LECTURE 3 OF 12

Elementary Sorting

Selection Sort, Insertion Sort, and Shellsort

Based on Robert Sedgewick & Kevin Wayne,
Algorithms, 4th Edition — Chapter 2.1

Estimated Duration: 4 hours (with break)

Learning Objectives

After completing this lecture, students will be able to:

1. **Explain** the sorting abstraction, the role of the `Comparable` interface, and the meaning of stability in sorting algorithms.
2. **Implement** Selection Sort and prove its running time is $\Theta(n^2)$ comparisons and $\Theta(n)$ exchanges in every case.
3. **Implement** Insertion Sort and analyze its best-case ($\Theta(n)$), average-case ($\Theta(n^2/4)$), and worst-case ($\Theta(n^2)$) behavior using the concept of *inversions*.
4. **Describe** the h-sorting idea behind Shellsort, implement it with Knuth's 3x+1 gap sequence, and explain why it achieves sub-quadratic performance in practice.
5. **Compare** elementary sorts along the axes of time complexity, space, stability, adaptivity, and number of data movements.
6. **Implement** the Knuth shuffle and explain why naive shuffling produces a biased permutation.
7. **Apply** the appropriate elementary sort to practical scenarios based on input characteristics.

1. Sorting Fundamentals ~30 MIN

1.1 Why Sorting Matters

Sorting is perhaps the most fundamental algorithmic problem in computer science. It appears directly in applications ranging from database indexing to graphics rendering, and it serves as a building block for many other algorithms:

- **Searching.** Binary search requires a sorted array, turning O(n) linear search into O(lg n).
- **Databases.** Indexes are maintained in sorted order for fast range queries and joins.
- **Duplicates.** Finding duplicates is trivial once data is sorted—just scan for adjacent equal elements.

- **Computational geometry.** Convex hull, closest-pair, and sweep-line algorithms rely on sorted coordinates.
- **Statistics.** Computing the median, percentiles, and order statistics.
- **Scheduling.** Priority-based scheduling sorts jobs by deadline or priority.

> *"Sorting is the most thoroughly studied problem in computer science. There are dozens of sorting algorithms, and this diversity reflects the rich variety of situations that arise in practice." — Sedgewick & Wayne*

1.2 The Sorting Abstraction

We study **comparison-based sorting**: the only way our algorithm learns about the data is by comparing two elements. This is formalized through Java's `Comparable` interface:

```
public class Example {

    public static void sort(Comparable[] a) {
        // sorting algorithm goes here
    }

    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    private static void exch(Comparable[] a, int i, int j) {
        Comparable t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

    private static boolean isSorted(Comparable[] a) {
        for (int i = 1; i < a.length; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }
}
```

KEY DESIGN PATTERN

By programming to the `Comparable` interface, our sorting code works for *any* data type that defines a natural order: `Integer`, `String`, `Double`, `Date`, or any user-defined type that implements `compareTo()`.

1.3 The `compareTo()` Contract

A type's `compareTo()` must define a **total order**:

Property	Requirement
Antisymmetry	If `v.compareTo(w) ≤ 0` and `w.compareTo(v) ≤ 0`, then `v == w`.
Transitivity	If `v ≤ w` and `w ≤ x`, then `v ≤ x`.
Totality	Either `v ≤ w` or `w ≤ v` (or both).

1.4 Stability

DEFINITION — STABLE SORT

A sorting algorithm is **stable** if it preserves the relative order of equal keys. That is, if two records have the same key, they appear in the output in the same order they appeared in the input.

Example. Suppose we sort students first by name, then by section number. A stable sort on section preserves the alphabetical ordering within each section. An unstable sort might scramble names within a section.

Name	Section
Anderson	2
Brown	3
Davis	3
Garcia	4
Harris	1
Jackson	3
Johnson	4
Jones	1
Martin	1
Moore	1
Robinson	2
Smith	4
Taylor	3
Thomas	2
Thompson	4
White	2
Williams	3
Wilson	4

After a **stable** sort by section, the names within each section remain alphabetical. After an **unstable** sort, names within a section could be in any order.

