CHAPTER 16 EFFICIENT ALGORITHMS

UNIVERSITEIT GENT

OBJECTIVES

- To estimate algorithm efficiency using the Big O notation (§16.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§16.2).
- To determine the complexity of various types of algorithms (§16.3).
- To analyze the binary search algorithm (§16.4.1).
- To analyze the selection sort algorithm (§16.4.2).
- To analyze the insertion sort algorithm (§16.4.3).
- ~~To analyze the Towers of Hanoi algorithm (§16.4.4).~~
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, exponential) (§16.4.5).
- ~~To design efficient algorithms for finding Fibonacci numbers (§16.5).~~
- ~~To design efficient algorithms for finding gcd (§16.6).~~
- ~~To design efficient algorithms for finding prime numbers (§16.7).~~
- ~~To design efficient algorithms for finding a closest pair of points (§16.8).~~
- ~~To solve the Eight Queens problem using the backtracking approach (§16.9).~~
- ~~To design efficient algorithms for finding a convex hull for a set of points (§16.10).~~

EXECUTING TIME

Suppose two algorithms perform the same task such as search (linear search vs. binary search) and sorting (selection sort vs. insertion sort). Which one is better? One possible approach to answer this question is to implement these algorithms in Java and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

UNIVERSITEIT GENT

GROWTH RATE

It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their growth rates.

BIG O NOTATION

Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires n comparisons for an array of size n. If the key is in the array, it requires n/2 comparisons on average. The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of n. Computer scientists use the Big O notation to abbreviate for "order of magnitude." Using this notation, the complexity of the linear search algorithm is O(n), pronounced as "order of n."

UNIVERSITEIT
GENT

BEST, WORST, AND AVERAGE CASES

For the same input size, an algorithm’s execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case. An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

UNIVERSITEIT
GENT

IGNORING MULTIPLICATIVE CONSTANTS

– The linear search algorithm requires n comparisons in the worst-case and n/2 comparisons in the average-case. Using the Big O notation, both cases require O(n) time. The multiplicative constant (1/2) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for n/2 or 100n is the same as n, i.e., O(n) = O(n/2) = O(100n).

n \ $f(n)$	n	$n/2$	$100n$
100	100	50	10000
200	200	100	20000
$f(200) / f(100)$	2	2	2

IGNORING NON-DOMINATING TERMS

Consider the algorithm for finding the maximum number in an array of n elements. If n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number. In general, it takes n-1 times of comparisons to find maximum number in a list of n elements. Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm’s efficiency. As n grows larger, the n part in the expression n-1 dominates the complexity. The Big O notation allows you to ignore the non-dominating part (e.g., -1 in the expression n-1) and highlight the important part (e.g., n in the expression n-1). So, the complexity of this algorithm is O(n).

UNIVERSITEIT
GENT

USEFUL MATHEMATIC SUMMATIONS

$$1+2+3+....+(n-1)+n=\frac{n(n+1)}{2}$$

$$a^0+a^1+a^2+a^3+....+a^{(n-1)}+a^n=\frac{a^{n+1}-1}{a-1}$$

$$2^0+2^1+2^2+2^3+....+2^{(n-1)}+2^n=\frac{2^{n+1}-1}{2-1}$$

UNIVERSITEIT GENT

EXAMPLES: DETERMINING BIG-O

- Repetition
- Sequence
- Selection
- Logarithm

UNIVERSITEIT
GENT

REPETITION: SIMPLE LOOPS

executed
n times

```
for i in range(n):
    k = k + 5
```

constant time

Time Complexity

T(n) = (a constant c) * n = cn = **O(n)**

Ignore multiplicative constants (e.g., “c”).

PerformanceTest

UNIVERSITEIT
GENT

REPETITION: NESTED LOOPS

executed
n times

```
for i in range(n):
    for j in range(n):
        k = k + i + j
```

inner loop
executed
n times

constant time

Time Complexity

$$T(n) = (\text{a constant } c) * n * n = cn^2 = O(n^2)$$

Ignore multiplicative constants (e.g., “c”).

