

Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 2.

Algorithm Analysis and Growth of Functions

M. Oguzhan Kulekci, , August 29/30, 2022

- We have reviewed the asymptotic notation in the previous lecture.
- Today, we will focus on using the asymptotic notation to analyze algorithms.
- The efficiency in time and space

Time & Space Complexity of an Algorithm

- We are curious about the asymptotic complexity, i.e., what happens when input size gets really large ?
 - We aim an **ACCURATE** measure of the time/memory complexity, hence, follow the theoretical way.
 - How many instructions will be executed ?
 - **Peak memory** usage during the execution ?
- ...as a function of the **input size** !

But, which “input” ?

- Best case
- **Worst case**
- Average case

Most interesting one ! Why ?

Understanding the asymptotic notation...

a[0..n-1] contains elements to be sorted		EXECUTION COUNT	COST
		<hr/>	<hr/>
1	for (i = 1; i < n; i++) {	n	c_1
2	// Invariant: a[0..i-1] is sorted // Invariant: a[i..n-1] not yet sorted		
3	int tmp = a[i];	$n - 1$	c_2
4	for (j=i; (j>0 && a[j]>tmp); j--) {	$\sum_{j=1}^{n-1} t_j$	c_3
5	// Invariant: hole is at a[j] a[j] = a[j-1];	$\sum_{j=1}^{n-1} (t_j - 1)$	c_4
6	}		
7	a[j] = tmp;	$n - 1$	c_5
	}		

t_j is the number times a_j is compared with the previous elements.

Notice that t_j is not related with the input size n , but the values in the array.

Understanding the asymptotic notation...

Total running time
of insertion sort is

$$C(n) = c_1 \cdot n + (c_2 + c_5) \cdot (n - 1) + c_3 \cdot \sum_{j=1}^{j=n} t_j + c_4 \cdot \sum_{j=1}^{j=n} (t_j - 1)$$

Worst case: Input sequence is in decreasing order, thus $t_j = j$ and

quadratic
function

$$C(n) = c_1 \cdot n + (c_2 + c_5) \cdot (n - 1) + c_3 \cdot \frac{n(n+1)}{2} + c_4 \cdot \frac{n(n-1)}{2} = A \cdot n^2 + B \cdot n + C$$

Best case: Input sequence is in increasing order, thus $t_j = 1$ and

$$C(n) = c_1 \cdot n + (c_2 + c_5) \cdot (n - 1) + c_3 \cdot n + c_4 \cdot 0 = D \cdot n + E$$

linear
function

We don't know the values of the constants A, B, C, D, E exactly,
but we know how $C(n)$ will change according to input size n !

Understanding the asymptotic notation...

$$C(n) = An^2 + Bn + C$$

1. The most influential term of the function is the dominating (highest rank) term.

Thus, An^2 is the dominating term of $C(n)$.

2. The **leading constants** in the dominating term does not effect the function value much.

Thus, A has limited effect (?) on $C(n)$ when $n \rightarrow \infty$.

$$C(n) \in O(n^2)$$

While analyzing the time complexity of an algorithm,
we consider the dominating term, which incurs due to the **loops** !

Watch the loops...

Mystery(*n*)

