24 Feb 2020

Searching Algorithm
Advanced Sorting Algorithm

CPE111: Programming with Data Structures

Reference

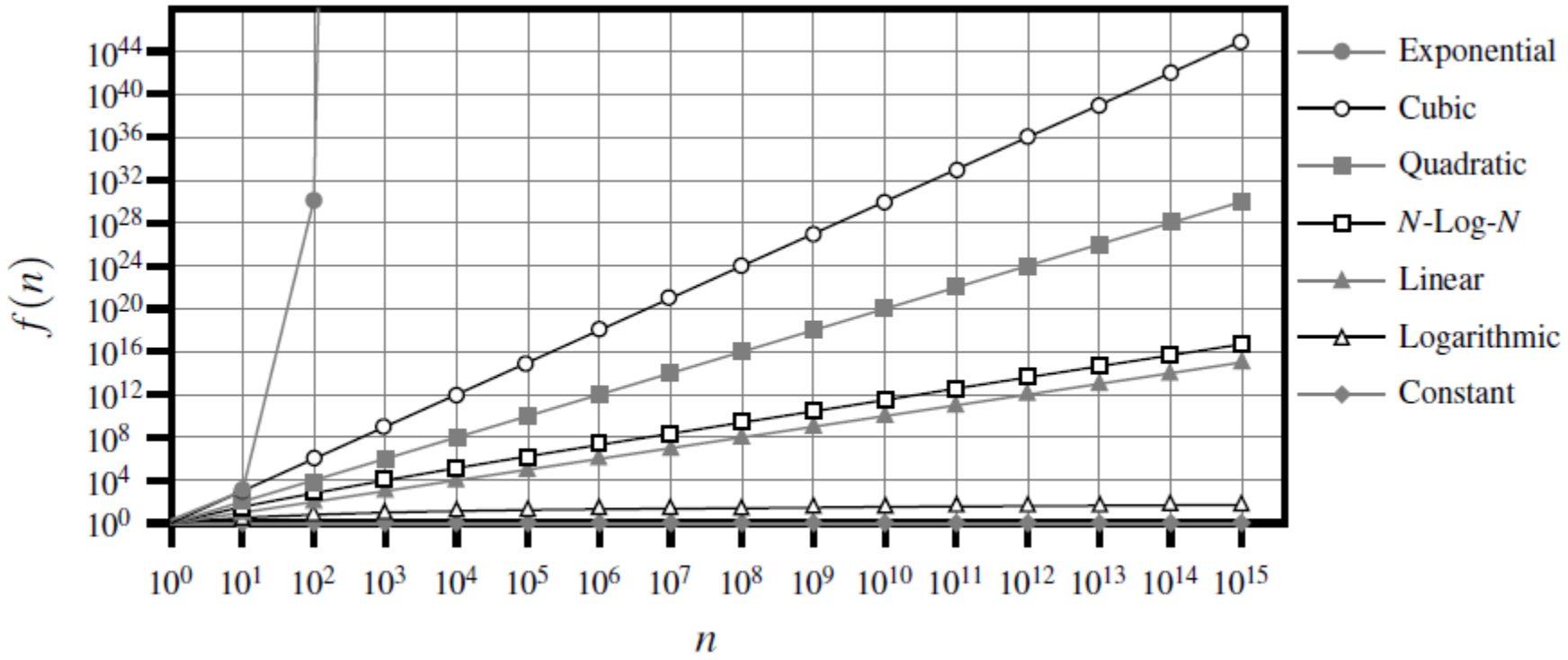
Rance D. Necaise. *Data Structures and algorithms using python*. *Chapter12*. John Wiley&Sons,Inc., 2011

Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. *Data Structures and Algorithms in python*. *Chapter12*. John Wiley&Sons,Inc. 2013

Compare Growth Rate

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

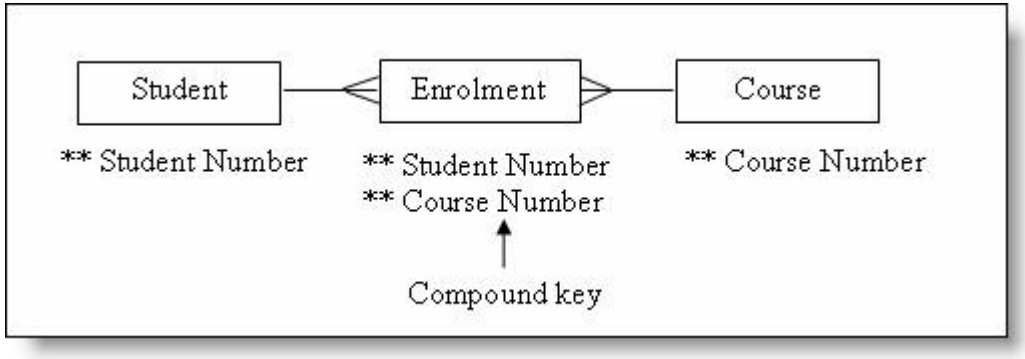
a is a constant and a > 1



S e a r c h i n g is the process of selecting particular information from a collection of data based on specific criteria.

A key or **A Target** is a unique value used to identify the data elements of a collection.

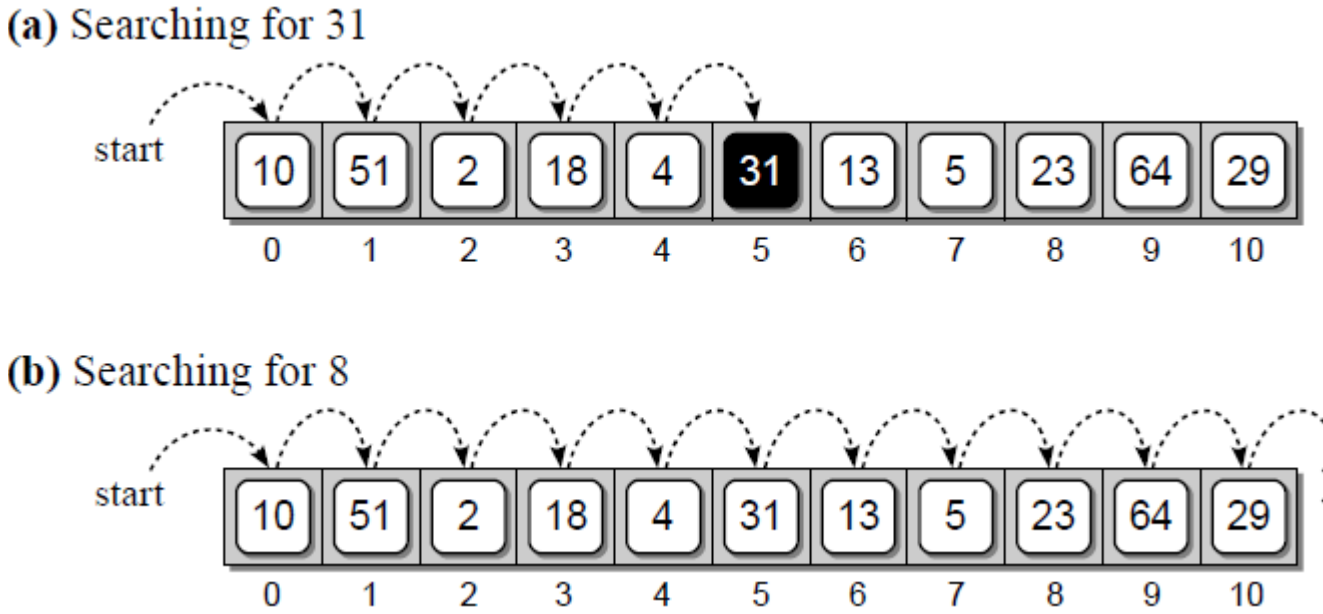
- In collections containing simple types such as integers or reals, the values themselves are the keys.
- For collections of complex types, a specific data component has to be identified as the key.
- In some instances, a key may consist of multiple components, which is also known as a **compound key**.



