

การลดเพื่อเอาชนะ Decrease and Conquer

การอบรมคอมพิวเตอร์โอลิมปิก สอวน. ค่ายที่ 2 ปีการศึกษา 2563



อ.ดร.สุภาพณา บุญชู

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี

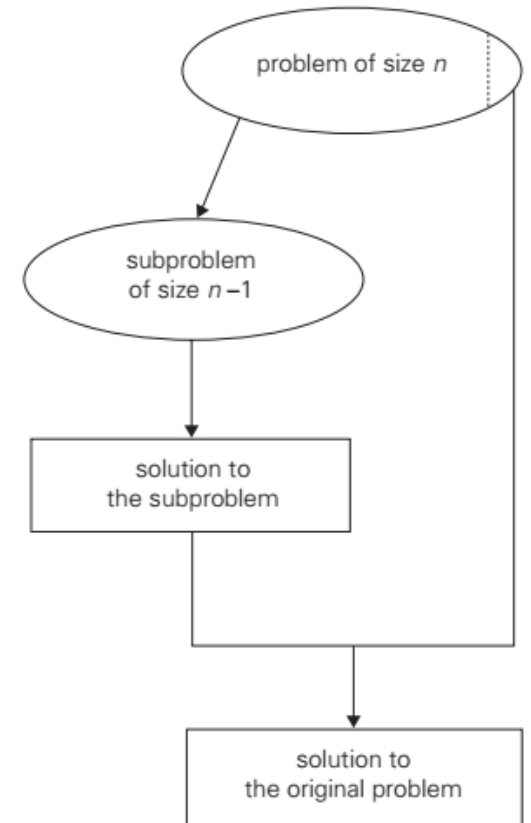
มหาวิทยาลัยธรรมศาสตร์

Decrease-and-Conquer

- The **decrease-and-conquer** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- Once such a relationship is established, it can be exploited either **top down** or **bottom up**.
 - The **top down** leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive.
 - The **bottom-up** variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the **incremental approach**.
- There are three major variations of decrease-and-conquer:
 - decrease by a constant
 - decrease by a constant factor
 - variable size decrease

Decrease-by-a-constant

- In the **decrease-by-a-constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.
- Typically, this constant is equal to one, although other constant size reductions do happen occasionally.
- Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a nonnegative integer.
 - The relationship between a solution to an instance of size n and an instance of size $n - 1$ is obtained by the obvious formula $a^n = a^{n-1}a$
 - So the function $f(n) = a^n$ can be computed either **“top down”** by using its recursive definition or **“bottom up”** by multiplying 1 by a n times.



Decrease-by-a-constant-factor

- The **decrease-by-a-constant-factor** technique suggests reducing a problem instance by **the same constant factor** on each iteration of the algorithm.
- In most applications, this constant factor is equal to two.
- For an example, let us revisit the exponentiation a^n problem.
 - If the instance of size n is to compute a^n , the instance of half its size is to compute $a^{\frac{n}{2}}$, with the obvious relationship between the two: $a^n = \left(a^{\frac{n}{2}}\right)^2$.
 - But since we consider here instances with integer exponents only, the former does not work for odd n .

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Guess time complexity?

$\Theta(\log n)$

Variable-size-decrease

- In the **variable-size-decrease** variety of decrease-and-conquer, the size-reduction pattern **varies** from one iteration of an algorithm to another.
- Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation.

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

แบบฝึกหัด

ให้นักเรียนเขียนโปรแกรมเพื่อหาค่าของฟังก์ชัน $f(n) = a^n$ และ
เปรียบเทียบเวลาการรันโปรแกรมของการเขียนโปรแกรม 2 แบบต่อไปนี้

