Cosc 201
Algorithms and Data Structures
Lecture 8 (18/3/2026)
Random Algorithms

Brendan McCane
brendan.mccane@otago.ac.nz

QuickSort Recap

Algorithm:

1. Choose a pivot element
2. **Partition:** rearrange so that elements $\leq$ pivot are on the left, elements $>$ pivot are on the right
3. Recurse on left and right sub-arrays

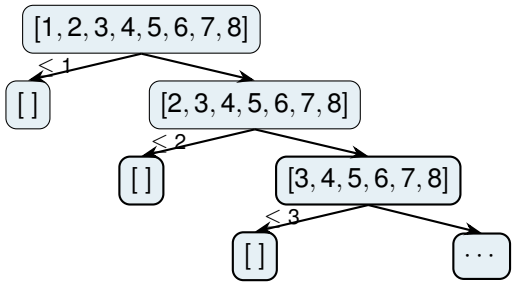
```
void quickSort(int[] a,
               int lo, int hi) {
  if (lo < hi) {
    int p = partition(a, lo, hi);
    quickSort(a, lo, p - 1);
    quickSort(a, p + 1, hi);
  }
}
```

Performance:

Best / Average:	$O(n \log n)$
Worst case:	$O(n^2)$

The worst case happens when the pivot is always the smallest (or largest) element – every partition is maximally unbalanced.

When Does Deterministic QuickSort Fail?



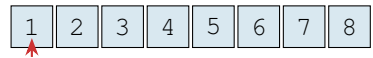
Pivot = first element on sorted input $\Rightarrow$ one side always empty.
Depth = n, work per level = $O(n) \Rightarrow O(n^2)$ total.

This is not an obscure edge case – sorted or nearly-sorted data is common in practice (e.g. appending to a log, re-sorting after a small update).

The Adversary Problem

Deterministic algorithms have a fixed strategy.
An **adversary** who knows your strategy can craft the worst-case input every time.

Example: If QuickSort always picks the first element as pivot, a sorted array triggers $O(n^2)$ behaviour.



Already sorted $\Rightarrow$ *worst-case!*

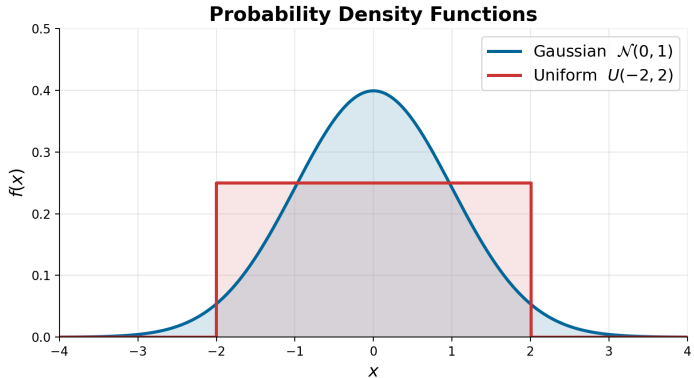
Key Insight

If the algorithm makes random choices, no single input can be "always bad." We defeat the adversary by **randomising our own strategy**.

Types of random numbers

There are many types of random numbers, but the most useful ones are:

- uniform random numbers
- Gaussian random numbers



Sampling random numbers in Java

`java.util.Random` – general purpose

```
Random rng = new Random();          // seeded from system clock
boolean b = rng.nextBoolean();      // equally likely true/false
int r = rng.nextInt(n);             // uniform in [0, n)
double d = rng.nextDouble();        // uniform in [0.0, 1.0)
double d = rng.nextGaussian();      // Gaussian (0.0, 1.0)
```