REPETITION: NESTED LOOPS

executed
n times

```
for i in range(n):
    for i in range(i):
        k = k + i + j
```

inner loop
executed
i times

constant time

Time Complexity

$T(n) = c + 2c + 3c + 4c + \ldots + nc = cn(n+1)/2 =$
$(c/2)n^2 + (c/2)n = O(n^2)$

Ignore non-dominating terms

Ignore multiplicative constants

UNIVERSITEIT
GENT

REPETITION: NESTED LOOPS

executed
n times

```
for i in range(n):
    for i in range(20):
        k = k + i + j
```

inner loop
executed
20 times

constant time

Time Complexity

T(n) = 20 * c * n = O(n)

*Ignore multiplicative constants (e.g., 20*c)*

UNIVERSITEIT
GENT

SEQUENCE

executed *10* times

```
for i in range(9):
    k = k + 4
```

executed *n* times

```
for i in range(n):
    for i in range(20):
        k = k + i + j
```

inner loop executed *20* times

Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$

UNIVERSITEIT GENT

SELECTION

O(n)

```
if (list.contains(e)) {
  System.out.println(e);
}
else
  for (Object t: list) {
    System.out.println(t);
  }
```

Let n be list.size(). Executed n times.

Time Complexity

T(n) = test time + worst-case (if, else)
= O(n) + O(n)
= O(n)

UNIVERSITEIT GENT

CONSTANT TIME

The Big O notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation O(1). For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

UNIVERSITEIT
GENT

LOGARITHM: ANALYZING BINARY SEARCH

The binary search algorithm presented searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by c. Let T(n) denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and k=logn. Since binary search eliminates half of the input after two comparisons,

$$T(n) = T(\frac{n}{2}) + c = T(\frac{n}{2^2}) + c + c = \ldots = T(\frac{n}{2^k}) + ck = T(1) + c\log n = 1 + c\log n$$

$$n = 2^k$$

$$= O(\log n)$$

UNIVERSITEIT GENT

LOGARITHMIC TIME

Ignoring constants and smaller terms, the complexity of the binary search algorithm is O(log n). An algorithm with the O(logn) time complexity is called a logarithmic algorithm. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.

ANALYZING SELECTION SORT

The selection sort algorithm presented finds the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number. The number of comparisons is n-1 for the first iteration, n-2 for the second iteration, and so on. Let T(n) denote the complexity for selection sort and c denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = (n-1) + c + (n-2) + c \ldots + 2 + c + 1 + c = \frac{n^2}{2} - \frac{n}{2} + cn$$

Ignoring constants and smaller terms, the complexity of the selection sort algorithm is O(n^2).

UNIVERSITEIT GENT

QUADRATIC TIME

An algorithm with the O(n2) time complexity is called a quadratic algorithm. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.

COMMON RECURRENCE RELATIONS

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 17)
$T(n) = 2T(n/2) + O(n \log n)$	$T(n) = O(n \log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

