

# Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

## Lecture 3

Recursion, Advanced Sorting Algorithms

# Recursive algorithms

The algorithms we saw so far solve their problem *in one go*.

For example:

- to search for an integer  $k$  in array  $A$ , scan  $A$  from left to right until you find  $k$ . Return -1 if not found.

There is another, more structured way.

To search for an integer  $k$  in array  $A$  of length  $n$ :

- check if  $A[0]$  is equal to  $k$ 
  - if they are equal, return 0
  - if not, search for  $k$  in the rest of  $A$ , i.e. in sub-array  $[A[1], \dots, A[n-1]]$

# Recursion

So, the algorithm that we write for searching inside an array **is calling itself** on a smaller array.

This is called **recursion**: the algorithm recurs to itself.

Recursive algorithms typically have two parts:

- the recursive part is the one that makes the recursive calls
- the base part is the one that returns without recursion.

The base part is needed for termination:

if the algorithm just keeps calling itself, how is it going to ever return anything?


# Sub-arrays

We first need a way to obtain sub-arrays of a given array.

In Python and in Java there are built-in methods for this.

- We will be using these freely! (but also implement them – how?)


```
A = [10,5,2,42]
B = A[1:3]
print(B) # prints [5,2]
B = A[1:]
print(B) # prints [5,2,42]
B = A[:3]
print(B) # prints [10,5,2]
```



Note: each B is a **copy** of the specified part of A, i.e. it is a **new** array (this is even clearer in the Java code).

So, for example, we have:

```
A = [10,42]
B = A[1:] # so B=[42]
B[0] = 0 # does not change A[1]
print(A) # prints [10,42]
```



```
int[] a = {10,5,2,42};
int[] b = Arrays.copyOfRange(a,1,3);
System.out.println(Arrays.toString(b));
...
```



# Recursive search

Here is our first attempt to recursive search:

```
def searchRecNope(A, k):  
    if A[0] == k:  
        return 0  
    return searchRecNope(A[1:], k)
```



Question: how many mistakes have we made in this code? (hint: >1)

# Recursive search

Here is our first attempt to recursive search:

```
def searchRecNope(A, k):  
    if A[0] == k:  
        return 0  
    return searchRecNope(A[1:], k)
```



Question: how many mistakes have we made in this code? (hint: >1)

1. It is missing the base case. So, if A does not contain k it will eventually call itself with A == [] and break (at A[0] == k).
2. It only ever returns 0!

So, searchRecNope([1, 2], 2) will return 0 instead of 1

## First solution: base case + auxiliary function

```
def searchRec(A, k):  
    return searchRecAux(A,k,0)  
  
def searchRecAux(A, k, lo):  
    if lo == len(A):  
        return -1  
    if A[lo] == k:  
        return lo  
    return searchRecAux(A,k,lo+1)
```



# First solution: base case + auxiliary function

```
def searchRec(A, k):  
    return searchRecAux(A,k,0)  
  
def searchRecAux(A, k, lo):  
    if lo == len(A):  
        return -1  
    if A[lo] == k:  
        return lo  
    return searchRecAux(A,k,lo+1)
```



The auxiliary method

`searchRecAux(A, k, lo)`

searches for `k` in the part of `A` starting from `lo` on.

To search the whole of `A`, we simply call the auxiliary search method with `lo = 0` i.e. we do `searchRecAux(A,k,0)`.

This solution solves both of the problems of our previous attempt.

Auxiliary functions are standard in recursive algorithms:

- this makes sense, since we typically apply recursion on smaller parts of the original input
- it is a bit clunky, as we write two functions for a simple algorithm like this



## Second solution: base case + plumbing

```
def searchRec2(A, k):  
    if A == []:  
        return -1  
    if A[0] == k:  
        return 0  
    recS = searchRec2(A[1:], k)  
    if recS == -1:  
        return -1  
    return recS + 1
```



## Second solution: base case + plumbing

```
def searchRec2(A, k):  
    if A == []:  
        return -1  
    if A[0] == k:  
        return 0  
    recS = searchRec2(A[1:], k)  
    if recS == -1:  
        return -1  
    return recS + 1
```



This solution is somewhat more elegant than using auxiliary functions but:

- it is dangerous, as we might miss some of the required plumbing
- sometimes auxiliary functions cannot be avoided (see sorting later)
- we are creating a lot of intermediate arrays `A[1:]` (wasteful)

This is closer to the informal description we started from, and does not need an auxiliary method.

But we need to do some *plumbing* so that we return the correct thing:

- if `k` is in position of `A[1:]`, then it is in position `i+1` of `A`.

So, if `searchRec2(A[1:])` returns `i` then we return `i+1`

- extra care is needed when `searchRec2(A[1:])` returns `-1` (we cannot return `i+1`!)

# Another example: factorial

Recall factorial numbers:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

Factorials can be defined recursively (using maths):

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

e.g.

$$5! = 120$$

$$10! = 3628800$$

$$0! = 1$$

We can program them in exactly the same way!

```
def factRec(n):  
    if n == 0:  
        return 1  
    return n*factRec(n-1)
```



# Other implementations of factorial

Here are two more implementations of factorial.

Could you comment on what are their pros and cons?

```
def factRecTail(n):  
    return factRecAux(n,1)  
  
def factRecAux(n,acc):  
    if n == 0:  
        return acc  
    return factRecAux(n-1,n*acc)
```



```
def fact(n):  
    acc = 1  
    for i in range(1,n+1):  
        acc *= i  
    return acc
```



`factRecAux` uses a special kind of recursion: the recursion is made at the very last step of the function before returning.

This is called **tail recursion** and is typically implemented faster in most programming languages (but not in Python).

# Sorting algorithms

*Every algorithms course continues with sorting.*

*– Anonymous*

SORTING is the following problem:

- given an array  $A$  of integers
- put the elements of  $A$  in increasing order

Notes:

- put in order means: re-arrange the elements of  $A$  so that they are in increasing order
- the array  $A$  can be arbitrarily large
- in general, SORTING is about arrays of elements of any type (not necessarily integers), so long as they can be ordered

## Sorting in $O(n^2)$ and $O(n \log n)$

So far we have looked at: insertion sort, selection sort, bubblesort (?).

