

The Engineer's Workbench

A workbench is where you make your own tools. Every language hands you a list, a dictionary and a sort, and you can call them for a whole career without knowing what they cost or why. This supplement builds them from nothing: the growable array, the hash table, the heap, the balanced tree and the sorts, then the probabilistic structures that trade a little accuracy for a great deal of memory. It ends where efficient answers stop existing at all.

CHAPTERS

8

CODE EXAMPLES

8

EXERCISES

24

STAGE BUDGET

250 h

Contents

How to use this volume	3
Part I · Build the containers	5
Chapter 01 · Dynamic arrays and ring buffers	6
Chapter 02 · Hash tables from scratch	12
Chapter 03 · Heaps, priority queues and A*	18
Chapter 04 · Balanced search trees	24
Part II · Sorting	30
Chapter 05 · Comparison sorts, and when each wins	31
Chapter 06 · Sorting without comparing	38
Part III · Beyond exact & efficient	44
Chapter 07 · Probabilistic structures: trading certainty for memory	45
Chapter 08 · NP-completeness: when no fast algorithm is known	53
The library checklist	60
The eight rules	62
Reading list	63

This is a supplement to **Sorted Again**, not a replacement for it. **Sorted Again** teaches the problem-solving patterns of stage 02, from two pointers to dynamic programming, through 150 curated problems with tests. It builds its solutions on Python's built-in list, dictionary, heap and sort, which is exactly right for learning patterns. The stage asks for more: "implement every structure once, from nothing, before you ever use the library version again". This volume does that, and covers the topics the course leaves out. The table shows where each of the stage's seventeen topics is taught.

STAGE 02 TOPIC	SORTED AGAIN MODULE	THIS VOLUME
Asymptotic analysis, amortised cost	Big-O and complexity	Ch. 1
Arrays, dynamic arrays, amortised append	Arrays and hashing	Ch. 1 (built)
Linked lists, stacks, queues, dequeues, circular buffers	Linked lists; Stacks	Ch. 1 (ring buffer)
Hash tables: collisions, load factor, resizing	uses <code>dict</code> and <code>set</code>	Ch. 2 (built)
Trees: BSTs, balancing, heaps, tries	Trees; Tries; Heaps	Ch. 3–4 (built)
Graphs: BFS, DFS, topological sort, cycles	Graphs	—
Shortest paths, A*, minimum spanning trees	Graphs (Dijkstra, Bellman–Ford, Prim, Kruskal)	Ch. 3 (A*)
Union-find with path compression	Graphs	—
Sorting algorithms, stability, when each wins	uses <code>sorted()</code>	Ch. 5–6 (built)
Binary search, including the answer space	Binary search	—
Recursion, divide and conquer, dynamic programming	Recursion; Dynamic programming	—
Greedy algorithms and the exchange	Greedy algorithms	—

argument

Backtracking and pruning	Backtracking	—
Two pointers, sliding window, prefix sums, monotonic stacks	Two pointers; Sliding window; Stacks	—
Bit manipulation	Bit manipulation and math	—
NP-completeness and reductions	not covered	Ch . 8
Bloom filters, count-min sketch, HyperLogLog	not covered	Ch . 7

Build first, then measure. Every structure in this volume is built in plain Python from a fixed-size list, and every example counts what it did: copies, probes, comparisons, cells expanded. Counts are the honest measure here. They don't depend on your machine, they come out the same every run, and they're what the complexity analysis actually predicts. Every example was run before publication, and the output printed under it is what it printed.

The deliverable is a library. The stage asks for every structure built from scratch, with tests and the complexity of each method documented. The appendix turns that into a checklist: each structure, its operations, what each one costs, and where in this volume or in Sorted Again it's built. Write your library against it, with tests that check behaviour against Python's built-ins on random input, as the examples here do.

THE ONE QUESTION

A complexity class on its own isn't enough to choose a data structure. "Constant time" hash lookups can take a thousand steps, and "logarithmic" trees can take four thousand.

So for every operation, ask: **what does it cost in the worst case, and what makes the worst case happen?** Sometimes the answer is "rarely, and it averages out", as with an array that occasionally doubles. Sometimes it's "only when the input is sorted", which in practice is all the time. And sometimes it's "whenever an attacker chooses the keys". Every chapter here is that question asked of one structure.

PART I

Build the containers

Four chapters, four structures you have used a thousand times without building: the growable array and the ring buffer, the hash table, the binary heap, and the balanced search tree. Each one is built from a fixed-size list and measured, so the costs that the library versions hide become numbers you can see.

Dynamic arrays and ring buffers

3 exercises

An array is the fastest structure there is: its elements sit side by side in memory, so any one of them is a single address calculation away. Its one flaw is a fixed size. A dynamic array grows by allocating a bigger block and copying, and growing by a multiple rather than a fixed amount is what makes the copying average out to a constant per append. When a queue needs to work at both ends, a ring buffer gets the same guarantee without moving anything at all.

Why contiguous memory wins

An array stores its elements next to each other, so the address of element `i` is the start address plus `i` times the element size: constant-time access with no pointers to follow. Contiguity also suits the hardware from stage 04. Reading one element pulls its neighbours into the same cache line, so a loop over an array streams through memory at full speed, while a loop over a linked list jumps to a new, probably uncached, address at every step. Python's list is an array, of references rather than values, and so are Java's `ArrayList`, C++'s `vector` and Go's slices.

Growing without paying for it

A fixed-size block can't grow in place, because the memory after it may belong to something else. So when a dynamic array is full, it allocates a new, larger block, copies every element across, and frees the old one. A single append can therefore cost as much as the whole array. What saves it is how much larger the new block is.

EXAMPLE `arrays.py` · a growable array with three growth policies, and a ring buffer

```
class DynamicArray:
    """A growable array on top of fixed-size storage: when it's full, allocate more and
    copy."""
    def __init__(self, grow=lambda cap: cap * 2, shrink_at=4):
        self.cap, self.n, self.slots = 1, 0, [None]
        self.grow, self.shrink_at, self.copies, self.resizes = grow, shrink_at, 0, 0
    def _resize(self, new_cap):
        slots = [None] * new_cap
        for i in range(self.n):
            slots[i] = self.slots[i]
            self.copies += 1
        self.slots, self.cap, self.resizes = slots, new_cap, self.resizes + 1
    def append(self, x):
        if self.n == self.cap:
            self._resize(max(self.cap + 1, self.grow(self.cap)))
        self.slots[self.n] = x
        self.n += 1
    def pop(self):
        self.n -= 1
```

```

        x, self.slots[self.n] = self.slots[self.n], None
        if self.cap > 1 and self.n <= self.cap // self.shrink_at:
            self._resize(max(1, self.cap // 2))    # give memory back, but not too eagerly
        return x
    def __getitem__(self, i):
        if not 0 <= i < self.n:
            raise IndexError(i)
        return self.slots[i]

N = 50_000
print(f"appending {N:,} items:")
for label, grow in (("double (x2)", lambda c: c * 2), ("grow by half (x1.5)", lambda c: c + c
// 2),
                    ("add 1,000 slots", lambda c: c + 1000)):
    a = DynamicArray(grow)
    for i in range(N):
        a.append(i)
    print(f"    {label:20} {a.resizes:4} resizes, {a.copies:>10,} copies = {a.copies / N:6.2f}
        per append")

print("pushing and popping at the boundary, 1,000 times:")
for label, shrink_at in (("shrink when half full", 2), ("shrink when a quarter full", 4)):
    a = DynamicArray(shrink_at=shrink_at)
    for i in range(1024):
        a.append(i)
    a.copies = a.resizes = 0
    for _ in range(1000):
        a.append(0); a.pop()
    print(f"    {label:27} {a.resizes:5} resizes, {a.copies:>9,} copies")

class RingBuffer:
    """A fixed-capacity queue: head and tail indices chase each other around one array."""
    def __init__(self, capacity):
        self.slots, self.head, self.size = [None] * capacity, 0, 0
    def push(self, x):
        if self.size == len(self.slots):
            raise OverflowError("full")
        self.slots[(self.head + self.size) % len(self.slots)] = x
        self.size += 1
    def pop(self):
        x = self.slots[self.head]
        self.head, self.size = (self.head + 1) % len(self.slots), self.size - 1
        return x

q, moved, out_ring, out_list, lst = RingBuffer(1000), 0, [], [], []
for i in range(1000):
    q.push(i); lst.append(i)
for i in range(1000, 6000):
    out_ring.append(q.pop()); q.push(i)
    moved += len(lst) - 1
    left
    out_list.append(lst.pop(0)); lst.append(i)
print(f"queue of 1,000 under 5,000 push/pop pairs: ring buffer moved 0 elements, "
      f"list.pop(0) moved {moved:,}; same order out: {out_ring == out_list}")

```

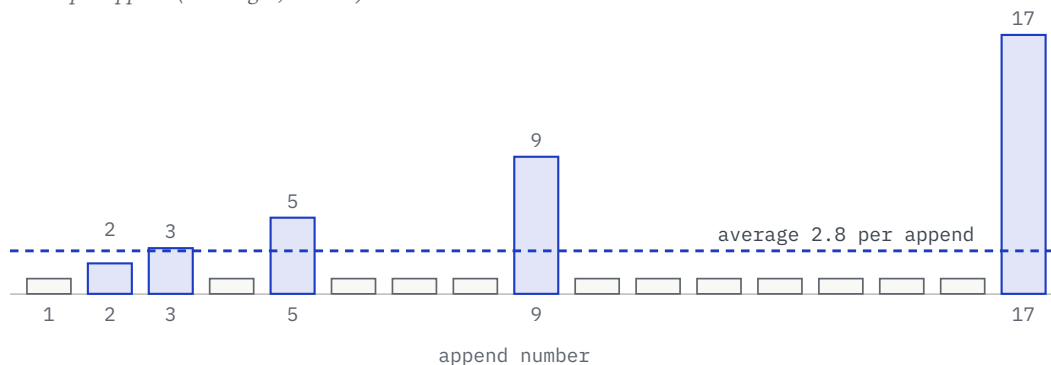
OUTPUT

```

appending 50,000 items:
double (x2)          16 resizes,      65,535 copies =   1.31 per append
grow by half (x1.5)  27 resizes,     122,904 copies =   2.46 per append
add 1,000 slots      50 resizes,    1,225,050 copies =  24.50 per append
pushing and popping at the boundary, 1,000 times:
shrink when half full      2000 resizes, 2,048,000 copies
shrink when a quarter full    1 resizes,   1,024 copies
queue of 1,000 under 5,000 push/pop pairs: ring buffer moved 0 elements, list.pop(0) moved
4,995,000; same order out: True

```

writes per append (bar height, to scale)



The cost of each append to a doubling array, to scale. Most appends write one element. The rare resizes copy everything first, but each one doubles the room, so they get exponentially rarer as they get bigger, and the average stays below three writes per append however long the array grows.

With doubling, 50,000 appends caused 16 resizes and 65,535 copies, 1.31 copies per append. The copies form a geometric series, $1 + 2 + 4 + \dots + 32,768$, which always totals less than twice the final size. That's the **amortised** cost: some appends are expensive, but averaged over any sequence of appends, each one costs a constant. Growing by half is just as constant, at 2.46 copies per append, and leaves less memory unused. Growing by a fixed 1,000 slots is a different kind of thing. It resizes every 1,000 appends and copies everything each time, so the total is quadratic: 1.2 million copies here, 24.5 per append, and the per-append figure keeps rising as the array grows.

An accounting argument gives the same result without the series. Charge every append three units: one pays for writing the element, and two go into a savings account. When the array doubles from k to $2k$, the $k/2$ elements appended since the last resize have saved k units, exactly enough to pay for copying all k elements. The account never goes negative, so three units per append always covers the work.