r = 0

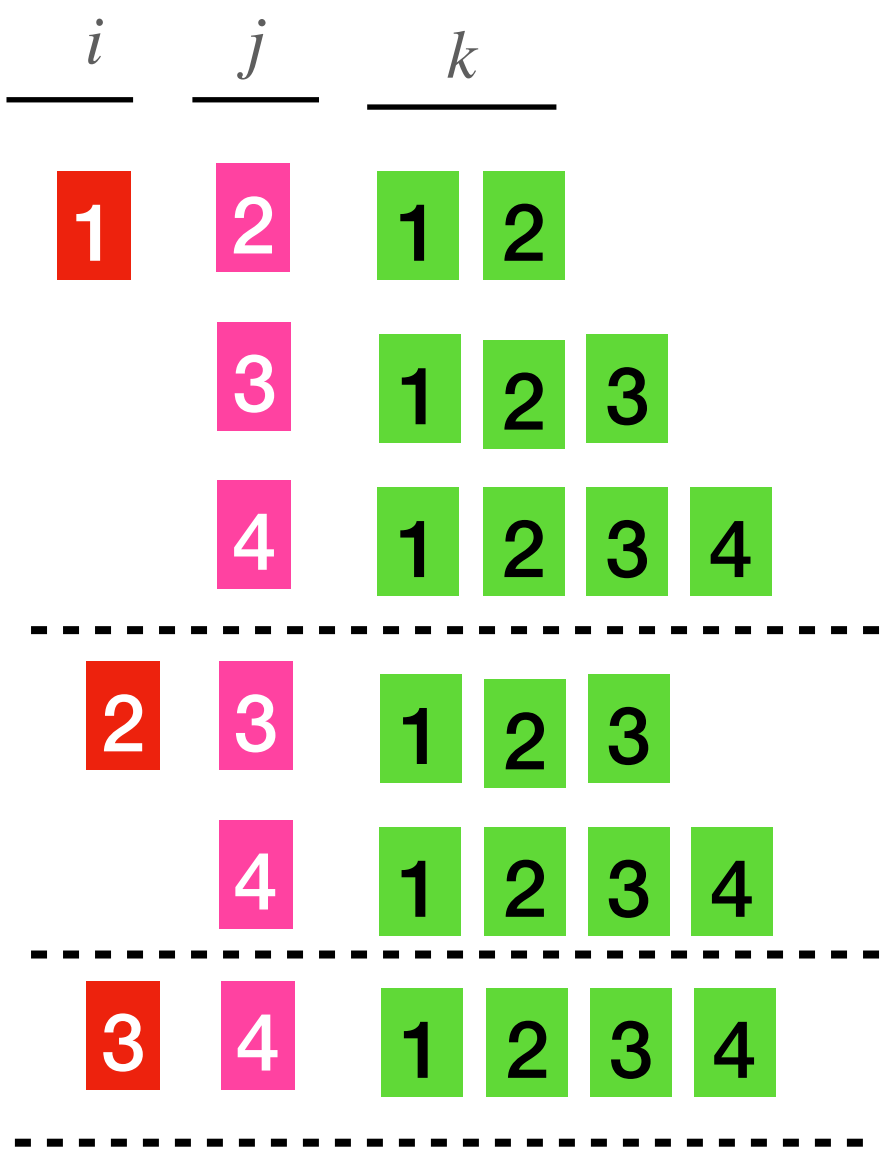
for *i* = 1 to *n* − 1 do

 for *j* = *i* + 1 to *n* do

 for *k* = 1 to *j* do

 How many times
 this line executes ? *r* = *r* + 1

 return(*r*)



Making observations with a small parameter, say *n* = 4, may help to understand better.

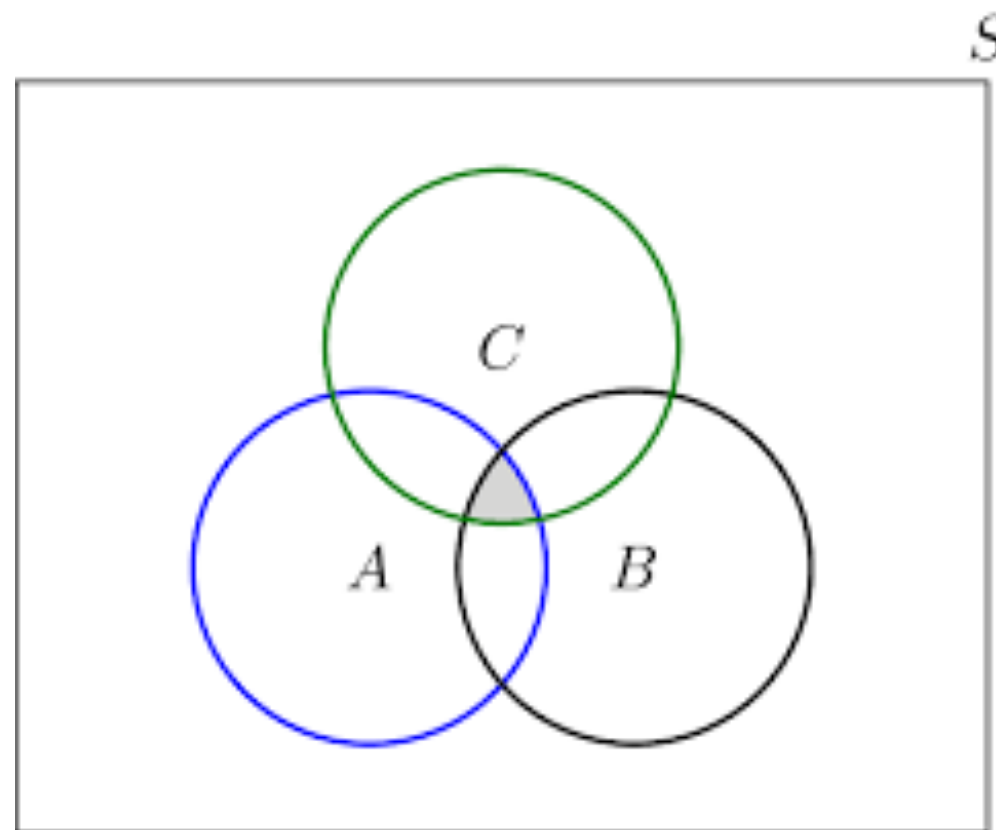
2	3	4	5	...	<i>n</i>	<i>i</i> =1
	3	4	5	...	<i>n</i>	<i>i</i> =2
		4	5	...	<i>n</i>	<i>i</i> =3
				...	<i>n</i>	
				...	<i>n</i>	
				...	<i>n</i>	<i>i</i> = <i>n</i> -1