Sequential search or **Linear search** is the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set

```
def sequential_search(seq,key):
    for i in range(len(seq))
        if seq[i] == key: return True
return False
```

```
def sequential_search(seq, key):
    for item in seq:
        if item == key: return True
    return False
```

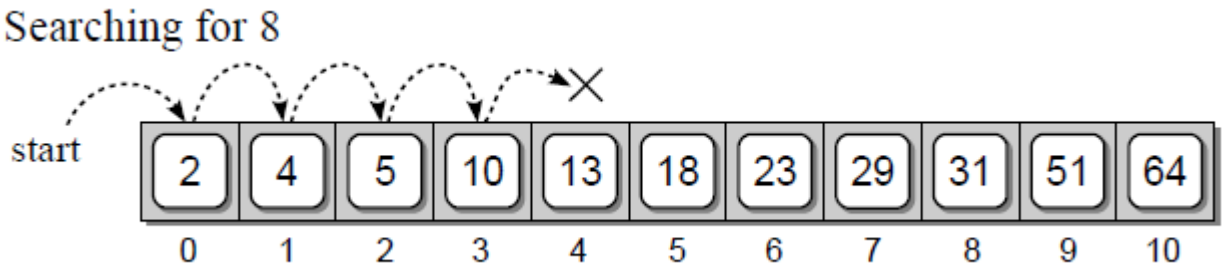


Assuming the sequence contains n items, the sequential search or linear search has a worst case time of O(n).

Searching a sorted sequence

A linear search on a sorted sequence works in the same fashion as that for the unsorted sequence, with one exception. It's possible to terminate the search early when the value is not in the sequence instead of always having to perform a complete traversal.

```
def ordered_sequential_search(seq, key):
    item = 0
    for item in seq:
        if item > key: return False
        if item == key: return True
    return False
```



Binary search

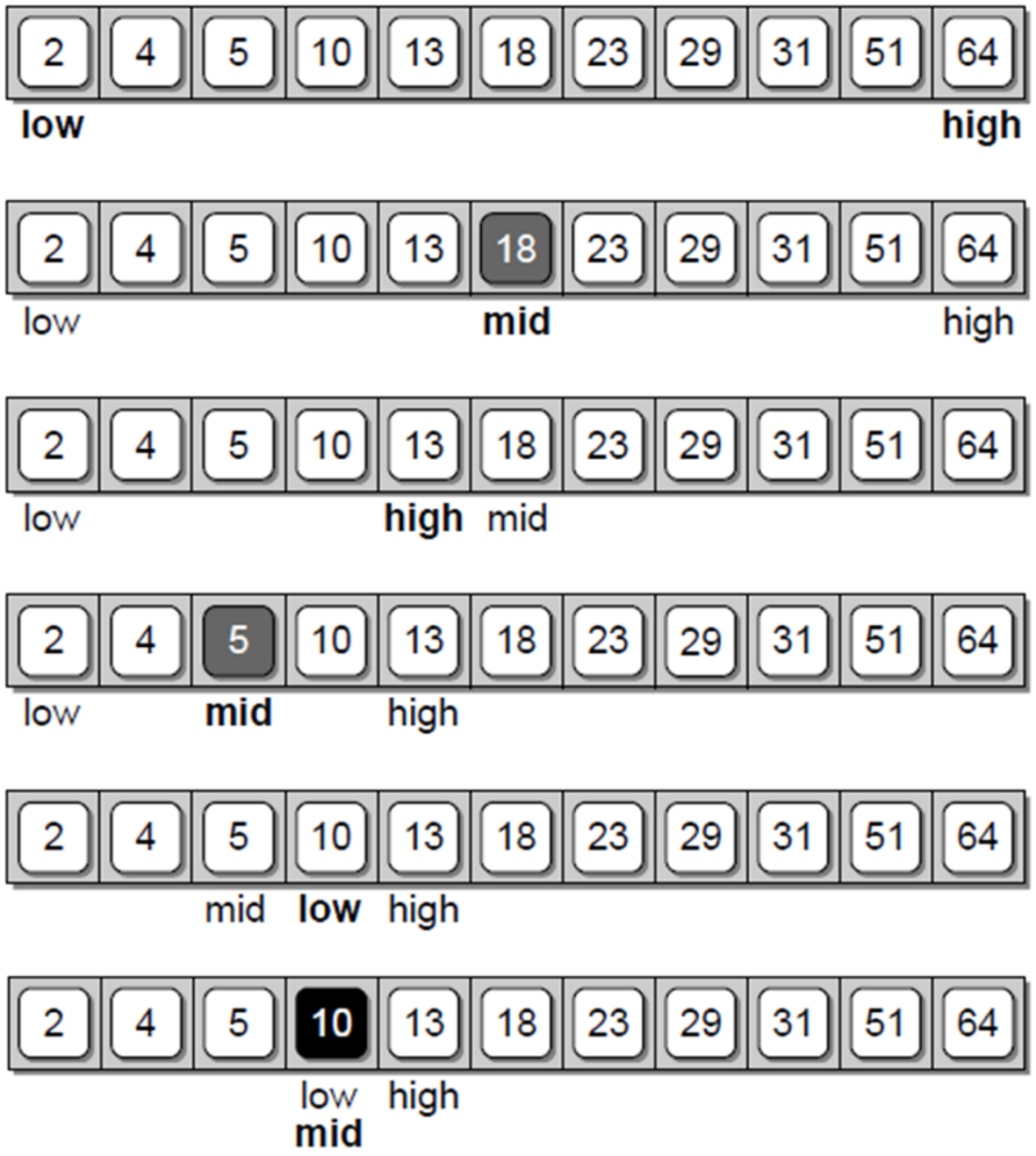
Binary Search is a improved linear search algorithm using when the sequence is ***sorted*** and ***indexable***, it is a much more efficient algorithm compare with the original linear search algorithm

The algorithm starts by examining the middle item of the sorted sequence, resulting in one of three possible conditions:

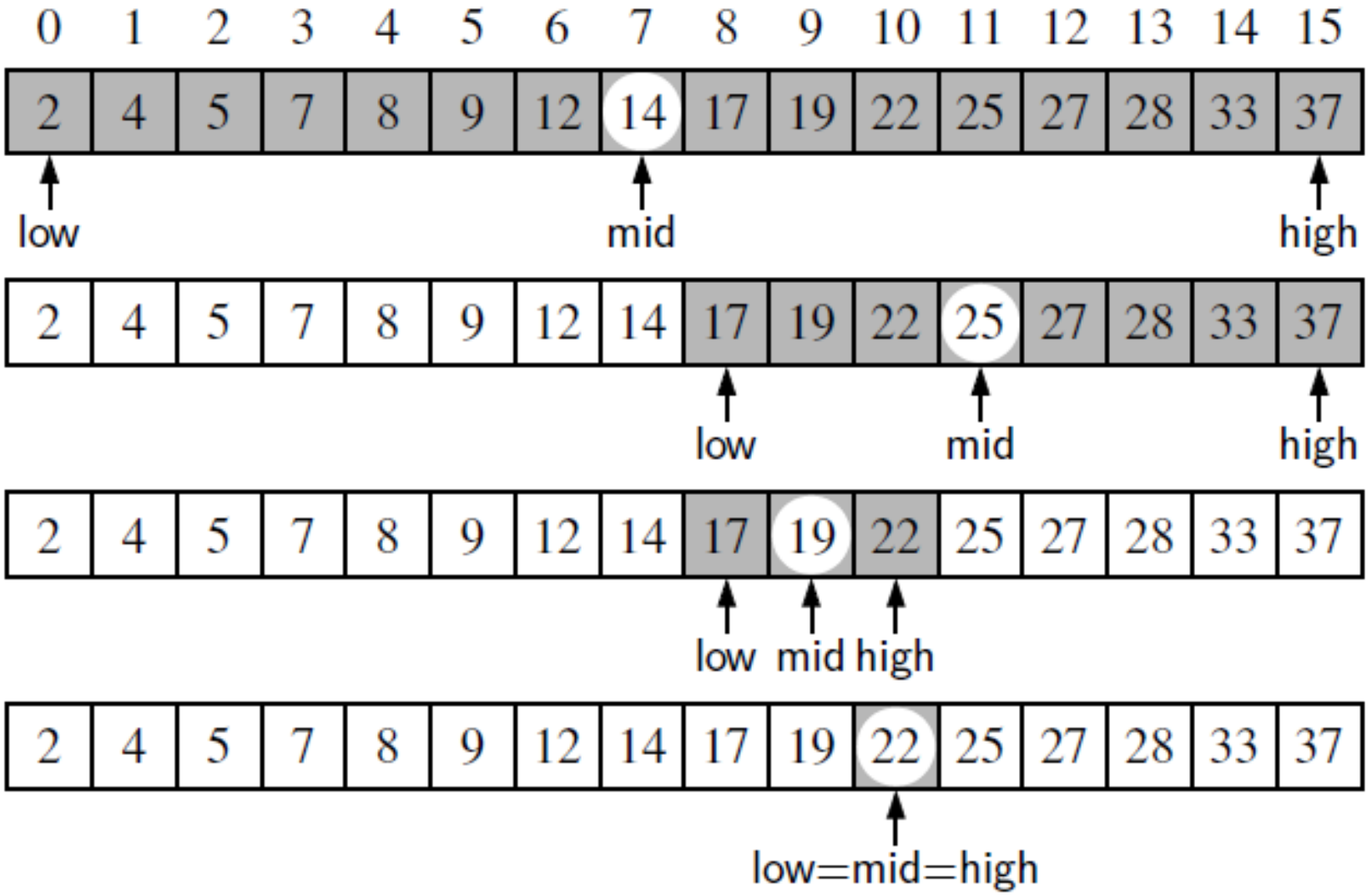
1. If **target == data[mid]**, then we have found the item we are looking for, and the search terminates successfully.
2. If **target < data[mid],** then we recur on the first half of the sequence, that is, on the interval of indices from **low to mid – 1**.
3. If **target > data[mid],** then we recur on the second half of the sequence, that is, on the interval of indices from **mid + 1 to high**.

```
def binary_search( data, target ) :
    low = 0
    high = len(data) - 1
  # Repeatedly subdivide the sequence in half until the target is found.
    while low <= high :
  # Find the midpoint of the sequence.
        mid = (high + low) // 2
      # Does the midpoint contain the target?
        if data[mid] == target : return True
      # Or does the target precede the midpoint?
        elif target < data[mid] : high = mid - 1
      # Or does it follow the midpoint?
        else : low = mid + 1
      # If the sequence cannot be subdivided further, we're done.
    return False
```

key = 10

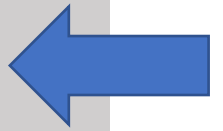

2 4 5 10 13 18 23 29 31 51 64
low
high
2 4 5 10 13 18 23 29 31 51 64
low
mid
high
2 4 5 10 13 18 23 29 31 51 64
low
high
mid
2 4 5 10 13 18 23 29 31 51 64
low
mid
high
2 4 5 10 13 18 23 29 31 51 64
mid
low
high
2 4 5 10 13 18 23 29 31 51 64
low
high
mid

key = 22



Whereas sequential search runs in O(n) time, the more efficient binary search runs in O(log n) time

```
def binary_search(data, target, low, high):
    if low > high:
        return False # interval is empty; no match
    else:
        mid = (low + high) // 2
        if target == data[mid]: # found a match
            return True
        elif target < data[mid]:
            # recur on the portion left of the middle
            return binary_search(data, target, low, mid - 1)
        else:
            # recur on the portion right of the middle
            return binary_search(data, target, mid + 1, high)
```

