Extended Introduction to Computer Science
CS1001.py

Lecture 11: Recursion (2) - Quicksort, Mergesort, Hanoi Towers

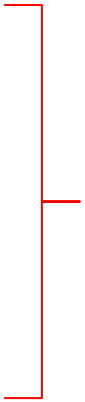
Instructors: Daniel Deutch, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Noam Parzanchevski, Ben Bogin

School of Computer Science
Tel-Aviv University
Winter Semester 2018-9
http://tau-cs1001-py.wikidot.com

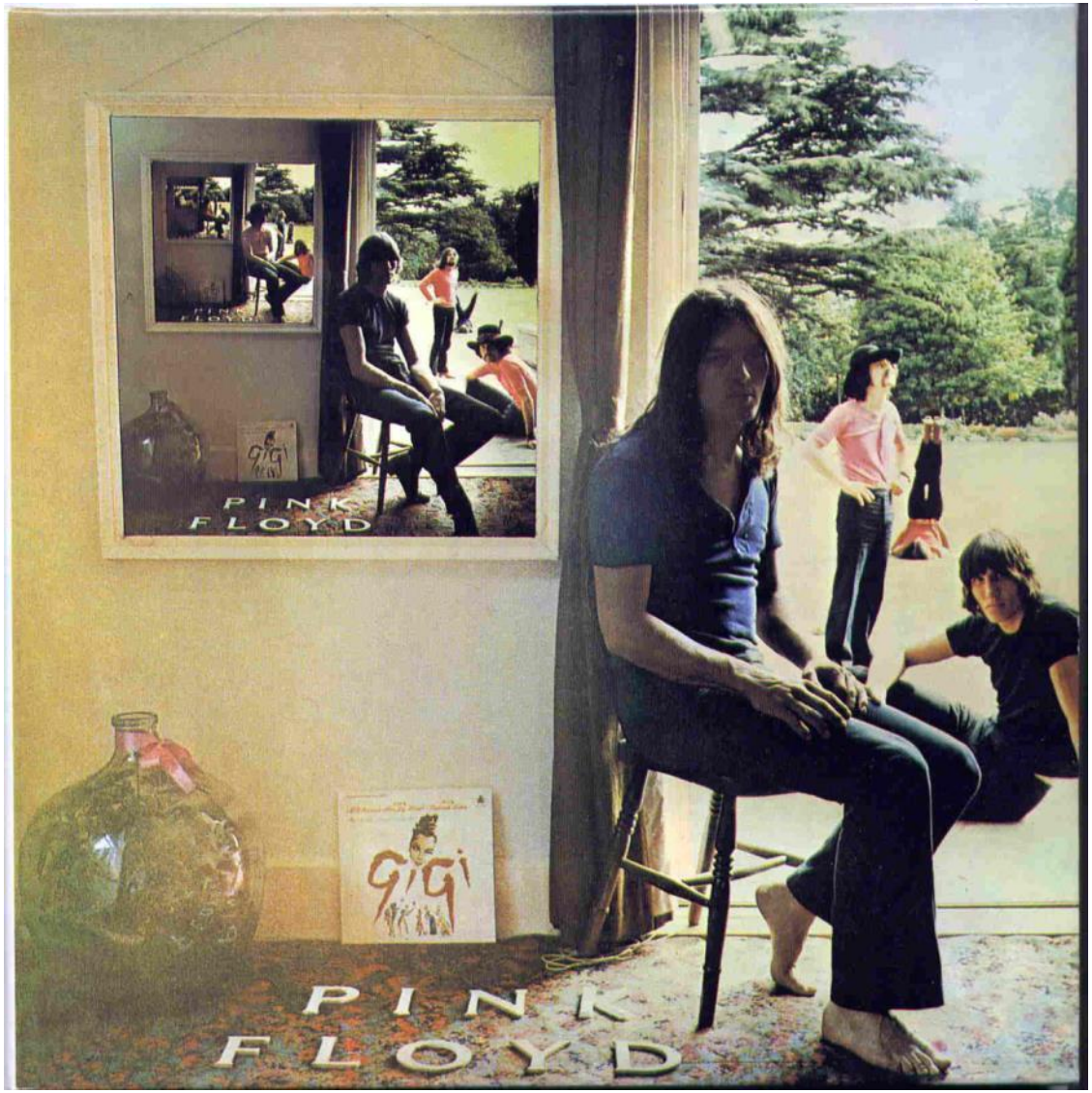
Lectures 10-12: Plan

- Recursion: basic examples and definition
  - Fibonacci
  - factorial
- Binary search - revisited
- Sorting
  - Quick-Sort
  - Merge-Sort
- Towers of Hanoi
- Improving recursion with memoization



today

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 $f(\cdot)$, whose definition contains a call to $f(\cdot)$ 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==0:
        return 1
    else:
        return n * factorial(n-1)
```

Fibonacci (reminder)

A second simple example are the Fibonacci numbers, defined by:

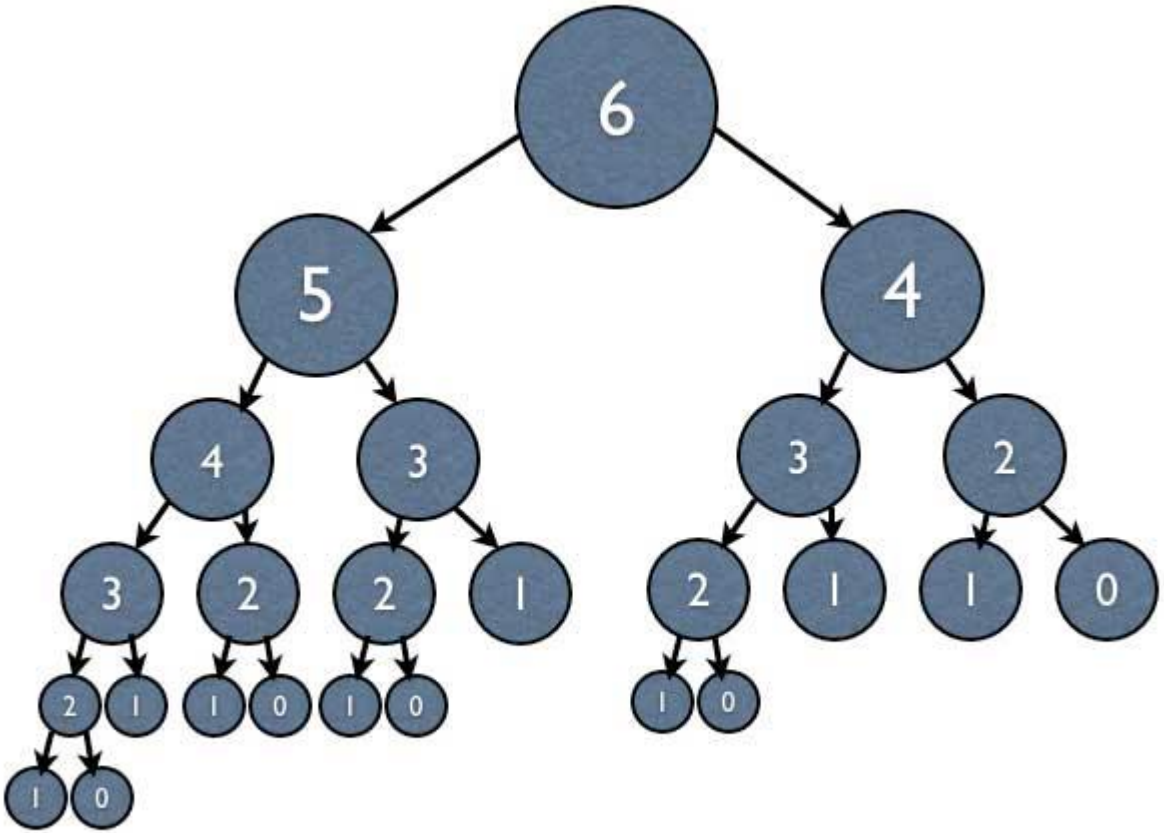
$F_0 = 1, F_1 = 1,$ and for $n > 1, F_n = F_{n-1} + F_{n-2}.$

A Fibonacci numbers function can be programmed in Python, using recursion:

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

Recursion Trees (reminder)

Recursion trees are a common visual representation of a recursive process. For example, here is the recursion tree for fibonacci, for n=6:



Recursion Trees (reminder)

Typically the order of calls from each node is from left to right.

Two special types of nodes in the tree are the root of the tree (a node without a "parent"), and a leaf (a node without children). There is exactly one root, but there can be many leaves.

Recursion trees enable a better understanding of the recursive process, complexity analyses, and may help designing recursive solutions. Two important notions in this context are:

- Time complexity: the total amount of time spent in the whole tree.
- Recursion depth: the maximal length of a path from root to leaf. This is also the maximal number of recursive calls that are simultaneously open.

Recursive Binary Search (reminder)

Here is a recursive implementation of the same task. This code follows the iterative code closely. It passes the key, the original list and two indices (left, right) that indicate the sub-list on which the search is limited to at each step.

```
def rec_binary_search(lst, key, left, right):
    """ recursive binary search.
        passing lower and upper indices """
    if left > right :
        return None

    middle = (left + right)//2

    if key == lst[middle]:
        return middle

    elif key < lst[middle]: # item cannot be in top half
        return rec_binary_search(lst, key, left, middle-1)