แบบที่ 1 ใช้วิธี **Brute Force**

แบบที่ 2 ใช้วิธี **Decrease and Conquer**

พัก 10 นาที

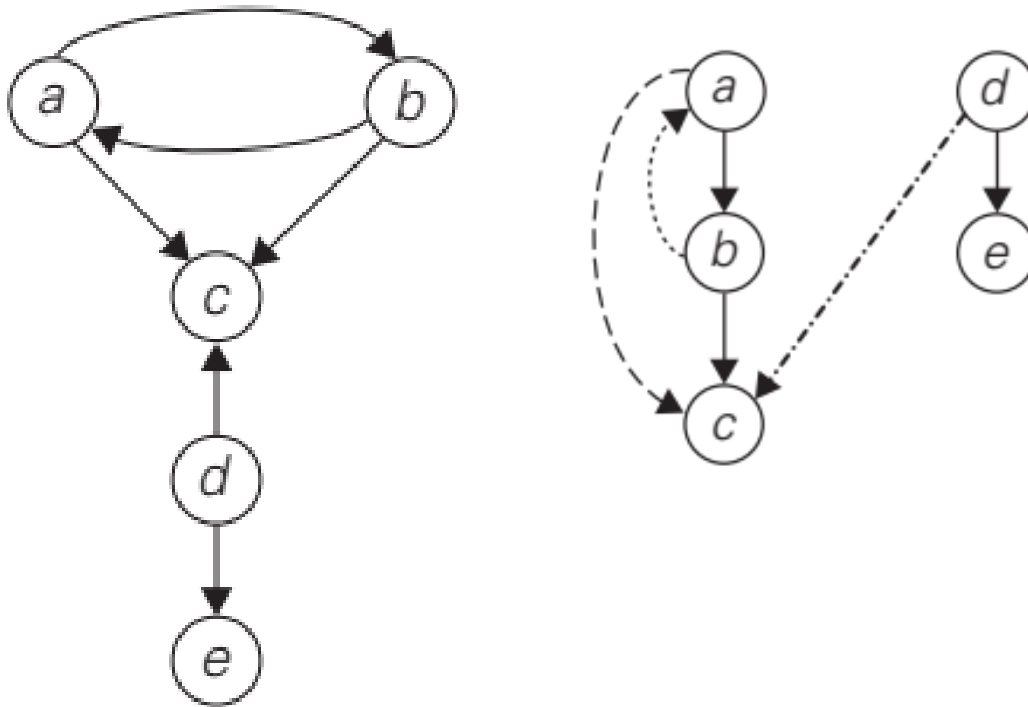
Insertion Sort



Topological Sorting

Topological Sorting

- What is directed graph, or digraph?
 - It is a graph with directions specified for all its edges.
 - The adjacency matrix and adjacency lists are still two principal means of representing a digraph.



Four types of edges
possible in a DFS forest:

Tree edges: *ab, bc, de*

Back edge : *ba*

Forward edge: *ac*

Cross edge: *dc*

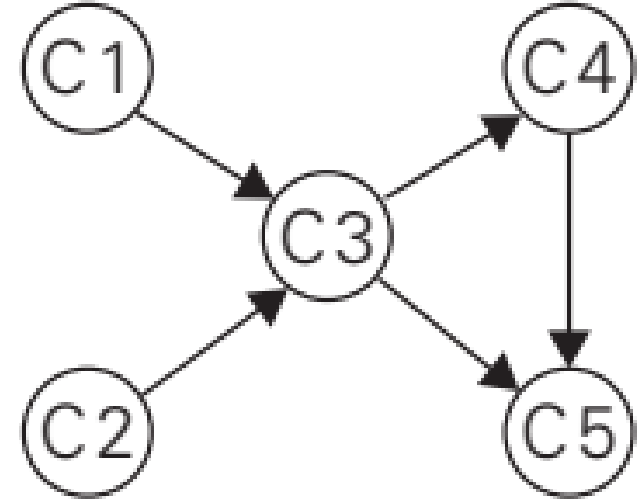
DAG: Directed Acyclic Graph?

Topological Sorting

- Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.
 - C1 and C2 have no prerequisites,
 - C3 requires C1 and C2,
 - C4 requires C3,
 - C5 requires C3 and C4.
 - The student can take only one course per term.
 - In which order should the student take the courses?
- Can this problem modeled by a digraph?
 - How?
 - Vertices -> Courses, Edge Directions -> Prerequisites

Topological Sorting

C1 and C2 have no prerequisites,
C3 requires C1 and C2,
C4 requires C3,
C5 requires C3 and C4.