They all are  $O(n^2)$ , i.e. run in  $\Theta(n^2)$  in the worst case.

In this lecture we will look at two algorithms that are faster:

- mergesort, which is  $O(n \log n)$  i.e.  $\Theta(n \log n)$  in the worst case
- quicksort, which is  $O(n^2)$  but typically runs in  $\Theta(n \log n)$

Both of them are recursive algorithms.

# Merge sort

Merge sort is a divide-and-conquer algorithm that:

1. divides the array  $A$  in two halves,
2. sorts the two halves,
3. merges the two halves together.

It is an example of a **divide and conquer** algorithm:

to solve the problem for an array  $A$ , we first solve it for two smaller arrays and then combine those solutions to get a solution for  $A$

# Merging in merge sort

The crux of the algorithm lies in the final step, i.e. **merging**.

For example, take:

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

Dividing  $A$  into two halves (initial and final part) we get:

$\text{half1} = [30, 25, 67, 99, 8]$

$\text{half2} = [16, 28, 63, 12, 20]$

Suppose these get sorted, by ~~magic~~ applying merge sort on them, so we have:

$\text{half1} = [8, 25, 30, 67, 99]$

$\text{half2} = [12, 16, 20, 28, 63]$

To merge the two halves we simply **interleave them in order** and put them back in  $A$ .



# Merging = interleaving in order

So, we had:

`half1 = [8, 25, 30, 67, 99]`

`half2 = [12, 16, 20, 28, 63]`

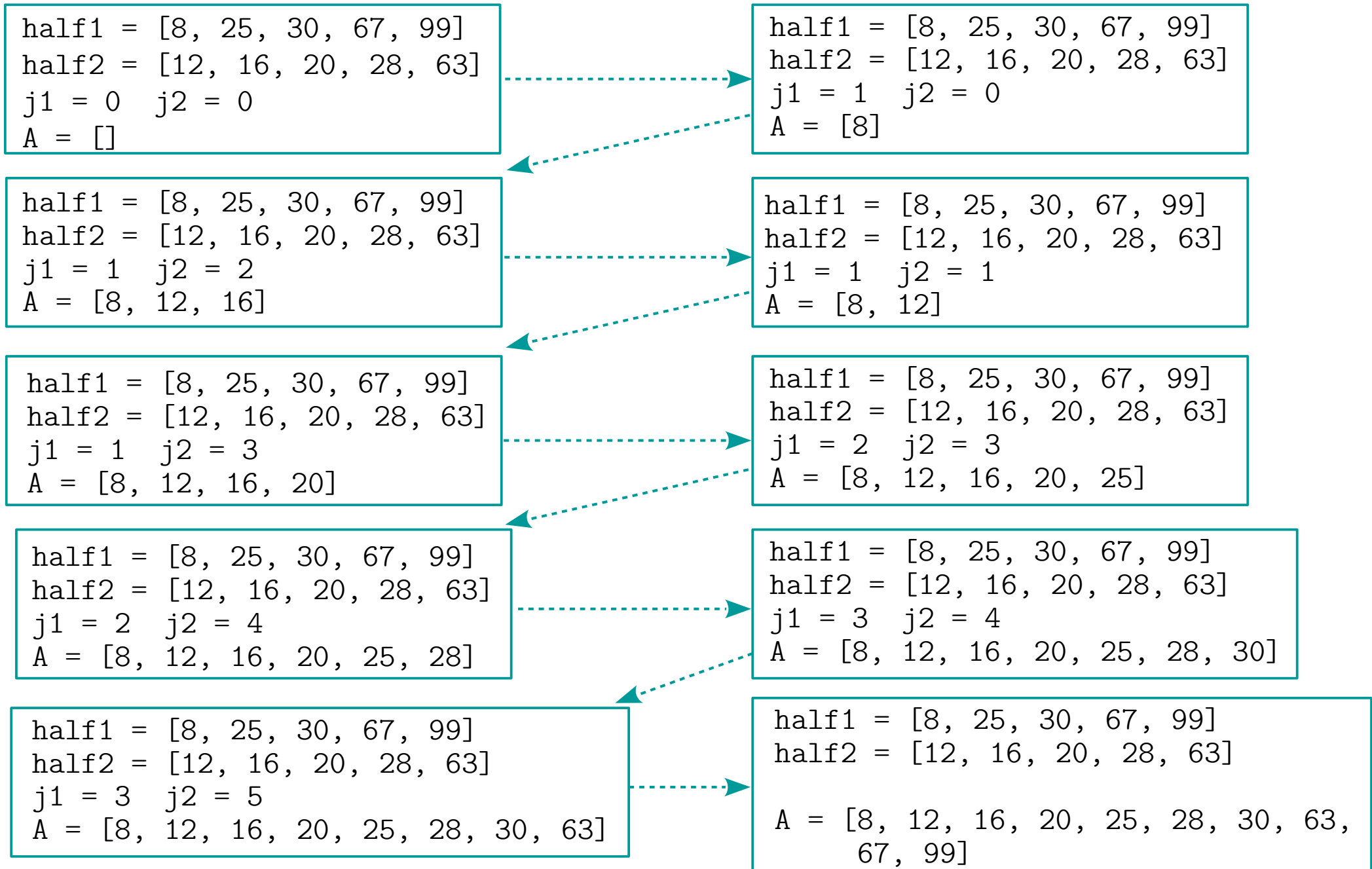
We go through the elements of `half1` and `half2` using counters `j1, j2`:

- If `half1[j1] < half2[j2]` : then we put `half1[j1]` in the next position of `A` and increase `j1` by 1.
- Otherwise : put `half2[j2]` in the next position of `A` and increase `j2` by 1.

We continue this way until we put all the elements of `half1` or `half2` in `A` (i.e. until `j1` reaches the end of `half1`, or `j2` reaches the end of `half2`).

We finally put all the remaining elements of the other half in `A` as well.