    else:                       # item cannot be in bottom half
        return rec_binary_search(lst, key, middle+1, right)
```

Sorting

As we saw, binary search requires preprocessing - sorting. We have seen one simple sorting algorithm – selection sort, with time complexity $O(n^2)$.

We will now see another approach to sorting (out of very many), called quicksort, which employs both randomization and recursion.

(Contents include Game Board, 6 Moving Pieces, 6 Tile Holders, 30 Colored Tiles, Over 300 Topic Cards,
Sand Timer & Instructions. For 2 to 6 players, ages 12 & up.)

Quicksort - Description

- Our input is an unsorted list, say
  [28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]

- We choose a pivot element, simply one of the elements in the list. For example, suppose we chose 20 (the second occurrence).
- We now compare all elements in the list to the pivot. We create three new lists, termed smaller, equal, greater. Each element from the original list is placed in exactly one of these three lists, depending on its size with respect to the pivot.

  - smaller = [12, 10, 12, 6]
  - equal = [20, 20]
  - greater = [28, 32, 27, 44, 26, 21]

- Note that the equal list contains at least one element, and that both smaller and greater are strictly shorter than the original list (why is this important?)

Quicksort - Description (cont.)

- What do we do next?
  - We recursively sort the sublists smaller and greater
  - And then we append the three lists, in order (recall that in Python + means append for lists). Note that equal need not be sorted.

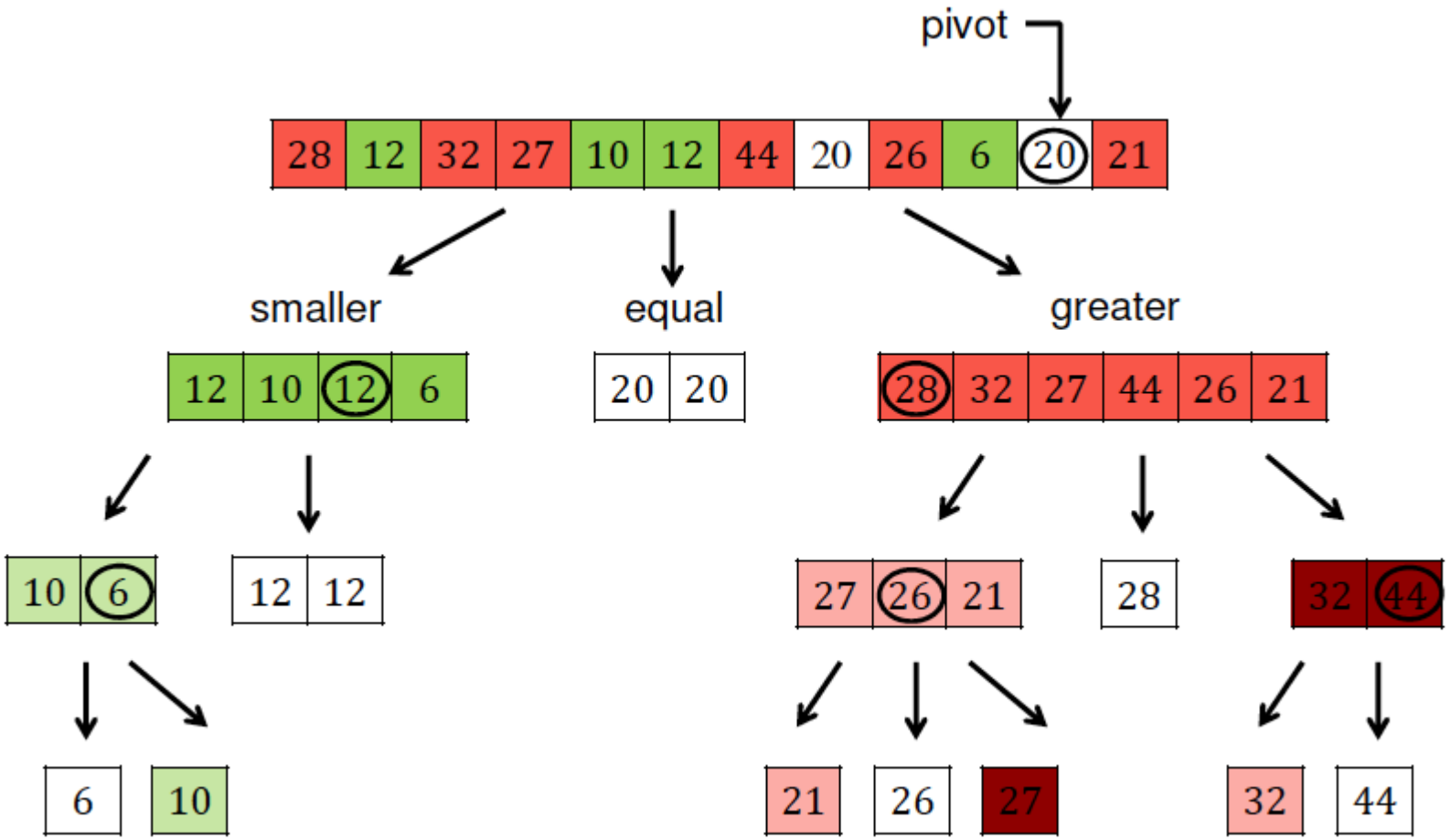
```
return quicksort(smaller) + equal + quicksort(greater)
```

- quicksort(smaller) = [6, 10, 12, 12].
  equal = [20, 20].
  quicksort(greater) = [21, 26, 27, 28, 32, 44].

  [6, 10, 12, 12] + [20, 20] + [21, 26, 27, 28, 32, 44]
  = [6, 10, 12, 12, 20, 20, 21, 26, 27, 28, 32, 44]

- The original list was
  [28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]

Quicksort: A Graphical Depiction



Quicksort: Python Code

```
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) # random element from lst

        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
```

The Use of List Comprehension

- The use of list comprehension is a very convenient mechanism for writing code, and Python is extremely good at it.

- This convenience is good for quickly developing code. It also helps to develop correct code. But this simplicity and elegance do not necessarily imply an efficient execution.

- For example, our quicksort algorithm goes three times over the original list. Furthermore, it allocates new memory for the three sublists.
  - There are versions of quicksort that go over the list only once, and swap original list elements inplace, reusing the same memory. These versions ate more efficient, yet more error prone and generally take longer to develop.
  - Eventually you will choose, on a case by case basis, which style of programming to use.

Quicksort: Convergence and Correctness

Base cases: If the list is either empty or of length one, we do not recurse but return the list itself (in both cases, such lists are sorted).

Convergence: Each time we make a recursive call, its argument (either smaller or greater) is strictly shorter than the current list. When the length hits zero or one, we are at a base case. Thus the quicksort algorithm always converges (no infinite executions).

Correctness: An inductive argument:
If smaller and greater are sorted, then

quicksort(smaller) + equal + quicksort(larger)

is a sorted list. Its elements are the same as the original list (multiplicities included), so the outcome is the original list, sorted.
QED

Quicksort: Complexity

To analyze the time complexity of quicksort, we will use recursion trees (on the board).

Quicksort: Complexity (worst case)

- The worst case running time (WCT) to sort a list of n elements occurs if at each invocation of the function, we choose either the minimum element as the pivot, or the maximum element. This makes either smaller or greater to be an empty list.

- The resulting run time is $O(n^2)$ (to be explained on the board, using recursion trees).

- The worst best case run time satisfies the recurrence relation

$$WCT(n) = c \cdot n + WCT(n-1)$$

  where c is some constant.

Quicksort: Complexity (best case)

- The best case running time (BCT) occurs if we are lucky at each invocation of the function, and pivot is the median, splitting the list to two equal parts (up to one, if number of list elements is even).

- The resulting run time is $O(n \cdot logn)$ (to be explained on the board, using recursion trees).

- The best case run time satisfies the recurrence relation

$$BCT(n) = c \cdot n + 2 \cdot BCT(n/2).$$

where c is some constant.

Quicksort: Average Complexity

- A more complicated analysis can be done for the average running time.
- Average over what?
- It can be shown that the best case and the average running times to sort a list of n elements are both $O(n \cdot logn)$
- The best case constant in the big *O* notation is slightly smaller than the "average case" constant.
- Rigorous analysis is deferred to the data structures course.

Deterministic Quicksort

- We could take the element with the first, last, middle or any other index in the list as the pivot.

- For example:

```
pivot = lst[0]
```

- This would usually work well (assuming some random distribution of input lists).

- However, in some cases this choice would lead to poor performance (even though the algorithm will always converge).

- For example, if the input list is already sorted (or close to sorted), and the pivot is the first or last element.

Quicksort: Pivot Selection, cont.

- Instead of a fixed choice, the recommended solution is choosing the pivot at random.
- With high probability, the randomly chosen pivot will be neither too close to the minimum nor too close to the maximum.
- This implies that both smaller and greater are substantially shorter than the original list, and yields good performance with high probability (at this point this is an intuitive claim, nothing rigorous.)

Running Quicksort on Sorted and Random Lists

```
def ordlist(n):
    return [i for i in range (0,n)]
```

```
import random

def randlist(n):
    """ generates a list of n random elements
        from [0,...,n**2] """
    return [random.randint(0,n**2) for i in range(0,n)]
```

Running Quicksort

```
import time

for n in [200, 400, 800]:
    print("n=", n, end=" ")
    lst = ordlist(n)
    #lst = randlist(n)

    t0 = time.clock()

    for i in range(100):
        quicksort(lst)
        #sort is not inplace, so lst remains unchanged!

    t1 = time.clock()