UNIVERSITEIT
GENT

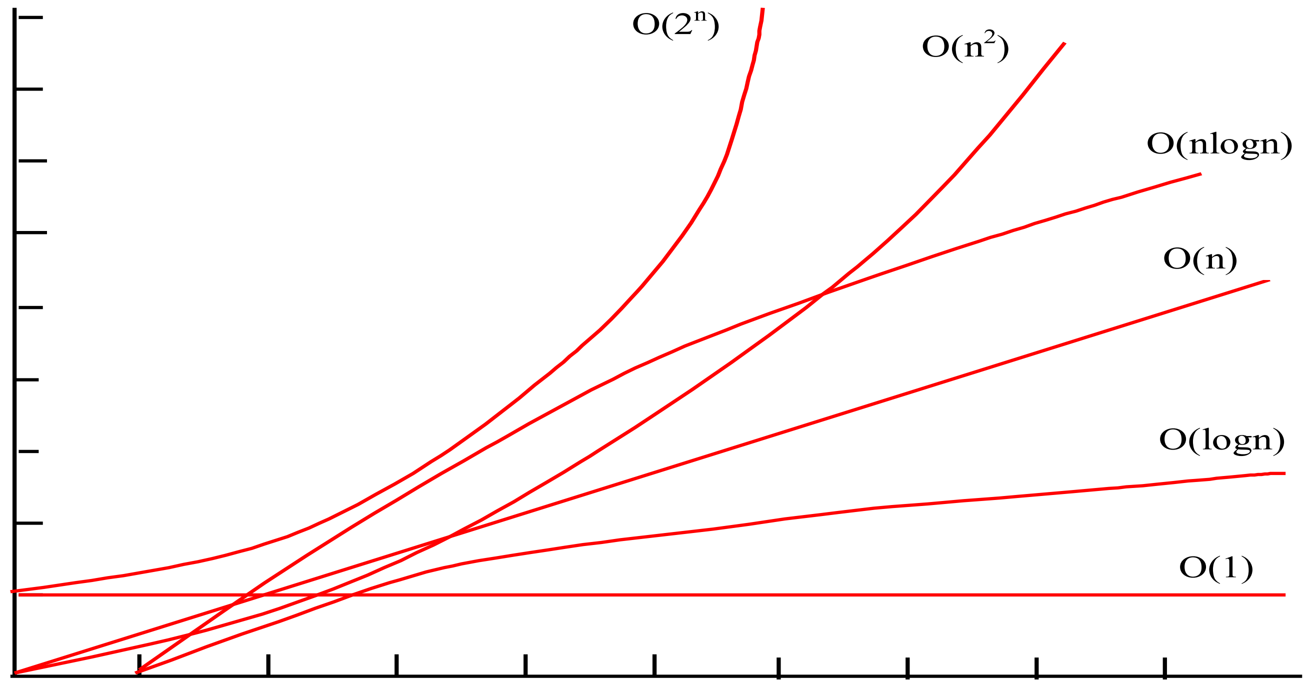
COMPARING COMMON GROWTH FUNCTIONS

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

COMPARING COMMON GROWTH FUNCTIONS

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



UNIVERSITEIT GENT

FACULTEIT ECONOMIE
EN BEDRIJFSKUNDE

CHAPTER 17 SORTING

UNIVERSITEIT
GENT

OBJECTIVES

- To study and analyze time efficiency of various sorting algorithms (§§17.2–17.7).
- To design, implement, and analyze insertion sort (§17.2).
- To design, implement, and analyze bubble sort (§17.3).
- To design, implement, and analyze merge sort (§17.4).
- To design, implement, and analyze quick sort (§17.5).
- ~~To design and implement a heap (§17.6).~~
- ~~To design, implement, and analyze heap sort (§17.6).~~
- ~~To design, implement, and analyze bucket sort and radix sort (§17.7).~~

UNIVERSITEIT
GENT

WHY STUDY SORTING?

- Sorting is a classic subject in computer science. There are three reasons for studying sorting algorithms.
  - First, sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems.
  - Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
  - Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

UNIVERSITEIT GENT

WHAT DATA TO SORT?

The data to be sorted might be integers, doubles, characters, or objects. For simplicity, this chapter assumes:

- data to be sorted are integers,
- data are stored in a list, and
- data are sorted in ascending order

UNIVERSITEIT GENT

INSERTION SORT

– myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step2: The sorted sublist is [2, 9]. Insert 5 into the sublist.