Real libraries pick different factors for different trade-offs. Java's `ArrayList` grows by half; GNU's C++ library doubles a `vector`, while Microsoft's grows it by half; Go doubles small slices and grows large ones more gently; CPython over-allocates a list by about an eighth plus a small constant. Smaller factors waste less memory, larger ones copy less, and every one of them is a multiple, never a fixed step.

Shrinking without thrashing

An array that never shrinks wastes memory after a burst of deletions, so dynamic arrays usually give space back. The obvious rule, halve the capacity when the array falls to half full, has a trap the second block of output shows. At the boundary, one append fills the array and forces a doubling, and the next pop empties it to half and forces a halving. Alternating the two copies the whole array every time: 2,000 resizes and over two million

copies for 1,000 pairs of operations. Shrinking only when the array falls to a quarter full, and then only to half, leaves room to move in both directions. The same 2,000 operations then cost one resize. This gap between the grow threshold and the shrink threshold, called **hysteresis**, appears wherever a system resizes itself, from hash tables to autoscaling groups.

Queues: the ring buffer

A queue adds at one end and removes at the other, and a plain array is bad at removing from the front: `list.pop(0)` shifts every remaining element left by one. Under a steady load of 1,000 queued items, 5,000 removals moved nearly five million elements. A **ring buffer** fixes this by never moving anything. It keeps a head index and a size, writes new items at `(head + size) mod capacity`, and advances the head on removal, wrapping around to the start of the array when it reaches the end. Both operations are constant time, with no copying, and the memory is allocated once. That predictability is why ring buffers sit inside network cards, audio pipelines, logging systems and the Linux kernel's `io_uring` interface. When a queue must also grow, use a ring buffer that doubles (exercise 1.3), or Python's `collections.deque`, a linked list of fixed-size blocks that is constant time at both ends.

COMMON TRAPS

Growing by a constant amount, which turns appends quadratic as the array grows.

Shrinking at half full, so operations at the boundary resize every time.

Using `list.pop(0)` as a queue, moving every element on every removal.

Treating amortised cost as worst-case cost: one append can still copy the whole array, which matters when latency is the requirement.

Keeping pointers or iterators into an array across a resize, since the elements now live somewhere else.

EXERCISES

EXERCISE 1.1 Count the copies

A doubling array starts with capacity 1. Exactly how many element copies do $2^k + 1$ appends cause? Check your formula against the example's 50,000 appends.

SOLUTION

Resizes happen when the array holds $1, 2, 4, \dots, 2^k$ elements, and each copies that many, so the total is $1 + 2 + \dots + 2^k = 2^{k+1} - 1$ copies. For $2^k + 1$ appends that's just under two copies per append, the worst ratio, because the last append has just paid for a full resize. Fifty thousand appends trigger resizes at $1, 2, \dots, 32,768$ elements, since $2^{15} = 32,768 < 50,000 < 65,536$, so the total is $2^{16} - 1 = 65,535$ copies, exactly what the example printed.

EXERCISE 1.2 Why a fixed step is quadratic

Show that growing by a fixed `c` slots makes `n` appends cost about $n^2 / (2c)$ copies, and check it against the example's +1,000 policy.

SOLUTION

Resizes happen at sizes $c, 2c, 3c, \dots$, up to about n , and each copies the current size, so the total is $c(1 + 2 + \dots + n/c) \approx c \cdot (n/c)^2 / 2 = n^2 / (2c)$. For $n = 50,000$ and $c = 1,000$ that's $2.5 \times 10^9 / 2,000 = 1.25$ million, against the example's 1,225,050: its resizes happen at $1, 1,001, 2,001, \dots, 49,001$ elements, which sum to exactly that. The cost per append, $n/(2c)$, grows with n , which is the signature of a quadratic total. Doubling the array would make it a constant.

EXERCISE 1.3 A ring buffer that grows

Extend the ring buffer so `push` on a full buffer doubles its capacity instead of failing. What must the copy do that an ordinary array resize doesn't?

SOLUTION

Allocate an array of twice the capacity and copy the elements *in queue order*, starting from the head and wrapping around: `new[i] = old[(head + i) % cap]` for `i` in `range(size)`. Then set `head = 0`. A plain slice copy of the old array would keep the wrap-around in the middle of the new array, so the items after the old array's end would be followed by empty slots instead of the items from its start. Copying in queue order "unrolls" the ring. The cost is amortised constant per push, by the same argument as the dynamic array, and it's how `ArrayDeque` in Java and Rust's `VecDeque` work.

- ☐ Can you explain why doubling makes appends amortised constant?
- ☐ Can you prove it with either the geometric series or the accounting method?
- ☐ Does your dynamic array shrink with hysteresis?
- ☐ Do your queues use a ring buffer or a deque rather than `pop(0)` ?
- ☐ Do you know when amortised cost isn't good enough?

GO DEEPER

COURSE MIT 6.006: data structures and dynamic arrays Erik Demaine et al.

SOURCE CPython's list implementation the over-allocation comment in `list_resize`

BOOK Introduction to Algorithms, amortised analysis Cormen, Leiserson, Rivest & Stein

Hash tables from scratch

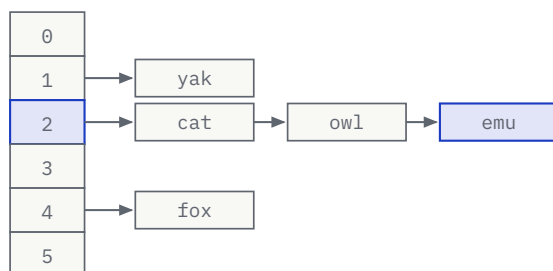
3 exercises

A hash table turns a key into an array index, so finding anything takes about one step. That holds only while three things stay true: the hash spreads keys evenly, the table never gets too full, and nobody can choose keys that collide on purpose. Every hash table design is a way of handling the collisions that remain.

From key to slot

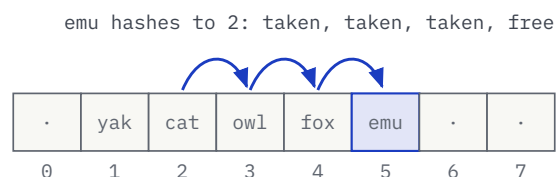
A hash function turns a key into a large number, and the number modulo the table size picks a slot. If every key landed in its own slot, a lookup would be one array access. They don't. Collisions arrive much sooner than intuition suggests, by the same arithmetic as the birthday paradox: with 365 slots, 23 keys are more likely than not to include a collision (exercise 2.1). So the real design question is what to do when two keys want the same slot, and there are two families of answers.

separate chaining



a lookup walks one slot's chain

linear probing



emu hashes to 2: taken, taken, taken, free

keys that collide pile up into one cluster, and the cluster makes later collisions likelier

The two families of collision handling. Chaining hangs a list off each slot, so a lookup scans one short list. Probing keeps every key in the array itself and steps to the next free slot, which is cache-friendly but forms clusters that grow faster as the table fills.

Separate chaining makes each slot a list, and a lookup scans the list for its key. It degrades gracefully: at an average of one key per slot, chains are short. **Open addressing** keeps every key in the array itself. On a collision, it probes other slots in a fixed sequence, the next one for **linear probing**, until it finds the key or an empty slot. Probing avoids the pointers and allocations of chains and reads consecutive memory, which modern hardware rewards, but it is far more sensitive to how full the table is.

EXAMPLE `hashing.py` · chaining, linear probing, tombstones and a collision attack

```

import random

MASK64 = (1 << 64) - 1

def mix(k):
    """A deterministic integer hash (splitmix64's finaliser): nearby keys land far apart."""
    k = (k + 0x9E3779B97F4A7C15) & MASK64
    k = ((k ^ (k >> 30)) * 0xBF58476D1CE4E5B9) & MASK64
    k = ((k ^ (k >> 27)) * 0x94D049BB133111EB) & MASK64
    return k ^ (k >> 31)

class ChainedMap:
    def __init__(self, hash_fn=mix, buckets=8):
        self.h, self.buckets, self.n, self.compares = hash_fn, [[] for _ in range(buckets)], 0, 0
    def _bucket(self, key):
        return self.buckets[self.h(key) % len(self.buckets)]
    def put(self, key, value):
        b = self._bucket(key)
        for pair in b:
            if pair[0] == key:
                pair[1] = value; return
        b.append([key, value]); self.n += 1
        if self.n > 0.75 * len(self.buckets):
            # keep chains short: double and
            rehash
            old, self.buckets = self.buckets, [[] for _ in range(2 * len(self.buckets))]
            for chain in old:
                for k, v in chain:
                    self._bucket(k).append([k, v])
    def get(self, key):
        for k, v in self._bucket(key):
            self.compares += 1
            if k == key: return v
        return None

class ProbingMap:
    """Open addressing with linear probing: on a collision, try the next slot."""
    EMPTY, GONE = object(), object()
    def __init__(self, capacity):
        self.slots, self.probes = [self.EMPTY] * capacity, 0
    def put(self, key, value):
        i = mix(key) % len(self.slots)
        while self.slots[i] is not self.EMPTY and self.slots[i] is not self.GONE and self.slots[i][0] != key:
            i = (i + 1) % len(self.slots)
        self.slots[i] = (key, value)
    def get(self, key):
        i = mix(key) % len(self.slots)
        while True:
            self.probes += 1
            s = self.slots[i]
            if s is self.EMPTY: return None
            if s is not self.GONE and s[0] == key: return s[1]
            i = (i + 1) % len(self.slots)
    def delete(self, key):
        i = mix(key) % len(self.slots)
        while self.slots[i] is not self.EMPTY:
            if self.slots[i] is not self.GONE and self.slots[i][0] == key:
                self.slots[i] = self.GONE; return # a tombstone keeps later probe chains intact
            i = (i + 1) % len(self.slots)

random.seed(2)
CAP = 20_000
print("linear probing: average probes per lookup (Knuth's formulas in brackets)")
for alpha in (0.25, 0.5, 0.75, 0.9):
    m, keys = ProbingMap(CAP), random.sample(range(10**9), int(CAP * alpha))

```

```

m, keys = ProbingMap(CAP), random.sample(range(10**9), int(CAP * alpha))

for k in keys: m.put(k, k)
m.probes = 0
for k in keys: m.get(k)
hit = m.probes / len(keys)
m.probes = 0
for k in random.sample(range(10**9, 2 * 10**9), 5000): m.get(k)
miss = m.probes / 5000
print(f"load {alpha:4}: hit {hit:5.2f} [{(1 + 1 / (1 - alpha)) / 2:5.2f}] "
      f"miss {miss:6.2f} [{(1 + 1 / (1 - alpha) ** 2) / 2:6.2f}]")

m, keys = ProbingMap(CAP), random.sample(range(10**9), CAP // 2)
for k in keys: m.put(k, k)
for k in keys[: CAP // 4]: m.delete(k) # delete half; the slots become
    tombstones
m.probes = 0
for k in random.sample(range(10**9, 2 * 10**9), 5000): m.get(k)
print(f"after deleting half the keys, a miss still costs {m.probes / 5000:.2f} probes: "
      "tombstones count as occupied until a rebuild")

chains = {}
for name, fn in (("good hash (mix)", mix), ("weak hash (k mod 1024)", lambda k: k % 1024)):
    hm = ChainedMap(fn, buckets=1024)
    hm.buckets = [[] for _ in range(1024)]
    attack = [i * 1024 for i in range(2000)] # keys chosen so k mod 1024 == 0
    for k in attack:
        b = hm.buckets[fn(k) % 1024]; b.append([k, k]) # insert without resizing, to
        show the shape
    for k in attack: hm.get(k)
    chains[name] = max(len(b) for b in hm.buckets)
    print(f"{name:24} 2,000 crafted keys: longest chain {chains[name]:5}, "
          f"{hm.compares / len(attack):7.1f} comparisons per lookup")

```

OUTPUT

```

linear probing: average probes per lookup (Knuth's formulas in brackets)
load 0.25: hit 1.16 [ 1.17] miss 1.40 [ 1.39]
load 0.5: hit 1.51 [ 1.50] miss 2.50 [ 2.50]
load 0.75: hit 2.54 [ 2.50] miss 8.63 [ 8.50]
load 0.9: hit 5.35 [ 5.50] miss 47.56 [ 50.50]
after deleting half the keys, a miss still costs 2.53 probes: tombstones count as occupied
until a rebuild
good hash (mix)          2,000 crafted keys: longest chain 8,      2.0 comparisons per
lookup
weak hash (k mod 1024)   2,000 crafted keys: longest chain 2,000, 1000.5 comparisons per
lookup

```

Load factor is everything

The first block measures linear probing at four load factors, the fraction of slots in use, against the formulas Donald Knuth derived in 1963. They agree closely. A successful lookup stays cheap for a long time: 1.16 probes at a quarter full, 2.54 at three-quarters. Unsuccessful lookups, which must probe until they hit an empty slot, explode: 2.5 probes at half full, 8.6 at three-quarters, and 47.6 at 90%. The cause is **clustering**. Keys that collide fill neighbouring slots, so the run of full slots grows, and every key that hashes anywhere into the run extends it further. That's why tables resize long before they're full: Java's `HashMap` at 0.75, CPython's `dict` at two-thirds. Resizing allocates a table twice the size and reinserts every key, because each key's slot depends on the table size. As with chapter 1's array, doubling makes the cost amortised constant.