+
1.2 + 2.3 + 3.4 + ... + (*n* − 1) . *n* =

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=1}^j 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^n j = \sum_{i=1}^{n-1} \frac{(n+i+1)(n-i)}{2} = \frac{(n-1) \cdot n \cdot (n+1)}{3}$$

Three-way Set Disjointness

Given three sequences of numbers, A, B, and C. Assume no duplicate element in each list. We aim to find if there is a number that appears in all of them.



```
1 def disjoint1(A, B, C):
2     """Return True if there is no element common to all three lists."""
3     for a in A:
4         for b in B:
5             for c in C:
6                 if a == b == c:
7                     return False           # we found a common value
8     return True                         # if we reach this, sets are disjoint
```

$O(n^3)$ solution

```
1 def disjoint2(A, B, C):
2     """Return True if there is no element common to all three lists."""
3     for a in A:
4         for b in B:
5             if a == b:
6                 for c in C:
7                     if a == c:
8                         return False       # (and thus a == b == c)
9     return True                         # we found a common value
                                         # if we reach this, sets are disjoint
```

$O(n^2)$ solution. Why ?

only check C if we found match from A and B

(and thus a == b == c)

we found a common value

if we reach this, sets are disjoint

Happy Meals, Secretary Hiring, etc...

In a village, a person becomes happy if the meal he/she is eating, is better than all previous meals he/she ate before. Assume a person eats n meals during her/his life. Can you say how many times that person becomes happy in her/his life ?

The HR department is making interviews to hire a secretary. When they interview with someone and he/she is better than the current secretary, then they fire the current, and hire the interviewed candidate. If there are n candidates, how many times firing/hiring occurs?

All are same actually. Assume an array encodes the quality scores, and we are curious about the maximum.

How many times we change the maximum?

