CSC 148 Winter 2017

Week 10

Efficiency considerations

Bogdan Simion

bogdan@cs.toronto.edu

http://www.cs.toronto.edu/~bogdan

University of Toronto, Department of Computer Science

Bad programmers worry about the code.
Good programmers worry about data structures and their relationships.
-- Linus Torvalds

Implicitly, efficiency of algorithms as well!

Outline – this week

- Recursion efficiency
- Searching
- Height analysis
- Sorting
- Big-Oh on paper

Redundancy

- Remember recursion:
  - Calculating Fibonacci numbers
    - if n < 2, fib(n) = 1
    - fib(n) = fib(n-1) + fib(n-2)
- Write a recursive program for this..

```
def fib(n):
    """
     Returns the n-th fibonacci number.
     @param int n: a non-negative number
     @rtype: int
    """
    pass
```

Redundancy

- Remember recursion:
  - Calculating Fibonacci numbers
    - if n < 2, f(n) = 1
    - f(n) = f(n-1) + f(n-2)
- Write a recursive program for this..

```python
def fib(n):
    """
    Returns the n-th fibonacci number.
    @param int n: a non-negative number
    @rtype: int
    """
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Redundancy

- Unnecessary repeated calculations => inefficient!
- Let's expand the recursion: fib(n) = fib(n-1) + fib(n-2)

fib(30)

fib(28) + fib(29)

fib(26) + fib(27) + fib(27) + fib (28)

fib(24) + fib(25) + fib(25) + fib (26) + fib(25) + fib(26) + fib(26) + fib (27)

How could we avoid calculating items we already calculated?

Solution? Memoize

- Keep track of already calculated values

```
def fib_memo(n, seen):
    """
    Returns the n-th fibonacci number, reasonably quickly, without redundancy.
    @param int n: a non-negative number
    @param dict[int, int] seen: already-seen results
    @rtype: int
    """
    if n not in seen:
        seen[n] = (n if n < 2
                   else fib_memo(n-2, seen) + fib_memo(n-1, seen) )
    return seen[n]
```

Running out of stack space

- Some programming languages have better support for recursion than others; python may run out of space on its stack for recursive function calls ...

- For example, recursively traversing a **very** long list ...

Recursive vs iterative

- Any recursive function can be written iteratively
  - May need to use a stack too, potentially
- Recursive functions are not more efficient than the iterative equivalent
  - Could be the same, with compiler support..
- Why ever use recursion then?
  - If the nature of the problem is recursive, writing it iteratively can be

    a) more time consuming, and/or

    b) less readable

Recursive functions are not more efficient than their iterative equivalent
But .. Recursion is a powerful technique for naturally recursive problems

Efficiency considerations:

Search speed

contains in a list

- Suppose v refers to a number: How efficient is the following statement in its use of time?

