Randomized approaches. Quick sort

Lecture 06.04

by Marina Barsky

Back to sorting

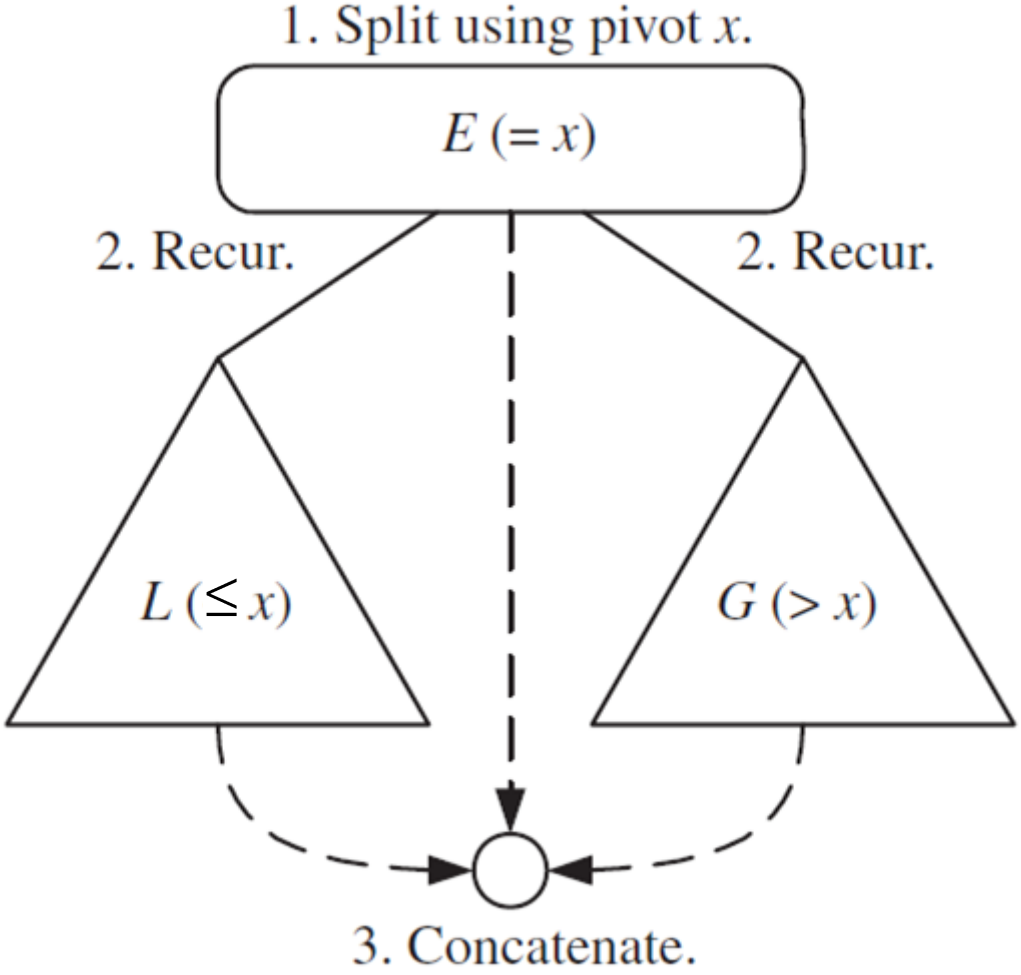
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.toptal.com/developers/sorting-algorithms

Quicksort: overview

- Divide array A into 2 subarrays
- Recursively sort each subarray
- Combine the sorted subarrays by a simple concatenation

Main idea



Select an element called ***pivot***

1. Divide elements into 2 groups *L* (less or equal), and *G* (greater than pivot)
2. Conquer: recursively sort *L* and *G*
3. Combine: concatenate L→E→R

Example: quick sort

6	4	8	2	9	3	9	4	7	6	1

Example: quick sort

Rearrange elements with respect to

$x = A[1]$

≤ 6 > 6

Example: quick sort

6 is in its final position

sort the two parts recursively

QuickSort(*A*, ℓ, *r*)

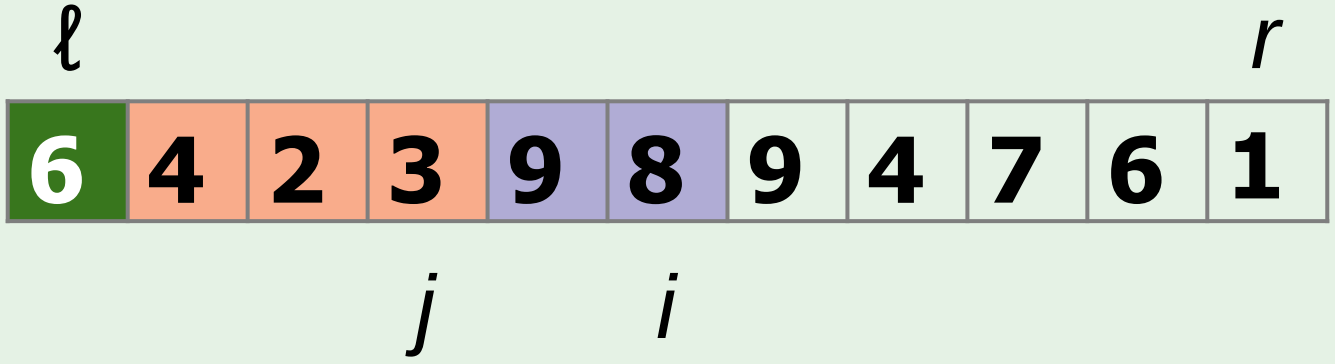
```
if ℓ ≥ r :
    return
m ← Partition(A, ℓ, r )
# A[m] is in the final position
QuickSort(A, ℓ, m − 1)
QuickSort(A, m + 1, r )
```