1.5 In-Place vs. Out-of-Place

> **DEFINITION — IN-PLACE SORTING**
>
> A sorting algorithm is **in-place** if it uses only a constant amount of extra memory beyond the input array (i.e., O(1) extra space). All three elementary sorts in this lecture are in-place.

1.6 Helper Functions: `less()` and `exch()`

Every sorting algorithm in this course is built from two primitive operations:

- `less(v, w)` — returns `true` if `v` is strictly less than `w`. This is our only way to inspect data.
- `exch(a, i, j)` — swaps `a[i]` and `a[j]`. This is our only way to rearrange data.

By counting the number of calls to `less()` (comparisons) and `exch()` (exchanges), we can precisely characterize algorithm performance.

2. Selection Sort ~45 MIN

2.1 Algorithm Description

Idea: On each pass through the array, *select* the smallest remaining element and put it in its final position.

1. Find the index `min` of the smallest element in `a[i..n-1]`.
2. Exchange `a[i]` with `a[min]`.
3. Increment `i` and repeat.

Invariant: After iteration `i`, the elements `a[0..i]` are in their final sorted positions, and no element in `a[i+1..n-1]` is smaller than `a[i]`.

> **INTUITION**
>
> Think of a coach selecting players for a lineup. In each round, the coach scans all remaining players, picks the best one, and places them next in line. The coach always looks at *every* remaining player, regardless of the current state of the lineup.

2.2 Implementation

```
public class Selection {

    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;                          // index of smallest in a[i..n-1]
            for (int j = i + 1; j < n; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);                      // put smallest in position i
        }
    }
}
```

2.3 Step-by-Step Trace

Sorting the array `[7, 10, 5, 3, 8, 4, 2, 9, 6]` (n = 9):

i	min	0	1	2	3	4	5	6	7	8
init	**-**	7	10	5	3	8	4	2	9	6
0	**6**	**2**	10	5	3	8	4	**7**	9	6
1	**3**	2	**3**	5	**10**	8	4	7	9	6
2	**5**	2	3	**4**	10	8	**5**	7	9	6
3	**5**	2	3	4	**5**	8	**10**	7	9	6
4	**8**	2	3	4	5	**6**	10	7	9	**8**
5	**6**	2	3	4	5	6	**7**	**10**	9	8
6	**8**	2	3	4	5	6	7	**8**	9	**10**
7	**7**	2	3	4	5	6	7	8	9	10
8	**8**	2	3	4	5	6	7	8	9	10

Green cells are in their final sorted position. Pink cells were just swapped.

2.4 Analysis

> **PROPOSITION**
>
> Selection sort uses **(n-1) + (n-2) + ... + 1 + 0 = n(n-1)/2 ~ n²/2** comparisons and exactly **n** exchanges to sort an array of **n** elements.

Proof sketch.

- **Comparisons:** In iteration `i`, the inner loop performs exactly `n - 1 - i` comparisons. Summing: $\sum_{i=0}^{n-1}$ (n - 1 - i) = n(n-1)/2.
- **Exchanges:** Each iteration of the outer loop performs exactly one exchange. There are `n` iterations, so exactly `n` exchanges.

Properties of Selection Sort

Property	Value
Best-case comparisons	$\sim n^2/2$
Average-case comparisons	$\sim n^2/2$
Worst-case comparisons	$\sim n^2/2$
Exchanges (all cases)	n
In-place?	Yes
Stable?	**No**
Adaptive?	**No** — running time is independent of input order

> **IMPORTANT**
>
> Selection sort's running time is **insensitive to input**. It takes just as long to sort an already-sorted array as a randomly-ordered one. This is its greatest weakness: it cannot exploit existing order.

2.5 Why Selection Sort Is Not Stable

Consider the array with duplicate keys: `[B1, B2, A]`. On the first pass, `A` is the minimum. We exchange `a[0]` with `a[2]`, producing `[A, B2, B1]`. The two `B` values have been reordered, violating stability.

