COM1009
Introduction to Algorithms and Data Structures

Topic 09: Quicksort and randomized algorithms

Essential Reading:
Chapter 7

Aims of this lecture

- To introduce the **QuickSort** algorithm: a popular algorithm which is fast in practice, despite a $\Theta(n^2)$ worst case time.
- To show an **average-case analysis**, revealing why QuickSort is fast in practice.
- To see another example of **divide-and-conquer.**
- To show how **randomness** can be used in the design of efficient algorithms.
- Glimpse into the **analysis of randomised algorithms**.

Idea behind QuickSort

- **Divide**:
  - Pick some element (called the **pivot**)
  - Move it to its final location in the sorted sequence such that **all smaller elements** are to its **left**, **larger** ones are to its **right.**
- **Conquer**:
  - Recursively sort the subarrays of smaller and larger elements
- **Combine**:
  - No work needed here – after the recursion the array is sorted.

QuickSort: The Algorithm

$\textsc{QuickSort}(A, p, r)$

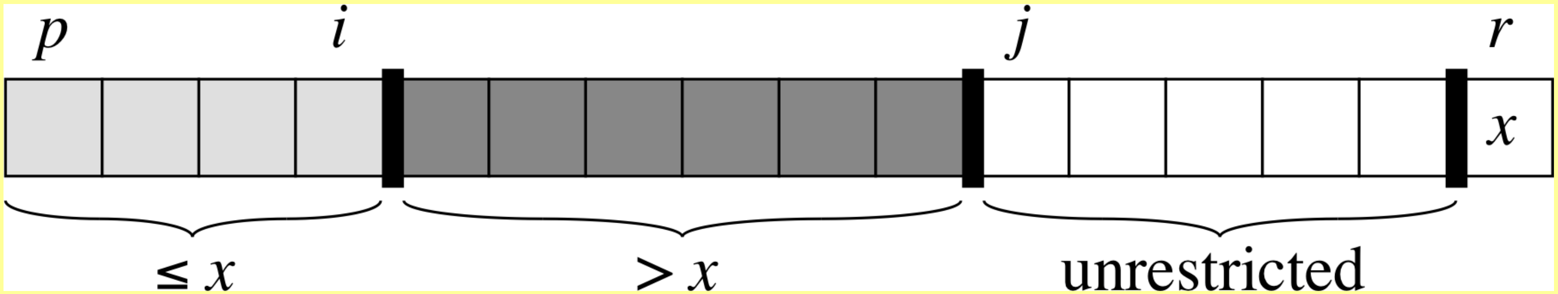
1: **if** $p < r$ **then**
2: $\quad q = \textsc{Partition}(A, p, r)$
3: $\quad \textsc{QuickSort}(A, p, q - 1)$
4: $\quad \textsc{QuickSort}(A, q + 1, r)$

Initial call: $\textsc{QuickSort}(A, 1, \text{A.length})$

Differences to MergeSort:

- Split the array at *q*, the position of the pivot in sorted array
  - We don't know *q* in advance, it is revealed by Partition
- No combine step at the end
- Partition plays a similar role to Merge

Partition in Pseudocode



PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Loop invariant:
See picture above –