```
v in [97, 36, 48, 73, 156, 947, 56, 236]
```

- Roughly how much longer would the statement take if the list were
  - 10 times longer?
  - 1,000 times longer?
  - 1,000,000,000 times longer?
- Does it matter whether we used a built-in Python list or our implementation of LinkedList?

Speed of search

- With either a Python list or a linked list of n elements
  - We must look at elements one by one
  - Each time we eliminate only one element from consideration
  - In the worst case, we look at all n elements
- Either way, it takes time proportional to n

- Can we make search in a Python list faster?

add order ...

- Suppose we know the list is sorted in ascending order
    - What strategy would you use to get to the value (if it exists) in a lot less steps than linear search?

- How does the running time scale up as we make the list 2, 4, 8, 16, 32, times longer?

lg (n)

- Key insight: the number of times I repeatedly divide *n* in half, before we are down to 1 element, is the same as the number of times I double 1 before I reach (or exceed) *n*:
  - $log_2(n)$, often known in CS as $lg(n)$

- For an *n*-element list, it takes time proportional to ***n*** steps to decide whether the list contains a value, but only time proportional to $lg(n)$ to do the same thing on an ordered list
- What does that mean if n is 1,000,000? What about 1,000,000,000?

Aside: logarithms

- Recall:
  - $\log_a x = y \iff a^y = x$
  - Example: $2^5 = 32 \iff \log_2 32 = 5$

  - *$log_2(n)$*, often known in CS as *lg(n) (or sometimes log n)*
    - After all, binary is our favorite base .. :)

What about search in a tree?

- How efficient is _contains_ on each of the following:
  - our general Tree class?
  - our general BinaryTree class?
  - our BST class?
- The last case should probably be answered "depends..."

Search speed in Tree or BinaryTree

- Strategy – similar to lists:
  - We must look at elements one by one
  - Each time we eliminate only one element from consideration
  - In the worst case, we look at all n elements
- Either way, it takes time proportional to n

What about search in a BST?

- Recall the BST property
  - Exploit the ordering property to go either left or right when searching, but not both
