# CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1]based on notes by Anna Bretscher and Albert Lai

# introduction

Today we begin studying a different way to look at complexity of algorithms.

So far we saw

- worst case analysis of individual operations
- amortised analysis over a sequence of operations

Today we add

- expected running time analysis
- need to review our probabilities!

# quicksort: idea

A divide-and-conquer algorithm.

1. if $|A| < 2$: stop
2. pick an element $x$, called "pivot"
3. partition $A$ into:
   - $L$: array of elements $\leq x$
   - $G$: array of elements $> x$

   ($\Theta(|A|)$ time; $|A| - 1$ comparisons against pivot)
4. recurse: sort $L$
5. recurse: sort $G$
6. concatenate sorted $L$, sorted $G$

We write $L$, $G$ as new arrays to show the idea.
In reality, "partition" is done in-place in $A$, so $L$ occupies the left side, $G$ occupies the right side, and there is no "concatenate" step.

# quicksort: algorithm

```
quicksort(A, p, r):
if p < r:
  q = partition(A, p, r)
  quicksort(A, p, q-1)
  quicksort(A, q+1, r)

partition(A, p, r):
x := A[r]   // pivot
i := p-1
for j in p, ..., r-1:
  if A[j] <= x:
    i := i + 1
    exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i + 1
```

# quicksort: example

Trace `quicksort` on the array

$$A = [2, 8, 7, 1, 3, 5, 6, 4]$$

## quicksort: correctness

The loop invariants of `partition`:

- $A[k] \leq x$ for $p \leq k \leq i$
- $A[k] > x$ for $i + 1 \leq k \leq j - 1$
- $A[k] = x$ for $k = r$

Proof: exercise.

## quicksort: complexity

- running time of `partition` is
- running time depends on partitioning
  -
- worst case when:
  -
- best case when:
  -
- average case:
  -

# randomised quicksort: idea

- remove "bias" in the input that may cause too many unbalanced partitions
- by selecting pivots at random
- we expect the split to be reasonably well balanced on average
- technique called <u>random sampling</u>

# randomised quicksort: algorithm

```
r-quicksort(A, p, r):
if p < r:
  q = r-partition(A, p, r)
  r-quicksort(A, p, q-1)
  r-quicksort(A, q+1, r)

r-partition(A, p, r):
exchange A[r] with A[random(p, r)]
x := A[r]      // pivot is now random from A[p..r]
i := p-1
for j in p, ..., r-1:
  if A[j] <= x:
    i := i + 1
    exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i + 1
```

# randomised quicksort: complexity

Idea:

- partition selects a pivot
- pivot never included in any future recursive calls
- at most ______ calls to partition over entire execution of quicksort
- partition is:
- focus on line 4 in for-loop
- can count number of times line 4 runs
- will give us bound on time spent in the for-loop over entire execution of quicksort

Formally:

-
-
-

## randomised quicksort: complexity

Notice:

- each pair of elements is compared at most once:
- elements are compared only to the pivot
- pivot is never used after call to `partition`

Let:

- elements of $A$: $\{z_1, z_2, ..., z_n\}$ where $z_i$ is the $i^{th}$ smallest element of $A$
- $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$ be set of elements between $z_i$ and $z_j$
- $X_{ij} = I\{z_i$ is compared to $z_j\}$ be indicator random variable

## stats flashback: indicator variable

An indicator random variable $X$ is

$$X_i = \begin{cases} 1 & \text{if an event } i \text{ occurs} \\ 0 & \text{if it does not} \end{cases}$$

## randomised quicksort: expected time

Total number of comparisons:

## randomised quicksort: expected time

When is $z_i$ compared to $z_j$?

In $z_i < \ldots < x < \ldots < z_j$,

- if $x$ becomes pivot first, $z_i$ is not compared to $z_j$
  ($z_i$ moved to left partition, $z_j$ moved to right partition)
- if $z_i$ becomes pivot first, $z_i$ is compared to $z_j$
- if $z_j$ becomes pivot first, $z_i$ is compared to $z_j$

# randomised quicksort: expected time

$\Pr(z_i \text{ is compared to } z_j) =$

## randomised quicksort: expected time

Total number of comparisons: