# Extended Introduction to Computer Science CS1001.py

## Lecture 11:
## Recursion: Quicksort and Mergsort

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University

Spring Semester 2017

http://tau-cs1001-py.wikidot.com

# Lecture 10: Reminder

Recursion, and recursive functions

- Basic examples and definition of recursion
  - Fibonacci
  - factorial
- Binary search - revisited
- Sorting
  - QuickSort

# Lecture 11 (1 hr only)

- Recursion, and recursion trees
  - QuickSort – cont.
  - MergeSort

- A word about space complexity

# Another Manifestation of Recursion (with a twist)

(Cover of Ummagumma, a double album by Pink Floyd, released in 1969.
Taken from Wikipedia. Thanks to Yair Sela for the suggestion.)

# Recursion (definition, reminder)

A function f(·), whose definition contains a call to f(·) itself, is called recursive.

A simple example is the factorial function, $n! = 1 \cdot 2 \cdot \ldots \cdot n$.
It can be coded in Python, using recursion, as follows:

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

A second simple example are the Fibonacci numbers, defined by
$F_1 = 1$, $F_2 = 1$, and for $n > 2$; $F_n = F_{n-2} + F_{n-1}$.

# Recursion and Convergence (reminder)

Two “design principles" to the correct design of recursive functions:

1. Have a base case (one or more), which is the halting condition (no deeper recursion). In the factorial example, the base case was the condition n==0. In the Fibonacci example, it was n<=1.

2. Make sure that all executions, or “runs", of the recursion will actually lead to one of these base cases.

# Recursion Trees

- A visual description of the course of recursion
- Helps analyzing recursion depth and time complexity
- For example, here is the recursion tree for fibonacci, for n=6:

```
def fibonacci(n):
  if n<=1:
     return 1
  else:
     return fibonacci(n-2) + fibonacci(n-1)
```

fib(6)

fib(4) fib(5)

fib(2) fib(3) fib(3) fib(4)

fib(0) fib(1) fib(1) fib(2) fib(1) fib(2) fib(2) fib(3)

fib(0) fib(1) fib(0) fib(1) fib(0) fib(1) fib(1) fib(2)

fib(0) fib(1)

# Recursion Trees

- Recursion depth – distance from root to deepest leaf
  - corresponds to maximal number of recursive calls that are open simultaneously
  - too many such calls may cause memory overload (will discuss this soon)

```
def fibonacci(n):
  if n<=1:
     return 1
  else:
     return fibonacci(n-2) + fibonacci(n-1)
```

- depth of this tree = 5
- for input n the depth is n-1 = O(n)

# Recursion Trees

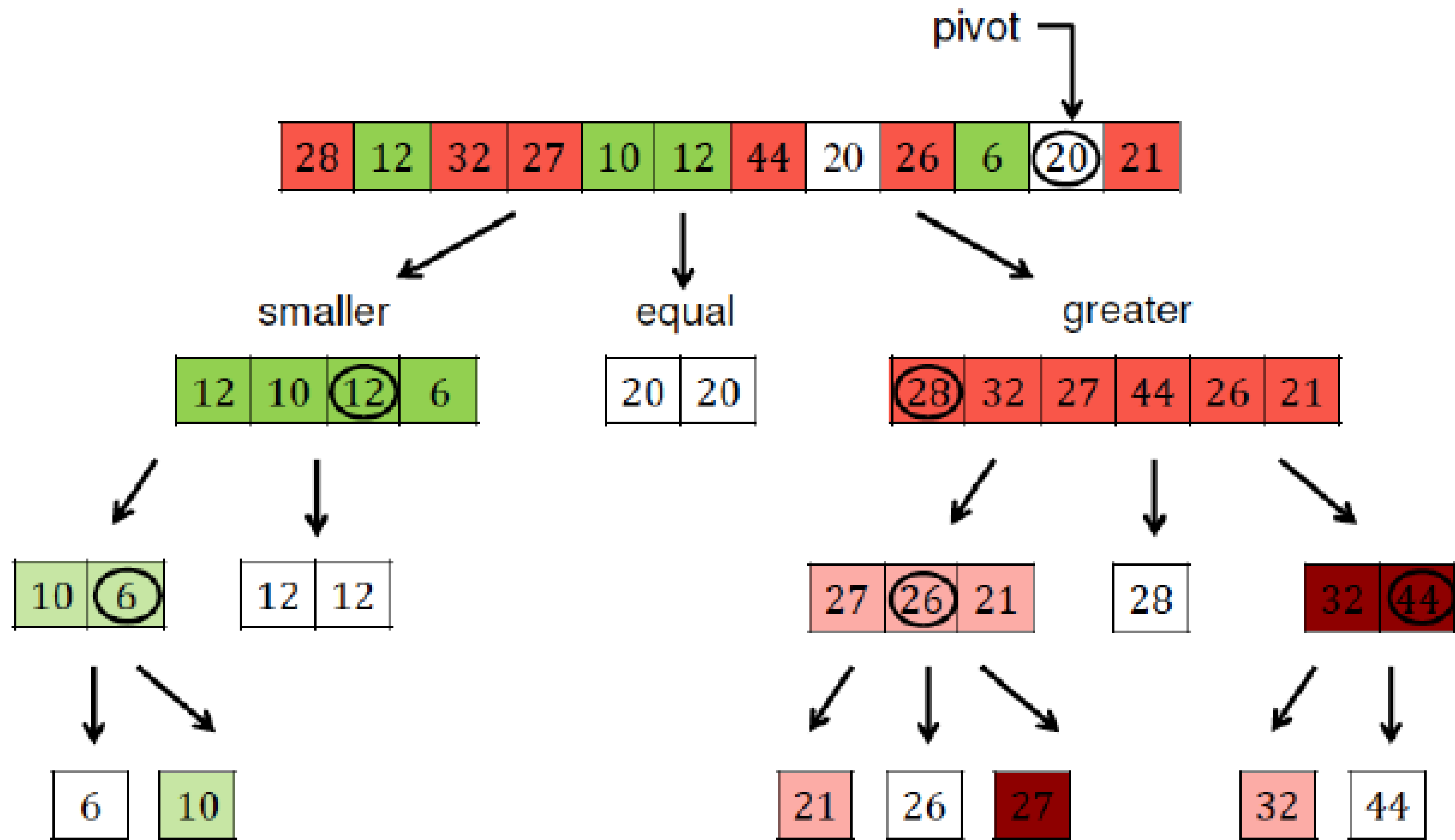
- Time complexity – overall number of operations in the whole tree

```
def fibonacci(n):
  if n<=1:
     return 1
  else:
     return fibonacci(n-2) + fibonacci(n-1)
```

fib(6)
fib(4) fib(5)
fib(2) fib(3) fib(3) fib(4)
fib(0) fib(1) fib(1) fib(2) fib(1) fib(2) fib(2) fib(3)
fib(0) fib(1) fib(0) fib(1) fib(0) fib(1) fib(1) fib(2)
fib(0) fib(1)

- each node here takes O(1) ops
- < $2^n$ nodes
- thus time complexity is O($2^n$) (not tight)
- tightest bound is $\left(\frac{1+\sqrt{5}}{2}\right)^n$ (without proof)

