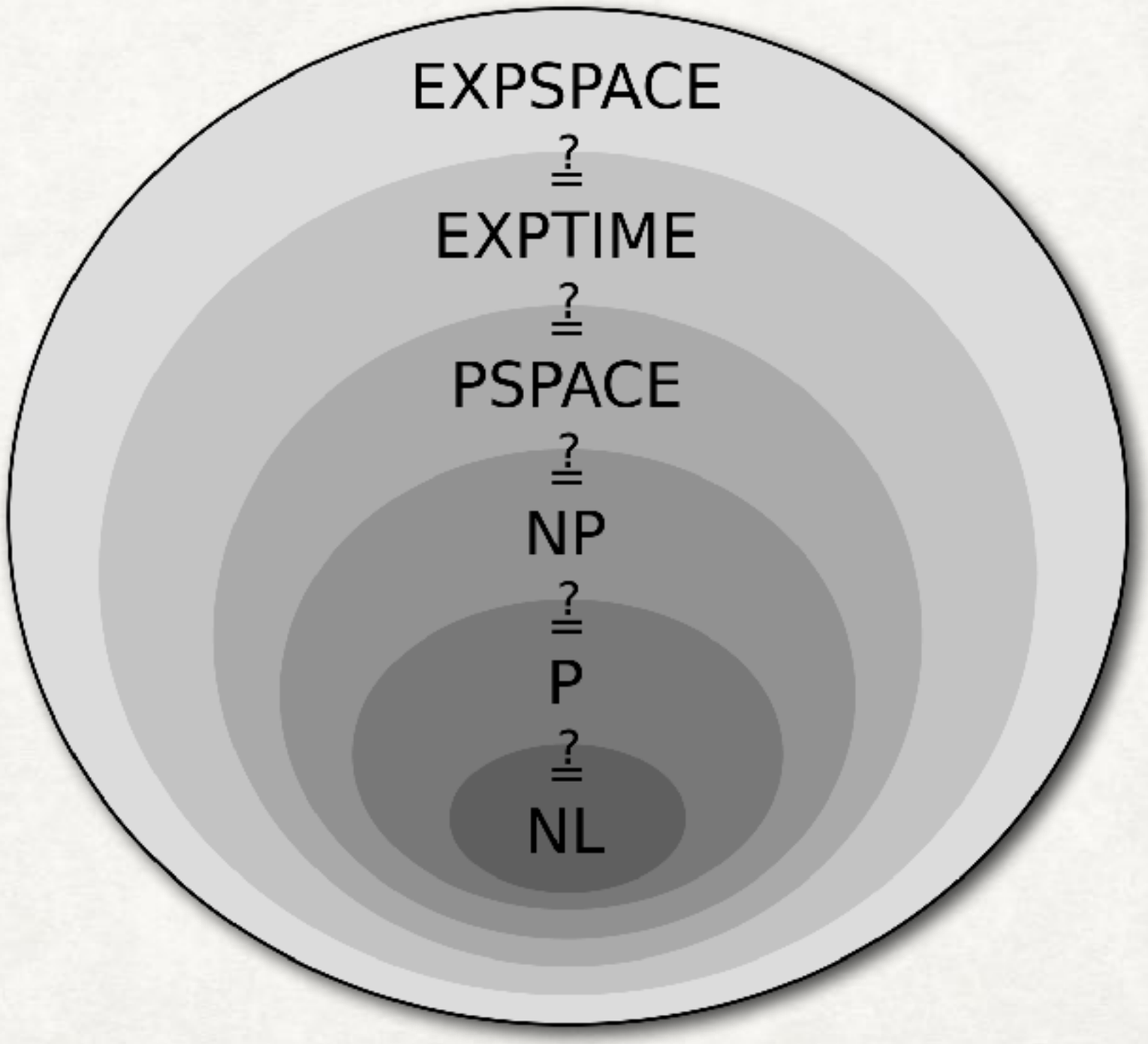
- *Prestructuring:*
  - Hashing
  - B-trees

- *Dynamic programming* (memoisasi)

SORTING BY COUNTING

- Menghitung frekuensi kemunculan setiap elemen.
- Untuk bilangan bulat, perlu memori tambahan sebesar O(*k*), *k* adalah nilai maksimum.

```
1  void counting_sort(int a[], int n, int max){
2       int count[50]={0},i,j;
3
4       for(i=0;i<n;++i)
5          count[a[i]]=count[a[i]]+1;
6
7       for(i=0;i<=max;++i)
8          for(j=1;j<=count[i];++j)
9             printf("%d ",i);
10 }
```



Bonus:

Complexity Zoo

MANFAAT COMPLEXITY ANALYSIS

- Ada patokan untuk membandingkan algoritme.
- Estimasi *running time* pada saat kompetisi.
- Kebiasaan menganalisis dahulu suatu algoritme sebelum membuat program —> mengurangi waktu sia-sia akibat memprogram algoritme yang tidak sesuai dengan *constraint*.
- Menghindari *useless optimization*.