2.6 When Selection Sort Is Useful

Despite its simplicity, selection sort has one notable advantage: it performs the **minimum number of exchanges** (exactly n). When swapping elements is very expensive (e.g., records with large satellite data), selection sort can be the practical choice.

3. Insertion Sort ~60 MIN

3.1 Algorithm Description

Idea: Build the sorted portion of the array one element at a time. For each new element, *insert* it into its correct position among the already-sorted elements, shifting larger elements to the right.

> **CARD SORTING ANALOGY**
>
> Insertion sort works exactly the way most people sort a hand of playing cards. You pick up cards one at a time and insert each card into its proper place among those already in your hand, sliding the other cards over to make room.

Invariant: After iteration `i`, the elements `a[0..i]` are in sorted order (but not necessarily in their final positions—later elements may need to be inserted among them).

3.2 Implementation

```
public class Insertion {

    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 1; i < n; i++) {
            // Insert a[i] into sorted sequence a[0..i-1]
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
        }
    }
}
```

Notice the inner loop walks backward from position `i`, exchanging adjacent elements as long as the current element is smaller than its predecessor. The loop terminates early if `a[j] >= a[j-1]`, which is the key to insertion sort's adaptivity.

3.3 Step-by-Step Trace

Sorting the array `[7, 10, 5, 3, 8, 4, 2, 9, 6]` (n = 9):

i	0	1	2	3	4	5	6	7	8	Exchanges
init	7	10	5	3	8	4	2	9	6	—
1	7	**10**	5	3	8	4	2	9	6	**0**
2	**5**	7	10	3	8	4	2	9	6	**2**
3	**3**	5	7	10	8	4	2	9	6	**3**
4	3	5	7	**8**	10	4	2	9	6	**1**
5	3	**4**	5	7	8	10	2	9	6	**4**
6	**2**	3	4	5	7	8	10	9	6	**6**
7	2	3	4	5	7	8	**9**	10	6	**1**
8	2	3	4	5	**6**	7	8	9	10	**4**

Green cells are the sorted prefix. Purple highlights show the final resting position of the element being inserted. Total exchanges: 21.

3.4 Analysis

Inversions

> **DEFINITION — INVERSION**
>
> An **inversion** is a pair of elements (a[i], a[j]) with i < j but a[i] > a[j]. Informally, an inversion is a pair of elements that are "out of order."

Example. In the array `[3, 1, 4, 1, 5]`, the inversions are: (3,1), (3,1), (4,1), giving 3 inversions total.

> **PROPOSITION**
>
> The number of exchanges performed by insertion sort is exactly equal to the number of inversions in the input. The number of comparisons is the number of inversions plus (n - 1).

Proof sketch. Each exchange removes exactly one inversion (it swaps two adjacent out-of-order elements). When no inversions remain, the array is sorted. Each pass starting at position `i` makes at least one comparison (even if no exchange is needed), and each exchange requires exactly one comparison.

Running-Time Summary

Case	Comparisons	Exchanges	When?
Best	n - 1	0	Already sorted (0 inversions)
Worst	$\sim n^2/2$	$\sim n^2/2$	Reverse sorted (max inversions = n(n-1)/2)
Average	$\sim n^2/4$	$\sim n^2/4$	Random order (expected inversions ~ n(n-1)/4)

WHY $\sim N^2/4$ ON AVERAGE?

For a random permutation, each pair (i, j) with i < j is an inversion with probability 1/2. There are C(n,2) = n(n-1)/2 such pairs, so the expected number of inversions is $n(n-1)/4 \sim n^2/4$.

3.5 Partially Sorted Arrays

DEFINITION — PARTIALLY SORTED

An array is **partially sorted** if the number of inversions is O(n). That is, no element is very far from its sorted position.

Examples of partially sorted arrays:

- A sorted array with a few elements appended at the end.
- An array where each element is at most k positions from its final position (k constant).
- An array formed by interleaving two sorted subarrays.
- An array with only a constant number of elements out of place.