Deletion leaves tombstones

Deleting from an open-addressed table can't simply empty the slot. A key placed further along the probe sequence, because this slot was occupied when it arrived, would become unreachable, since a lookup stops at the first empty slot it meets. So deletion leaves a **tombstone**: a marker that lookups step over and inserts may reuse. The example's second line shows the cost. After deleting half the keys, the table holds as many live keys as a quarter-full one, but a miss still costs 2.53 probes, the half-full figure, because tombstones count as occupied. They are cleared only when the table is rebuilt. Alternative schemes avoid tombstones by shifting later entries back on deletion, or by Robin Hood hashing, which keeps probe lengths even.

Hash flooding

Every guarantee above assumes keys are spread evenly. The last two lines show what happens when an adversary chooses them. With a weak hash, the key modulo the table size, 2,000 keys that are all multiples of 1,024 land in one bucket, and every lookup scans a chain of 2,000: 1,000.5 comparisons where the mixing hash needs 2.0. This isn't theoretical. In December 2011, researchers showed that a single crafted HTTP request with thousands of colliding form field names could keep a web server's CPU busy for minutes, across PHP, Java, Python, Ruby and others, because those languages parsed request parameters into hash tables with predictable hashes. The fixes are why modern hash tables look the way they do. Python 3.4 adopted SipHash, a keyed hash with a random key per process (PEP 456), so attackers can't predict collisions. Java 8's `HashMap` turns any long chain into a balanced tree (chapter 4), capping the damage at logarithmic.

What the library versions add

Production hash tables combine these ideas with engineering for the hardware. CPython's `dict` keeps entries in a compact array in insertion order and a separate small index of slots, which is why dictionaries remember insertion order (guaranteed since Python 3.7). Google's Swiss tables and Rust's `hashbrown` store a byte of each hash in a metadata array and check sixteen slots at once with SIMD instructions. All of them still depend on the same three conditions: a good hash, a bounded load factor, and keys an attacker can't aim.

COMMON TRAPS

Mutating a key after inserting it, so it now hashes to a different slot and can't be found.

Equality and hashing that disagree: equal objects must have equal hashes, or a dictionary holds duplicates.

Letting the load factor climb, where missed lookups degrade sharply.

Weak or predictable hashes on keys users control, the recipe for hash flooding.

Relying on iteration order in languages and versions that don't promise one.

EXERCISES

EXERCISE 2.1 The first collision

With a 1,000-slot table and a perfectly uniform hash, roughly how many keys can you insert before a collision is more likely than not?

SOLUTION

The chance that `n` keys in `m` slots are all distinct is about $e^{-n^2/(2m)}$. Setting that to one half gives $n \approx \sqrt{(2m \ln 2)} \approx 1.18\sqrt{m}$. For $m = 1,000$ that's about 37 keys, under 4% of the table. For $m = 365$ it's about 23, the birthday paradox. The lesson for design: collisions are a certainty from almost the first insertion, so collision handling is the main path, not an edge case.

EXERCISE 2.2 Deleting from each design

Write the deletion logic for a chained table and for a linear-probing table, and explain why emptying the slot breaks the probing table.

SOLUTION

Chained: find the key's bucket and remove its entry from the list. Nothing else is affected, because each key is only ever in its own bucket. Probing: find the slot holding the key and replace the entry with a tombstone, as the example's `delete` does. Emptying it instead breaks every key that was placed further along the same probe run. Suppose `cat`, `owl` and `emu` all hash to slot 2 and sit in slots 2, 3 and 5. Empty slot 3, and a lookup for `emu` starts at 2, finds `cat`, steps to the empty slot 3, and stops: `emu` is still in the table and can no longer be found. The alternative to tombstones is backward-shift deletion, which moves the later entries of the run back to fill the hole.

EXERCISE 2.3 When equality and hashing disagree

A Python class defines `__eq__` to compare a customer's email, and `__hash__` to return `id(self)`. Two objects with the same email go into a set. What happens, and how do you fix it?

SOLUTION

The two objects are equal but almost certainly have different hashes, so they land in different slots and the set keeps both: a duplicate the set exists to prevent. Lookups behave just as badly:

```
Customer("ana@x.com") in seen
```

 hashes a new object to yet another slot and reports `False` even though an equal customer is in the set. The rule is that equal objects must have equal hashes, so hash exactly what `__eq__` compares:

```
def __hash__(self): return hash(self.email)
```

 . The key must also never change while it's in a set or dictionary, since a changed email would change the hash. Python enforces half of this rule for you: defining `__eq__` without `__hash__` makes instances unhashable, so this bug needs a deliberately wrong `__hash__` .

CHECKLIST

- ☐ Can you explain chaining and open addressing, and when each suits the hardware?
- ☐ Does your table resize well before it's full?
- ☐ Does your probing table handle deletion with tombstones or backward shifts?
- ☐ Are hash and equality consistent for every key type you store?
- ☐ Are keys that users control hashed with a keyed, randomised hash?

GO DEEPER

REFERENCE [PEP 456: secure and interchangeable hash algorithm](#) Christian Heimes

DESIGN NOTE [Swiss tables design notes](#) Abseil

SOURCE [CPython's dict implementation](#) the long comment at the top explains the probing

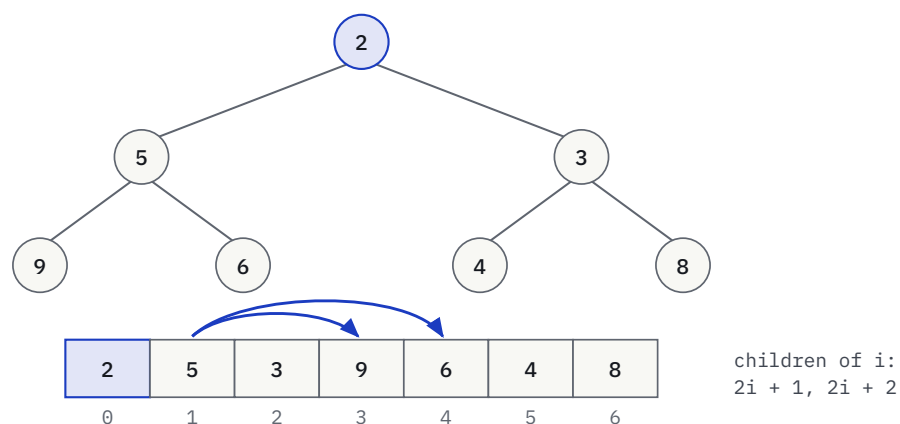
Heaps, priority queues and A*

3 exercises

A priority queue answers one question over and over: what is the most urgent item right now? A binary heap answers it in logarithmic time with nothing but an array and arithmetic on its indexes. It is the engine inside schedulers, event simulations, top-*k* queries and every shortest-path search, including A*, which adds a guess at the distance remaining and searches far less of the map.

A tree hiding in an array

A binary heap is a binary tree with one rule, the **heap property**: every parent is no larger than its children (for a min-heap). It doesn't keep the whole tree sorted, only each parent above its children, which is much cheaper to maintain and still puts the smallest item at the root. The second trick is that the tree is always **complete**, filled level by level from the left, so it fits in an array with no gaps and no pointers. The root is at index 0, and the children of index *i* are at $2i + 1$ and $2i + 2$.



The same heap as a tree and as an array. The array has no gaps because the tree is filled level by level, so moving between parent and children is arithmetic on indexes, with no pointers to store or follow.

Both operations restore the heap property along one path, so both cost the height of the tree, about $\log_2 n$. To **push**, append the item at the end and **sift it up**, swapping it with its parent while it's smaller. To **pop** the minimum, take the root, move the last item into its place, and **sift it down**, swapping it with its smaller child while either child is smaller.

EXAMPLE `heaps.py` · a binary heap, two ways to build one, and Dijkstra against A*

```
import random
```

```

class MinHeap:
    """A binary heap stored in a list: the children of index i live at 2i+1 and 2i+2."""
    def __init__(self, items=()):
        self.a, self.compares = list(items), 0
        for i in range(len(self.a) // 2 - 1, -1, -1):      # bottom-up heapify: O(n), not O(n
            log n)
            self._down(i)
    def _less(self, i, j):
        self.compares += 1
        return self.a[i] < self.a[j]
    def _up(self, i):
        while i > 0 and self._less(i, (i - 1) // 2):
            p = (i - 1) // 2
            self.a[i], self.a[p] = self.a[p], self.a[i]
            i = p
    def _down(self, i):
        n = len(self.a)
        while True:
            l, r, small = 2 * i + 1, 2 * i + 2, i
            if l < n and self._less(l, small): small = l
            if r < n and self._less(r, small): small = r
            if small == i: return
            self.a[i], self.a[small] = self.a[small], self.a[i]
            i = small
    def push(self, x):
        self.a.append(x); self._up(len(self.a) - 1)
    def pop(self):
        last = self.a.pop()
        if not self.a: return last
        top, self.a[0] = self.a[0], last
        self._down(0)
        return top
    def __len__(self):
        return len(self.a)

random.seed(3)
N = 100_000
print(f"building a heap of {N:,} items, counting comparisons:")
for label, data in (("random order", [random.random() for _ in range(N)]),
                    ("descending order", list(range(N, 0, -1)))):      # every push climbs to
                                the root
    one_by_one = MinHeap()
    for x in data: one_by_one.push(x)
    bottom_up = MinHeap(data)
    print(f"    {label:17} one push at a time {one_by_one.compares:>9,}    bottom-up heapify
          {bottom_up.compares:>9,}")
print("heapsort matches sorted():", [bottom_up.pop() for _ in range(len(bottom_up))] ==
    sorted(data))

# The same heap drives shortest-path search. A* adds a guess of the remaining distance.
W = H = 60
def search(heuristic, walls):
    start, goal = (0, 0), (W - 1, H - 1)
    best, frontier, expanded = {start: 0}, MinHeap([(heuristic(start), 0, start)]), 0
    while frontier:
        _, neg_g, (x, y) = frontier.pop()
        g = -neg_g
        if g > best.get((x, y), float("inf")): continue      # a stale entry: a shorter way was
            found
        expanded += 1
        if (x, y) == goal: return g, expanded
        for nx, ny in ((x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)):
            if 0 <= nx < W and 0 <= ny < H and (nx, ny) not in walls and g + 1 < best.get((nx,
                ny), float("inf")):
                best[(nx, ny)] = g + 1