# In detail



# Merge sort formally

```
def mergeSort(A):
    if len(A) <= 1:
        return
    mid = len(A)//2
    half1 = A[:mid]
    half2 = A[mid:]
    mergeSort(half1)
    mergeSort(half2)
    merge(half1, half2, A)

def merge(h1, h2, A):
    i=0; j1=0; j2=0
    while j1<len(h1) and j2<len(h2):
        if h1[j1] < h2[j2]:
            A[i] = h1[j1]
            j1 += 1; i += 1
        else:
            A[i] = h2[j2]
            j2 += 1; i += 1
    while j1 < len(h1):
        A[i] = h1[j1]
        j1 += 1; i += 1
    while j2 < len(h2):
        A[i] = h2[j2]
        j2 += 1; i += 1
```



`mergeSort(A)` returns straight away if `A` has length at most 1 (it is already sorted). It then has 4 steps:

- it first divides `A` in two halves: `half1` and `half2`
- it calls itself on these two half arrays (thus sorting them)
- it then merges the two halves into `A`.

`merge(h1, h2, A)` merges as follows:

- it uses 3 indices: `i` ranges over positions in `A`, `j1` is for `h1`, and `j2` for `h2`
- it first goes through `h1` and `h2`, each time copying into `A` the next smallest element, until `j1` reaches the end of `h1`, or `j2` reaches the end of `h2`
- it then goes on and copies into `A` all the remaining elements of `h1` or `h2`

# The magic part: recursion

Recall the recipe for merge sort:

1. divide the array  $A$  in two halves,
2. sort the two halves,
3. merge the two halves together.

Let us look closely at step 2: how are the two halves sorted?

They are sorted by **calling merge sort again** on them, i.e. by using recursion.

Merge sort on  $A$  calls merge sort on each of the two halves of  $A$ .

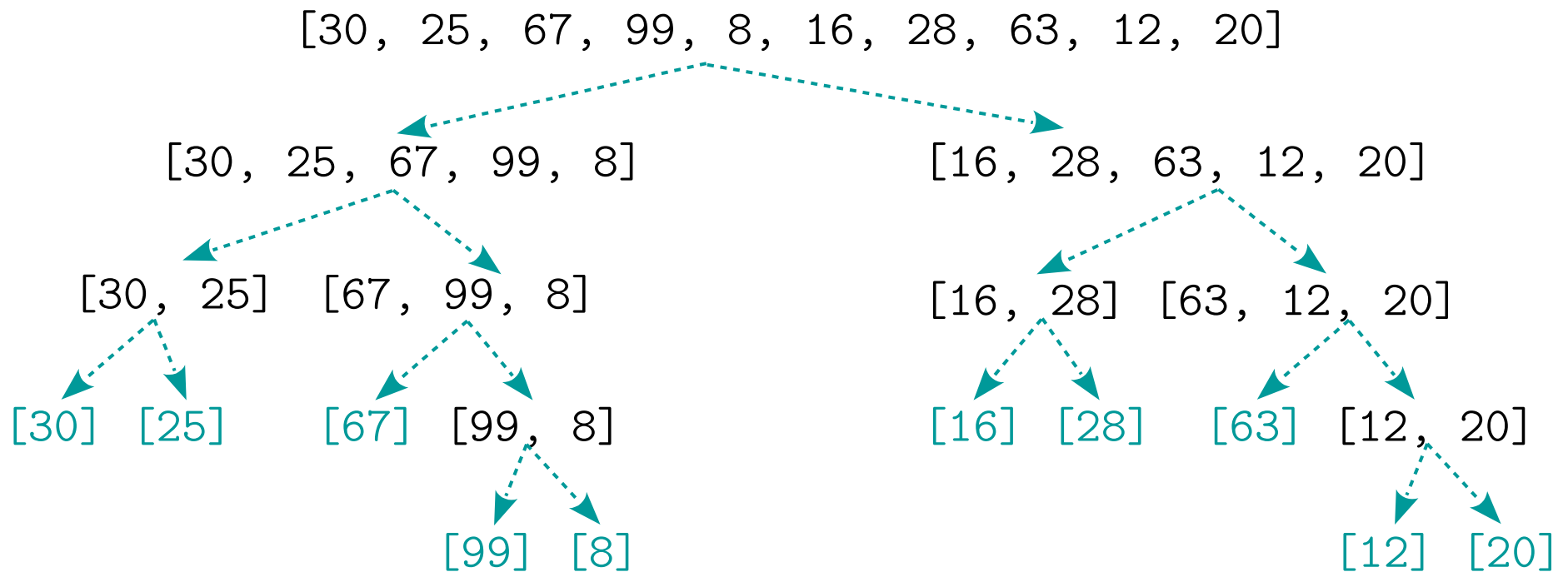
*-- How does this ever terminate?*

If we keep calling merge sort on smaller arrays, we are bound to reach a point where the array we want to sort has length 1.

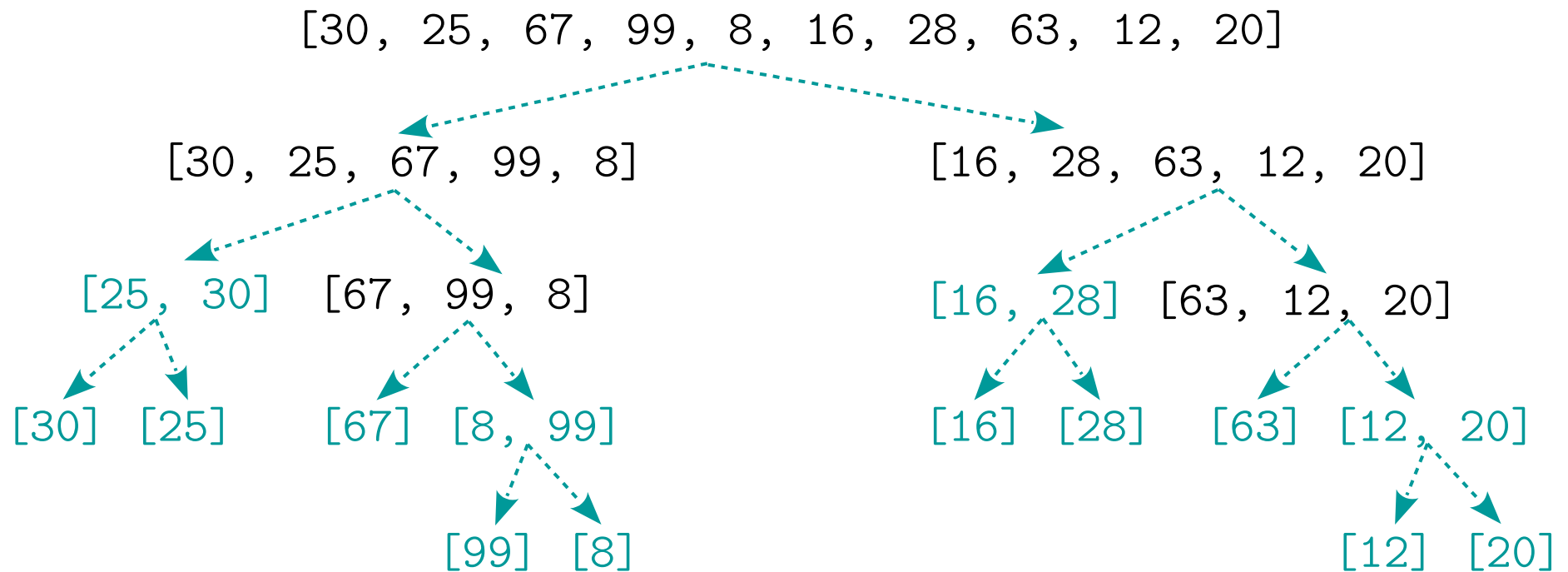
At that point, the base part kicks in and we return.

```
if len(A) <= 1:  
    return
```