# Quicksort: A Graphical Depiction (reminder)



# Quicksort: Python code (reminder)

```
import random     # a package for (pseudo) random generation

def quicksort(lst):
    if len(lst)<=1:      # empty lists or length one lists
        return lst
    else:
        pivot = random.choice(lst)
                # select a random element from the list
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot]
                # ain't these selections neat?
        return quicksort(smaller) + equal + quicksort(greater)
                # two recursive calls
```

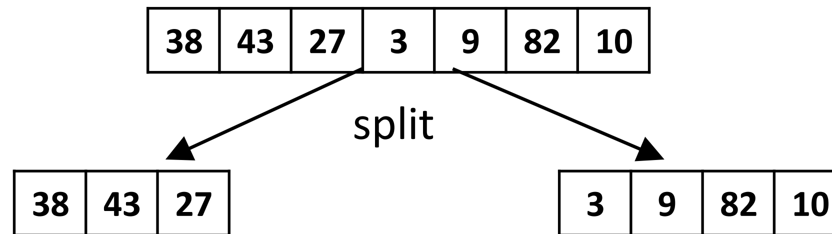
# Quicksort: recursion depth and time complexity

- In class, on board, using recursion trees.

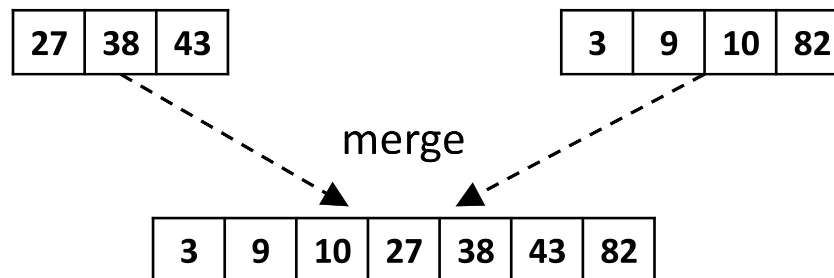
# Merge Sort

Mergesort is a recursive, deterministic, sorting algorithm, i.e. it also follows a divide and conquer approach.

An input list (unsorted) is split to two -- elements with indices from 0 up to the middle, and those from the middle up to the end of the list.




If we could sort these 2 halves, we would be done by merging them.




Well, does anybody know a good sorting algorithm?

# Merge Sort

An input list (unsorted) is split to two -- elements with indices from 0 up to the middle, and those from the middle up to the end of the list.

Each half is sorted recursively.

The two sorted halves are then merged to one, sorted list.

# Merge Sort – example

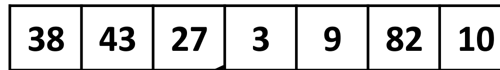
| 38 | 43 | 27 | 3 | 9 | 82 | 10 |
|---|---|---|---|---|---|---|