```

```

        frontier.push((g + 1 + heuristic((nx, ny)), -(g + 1), (nx, ny))) # ties go to
                                the deeper cell
    return None, expanded

def manhattan(p): return (W - 1 - p[0]) + (H - 1 - p[1]) # never overestimates on a 4-
connected grid
for seed in range(100): # the first maze, in a fixed order,
    with a way through
    random.seed(seed)
    maze = {(x, y) for x in range(W) for y in range(H) if random.random() < 0.3} - {(0, 0), (W
        - 1, H - 1)}
    if search(lambda p: 0, maze)[0] is not None:
        break
print(f"shortest path across a {W}x{H} grid, cells expanded:")
for label, walls in (("open ground", set()), ("maze, 30% walls", maze)):
    (d_len, d_exp), (a_len, a_exp) = search(lambda p: 0, walls), search(manhattan, walls)
    print(f"    {label:16} Dijkstra {d_exp:5,} cells, A* {a_exp:5,} cells, same length: {d_len
        == a_len} ({a_len} steps)")

```

OUTPUT

```

building a heap of 100,000 items, counting comparisons:
    random order      one push at a time   228,554   bottom-up heapify   188,339
    descending order  one push at a time 1,468,946   bottom-up heapify   199,978
heapsort matches sorted(): True
shortest path across a 60x60 grid, cells expanded:
    open ground      Dijkstra 3,600 cells, A*   119 cells, same length: True (118 steps)
    maze, 30% walls  Dijkstra 2,465 cells, A* 2,005 cells, same length: True (128 steps)

```

Building a heap in linear time

The first block compares two ways to turn 100,000 items into a heap. Pushing them one at a time costs up to $\log n$ per item, and on descending input, where every new item is the smallest yet and climbs all the way to the root, it costs 1.47 million comparisons. On random input it's far cheaper, because most new items stop near the bottom. **Bottom-up heapify** works the other way: it sifts down every internal node, from the last one back to the root. It costs about 200,000 comparisons on either input, because it's guaranteed linear. Half the nodes are leaves and never move, a quarter can sift down at most one level, an eighth at most two, and the sum of those distances is less than n . When you have all the items up front, heapify them; Python's `heapq.heapify` does exactly this.

Popping everything from a heap returns the items in order, which is **heapsort**: $O(n \log n)$ in the worst case, in place, with no extra memory. It's rarely the library sort anyway. It isn't stable, and its jumps between parents and children defeat the cache, so in practice it serves as a safety net: introsort, the usual C++ `std::sort`, switches to heapsort when quicksort starts going badly (chapter 5).

Priority queues in the wild

Anything that repeatedly asks for the most urgent item is a priority queue. Operating system schedulers pick the next task, event simulations process the earliest event, timers fire the soonest deadline, and "the top 100 of ten million" keeps a heap of 100 (exercise 3.2). One operation the basic heap lacks is changing the priority of an item already inside it, which Dijkstra's algorithm wants. Production code either keeps an index from items to

positions (an indexed heap), or does what the example does: push the item again with its new priority and skip the stale copy when it's popped. Python's `heapq` has no decrease-key at all, so that second approach is standard there.

A*: Dijkstra with a sense of direction

Dijkstra's algorithm, which Sorted Again's graphs module teaches, expands nodes in order of their distance from the start, `g`. It finds the shortest path, but it searches outwards in every direction equally, including away from the goal. A* orders the queue by `g + h` instead, where `h` is an estimate of the distance still to go. If the estimate never overestimates, it's called **admissible**, and A* is guaranteed to find a shortest path while expanding far fewer nodes. Dijkstra is simply A* with `h = 0`.

The second block shows how much depends on the estimate. On open ground, the Manhattan distance is exactly right, and A* expands only the 119 cells of the path itself, where Dijkstra expands all 3,600. In a maze with 30% walls, the estimate is optimistic, since walls force detours it can't see, so A* saves less: 2,005 cells against 2,465. Both always find the same length. Two details make it work: the heap holds `(g + h, -g, cell)`, so among equally promising cells the deeper one goes first, and a popped entry that's worse than a distance already found is skipped as stale. Route planners for real road networks go further: stronger estimates built from precomputed distances to landmarks, and preprocessing schemes such as contraction hierarchies that answer continental queries thousands of times faster than plain Dijkstra.

COMMON TRAPS

Building a heap by repeated pushes when bottom-up heapify does it in linear time.

Forgetting stale entries when you re-push instead of decreasing a key.

An estimate that overestimates, which makes A* fast and wrong.

Expecting a heap to iterate in sorted order; only the root is guaranteed to be the minimum.

Forgetting that `heapq` is a min-heap, and getting the order backwards when you need the largest.

EXERCISES

EXERCISE 3.1 Index arithmetic

In a heap stored from index 0, what are the parent and children of index 5? Why can a binary heap be stored without gaps when an ordinary binary search tree can't?

SOLUTION

The parent of 5 is $(5 - 1) // 2 = 2$, and its children are $2 \cdot 5 + 1 = 11$ and 12. A heap has no gaps because it's always a complete tree: every level is full except possibly the last, which fills from the left, and push and pop preserve that shape by only ever adding or removing the last array element. A binary search tree's shape depends on the keys, so it has holes wherever a node has no left or right child. Storing it this way would waste space exponentially on a lopsided tree, which is why BSTs use pointers.

EXERCISE 3.2 Top 100 of ten million

Find the 100 largest values in a stream of 10 million, using as little memory as possible. Which heap, what size, and what does it cost?

SOLUTION

Keep a **min**-heap of size 100. Push the first 100 values. For each later value, if it's larger than the heap's minimum (its root), replace the root with it and sift down. The heap always holds the 100 largest values seen so far, and its root is the smallest of them, the bar a newcomer must clear. The cost is $O(n \log k)$: 10 million items $\times$ about 7 levels at worst, and far less on typical data, since most values fail the comparison with the root and cost one comparison. The memory is 100 items. Sorting everything would need all 10 million in memory and $O(n \log n)$ time. Python's `heapq.nlargest` does exactly this.

EXERCISE 3.3 A faster, wrong A*

A game multiplies its A* estimate by 1.5 to make pathfinding faster. What does it gain, what does it lose, and when is that a reasonable trade?

SOLUTION

It gains speed: an inflated estimate pulls the search harder towards the goal, so it expands fewer cells, especially in open terrain. It loses the guarantee of a shortest path, because the estimate can now overestimate and A* may commit to a route before it has ruled out a shorter one. This is called **weighted A***, and it does keep a bound: with a weight of 1.5, the path found is never more than 1.5 times the

shortest. For game characters, where a path that looks reasonable and is found quickly beats a perfect one found late, that's a common, sensible trade. For routing a delivery van, where every extra kilometre costs money, it isn't.

CHECKLIST

- ☐ Can you implement sift-up, sift-down and bottom-up heapify from memory?
- ☐ Can you explain why heapify is linear?
- ☐ Do you handle priority changes with an indexed heap or by skipping stale entries?
- ☐ Can you say what makes an A* estimate admissible, and what happens when it isn't?
- ☐ Do you reach for a size-k heap for top-k problems?

GO DEEPER

GUIDE [Introduction to the A* algorithm](#) Amit Patel, Red Blob Games

REFERENCE [heapq: heap queue algorithm](#) Python documentation

PAPER [A formal basis for the heuristic determination of minimum cost paths](#) Hart, Nilsson & Raphael, 1968

Balanced search trees

3 exercises

A binary search tree keeps keys in order and finds any of them in time proportional to its height. Insert keys in sorted order and the height becomes the number of keys: the tree has turned into a linked list. Balanced trees prevent this by restructuring themselves with small local rotations as they go, guaranteeing logarithmic height whatever order the keys arrive in.

What a search tree is for

Sorted Again's trees module covers binary search trees: every key in a node's left subtree is smaller than it, every key in the right subtree is larger, an in-order traversal visits the keys in sorted order, and a search follows a single path from the root. Hash tables are faster for exact lookups, so the reason to use a tree is everything a hash table can't do: iterate in order, find every key in a range, find the next key above or below a value, or the k-th smallest. The catch is that every operation costs the height of the tree, and nothing in a plain BST controls the height.

EXAMPLE `balanced.py` · a plain BST against an AVL tree, on sorted and random keys

```
import random

class BST:
    """A plain binary search tree: fine on random input, a linked list on sorted input."""
    def __init__(self):
        self.root = None  # nodes are [key, left, right]
    def insert(self, key):
        if self.root is None:
            self.root = [key, None, None]; return
        node = self.root
        while True:
            very deep
            side = 1 if key < node[0] else 2
            if node[side] is None:
                node[side] = [key, None, None]; return
            node = node[side]
    def height(self):
        h, level = 0, [self.root] if self.root else []
        while level:
            h, level = h + 1, [c for n in level for c in (n[1], n[2]) if c]
        return h

class AVL:
    """Keeps every node's two subtree heights within one of each other, by rotating."""
    class Node:
        __slots__ = ("key", "left", "right", "h")
        def __init__(self, key): self.key, self.left, self.right, self.h = key, None, None, 1
    def __init__(self):
        self.root, self.rotations = None, 0
    @staticmethod
```

```

def _h(n): return n.h if n else 0
def _fix(self, n): n.h = 1 + max(self._h(n.left), self._h(n.right))
def _rotate_right(self, y):
    x = y.left
    y.left, x.right = x.right, y
    self._fix(y); self._fix(x); self.rotations += 1
    return x
def _rotate_left(self, x):
    y = x.right
    x.right, y.left = y.left, x
    self._fix(x); self._fix(y); self.rotations += 1
    return y
def _insert(self, n, key):
    if n is None: return self.Node(key)
    if key < n.key: n.left = self._insert(n.left, key)
    else: n.right = self._insert(n.right, key)
    self._fix(n)
    balance = self._h(n.left) - self._h(n.right)
    if balance > 1: # left-heavy
        if key >= n.left.key: n.left = self._rotate_left(n.left) # left-right case
        return self._rotate_right(n)
    if balance < -1: # right-heavy
        if key < n.right.key: n.right = self._rotate_right(n.right) # right-left case
        return self._rotate_left(n)
    return n
def insert(self, key):
    self.root = self._insert(self.root, key)
def inorder(self):
    out, stack, n = [], [], self.root
    while stack or n:
        while n: stack.append(n); n = n.left
        n = stack.pop(); out.append(n.key); n = n.right
    return out

N = 4095 # 2^12 - 1: a perfect tree would
        have height 12
random.seed(4)
heights = {}
for label, keys in (("sorted keys", list(range(N))), ("random keys",
    random.sample(range(10**6), N))):
    plain, avl = BST(), AVL()
    for k in keys:
        plain.insert(k); avl.insert(k)
    heights[label] = (plain.height(), avl.root.h)
    print(f"{label:12} {N:}, inserts: plain BST height {plain.height():5} "
        f"AVL height {avl.root.h:3} ({avl.rotations:}, rotations) AVL in order:
        {avl.inorder() == sorted(keys)}")
worst_plain, worst_avl = heights["sorted keys"]
print(f"a search walks one root-to-leaf path, so on sorted input it costs up to {worst_plain:},
    steps "
        f"in the plain tree and {worst_avl} in the AVL tree")

