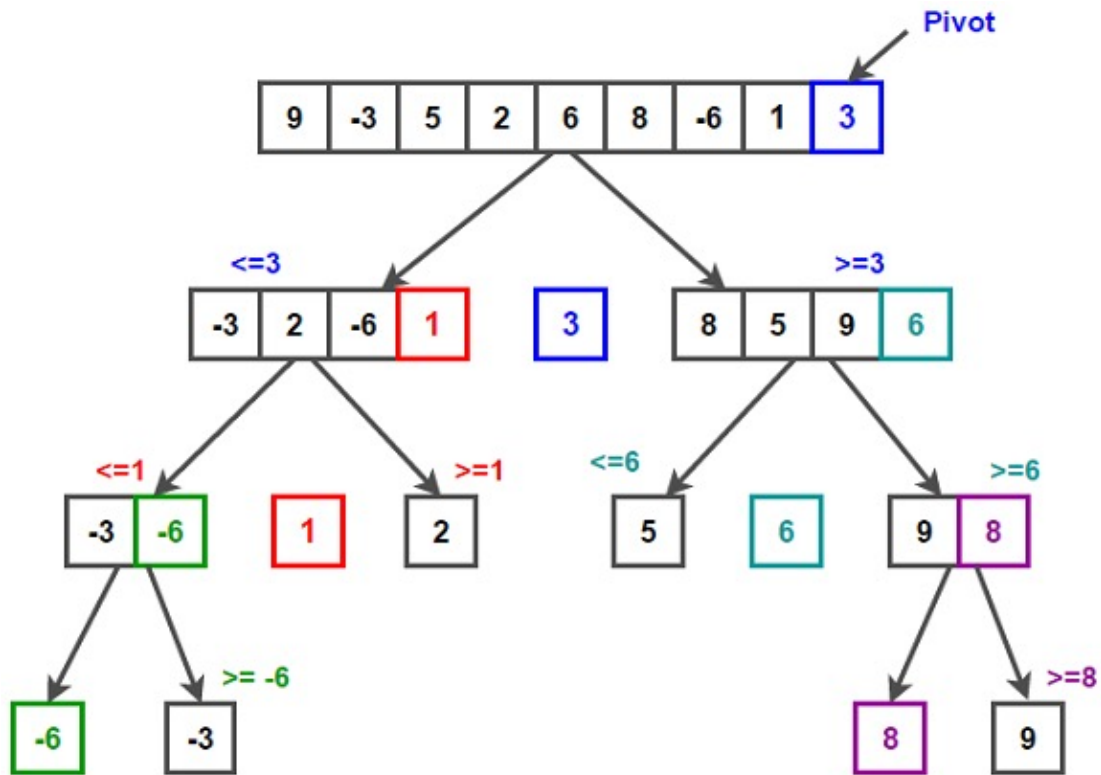


# Quicksort

CLRS 7.1 – 7.4  
(+ some supplemental material)



# Recap

- **Divide-and-conquer** is a general algorithm design paradigm:
  - **Divide**: divide the input data  $S$  into disjoint subsets
  - **Conquer**: solve the subproblems associated with smaller subproblems
    - the base case for the recursion are subproblems of size 0 or 1
  - **Combine**: combine the solutions for subproblems into a solution for  $S$
- Merge sort was a divide and conquer approach
  - Divide into 2 lists
  - Recursively sort lists
  - **Merge** two now sorted lists into one sorted list
- Our 2 lists were not sorted *with respect to each other*, so the bottleneck of this approach was in the **merge** step.
- What if we were more careful with how we divided starting array?