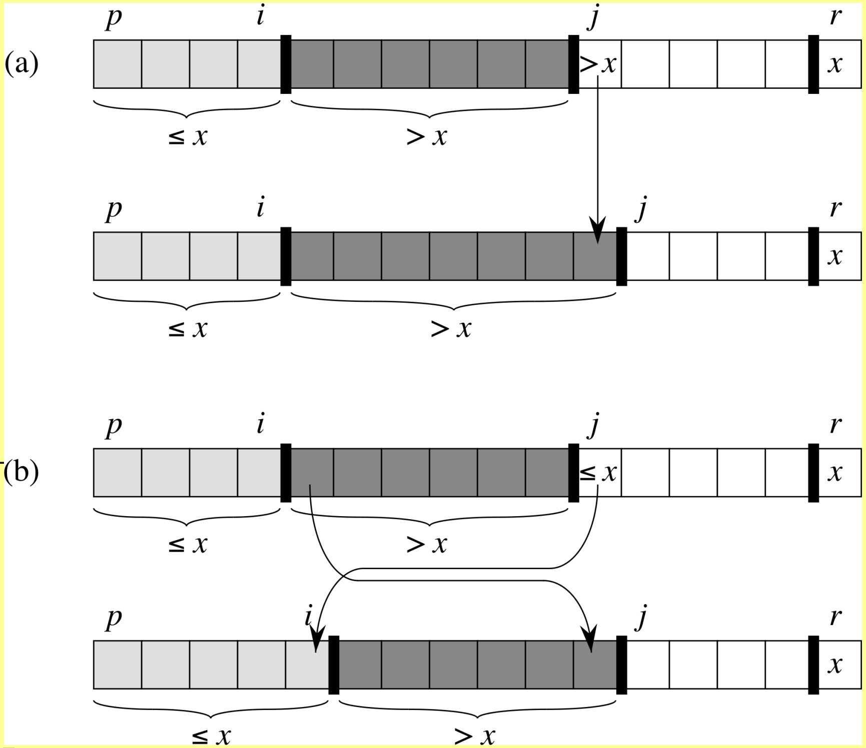
$A[p]..A[i] \leq x$
and
$A[i+1]..A[j-1] > x$.

Trivially true at initialisation.

Partition: Maintaining the loop invariant

Partition(A, p, r)

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



Termination:
After the last swap in line 7,

$$A[p]..A[i] \leq x < A[i+2]..A[r]$$

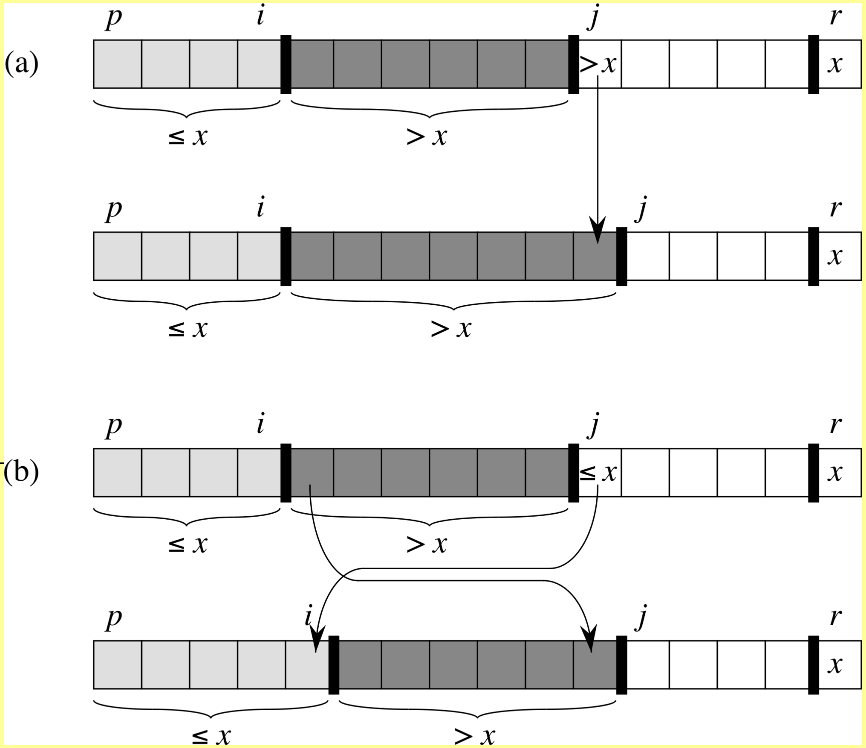
and Partition returns the position of x.

Exercise: Analyse the Runtime of Partition

Q: What is the runtime of Partition on a subarray of size n? Why?