m is the final position of element **A[ℓ]**

PIVOT

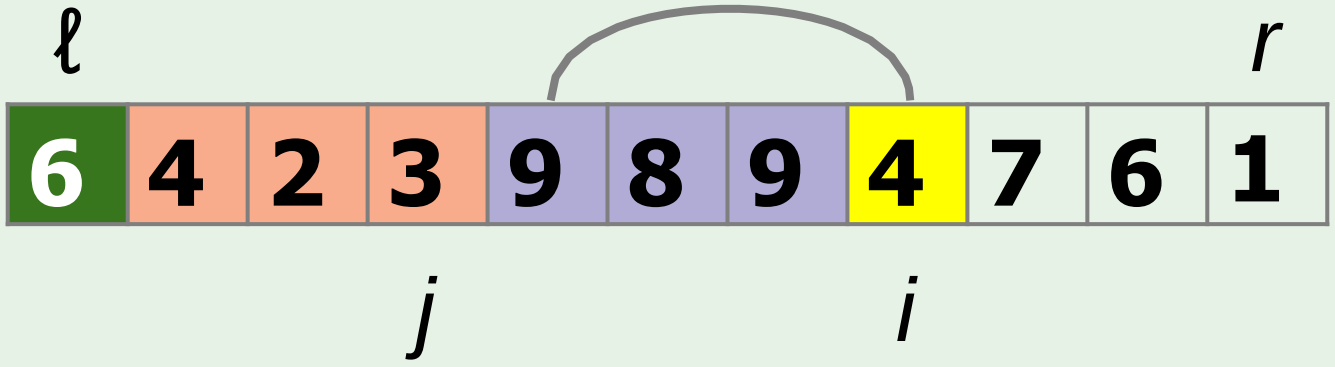
Partitioning: example

- the **pivot** is $x = A[\ell]$
- loop i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$



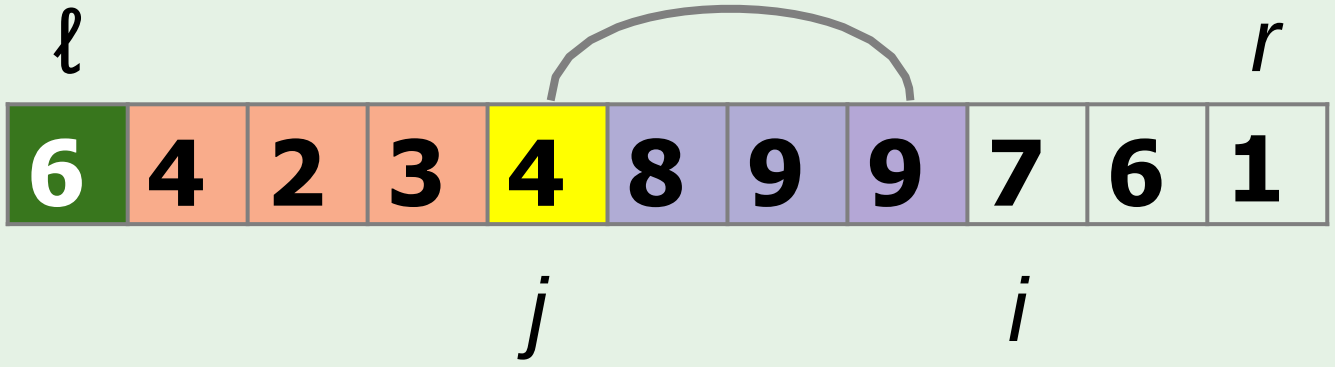
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \le x$ for all $\ell + 1 \le k \le j$
  - $A[k] > x$ for all $j + 1 \le k \le i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



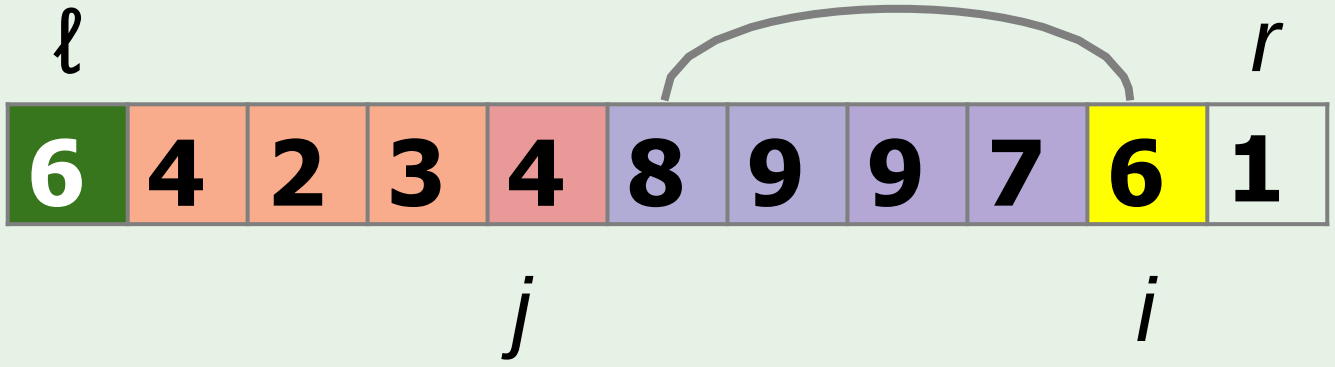
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