```

OUTPUT

```

sorted keys  4,095 inserts: plain BST height 4095   AVL height 12 (4,083 rotations)   AVL in
order: True
random keys  4,095 inserts: plain BST height 24    AVL height 15 (2,767 rotations)   AVL in
order: True
a search walks one root-to-leaf path, so on sorted input it costs up to 4,095 steps in the
plain tree and 12 in the AVL tree

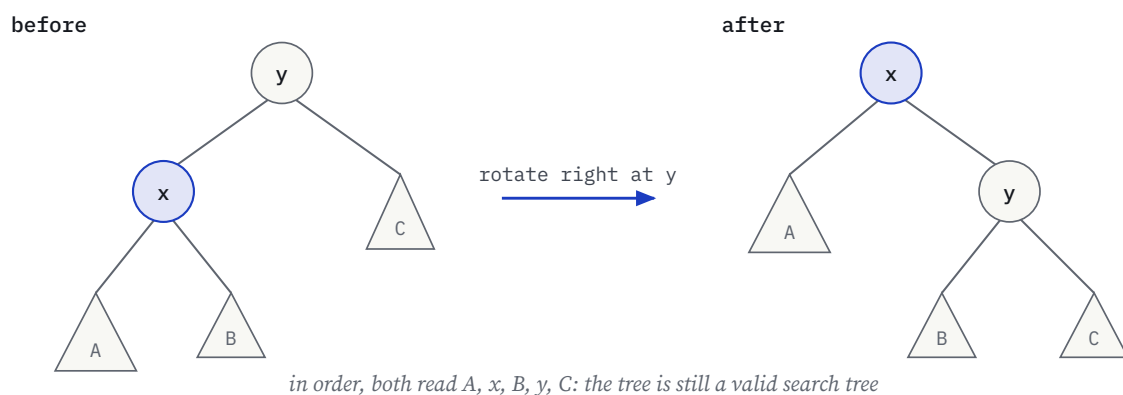
```

How a tree degenerates

Inserting 4,095 keys in sorted order into a plain BST gives a tree of height 4,095. Each new key is larger than everything before it, so it becomes the right child of the previous one, and the "tree" is a linked list with extra pointers. A search for the largest key walks all 4,095 nodes. Sorted input isn't an unlucky accident: auto-incrementing ids, timestamps, alphabetical imports and already-sorted files are some of the most common inputs there are. Random keys do much better, a height of 24 here, but a structure whose speed depends on the order the data happens to arrive in can't be trusted.

Rotations

A **rotation** is a local restructuring that changes which of two nodes is the parent while keeping the in-order sequence, and so the search-tree property, intact. It touches three pointers and runs in constant time.



A right rotation. The subtree B changes parent, from x to y, and x moves up. Everything in A stays below x and everything in C stays below y, so the order of keys is preserved, while the heavy left side gets one level shorter.

AVL trees: never more than one apart

The AVL tree, the first self-balancing tree (Adelson-Velsky and Landis, 1962), keeps one invariant: at every node, the heights of the two subtrees differ by at most one. After an insertion, the code walks back up the path, updates heights, and wherever a node's balance reaches two, fixes it with a rotation. There are four cases. In the left-left and right-right cases, a single rotation fixes it. In the left-right and right-left cases, the inner child is rotated first to turn it into one of the simple cases. The invariant bounds the height at about $1.44 \log_2 n$, so every search, insert and delete is logarithmic in the worst case.

The output shows the payoff. On the same 4,095 sorted keys that flattened the plain tree, the AVL tree has height 12, exactly what a perfect tree of 4,095 nodes would have, after 4,083 rotations: sorted input triggers one at almost every insertion, and the rotations pack the keys perfectly. On random keys it's 15, well within its bound, and an in-order traversal returns the keys sorted in both cases, confirming the rotations never broke the search-tree property.

The other balanced trees

Red-black trees keep a looser invariant, colouring nodes red or black and allowing a height of up to $2 \log_2(n + 1)$. In return, insertions and deletions need fewer rotations, which makes them the usual choice for general-purpose ordered maps: Java's `TreeMap`, most implementations of C++'s `std::map`, and many structures inside the Linux kernel. **B-trees** balance in a different way, with hundreds of keys per node to suit disk pages, which is why databases use them (The Engineer's Ledger, chapter 6). **Skip lists** get logarithmic expected time from random coin flips instead of rotations; Redis uses them for sorted sets, and LevelDB and RocksDB use them for their memtables. Python's standard library has no balanced tree at all. Where you need one, the popular `sortedcontainers` package uses a list of sorted lists, which performs very well in practice.

COMMON TRAPS

Assuming a binary search tree is balanced. A plain one is only as good as the order of its input.

Feeding sorted keys to an unbalanced tree: ids, timestamps and sorted files make that the common case.

Recursive algorithms on a degenerate tree, which exhaust the call stack long before they finish.

Using a tree where a hash table suffices, paying a logarithm for ordering you never use.

Writing your own balanced tree for production when tested library implementations exist; build one to learn, not to ship.

EXERCISES

EXERCISE 4.1 Rotate by hand

Insert 1, 2, 3, 4, 5, 6, 7 in that order into an empty AVL tree. Which rotations happen, and what does the final tree look like?

SOLUTION

After 1, 2, 3, the root 1 is right-heavy by two, a right-right case: rotate left at 1, making 2 the root with children 1 and 3. Inserting 4 and then 5 makes 3 right-heavy: rotate left at 3, giving $2 \rightarrow (1, 4 \rightarrow (3, 5))$. Inserting 6 makes the root 2 right-heavy: rotate left at 2, so 4 becomes the root with children 2 (over 1 and 3) and 5 (over 6). Inserting 7 makes 5 right-heavy: rotate left at 5, giving 6 over 5 and 7. The final tree is perfect: 4 at the

root, then 2 and 6, then 1, 3, 5 and 7. That's four rotations for seven sorted keys, the same pattern that gave the example its height of 12 for 4,095.

EXERCISE 4.2 Height bounds

A tree holds one million keys. What's the height of a perfectly balanced tree, the worst case for an AVL tree, the worst case for a red-black tree, and the worst case for a plain BST?

SOLUTION

Perfectly balanced: $\lceil \log_2(1,000,001) \rceil = 20$. AVL, at most about $1.44 \times \log_2 n \approx 1.44 \times 20 \approx 29$. Red-black, at most $2 \times \log_2(n + 1) \approx 40$. A plain BST on sorted input: one million. The balanced trees differ by a small constant factor, about which reasonable people choose differently: AVL is tighter, so better for lookup-heavy work, and red-black rotates less, so better for update-heavy work. The plain BST is in a different universe, which is the point of the chapter.

EXERCISE 4.3 Tree or hash table?

Choose a hash table or a balanced tree for each: (a) looking up sessions by id; (b) a leaderboard that must answer "what rank is this player?"; (c) a scheduler that repeatedly takes the task with the earliest deadline; (d) "all orders placed between 9:00 and 9:15".

SOLUTION

- Hash table.** Exact lookups by key, no ordering needed.
- Balanced tree**, augmented so each node stores the size of its subtree. Rank is then a single root-to-leaf walk, adding the sizes of the left subtrees passed; a skip list does the same job in Redis's sorted sets.
- A heap** (chapter 3), if all you need is the earliest. A balanced tree also works, and wins if you also need to cancel or reschedule arbitrary tasks.
- Balanced tree** (or a B-tree index in a database). A range query is one descent to 9:00, then an in-order walk until 9:15. A hash table would have to check every order.

CHECKLIST

- ☐ Can you explain why a plain BST degenerates, and on which inputs?
- ☐ Can you perform a left and a right rotation on paper?
- ☐ Can you name the four AVL rebalancing cases and the rotations each needs?
- ☐ Do you know the height bounds of AVL and red-black trees?
- ☐ Do you know which problems need a tree rather than a hash table?

GO DEEPER

BOOK SITE [Balanced search trees Sedgewick & Wayne, Algorithms](#)

REFERENCE [Red-black trees in Linux Linux kernel documentation](#)

PAPER [Skip lists: a probabilistic alternative to balanced trees William Pugh, 1990](#)

PART II

Sorting

Two chapters on the most studied problem in computing. First the comparison sorts, which no algorithm can push below a hard mathematical floor, and how they differ in everything else; then the sorts that never compare at all, and run in linear time by knowing something about their keys.

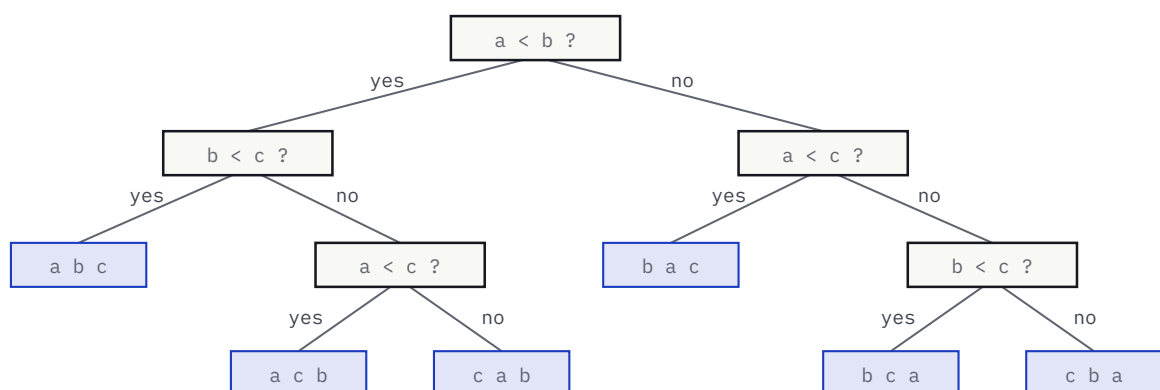
Comparison sorts, and when each wins

3 exercises

There is no best sorting algorithm, only the best one for a given input and requirement. No sort that works by comparing items can beat about $n \log n$ comparisons, which is a mathematical floor, not a lack of cleverness. Above that floor, sorts differ in how they handle sorted, reversed and repetitive input, whether they keep equal items in their original order, and how much memory they need. Production libraries combine several to get the best of each.

The floor under every comparison sort

A comparison sort learns about the order only by asking "is this item smaller than that one?" Picture every possible run of the algorithm as a tree of those questions, with a yes branch and a no branch at each one. Each leaf is a final ordering, and since any of the $n!$ orderings of n distinct items might be the right one, the tree needs at least $n!$ leaves. A binary tree with $n!$ leaves has height at least $\log_2(n!)$, so some input must cost that many comparisons. By Stirling's approximation, $\log_2(n!)$ is about $n \log_2 n - 1.44n$, which for 2,000 items is 19,053.



6 orderings need 6 leaves, so some path asks at least $\log_2(6) \approx 2.6$, really 3, questions

Every comparison sort is a decision tree like this one. Each question at best halves the orderings still possible, so separating $n!$ orderings takes at least $\log_2(n!)$ questions on some input, whatever the algorithm.

Seven sorts, four inputs

The example implements the classic algorithms and counts the comparisons each makes on 2,000 items in four arrangements: random, already sorted, reversed, and drawn from only ten distinct values.