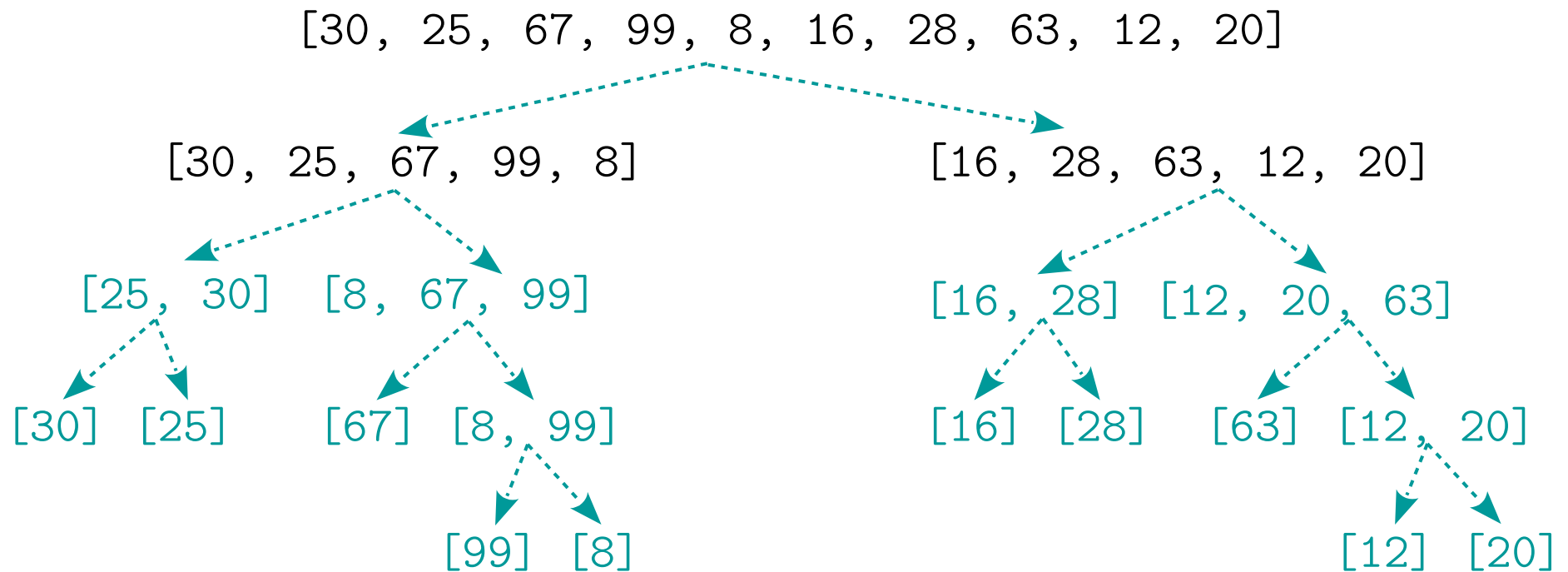
# Recursive calls in example



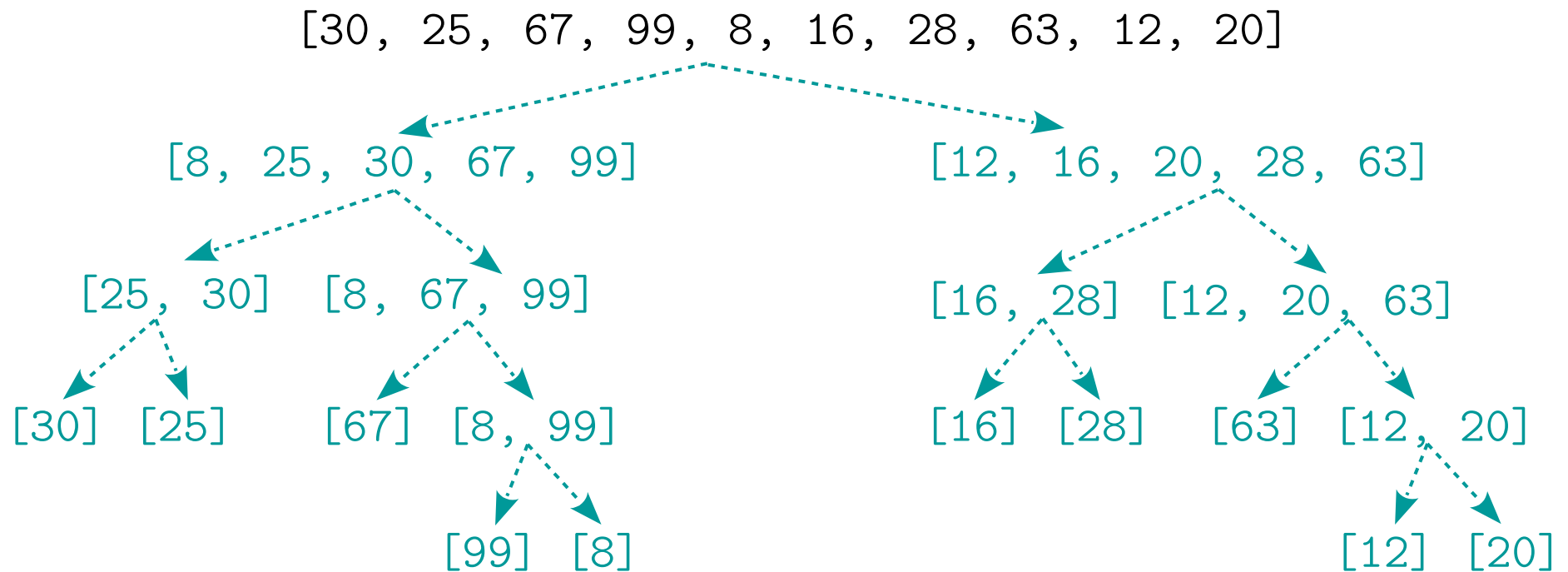
# Recursive calls in example



# Recursive calls in example

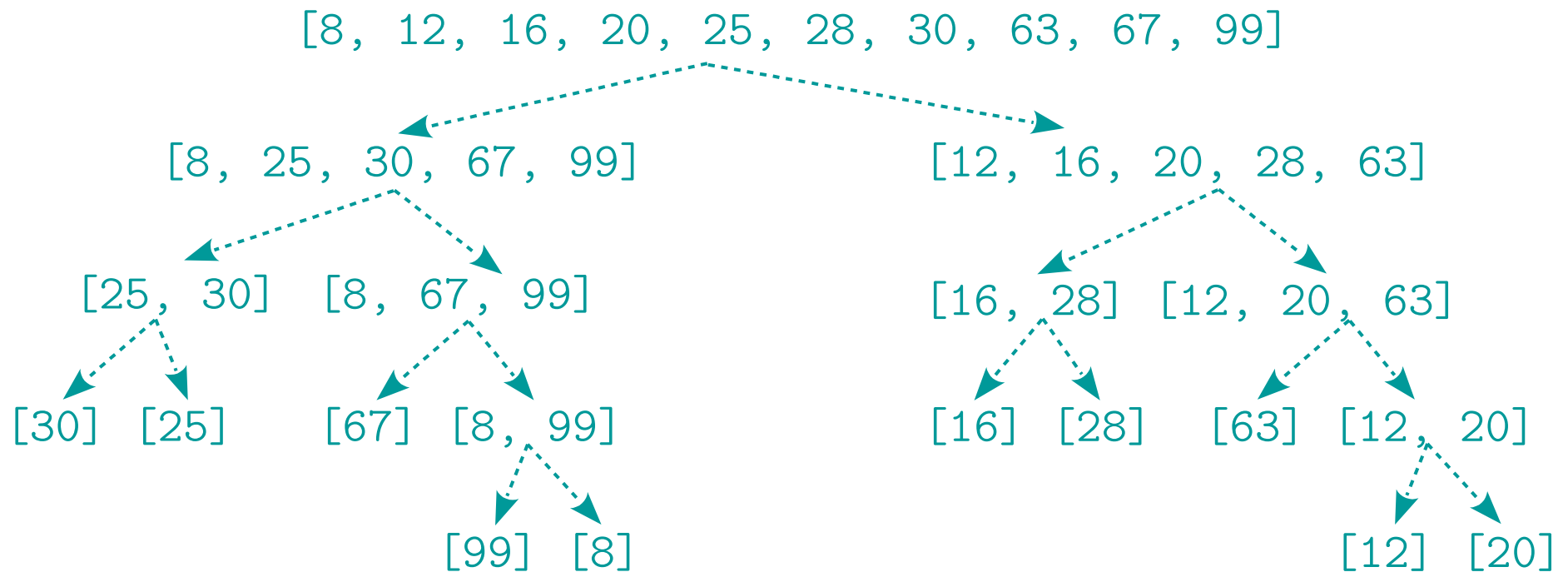


# Recursive calls in example





# Recursive calls in example



# Time complexity analysis

Suppose array  $A$  has  $n$  elements. We count the number of comparisons made (between array elements) by merge sort when applied on  $A$ .

Comparisons happen in the merge phase:

- to merge two ordered halves of size  $n/2$  each, we need to make at most  $n-1$  comparisons
- each of the 2 halves was sorted by merging two quarters of size  $n/4$  each, which required at most  $n/2 - 1$  comparisons each,
- and so on ...

We can divide the array  $A$  at most  $\log n$  times.

So, max total comparisons (approximately):

$$\underbrace{n + 2(n/2) + 4(n/4) + \dots}_{\log n \text{-many times } n} = n(\log n)$$

# Merge sort verdict

So, we have a worst case time complexity of  $\Theta(n \log n)$ .

This means merge sort is  $O(n \log n)$ .

This is better than the  $O(n^2)$  of insertion sort, et al!

In general, it can be shown that  $O(n \log n)$  is the best we can hope for in sorting algorithms. In that sense, merge sort is optimal.