PROPOSITION

Insertion sort runs in **O(n)** time on partially sorted arrays (arrays with O(n) inversions). Specifically, the running time is O(n + I), where I is the number of inversions.

This adaptivity property makes insertion sort the algorithm of choice for:

- Small arrays ($n \leq 15$ or so)—used as a base case in mergesort and quicksort.

- Nearly-sorted data—common in practice (e.g., appending to a sorted log).
- As a finishing pass after a gap-based presort (this is exactly what Shellsort does).

3.6 Stability of Insertion Sort

> **INSERTION SORT IS STABLE**
>
> Because insertion sort only exchanges *adjacent* elements, and it does so only when `a[j] < a[j-1]` (strictly less), equal elements are never exchanged. Their original relative order is preserved.

3.7 Optimized Variant: Half-Exchanges (Shifting)

Instead of repeated exchanges, we can shift elements right and place the new element once:

```
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        Comparable key = a[i];
        int j = i;
        while (j > 0 && less(key, a[j-1])) {
            a[j] = a[j-1];          // shift right (half-exchange)
            j--;
        }
        a[j] = key;                  // place element in its position
    }
}
```

This reduces the number of array accesses: each "exchange" now takes 1 array write instead of 3 (a temp + two writes). Same asymptotic complexity, but roughly 2x faster in practice due to fewer memory operations.

4. Shellsort ~60 MIN

4.1 Motivation

Insertion sort is slow for large unordered arrays because it only exchanges *adjacent* elements. An element far from its destination requires many exchanges to get there. If the smallest element happens to be at the far end of the array, it must be exchanged all the way across—one position at a time.

Shellsort (invented by Donald Shell in 1959) addresses this by allowing exchanges of elements that are far apart. The key insight: move elements large distances early, then clean up with insertion sort at the end.

4.2 h-Sorting

> **DEFINITION — H-SORTED ARRAY**
>
> An array is **h-sorted** if every h-th element (starting anywhere) forms a sorted subsequence. Equivalently, `a[i] ≤ a[i+h]` for all valid `i`.

An h-sorted array consists of `h` independent sorted subsequences, interleaved together:

Example: a 4-sorted array

```
Index:   0   1   2   3   4   5   6   7   8   9   10  11
Value:   2   7   3   5   4   8   6   9   8   10  9   12

Subseq 0: a[0], a[4], a[8]   =  2, 4, 8     (sorted)
Subseq 1: a[1], a[5], a[9]   =  7, 8, 10    (sorted)
Subseq 2: a[2], a[6], a[10]  =  3, 6, 9     (sorted)
Subseq 3: a[3], a[7], a[11]  =  5, 9, 12    (sorted)
```

> **KEY PROPERTY**
>
> An h-sorted array that is then k-sorted **remains h-sorted**. This is a non-trivial fact proved by Knuth: sorting a subsequence cannot unsort other interleaved subsequences.

4.3 The Shellsort Algorithm

Shellsort is a generalization of insertion sort. It h-sorts the array for a decreasing sequence of gap values h, ending with h = 1 (ordinary insertion sort). Because the array is partially sorted by the time h = 1, the final insertion sort pass is efficient.

Algorithm:

1. Choose a sequence of gap values $h_t > h_{t-1} > \ldots > h_1 = 1$.
2. For each gap h (from largest to smallest), h-sort the array using insertion sort with stride h.

4.4 Knuth's Gap Sequence (3x + 1)

The choice of gap sequence profoundly affects performance. Knuth proposed the simple sequence:

1, 4, 13, 40, 121, 364, 1093, 3280, ...

Each term is 3 times the previous plus 1: $h_k = 3h_{k-1} + 1$, starting with $h_1 = 1$.