Partition(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1 do
        if A[j] ≤ x then
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

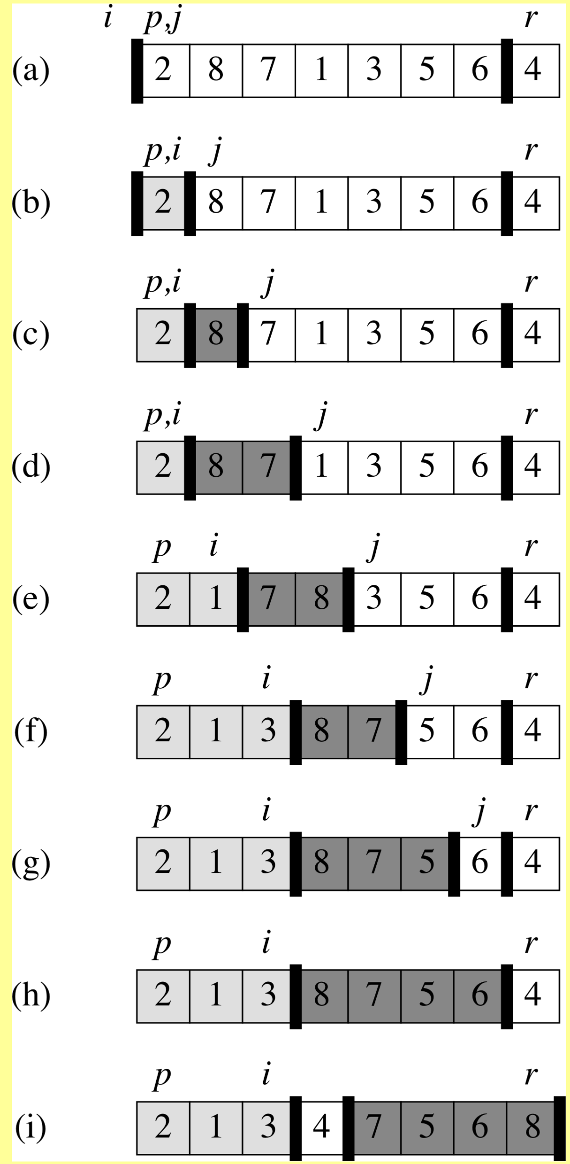


Partition: Example

- Figure 7.1 in the book

PARTITION(A, p, r)

```
1: x = A[r]
2: i = p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i = i + 1
6:         exchange A[i] with A[j]
7: exchange A[i + 1] with A[r]
8: return i + 1
```



(step (i) swaps pivot into place, line 7)

Worst-case and Best-case Partitionings

- The overall runtime depends on **how the array is partitioned** as that determines the sizes $q-1$ and $r-q$ of the subarray to be sorted recursively.
  - Recall that we don't know in advance where the pivot will end up.
- **Questions**:
  - What might be a **worst-case partitioning** for the runtime?
  - What might be a **best-case partitioning** for the runtime?

$\textsc{QuickSort}(A, p, r)$

1: **if** $p < r$ **then**
2: $\quad q = \textsc{Partition}(A, p, r)$
3: $\quad \textsc{QuickSort}(A, p, q-1)$
4: $\quad \textsc{QuickSort}(A, q+1, r)$

Worst-case Partitioning

- The worst case is attained when Partition always produces one subproblem with $n-1$ and one with 0 elements.
- This is the case, for example, when the array is already sorted.
- This leads to the following recurrence:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n). \end{aligned}$$

- Solving this gives $T(n) = \Theta(n^2)$.