However, merge sort has some practical drawbacks:

- recursion is heavy computationally: too many function calls, each of them requiring its own computational environment (i.e. call stack)
- starting from an array  $A$  of length 10, we ended up constructing 18 smaller arrays

# Java: mergeSort

```
public static void mergeSort(int[] a) {
    if (a.length>1) {
        int i, mid = a.length/2;
        int[] half1 = new int[mid];
        int[] half2 = new int[a.length-mid];
        for (i=0; i<mid; i++) half1[i] = a[i];
        for (; i<a.length; i++) half2[i-mid] = a[i];

        mergeSort(half1);
        mergeSort(half2);

        int j=0, k=0;
        for (i=0; j<half1.length && k<half2.length; i++)
            if (half1[j] < half2[k]) { a[i] = half1[j]; j++; }
            else { a[i]=half2[k]; k++; }

        for (; j<half1.length; i++, j++) a[i] = half1[j];
        for (; k<half2.length; i++, k++) a[i] = half2[k];
    }
}
```



# Quicksort

Quicksort is a divide-and-conquer sorting algorithm that typically performs very fast. It is based on the following idea.

To sort an array  $A$ :

- we **pick an element** of  $A$  that we call the *pivot*
- we **partition** (i.e. re-arrange)  $A$  so that:
  - all its elements that are less than *pivot* are in the start
  - all its elements that are greater than or equal to *pivot* are in the end
  - we put the *pivot* in between the start and the end parts
- we **recursively sort** the beginning and the end parts of  $A$

## Taking first element as pivot

Quicksort is a divide-and-conquer sorting algorithm that typically performs very fast.

To sort an array  $A$  with more than 1 elements:

- we pick  $A[0]$  and call it the *pivot*
- we partition (i.e. re-arrange)  $A$  so that:
  - all its elements that are less than *pivot* are in the start
  - all its elements that are greater than or equal to *pivot* are in the end
  - we put the *pivot* in between the start and the end parts
- we recursively sort the beginning and the end parts of  $A$

# Quicksort example

A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]  
pivot = 30

A = [25, 8, 16, 28, 12, 20, 30, 63, 99, 67]

So, after partition, we are left with two smaller parts to sort:

- [25, 8, 16, 28, 12, 20] (i.e. A[0:6])
- [63, 99, 67] (i.e. A[7:10])

which we do by applying the same procedure on them (i.e. by recursion):

- we select a pivot element for each of them and partition them,
- and so on ...

# Partitioning of A

The easiest way to partition the elements of A with respect to `pivot` is by using an auxiliary array B, initially filled with 0's:

```
A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]  
pivot = 30
```

```
B = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

We go through all the elements of A [1 : 10]: if they are smaller than `pivot` then we put them at the beginning of B, otherwise at its end:

```
A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]  
pivot = 30
```

```
B = [25, 8, 16, 28, 12, 20, 0, 63, 99, 67]
```

We finally put `pivot` in the unfilled position of B and copy B back to A:

```
B = [25, 8, 16, 28, 12, 20, 30, 63, 99, 67]
```

```
A = [25, 8, 16, 28, 12, 20, 30, 63, 99, 67]
```



## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 1, loB = 0, hiB = 9$

Now,  $A[i] = 25 < pivot$  so we copy 25 in  $B[loB]$  and increase  $loB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [25, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 2, loB = 1, hiB = 9$

Now,  $A[i] = 67 \geq pivot$  so we copy 67 in  $B[hiB]$  and decrease  $hiB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [25, 0, 0, 0, 0, 0, 0, 0, 0, 67]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 3, loB = 1, hiB = 8$

Now,  $A[i] = 99 \geq pivot$  so we copy 99 in  $B[hiB]$  and decrease  $hiB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [25, 0, 0, 0, 0, 0, 0, 0, 99, 67]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 4, loB = 1, hiB = 7$

Now,  $A[i] = 8 < pivot$  so we copy 8 in  $B[loB]$  and increase  $loB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [25, 8, 0, 0, 0, 0, 0, 0, 99, 67]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 5, loB = 2, hiB = 7$

Now,  $A[i] = 16 < pivot$  so we copy 16 in  $B[loB]$  and increase  $loB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [25, 8, 16, 0, 0, 0, 0, 0, 99, 67]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 6, loB = 3, hiB = 7$

Now,  $A[i] = 28 < pivot$  so we copy 28 in  $B[loB]$  and increase  $loB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$

$i \uparrow$

$B = [25, 8, 16, 28, 0, 0, 0, 0, 99, 67]$

$loB \uparrow$

$hiB \uparrow$

$pivot = 30, i = 7, loB = 4, hiB = 7$

Now,  $A[i] = 63 \geq pivot$  so we copy 63 in  $B[hiB]$  and decrease  $hiB$  by 1. We also increase  $i$  by 1.

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 12, 28, 63, 12, 20]$   
 $i \uparrow$

$B = [25, 8, 16, 28, 0, 0, 0, 63, 99, 67]$   
 $loB \uparrow$        $hiB \uparrow$

$pivot = 30, i = 8, loB = 4, hiB = 6$


Now,  $A[i] = 12 < pivot$  so we copy 12 in  $B[loB]$  and increase  $loB$  by 1. We also increase  $i$  by 1.

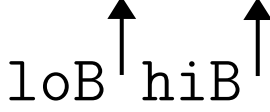


## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 12, 28, 63, 12, 20]$   


$B = [25, 8, 16, 28, 12, 0, 0, 63, 99, 67]$   


$\text{pivot} = 30, i = 8, loB = 5, hiB = 6$

Now,  $A[i] = 20 < \text{pivot}$  so we copy 20 in  $B[loB]$  and increase  $loB$  by 1. We also increase  $i$  by 1, and we are done with reading from  $A$ .

## Partitioning of A in B

How can the partitioning be implemented? We use three pointers:

- $i$  to go through the elements of  $A$  after  $A[0]$
- $loB$  for the beginning of  $B$  that we have filled
- $hiB$  for the end of  $B$  that we have filled

$A = [30, 25, 67, 99, 8, 16, 28, 63, 12, 20]$   
 $i \uparrow$

$B = [25, 8, 16, 28, 12, 20, 0, 63, 99, 67]$   
 $loB \uparrow \uparrow hiB$

$pivot = 30, i = 8, loB = 5, hiB = 6$

We now write  $pivot$  in  $B[loB]$  and the partitioning is complete:

$B = [25, 8, 16, 28, 12, 20, 30, 63, 99, 67]$

# Implementation