    print(t1-t0)
```

Time Measurements

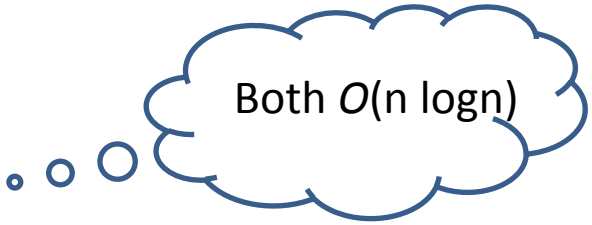
Random quicksort

Ordered list

n= 200 0.10658217853053813
n= 400 0.23033834734888864
n= 800 0.48683051048205617

Random list

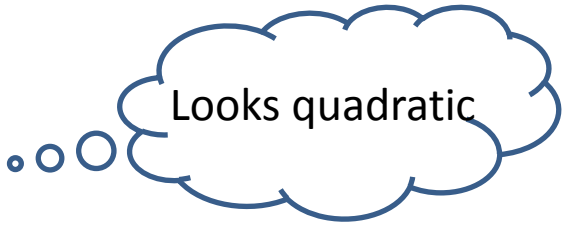
n= 200 0.11578862173792887
n= 400 0.24735086264755812
n= 800 0.49966520589048125



Deterministic quicksort

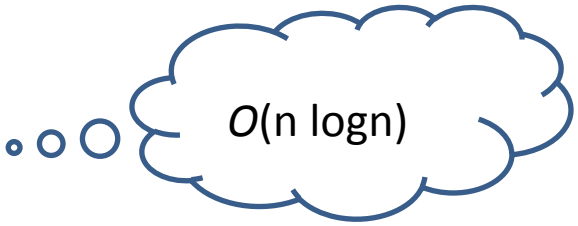
Ordered list

n= 200 0.5891511705949701
n= 400 2.1920545619645466
n= 800 8.422338821114138



Random list

n= 200 0.08930474949662441
n= 400 0.1763828023214497
n= 800 0.36725255635832443



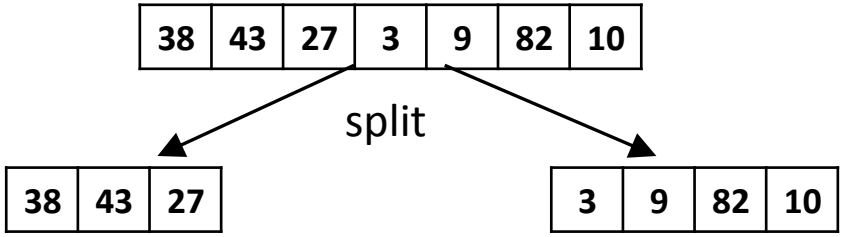
Can you explain why for random lists the deterministic QS is slightly better?

Merge Sort

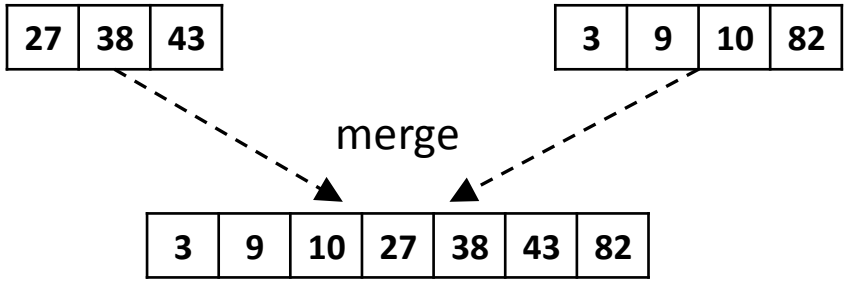
Merge Sort

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 sorted these 2 halves, we would then only need to merge them.