We start with the largest h that is less than n/3, then decrease by dividing by 3:

4.5 Implementation

```
public class Shell {

    public static void sort(Comparable[] a) {
        int n = a.length;
        int h = 1;

        // Compute the largest h in the 3x+1 sequence less than n/3
        while (h < n/3) h = 3*h + 1;   // 1, 4, 13, 40, 121, 364, ...

        while (h >= 1) {
            // h-sort the array (insertion sort with stride h)
            for (int i = h; i < n; i++) {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h / 3;               // move to next smaller gap
        }
    }
}
```

> **OBSERVATION**
>
> The code structure is identical to insertion sort, except that the inner loop decrements by `h` instead of `1`, and compares elements `h` positions apart. When `h = 1`, it is exactly insertion sort.

4.6 Step-by-Step Trace

Sorting `[S, H, E, L, L, S, O, R, T, E, X, A, M, P, L, E]` (n = 16).

The 3x+1 sequence for n = 16: starting h = 1, then 4, then 13. Since 13 < 16/3 is false (16/3 ~ 5.3), we start with h = 13.

Actually, let us recalculate: h = 1 → 4 → 13. Since 13 < 16/3 = 5.33 is false, we stop at h = 4. So the gaps are **4, 1**.

Wait: h starts at 1. Is 1 < 16/3 ≈ 5.33? Yes, so h = 4. Is 4 < 5.33? Yes, so h = 13. Is 13 < 5.33? No. So we start with h = 13.

Gap sequence used: 13, 4, 1

Pass 1: h = 13

Compare elements 13 positions apart. With n = 16 and h = 13, only 3 pairs to consider: (0,13), (1,14), (2,15).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
before	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	**P**	H	E	L	L	S	O	R	T	E	X	A	M	**S**	L	E

S > P, so they swap. H ≤ L, no swap. E ≤ E, no swap.

Pass 2: h = 4

Now we h-sort with gap 4. This produces 4 interleaved subsequences, each sorted by insertion sort:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
before	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R

The four subsequences after 4-sorting: {L,M,P,T}, {E,H,S,S}, {E,L,O,X}, {A,E,L,R} — each sorted.

Pass 3: h = 1 (Final insertion sort)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
before	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Because the array is already 4-sorted, the final 1-sort pass requires relatively few exchanges. No element is more than 3 positions from its final location.

4.7 Analysis

> **PROPOSITION (KNUTH, 1973)**
>
> The number of comparisons for Shellsort with the 3x+1 gap sequence is $\mathbf{O(n^{3/2})}$.

The exact complexity of Shellsort is one of the great open problems in the analysis of algorithms. The actual bound depends critically on the gap sequence:

Gap Sequence	Author	Worst-Case Comparisons
n/2, n/4, ..., 1	Shell (1959)	$\Theta(n^2)$
1, 3, 7, 15, ..., 2^k-1	Hibbard (1963)	$O(n^{3/2})$
1, 4, 13, 40, ..., $(3^k-1)/2$	Knuth (1973)	$O(n^{3/2})$
1, 8, 23, 77, 281, ...	Sedgewick (1986)	$O(n^{4/3})$
1, 5, 19, 41, 109, ...	Ciura (2001)	Unknown (best empirical)

OPEN PROBLEM

No one has been able to prove a tight bound better than $O(n \log^2 n)$ for any gap sequence. The exact asymptotic complexity of Shellsort remains unknown despite over 60 years of research.

4.8 Why Does Shellsort Work?

The power of Shellsort comes from a subtle interaction between gap sizes:

1. **Large gaps move elements far.** Early passes with large h values move elements close to their final positions with few exchanges.
2. **The h-sort property is preserved.** After an h-sort followed by a k-sort, the array is still h-sorted. This means each successive pass refines the order without destroying previous work.
3. **Small gaps finish efficiently.** By the time h = 1, the array is already partially sorted, so insertion sort runs quickly (close to O(n)).

4.9 Practical Performance

With Knuth's 3x+1 sequence, Shellsort handles arrays with tens of thousands of elements in reasonable time. Some empirical benchmarks:

n	Insertion Sort	Shellsort (3x+1)	Speedup
1,000	0.05s	0.002s	~25x
10,000	4.5s	0.04s	~112x
100,000	~450s	0.7s	~640x
1,000,000	(infeasible)	12s	—

Approximate wall-clock times for random integer arrays on a modern machine.