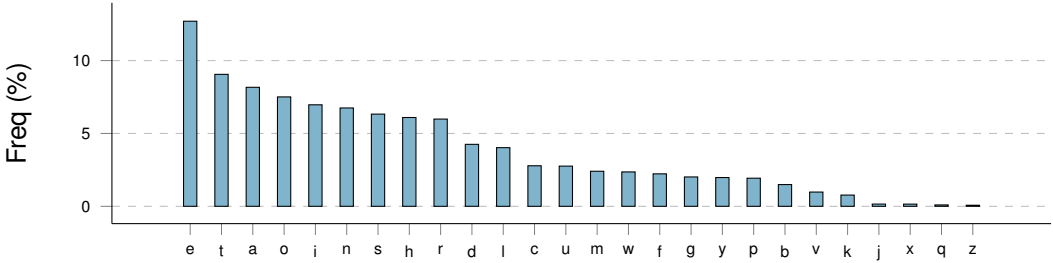
`Math.random()` – quick one-liner

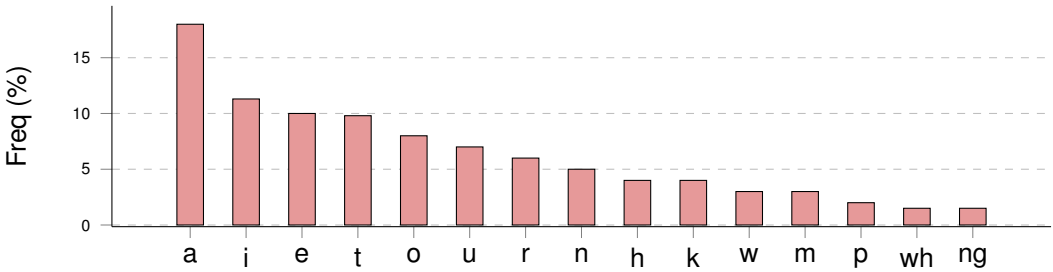
```
int r = (int)(Math.random() * n);    // uniform in [0, n)
```

All use **pseudorandom number generators** (PRNGs) – deterministic sequences that look random. Good enough for algorithm design; not for cryptography.

Sampling from a numerical distribution

- Let's say (just for fun) that we want to generate new words, based on the frequency of letters in English (or Māori).





Sampling from a numerical distributions

We have two basic options and a more advanced option:

1. Store whole dataset in an array, generate a random index uniformly and return that letter. Sampling is $\Theta(1)$, but space is $\Theta(n)$, where n is the total number of letters in the dataset.
2. Store the (normalised) frequencies in an array, generate a random number uniformly in $(0, 1)$, scan along the array until the sum of frequencies exceeds the random number and return the previous letter. Space is $\Theta(m)$ where m is the size of the alphabet ($m << n$), time is $O(m)$ as on average, we scan half of the array.
3. Choose a fixed array size (say 1000), and round the normalised frequencies to 1/1000, then sample uniformly from the array. Space is $\Theta(1)$ and time is $\Theta(1)$, but the sampling will be a bit inaccurate.

Las Vegas vs. Monte Carlo

Las Vegas Algorithms

- **Always correct**
- Running time is random
- We analyse expected time
- Example: Randomised QuickSort

Monte Carlo Algorithms

- Fixed (or bounded) running time
- **May be incorrect** with some probability
- We bound the error probability
- Example: Miller–Rabin primality test

Today we focus on a **Las Vegas** algorithm: randomised QuickSort gives the correct sorted output every time – only the number of comparisons varies.

Randomise the Pivot

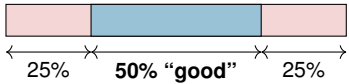
```
void randomQuickSort(int[] a, int lo, int hi) {
    if (lo < hi) {
        // Pick a uniformly random pivot
        int pivotIdx = lo +
            rng.nextInt(hi - lo + 1);
        swap(a, pivotIdx, hi);        // move pivot to end

        int p = partition(a, lo, hi); // standard Lomuto partitio
        randomQuickSort(a, lo, p - 1);
        randomQuickSort(a, p + 1, hi);
    }
}
```

Note: care is needed in recursive code not to call `new Random()` inside recursive methods – as this could create thousands of `Random` objects. Create the rng object once, then call it in a recursive method.

Why Does This Help? – Intuition

Think of the pivot as splitting the array into two “halves.”
A pivot is **“good enough”** if it lands in the **middle 50%** of the sorted order.

- Probability of a good pivot: **1/2**
- A good pivot gives at worst a 25%–75% split
- Each good split reduces the problem size by at least 25%



Half of all possible pivots fall in the good range.

Consequence

On any input, we expect to get a good pivot every other call. So the recursion depth is $O(\log n)$ in expectation, not $O(n)$.

Informal "Proof": Expected Comparisons

Claim: Randomised QuickSort makes $O(n \log n)$ expected comparisons.

Sketch of argument (by levels of recursion):

1. At each recursive call, we do $O(k)$ comparisons to partition a sub-array of size k.
2. The total work at each level of the recursion tree is at most $O(n)$ (every element compared at most once per level).
3. The key question: how many levels deep does the recursion go?

Bounding the depth:

- After each "good" pivot (probability $\geq 1/2$), the largest sub-problem shrinks to $\leq 3n/4$.
- After k good pivots the size is $\leq n \cdot (3/4)^k$.
- We need $(3/4)^k \leq 1/n$, so $k \geq \log_{4/3} n = O(\log n)$.
- On average, we need $\leq 2k$ total pivots to get k good ones (like flipping a fair coin).

$\Rightarrow$ Expected depth $= O(\log n)$, work per level $= O(n)$, total $= O(n \log n)$.

Proof by Induction: Setting Up the Recurrence

Let $T(n)$ = expected number of comparisons on an array of size n.