Well, does anybody know a sorting algorithm to apply on each half?

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.

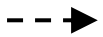
Each half is sorted recursively.

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

Merge Sort – example

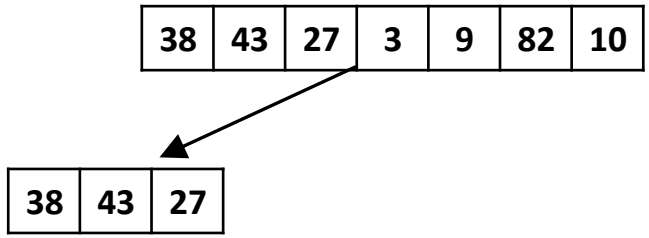
38	43	27	3	9	82	10


recursive call


recursion fold

\+ merge

Merge Sort – example

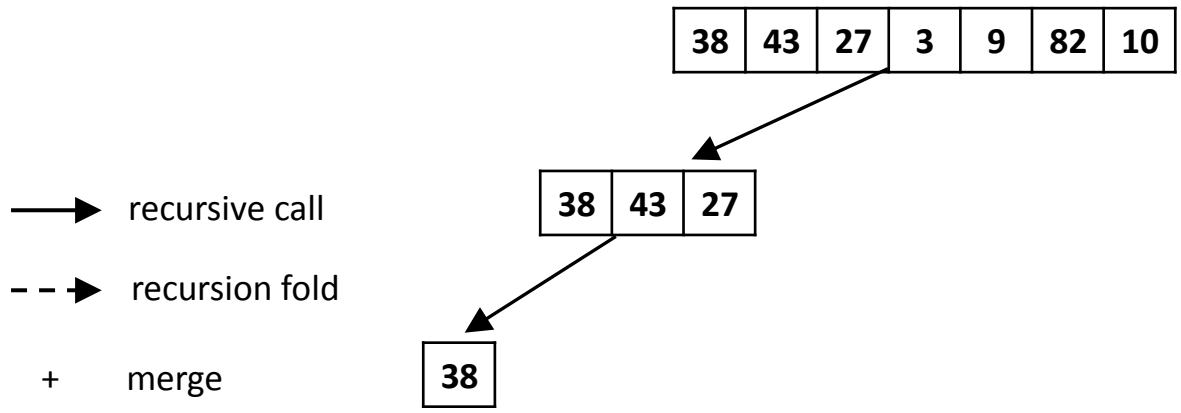


recursive call

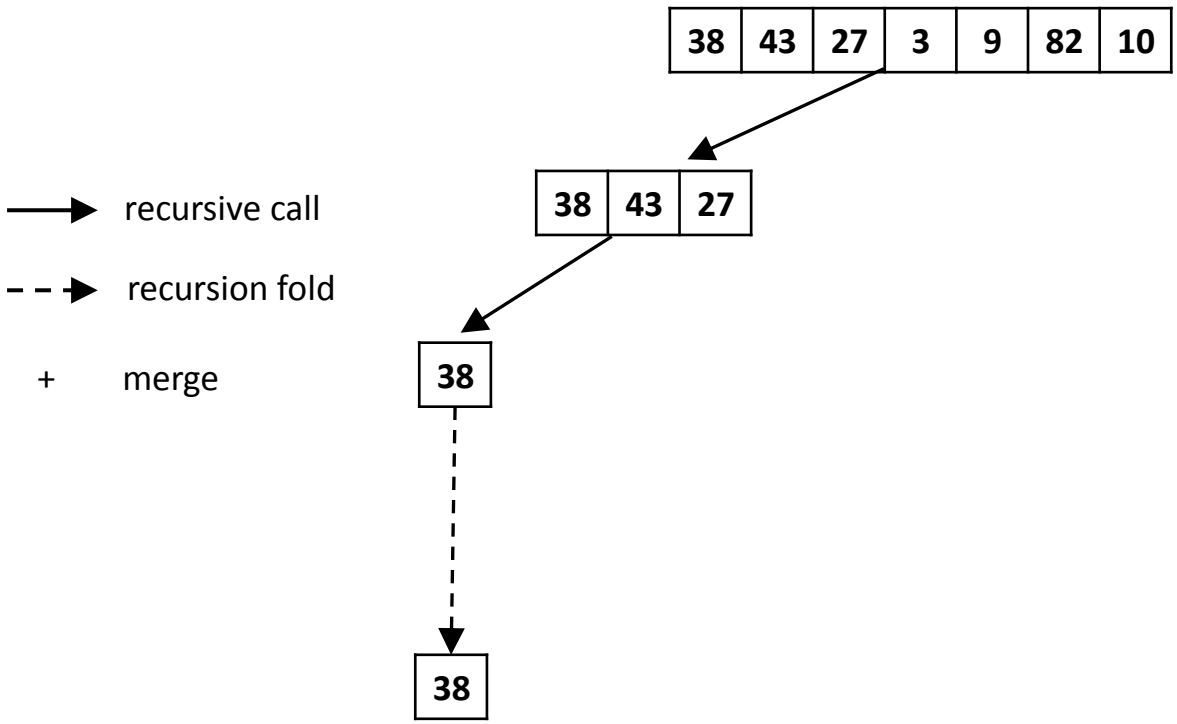
recursion fold

\+ merge

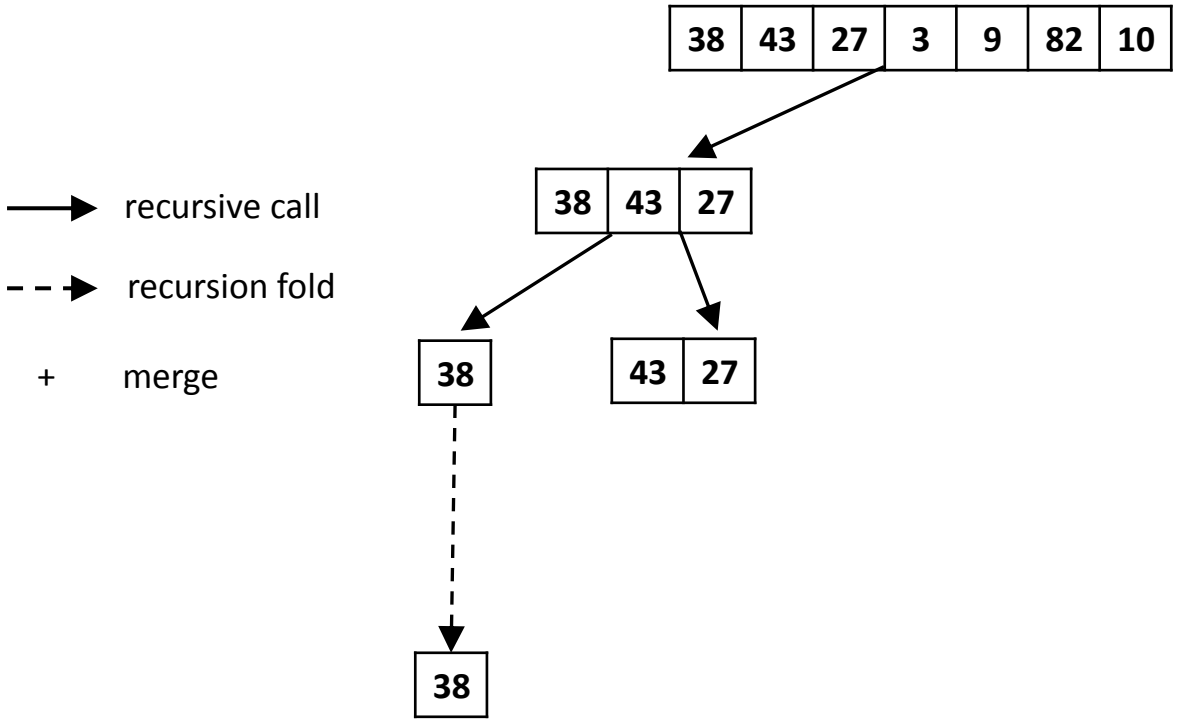
Merge Sort – example



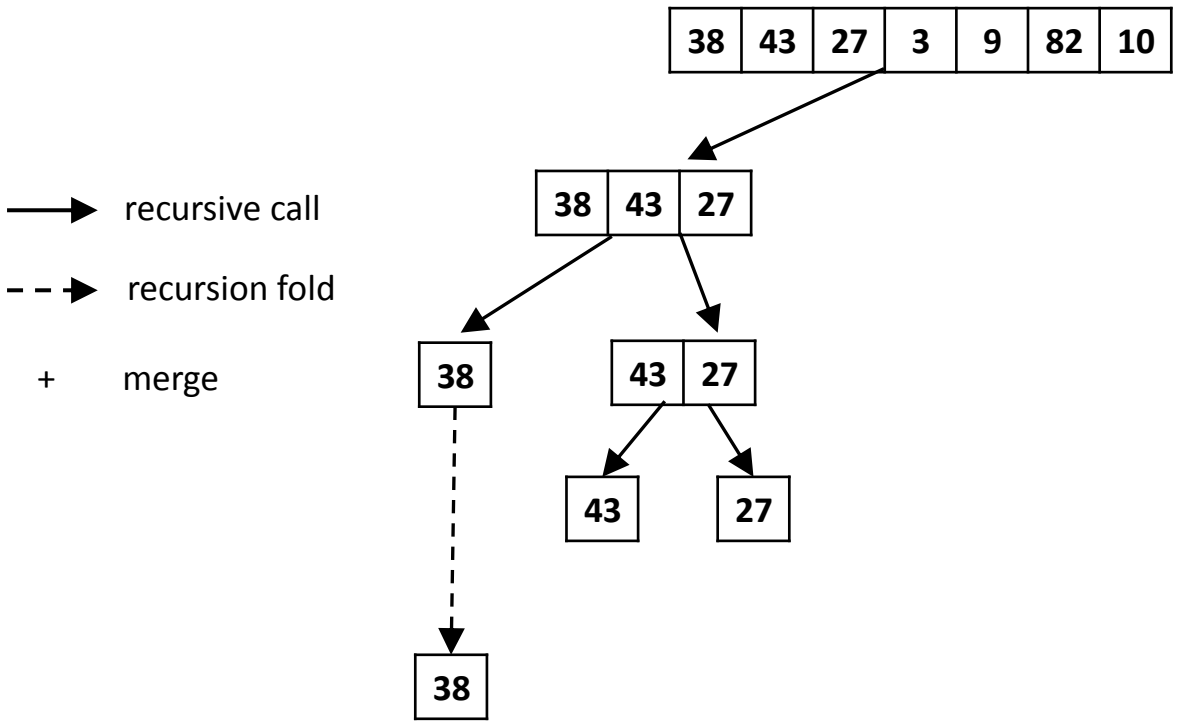
Merge Sort – example



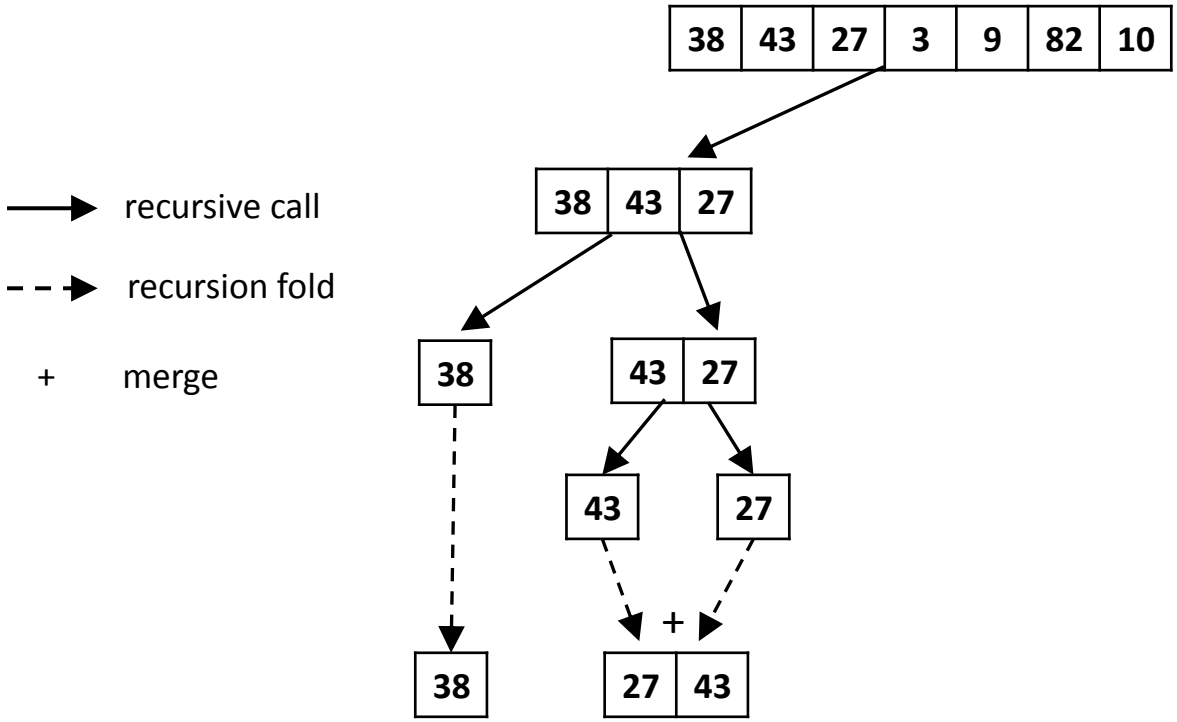
Merge Sort – example



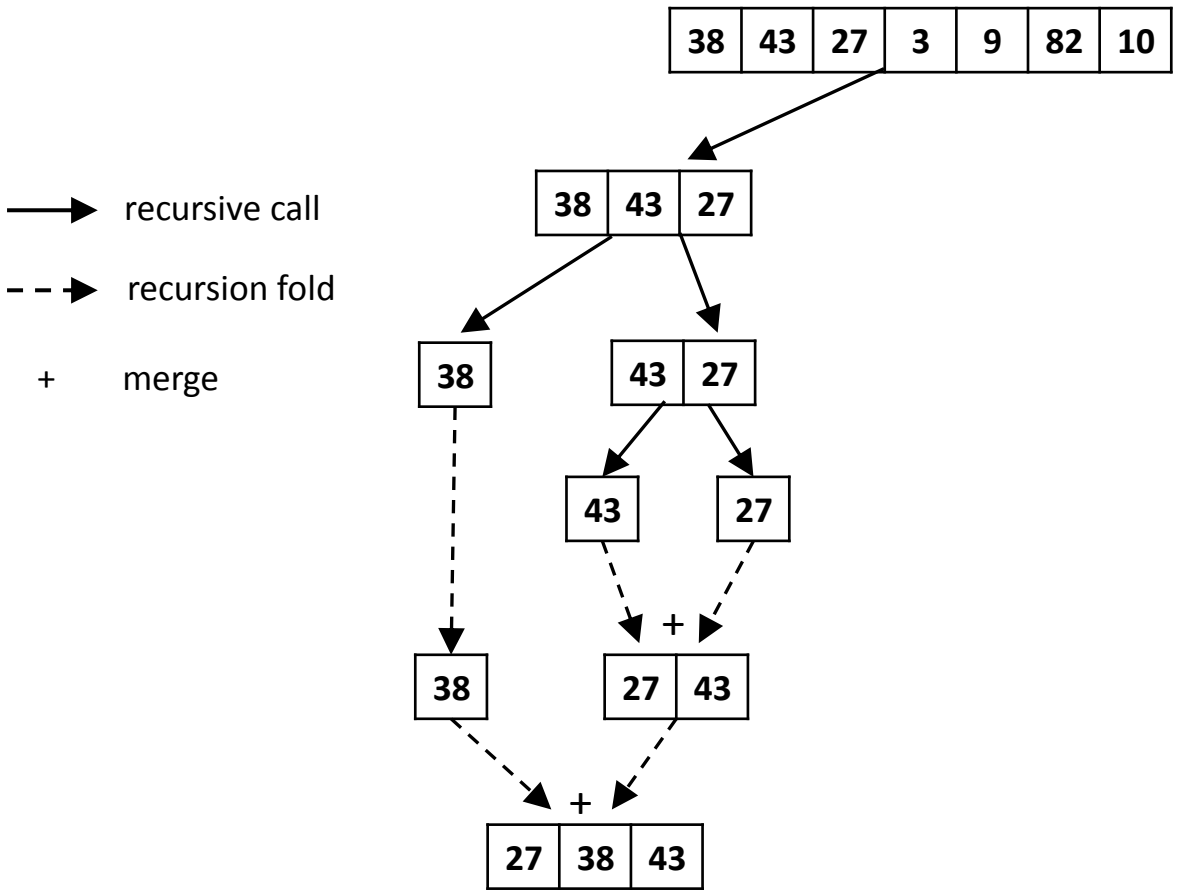
Merge Sort – example



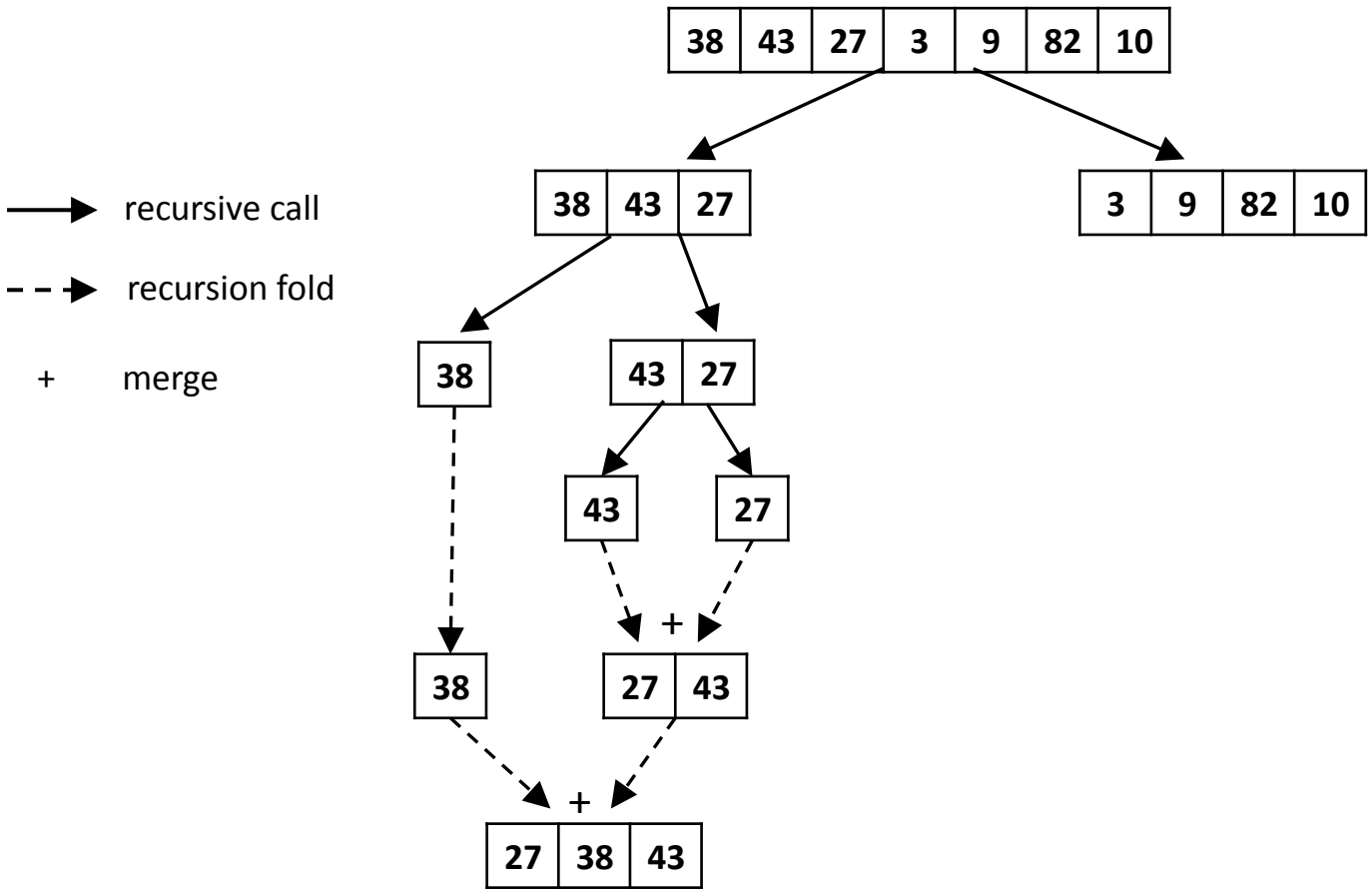
Merge Sort – example



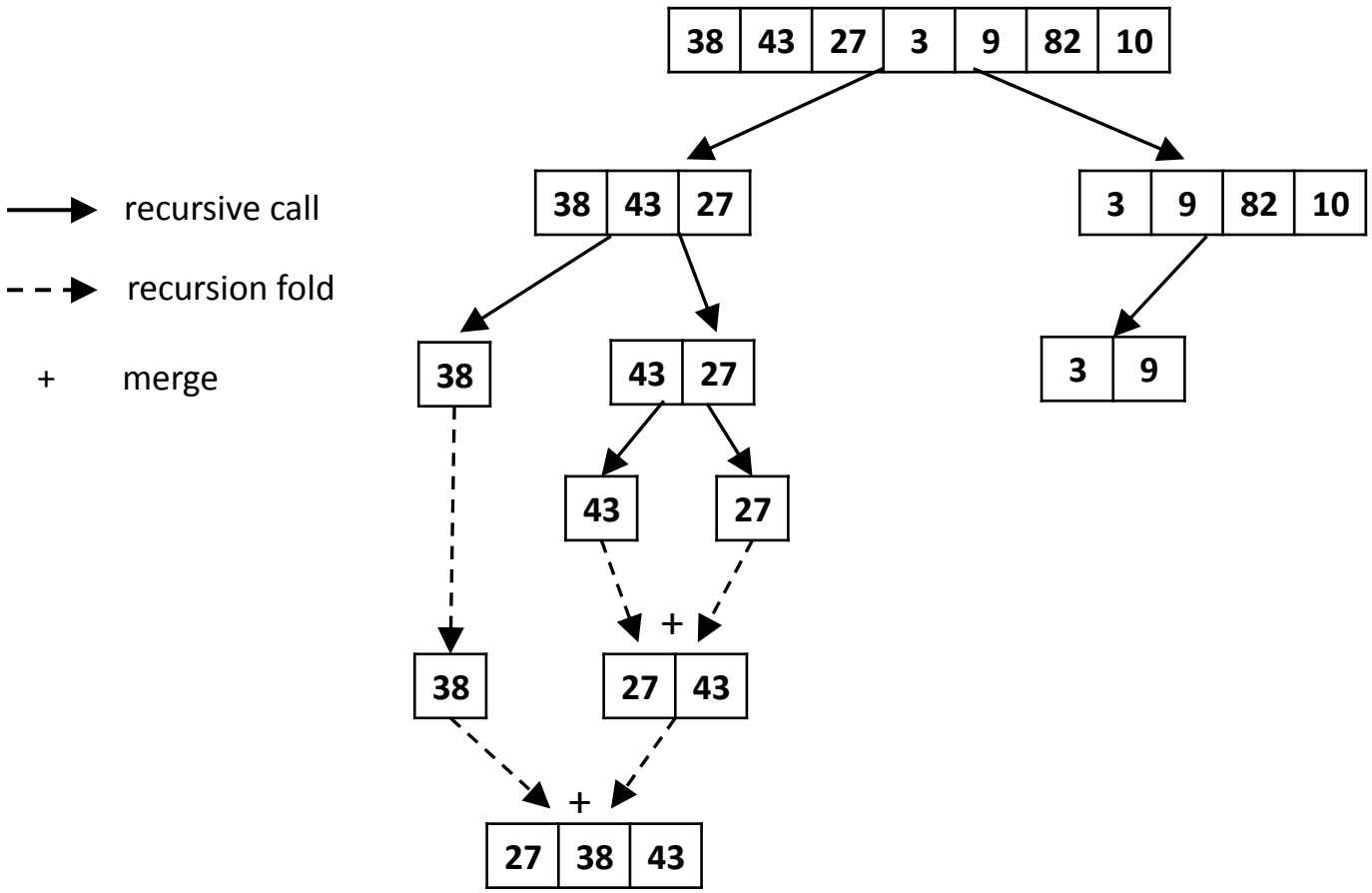
Merge Sort – example