4.10 Properties of Shellsort

Property	Value
Best case	O(n log n) — with good gap sequences
Average case	Depends on gap sequence; ~$O(n^{5/4})$ empirically with 3x+1
Worst case (3x+1)	$O(n^{3/2})$
Space	O(1) — in-place
Stable?	**No** — long-distance exchanges can reorder equal keys
Adaptive?	Yes — benefits from partial order (final h=1 pass is fast)

WHEN TO USE SHELLSORT

Shellsort is an excellent choice when you need a simple, compact, sub-quadratic sorting algorithm with no extra memory. It is often used in embedded systems, hardware sort implementations, and as the sort in `uClibc`. Its code is tiny—only a few more lines than insertion sort—yet its performance is dramatically better on medium-sized arrays.

5. Comparing Elementary Sorts ~30 MIN

5.1 Summary Comparison Table

Property	Selection Sort	Insertion Sort	Shellsort (3x+1)
Best case	$\sim n^2/2$	n - 1	O(n log n) *
Average case	$\sim n^2/2$	$\sim n^2/4$	$\sim O(n^{5/4})$ *
Worst case	$\sim n^2/2$	$\sim n^2/2$	$O(n^{3/2})$
Exchanges (worst)	n	$\sim n^2/2$	$O(n^{3/2})$
Extra space	O(1)	O(1)	O(1)
Stable?	No	**Yes**	No
Adaptive?	No	**Yes**	Yes
Online?	No	**Yes**	No

** Empirical estimates; exact complexity depends on gap sequence and is not fully characterized.*

5.2 When to Use Which Sort

Scenario	Best Choice	Reason
Tiny arrays (n ≤ 15)	Insertion sort	Low overhead, fast on small inputs
Nearly sorted data	Insertion sort	Adaptive — O(n + inversions)
Expensive swaps, cheap comparisons	Selection sort	Minimum number of exchanges (exactly n)
Medium arrays, no extra memory	Shellsort	Sub-quadratic with O(1) space
Streaming / online data	Insertion sort	Can sort as data arrives
Need stability	Insertion sort	Only stable elementary sort
Large arrays	(Mergesort / Quicksort)	Elementary sorts too slow; covered in Lectures 4-5

5.3 Visualizing Sorting Algorithms

A useful way to understand sorting behavior is to visualize array accesses. For each algorithm, we can plot array index (y-axis) against time/operations (x-axis), marking each comparison and exchange:

VISUAL PATTERNS

- **Selection sort** produces a triangular pattern: the scan length decreases by 1 on each pass, with a single exchange dot at the end of each scan.
- **Insertion sort** produces a similar triangle but mirrored: most work is in the upper-left (early elements into a small sorted prefix), with the amount of backtracking varying by input. Sorted input produces a single diagonal line.
- **Shellsort** produces striking banded patterns: coarse bands for large h, finer bands for smaller h, and a nearly clean diagonal for h = 1.

5.4 Preview: Lower Bounds

> **THEOREM (DECISION-TREE LOWER BOUND)**
>
> Any comparison-based sorting algorithm must make at least **⌈lg(n!)⌉ ~ n lg n** comparisons in the worst case to sort n elements.

This means all three elementary sorts are *asymptotically suboptimal* in the worst case (they all exceed n lg n comparisons). In Lectures 4 and 5, we will study mergesort and quicksort, which achieve the O(n lg n) lower bound.

However, this lower bound only applies in the worst case. Insertion sort beats n lg n on nearly-sorted inputs, which is perfectly consistent with the lower bound.

6. Shuffling ~15 MIN

6.1 The Knuth Shuffle

Shuffling is the "inverse" of sorting: we want to produce a *uniformly random permutation* of the input array. The standard algorithm is the **Knuth shuffle** (also called the Fisher-Yates shuffle):

```
public static void shuffle(Object[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        // Choose a random index r in [i, n-1]
        int r = i + (int) (Math.random() * (n - i));
        exch(a, i, r);
    }
}
```

Invariant: After iteration `i`, `a[0..i]` is a uniformly random permutation of those `i+1` elements.

> **PROPOSITION**
>
> The Knuth shuffle produces each of the n! permutations with equal probability 1/n!, in O(n) time and O(1) extra space.

Proof sketch. In the first iteration, any of the n elements is equally likely to land in position 0. In the second, any of the remaining n-1 elements is equally likely for position 1. And so on. The total number of equally-likely outcomes is n × (n-1) × ... × 1 = n!, matching the number of permutations.

6.2 Why Naive Shuffling Is Biased

A common (incorrect) shuffle picks a random position from `[0, n-1]` on every iteration:

```
// BROKEN: biased shuffle
for (int i = 0; i < n; i++) {
    int r = (int) (Math.random() * n);   // r in [0, n-1] — WRONG!
    exch(a, i, r);
}
```

> **WHY THIS IS WRONG**
>
> This generates n^n equally-likely execution paths, but there are only n! permutations. Since n^n is generally not divisible by n!, some permutations must be more likely than others. For n = 3: $n^n = 27$, but n! = 6, and 27 is not divisible by 6.

Concrete example for n = 3: Starting with `[1, 2, 3]`, the naive shuffle produces 27 equally-likely sequences of random choices. Each permutation should appear 27/6 = 4.5 times, which is impossible. Some permutations appear 4 times, others 5 times—a non-uniform distribution.

6.3 Practical Importance

Shuffling correctly matters in practice:

- **Online poker.** In 1999, a security researcher discovered that a popular online poker site used a broken shuffle, allowing players to predict the order of the deck.
- **Randomized algorithms.** Quicksort's expected performance relies on random shuffling of the input.
- **Sampling.** Selecting a random subset requires unbiased permutations.

7. Summary and Key Takeaways

7.1 The Big Picture

1. **Selection sort** is simple and makes the minimum number of exchanges (n), but always does $\sim n^2/2$ comparisons regardless of input. It is not adaptive and not stable.
2. **Insertion sort** is the most practical elementary sort: it is adaptive (O(n) on nearly-sorted data), stable, and runs well on small arrays. It is used as a subroutine in more advanced sorts.
3. **Shellsort** is a remarkable extension of insertion sort. By allowing long-distance exchanges through decreasing gap sequences, it achieves sub-quadratic performance with only a few extra lines of code. Its exact complexity remains an open research problem.
4. **The Knuth shuffle** is the correct way to generate a uniformly random permutation: pick a random index from the remaining elements, not from the entire array.

7.2 Comprehensive Comparison

	Selection Sort	Insertion Sort	Shellsort
Time (best)	$\Theta(n^2)$	$\Theta(n)$	O(n log n) *
Time (avg)	$\Theta(n^2)$	$\Theta(n^2)$	subquadratic *
Time (worst)	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^{3/2})$
Space	O(1)	O(1)	O(1)
Stable	No	Yes	No
Adaptive	No	Yes	Yes
Comparisons insensitive to input	Yes	No	No
Min exchanges	Yes (n)	No	No
Code complexity	Simple	Simple	Simple

7.3 Looking Ahead

- **Lecture 4: Mergesort.** An O(n lg n) algorithm that uses O(n) extra space. Divide-and-conquer.
- **Lecture 5: Quicksort.** An O(n lg n) expected-time algorithm that is in-place. Randomization via the Knuth shuffle.
- Both mergesort and quicksort use **insertion sort** as a base case for small subarrays.

8. Practice Problems

Problem 1: Trace Selection Sort

Show the state of the array `[6, 4, 2, 8, 1, 3]` after each pass of selection sort. Count the total number of comparisons and exchanges.

<details>
<summary>Solution</summary>
</details>

Problem 2: Inversions

Count the number of inversions in the array `[5, 3, 1, 4, 2]`. How many comparisons and exchanges will insertion sort perform on this array?

<details>
<summary>Solution</summary>
</details>

Problem 3: Best and Worst Cases

For an array of 6 elements:

a. What input produces the *best* case for insertion sort? How many comparisons?

b. What input produces the *worst* case for insertion sort? How many comparisons?

c. Does the same best/worst-case input also produce the best/worst case for selection sort?

<details>
<summary>Solution</summary>
</details>

Problem 4: Stability

Suppose you have the following array of (name, age) pairs, sorted by name:

```
(Alice, 25), (Bob, 22), (Carol, 25), (Dave, 22)
```

You want to sort by age. After sorting by age:

a. What order do you get with a *stable* sort?

b. Give one possible order with an *unstable* sort.

▶ **Solution**

Problem 5: Shellsort Gap Sequence

For an array of n = 100 elements, list the gap values used by the 3x+1 sequence. How many passes does Shellsort make?

▶ **Solution**

Problem 6: Partially Sorted Array

An array of 1000 elements is sorted except for 10 elements that have been randomly placed. Approximately how many comparisons will insertion sort use? What about selection sort?

▶ **Solution**

Problem 7: Knuth Shuffle Correctness

Trace the Knuth shuffle on `[A, B, C, D]` assuming the random numbers chosen are: r_0=2, r_1=3, r_2=2, r_3=3. What is the final permutation?

▶ **Solution**

Problem 8: Algorithm Identification

You observe the following about a sorting algorithm's behavior on a random array of 10,000 elements:

- It makes approximately 50,000,000 comparisons.
- It makes exactly 10,000 exchanges.
- The same number of comparisons occurs regardless of input order.

Which sorting algorithm is it? Justify your answer.

▶ **Solution**

Appendix A: Complete Java Implementations

A.1 Selection Sort

```
public class Selection {
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i + 1; j < n; j++)
                if (less(a[j], a[min])) min = j;
            exch(a, i, min);
        }
    }

    private static boolean less(Comparable v, Comparable w)
    {  return v.compareTo(w) < 0;  }

    private static void exch(Comparable[] a, int i, int j)
    {  Comparable t = a[i]; a[i] = a[j]; a[j] = t;  }
}
```

A.2 Insertion Sort

```
public class Insertion {
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 1; i < n; i++)
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
    }

    private static boolean less(Comparable v, Comparable w)
    {  return v.compareTo(w) < 0;  }

    private static void exch(Comparable[] a, int i, int j)
    {  Comparable t = a[i]; a[i] = a[j]; a[j] = t;  }
}
```

A.3 Shellsort

```
public class Shell {
    public static void sort(Comparable[] a) {
        int n = a.length;
        int h = 1;
        while (h < n/3) h = 3*h + 1;

        while (h >= 1) {
            for (int i = h; i < n; i++)
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            h = h/3;
        }
    }

    private static boolean less(Comparable v, Comparable w)
    {  return v.compareTo(w) < 0;  }

    private static void exch(Comparable[] a, int i, int j)
    {  Comparable t = a[i]; a[i] = a[j]; a[j] = t;  }
}
```

A.4 Knuth Shuffle

```
public class KnuthShuffle {
    public static void shuffle(Object[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int r = i + (int) (Math.random() * (n - i));
            Object t = a[i];
            a[i] = a[r];
            a[r] = t;
        }
    }
}
```

Appendix B: Glossary

Term	Definition
Adaptive	A sorting algorithm whose running time depends on the input order, not just input size.
Comparable	Java interface that defines a natural ordering via `compareTo()`.
Exchange	Swapping two elements in an array.
Gap sequence	The decreasing sequence of h-values used in Shellsort.
h-sorted	An array in which every h-th element forms a sorted subsequence.
In-place	Using O(1) extra memory beyond the input.
Inversion	A pair (i, j) with i < j and a[i] > a[j].
Online algorithm	An algorithm that can process input as it arrives, one element at a time.
Partially sorted	An array with O(n) inversions.
Stable sort	A sort that preserves the relative order of equal keys.
Total order	A relation that is antisymmetric, transitive, and total.