$$f(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

$$f(n) \in O(\log n)$$

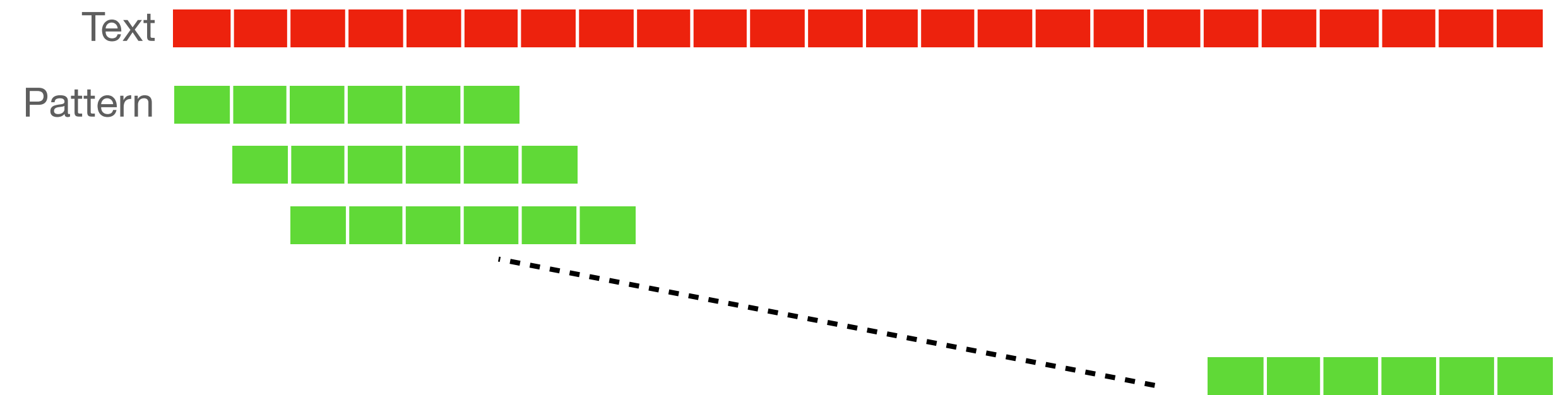
nth harmonic number

More than one parameter ...

```
int findmatch(char *p, char *t) {
    int i, j;           /* counters */
    int plen, tlen;     /* string lengths */

    plen = strlen(p);
    tlen = strlen(t);

    for (i = 0; i <= (tlen-plen); i = i + 1) {
        j = 0;
        while ((j < plen) && (t[i + j] == p[j])) {
            j = j + 1;
        }
        if (j == plen) {
            return(i); /* location of the first match */
        }
    }
    return(-1);        /* there is no match */
}
```



What can be the worst case ?

The worst text and the worst pattern to search on it.

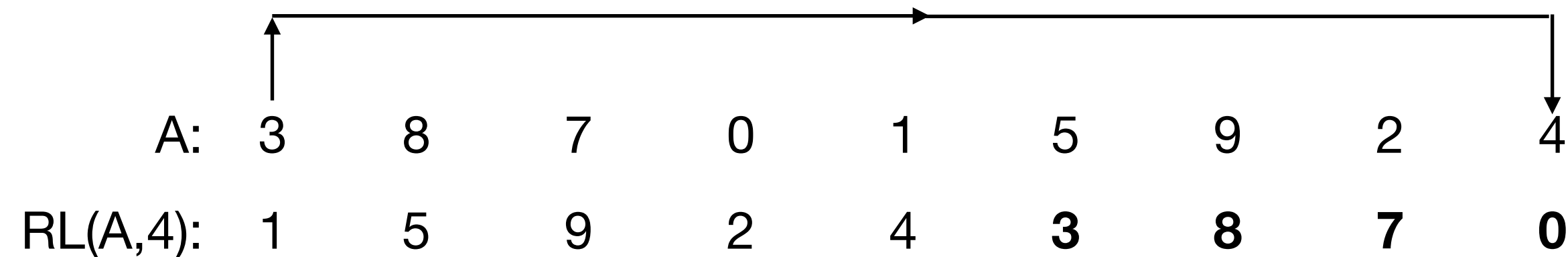
Assume the lengths of text and pattern are n and m , respectively.

How many times the loops execute?
What is the worst case time complexity?

$$O(n \cdot m)$$

Time-memory trade off

RL(A,d): **Rotate** a given array by **d** positions to the **left**.



```
for (i=1 to d)
  t = a[1]
  for (j=1 to n-1)
    a[j] = a[j+1]
  a[n] = t
```

$O(n \cdot d)$ -time
 $O(1)$ -space

Version 1. Repeat RL(A,1) d times

```
t[1..d] = a[1..d]
for (i=d+1 to n-1)
  a[i-d] = a[i]
a[n-d+1..n] = t[1..d]
```

$O(n)$ -time
 $O(d)$ -space

Version 2. Save first d items,
shift remaining to the left by d,
paste saved items to the rightmost

```
for (i=1 to d/2) //reverse(a[1..d])
  t = a[i]
  a[i]=a[d+1-i]
  a[d-i+1]=t
for (i=d+1 to d+((n-d)/2))
  t = a[i] //reverse(a[d+1..n])
  a[i]=a[n-i+1+d]
  a[n-i+1+d]=t
for (i=1 to n/2) //reverse(a[1..n])
  t = a[i]
  a[i]=a[n-i+1]
  a[n-i+1]=t
```

$O(n)$ -time
 $O(1)$ -space

Version 3. Reverse initial d items,
reverse the rest, and finally, reverse everything

But accesses each location twice !