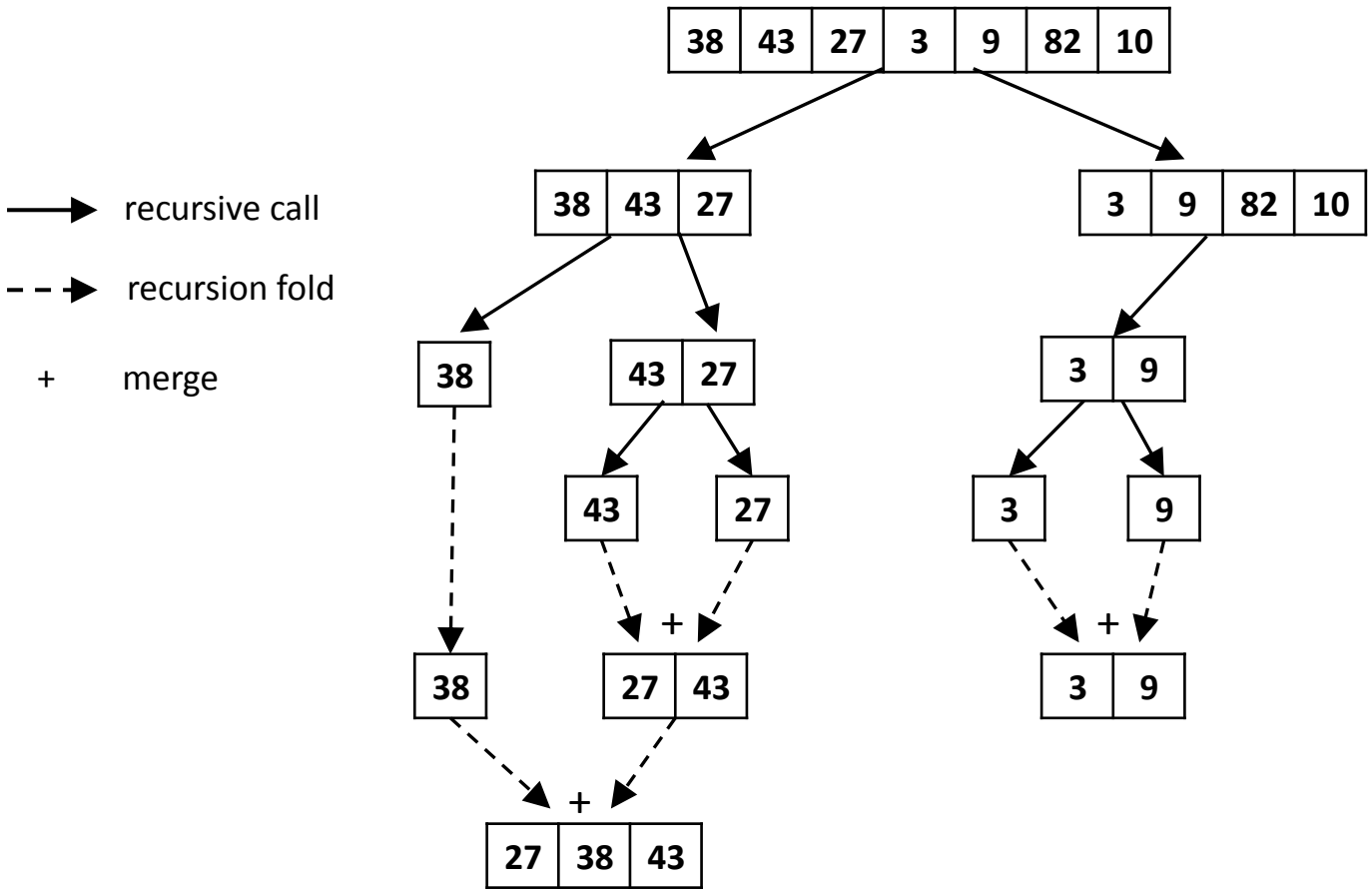
Merge Sort – example



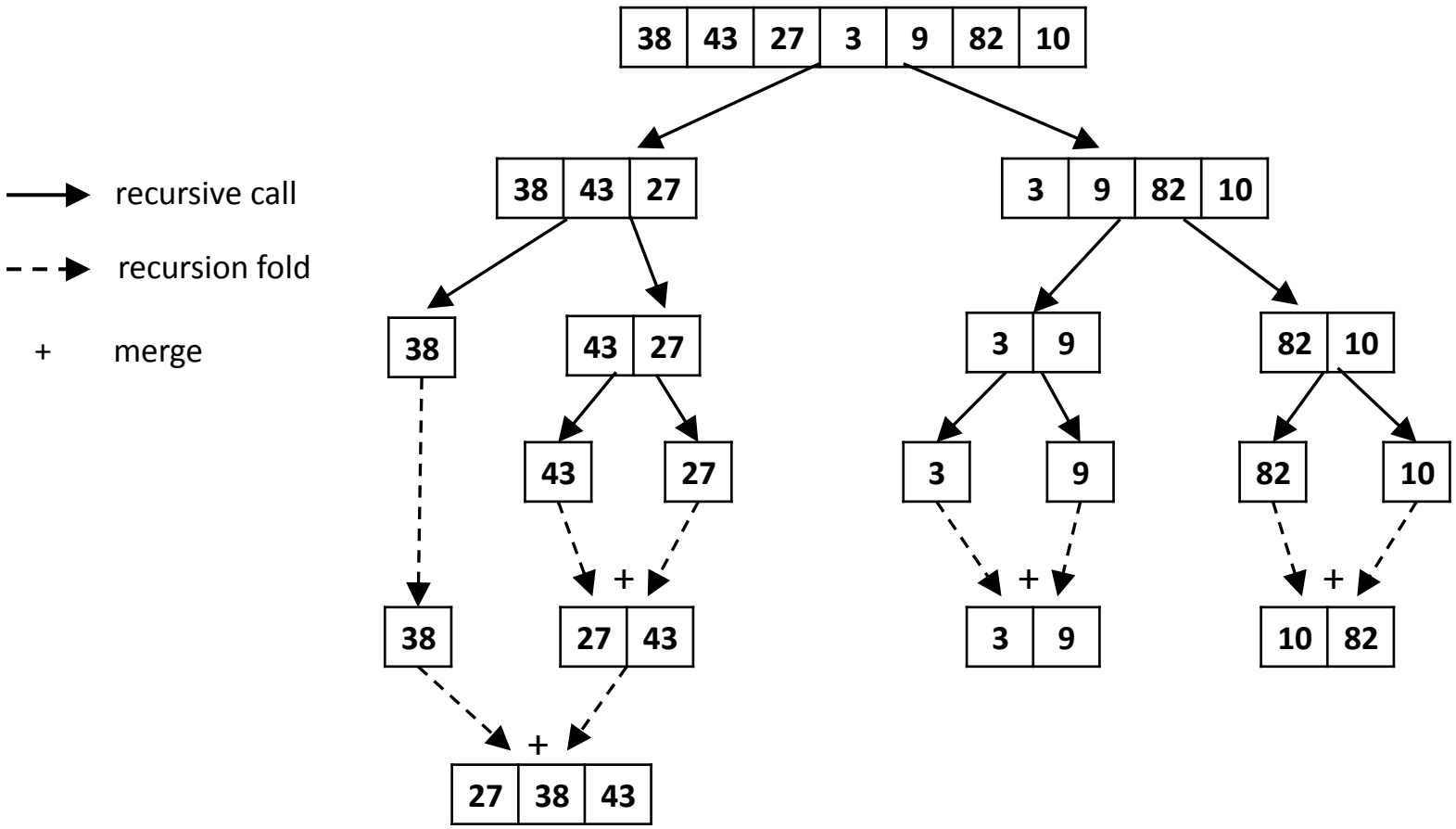
Merge Sort – example



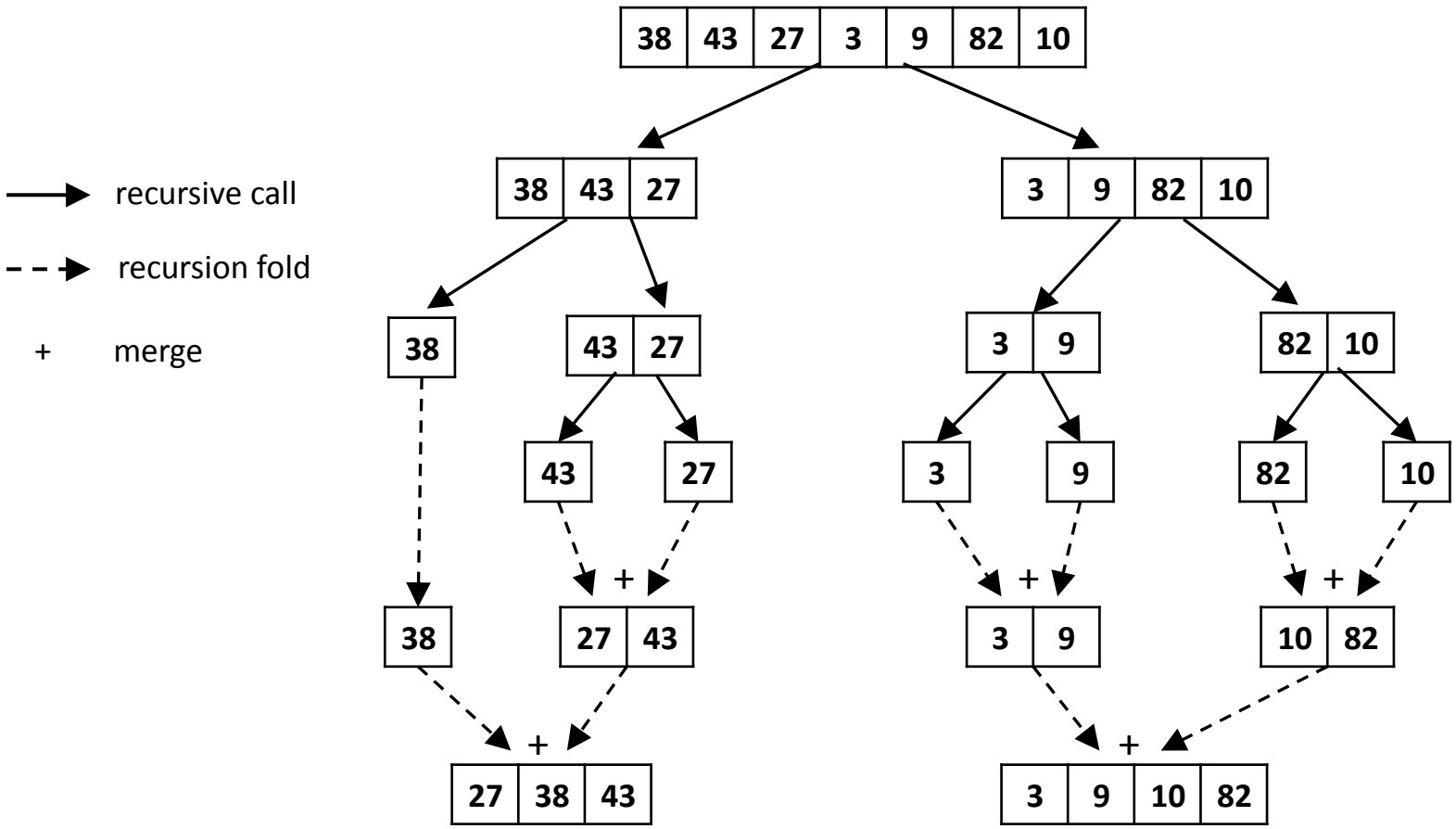
Merge Sort – example



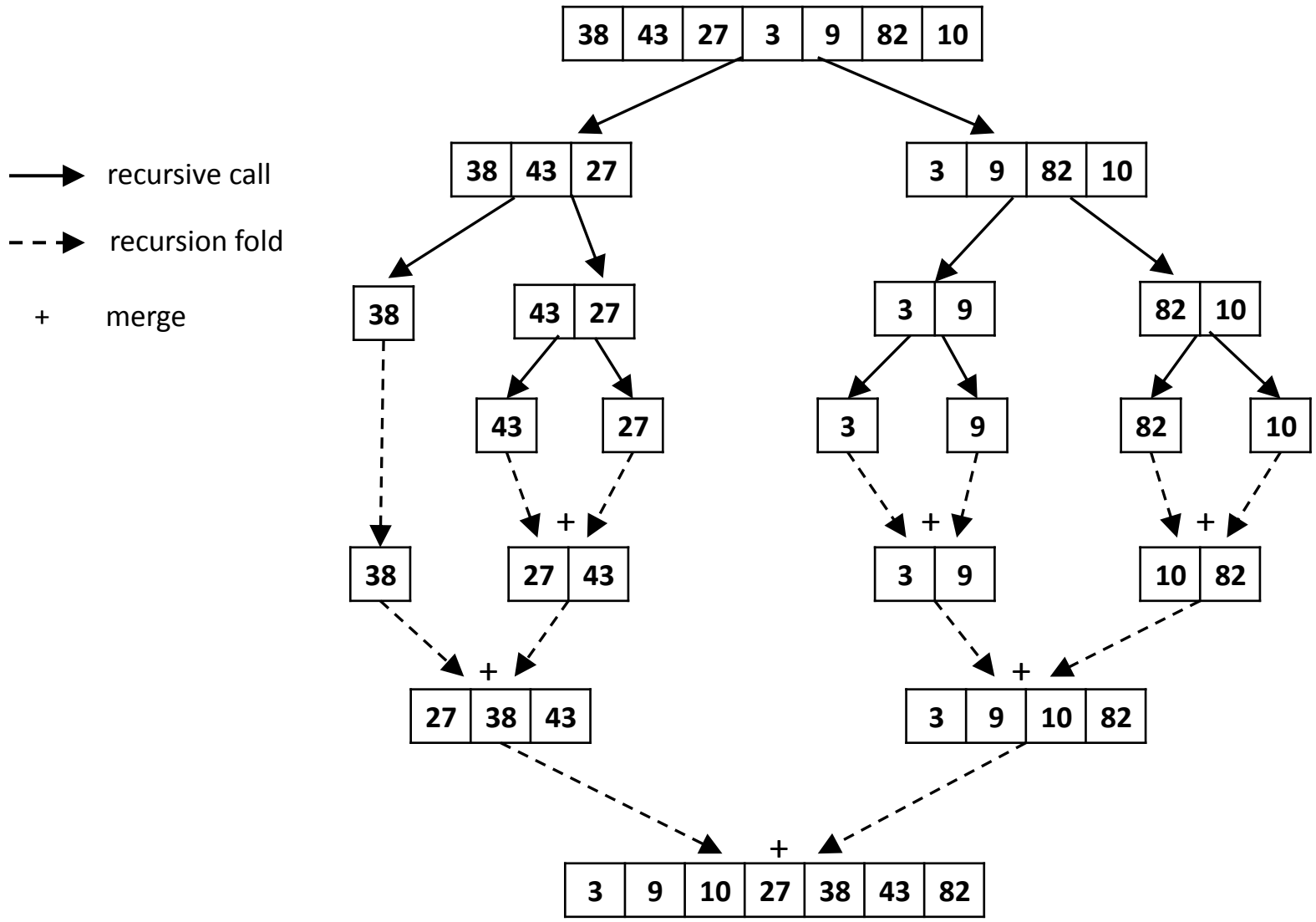
Merge Sort – example



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.

Merging sorted lists (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:
      return _____________________________
    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 complexity analysis.

Question:

- 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, for both best and worst case, 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 binary search.

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.

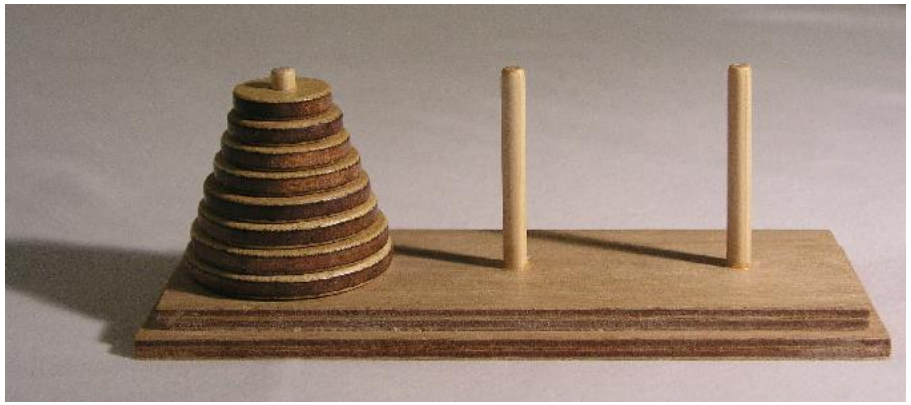
```
>>> from quicksort import *   #the file quicksort.py
>>> from mergesort import *   #the file mergesort.py