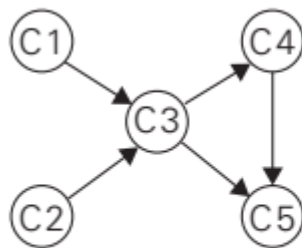
In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

This problem is called *topological sorting*.

? Is it possible to find the solution for the a digraph has a directed cycle.

Topological Sorting

- There are two efficient algorithms that both verify whether a digraph is a **dag** and, if it is, produce an ordering of vertices that solves the topological sorting problem.
 - First, perform a DFS traversal and note the order in which vertices become dead-ends. (i.e., popped off the traversal stack)
 - **Reversing** this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal.
 - If a **back edge** has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.



(a)

C5₁
C4₂
C3₃
C1₄ C2₅

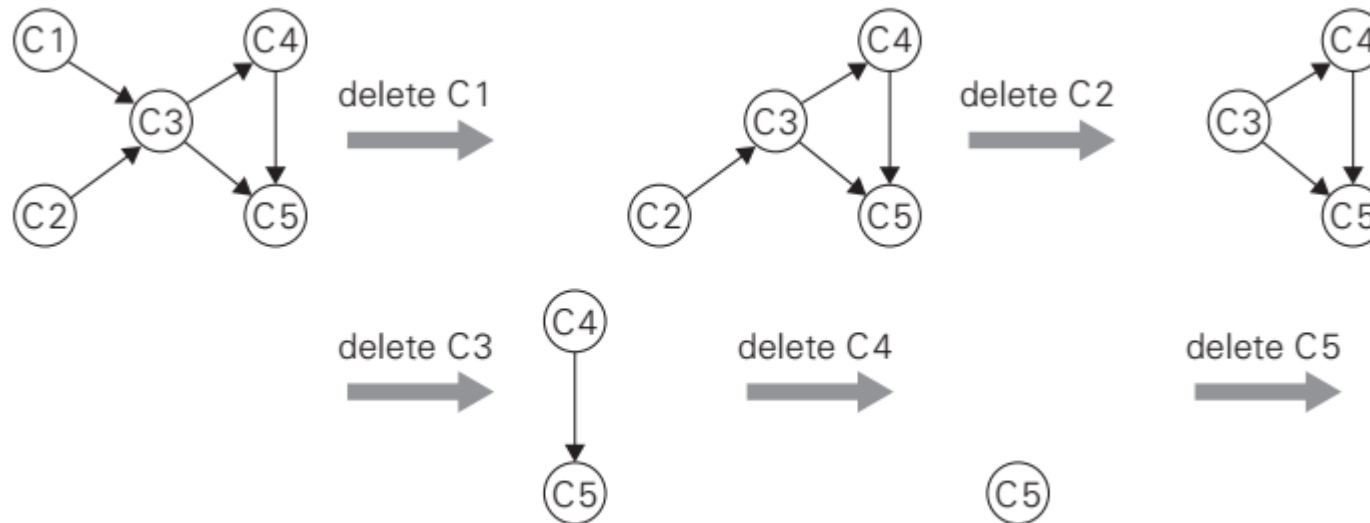
(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:
C2 C1 → C3 → C4 → C5

(c)

Topological Sorting

- The second algorithm is based on a direct implementation of the **decrease-(by one)-and-conquer** technique: repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.



The solution obtained is C1, C2, C3, C4, C5

Exercise

- ให้นักศึกษาเขียนโปรแกรมสำหรับแก้ปัญหา Topological Sorting โดยใช้ Decrease and Conquer method
- 20 นาที