→ recursive call

- - → recursion fold

\+ merge

# Merge Sort – example

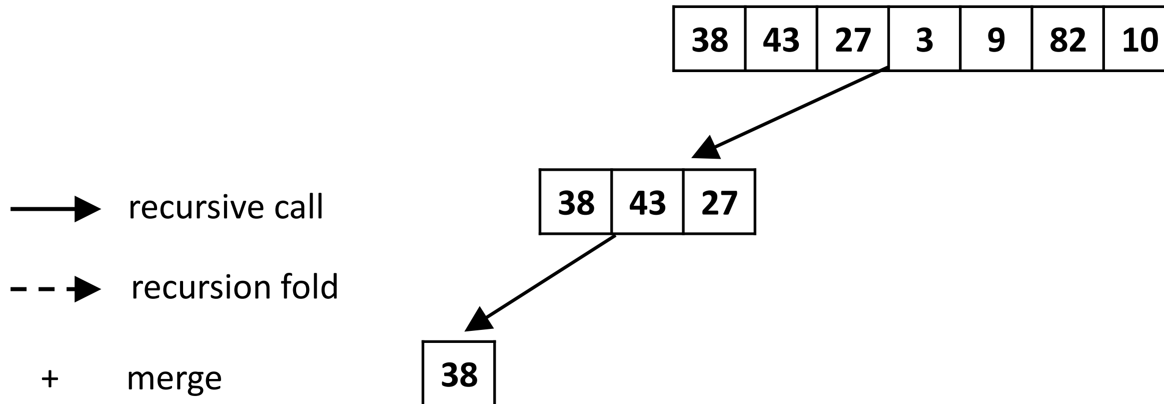


recursive call

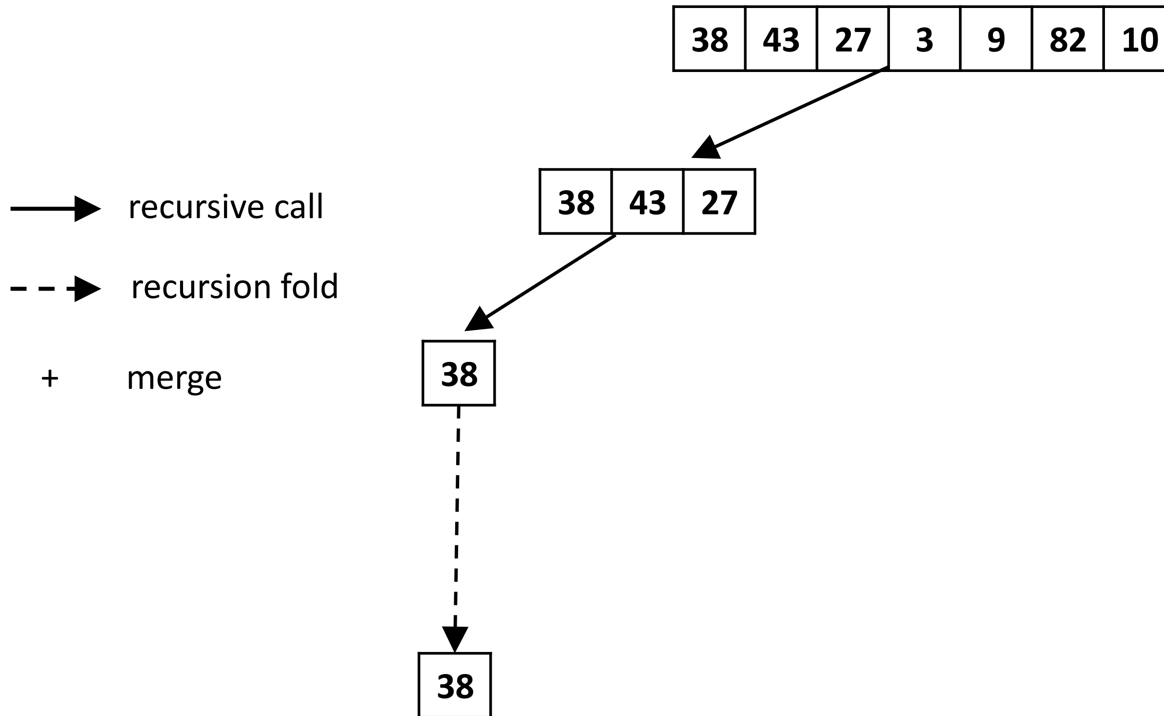
recursion fold

\+ merge

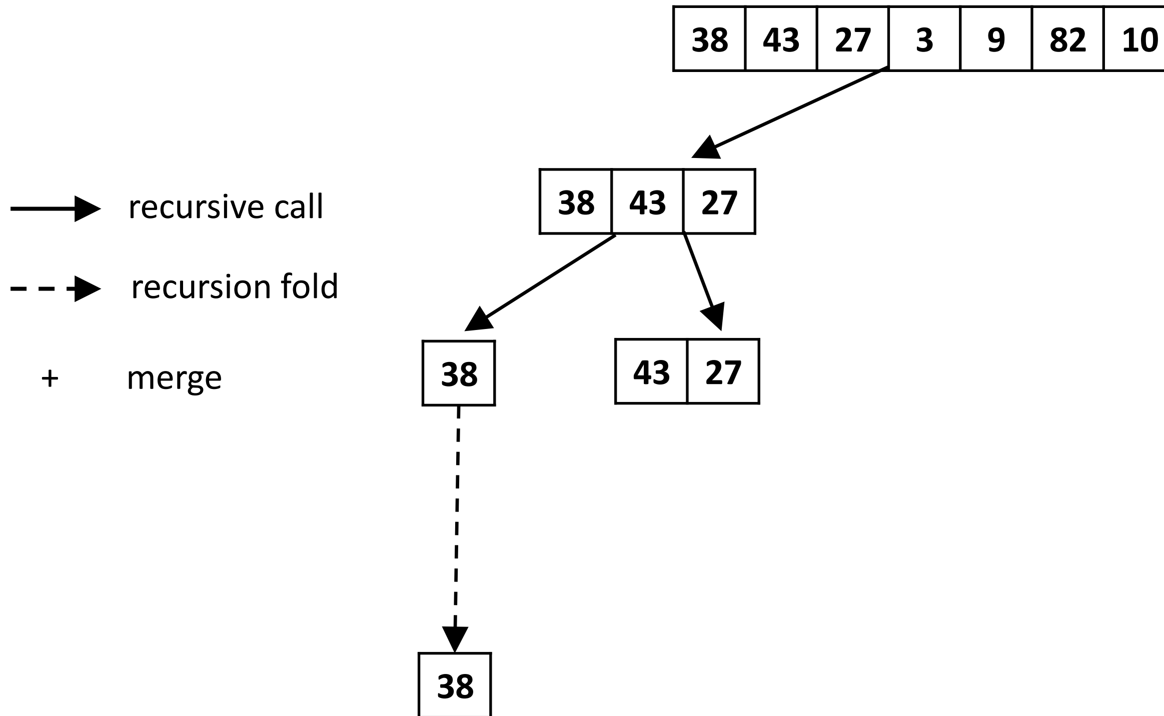
# Merge Sort – example



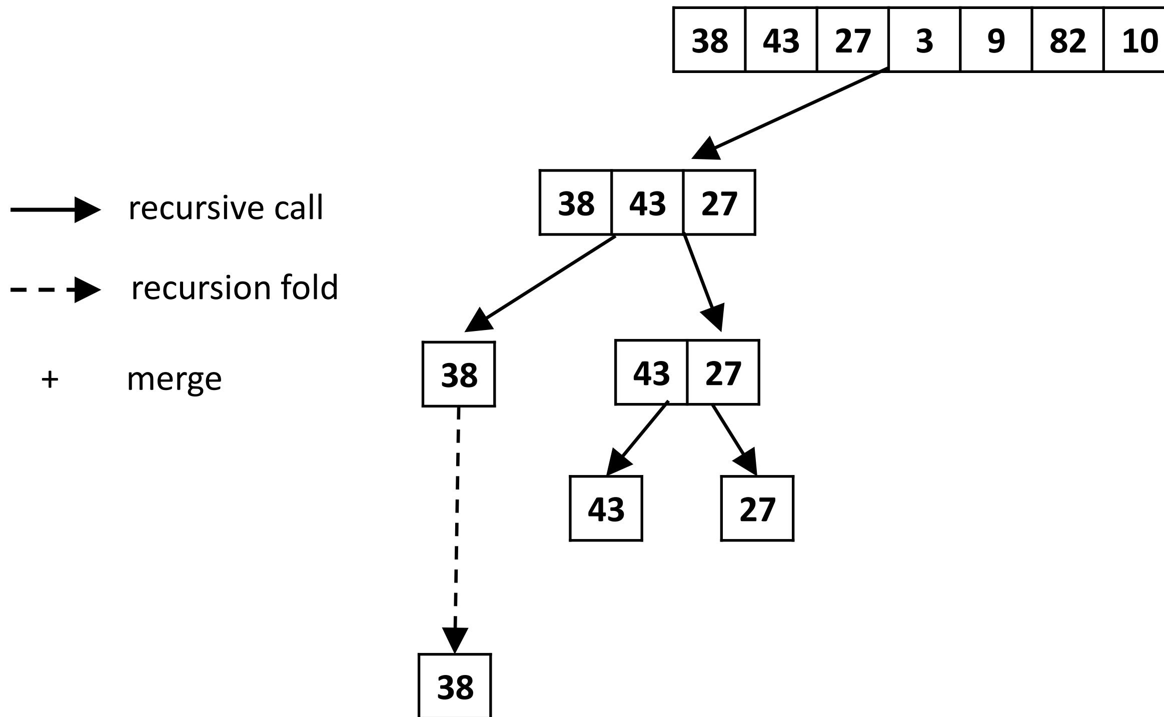