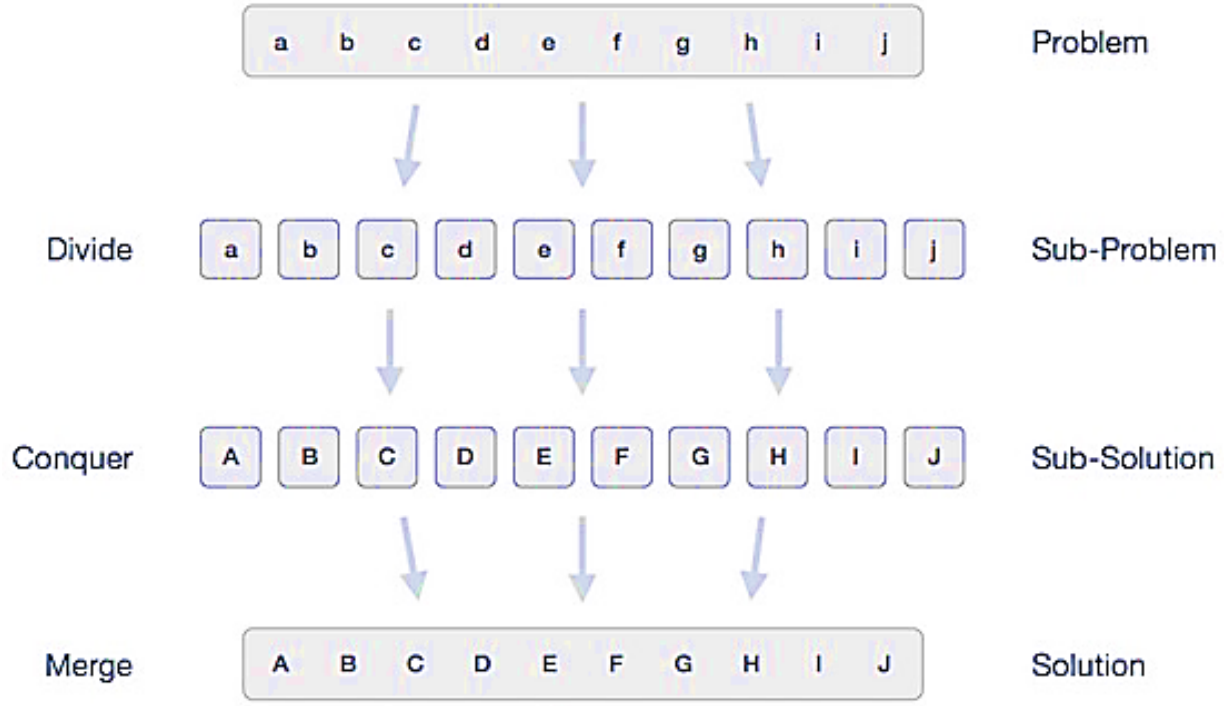


recursion

recursion

Divide and Conquer is an algorithmic paradigm. In Divide and Conquer Approach, the problem is divided into smaller sub-problems and then each problem is solved independently. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide:* Break the given problem into subproblems of same type.
2. *Conquer:* Recursively solve these subproblems
3. *Combine or Merge:* Appropriately combine the answers



Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a **r e c u r s i v e f u n c t i o n**.

A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion.

** A recursion can lead to an infinite loop, if the base case is not met in the calls. **

Example:

4! = 4 * 3 * 2 * 1

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1

Example:

4! = 4 * 3 * 2 * 1

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1

```
def factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result
```

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Example: Fibonacci sequence

n	0	1	2	3	4	5	6	7	8	9	10	…
x(n)	0	1	1	2	3	5	8	13	21	34	55	…

$$x_n = x_{n-1} + x_{n-2}$$

```
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
def fibo(n):
    fibo = f0 = 0
    f1 = c = 1
    if n < 2: return n
    else:
        while True:
            if c >= n: break
            fibo = f0 + f1
            c = c+1
            f0 = f1
            f1 = fibo
    return fibo
```

Advanced Sorting Algorithm

Most sorting algorithms can be divided into two categories:

- **Comparison sort** the data items can be arranged in either ascending (from smallest to largest) or descending (from largest to smallest) order

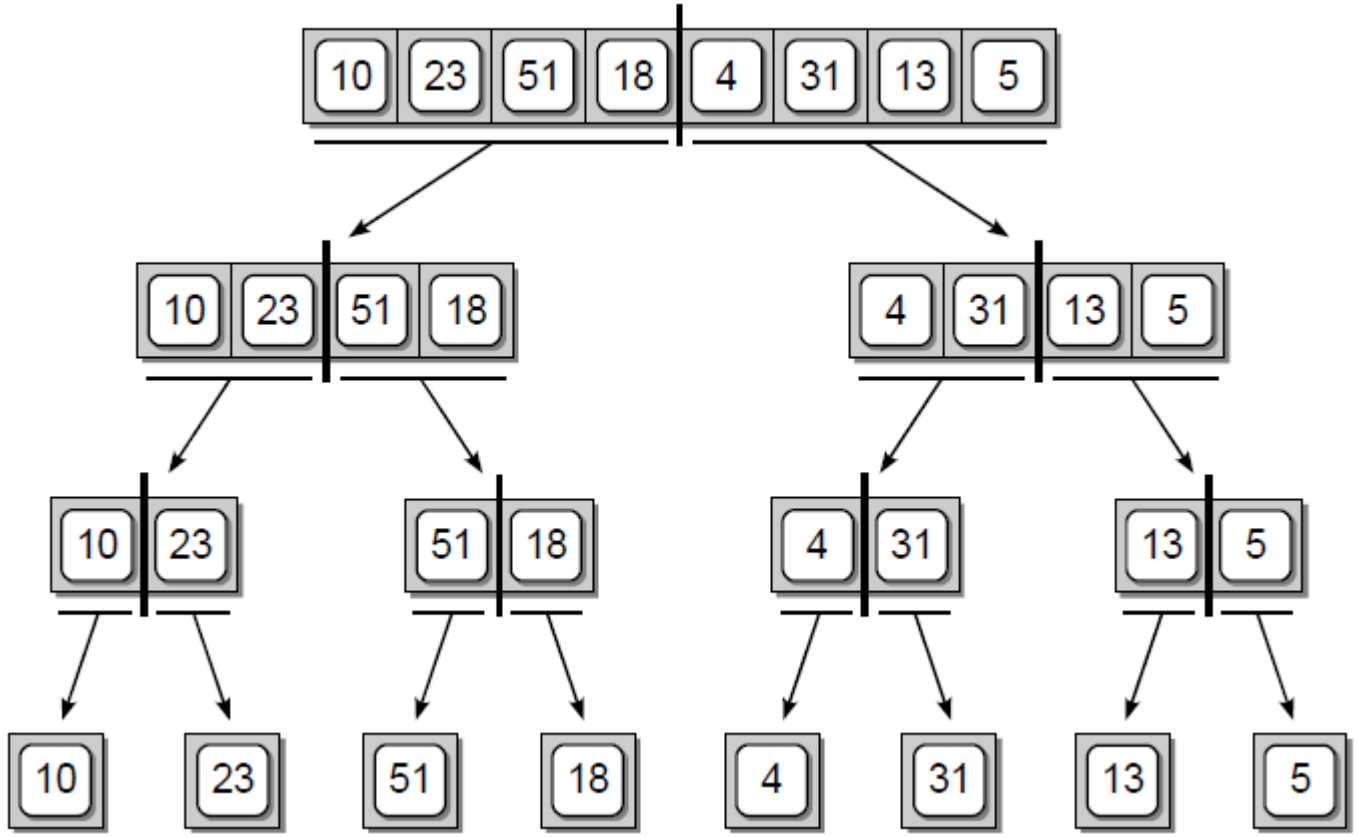
- **Distribution sort** distributes or divides the sort keys into intermediate groups or collections based on the individual key values.