# Quicksort

Quicksort works on an input sequence with  $n$  elements and consists of three steps:

- **Divide:** **partition** the  $n$ -element sequence to be sorted into lists based on a select **pivot** element  $x$ 
  - subsequence 1: list of other elements  $\leq x$
  - subsequence 2: list of other elements  $> x$
- **Conquer:** sort the two subsequences recursively using quicksort
- **Combine:** subsequences are already sorted internally and with respect to each other, so no work is needed to combine



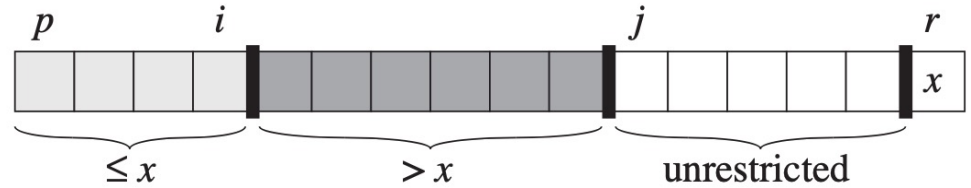
QUICKSORT( $A, p, r$ )

- 1 **if**  $p < r$
- 2      $q = \text{PARTITION}(A, p, r)$
- 3     QUICKSORT( $A, p, q - 1$ )
- 4     QUICKSORT( $A, q + 1, r$ )

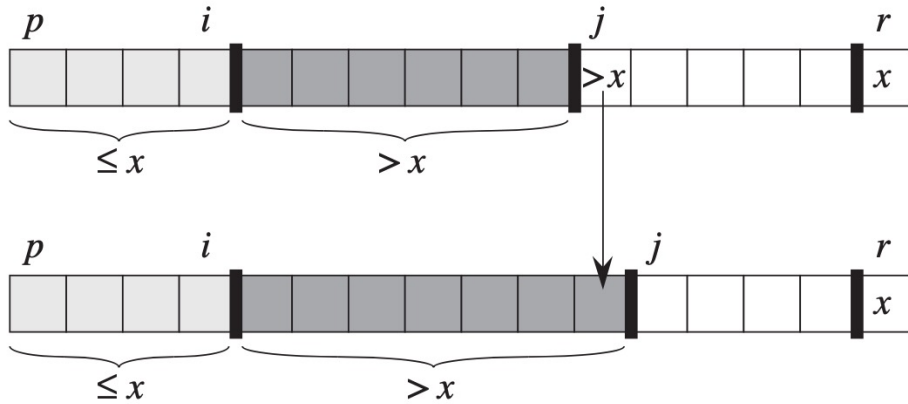
Quicksort is initially called as *Quicksort*( $A, 1, A.length$ )

# Quicksort partition

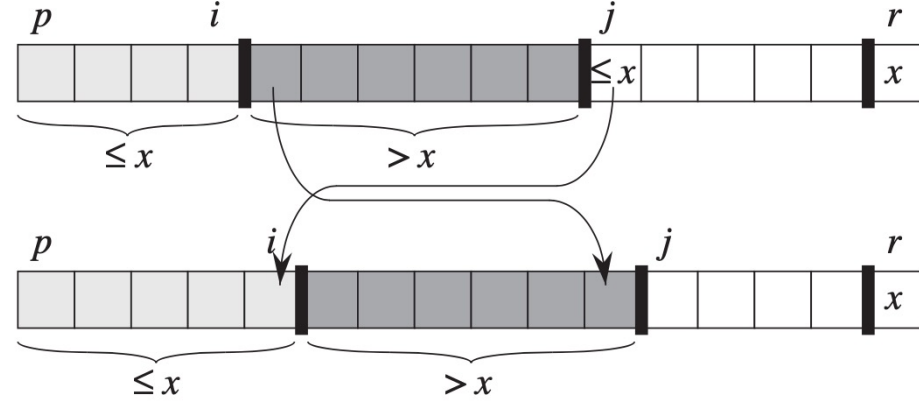
- Maintain four partitions
- As the next item is processed, compare it to the pivot to determine which subsequence it belongs to



*If next item is greater than the pivot*



*If next item is less than or equal to the pivot*



PARTITION( $A, p, r$ )

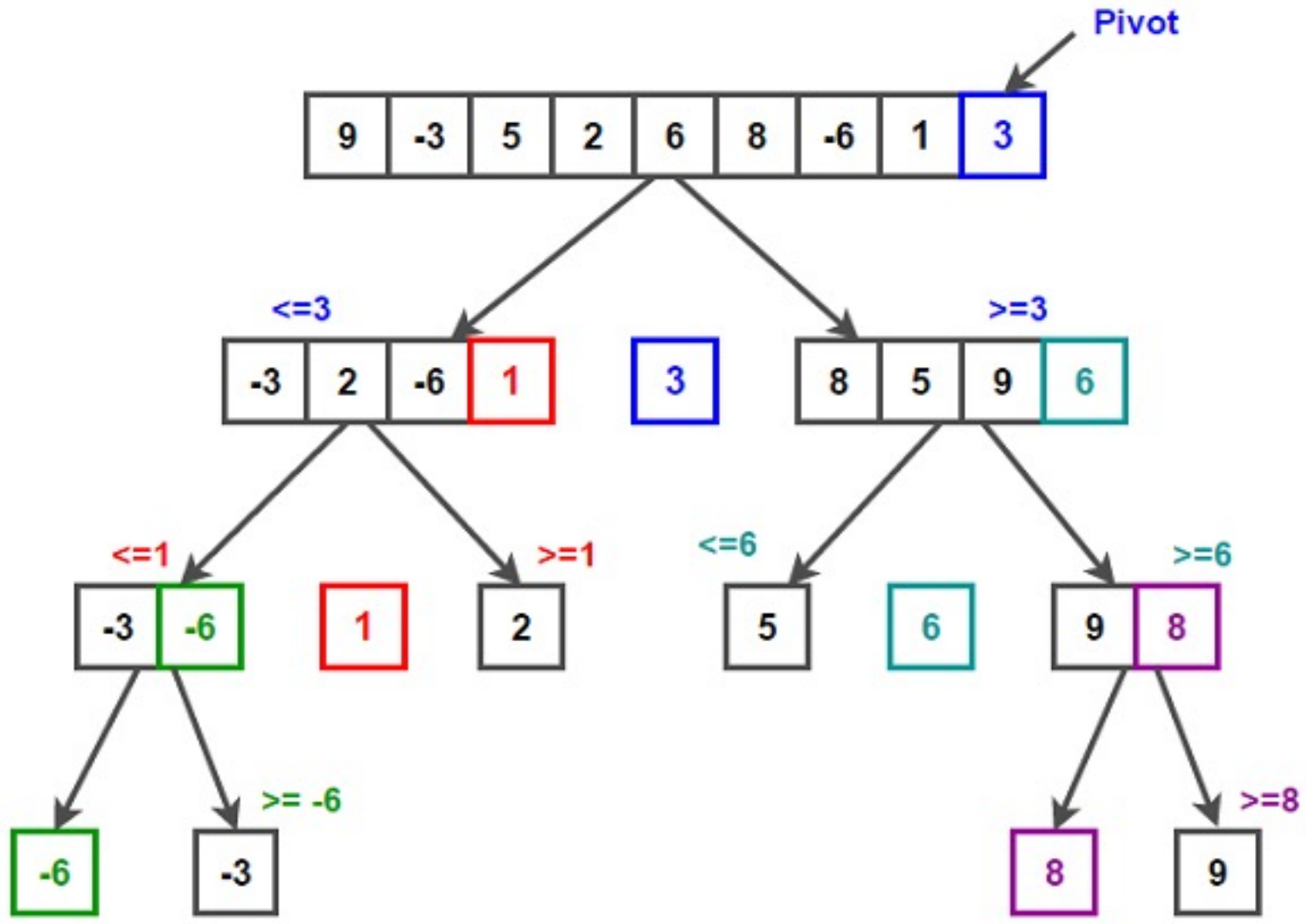
```

1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
    
```

Partitions  $n$  elements  
in-place in  $O(n)$  time

# Illustration of the execution of quicksort

- Using the last element as pivot



# Quicksort run time

- Like mergesort, the non-recursive overhead at each level is  $O(n)$ 
  - This is the cost of the partitioning method
- Q: How many recursive calls can be made in the worst case?
  - $O(n)$  – this happens if we pick a **bad pivot** each time -- the minimum or maximum element
  - Example: if the list is already sorted
- **Worst-case** run time:  $O(n^2)$  -- happens if we pick a bad pivot each time
- **Best-case** run time:  $O(n \log n)$  -- happens if we pick a good pivot each time
- We can argue about the average case, provided that we know some information about the input sequence
  - **Assuming the input list is randomly distributed**, we can show that the last element is **usually** a good pivot
  - In this case, the previous version of quicksort runs in  $O(n \log n)$  average time
- Rather than making assumptions about the input, we can instead use a randomized version of quicksort which picks a random pivot each time.