EXAMPLE sorting.py · seven sorts on four kinds of input, and a stability test

```
import functools, math, random

C = 0
def less(a, b):
    global C
    C += 1
    return a < b

def insertion_sort(a):
    for i in range(1, len(a)):
        j = i
        while j > 0 and less(a[j], a[j - 1]):          # slide each item left into place
            a[j], a[j - 1] = a[j - 1], a[j]
            j -= 1

def merge_sort(a):
    if len(a) <= 1: return a
    mid = len(a) // 2
    left, right = merge_sort(a[:mid]), merge_sort(a[mid:])
    out, i, j = [], 0, 0
    while i < len(left) and j < len(right):
        if less(right[j], left[i]): out.append(right[j]); j += 1
        else: out.append(left[i]); i += 1              # ties take the left item: stable
    return out + left[i:] + right[j:]

def quick_sort(a, pivot_rule):
    stack = [(0, len(a) - 1)]                          # an explicit stack: no recursion
    limit
    while stack:
        lo, hi = stack.pop()
        if lo >= hi: continue
        p = pivot_rule(lo, hi)
        a[p], a[hi] = a[hi], a[p]
        store = lo
        for i in range(lo, hi):                        # partition around the pivot at a[hi]
            if less(a[i], a[hi]):
                a[i], a[store] = a[store], a[i]; store += 1
        a[store], a[hi] = a[hi], a[store]
        stack += [(lo, store - 1), (store + 1, hi)]

def quick_sort_3way(a, rng):
    stack = [(0, len(a) - 1)]
    while stack:
        lo, hi = stack.pop()
        if lo >= hi: continue
        pivot = a[rng.randint(lo, hi)]
        lt, i, gt = lo, lo, hi                         # < pivot | == pivot | unknown | >
        pivot
        while i <= gt:
            if less(a[i], pivot): a[lt], a[i] = a[i], a[lt]; lt += 1; i += 1
            elif less(pivot, a[i]): a[i], a[gt] = a[gt], a[i]; gt -= 1
            else: i += 1
        stack += [(lo, lt - 1), (gt + 1, hi)]           # the equal block is already in place

def heap_sort(a):
    n = len(a)
    def down(i, end):
        while True:
            l, r, big = 2 * i + 1, 2 * i + 2, i
            if l < end and less(a[big], a[l]): big = l
            if r < end and less(a[big], a[r]): big = r
            if big == i: return
            a[i], a[big] = a[big], a[i]; i = big
```

```

for i in range(n // 2 - 1, -1, -1): down(i, n)           # max-heap
for end in range(n - 1, 0, -1):
    a[0], a[end] = a[end], a[0]; down(0, end)          # move the max to the back

def timsort(a):
    def compare(x, y):                                # one call is one comparison
        global C
        C += 1
        return (x > y) - (x < y)
    a.sort(key=functools.cmp_to_key(compare))

N = 2000
random.seed(5)
inputs = {"random": random.sample(range(10**6), N), "sorted": list(range(N)),
          "reversed": list(range(N, 0, -1)), "10 values": [random.randrange(10) for _ in
          range(N)]}
rng = random.Random(9)
algorithms = {
    "insertion": insertion_sort,
    "merge": lambda a: a.__setitem__(slice(None), merge_sort(a)),
    "quick, first pivot": lambda a: quick_sort(a, lambda lo, hi: lo),
    "quick, random pivot": lambda a: quick_sort(a, lambda lo, hi: rng.randint(lo, hi)),
    "quick, 3-way": lambda a: quick_sort_3way(a, rng),
    "heap": heap_sort,
    "Timsort (sorted())": timsort,
}
print(f"comparisons to sort {N:,} items (lower bound for any comparison sort: log2({N}!) =
      {math.lgamma(N + 1) / math.log(2):,.0f})")
print(f"{'':22}" + "".join(f"{k:>12}" for k in inputs))
for name, sort in algorithms.items():
    row = []
    for data in inputs.values():
        a = list(data); C = 0
        sort(a)
        assert a == sorted(data)
        row.append(C)
    print(f"{name:22}" + "".join(f"{c:>12,}" for c in row))

class Rec:
    """A record compared by its key only, so records with equal keys are ties to the sort."""
    def __init__(self, key, pos): self.key, self.pos = key, pos
    def __lt__(self, other): return self.key < other.key
def keeps_order(recs):
    return all(a.pos < b.pos for a, b in zip(recs, recs[1:])) if a.key == b.key

recs = [Rec(random.randrange(5), i) for i in range(30)]
by_merge = merge_sort(list(recs))
by_heap = list(recs); heap_sort(by_heap)
print(f"\nequal keys keep their original order? merge: {keeps_order(by_merge)} "
      f"heap: {keeps_order(by_heap)} sorted(): {keeps_order(sorted(recs))}")

```

OUTPUT

```

comparisons to sort 2,000 items (lower bound for any comparison sort: log2(2000!) = 19,053)

```

	random	sorted	reversed	10 values
insertion	1,014,248	1,999	1,999,000	883,754
merge	19,422	10,864	11,088	18,803
quick, first pivot	23,845	1,999,000	1,999,000	205,652
quick, random pivot	23,182	27,573	24,797	208,800
quick, 3-way	37,460	39,061	38,206	13,057
heap	37,789	39,159	35,964	35,302
Timsort (sorted())	19,303	1,999	1,999	13,822

```

equal keys keep their original order? merge: True heap: False sorted(): True

```

Insertion sort slides each item left into place. On random input it's quadratic, a million comparisons, and on reversed input two million, but on sorted input it makes one comparison per item and stops: 1,999. That makes it the fastest sort for tiny or nearly sorted arrays, which is why every hybrid sort uses it for small pieces.

Merge sort splits the list in half, sorts each half, and merges them. It's close to the floor on random input, 19,422 against 19,053, never worse than $n \log n$, and stable, but it needs a second array as large as the input. Its merging of sorted runs is also the basis of external sorting, where data is too large for memory.

Quicksort partitions around a pivot and recurses on each side, in place. Its weakness is the pivot. Always taking the first item is quadratic on sorted and reversed input, 1,999,000 comparisons, exactly $n(n-1)/2$, because every partition peels off one item. A random pivot makes that vanishingly unlikely, at around 23,000 to 27,000 comparisons on any order. But the same row shows a second weakness: with only ten distinct values, a two-way partition puts all the equal items on one side, and it climbs back to 208,800. **Three-way partitioning** groups the items equal to the pivot in the middle and never touches them again, which brings the duplicates case down to 13,057. It pays for that with two comparisons per item on distinct keys, about 37,500.

Heapsort (chapter 3) is the steadiest: about twice the floor on every input, in place, with an $O(n \log n)$ worst case. It's slow in practice because its jumps through the array defeat the cache, and it isn't stable.

Timsort, which Tim Peters designed for Python in 2002, is a merge sort that first looks for runs of data already in order, ascending or descending, then merges them carefully. On random input it lands near the floor, 19,303. On sorted and reversed input it finds one long run and finishes in $n-1$ comparisons, and it does well with duplicates too. It's stable, and it's the sort behind Python's `sorted()`, Java's sort for objects since Java 7, and JavaScript's `Array.prototype.sort` in V8.

Stability

A sort is **stable** if items that compare equal keep their original relative order. The last line tests it with records sorted by a key that has only five values: merge sort and Python's `sorted()` preserve the original order within each key, and heapsort scrambles it. Stability matters whenever you sort records by one field and care about the order from a previous sort: a table sorted by date, then by customer, shows each customer's orders still in date order only if the second sort is stable. It also makes multi-key sorting simple. Sort by the least important key first, then by the next, each with a stable sort (exercise 5.3), which is exactly how the radix sort in chapter 6 works.

What libraries actually do

No single algorithm wins every column, so standard libraries combine them. **Introsort**, the usual C++ `std::sort`, runs quicksort, switches to heapsort if the recursion gets suspiciously deep, and finishes small pieces with insertion sort: quicksort's speed,

heapsort's worst case, insertion sort's small-array efficiency. **Pattern-defeating quicksort**, used by Go's sort package since version 1.19 and long used for Rust's unstable sort, adds detection of sorted and repetitive input. Stable library sorts are typically Timsort or a relative of it. The practical rule follows. Use the library sort unless you have measured a reason not to, know whether it's stable, and reach for a specialised algorithm only for a specific need: a partial sort for the top k , quickselect for a median, an external merge sort for data larger than memory.

COMMON TRAPS

Quicksort with a fixed pivot on data that's often already sorted.

Assuming a sort is stable: C++'s `std::sort` isn't; `std::stable_sort` is.

Comparators that aren't consistent, such as comparisons involving NaN, or rules that aren't transitive, which produce wrong orders or errors.

Sorting everything to find the top few, when a heap or selection algorithm does it in linear time.

Two-way partitioning on data with many duplicates, which drifts back towards quadratic.

EXERCISES

EXERCISE 5.1 Where $n \log n$ comes from

Show that $\log_2(n!)$ grows like $n \log_2 n$ without using Stirling's formula, by bounding $n!$ from above and below.

SOLUTION

Upper bound: $n! = 1 \times 2 \times \dots \times n \leq n^n$, so $\log_2(n!) \leq n \log_2 n$. Lower bound: the largest $n/2$ factors of $n!$ are each at least $n/2$, so $n! \geq (n/2)^{n/2}$, and $\log_2(n!) \geq (n/2) \log_2(n/2)$, which is also proportional to $n \log n$. Squeezed between two multiples of $n \log n$, $\log_2(n!)$ is $\Theta(n \log n)$. So the decision-tree argument shows every comparison sort needs $\Omega(n \log n)$ comparisons in the worst case, and merge sort, heapsort and Timsort meet that bound up to a constant.

EXERCISE 5.2 A comparator that lies

A list of floats contains a NaN. What does `sorted()` do with it, and why can't any comparison sort do much better?

SOLUTION

Every comparison with NaN is false: `nan < 3` and `3 < nan` are both false, so to the sort NaN looks "equal" to everything while the numbers around it aren't equal to each other. The ordering is no longer consistent, and the sort's correctness argument, which assumes a consistent ordering, no longer holds. Python returns a list that may be only partly sorted, silently, with the NaN wherever the merges happened to leave it. Java detects some such inconsistencies and throws "Comparison method violates its general contract!". The fix is in the data, not the algorithm: remove NaNs first, or sort with a key that places them explicitly, such as `key=lambda x: (math.isnan(x), x)`.

EXERCISE 5.3 Two keys with stable sorts

Sort employees by department, and within each department by salary from highest to lowest, using only stable single-key sorts.

SOLUTION

Sort by the secondary key first, then by the primary key:

```
staff.sort(key=lambda e: e.salary, reverse=True), then
staff.sort(key=lambda e: e.department) .
```

The second sort groups by department, and because it's stable, each department's employees keep the salary order the first sort gave them. Python guarantees that `reverse=True` keeps the sort stable, so equal salaries keep their original order too. A single sort with a tuple key, `key=lambda e: (e.department, -e.salary)`, does the same in one pass when the secondary key can be negated. The two-pass version works for any keys, which is the general technique.

CHECKLIST

- ☐ Can you explain the decision-tree argument for the $n \log n$ floor?
- ☐ Can you implement insertion, merge, quick (with three-way partitioning) and heap sort?
- ☐ Do you know which inputs make each sort fast or slow?
- ☐ Do you know whether your library's sort is stable?
- ☐ Do you reach for selection or a heap when you don't need a full sort?

GO DEEPER

SOURCE [listsort.txt: Timsort explained by its author Tim Peters](#)

BOOK SITE [Sorting Sedgewick & Wayne, Algorithms](#)

PAPER [Pattern-defeating quicksort Orson Peters](#)

Sorting without comparing

3 exercises

The $n \log n$ floor applies only to sorts that learn the order by comparing pairs of items. When keys are small integers, or can be cut into digits, you can put each item straight into its place and sort in linear time. The price is assumptions about the keys, extra memory, and a reliance on stability that is easy to break and hard to notice.

Counting sort

If every key is an integer between 0 and $k-1$, you don't need to compare anything. Count how many items have each key, turn the counts into starting positions with a running total, and place each item at its key's next free position. That's two passes over the items and one over the counts: $n + k$ steps, linear when k isn't much larger than n . Placing items in their original order makes it **stable**, a property the next section depends on entirely.