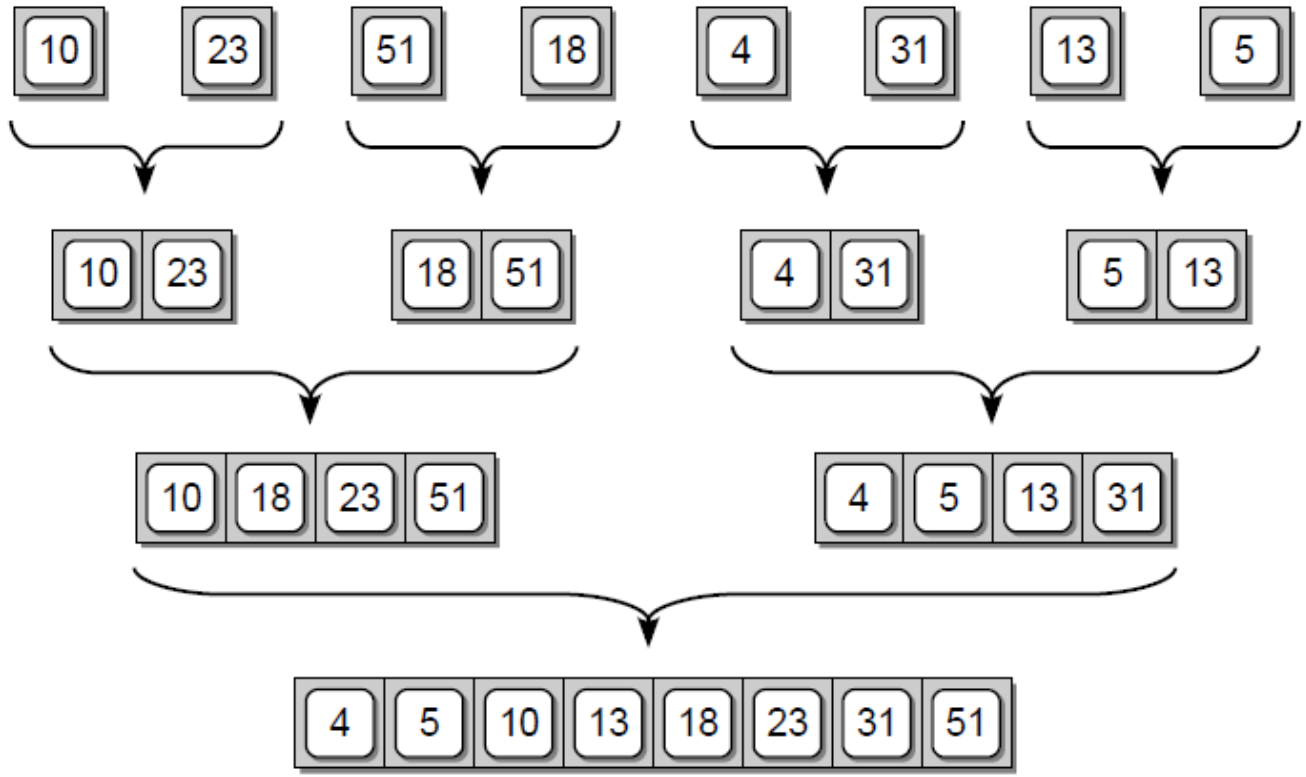
The merge sort algorithm

Merge sort is a good example of recursive divide and conquer algorithm.
The behind concept is:

If the length of **sequence** is at most 1, then **the sequence** is already sorted, so we do nothing. Otherwise, we split **the sequence** into two halves, **left half** = item[0],..,item[n/2-1] and **right half** = item[n=2],..,item[n - 1]. We recursively sort **left half** and **right half**, and then we merge **left half** and **right half** to get our fully sorted **sequence**:

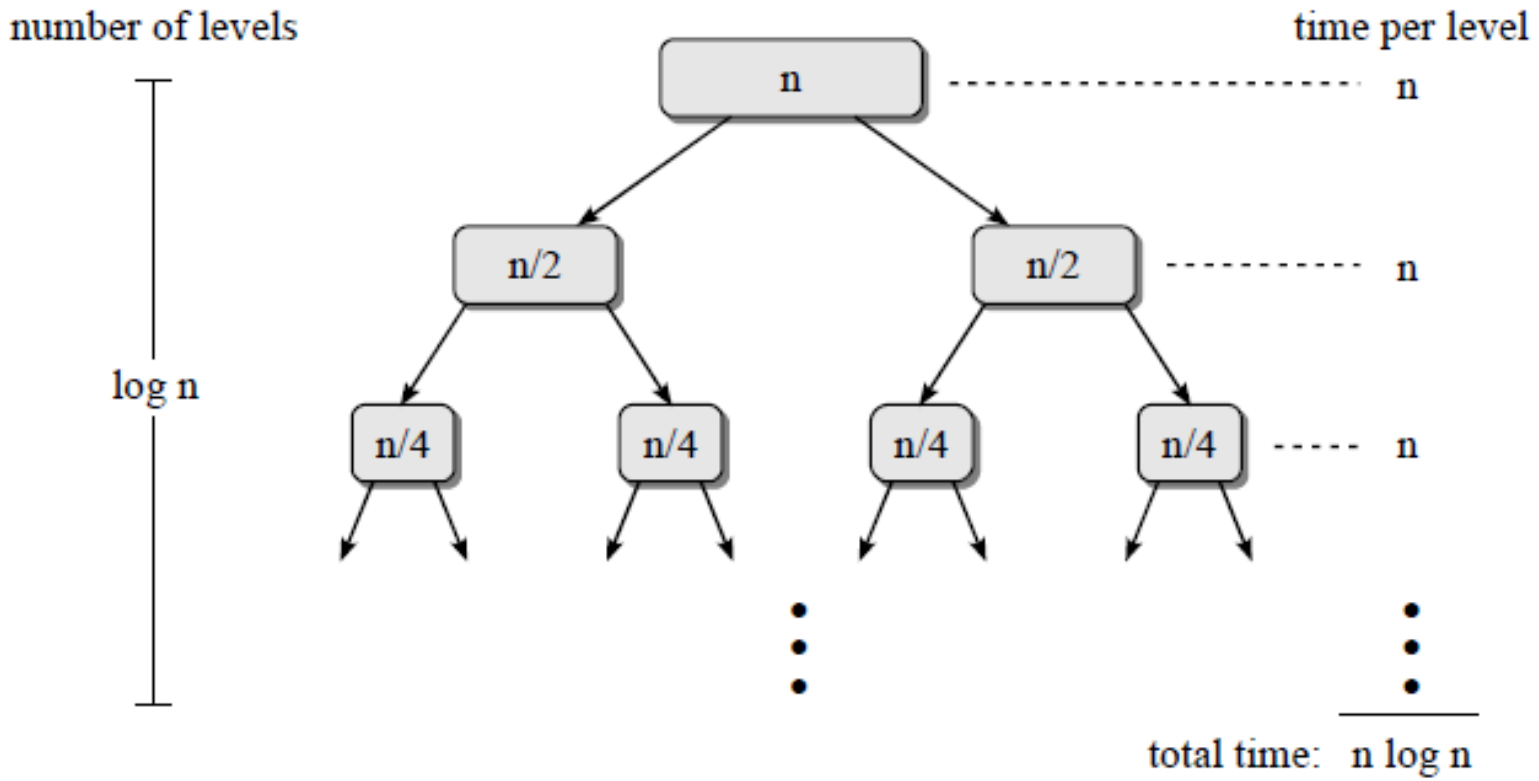
```
def merge_sort(seq):
    n = len(seq)
    if n < 2:
        return # list is already sorted

    # divide
    mid = n // 2
    left = seq[0:mid] # copy of first half
    right = seq[mid:n] # copy of second half

    # conquer (with recursion)
    merge_sort(left) # sort copy of first half
    merge_sort(right) # sort copy of second half

    # merge results
    seq = merge(left, right, seq)
    return seq
```