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.

Towers of Hanoi

Towers of Hanoi

Towers of Hanoi is a well known mathematical puzzle, and no class on recursion, including this one (a recursive claim in itself :-), is complete without discussing it.



(figure from Wikipedia)

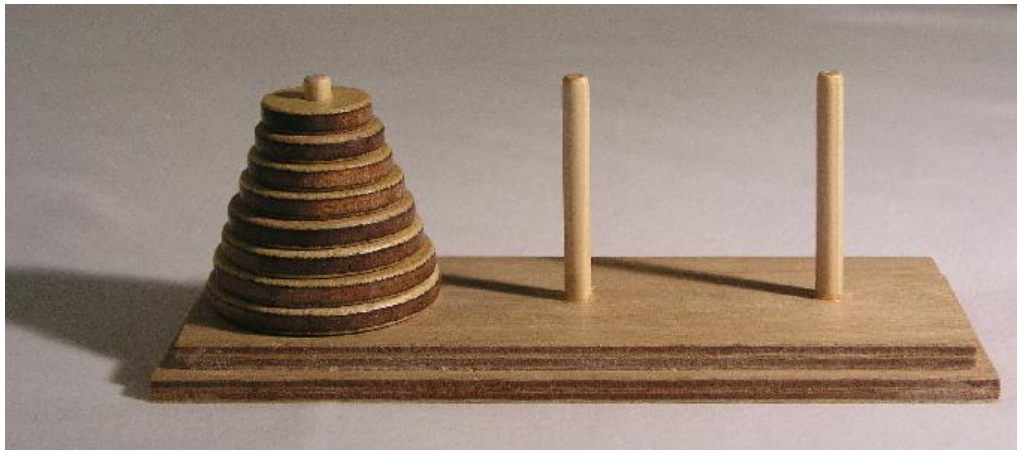
Towers of Hanoi - origin

The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle will be completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it.