Base case: $T(0) = T(1) = 0$ (nothing to compare).

Recursive case: The random pivot is equally likely to be any rank $i \in \{1, \ldots, n\}$. If the pivot has rank i, the partition costs $n - 1$ comparisons and produces sub-arrays of size $i - 1$ and $n - i$.

$$T(n) \;=\; (n-1) \;+\; \frac{1}{n}\sum_{i=1}^{n}\big[T(i-1) + T(n-i)\big]$$

By symmetry ($i - 1$ and $n - i$ each range over $0, \ldots, n - 1$), the two sums are identical:

$$\boxed{T(n) \;=\; (n-1) \;+\; \frac{2}{n}\sum_{k=0}^{n-1} T(k)}$$

We now want to show $T(n) \leq 2\, n \ln n$ by **induction**.

Proof by Induction: $T(n) \leq 2\,n \ln n$

Inductive step: assume $T(k) \leq 2\,k \ln k$ for all $k < n$.

Substitute into the recurrence:

$$T(n) \;\leq\; (n-1) + \frac{2}{n}\sum_{k=1}^{n-1} 2\,k \ln k \;=\; (n-1) + \frac{4}{n}\sum_{k=1}^{n-1} k \ln k$$

Bound the sum with an integral (since $k \ln k$ is increasing):

$$\sum_{k=1}^{n-1} k \ln k \;\leq\; \int_1^n x \ln x \, dx \;=\; \frac{n^2 \ln n}{2} - \frac{n^2}{4} + \frac{1}{4}$$

Substituting back and simplifying:

$$T(n) \;\leq\; (n-1) + 2n \ln n - n + \frac{1}{n} \;=\; 2\,n \ln n - 1 + \frac{1}{n} \;\leq\; 2\,n \ln n \qquad \checkmark$$

Result: $T(n) \leq 2\,n \ln n = 2n \frac{\log_2 n}{\log_2 e} \approx 1.39\,n \log_2 n$ expected comparisons. $\square$

Convergence: How Reliable Is “Expected”?

“$O(n \log n)$ expected” – but could we get unlucky?

Yes, but it’s astronomically unlikely.

- The probability of picking $> c \log n$ bad pivots in a row is $(1/2)^{c \log n} = 1/n^c$.
- Think of it like coin flips: the chance of getting 40 tails in a row from a fair coin is about 10^{-12}.
- By a Markov bound: the probability that the running time exceeds t times the expectation is $\leq 1/t$.

Concentration

For large n, the actual running time is tightly concentrated around $\approx 1.39\, n \log_2 n$ comparisons.

In practice you will essentially always see $O(n \log n)$ behaviour.

This is why randomised QuickSort is the default sorting strategy in many standard libraries.

Comparing the Two Versions

	Deterministic QS	**Randomised QS**
Worst case	$O(n^2)$	$O(n^2)^*$
Expected	depends on input	$O(n \log n)$ **always**
Sorted input	$O(n^2)$	$O(n \log n)$ expected
In-place?	Yes	Yes
Extra code	–	1 line

*Technically possible but probability $\to 0$ as n grows.

The randomised version's worst case still exists in theory, but it doesn't depend on the input – it depends on the random choices, and bad random choices are vanishingly unlikely.

Where Else Does Randomness Help?

Data Structures

- **Hash tables** – universal hashing avoids adversarial collisions – covered in hashing lectures
- **Skip lists** – randomised balanced structure

Number Theory / Crypto

- **Miller–Rabin** primality test (Monte Carlo)
- Key generation in RSA, Diffie–Hellman

Graph Algorithms

- **Karger's min-cut** – contract random edges
- Randomised rounding for approximation

Verification

- **Freivald's algorithm** – check matrix multiplication in $O(n^2)$ instead of $O(n^3)$

The common theme: randomised algorithms are often used when exact algorithms are infeasible.

Key Takeaways

1. **Randomness defeats adversaries.** A deterministic strategy can be exploited; a random one cannot.
2. **Las Vegas algorithms** are always correct – only the running time is random.
3. **Randomised QuickSort** is a one-line change that eliminates input-dependent worst cases.
4. The expected number of comparisons is $\approx 1.39\, n \log_2 n$, and deviations are extremely unlikely.
5. Randomisation is a **fundamental technique**, not a hack – it appears throughout algorithms, data structures, and cryptography.

Further Reading

- **Textbook:** Kleinberg & Tardos, Algorithm Design, Chapter 13 (Randomized Algorithms).
- **Textbook:** Cormen et al. (CLRS), Chapter 7 (QuickSort) – includes the full expected-case analysis.
- **Advanced:** Motwani & Raghavan, Randomized Algorithms – the classic graduate text, Chapter 1 is very accessible.