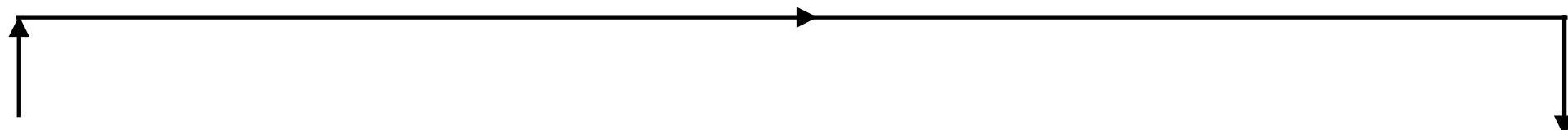
It is better to solve one problem five
different ways, than to solve five
problems one way.

— George Polya —

AZ QUOTES

Time-memory trade off

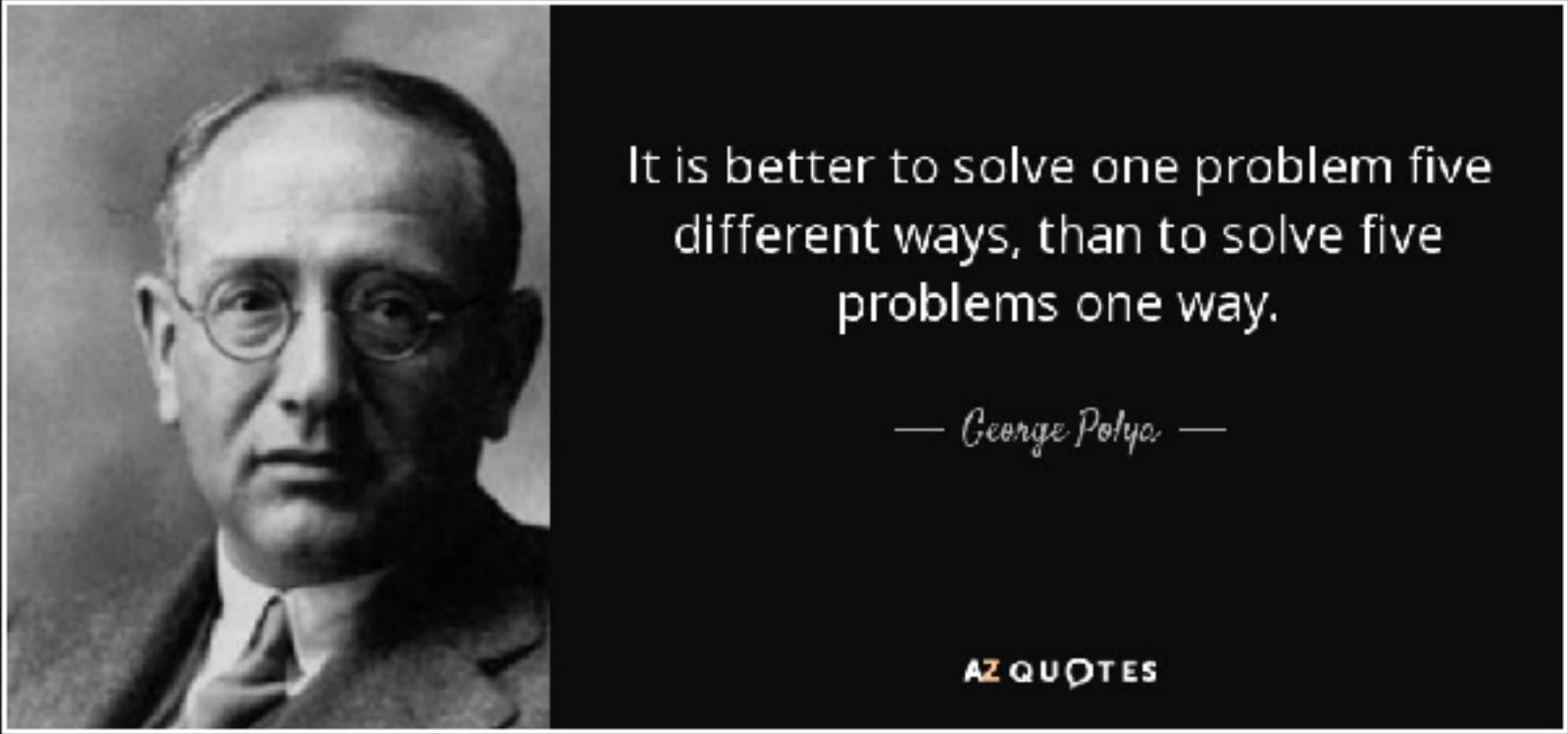
RL(A,d): **Rotate** a given array by **d** positions to the **left**.