Best-case Partitioning

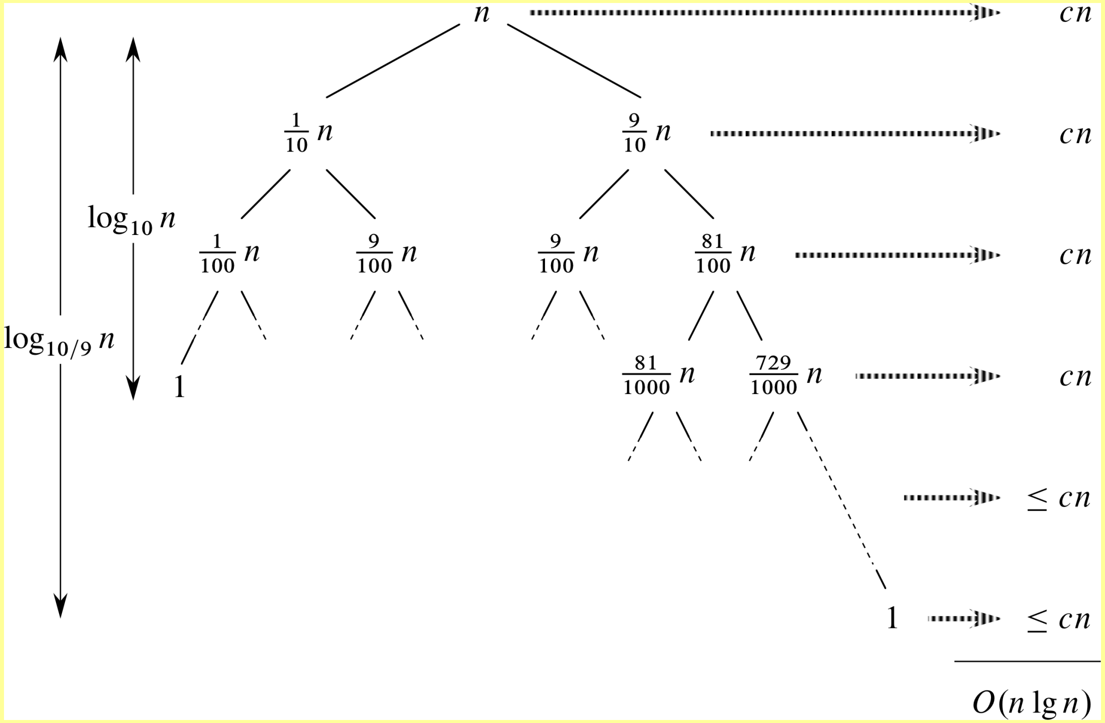
- Best case: split into two subproblems of sizes $\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil - 1$.
- Ignoring floors, ceilings, and -1 we get the recurrence:

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

- Deja vu?
- This is $\Theta(n \log n)$ from the analysis of MergeSort.
- True to the spirit of divide-and-conquer.

Towards an average case

- What if the split was always $\frac{9}{10} \cdot n$ and $\frac{1}{10} \cdot n$?
- Getting the recurrence $T(n) = T(9n/10) + T(n/10) + cn$



Average case (only sketched as Problem 7-3 in the book)

- Assume each split $q = 1, 2, \ldots, n$ was equally likely.
- This situation occurs when the input is chosen **uniformly at random** amongst all $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ possible orderings.
- Then

$$
\begin{aligned}
T(n) &= \frac{1}{n} \cdot \sum_{q=1}^{n} \left(T(q-1) + T(n-q) + \Theta(n)\right) \\
&= \frac{1}{n} \cdot \sum_{q=1}^{n} T(q-1) + \frac{1}{n} \cdot \sum_{q=1}^{n} T(n-q) + \frac{1}{n} \cdot \sum_{q=1}^{n} \Theta(n) \\
&= \frac{1}{n} \cdot \sum_{k=0}^{n-1} 2T(k) + \Theta(n)
\end{aligned}
$$

- Average over all problem sizes for 2 subproblems $+\Theta(n)$.
- Solving this recurrence gives a bound of $O(n \log n)$.

Improvements to QuickSort

- QuickSort is fast in practice because of small constants in the asymptotic running time.
- Improvements for handling **equal values** (exercise)
  - Partition into smaller, equal and larger elements
  - Only need to sort smaller and larger subarrays
- Choose the pivot as **median of 3** elements (or 5, 7, 9…)
  - Slightly faster in practice, but still quadratic worst case
- **Dual-Pivot QuickSort** by Vladimir Yaroslavskiy
  - Use two pivots instead of one and partition array in 3 areas
  - Used in Java 7

A Randomised Version of QuickSort

- Choosing the right pivot element can be tricky – we have no idea *a priori* which pivot elements are good.
- **Solution**: **leave it to chance!**

RANDOMISED-PARTITION(A, p, r)

```
1: i = RANDOM(p, r)
2: exchange A[r] with A[i]
3: return PARTITION(A, p, r)
```

"Random" picks pivot uniformly at random among all elements.