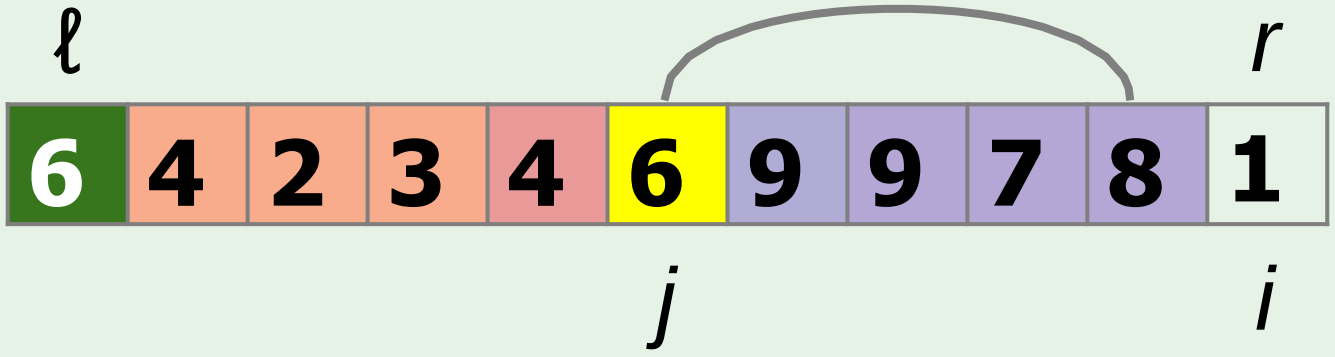
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



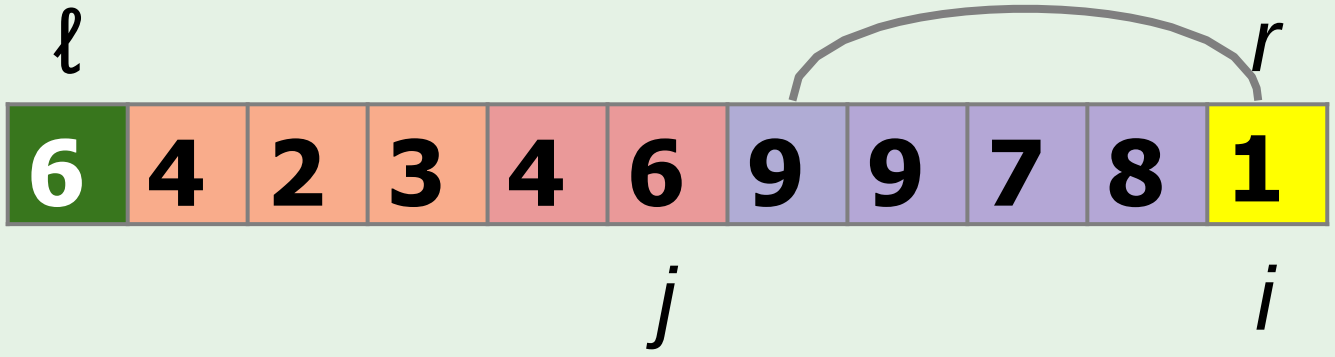
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



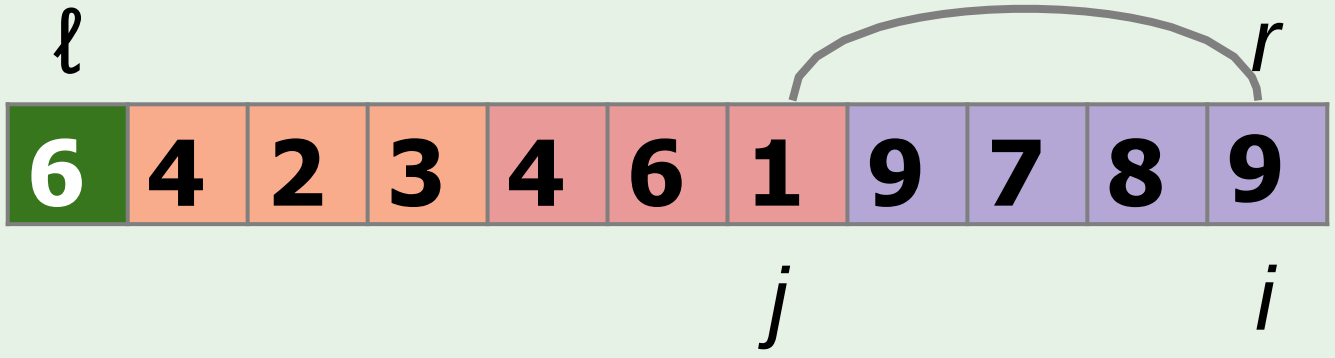
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



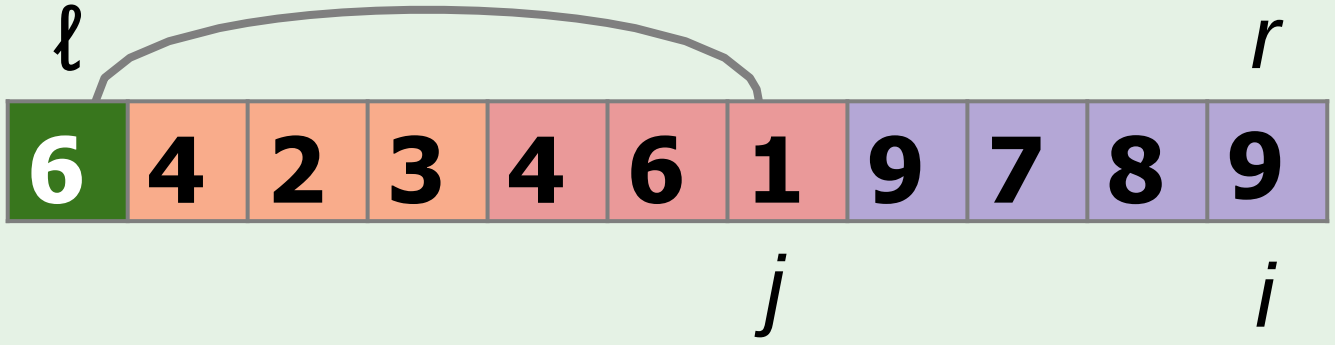
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



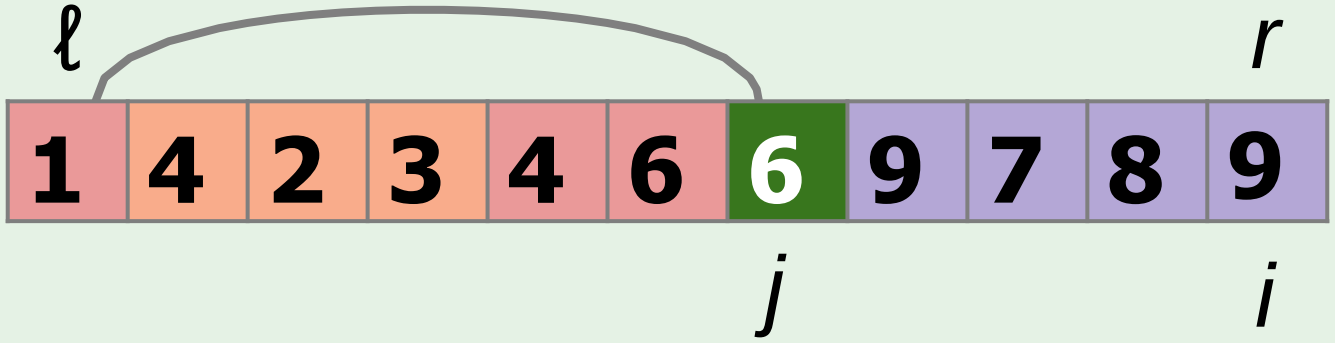
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- in the end, move $A[\ell]$ to its final place j



Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- in the end, move $A[\ell]$ to its final place j



Partition(A, ℓ, r)

```
x ← A[ℓ]  # pivot
j ← ℓ
for i from ℓ + 1 to r :
    if A[i] ≤ x :
        j ← j + 1
        swap A[j] and A[i]
swap A[ℓ] and A[j]
return j
# A[ℓ + 1 . . . j] ≤ x , A[j + 1 . . . i] > x
```

Quick Sort: summary

- ❑ Simple
- ❑ Comparison-based
- ❑ Very fast in practice

Running time of Quick Sort

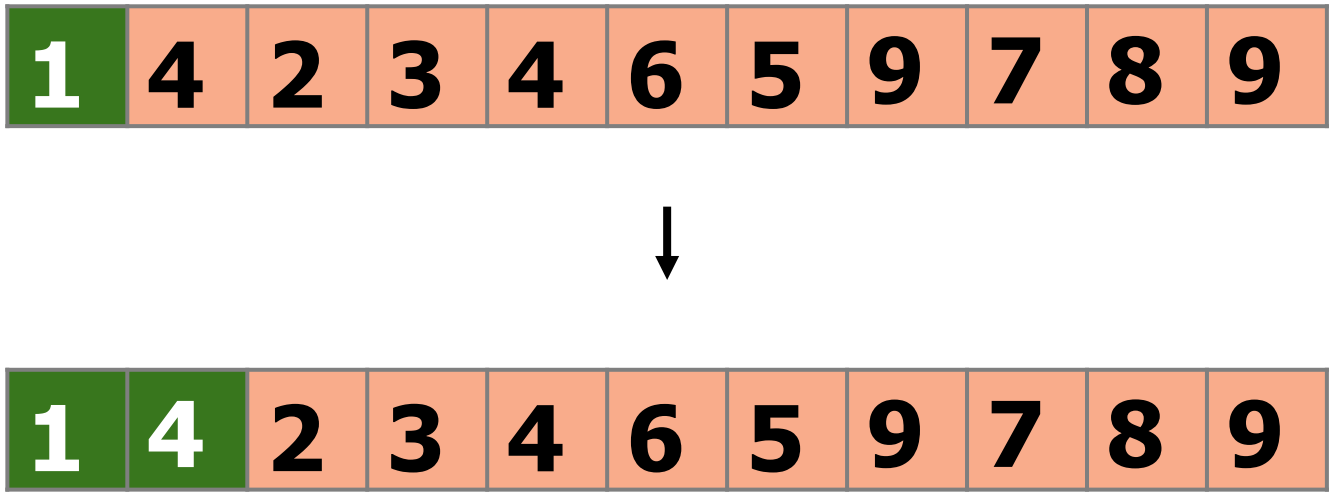
If we happen to choose the pivot x in such a way that after the partitioning the array A is split into even halves:

$$T(n) = 2T(n/2) + n$$

$$T(n) = O(n \log n)$$

Unlucky choice of pivot

If we choose a pivot in such a way that **all values are greater than it**, then we decrement a size of the problem only by 1:



Unlucky choices of pivot

$T(n) = n + T(n-1)$

$T(n) = n+(n-1)+(n-2)+\cdots = O(n^2)$

If n is decreasing by a constant number at each step:

$T(n) = n + T(n-5) + T(4)$:

$T(n) = n+(n-5)+(n-10)+\cdots = O(n^2)$

Still arithmetic series of the form 1 …. n with d=5, and total of n/5 elements
Sum = n(a1 + an)/2 = n/5*(1 +n)/2 = 1/10 *n(n+1)