									
Position:	1	2	3	4	5	6	7	8	9
A:	a	b	c	d	e	f	g	h	i
RL(A,4):	e	f	g	h	i	a	b	c	d
RL(A,3):	d	e	f	g	h	i	a	b	c

$RL(A[1..9],4) : 1 \leftarrow 5 \leftarrow 9 \leftarrow 4 \leftarrow 8 \leftarrow 3 \leftarrow 7 \leftarrow 2 \leftarrow 6$

$RL(A[1..9],3) : \left. \begin{array}{l} 1 \leftarrow 4 \leftarrow 7 \leftarrow 1 \\ 2 \leftarrow 5 \leftarrow 8 \leftarrow 2 \\ 3 \leftarrow 6 \leftarrow 9 \leftarrow 3 \end{array} \right\} \begin{array}{l} \text{How many iterations?} \\ GCD(n, d) \end{array}$

Which solution would you prefer ?



$O(n)$ -time
 $O(1)$ -space

Version 4. In $GCD(n,d)$ iterations,
perform chains of moves!

```
for (i=1 to GCD(n,d))
  t = a[i]
  j = i
  while (1)
    k = j+d
    if (k>n) k=k-n
    if (k==i) break
    a[j] = a[k]
    j = k
  a[j] = t
```

Moves each location only ONCE !

An Experimental Approach to Analyze the Complexity

Assume we are given a black-box algorithm, e.g., we do not have access to its code. How can we test its complexity ?

```
$ time a.out 2000
real    5.85s
$ time a.out 4000
real    21.65s
$ time a.out 8000
real    85.11s
```