EXAMPLE `nocompare.py` · counting, radix and bucket sort, and what an unstable pass does

```
import random

def counting_sort(items, key, k):
    """Stable sort of items whose keys are integers in [0, k): n + k steps, no comparisons."""
    counts = [0] * k
    for x in items: counts[key(x)] += 1
    total = 0
    for v in range(k):
        # prefix sums: where each key's run
        # starts
        counts[v], total = total, total + counts[v]
    out = [None] * len(items)
    for x in items:
        # walk forwards: equal keys keep
        # their order
        out[counts[key(x)]] = x
        counts[key(x)] += 1
    return out, len(items) * 2 + k

def radix_sort(nums, digit_pass=counting_sort):
    """LSD radix sort of 32-bit integers, one byte at a time, least significant first."""
    work = 0
    for shift in (0, 8, 16, 24):
        nums, w = digit_pass(nums, lambda x: (x >> shift) & 0xFF, 256)
        work += w
    return nums, work

def unstable_pass(items, key, k):
    """The same buckets, filled back to front: still groups by digit, but reverses equal
    keys."""
    counts = [0] * k
    for x in items: counts[key(x)] += 1
    total = 0
    for v in range(k): counts[v], total = total, total + counts[v]
    ends = [counts[v + 1] if v + 1 < k else len(items) for v in range(k)]
    out = [None] * len(items)
    for x in items:
```

```

        ends[key(x)] -= 1; out[ends[key(x)]] = x
    return out, len(items) * 2 + k

C = 0
def merge_sort(a):
    global C
    if len(a) <= 1: return a
    l, r = merge_sort(a[:len(a) // 2]), merge_sort(a[len(a) // 2:])
    out, i, j = [], 0, 0
    while i < len(l) and j < len(r):
        C += 1
        if r[j] < l[i]: out.append(r[j]); j += 1
        else: out.append(l[i]); i += 1
    return out + l[i:] + r[j:]

random.seed(6)
N = 100_000
nums = [random.getrandbits(32) for _ in range(N)]
result, work = radix_sort(nums)
merged = merge_sort(nums)
print(f"sorting {N:,} random 32-bit integers:")
print(f"    merge sort          {C:>10,} comparisons")
print(f"    LSD radix sort      {work:>10,} steps (4 passes of n + 256), no comparisons; correct: {result == sorted(nums)}")

ages = [random.randrange(120) for _ in range(N)]
out, work = counting_sort(ages, key=lambda x: x, k=120)
print(f"    counting sort of {N:,} ages 0-119: {work:,} steps; correct: {out == sorted(ages)}")
print(f"    counting sort of 64-bit ids would need a count array of 2^64 entries: "
      f"{(2**64 * 8) / 2**60:,.0f} EiB of memory")

floats = [random.random() for _ in range(N)]
buckets = [[] for _ in range(N)]
for f in floats: buckets[int(f * N)].append(f)          # uniform data: about one item per
    bucket
longest = max(len(b) for b in buckets)
out = [f for b in buckets for f in sorted(b)]
print(f"    bucket sort of {N:,} uniform floats: longest bucket {longest}; correct: {out == sorted(floats)}")

broken, _ = radix_sort(nums, digit_pass=unstable_pass)
first_bad = next(i for i in range(N - 1) if broken[i] > broken[i + 1])
print(f"\nradix sort with an unstable digit pass: sorted? {broken == sorted(nums)} "
      f"(first out-of-order pair at position {first_bad:,})")

```

OUTPUT

```

sorting 100,000 random 32-bit integers:
    merge sort          1,536,586 comparisons
    LSD radix sort      801,024 steps (4 passes of n + 256), no comparisons; correct: True
    counting sort of 100,000 ages 0-119: 200,120 steps; correct: True
    counting sort of 64-bit ids would need a count array of 2^64 entries: 128 EiB of memory
    bucket sort of 100,000 uniform floats: longest bucket 9; correct: True

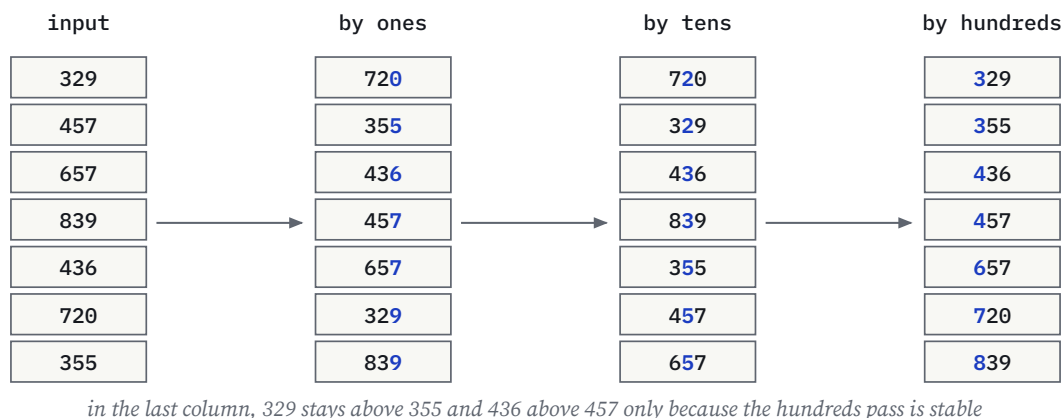
radix sort with an unstable digit pass: sorted? False (first out-of-order pair at position 1)

```

Sorting 100,000 ages between 0 and 119 takes 200,120 steps, with no comparisons at all. The next line shows the limit. The count array has one slot per possible key, so sorting 64-bit ids directly would need 2^{64} counters, 128 exbibytes of memory. Counting sort is for small key ranges: ages, grades, status codes, bytes.

Radix sort

Radix sort extends counting sort to large keys by sorting them one digit at a time, using a stable counting sort for each digit. The least significant digit (LSD) version starts from the ones digit and works up. It looks backwards, but it works, and the diagram shows why.



LSD radix sort, one stable pass per digit. After the tens pass, numbers with the same tens digit are still in the order the ones pass left them, so ties on the higher digit are already broken correctly by the lower ones. Stability is what carries each pass's work forward.

Take 329 and 355. They share a hundreds digit, so the hundreds pass can't tell them apart: a stable pass leaves them in the order they arrived in. They arrived in the right order because the tens pass sorted them by their tens digits, 2 before 5. So the final order of two numbers that tie on a digit is decided by the earlier passes, and it survives only if every later pass is stable. That's the whole correctness argument for LSD radix sort.

The example sorts 100,000 random 32-bit integers in four passes of one byte each, 801,024 steps against merge sort's 1,536,586 comparisons, and the gap widens with n , since radix sort is linear and merge sort isn't. The last line is the warning. Swap in a digit pass that groups by digit correctly but reverses the order of equal digits, and the output is wrong from position 1 onwards, with no error raised. The bug is silent: every pass looks correct on its own.

Bucket sort

Bucket sort assumes the keys are spread evenly over a known range. It splits the range into n buckets, drops each item into its bucket in one pass, sorts each tiny bucket, and concatenates them. With 100,000 uniformly random floats and 100,000 buckets, the fullest bucket held 9 items, so the whole sort is linear in expectation. The assumption carries all the weight. If the data clusters, say most values near zero, most items land in a few buckets, and the sort degrades to whatever sorts those buckets.

When the floor doesn't apply

None of this contradicts chapter 5. The $n \log n$ bound counts comparisons, and these sorts don't compare: they use the structure of the keys, which a comparison sort isn't allowed to see. That's also why libraries rarely use them by default. A general-purpose sort can't know that your keys are small integers or uniformly spread. Where those facts are known, they win clearly: GPU libraries sort with radix sort, databases sort fixed-width keys with it, suffix-array construction depends on it, and counting sort is the inner loop of many of them.

COMMON TRAPS

Counting sort on a large key range, trading time for an impossible amount of memory.

An unstable digit pass inside a radix sort, which produces wrong output without any error.

Signed integers in a radix sort, where negative numbers sort after positive ones unless the sign bit is handled.

Bucket sort on skewed data, which piles everything into a few buckets.

LSD radix on variable-length strings without padding, or without switching to most-significant-digit order.

EXERCISES

EXERCISE 6.1 Ten million status codes

Ten million records each carry a 2-byte status code. Compare the cost of a counting sort by status with a comparison sort.

SOLUTION

A 2-byte code has $k = 65,536$ possible values, so counting sort costs about $2n + k = 20,065,536$ steps, plus a count array of 65,536 integers, half a megabyte. A comparison sort needs at least $\log_2(10^7!) \approx 10^7 \times (23.25 - 1.44) \approx 218$ million comparisons. So counting sort does about ten times less work here. It also needs a second array for the output, and it's stable, so records with the same status keep their previous order.

EXERCISE 6.2 Why stable passes

Using the diagram's numbers, show what goes wrong if the hundreds pass is unstable. Then explain why, for these particular seven numbers, an unstable tens pass would have done no harm.

SOLUTION

After the tens pass, 329 comes before 355 (tens digits 2 and 5) and 436 before 457 (3 and 5). In the hundreds pass each pair ties, on 3 and on 4. A stable pass keeps them as they arrived, and the output is sorted. An unstable pass is free to reverse either pair and could output 355, 329, 457, 436, ..., an unsorted result, with no error raised. An unstable tens pass, by contrast, can only reorder numbers that tie on their tens digit: 720 and 329, 436 and 839, or 355, 457 and 657. Every one of those groups has different hundreds digits, so the hundreds pass would put them right anyway. The rule is about ties: a pass must keep, among items that tie on its digit, the order the earlier passes established. With other data, such as 425 and 427 tying on both tens and hundreds, an unstable tens pass would break the result just the same.

EXERCISE 6.3 Negative numbers

The example sorts unsigned 32-bit integers. What goes wrong with two's-complement signed integers, and what's the smallest fix?

SOLUTION

In two's complement, negative numbers have their top bit set, so as unsigned values they're larger than every positive number, and a plain LSD radix sort places them after all the positives (internally in the right order, since -1 is the largest of them). The fix is to flip the sign bit before sorting, or equivalently to treat the most significant byte's digit as `byte ^ 0x80` in the last pass: negatives then map to 0–127 and positives to 128–255, and the order comes out right. Floating-point keys take a similar trick, flipping all the bits of negative numbers and only the sign bit of positive ones, which is described in Michael Herf's "Radix Tricks".

CHECKLIST

- ☐ Can you implement a stable counting sort with prefix sums?
- ☐ Can you explain why LSD radix sort needs stable passes?
- ☐ Do you know the key-range and distribution assumptions behind each linear sort?

- ☐ Can you handle signed keys in a radix sort?
- ☐ Can you say why the $n \log n$ bound doesn't apply to these sorts?

GO DEEPER

BOOK SITE String sorts: key-indexed counting, LSD and MSD radix sort Sedgewick & Wayne, Algorithms

ARTICLE Radix tricks Michael Herf

BOOK Introduction to Algorithms, chapter 8: sorting in linear time Cormen, Leiserson, Rivest & Stein

PART III

Beyond exact & efficient

Two chapters on what to do when the usual promises break. First the structures that give up exact answers to answer questions about billions of items in a few kilobytes; then the problems for which no efficient exact algorithm is known at all, how to recognise them, and what to build instead.