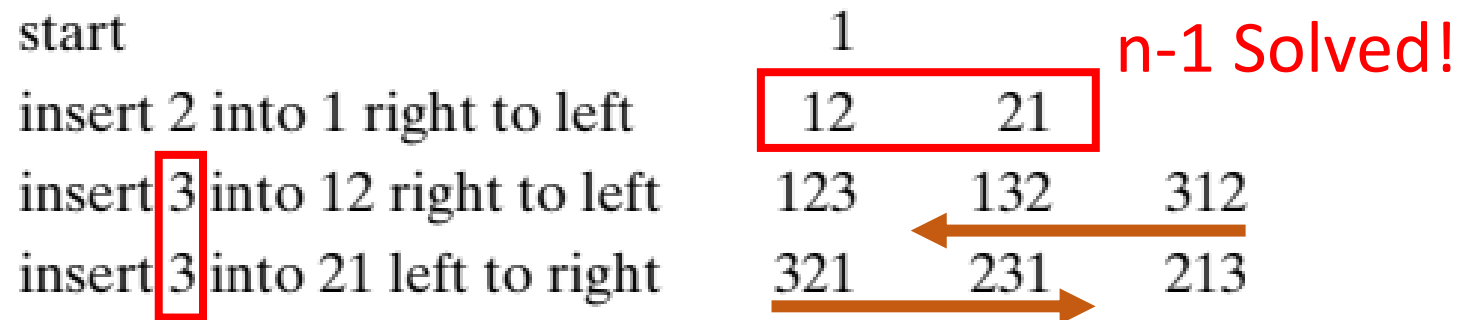
Algorithms for Generating Combinatorial Objects

Combinatorial Objects

- The most important types of *combinatorial objects* are permutations, combinations, and subsets of a given set.
- Mathematicians, of course, are primarily interested in different counting formulas; we should be grateful for such formulas because they tell us how many items need to be generated.
- In particular, they warn us that the number of combinatorial objects typically grows **exponentially** or even faster as a function of the problem size.
- But our primary interest here lies in algorithms for generating combinatorial objects, not just in counting them.

Generating Permutations

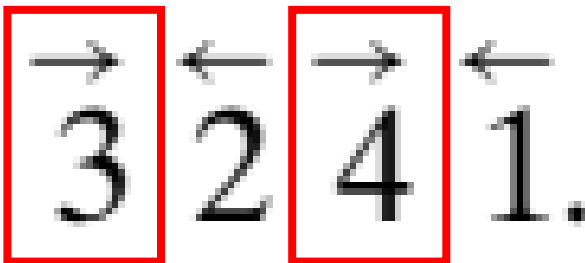
- we assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n ; more generally, they can be interpreted as indices of elements in an n -element set $\{a_1, \dots, a_n\}$.
- Assuming that the smaller problem is solved, we can get a solution to the larger one by inserting n in each of the n possible positions among elements of every permutation of $n - 1$ elements.



Generating Permutations

- It is possible to get the same ordering of permutations of n elements **without** explicitly generating permutations for smaller values of n .
- It can be done by associating a **direction** with each element k in a permutation.
- The element k is said to be **mobile** in such an arrow-marked permutation if its arrow points to a smaller number adjacent to it.

3 and 4 are mobile



The diagram shows a permutation of the numbers 1, 2, 3, and 4. Above each number is a small arrow indicating its direction: 3 has a right arrow, 2 has a left arrow, 4 has a right arrow, and 1 has a left arrow. The numbers 3 and 4 are each enclosed in a red rectangular box. The text "3 and 4 are mobile" is written in red to the left of the boxes.

Johnson-Trotter algorithm

$$\Theta(n!)$$

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations

//Input: A positive integer n

//Output: A list of all permutations of $\{1, \dots, n\}$

initialize the first permutation with $\overleftarrow{1} \overleftarrow{2} \dots \overleftarrow{n}$

while the last permutation has a mobile element **do**

 find its largest mobile element k

 swap k with the adjacent element k 's arrow points to

 reverse the direction of all the elements that are larger than k

 add the new permutation to the list

$\overleftarrow{1} \overleftarrow{2} \overleftarrow{3} \quad \overleftarrow{1} \overleftarrow{3} \overleftarrow{2} \quad \overleftarrow{3} \overleftarrow{1} \overleftarrow{2} \quad \overrightarrow{3} \overleftarrow{2} \overleftarrow{1} \quad \overleftarrow{2} \overrightarrow{3} \overleftarrow{1} \quad \overleftarrow{2} \overleftarrow{1} \overrightarrow{3}.$

Generating Subsets

- The decrease-by-one idea is immediately applicable to this problem.
- All subsets of $A = \{a_1, \dots, a_n\}$ can be divided into two groups:
 - those that do not contain a n and those that do.
 - The former group is nothing but all the subsets of $\{a_1, \dots, a_{n-1}\}$, while each and every element of the latter can be obtained by **adding a_n** to a subset of $\{a_1, \dots, a_{n-1}\}$.