# Randomized algorithms

There are some variations

- **Las Vegas algorithms**
  - Correct output guaranteed
  - Randomization makes **good average running time**
  - Introduced by Hungarian professor at University of Chicago László Babai
  - **Gambles with running time**
- **Monte Carlo algorithms**
  - Running time guaranteed
  - Randomization makes **probably correct output**
  - **Gambles with correctness**

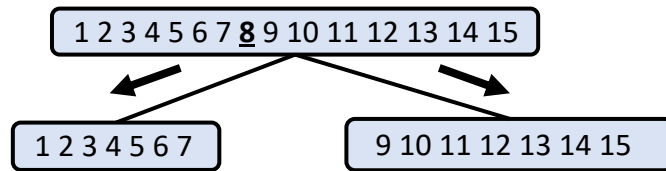
The following randomized version of quicksort is a Las Vegas algorithm.

- The only difference is that we pick always pick a random pivot, rather than the last element as pivot

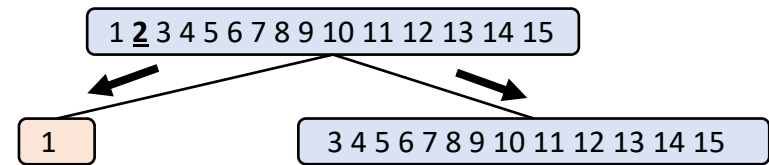
# Expected running time of randomized quicksort

Consider a recursive call of quick-sort on a sequence of size  $s$

- **Good call:** the sizes of L and G are each less than  $3s/4$
- **Bad call:** one of L and G has size greater than  $3s/4$



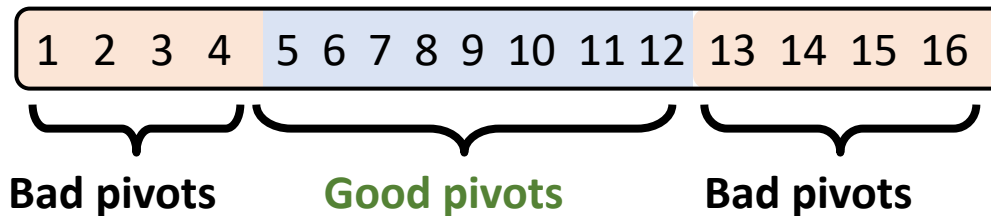
**Good call**



**Bad call**

A call is **good** with probability  $\frac{1}{2}$

- $\frac{1}{2}$  of the possible pivots cause good calls
- We can show this by visualizing an already sorted list, and counting the number of good pivots





# Expected running time of randomized quicksort

**Probabilistic Fact:** The expected number of coin tosses required in order to get  $k$  heads is  $2k$ .

For a node of depth  $i$ , we expect

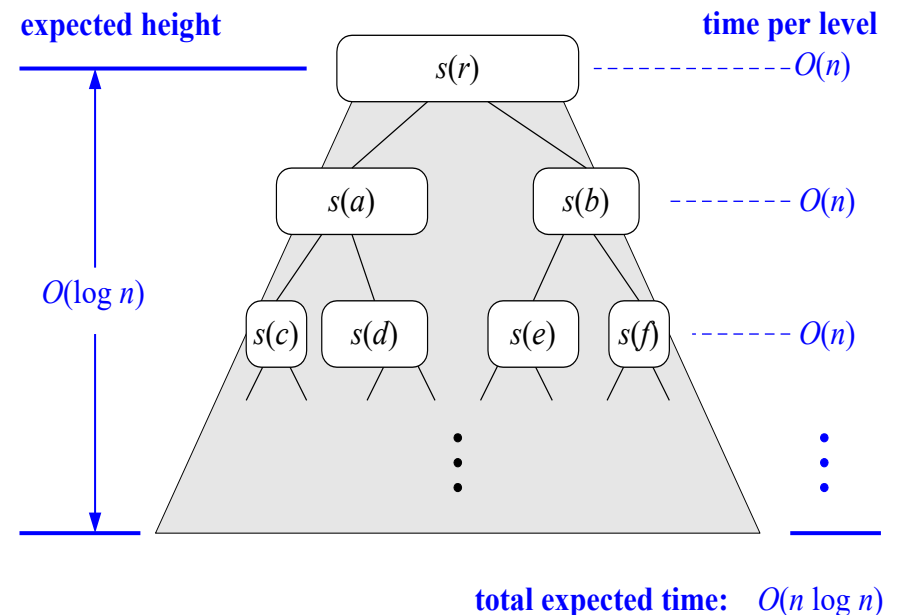
- $i/2$  ancestors are good calls
- size of the input sequence for the current call is at most  $\left(\frac{3}{4}\right)^{\frac{i}{2}} n$