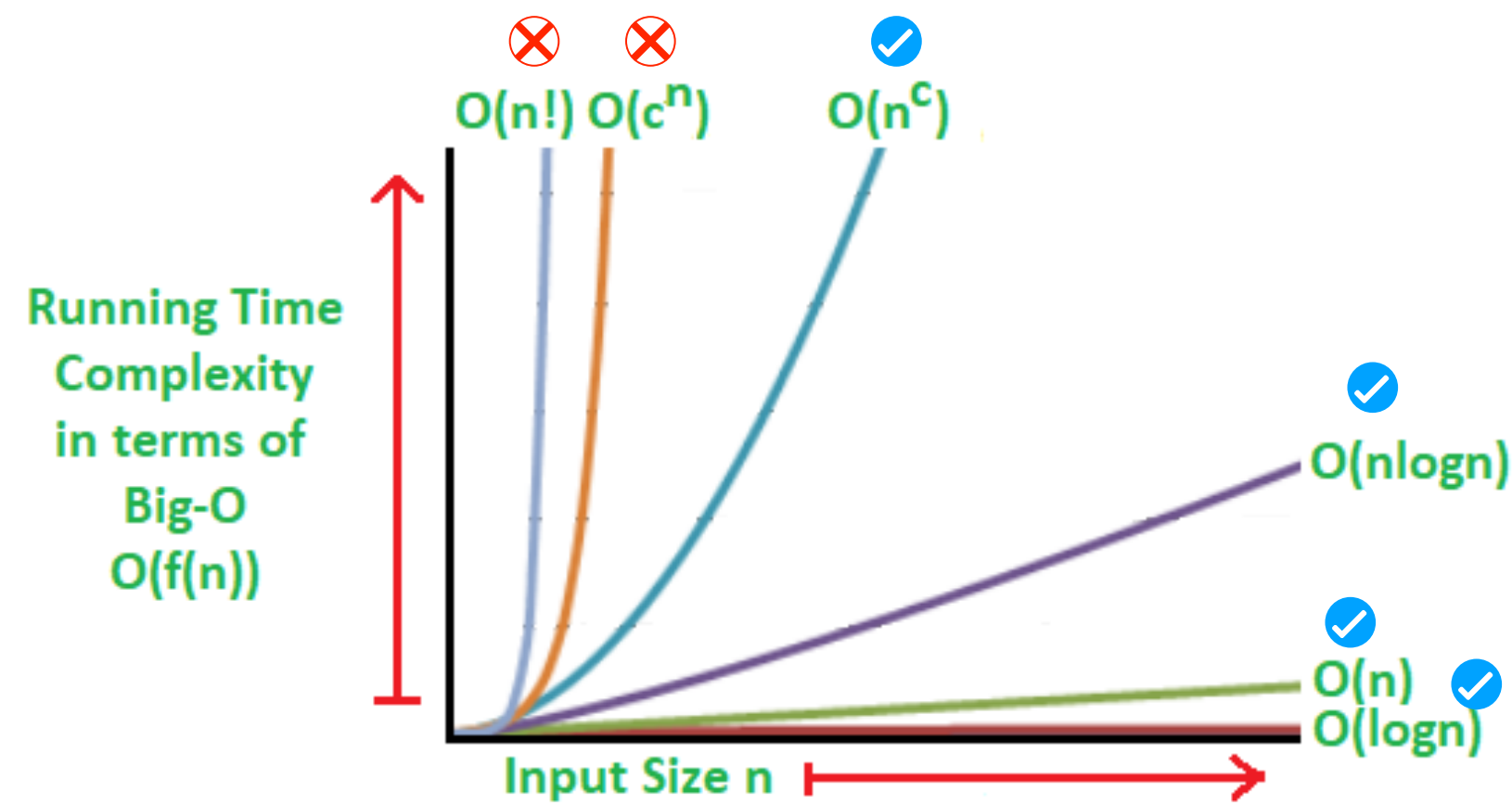
See Catherine Mcgeoch, A guide to experimental Algorithmics

Doubling experiments:

How much does the indicator (time/memory) change when you double the size of the input, e.g., $n = N, 2N, 4N, 8N, \dots$?

- If indicator does not change then the effect is constant. No need to investigate.
- If increment by a constant, then the relation is logarithmic, $O(\log n)$.
- If the indicator doubles as well, then the relation is linear, $O(n)$.
- $O(n \log n)$?
- $O(n^2)$?

Performance comparison...



PATHS, TREES, AND FLOWERS

JACK EDMONDS

2. Digression. An explanation is due on the use of the words “efficient algorithm.” First, what I present is a conceptual description of an algorithm and not a particular formalized algorithm or “code.”

For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, “efficient” means “adequate in operation or performance.” This is roughly the meaning I want—in the sense that it is conceivable for maximum matching to have no efficient algorithm. Perhaps a better word is “good.”

I am claiming, as a mathematical result, the existence of a *good* algorithm for finding a maximum cardinality matching in a graph.

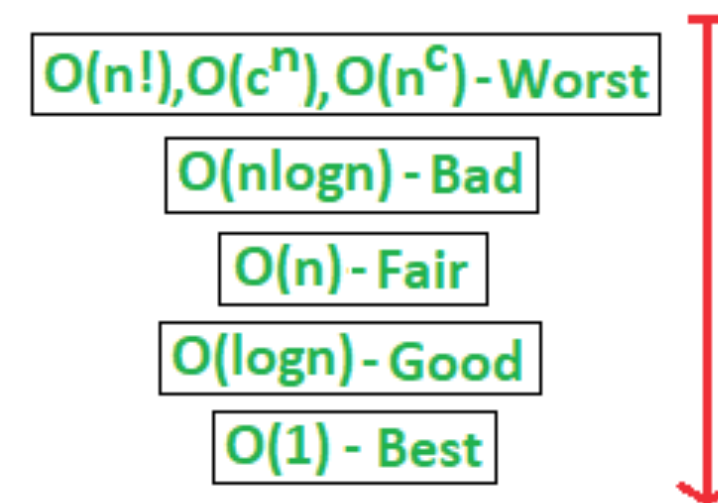
There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph.