n	subsets $\{a_1, a_2, a_3\}$ (in squashed order)							
0	\emptyset							
1	\emptyset	$\{a_1\}$						
2	\emptyset	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	\emptyset	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

bit strings	000	001	010	011	100	101	110	111
subsets	\emptyset	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Binary reflected Gray code

ALGORITHM $BRGC(n)$

//Generates recursively the binary reflected Gray code of order n

//Input: A positive integer n

//Output: A list of all bit strings of length n composing the Gray code

if $n = 1$ make list L containing bit strings 0 and 1 in this order

else generate list $L1$ of bit strings of size $n - 1$ by calling $BRGC(n - 1)$

 copy list $L1$ to list $L2$ in reversed order

 add 0 in front of each bit string in list $L1$

 add 1 in front of each bit string in list $L2$

 append $L2$ to $L1$ to get list L

return L

Exercise

- STL of C++: Generating Permutation:

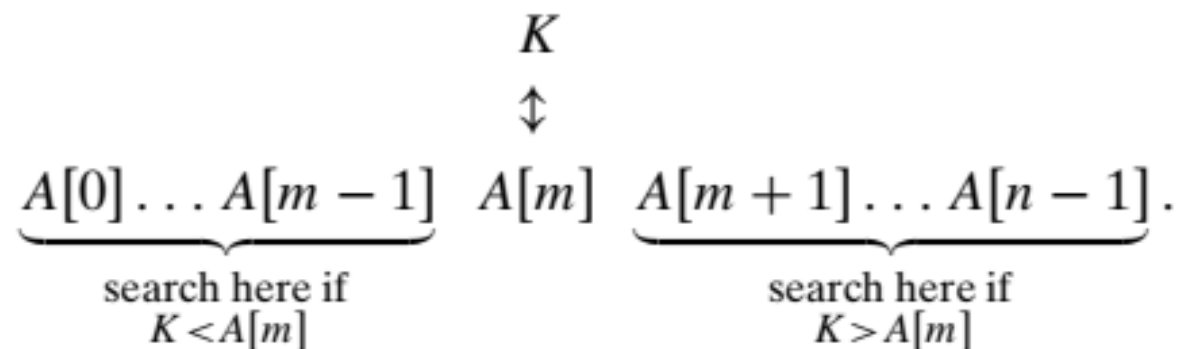
Decrease-by-a-Constant-Factor Algorithms

Decrease-by-a-Constant-Factor Algorithms

- The most important and well-known of them is **binary search**.
- **Decrease-by-a-constant-factor** algorithms usually run in logarithmic time, and, being very efficient, do not happen often; a reduction by a factor other than two is especially rare.

Binary Search

Array A is assumed to be sorted.



ALGORITHM *BinarySearch*($A[0..n-1]$, K)

//Implements nonrecursive binary search

//Input: An array $A[0..n-1]$ sorted in ascending order and

// a search key K

//Output: An index of the array's element that is equal to K

// or -1 if there is no such element

$l \leftarrow 0$; $r \leftarrow n - 1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$ $r \leftarrow m - 1$

else $l \leftarrow m + 1$

return -1

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l, m	r				

Variable-Size-Decrease Algorithms

Computing a Median and the Selection Problem

- The **selection problem** is the problem of finding the k th smallest element in a list of n numbers.
 - This number is called the **k^{th} order statistic**.
- How to find the case that $k = 1$ and $k = n$?
- What if $k = \lfloor n/2 \rfloor$?
 - This is to find the **median**.
- Obviously, we can find the k^{th} smallest element in a list by sorting the list first and then selecting the k^{th} element in the output of a sorting algorithm. The time of such an algorithm is determined by the efficiency of the sorting algorithm used.
 - Thus, with a fast sorting algorithm such as mergesort (discussed in the next chapter), the algorithm's efficiency is in $O(n \log n)$.

Lomuto partitioning

- Indeed, we can take advantage of the idea of partitioning a given list around some value p of, say, its first element.
- In general, this is a rearrangement of the list's elements so that the left part contains all the elements smaller than or equal to p , followed by the pivot p itself, followed by all the elements greater than or equal to p .