Worst-case running time of Quick Sort

$$T(n) = O(n^2)$$

Pathological case

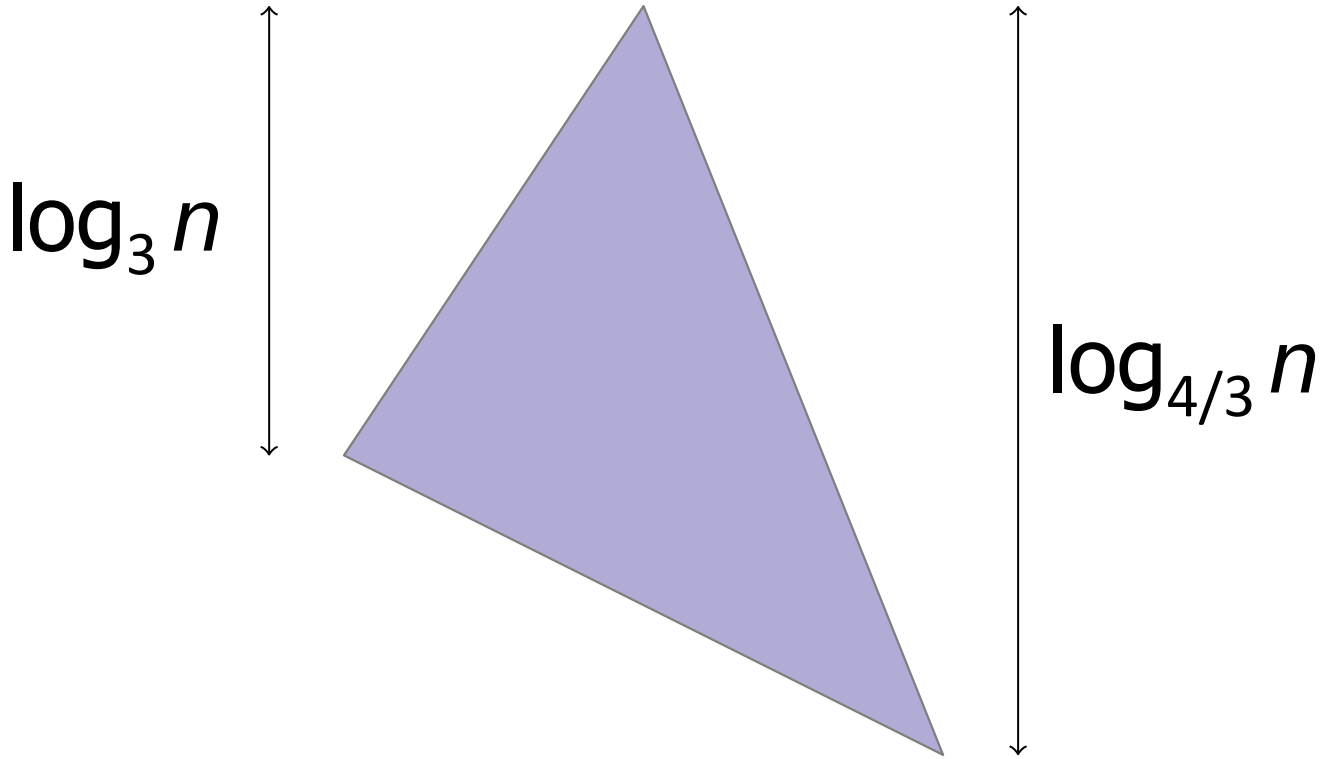
It requires $O(n^2)$ time to process the already sorted array which seems unnecessary since the array is already sorted!

Goal: Balanced Partitions

- The *QuickSort* algorithm so far seems like a bad imitation of *MergeSort*
- If we only could choose a good "splitter" *x* that breaks an array into two equal parts!
  - To achieve $O(n \log n)$ running time, it is not actually necessary to find a perfectly equal (50/50) split
  - The algorithm will achieve $O(n \log n)$ running time even if *{<=m}* and *{>m}* are both at least $n/4$ (or n/c at each step).

Balanced Partition

$$T(n) = T(n/4) + T(3n/4) + O(n)$$



The largest number of steps to go from problem size n to problem size 1

The running time is still O(n log n), as long as we can guarantee that the input is split into at least n/c + (n – n/c)

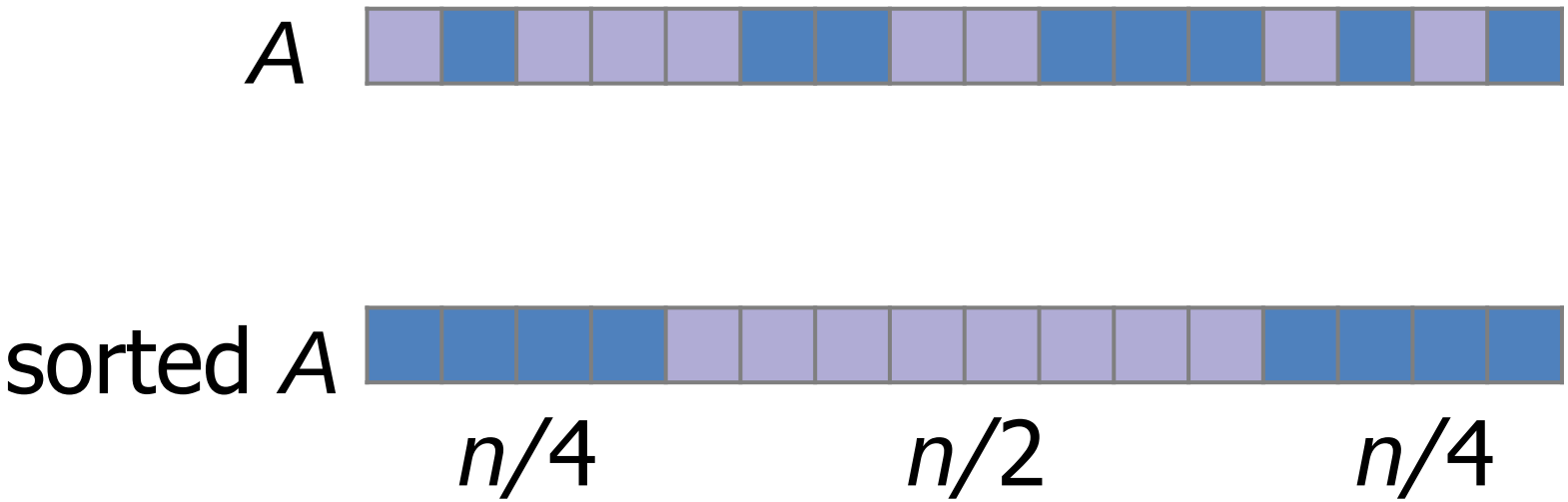
Choosing random pivot

- It implies that of n possible choices of pivot at least $3n/4-n/4 = n/2$ of them make good splitters!