10
23
51
18
4
31
13
5
10
23
18
51
4
31
5
13
10
18
23
51
4
5
13
31
4
5
10
13
18
23
31
51

```
def merge(left, right, seq):
    i = j = 0
    while i + j < len(seq):
        if j == len(right) or (i < len(left) and left[i] < right[j]):
            seq[i+j] = left[i] # copy ith element of S1 as next item of S
            i += 1
        else:
            seq[i+j] = right[j] # copy jth element of S2 as next item of S
            j += 1
```

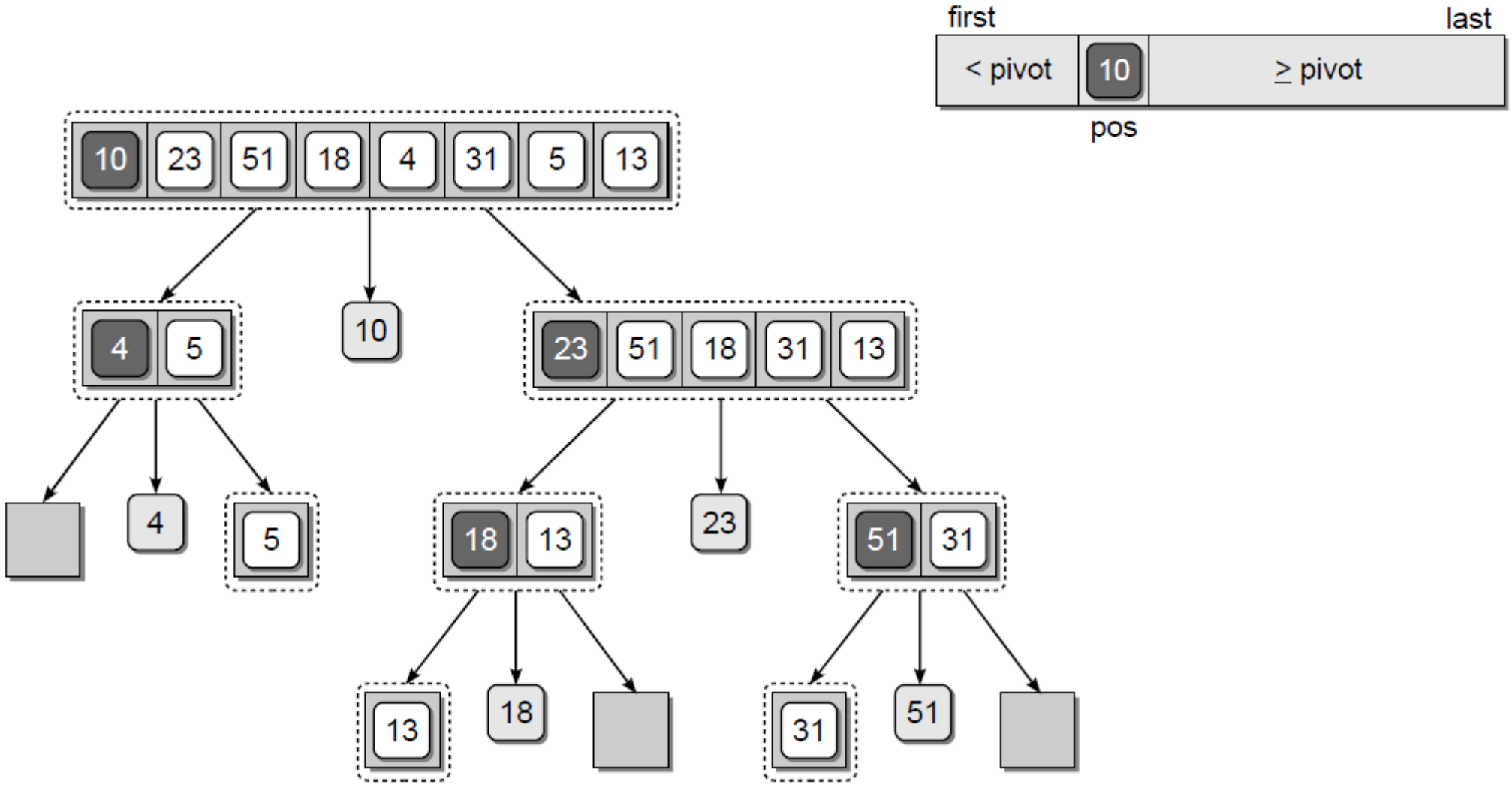
Time analysis of the merge sort algorithm