```
def quickSort(A):  
    quickSortRec(A, 0, len(A))  
  
def quickSortRec(A, lo, hi):  
    # sorts A[lo:hi]  
    if hi-lo <= 1:  
        return  
    iPivot = partition(A, lo, hi)  
    quickSortRec(A, lo, iPivot)  
    quickSortRec(A, iPivot+1, hi)  
  
def partition(A, lo, hi):  
    # partitions A[lo:hi]  
    # (to be defined)
```

We need two auxiliary functions:

- `quickSortRec(A, lo, hi)` will sort the part of A between lo and hi-1.

We need this because we want to recursively sort smaller parts of A.

- `partition(A, lo, hi)` partitions A[lo:hi] using the algorithm we saw before. It uses A[0] as a pivot, and returns its position in A once A[lo:hi] has been partitioned.

# Implementation of partition

```
def partition(A, lo, hi):
    pivot = A[lo]
    B = [0 for i in range(lo,hi)]
    loB = 0
    hiB = len(B)-1
    for i in range(lo+1,hi):
        if A[i] < pivot:
            B[loB] = A[i]
            loB += 1
        else:
            B[hiB] = A[i]
            hiB -= 1
    B[loB] = pivot
    for i in range(lo,hi):
        A[i] = B[i-lo]
    return lo+loB
```

To partition  $A[lo:hi]$  (i.e.  $A[lo] \dots A[hi-1]$ ) we follow these steps:

- we create a new array B that is initially set to  $[0, \dots, 0]$ , and in which we will store the partitioned copy of  $A[lo:hi]$
- we set the pivot to be  $A[lo]$
- we set loB, hiB to point to the beginning and the end of B respectively
- we loop through the elements of  $A[lo+1:hi]$  and for each such  $A[i]$ :
  - if  $A[i] < \text{pivot}$  then we set  $B[loB]$  to  $A[i]$  and increase loB by 1
  - otherwise we set  $B[hiB]$  to  $A[i]$  and decrease hiB by 1
- we set  $B[loB]$  to pivot (note that loB is now equal to hiB as the two pointers must have “met”)
- we copy the values of B back to A

We then return the new position of pivot in A

# Example in full

[30, 25, 67, 99, 8, 16, 28, 63, 12, 20]



[25, 8, 16, 28, 12, 20, 30, 63, 99, 67]



[8, 16, 12, 20, 25, 28, 30, 63, 99, 67]



[8, 16, 12, 20, 25, 28, 30, 63, 67, 99]



[8, 12, 16, 20, 25, 28, 30, 63, 67, 99]

# Time complexity with perfect pivots

Suppose array  $A$  has  $n$  elements. To find the running time, we will count the number of comparisons made by quicksort when applied on  $A$ .

Comparisons happen in the partition phase:

- To partition the whole of  $A$  we need  $n-1$  comparisons (because every other element is compared with  $A[0]$ )
- If partition gives us two sub-arrays of equal length, i.e.  $(n-1)/2$  each, to partition each of them we need  $(n-1)/2 - 1$  comparisons
- and so on ...

We can divide the array  $A$  at most  $\log n$  times. So, total comparisons (approximately):

$$\underbrace{n + 2(n/2) + 4(n/4) + \dots}_{\log n \text{ many times } n} = n(\log n)$$

# Worst case time complexity

Our previous analysis was based on the assumption that each partition splits the array in two sub-arrays of equal size.

But this is not necessarily the case!

Partitions depend on the choice of pivot and there is no guarantee that the pivot will be the median (e.g. see our previous example) ...

In the worst case, the pivot could be the least element of the array:

- To partition the whole of  $A$  we need  $n-1$  comparisons.  
If the pivot is the least element, the partition gives us one sub-array of length 0 and one of length  $n-1$ .
- To partition the sub-array of length  $n-1$  we need at most  $n-2$ -comparisons.  
If pivot is least element, we are left with a sub-array of length  $n-3$ .
- and so on ...

So, total comparisons:  $n-1 + n-2 + n-3 + \dots + 1 = n(n-1)/2$  (i.e.  $\Theta(n^2)$ )

# Quicksort verdict

Quicksort **could have** a worst case time complexity of  $\Theta(n \log n)$  **if we had a guarantee** that pivots are selected well.

But there is no such guarantee, so it is  $\Theta(n^2)$  in the worst case.

So, quicksort is  $O(n^2)$ .

In practice, though, quicksort is a fast algorithm:

- different tricks are used so that the selection of pivot is random, and therefore has a good chance of not being a bad choice
- in contrast to merge sort, the creation of a new array is only temporary ( $B$  is discarded when partition is finished), and therefore the total memory used is much less (in fact, the creation of  $B$  can be sidestepped altogether)



# From sorting integers to sorting objects

We may want to sort other things than integers.

For example, here is a simple class of student scripts, where each script has a variable `id` for the student's ID, and `mark` for the student's mark:

```
class Script:
    def __init__(self, sid, mark):
        self.sid = sid
        self.mark = mark
```

We have an array `A` of `Script`'s and want to sort it, say by student ID.

How can we do it using the algorithms we have seen?

# Sorting Scripts (ad-hoc solution)

```
class Script:
    def __init__(self, sid, mark):
        self.id = sid
        self.mark = mark
```

One solution is to pick our favourite sorting algorithm and adapt it to Script. Here, we take quicksort:

```
def quickSort(A):
    quickSortRec(A,0,len(A))

def quickSortRec(A, lo, hi):
    # sorts A[lo:hi]
    if hi-lo <= 1:
        return
    iPivot = partition(A,lo,hi)
    quickSortRec(A,lo,iPivot)
    quickSortRec(A,iPivot+1,hi)
```