- If we choose x randomly there is at least 50% chance that a good pivot will be chosen!

Why Random?

Half of the elements of *A* guarantee a balanced partition:



RandomizedQuickSort(*A*, ℓ, *r*)

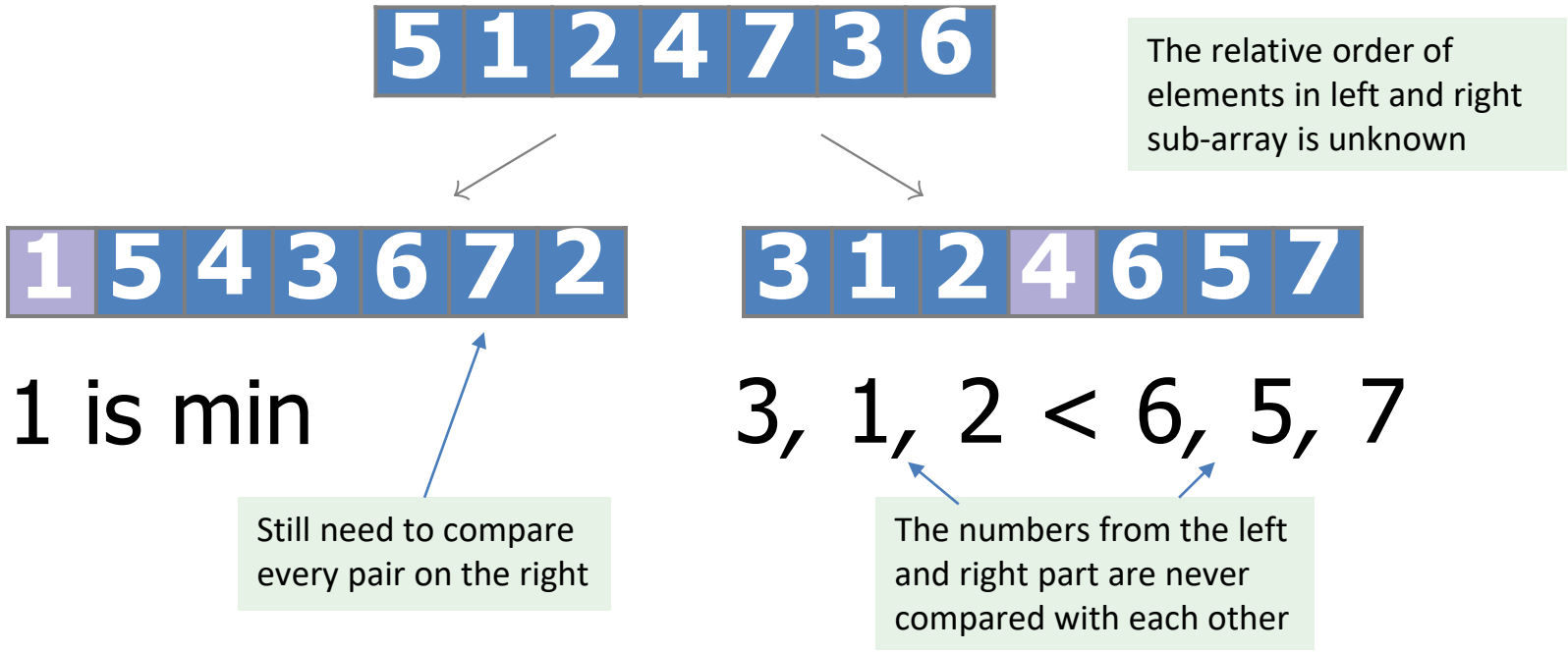
```
if ℓ ≥ r :
    return
k ← random number between ℓ and r
swap A[ℓ] and A[k]
m ← Partition(A, ℓ, r )
# A[m] is in the final position
RandomizedQuickSort(A, ℓ, m – 1)
RandomizedQuickSort(A, m + 1, r )
```

Theorem

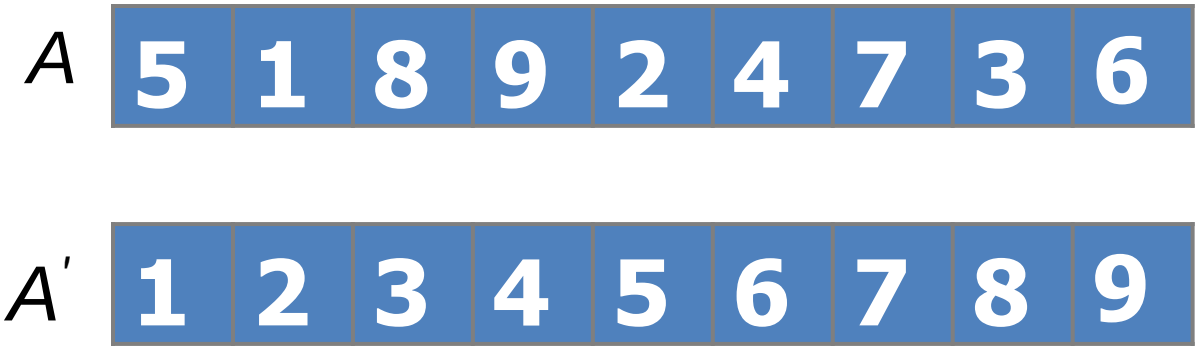
Assume that all the elements of $A[1 \ldots n]$ are distinct. Then the expected running time of ***RandomizedQuickSort***(A) is $O(n \log n)$.