# Merge Sort – example



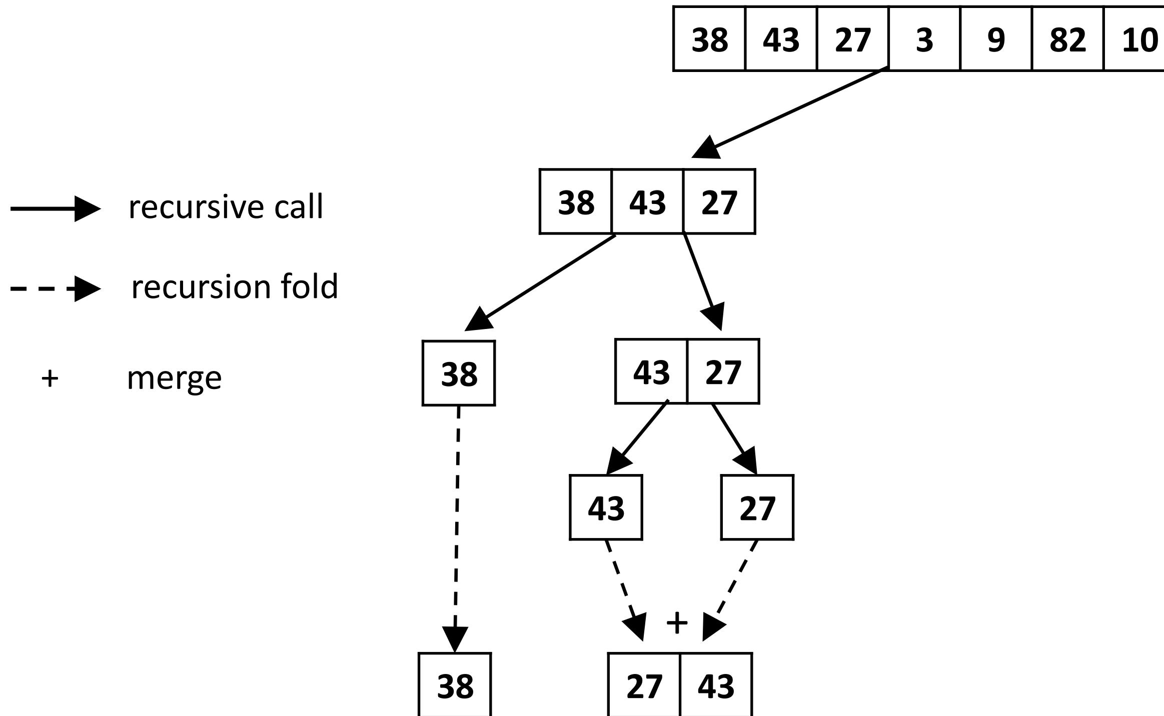
# Merge Sort – example



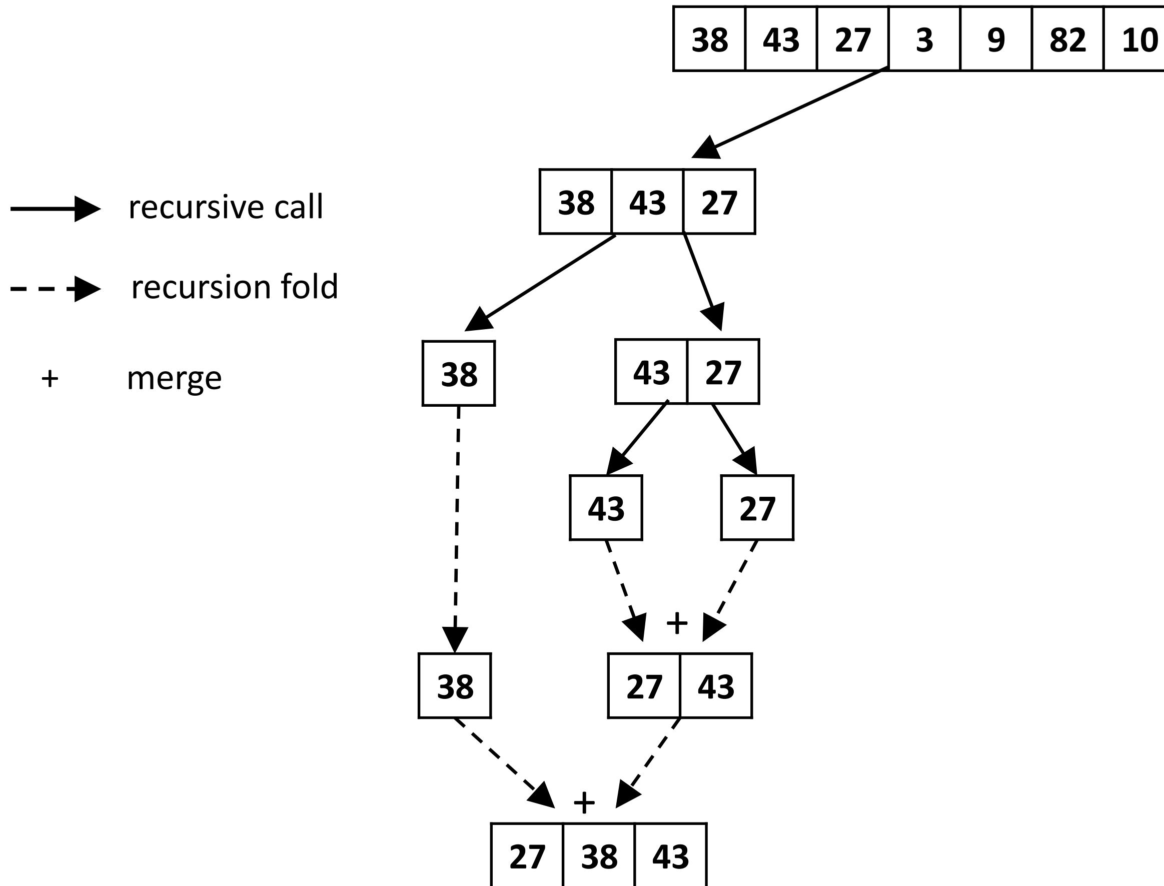
# Merge Sort – example



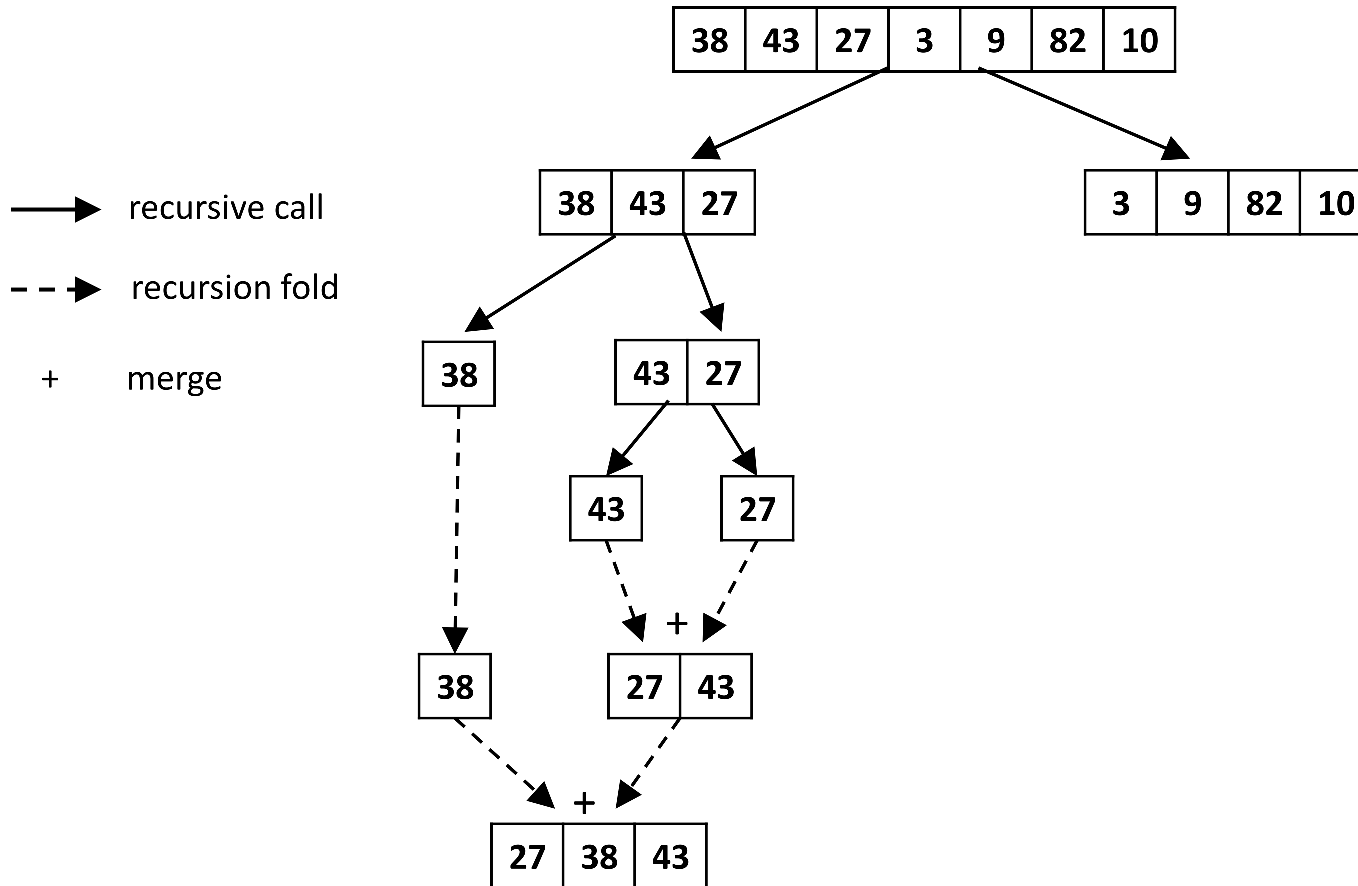
# Merge Sort – example



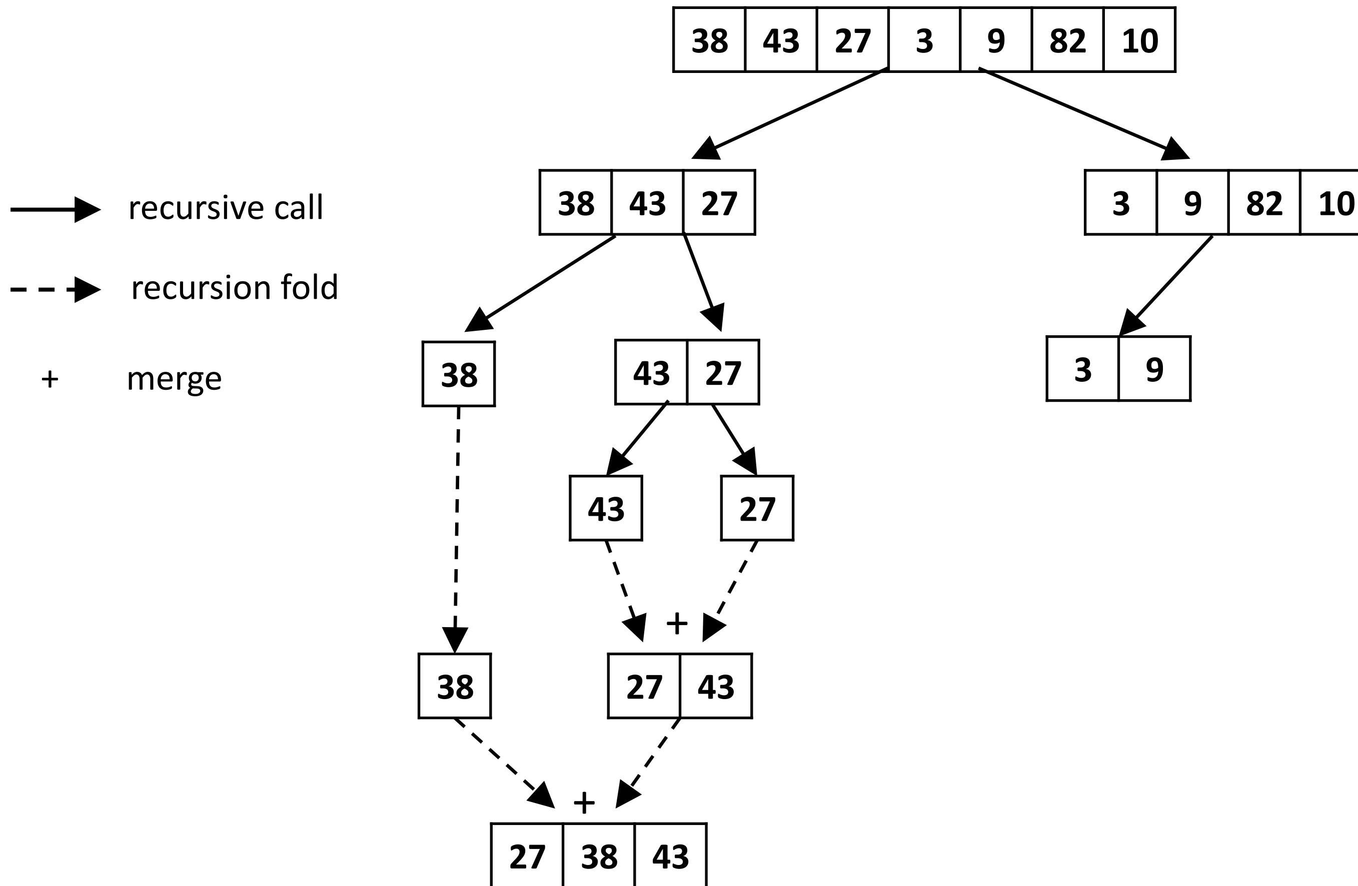