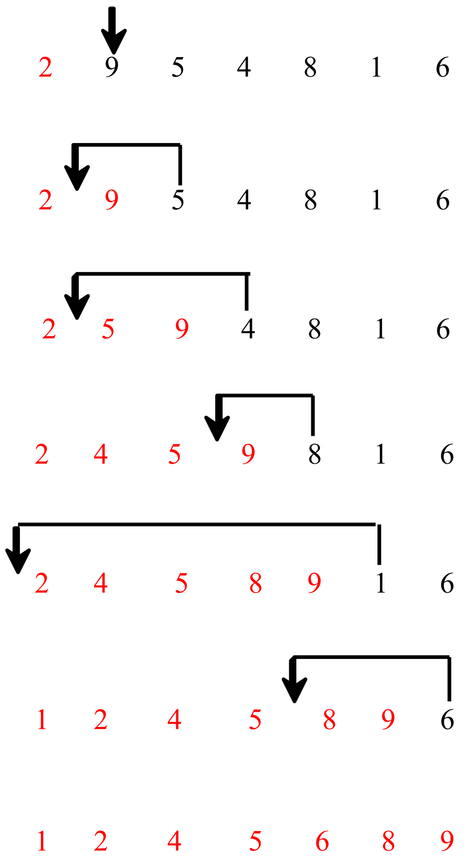
Step 3: The sorted sublist is [2, 5, 9]. Insert 4 into the sublist.

Step 4: The sorted sublist is [2, 4, 5, 9]. Insert 8 into the sublist.

Step 5: The sorted sublist is [2, 4, 5, 8, 9]. Insert 1 into the sublist.

Step 6: The sorted sublist is [1, 2, 4, 5, 8, 9]. Insert 6 into the sublist.

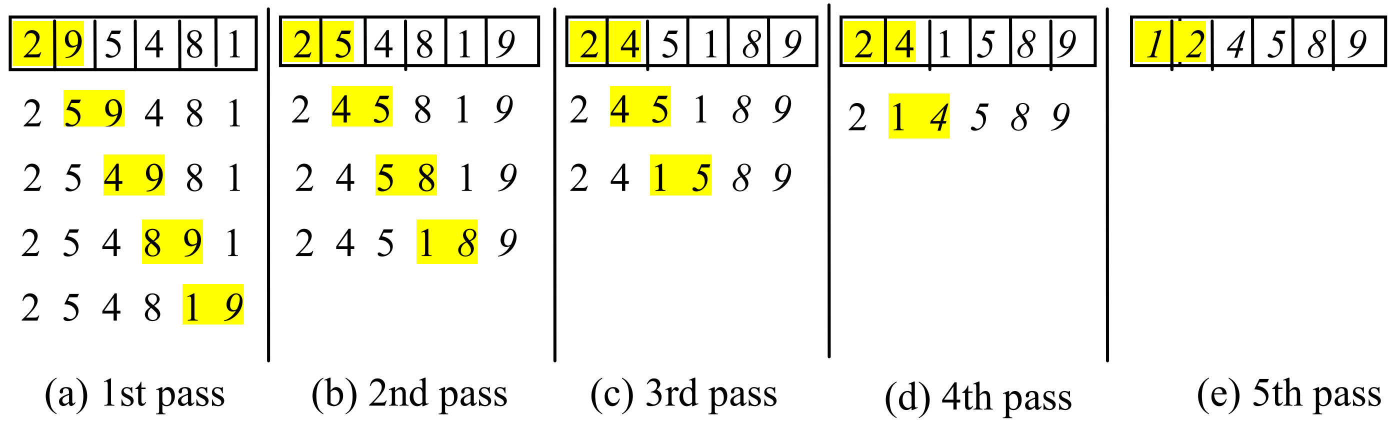
Step 7: The entire list is now sorted



UNIVERSITEIT GENT

BUBBLE SORT

Bubble sort time: O(n2)