The mathematical significance of this paper rests largely on the assumption that the two preceding sentences have mathematical meaning. I am not prepared to set up the machinery necessary to give them formal meaning, nor

Edmunds’65:

Definition of Efficient Algorithm

Is it efficient ?



$O(1)$: Constant time/space

$O(\log n)$: Logarithmic

$O(n)$: Linear

$O(n \cdot \log n)$: Log-Linear

$O(n^k)$: **Polinomial**

$O(k^n)$: Exponential

$O(n!)$: Factorial

Theoretically, polynomial-time solutions are efficient.

How about in practice ???

NP-hard, NP-complete, etc...
another world !

Don't forget the elegance?

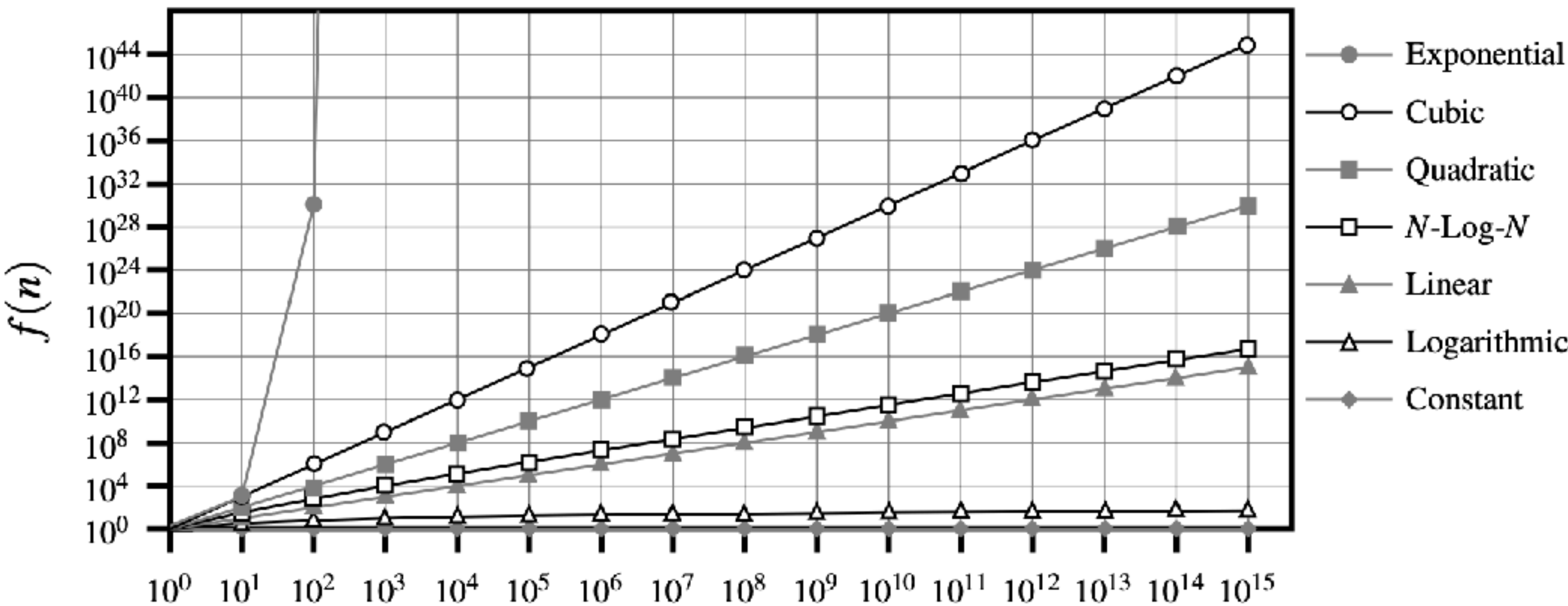
Elegance is beauty that shows unusual effectiveness and simplicity....

Performance comparison...

n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Skiena

constant	logarithm	linear	n -log- n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Goodrich et al.

Questions, comments ?

- We are done with the asymptotic notation and basic algorithm analysis.
- Chapter 1 and 2 from Skiena. Please also check the related chapters of other books to improve our understanding.
- Next lecture, we will start reviewing the basic data structures, arrays, linked list, stack, queue, tree, etc...