# Merge Sort – example



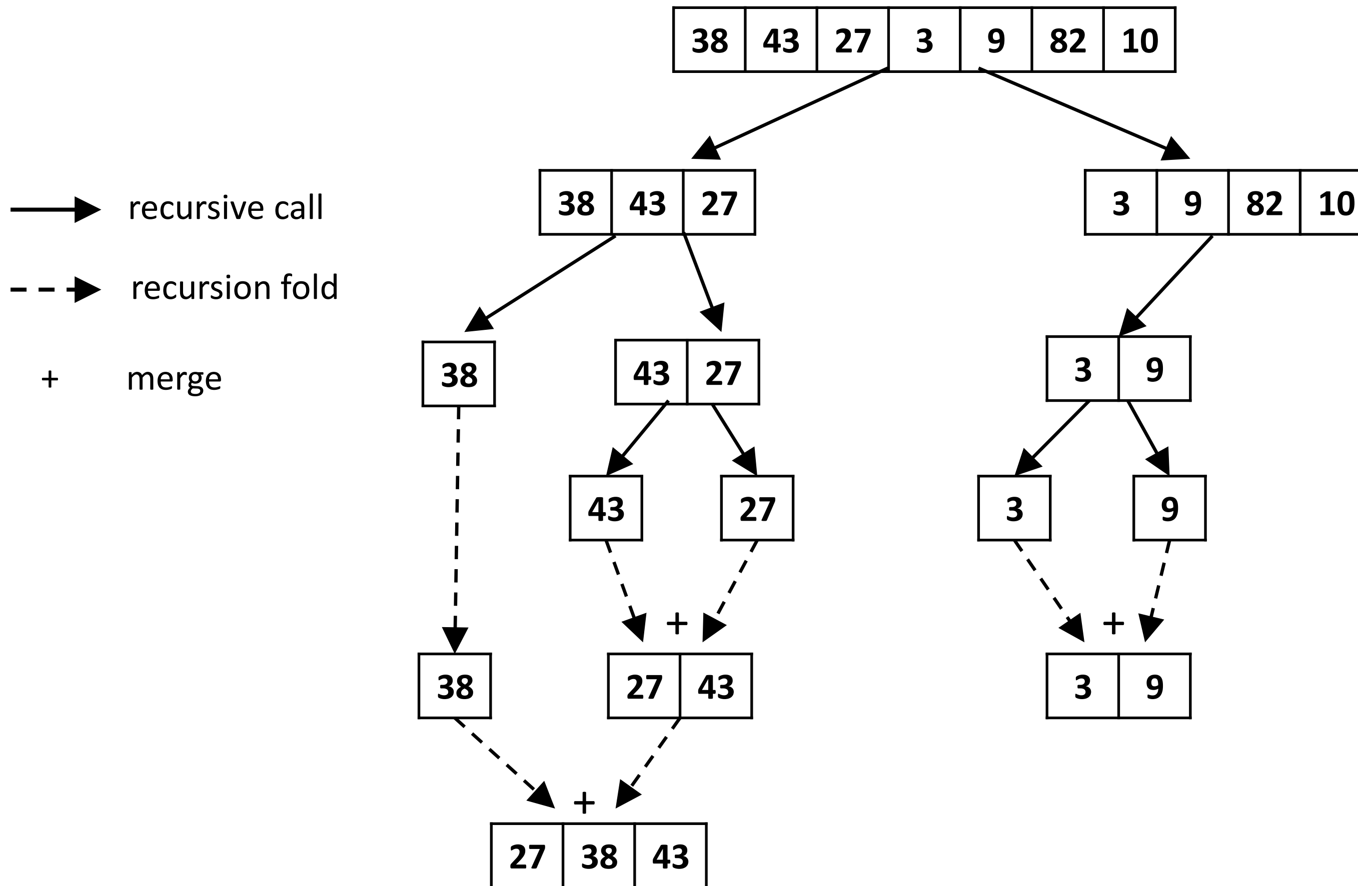
# Merge Sort – example



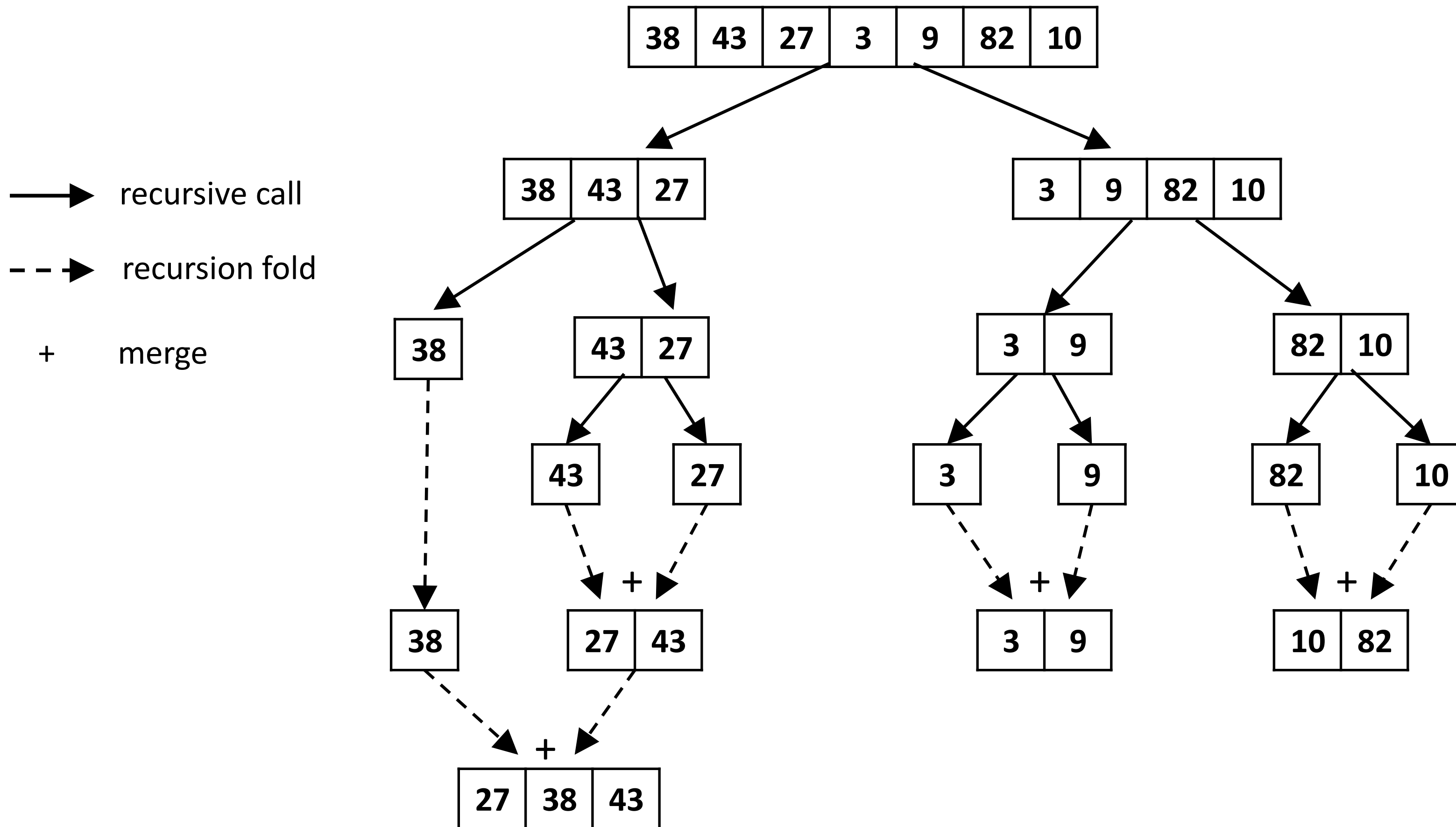
# Merge Sort – example



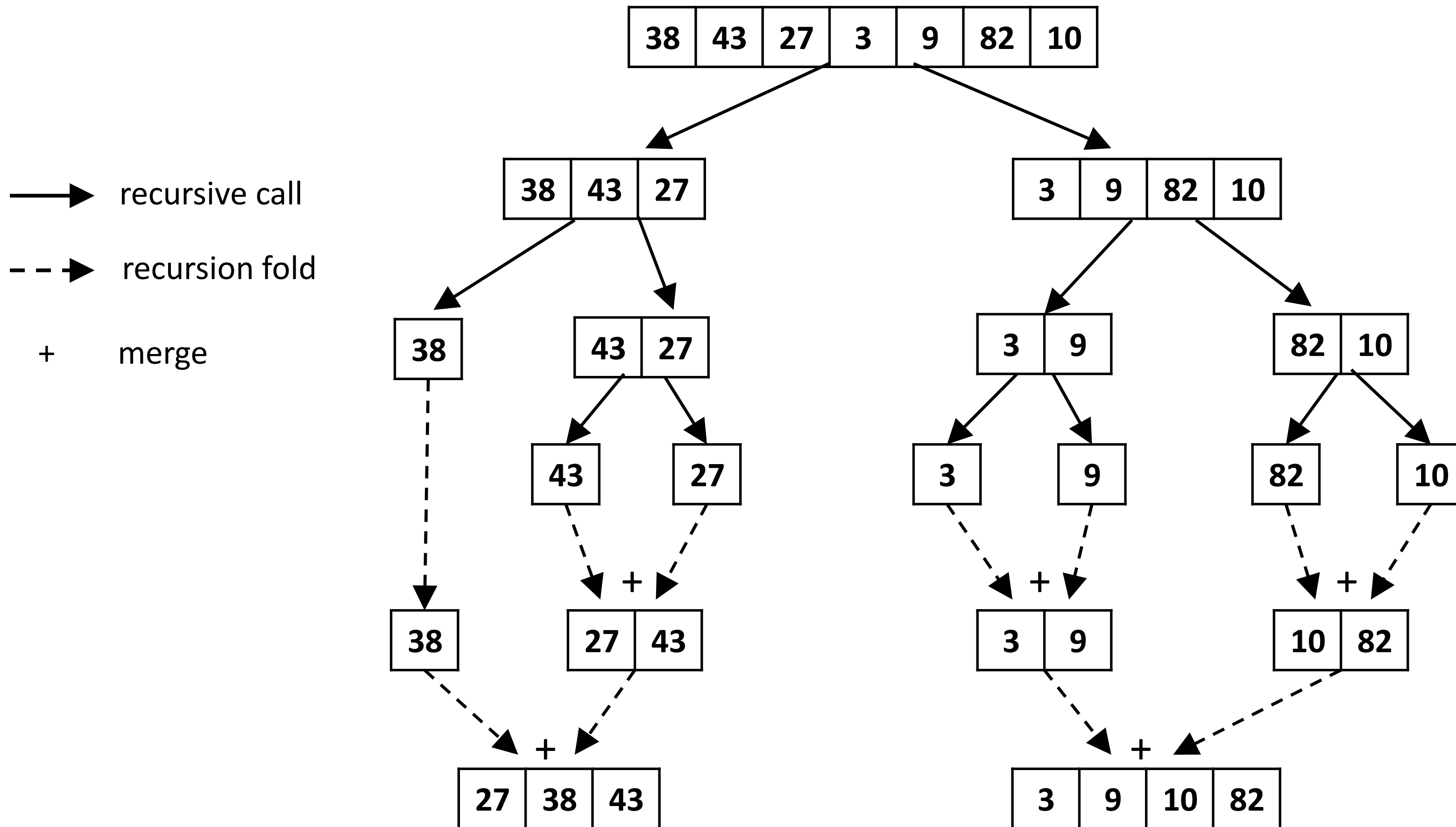
# Merge Sort – example



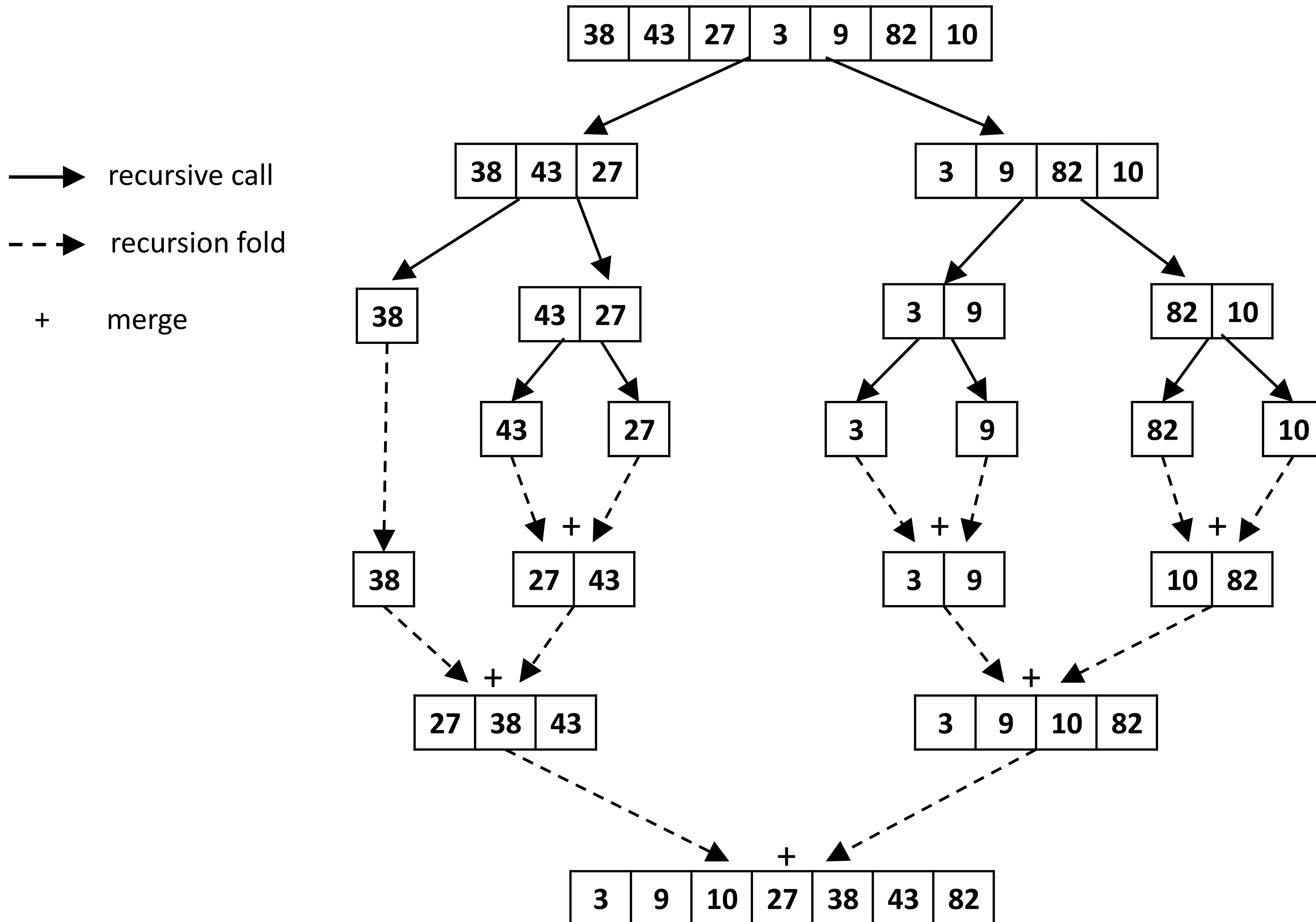
# Merge Sort – example



# Merge Sort – example



# Merge Sort – example



# Merge Sort, cont.

Suppose the input is the following list of length 13

[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21, 0].

We split the list in half, to

[28, 12, 32, 27, 10, 12] and [44, 20, 26, 6, 20, 21, 0].

And recursively sort the two smaller lists, resulting in

[10, 12, 12, 27, 28, 32] and [0, 6, 20, 20, 21, 26, 44].

We then merge the two lists, getting the final, sorted list

[0, 6, 10, 12, 12, 20, 20, 21, 26, 27, 28, 32, 44].