  - => Search is narrowed down to only half of the yet-to-be-considered parts of the tree
  - If value is not in the tree, search all the way to leaf
  - => Worst case, maximum number of steps is equal to the max height of the tree
    - How big is the height in relation to the number of nodes?

Max nodes for height h

- What is the maximum number of nodes in a binary tree of height h:
  - 0? 0
  - 1? 1
  - 2? 3
  - 3? 7
  - 4? 15
  - h? $2^h - 1$

Height h based on number of nodes n

- Let's figure out tree height h for a given number of nodes n
- We know that $n \leq 2^h - 1$

  $\Rightarrow n + 1 \leq 2^h$

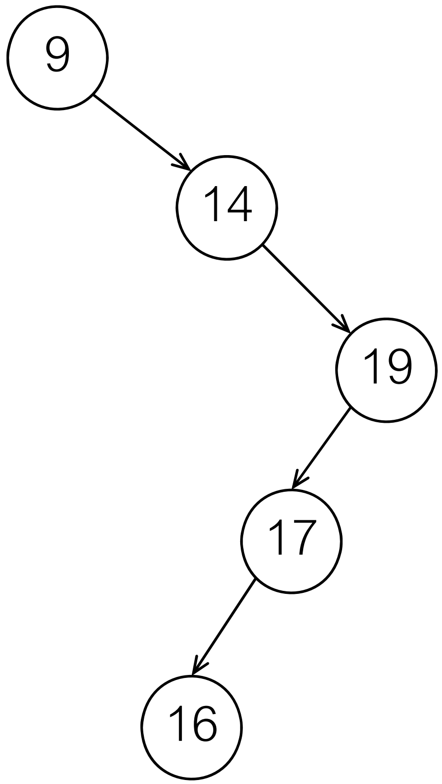
  $\Rightarrow \log_2 (n + 1) \leq h$

  $\Rightarrow h \geq \log_2 (n + 1)$

- So, time will be proportional to *lg n*
  - *Or will it?*

Search speed in BST

- Time will be proportional to *lg n,* only if the tree is balanced!
- Example (imbalanced tree):
  - Time takes proportional to *n* in this case



- In later courses, you will learn about balanced trees (AVL trees, red-black trees, etc.)

Other trees and getting to a leaf faster..

- If you have enormous amounts of data, binary trees won't cut it (getting to a leaf is still expensive)
- Databases are such examples (large volumes of data, must fit in memory)
  - Increase the arity/branching factor!
  - Make heavy use of B-trees..
  - You will see this in later courses (CSC343, CSC443)

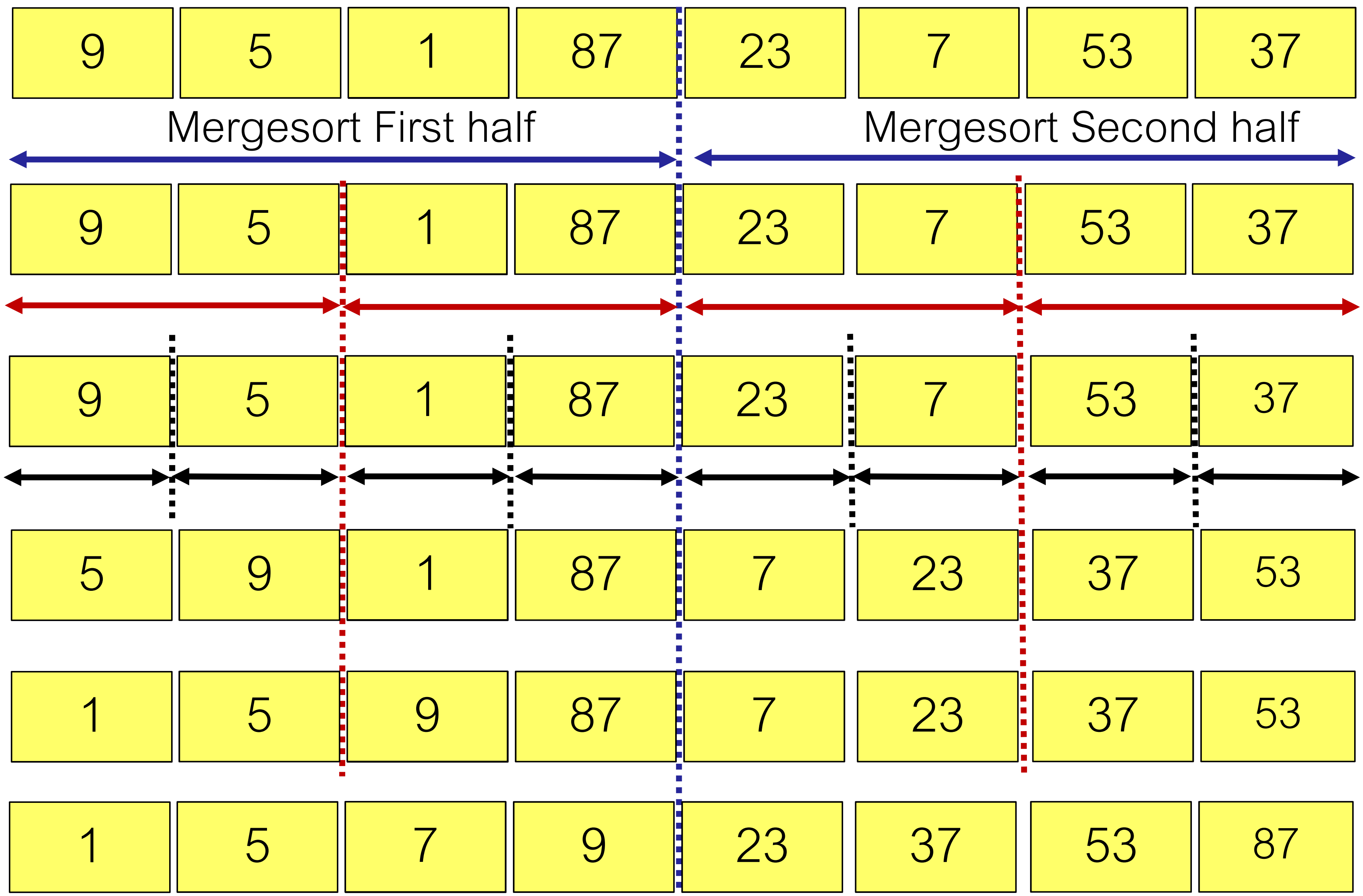
sorting

- How does the time to sort a list with n elements vary with n?
- It depends:
    - bubble sort -> n^2
    - selection sort -> n^2
    - insertion sort -> n^2
    - some other sort?
        - quick sort?
        - radix sort?
        - merge sort?

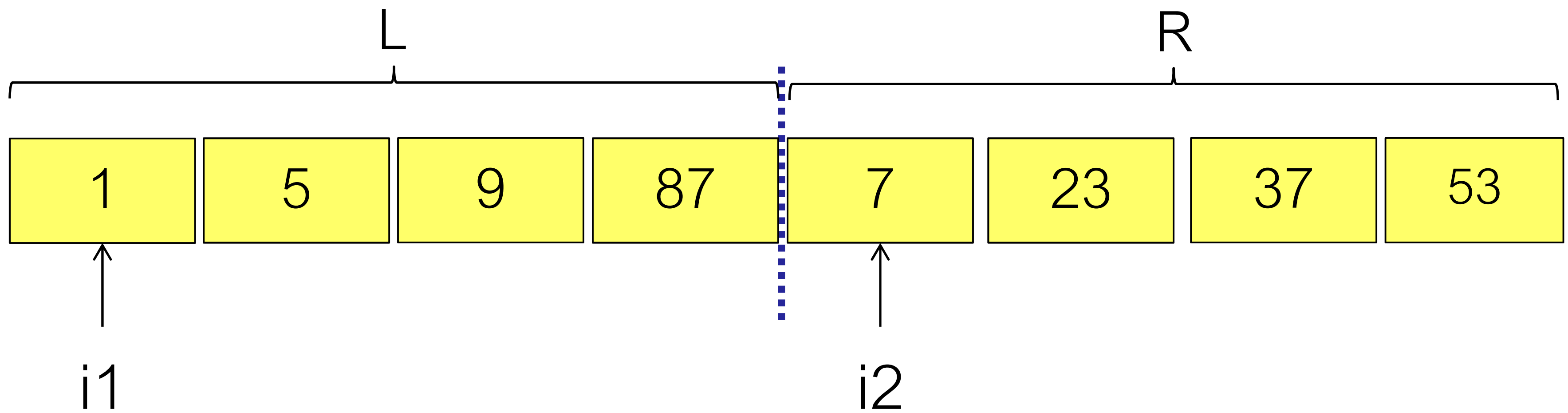
Mergesort

- Sorting algorithm
  - Split a list in two halves repeatedly
  - Halves with 0 or 1 elements are guaranteed sorted
  - Merge the two halves "on the way back"

Mergesort: split all the way, then merge



Merge step: merge(L, R)



sorted list (different than L or R)

1	5	7	9	23	37	53	87

The halves might not be perfectly equal though...

mergesort

- idea: break a list up (partition) into two halves, mergesort each half, then recombine (merge) the halves

```
def mergesort(list_):
    """
     Produce a copy of list_ in sorted order.
    """
    if len(list_) < 2:
        return list_[:]
    else:
        return merge(mergesort(list_[:len(list_) // 2]),
                     mergesort(list_[len(list_) // 2 :]))
```

Lists of length < 2 are already sorted

First half

Second half

Merge the two sorted halves, "on the way back". How?

Counting mergesort

- Assume a list of size n
- Merge operation takes linear time ... why?
- The "divide" step also takes linear time (approx n steps) ... why?
- What about the cost of the two recursive calls?

Counting mergesort: $n = 8$

prop. to

lg(n) splits

ms([4, 2, 6, 8, 1, 3, 5, 7])

merge(ms([4, 2, 6, 8]) , ms([1, 3, 5, 7]))

merge(merge(ms([4,2]), ms([6,8]) , merge(ms([1,3]), ms ([5,7])

merge(merge(merge(ms([4],ms([2]),merge(ms([6]),ms([8]), merge(merge(ms([1],ms([3]),merge(ms([5]),ms([7])

merge(merge(merge([4],[2]),merge([6],[8])), merge(merge([1],[3]),merge([5],[7])),

merge(merge([2,4]),[6,8])),merge([1,3]),[5,7]))

merge([2,4,6,8], [1,3,5,7])

[1, 2, 3, 4, 5, 6, 7, 8]

lg(n) merge with

prop. to n copies

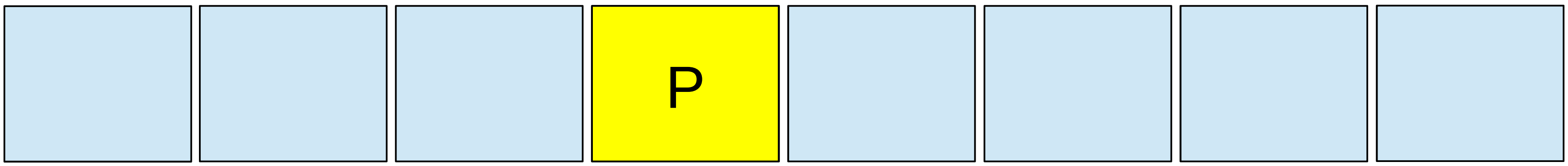
n x lg n

Quicksort

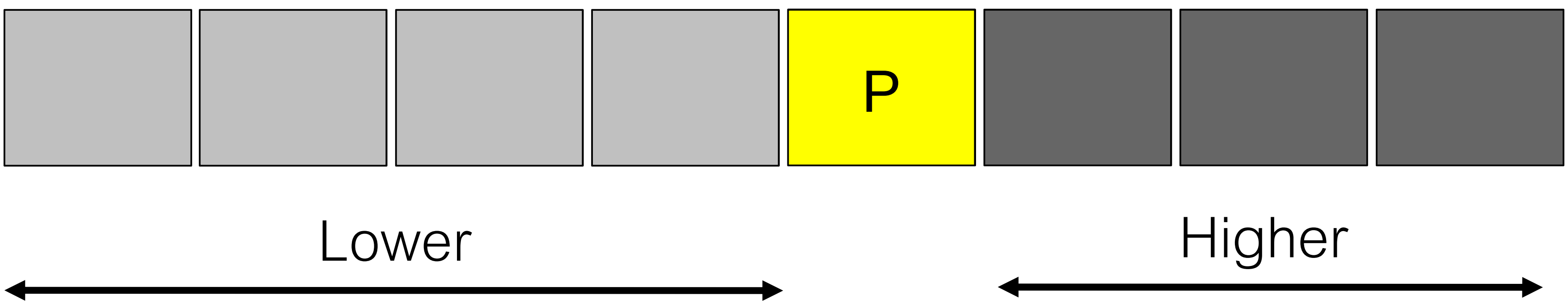
- Efficient sorting algorithm
- Works by:
  - Splitting a list (partitioning it) into the part smaller than some value (called pivot) and the part not smaller than that value
  - Sort these two parts
  - Recombine the list

Partition step

- Begin with the unsorted list and select a pivot P at random

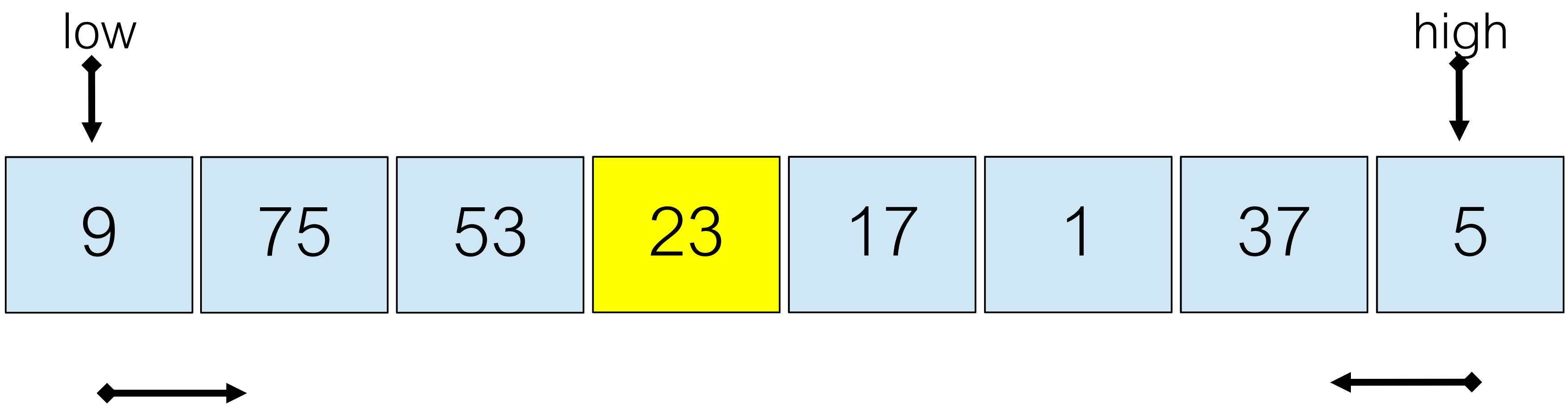


- Split list such that all elements to the left are lower than P and all to the right are higher



- Several ways to do the partition step ...

Partition step (more complex way..)

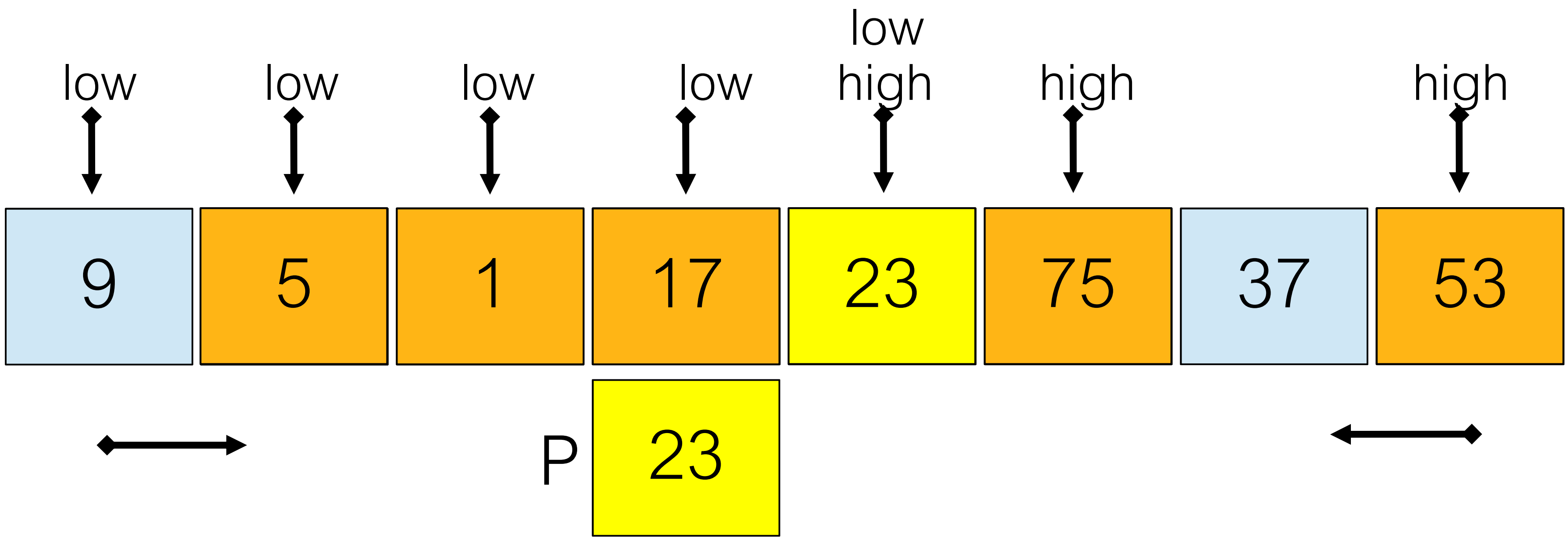


1. Take out pivot P and leave a hole in the list.
2. Increment low until we find: lst[low] >= P

   Move this element in the hole => new hole in list