For a node of depth  $2 \log_{4/3} n$   
the expected input size is one

- the expected height of the quick-sort tree is  $O(\log n)$

The amount of work done at the nodes of the same depth is  $O(n)$

Thus, the expected running time of randomized quick-sort is  $O(n \log n)$



```

DEFINE HALFHEARTEDMERGESORT(LIST):
  IF LENGTH(LIST) < 2:
    RETURN LIST
  PIVOT = INT(LENGTH(LIST) / 2)
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
  // UMMMMM
  RETURN [A, B] // HERE. SORRY.

```

```

DEFINE FASTBOGOSORT(LIST):
  // AN OPTIMIZED BOGOSORT
  // RUNS IN O(N LOG N)
  FOR N FROM 1 TO LOG(LENGTH(LIST)):
    SHUFFLE(LIST):
      IF ISSORTED(LIST):
        RETURN LIST
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

```

```

DEFINE JOBIINTERVIEWQUICKSORT(LIST):
  OK SO YOU CHOOSE A PIVOT
  THEN DIVIDE THE LIST IN HALF
  FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    NO, WAIT, IT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PIVOT
    THE BIGGER ONES GO IN A NEW LIST
    THE EQUAL ONES GO INTO, UH
    THE SECOND LIST FROM BEFORE
  HANG ON, LET ME NAME THE LISTS
  THIS IS LIST A
  THE NEW ONE IS LIST B
  PUT THE BIG ONES INTO LIST B
  NOW TAKE THE SECOND LIST
  CALL IT LIST, UH, A2
  WHICH ONE WAS THE PIVOT IN?
  SCRATCH ALL THAT
  IT JUST RECURSIVELY CALLS ITSELF
  UNTIL BOTH LISTS ARE EMPTY
  RIGHT?
  NOT EMPTY, BUT YOU KNOW WHAT I MEAN
  AM I ALLOWED TO USE THE STANDARD LIBRARIES?

```

```

DEFINE PANICSORT(LIST):
  IF ISSORTED(LIST):
    RETURN LIST
  FOR N FROM 1 TO 10000:
    PIVOT = RANDOM(0, LENGTH(LIST))
    LIST = LIST[PIVOT:] + LIST[:PIVOT]
  IF ISSORTED(LIST):
    RETURN LIST
  IF ISSORTED(LIST):
    RETURN LIST:
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING
    RETURN LIST
  IF ISSORTED(LIST): // COME ON COME ON
    RETURN LIST
  // OH JEEZ
  // I'M GONNA BE IN SO MUCH TROUBLE
  LIST = [ ]
  SYSTEM("SHUTDOWN -H +5")
  SYSTEM("RM -RF ./")
  SYSTEM("RM -RF ~/*")
  SYSTEM("RM -RF /")
  SYSTEM("RD /S /Q C:\*") // PORTABILITY
  RETURN [1, 2, 3, 4, 5]

```

# Other: nuts and bolts

You are given a collection of  $n$  bolts of different widths, and  $n$  corresponding nuts.

- You can test whether a given nut and bolt fit together, from which you learn whether the nut is too large, too small, or an exact match for the bolt.
- The differences in size between pairs of nuts or bolts are too small to see by eye, so you cannot compare the sizes of two nuts or two bolts directly.
- You are to match each bolt to each nut.

Give an efficient algorithm to solve the nuts and bolts problem.