(text from Wikipedia)

Towers of Hanoi - Description

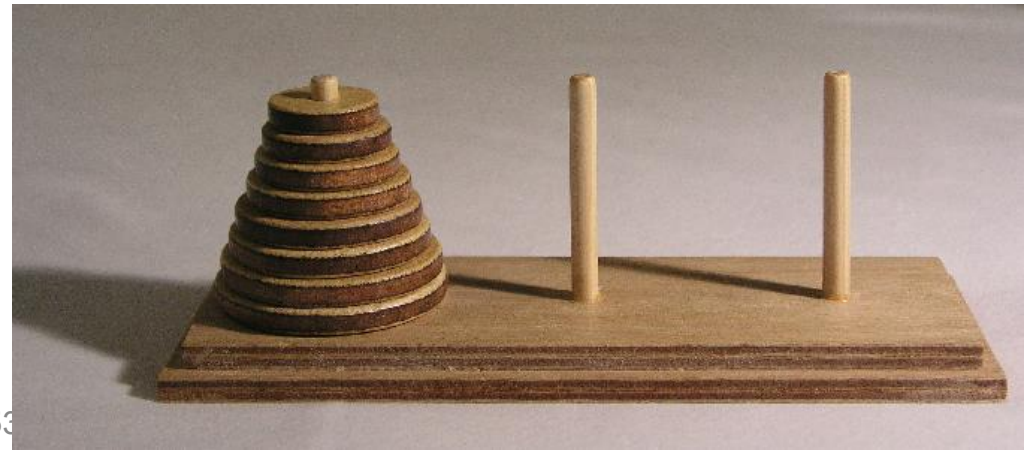
- There are three rods, named A, B, C, and n disks of different sizes which can be placed onto any rod.
- The puzzle starts with all n disks in a stack in ascending order of size on one rod, say A, so that the smallest is at the top (see figure).



(figure from Wikipedia)

Towers of Hanoi: Rules of Game

- The objective of the puzzle is to move the entire stack of all n disks to another rod, say C, obeying the following rules:
  - Only one disk may be moved at a time.
  - Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
  - No disk may be placed on top of a smaller disk.



(figure and some text from Wikipedia)

Towers of Hanoi: recursive view

In order to think about a recursive solution, we should first have a recursive definition of a Hanoi tower:

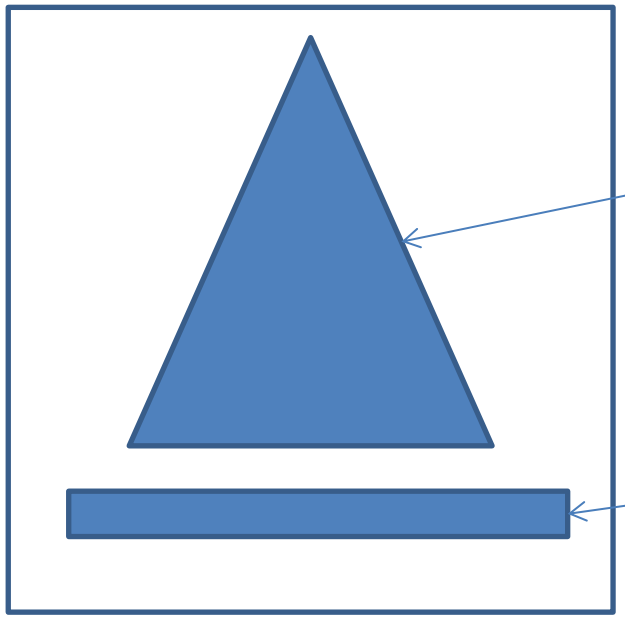
it is either empty,

This is the base case

or it is a tower on top of a larger disk (larger than all the disks in the tower on top).

Another possibility is to let the base case be a tower of one disk

Schematically:



Towers of Hanoi: The algorithm

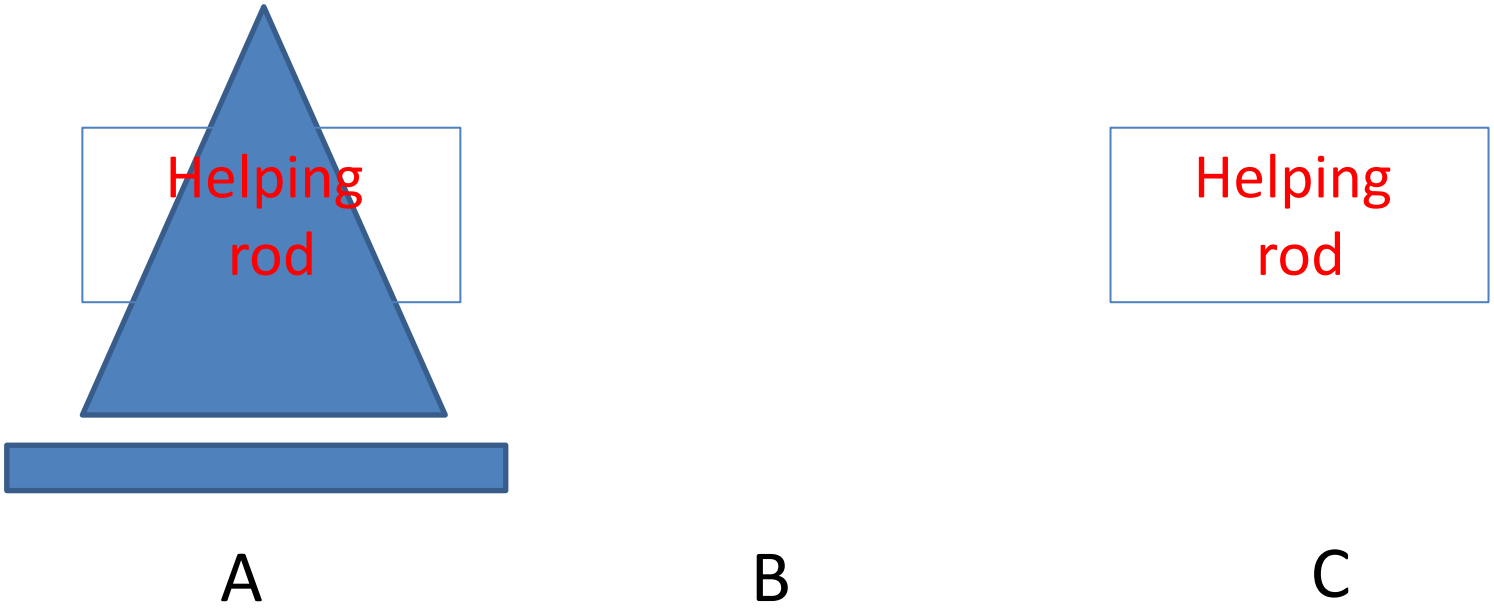
We can now describe a recursive algorithm to move a stack of n disks from rod A to rod C using rod B as a helping rod.

In the base case , when n=0, there is nothing to do.

The non- base case (n>0) will be shown in the next slide.

[If we chose n=1 as the base case, then the base case would be to move the single rod from A to C]

Towers of Hanoi: recursive algorithm

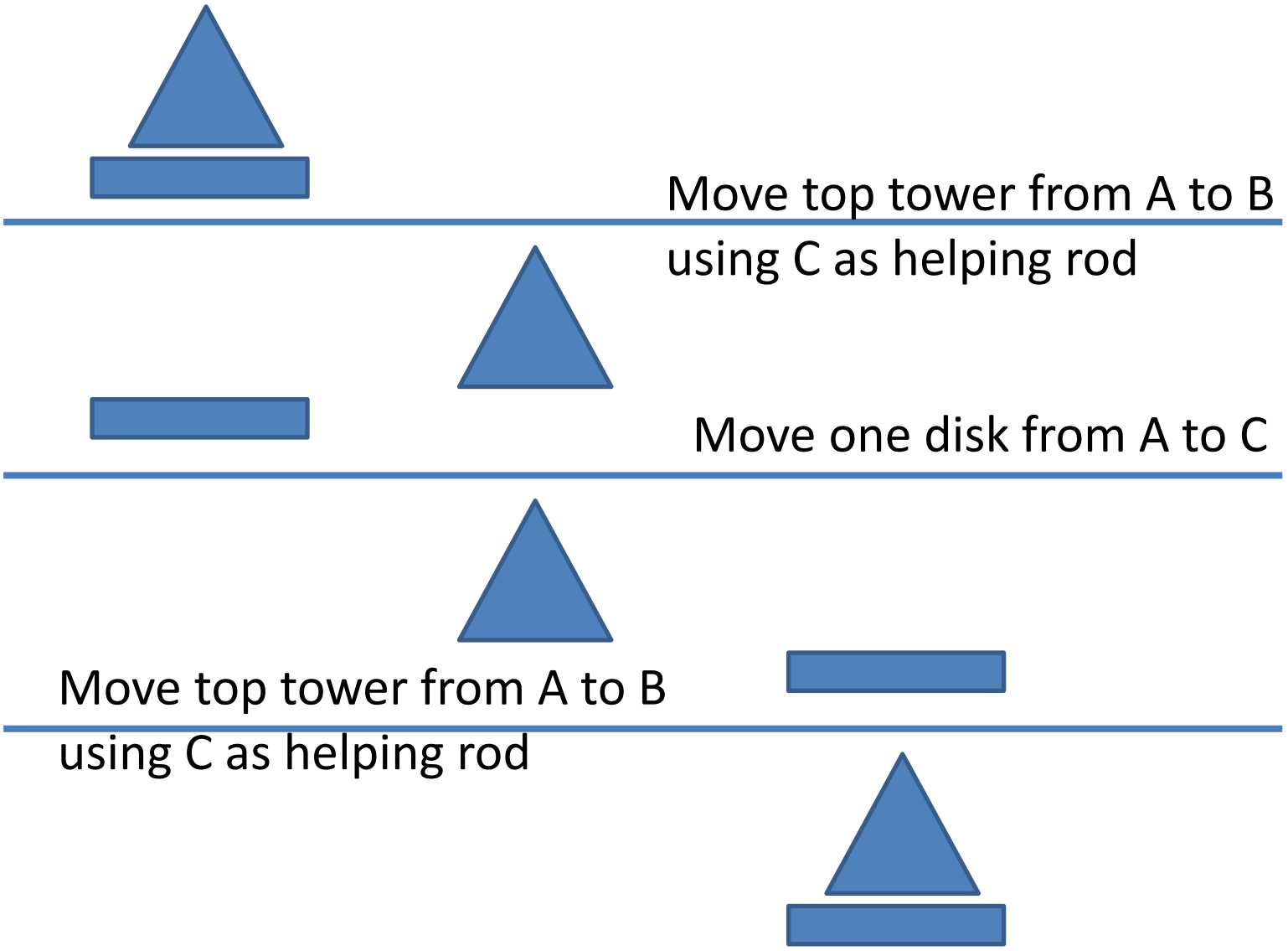


Move top tower from A to B using C as helping rod

Move one disk from A to C

Move top tower from B to C using A as helping rod

Towers of Hanoi: another picture



Towers of Hanoi: Recursive Solution

To move n disks from rod A to rod C, using B as a “helping rod":
If n = 0, there is nothing to do.
Otherwise (namely n > 0):

1) Move n - 1 disks from rod A to rod B, using C as a “helping rod".
2) Move the single disc n directly from rod A to rod C.
3) Move n - 1 disks from rod B to rod C, using A as a “helping rod".