RANDOMISED-QUICKSORT(A, p, r)

```
1: if p < r then
2:     q = RANDOMISED-PARTITION(A, p, r)
3:     RANDOMISED-QUICKSORT(A, p, q-1)
4:     RANDOMISED-QUICKSORT(A, q+1, r)
```

▶Performance of Randomised-QuickSort

- Suppose all of the values are distinct.
  - What is a worst-case input for Randomised QuickSort?
  - **Answer**: **there is no worst case for Randomised QuickSort!**
- Reason: all inputs lead to the **same runtime behaviour**.
  - The i-th smallest element is chosen with uniform probability.
  - Every split is equally likely, regardless of the input.
  - The runtime is random, but the **random process (probability distribution) is the same** for every input.
- Randomness levels the playing field for all inputs.
  - No one can provide a worst-case input for Randomised-QS.

▶Runtime of Randomised Algorithms

- For randomised algorithms (in contrast to **deterministic algorithms**) we consider the **expected running time** $E(T(n))$.
- **Expectation** of a random variable X (see also: com1002)

$$\mathrm{E}(X) = \sum_x x \cdot \Pr(X = x)$$

- **Example**: for X = roll of fair 6-sided die, the expected result is

$$\mathrm{E}(X) = \sum_x x \cdot \Pr(X = x) = \sum_{x=1}^{6} x \cdot \frac{1}{6} = \frac{21}{6} = 3.5$$

- **Note**: The “expected” value need not be a possible value
  - “the average UK household has 2.4 people in it”

Linearity of Expectation

- Linearity of expectation:

$$\mathrm{E}(X_1 + X_2) = \mathrm{E}(X_1) + \mathrm{E}(X_2)$$

- Expected number of times 100 coin tosses come up heads:

$$\mathrm{E}(X_1 + \cdots + X_{100}) = \mathrm{E}(X_1) + \cdots + \mathrm{E}(X_{100}) = 100 \cdot \mathrm{Pr}(\text{heads})$$

- Note: for 0/1-variables the expectation boils down to probabilities.
  - **Example:** Write *tails* = 0, *heads* = 1 and take $X \in \{tails,\ heads\}$: Then the expected number of times a coin toss shows *heads* is Pr(*heads*) because

$$\mathrm{E}(X) = \sum_x x \cdot \mathrm{Pr}(X = x) = 0 \cdot \mathrm{Pr}(\text{tails}) + 1 \cdot \mathrm{Pr}(\text{heads}) = \mathrm{Pr}(\text{heads}).$$

Focus on the *number of comparisons*

For analysing sorting algorithms the number of comparisons is a useful quantity:

- For QuickSort and other algorithms it can be used as a proxy (= substitute) for the overall running time, e.g. when we looked at comparison sorts' worst case performance.
- Analysing the number of comparisons can be easier than analysing the number of elementary operations.
- comparisons can be costly if the keys to be compared are not numbers, but more complex objects (Strings, Arrays, etc.) – so they may be the things that matter most in many situations
- Algorithms making fewer comparisons might be preferable, even if the overall runtime is the same.

Number of Comparisons vs. Runtime

- Let $X = X(n)$ be the **number of comparisons** of elements made by QuickSort.
- Comparisons are elementary operations, hence $X(n) \leq T(n)$.
- For each comparison QuickSort only makes $O(1)$ other operations in the for loop.
- Other operations sum to $O(1)$.
- So $X(n) \leq T(n) = O(X(n))$ and thus $T(n) = \Theta(X(n))$

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1 do
        if A[j] ≤ x then
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Conclusion: it's enough to analyse the **number of comparisons** as a substitute for the runtime of QuickSort in the RAM model.

▶Expected Time for Randomised-QuickSort

- **Theorem**: the **expected number of comparisons** of Randomised-QuickSort is $O(n \log n)$ for every input where all elements are distinct.
- Proof outline:
  1. Show that here the expectation boils down to probabilities of comparing elements.
  2. Work out the probability of comparing elements.
  3. Putting points 1 and 2 together + some maths.
- Follows Section 7.4.2 in the book.