The key to the efficiency of merge sort is the fact that **as we saw**, merging two lists is done in time $O(n+m)$, where $n$ is the length of first list and $m$ is the length of second list.

# Merge (reminder)

```
def merge(A, B):
    ''' Merge list A of size n and list B of size m
        A and B must be sorted! '''
    n = len(A)
    m = len(B)
    C = [0 for i in range(n + m)]

    a=0; b=0; c=0
    while  a<n  and  b<m: #more element in both A and B
          if A[a] < B[b]:
              C[c] = A[a]
              a += 1
          else:
              C[c] = B[b]
              b += 1
          c += 1

    C[c:] = A[a:] + B[b:] #append remaining elements

    return C
```

Time complexity: O(n+m)

# Merge Sort: Python Code

```
def mergesort(lst):
    """ recursive mergesort """
    n = len (lst)
    if n <= 1:
        ______________________________
    else:
        return merge(______________________________ ,\
                     ______________________________)
                  # two recursive calls
```

# Merge Sort: Python Code

```
def mergesort(lst):
    """ recursive mergesort """
    n = len (lst)
    if n <= 1:
      return lst
    else:
      return merge(mergesort(lst[0:n//2]) ,\
                   mergesort(lst[n//2:n]))
              # two recursive calls
```

# Merge Sort: Complexity Analysis

Given a list with n elements, mergesort makes 2 recursive calls. One to a list with $\lfloor n/2 \rfloor$ elements, the other to a list with $\lceil n/2 \rceil$ elements.

The two returned lists are subsequently merged.

On board: Recursion tree and time analysis.

<u>Question</u>:

- Is there a difference between worst-case and best-case for mergesort?

# Merge Sort: Complexity Analysis

The runtime of mergesort on lists with $n$ elements satisfies the recurrence relation $T(n) = c \cdot n + 2 \cdot T(n/2)$ , where c is a constant.

The solution to this relation is $T(n) = O(n \cdot \log n)$.

Recall that in the rec_slice_binary_search function, slicing resulted in $O(n)$ overhead to the time complexity, which is disastrous for searching.

Here, however, we deal with sorting, and an $O(n)$ overhead is asymptotically negligible.

# A Three Way Race

Three sorting algorithms left Haifa at 8am, heading south. Which one will get to TAU first?
We will run them on random lists of lengths 200, 400, 800.

```
>>> from quicksort import *   #the file quicksort.py
>>> from mergesort import *   #the file mergesort.py

3 way race
quicksort
n= 200 0.17896895999999998
n= 400 0.38452376
n= 800 0.87327308
mergesort
n= 200 0.24297283999999997
n= 400 0.49345808000000013
n= 800 1.0856526
sorted # Python 's sort
n= 200 0.007834879999999877
n= 400 0.02002811999999965
n= 800 0.04940151999999998  # I think we have a winner!
```

The results, ahhhm... speak for themselves.

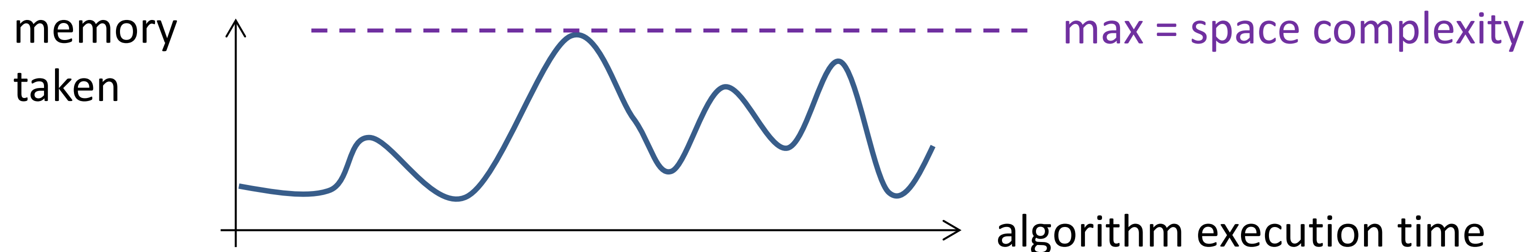
# Recursive Formulae of Algorithms Seen in our Course

| דוגמא | פעולות מעבר לרקורסיה במונחי O( ) | גודל הקריאות רקורסיביות (עד כדי $\pm$1) | נוסחת נסיגה | סיבוכיות |
|---|---|---|---|---|
| max1 (מהתרגול), עצרת | 1 | N-1 | T(N)=1+T(N-1) | O(N) |
| חיפוש בינארי | 1 | N/2 | T(N)=1+T(N/2) | O(log N) |
| Quicksort (worst case) | N | N-1 | T(N)=N+ T(N-1) | $O(N^2)$ |
| Mergesort<br>Quicksort (best case) | N | N/2 ,N/2 | T(N)=N+2T(N/2) | O(N log N) |
| חיפוש בינארי עם slicing | N | N/2 | T(N)=N+T(N/2) | O(N) |
| max2 (מהתרגול) | 1 | N/2 ,N/2 | T(N)=1+2T(N/2) | O(N) |
| האנוי | 1 | N-1, N-1 | T(N)=1+2T(N-1) | $O(2^N)$ |
| פיבונאצ'י | 1 | N-1, N-2 | T(N)=1+T(N-1)+T(N-2) | $O(2^N)$ (לא הדוק) |

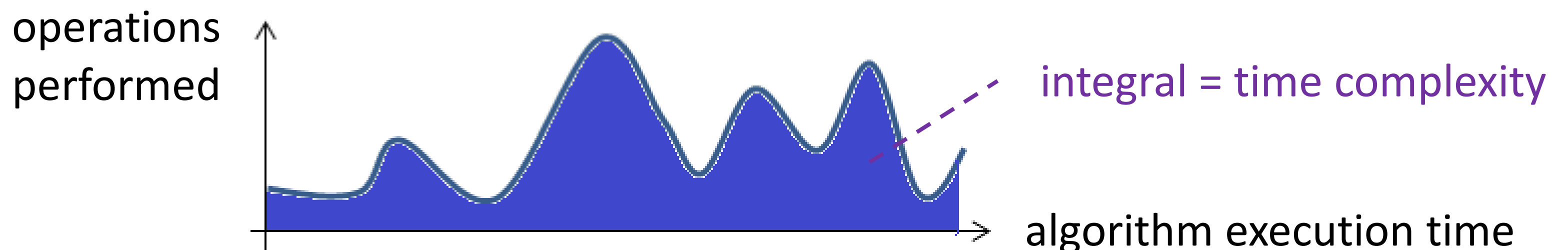
# A word about space (memory) complexity

- A measure of how much memory cells the algorithm needs
  - not including memory allocated for the input of the algorithm

- This is the maximal amount of memory needed at any time point during the algorithm's execution




- Compare to time complexity, which relates to the cumulative amount of operations made along the algorithm's execution



# Space (memory) Complexity

- we will not require space complexity analysis of recursive algorithms
  - note that recursion depth is related to space complexity

- we do require understanding of space allocation requirements in basic scenarios such as:
  - copying (parts of) the input
  - list / string slicing
  - using + operator for lists (as opposed to += or lst.append)

  etc.