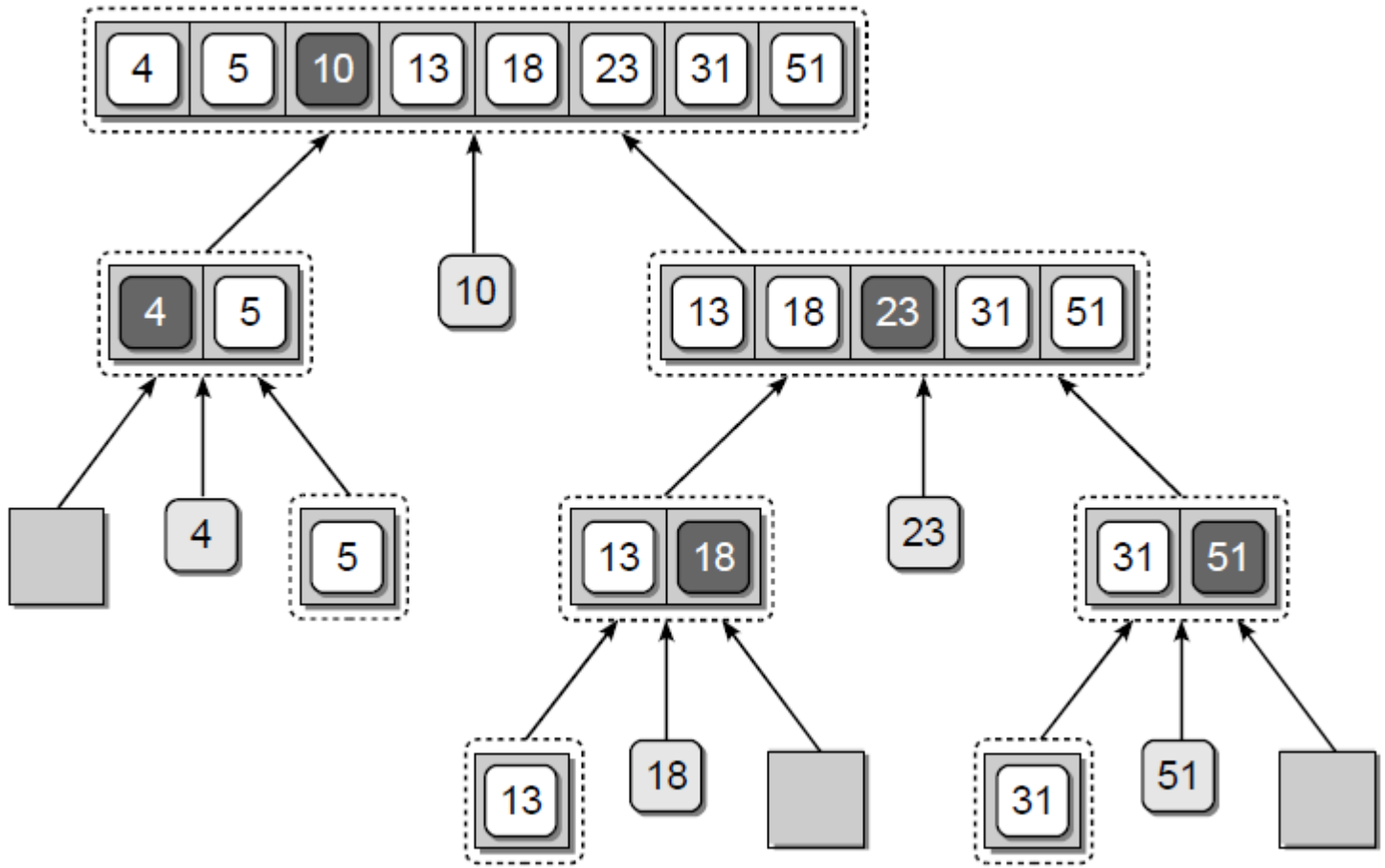


The quick sort algorithm

The quick sort also uses the divide and conquer strategy. But unlike the merge sort, which splits the sequence of keys at the midpoint, the quick sort partitions the sequence by dividing it into two segments based on a selected pivot key.

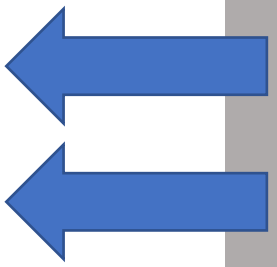
Quicksort is simple to describe: Pick a pivot item (may be the first item), x, from the sequence; partition the sequence into the set of items less than x, the set of items equal to x, and the set of items greater than x; and, finally, recursively sort the first and third sets in this partition.


first
last
< pivot
10
≥ pivot
pos
10 23 51 18 4 31 5 13
4 5
10
23 51 18 31 13
4
5
18 13
23
51 31
13
18
31
51


4
5
10
13
18
23
31
51
4
5
10
13
18
23
31
51
4
5
13
18
23
31
51
13
18
31
51

```
for x in seq
  if x < = pi:
      lo.append(x)
```

```
for x in seq
  if x > pi:
      hi.append(x)
```



```
def quick_sort(seq):
    if len(seq) < 2 : return seq
    mid = len(seq)//2
    pi = seq[mid]
    seq = seq[:mid] + seq[mid+1:]
    lo = [x for x in seq if x <= pi]
    hi = [x for x in seq if x > pi]
    return quick_sort(lo) + [pi] + quick_sort(hi)
```

For **the comparison sort algorithms**

- The previous three sorting algorithm: bubble, selection, and insertion, have a worst case time of O(n^2)
- The merge sort has a worst case time of O(n log n).
- The quick sort, the more commonly used algorithm in language libraries, is O(n^2) in the worst case but it has an expected or average time of O(n log n).

Sorting	Time Complexity		
	Best Case	Average Case	Worst Case
bubble	O(n)	O(n^2)	O(n^2)
selection	O(n^2)	O(n^2)	O(n^2)
insertion	O(n)	O(n^2)	O(n^2)
merge	O(n log n)	O(n log n)	O(n log n)
quick	O(n log n)	O(n log n)	O(n^2)

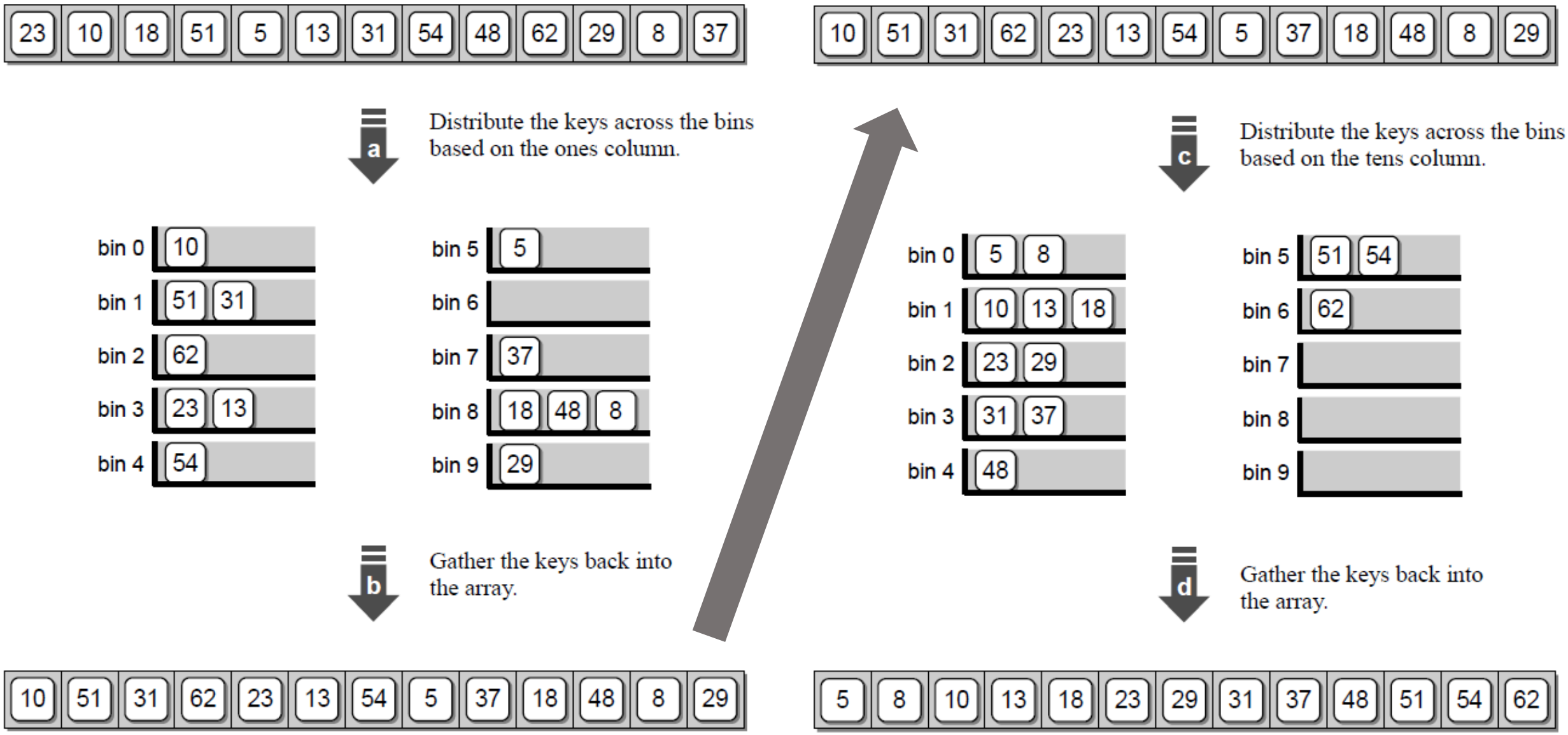
The Radix sort Algorithm

Radix sort is a distribution sorting algorithm that orders keys by examining the individual components of the keys instead of comparing the keys themselves. For example, when sorting integer keys, the individual digits of the keys are compared from least significant to most significant.