Proof:

- Suppose the elements are $z_1 < z_2 < \ldots < z_n$ (z_i is the *i*-th smallest element).
  - We've assumed they're all different so we can definitely number them in this way.
  - This is the order they will eventually be in once we've sorted them. We don't know what ordered they're in at the start.
- **Observation**: each pair of elements is compared at most once, because:
  - elements are only compared against the pivot
  - after Partition ends the pivot is never used again

▶Proof (1). Reduce problem to probabilities

- Let $X_{i,j}$ be the number of times z_i and z_j are compared:

$$X_{i,j} := \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$$

- Then the total number of comparisons is

$$X := \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

- Taking expectations on both sides and using linearity of expectations:

$$E(X) = E\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}\right) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{i,j}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j)$$

▶Proof (2). When do we compare z_i, z_j?

- When is z_i (*i*-th smallest) compared against z_j (*j*-th smallest)?
  - If pivot is $x < z_i$ or $z_j < x$ then the decision whether to compare z_i, z_j is **postponed** to a recursive call.
  - If pivot is $x = z_i$ or $x = z_j$ then z_i, z_j **are compared**.
  - If pivot is $z_i < x < z_j$ then z_i and z_j become separated and are **never compared**!
- A decision is only made if $z_i \leq x \leq z_j$.
  - There are $j - i + 1$ values in this range
  - Only 2 of these ($x = z_i$ or z_j) lead to z_i, z_j being compared.

▶Proof (3). When do we compare z_i, z_j?

- Just seen:
  - A decision is only made if $z_i \le x \le z_j$. There are $j - i + 1$ values in this range, out of which 2 lead to z_i, z_j being compared.
- As the pivot element is chosen uniformly at random,

$$\Pr(z_i \text{ is compared to } z_j) = \frac{2}{j - i + 1}$$

- Unexpected consequences?
  - similar numbers are more likely to be compared than dissimilar ones

▶Proof (4). Putting things together

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

- Substituting $k := j - i$ yields

$$E(X) = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \leq 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} = 2n \sum_{k=1}^{n} \frac{1}{k}$$

- The sum $\sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is called the **harmonic sum**
- ... which is known to be bounded: $\sum_{k=1}^{n} \frac{1}{k} \leq (\ln n) + 1$
- So we get

$$E(X) \leq 2n \sum_{k=1}^{n} \frac{1}{k} = O(n \log n)$$

... which confirms that

- When X is the runtime of randomised quicksort:

$$E(X) \leq 2n \sum_{k=1}^{n} \frac{1}{k} = O(n \log n)$$

- Compare this with deterministic quicksort:
  - Worst case: $\Theta(n^2)$
  - Best case: $O(n \log n)$

Random Input vs. Randomised Algorithm

- QuickSort is efficient if
  1. The input is random or
  2. The pivot element is chosen randomly
- We have no control over 1., but we can make 2. happen.
- **(Deterministic) QuickSort**
  - **Pro**: the runtime is deterministic for each input
  - **Con**: may be inefficient on some inputs
- **Randomised QuickSort**
  - **Pro**: same behaviour on all inputs
  - **Con**: runtime is random, running it twice gives different times

Other Applications of Randomisation

- **Random sampling**
  - Great for big data
  - Sample likely reflects properties of the set it is taken from
- **Symmetry breaking**
  - Vital for many distributed algorithms
- **Randomised search heuristics**
  - General-purpose optimisers, great for complex problems
    - Evolutionary Algorithms / Genetic Algorithms
    - Simulated Annealing
    - Swarm Intelligence
    - Artificial Immune Systems

Summary

- QuickSort has a bad worst-case runtime of $\Theta(n^2)$, but is fast on average.
  - Average-case performance on **random inputs** is $O(n \log n)$.
  - **Randomised QuickSort** sorts any input in **expected time** $O(n \log n)$.
  - Constants hidden in the asymptotic terms are small.
- QuickSort is used in modern programming languages
- **Randomness** can eliminate worst-case scenarios:
  - For randomised QuickSort all inputs are treated the same.
  - The running time is random and can be quantified by considering the **expected running time**: $O(n \log n)$.