Intuitive idea 1: total comparisons

- The running time is proportional to the number of comparisons made
- Balanced partitions are better since they reduce the number of comparisons needed:



Intuitive idea 2: probability of comparisons



Prob (1 and 9 are compared) = 2/9

Prob (3 and 4 are compared) = 1

The probability that 2 numbers are compared through the entire algorithm depends on how close are these numbers in their sorted order A'

Probability recap: random variable

A random variable is a numerical description of the outcome of a statistical experiment.

Example: Let χ be the random variable that equals the number of heads that appear when the unbiased coin is flipped 3 times.

Then χ takes on the following values:

TTT $\rightarrow$ 0
HTT, THT, TTH $\rightarrow$ 1
HHT, THH, HTH $\rightarrow$ 2
HHH $\rightarrow$ 3

Expected value

The expected value of a random variable is the sum over all outcome values of the product of the probability of this outcome and the value of this outcome.

In other words, the expected value is a weighted average of the values of a random variable.

TTT → 0
HTT, THT, TTH → 1
HHT, THH, HTH → 2
HHH → 3

Example continued:

$E(\mathcal{X}) = 0*1/8 + 1*3/8 + 2*3/8 + 3*1/8 = 1.5$

The expected number of heads in 3 tosses is 1.5 heads

Expected value in algorithm analysis

Many questions can be formulated in terms of the expected value of a random variable - which is its average value when an experiment is performed a large number of times.

Another example: expected value of a die
Let X be the number that comes up when a fair die is rolled.
What is the expected value of X?

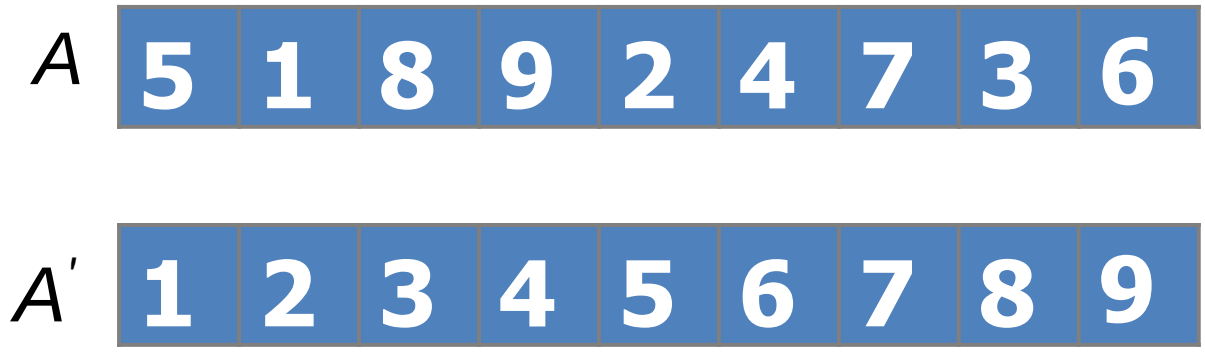
The random variable X takes values 1, 2, 3, 4, 5, or 6, each with probability 1/6.

$E(X) = 1*1/6 + 2*1/6 + 3*1/6 + 4*1/6 + 5*1/6 + 6*1/6 = 3.5$

For algorithms:

- What is the expected number of comparisons used to find an element in a list using a linear search?
- What is the expected number of collisions produced by a particular hash function
- …

Back to Randomized Quicksort



- let, for $i < j$, random variable χ be defined as:

$$\chi_{ij} = \begin{cases} 1 & A'[i] \text{ and } A'[j] \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

- for all intervals $[i : j]$, $A'[i]$ and $A'[j]$ are either compared exactly once or not compared at all (as we compare with a pivot)

Expected # comparisons for 2 numbers

- let, for $i < j$, random variable χ be defined as:

$$\chi_{ij} = \begin{cases} 1 & A'[i] \text{ and } A'[j] \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

- for all $i < j$, $A'[i]$ and $A'[j]$ are either compared exactly once or not compared at all

- crucial observation: $\chi_{ij} = 1$ iff the first selected pivot in $A'[i : j]$ is $A'[i]$ or $A'[j]$

Expected # comparisons for 2 numbers

$$\chi_{ij} = \begin{cases} 1 & A'[i\,] \text{ and } A'[j] \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

- crucial observation: $\chi_{ij} = 1$ iff the first selected pivot in $A'[i : j]$ is $A'[i]$ or $A'[j]$ (2 cases out of $(j - i + 1)$ total cases
- then Prob(χ_{ij}) = 2/(j - i +1)
- and for each pair *i,j:*

  E (χ_{ij}) = 1*2/(j - i + 1) + 0*(i – j)

  E (χ_{ij}) = 2/(j - i + 1)

Expected total number of comparisons

Then (the expected value of) the running time over all combinations of *i* and *j* is:

$$E = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \chi_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(\chi_{ij})$$

$$= 2 \sum_{i<j}^{n} 1/(j - i + 1)$$

$$\leq 2n \cdot (1/2 + 1/3 \ldots 1/n)$$

$$= O(n \log n)$$ ■

```
sum: = 0
for i from 1 to n
    for j from i+1 to n
        sum: += 2/(j – i + 1)
```

Harmonic series!

Repeated at most n times

sum = (1/2 + 1/3 + 1/4 + … + 1/n)
+ 1/3 + 1/4 + … + 1/n
+ … 1/n

Theorem

Assume that all the elements of $A[1 \ldots n]$ are **distinct**. Then the expected running time of ***RandomizedQuickSort***(A) is $O(n \log n)$.

Proven ■

Problem: Equal Elements

- What if all the elements of the given array are equal to each other?
- The array is always split into two parts of size 0 and $n - 1$

  $T(n) = n + T(n - 1) + T(0)$ and hence

  $T(n) = \Theta(n^2)$!

To handle equal elements, we replace the line

$m \leftarrow \text{Partition}(A, \ell, r)$

with the line

$(m_1, m_2) \leftarrow \text{Partition3}(A, \ell, r)$

such that

- for all $\ell \le k \le m_1 - 1$, $A[k] < x$
- for all $m_1 \le k \le m_2$, $A[k] = x$
- for all $m_2 + 1 \le k \le r$, $A[k] > x$

Ternary quicksort

RandomizedQuickSort3(*A*, ℓ, *r*)

if ℓ ≥ *r* :

    return

k ← random number between ℓ and *r*

swap *A*[ℓ] and *A*[*k*]

(m_1, m_2) ← Partition3(*A*, ℓ, *r*)

A[m_1 . . . m_2] are in final position

RandomizedQuickSort3(*A*, ℓ, $m_1 - 1$)

RandomizedQuickSort3(*A*, $m_2 + 1$, *r*)

Quick sort: Summary

- Quick sort is a comparison-based algorithm based on random partitioning
- Expected running time: $O(n \log n)$
- $O(n^2)$ in the worst case
- Very fast in practice

Quicksort: randomized running time, not the results

- The key advantages of randomized algorithms: performance and simplicity
- Note that *RandomizedQuickSort*, despite making random decisions, **always returns the correct solution of the sorting problem**
- The only variable from one run to another is its running time, not the result

Two types of randomized algorithms:

Las Vegas

- A *Las Vegas algorithm* is a randomized algorithm that always gives the correct result but gambles with resources
- It always returns the correct answer, but its runtime bounds hold only in expectation

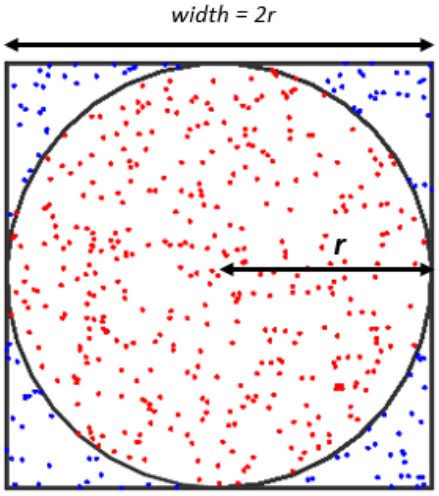
Two types of randomized algorithms:

Monte Carlo

- A *Monte Carlo algorithm* is a randomized algorithm whose output may be incorrect with a certain, typically small, probability.
- A Monte Carlo algorithm may fail or return incorrect answers, but has runtime independent of the randomness.

Monte Carlo: example

Compute value of π



Square area: $S=(2r)^2$ (total number of points)
Circle area: $C=\pi r^2$(number of points inside circle)
$C/S = \pi/4$
$\pi = 4*C/S$

```
count:=0
repeat n times:
    generate random point (x, y)
                where -1 < x < 1 and -1 < y < 1
    if x² + y² ≤ 1
        increment  count
return 4*count/n
```

Median

Activity

The *Median* problem

The *median* of a list of numbers is its 50th percentile:

- Half the numbers are bigger than it, and half are smaller
- It is a middle element when the numbers are arranged in order

[45, 1, 10, 30, 25] → [1, 10, 25, 30, 45]

median: 25

If the list has even length, there are two choices, in which case we pick the smaller of the two, say.

Median problem

Input: array *A* of *n* elements
Output: Median - the middle value of elements in A

Motivation:

- The purpose of the median is to summarize a set of numbers by a single, typical value
- The *mean*, or average, is also very commonly used for this, but the median is more typical of the data - it is always one of the data values and it is less sensitive to outliers

[1,1,1,1,1,1,1,1,1,1,100]
What is the mean? And what is the median?

Computing the median via sorting

- Computing the median of *n* numbers is easy: just sort them and pick A[⌊n/2⌋]

***O*(*n* log *n*)** time

Can we do better?

- We have hope, because sorting is doing far more work than we really need - we just want the middle element and don't care about the relative ordering of the rest of them

Divide-and-conquer approach

For some number x, split array A into two categories: elements <= x, and those > x.

For instance, if the array

A = [2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1]

is split on x = 5, the two subarrays generated are

A_L = [2, 5, 4, 1]

x = [5]

A_G = [36, 21, 8, 13, 11, 20]

Narrow the search for the n/2-th smallest element

A = [2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1]

A_L = [2, 5, 4, 1]

x = [5]

A_G = [36, 21, 8, 13, 11, 20]

- The search can instantly be narrowed down to one of these subarrays
- By checking ceiling(*n/2*) against the sizes of these subarrays, we can quickly determine which of them holds the desired element

How to choose a pivot *v*

- It should be picked quickly, and it should shrink the array substantially, the ideal situation being $|A_L|, |A_G| \approx \frac{1}{2} |A|$
- If we could always guarantee this situation, we would get a running time of:

$$T(n) = T(n/2) + O(n)$$

BTW, What is big-Oh of this recurrence? We will learn soon

To get this running time, we need to pick the median element!
But we do not know it - finding median is our ultimate goal! What to do?