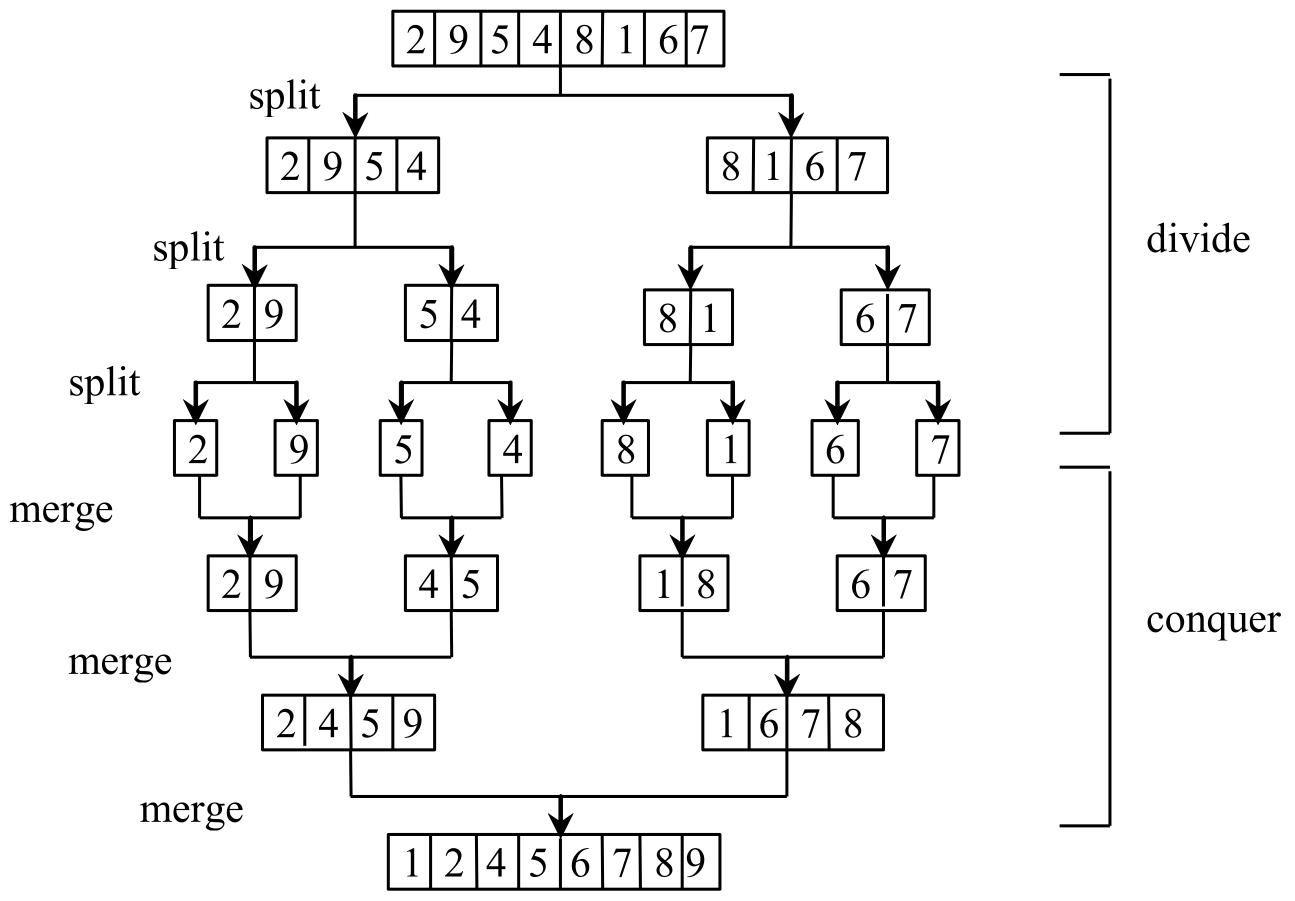
$$(n-1)+(n-2)+...+2+1=\frac{n^2}{2}-\frac{n}{2}$$

BubbleSort

https://liveexample.pearsoncmg.com/dsanimation/BubbleSortNeweBook.html

UNIVERSITEIT GENT

MERGE SORT



MergeSort

UNIVERSITEIT GENT

MERGE SORT TIME

Let T(n) denote the time required for sorting an array of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + mergetime$$