```
def partition(A, lo, hi):
    pivot = A[lo]
    B = [None for i in range(lo,hi)]
    loB = 0
    hiB = len(B)-1
    for i in range(lo+1,hi):
        if A[i].sid < pivot.sid:
            B[loB] = A[i]
            loB += 1
        else:
            B[hiB] = A[i]
            hiB -= 1
    B[loB] = pivot
    for i in range(lo,hi):
        A[i] = B[i-lo]
    return lo+loB
```

# Sorting Scripts (OO solution)

A more principled solution is to add in the `Script` class a comparison function for objects of that class:

```
class Script:
    def __init__(self, sid, mark):
        self.sid = sid
        self.mark = mark

    def __lt__(self, other):
        return self.sid < other.sid
```

We can now write our sorting algorithm as usual as long as the only comparisons it uses are by `<`. For each such comparison, Python will automatically use the `lt` function instead.

We may still need to fix array initialisations and other details that are integer-specific inside our sorting algorithm.

In Java there is a similar solution, whereby we make our class implement the `Comparable` interface and in particular the `compareTo` method.

# Exercises

1. Write a Python function

```
def subArray(A, lo, hi)
```

that returns a copy of A containing the elements  $A[lo], A[lo+1], \dots, A[hi-1]$ .

2. Using recursion, write Python functions

```
def searchLast(A, k)
```

```
def findMax(A)
```

that return the index of the last occurrence of integer k in A (or -1), and the biggest element in integer array A respectively.

3. If we call merge sort on an array of size 64, how many smaller arrays do we construct in total?

4. Apply quicksort step-by-step to

```
A = [8, 12, 16, 20, 25, 28, 30, 63, 67, 99]
```

and thus verify that, when quicksort is called on an array that is already sorted, it performs as in the worst case analysis.

5. Recall the class we defined for scripts:

```
class Script:
```

```
    def __init__(self, id, mark):
```

```
        self.id = id
```

```
        self.mark = mark
```

Write a Python function

```
def sortScriptsByID(A)
```

that takes an array A of Script's and sorts it first with respect to mark and then by student ID.

6. Write a Python function

```
def sortScriptsByID2(A, sids)
```

that takes an array A of Script's and sorts it with respect to student ID.

The function also takes an array sids which contains all the students' IDs in sorted form. We assume that student ID's are unique. Can you write an  $O(n \log n)$  solution without using merge sort?

7. Suppose that the student IDs are consecutive: 1, 2, 3, ... . Can you write a sorting function for arrays of Script's that sorts with respect to student ID and is  $O(n)$ ?

8. (*harder*) Suppose that the marks are from 0 to 100. Can you write a sorting function for arrays of Script's that sorts wrt mark and is  $O(n)$ ?

9. (*harder*) Implement a version of insertion sort that uses binary search in order to find the position of each element in its insertion routine.

Is your function  $O(n \log n)$ ?

# Exercises reference code

```
def quickSort(A):
    quickSortRec(A,0,len(A))

def quickSortRec(A, lo, hi):
    # sorts A[lo:hi]
    if hi-lo <= 1:
        return
    iPivot = partition(A,lo,hi)
    quickSortRec(A,lo,iPivot)
    quickSortRec(A,iPivot+1,hi)

def partition(A, lo, hi):
    pivot = A[lo]
    B = [0 for i in range(lo,hi)]
    loB = 0
    hiB = len(B)-1
    for i in range(lo+1,hi):
        if A[i] < pivot:
            B[loB] = A[i]
            loB += 1
        else:
            B[hiB] = A[i]
            hiB -= 1
    B[loB] = pivot
    for i in range(lo,hi):
        A[i] = B[i-lo]
    return lo+loB
```

```
def mergeSort(A):
    if len(A) <= 1:
        return
    mid = len(A)//2
    half1 = A[:mid]
    half2 = A[mid:]
    mergeSort(half1)
    mergeSort(half2)
    merge(half1,half2,A)

def merge(h1, h2, A):
    i=0; j1=0; j2=0
    while j1<len(h1) and j2<len(h2):
        if h1[j1] < h2[j2]:
            A[i] = h1[j1]
            j1 += 1; i += 1
        else:
            A[i] = h2[j2]
            j2 += 1; i += 1
    while j1 < len(h1):
        A[i] = h1[j1]
        j1 += 1; i += 1
    while j2 < len(h2):
        A[i] = h2[j2]
        j2 += 1; i += 1
```

# Exercises – Solutions

1. Here is a concise implementation:

```
def subArray(A, lo, hi):  
    return [A[i] for i in range(lo, hi)]
```

2. For `searchLast` we give two solutions, for `findMax` just one:

```
def searchLast(A, k):  
    return searchLastAux(A, k, len(A))  
  
def searchLastAux(A, k, hi):  
    if hi == 0:  
        return -1  
    if A[hi-1] == k:  
        return hi-1  
    return searchLastAux(A, k, hi-1)  
  
def searchLast2(A, k):  
    if A == []:  
        return -1  
    if A[len(A)-1] == k:  
        return len(A)-1  
    return searchLast2(A[:len(A)-1], k)  
  
def findMax(A):  
    if A == []:  
        return None  
    recM = findMax(A[1:])  
    if recM == None or A[0] > recM:  
        return A[0]  
    return recM
```

3. We end up creating smaller arrays.

More generally, if we call merge sort on A of size  $n = 2^k$ , we create:

- 2 arrays that are halves of A
- 4 arrays that are quarters of A
- 8 arrays that are eighths of A
- and so on
- $2^k$  arrays that are 1-element bits of A

which sum up to:

$$2 + 4 + 8 + \dots + 2^k = 2^{k+1} - 2 = 2n - 2$$

5. We use a variant of the first method we saw:

```
def sortScriptsByID(A)  
    quicksortRec(A, lo, hi)  
  
# quicksortRec as before  
# partition as before, apart from:  
#     if (x.mark < y.mark or  
#         (x.mark == y.mark and x.sid < y.sid))  
#         B[loB] = A[i]  
#         loB += 1  
  
Alternatively, we can define the method __lt__ in  
Script as:  
  
def __lt__(self, other):  
    return (x.mark < y.mark or  
            (x.mark == y.mark and x.sid < y.sid))
```