This is a special purpose sorting algorithm but can be used to sort many types of keys, including positive integers, strings, and floating-point values. The radix sort algorithm also known as **“bin sort”**

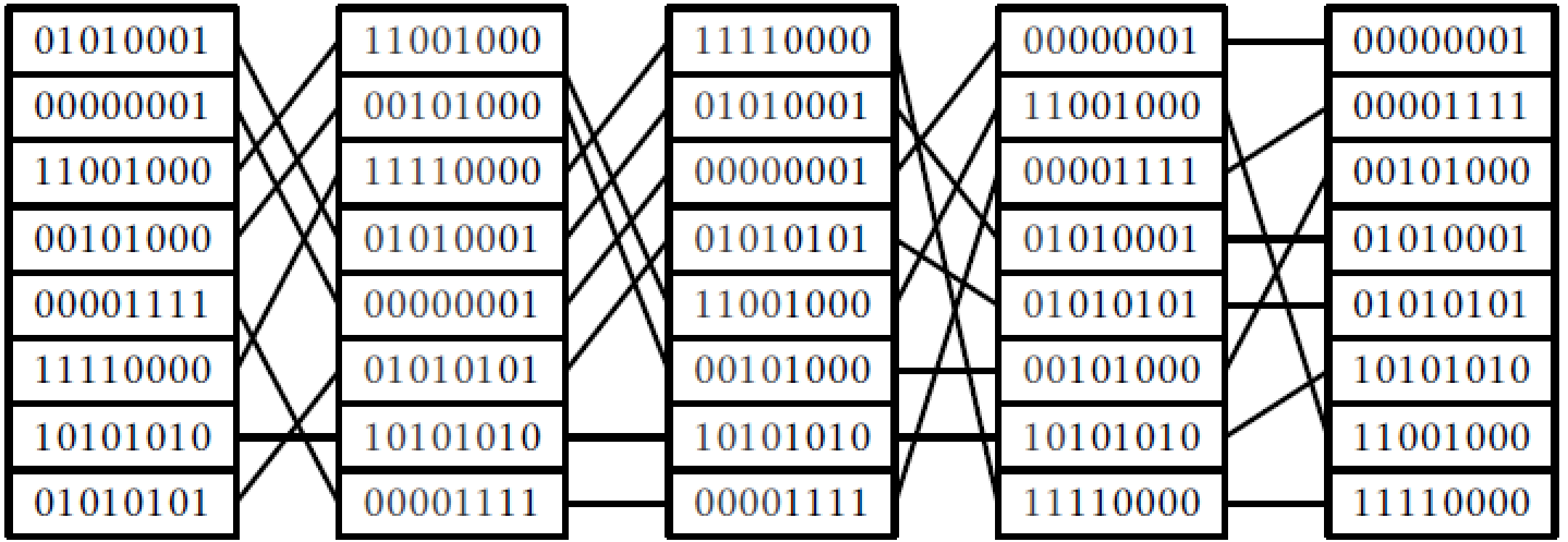
The sorting process starts by distributing the items into the bins based on the digits from the least significant digit, sort the least significant digit, and gathered back into one bin. Again distributing the items into various bins based on next significant digit, repeat the process until all of digits are sorted.

- The individual bins store groups of keys based on the individual digits.
- Keys with duplicate digits (in a given column) are stored in the same bin, but following any that are already there.
- When the keys are gathered from the bins, they have to be stored back into the original sequence. This is done by removing them from the bins in a first-in first-out ordering.

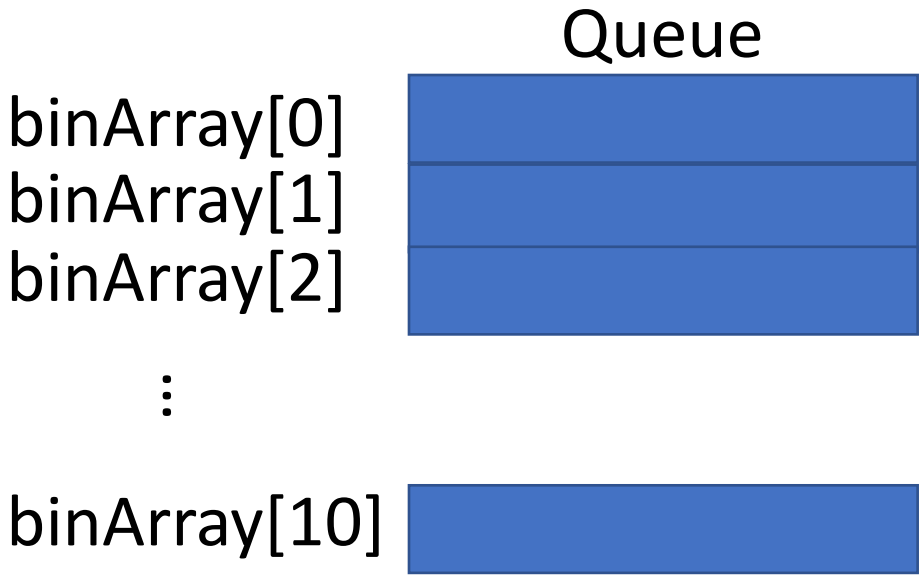

23 10 18 51 5 13 31 54 48 62 29 8 37
a
Distribute the keys across the bins based on the ones column.
bin 0 10
bin 1 51 31
bin 2 62
bin 3 23 13
bin 4 54
bin 5 5
bin 6
bin 7 37
bin 8 18 48 8
bin 9 29
b
Gather the keys back into the array.
10 51 31 62 23 13 54 5 37 18 48 8 29
10 51 31 62 23 13 54 5 37 18 48 8 29
c
Distribute the keys across the bins based on the tens column.
bin 0 5 8
bin 1 10 13 18
bin 2 23 29
bin 3 31 37
bin 4 48
bin 5 51 54
bin 6 62
bin 7
bin 8
bin 9
d
Gather the keys back into the array.
5 8 10 13 18 23 29 31 37 48 51 54 62

Sorting the binary number

no. of bin = 2→ [0,1]



```
from linklistedQueue import Lqueue
from array import Array
def radixSort( seq, numDigits ):
    binArray = Array(10)
    for k in range( 10 ): binArray[k] = LQueue()
    column = 1
    for d in range( numDigits ):
        for key in seq :
            digit = (key // column) % 10
            binArray[digit].enqueue( key ).
        i = 0
        for bin in range(len(binArray)) :
            while not binArray[bin].isEmpty() :
                seq[i] = binArray[bin].dequeue()
                i += 1
        column *= 10
    return seq
```



Sorting	Time Complexity		
	Best Case	Average Case	Worst Case
bubble	O(n)	O(n^2)	O(n^2)
selection	O(n^2)	O(n^2)	O(n^2)
insertion	O(n)	O(n^2)	O(n^2)
merge	O(n log n)	O(n log n)	O(n log n)
quick	O(n log n)	O(n log n)	O(n^2)
radix	O(nk)	O(nk)	O(nk)

k is no. of digits