3. Decrement high until we find: lst[high] <= P

   Move this element in the hole => new hole in list
4. Repeat steps 2 and 3, until low == high
5. Move P into the remaining hole.

Partition step (more complex way..)



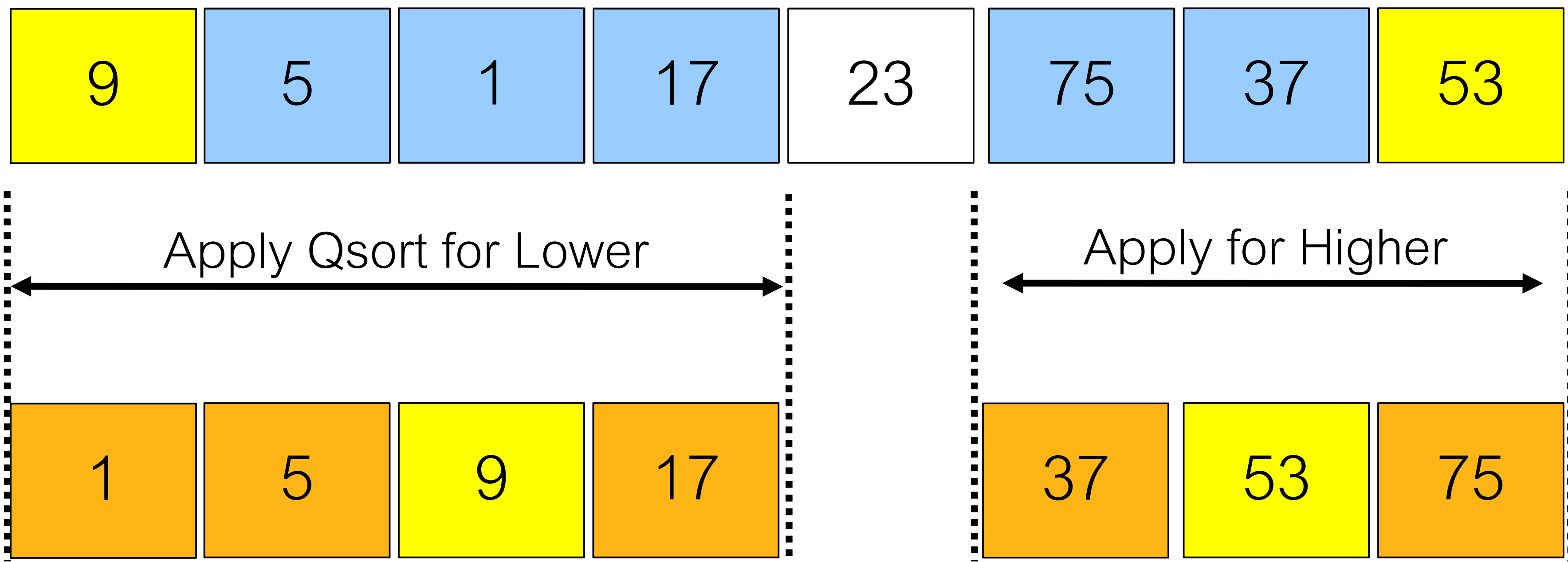
1. Take out pivot P and leave a hole in the list.
2. Increment low until we find: lst[low] >= P

   Move this element in the hole => new hole in list
3. Decrement high until we find: lst[high] <= P

   Move this element in the hole => new hole in list
4. Repeat steps 2 and 3, until low == high
5. Move P into the remaining hole.

Quicksort Recursion

- Recurse: repeat the same idea for the two partitions
- Pick pivot, process such that all lower than it are on the left, all higher on the right



Quicksort

- idea: break a list up (partition) into the part smaller than some value (pivot) and not smaller than that value, sort these parts, then recombine the list:

```
def qs(list_):
    """
     Return a new list consisting of the elements of list_ in ascending order.
     @param list list_: a list of comparables
     @rtype: list
    """
    if len(list_) < 2:
        return list_[:]
    else:
        smaller = [i  for  i  in  list_[1:]  if  i < list_[0]]
        larger = [i  for  i  in  list_[1:]  if  i >= list_[0]]
        return (qs(smaller) +
                [list_[0]] +
                qs(larger) )
```

Lists of length < 2 are already sorted

Simpler partition step

Sort smaller elements

in its correct position

Sort larger elements

Counting quicksort: $n = 7$

lg(n) splits

n comparisons at each split

qs([4, 2, 6, 1, 3, 5, 7])

qs([2, 1, 3]) + [4] + qs([6, 5, 7])

qs([1]) + [2] + qs([3]) + [4] + qs([5]) + [6] + qs([7])

[1] + [2] + [3] + [4] + [5] + [6] + [7]

[1, 2, 3] + [4] + [5, 6, 7]

[1, 2, 3, 4, 5, 6, 7]

lg(n) concatenation, with n work to concatenate

n x lg n

Quicksort analysis – digging deeper

- Do we always have *n log n* though?
  - Mergesort: we know we always split in halves, no matter what
  - Quicksort: no guarantees, depends on how we pick the pivot
  - What's the worst case?

reverse_list ...