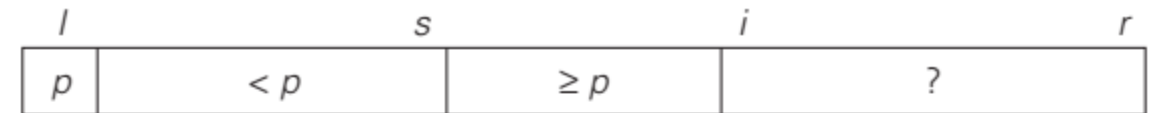


Lomuto partitioning

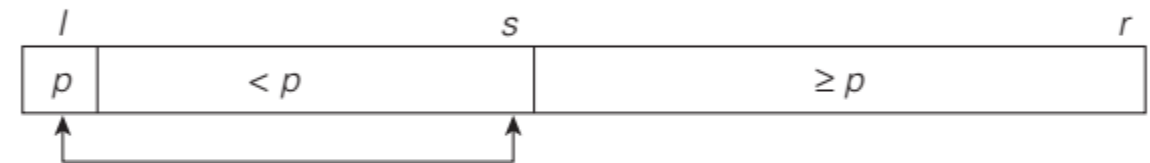
- We may think of an array—or, more generally, a subarray $A[l..r]$ ($0 \leq l \leq r \leq n - 1$)—under consideration as composed of three contiguous segments.

ALGORITHM *LomutoPartition*($A[l..r]$)

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray  $A[l..r]$  of array  $A[0..n - 1]$ , defined by its left and right
//       indices  $l$  and  $r$  ( $l \leq r$ )
//Output: Partition of  $A[l..r]$  and the new position of the pivot
 $p \leftarrow A[l]$ 
 $s \leftarrow l$ 
for  $i \leftarrow l + 1$  to  $r$  do
    if  $A[i] < p$ 
         $s \leftarrow s + 1$ ;  $\text{swap}(A[s], A[i])$ 
 $\text{swap}(A[l], A[s])$ 
return  $s$ 
```



(a)



(b)



(c)

Quick Select

- How can we take advantage of a list partition to find the k^{th} smallest element in it?
 - Let s be the partition's split position i.e., the index of the array's element occupied by the pivot after partitioning.
 - If $s = k - 1$, pivot p itself is obviously the k^{th} smallest element, which solves the problem.
 - If $s > k - 1$, the k th smallest element in the entire array can be found as the k^{th} smallest element in the left part of the partitioned array.
 - If $s < k - 1$, it can be found as the $(k - s)$ th smallest element in its right part.

ALGORITHM *Quickselect*($A[l..r]$, k)

```
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray  $A[l..r]$  of array  $A[0..n - 1]$  of orderable elements and
//      integer  $k$  ( $1 \leq k \leq r - l + 1$ )
//Output: The value of the  $k$ th smallest element in  $A[l..r]$ 
 $s \leftarrow \text{LomutoPartition}(A[l..r])$  //or another partition algorithm
if  $s = k - 1$  return  $A[s]$ 
else if  $s > l + k - 1$  Quickselect( $A[l..s - 1]$ ,  $k$ )
else Quickselect( $A[s + 1..r]$ ,  $k - 1 - s$ )
```

Quick Select

	0	1	2	3	4	5	6	7	8
<i>s</i>		<i>i</i>							
4		1	10	8	7	12	9	2	15
		<i>s</i>	<i>i</i>						
4		1	10	8	7	12	9	2	15
		<i>s</i>						<i>i</i>	
4		1	10	8	7	12	9	2	15
			<i>s</i>					<i>i</i>	
4		1	2	8	7	12	9	10	15
			<i>s</i>						<i>i</i>
4		1	2	8	7	12	9	10	15
2		1	4	8	7	12	9	10	15



<i>s</i>	<i>i</i>				
8	7	12	9	10	15
	<i>s</i>	<i>i</i>			
8	7	12	9	10	15
	<i>s</i>				<i>i</i>
8	7	12	9	10	15
7	8	12	9	10	15

End

- Q&A