Correctness: (no rules are violated)

- During the entire stage (1), disk n stays put on rod A. As it was the biggest of all n disks, no rule will be violated if some of the n - 1 disks are placed on top of it during the recursion in (1).
- In step (2), all n - 1 smaller disks are on rod B, so moving disc n directly from rod A to rod C is legal.
- The argument for step (3) is identical to the argument for step (1).

Towers of Hanoi: Python Code

- We write a function of four arguments,
  `HanoiTowers(start, via, target, n).`
- The first three arguments are the three rods' "names", such as "A", "B", and "C". The last argument, n, is the number of discs.
- The function prints the moves, and does not return anything.

```
def HanoiTowers(start, via, target, n):
    if n>0:
        HanoiTowers(start, target, via, n-1)
        print("disk", n, "from", start, "to", target)
        HanoiTowers(via, start, target, n-1)
```

- Question: what is the base case here (it appears indirectly in the code).

Towers of Hanoi: Python Code

- We have set n == 0 as the base case of the recursion (in that case, None is returned, and the recursion stops).
- Every positive value of n results in two recursive calls with n-1, and one print (corresponding to the move of the bottom disc at the current recursion level).

```
def HanoiTowers(start, via, target, n):
    """ computes a list of discs steps to move a stack
        of n discs from rod "start" to rod "target"
        employing intermediate rod "via"
    """
    if n>0:
        HanoiTowers(start, target, via, n-1)
        print("disk", n, "from", start, "to", target)
        HanoiTowers(via, start, target, n-1)
```

Towers of Hanoi: Running the Code

```
>>> HanoiTowers("A", "B", "C", 1)
disk 1 from A to C

>>> HanoiTowers("A", "B", "C", 2)
disk 1 from A to B
disk 2 from A to C
disk 1 from B to C

>>> HanoiTowers("A", "B", "C", 3)
disk 1 from A to C
disk 2 from A to B
disk 1 from C to B
disk 3 from A to C
disk 1 from B to A
disk 2 from B to C
disk 1 from A to C
```

Towers of Hanoi: Number of Moves

- Let us denote by $H(n)$ the number of moves required to solve an n disc instance of the puzzle.
- In the recursive solution outlined above, to solve an n discs instance we solve two instances of $n-1$ discs, plus one actual move. This gives us the recursive relation

$$H(0) = 0$$
$$\text{For } n > 0,\ H(n) = 2 \cdot H(n-1) + 1$$

whose solution is $H(n) = 2^n - 1$ (You should be able to verify the last equality, using recursion trees or induction.)

Time Complexity analysis

- We claimed that the number of moves required to solve an instance with n disks is $H(n) = 2^n - 1.$ Our program generates such a list of disk moves. It runs in $O(H(n))$ time = $O(2^n).$

- The recursion depth here is “just" $O(n)$. But the size of the recursion tree is $O(2^n)$, which is exponential in n.

Optimality of Number of Moves

- Hey, wait a minute. $H(n) = 2^n - 1$ is the number of moves in the solution presented above. Can't we find a more efficient solution?

- This is very good thinking in general.
- But in this case, we can argue that $H(n) = 2^n - 1$ moves are required from any solution strategy. (Of course, more inefficient strategies do exist).
- Can you explain why?

Towers of Hanoi Nightmare

Suppose a monster demanded to know what the $3^{97} + 19$ 'th move in an n = 200 disk Towers of Hanoi puzzle is, or else . . . .

Having seen and even understood the material, you realize that either expanding all $H(200) = 2^{200} - 1$ moves, or even just the first $3^{97} + 19$, is out of computational reach in any conceivable future, and the monster should try its luck elsewhere.

You eventually decide to solve this new problem. The first step towards taming the monster is to give the new problem a name:

```
hanoi_move(start, via, target, n, k)
```

Towers of Hanoi Nightmare

To compute the k-th move in the an n disk Tower of Hanoi puzzle, we recall the solution of the Tower of Hanoi puzzle, and think recursively:

The solution to `HanoiTowers(start,via,target,n)` takes $2^n - 1$ steps altogether (so $1 \leq k \leq 2^n - 1$), and consists of three (unequal) parts.

- In the first part, which takes $2^{n-1} - 1$ steps, we move n-1 disks. If $1 \leq k \leq 2^{n-1} - 1$ the move we are looking for is within this part.

- In the second part, which takes exactly one step , we move disk number n. If $k = 2^{n-1}$ this is the move we want.

- In the last part, which again takes $2^{n-1} - 1$ steps, we again move n-1 disks. If $2^{n-1} + 1 \leq k \leq 2^n - 1$ the move is within this part, and is the $k - 2^{n-1}$ 'th move of this part.

Hanoi Monster - Code

```
def hanoi_move(start, via, target, n, k):
    """ finds the k-th move in Hanoi Towers with n disks """
    if n<=0:
        print("zero or fewer disks")
    elif k<=0 or k>=2**n or type(k)!=int:
        print("number of moves is illegal")
    elif k==2**(n-1):
        print("disk", n, "from", start, "to", target)
    elif k < 2**(n-1):
        hanoi_move(start, target, via, n-1, k)
    else:
        hanoi_move(via, start, target, n-1, k-2**(n-1))
```

Note the roles of the rods, as in the HanoiTowers function.

Recursive Monster Code: Executions

We first test it on some small cases, which can be verified by running the HanoiTowers program.

```
>>> hanoi_move("A","B","C" ,1,1)
'disk 1 from A to C'
>>> hanoi_move("A","B","C" ,2,1)
'disk 1 from A to B'
>>> hanoi_move("A","B","C" ,2,2)
'disk 2 from A to C'
>>> hanoi_move("A","B","C" ,3,7)
'disk 1 from A to C'
>>> hanoi_move("A","B","C" ,4,8)
'disk 4 from A to C'
```

Once we are satisfied with this, we solve the monster's question.

```
>>> hanoi_move("A","B","C" ,200,3**97+19)
'disk 2 from B to A'   #saved!
```

Recursive Monster Solution and Binary Search

- The recursive hanoi_move(start,via,target,n,k) makes at most one recursive call.
- The way it “homes" on the right move employs the already familiar paradigm of binary search: it first determines if move number k is exactly the middle move in the n disk problem. If it is, then by the nature of the problem it is easy to exactly determine the move.
- If not, it determines if the move is in the first half of the moves‘ sequence ($k < 2^{n-1}$) or in the second half ($k > 2^{n-1}$), and makes a recursive call with the correct permutation of rods.
- The execution length is linear in n (and not in $2^n - 1$, the length of the sequence of moves).

Binary Search

- We have already seen binary search and realized it is widely applicable (not only when monsters confront you). We can use binary search when we look for an item in a huge space, in cases where that space is structured so we could tell if the item is

  1. right at the middle,
  2. in the top half of the space,
  3. or in the lower half of the space.

- In case (1), we solve the search problem in the current step. In cases (2) and (3), we deal with a search problem in a space of half the size.

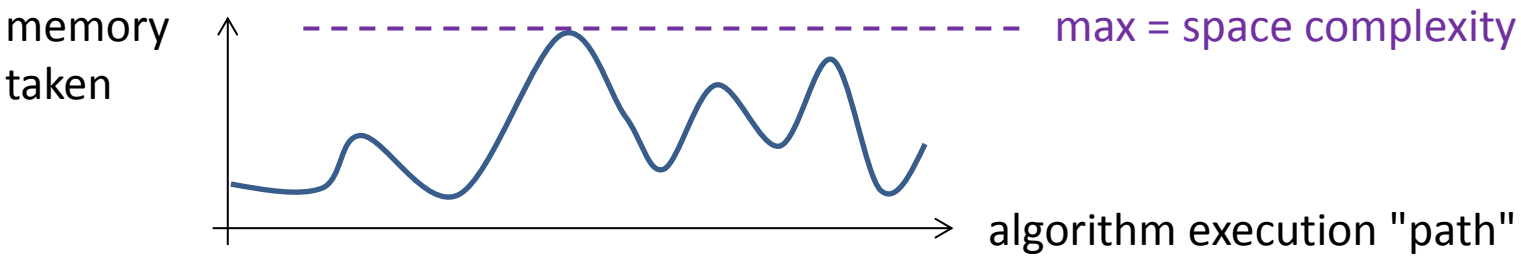
- In general, this process will thus converge in a number of steps which is $\log_2$ of the size of the initial search space. This makes a huge difference. Compare the performance to going linearly over the original space of $2^n - 1$ moves, item by item.

- The binary search idea is also known as lion in the desert idea.

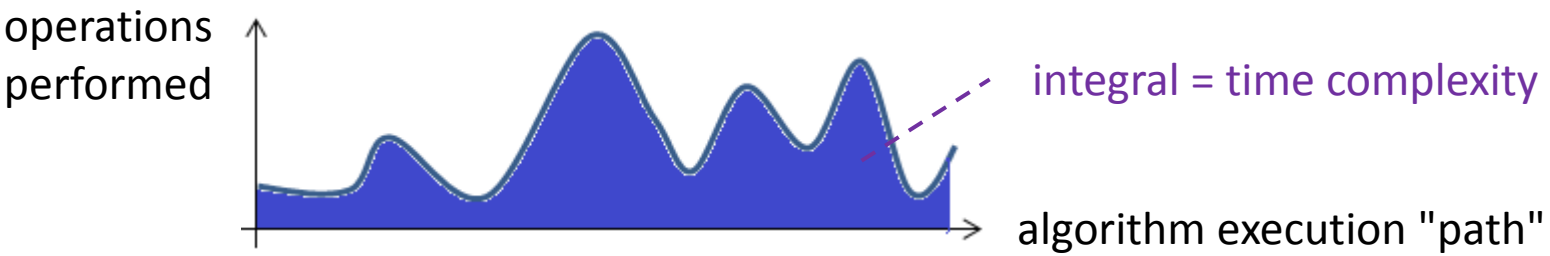
A word about space (memory) complexity

- 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

- Recursion depth has an implication on the space (memory) complexity, as each recursive call required opening a new environment in memory.

- In this course, we will not require space complexity analysis of recursive algorithms

- 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.

Thoughts about Recursion

- Recursion often provides a natural solution (algorithm) to given problems.
- Designing recursive algorithm is not easy
  - Usually follow a recursive definition of the underlying objects
- Are recursive algorithms efficient or costly?
- Can we improve the complexity of recursive algorithms?
  - With/without eliminating the recursion
- More on this next time