UNIVERSITEIT GENT

MERGE SORT TIME

The first T(n/2) is the time for sorting the first half of the array and the second T(n/2) is the time for sorting the second half. To merge two subarrays, it takes at most n-1 comparisons to compare the elements from the two subarrays and n moves to move elements to the temporary array. So, the total time is 2n-1. Therefore,

$$T(n) = 2T(\frac{n}{2}) + 2n - 1 = 2(2T(\frac{n}{4}) + 2\frac{n}{2} - 1) + 2n - 1 = 2^2 T(\frac{n}{2^2}) + 2n - 2 + 2n - 1$$

$$= 2^k T(\frac{n}{2^k}) + 2n - 2^{k-1} + \ldots + 2n - 2 + 2n - 1$$

$$= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + \ldots + 2n - 2 + 2n - 1$$

$$= n + 2n\log n - 2^{\log n} + 1 = 2n\log n + 1 = O(n\log n)$$

UNIVERSITEIT GENT

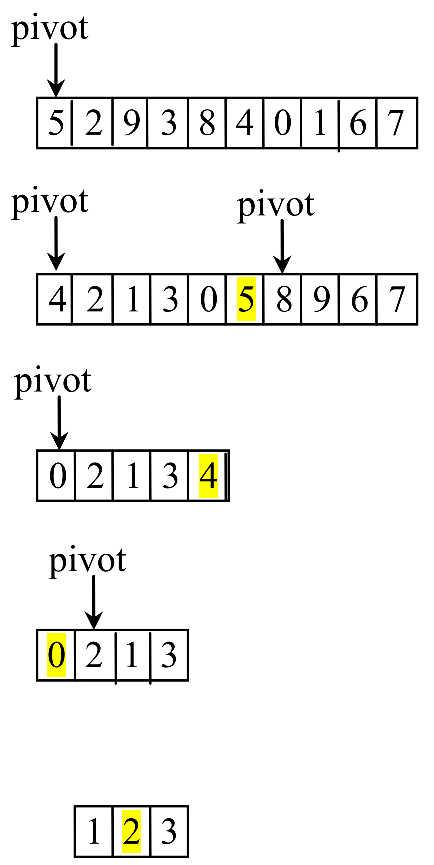
QUICK SORT

Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the pivot, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

UNIVERSITEIT GENT

https://liveexample.pearsoncmg.com/dsanimation/QuickSortNeweBook.html

QUICK SORT



(a) The original array

(b)The original array is partitioned

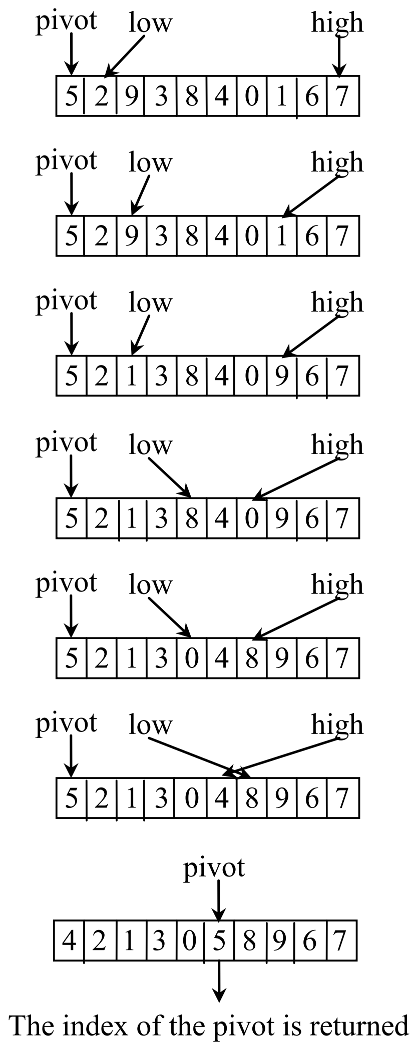
(c) The partial array (4 2 1 3 0) is partitioned

(d) The partial array (0 2 1 3) is partitioned

(e) The partial array (2 1 3) is partitioned

UNIVERSITEIT GENT

PARTITION



(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

QuickSort

UNIVERSITEIT GENT

QUICK SORT TIME

To partition an array of n elements, it takes n-1 comparisons and n moves in the worst case. So, the time required for partition is O(n).