Probabilistic structures: trading certainty for memory

3 exercises

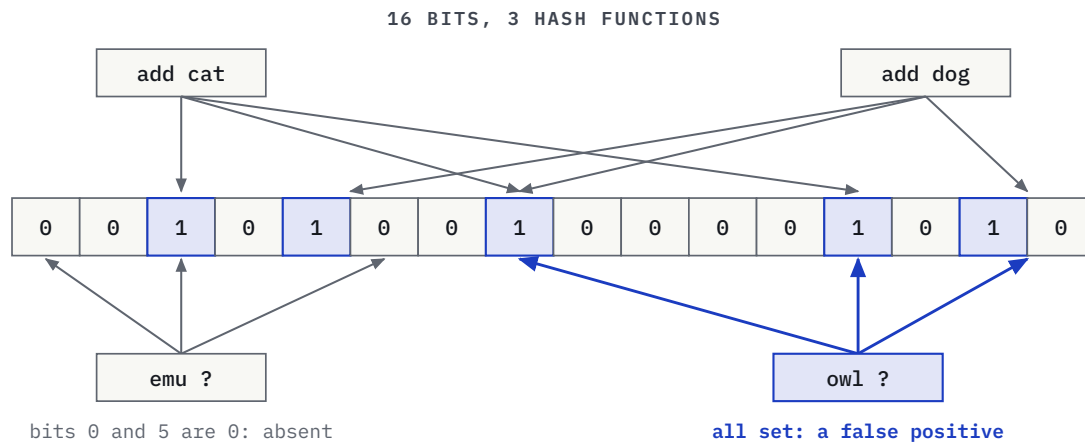
Exact answers about a huge stream cost memory in proportion to everything you've seen. A Bloom filter, a count-min sketch and a HyperLogLog answer three common questions, "have I seen this?", "how often?" and "how many distinct?", in a small, fixed amount of memory, by accepting an error you choose in advance. Each error has a known shape: a Bloom filter only ever says yes wrongly, a count-min sketch only ever overcounts, and a HyperLogLog is off by a predictable percentage. All three can be merged across machines, which is why they're everywhere in databases, networks and analytics.

The price of exact

Answering "have I seen this user before?" exactly means storing every user. Answering "how many distinct visitors today?" exactly means remembering every visitor until midnight. At a few hundred million items that's gigabytes per question, per machine. Hashing offers a way out. A good hash function makes any item look like a random number, and random numbers have statistics you can rely on. Each structure in this chapter stores only those statistics, in an amount of memory set by the error you'll accept rather than by how much data arrives.

Bloom filters: membership with one-sided error

A **Bloom filter**, invented by Burton Bloom in 1970, is an array of m bits, all zero, and k hash functions. To add an item, compute its k positions and set those bits to 1. To query an item, check the same k positions. If any of them is 0, the item was definitely never added, because adding it would have set that bit. If all of them are 1, the item was probably added, or other items happened to set all k bits between them, which is a **false positive**. There are never false negatives.



Adding sets bits; querying only reads them. Emu hits two bits nobody set, so it was certainly never added. Owl was never added either, but cat and dog between them set all three of its bits, so the filter answers "probably", wrongly. A zero bit is proof; a set bit is only evidence.

After n insertions, the chance that a given bit is still 0 is about $e^{-kn/m}$, so the false-positive rate is about $(1 - e^{-kn/m})^k$. With too few hash functions each query checks too little; with too many the array fills with ones. The best choice is $k = (m/n) \ln 2$, which leaves half the bits set. At that k , the bits you need depend only on the error rate, not on the size of the items: about $1.44 \log_2(1/p)$ bits per key, which is 9.6 bits for 1% and 14.4 bits for 0.1%. Each tenfold improvement costs another 4.8 bits per key. Implementations don't compute k independent hashes. Adam Kirsch and Michael Mitzenmacher showed that two hashes combined as $h_1 + i \cdot h_2$ work as well, which is what the example does with one blake2b digest split in half.

The classic use is guarding an expensive lookup that usually fails. LSM-tree databases such as RocksDB and Cassandra keep a Bloom filter for each file on disk, so a read skips every file that can't contain the key. Akamai's CDN used one to cache an object only on its second request, because most objects are requested exactly once, as Bruce Maggs and Ramesh Sitaraman describe in "Algorithmic nuggets in content delivery". In both cases a false positive costs one wasted lookup, and a false negative, which never happens, would have cost a wrong answer.

Count-min sketch: frequencies that only overcount

A **count-min sketch**, from Graham Cormode and S. Muthukrishnan in 2005, is d rows of w counters, each row with its own hash function. Adding an item increments one counter in each row. The estimate for an item is the smallest of its d counters. Every counter the item touches holds its true count plus whatever else hashed to the same place, so each one is an overestimate, and the minimum is the least wrong. With $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, the estimate exceeds the true count by at most ϵN , where N is the total number of events, with probability at least $1 - \delta$.

That bound is relative to the whole stream, not to the item's own count. An error of ϵN is negligible for an item seen thousands of times and enormous for one seen twice. So the count-min sketch is a tool for **heavy hitters**: the most frequent source addresses on a network link, trending search terms, the noisiest API keys for rate limiting. Combined with a small heap of the current leaders (chapter 3), it answers "what are the top 100?" over a stream it never stores.

HyperLogLog: counting distinct items in kilobytes

Start with coins. If someone tells you the longest run of heads they saw before a tail was 10, they probably flipped about 2^{10} times, since a run that long is a one-in-a-thousand event. Hash every item to a random-looking 64-bit number and look at its leading zero bits: a hash starting with $r - 1$ zeros and then a one turns up with probability 2^{-r} , so the largest such r seen estimates $\log_2$ of the number of distinct items. Duplicates hash identically and can't raise the maximum, which is exactly why the estimate counts *distinct* items.

A single maximum is noisy, since one lucky hash could claim a billion items. **HyperLogLog**, from Philippe Flajolet and colleagues in 2007, splits the hashes among m registers using their first p bits, keeps the largest rank seen in each register, and combines the registers with a harmonic mean, which damps the occasional huge value. Its standard error is $1.04/\sqrt{m}$: 3.25% with 1,024 registers, 0.81% with 16,384. A rank never exceeds 64, so a register needs only 6 bits, and 16,384 registers fit in 12 KB, which is what Redis's `PFCOUNT` uses. The example spends a whole byte per register for clarity. For small counts, when many registers are still empty, it switches to **linear counting**, an estimate from the number of empty registers that's more accurate in that range.

All three, measured

The example builds all three from the definitions above and measures them against exact answers: a Bloom filter at two sizes, a count-min sketch over a skewed stream in which a few items are very common, and HyperLogLog at two register counts.

EXAMPLE

sketches.py · a Bloom filter, a count-min sketch and HyperLogLog, measured against exact answers

```
import hashlib, math, random

def h128(item):
    d = hashlib.blake2b(str(item).encode(), digest_size=16).digest()
    return int.from_bytes(d[:8], "little"), int.from_bytes(d[8:], "little") | 1

class BloomFilter:
    """m bits and k hash functions: 'definitely not present' or 'probably present'."""
    def __init__(self, m, k):
        self.m, self.k, self.bits = m, k, bytearray((m + 7) // 8)
    def _positions(self, item):
        a, b = h128(item)
        return [(a + i * b) % self.m for i in range(self.k)]  # k hashes from two (Kirsch-Mitzenmacher)
```

```

def add(self, item):
    for p in self._positions(item): self.bits[p // 8] |= 1 << (p % 8)
def __contains__(self, item):
    return all(self.bits[p // 8] >> (p % 8) & 1 for p in self._positions(item))

class CountMin:
    """d rows of w counters; an item's estimate is its smallest counter, never an
    undercount."""
    def __init__(self, eps, delta):
        self.w, self.d = math.ceil(math.e / eps), math.ceil(math.log(1 / delta))
        self.rows = [[0] * self.w for _ in range(self.d)]
    def _cols(self, item):
        a, b = h128(item)
        return [(a + r * b) % self.w for r in range(self.d)]
    def add(self, item):
        for r, c in enumerate(self._cols(item)): self.rows[r][c] += 1
    def estimate(self, item):
        return min(self.rows[r][c] for r, c in enumerate(self._cols(item)))

class HyperLogLog:
    """Estimates how many distinct items it has seen from the longest runs of zero bits in
    their hashes."""
    def __init__(self, p):
        self.p, self.m, self.reg = p, 1 << p, bytearray(1 << p)
    def add_hash(self, x):
        j = x >> (64 - self.p) # the first p bits pick a
            register
        rest = (x << self.p) & ((1 << 64) - 1)
        rank = 64 - self.p + 1 if rest == 0 else (64 - rest.bit_length()) + 1 # leading zeros
            + 1
        self.reg[j] = max(self.reg[j], rank)
    def count(self):
        alpha = 0.7213 / (1 + 1.079 / self.m)
        raw = alpha * self.m * self.m / sum(2.0 ** -r for r in self.reg)
        zeros = self.reg.count(0)
        if raw <= 2.5 * self.m and zeros: # small-range correction: linear
            counting
            return self.m * math.log(self.m / zeros)
        return raw

random.seed(7)
members = [f"user-{i}" for i in range(10_000)]
others = [f"visitor-{i}" for i in range(100_000)]
print("Bloom filter, 10,000 members, 100,000 non-members tested:")
for bits_per_key in (8, 12):
    m = bits_per_key * len(members)
    k = round(bits_per_key * math.log(2)) # the k that minimises false
        positives
    bf = BloomFilter(m, k)
    for x in members: bf.add(x)
    assert all(x in bf for x in members) # no false negatives, ever
    fp = sum(x in bf for x in others) / len(others)
    theory = (1 - math.exp(-k * len(members) / m)) ** k
    print(f"    {bits_per_key:2} bits per key, k={k}: false positives {fp:.2%} (formula
        {theory:.2%}), "
          f"{m // 8:,} bytes")

ranks = list(range(1, 10_001))
weights = [1 / r for r in ranks] # a few items are very common
stream = random.choices(ranks, weights, k=200_000)
truth = {}
for x in stream: truth[x] = truth.get(x, 0) + 1
cm = CountMin(eps=0.001, delta=0.01)
for x in stream: cm.add(x)
errors = [cm.estimate(x) - truth[x] for x in truth]
print(f"count-min sketch ({cm.d} x {cm.w} = {cm.d * cm.w:,} counters) over {len(stream):,}")

```

```

events, {len(truth):,} items:")
print(f"    item 1: true {truth[1]:,}, estimate {cm.estimate(1):,}    item 500: true
      {truth[500]}, estimate {cm.estimate(500)}")
print(f"    never undercounts: {min(errors) >= 0}; worst overcount {max(errors)} (bound eps*N =
      {0.001 * len(stream):.0f}); "
      f"{sum(e <= 0.001 * len(stream) for e in errors) / len(errors):.1%} of items within it")

stream = [random.randrange(300_000) for _ in range(1_000_000)]
true_distinct = len(set(stream))
sketches = {p: HyperLogLog(p) for p in (10, 14)}
for x in stream:
    a, _ = h128(x)
    for s in sketches.values(): s.add_hash(a)
print(f"HyperLogLog over {len(stream):,} events with {true_distinct:,} distinct values:")
for p, s in sketches.items():
    est = s.count()
    print(f"    {s.m:6,} registers ({s.m:,} bytes): estimate {est:,.0f}, error {abs(est -
      true_distinct) / true_distinct:.2%} "
          f"(typical error 1.04/sqrt(m) = {1.04 / math.sqrt(s.m):.2%})")

```

OUTPUT

```

Bloom filter, 10,000 members, 100,000 non-members tested:
    8 bits per key, k=6: false positives 2.14% (formula 2.16%), 10,000 bytes
    12 bits per key, k=8: false positives 0.28% (formula 0.31%), 15,000 bytes
count-min sketch (5 x 2719 = 13,595 counters) over 200,000 events, 9,653 items:
    item 1: true 20,345, estimate 20,350    item 500: true 38, estimate 46
    never undercounts: True; worst overcount 96 (bound eps*N = 200); 100.0% of items within it
HyperLogLog over 1,000,000 events with 289,162 distinct values:
    1,024 registers (1,024 bytes): estimate 283,435, error 1.98% (typical error 1.04/sqrt(m) =
      3.25%)
    16,384 registers (16,384 bytes): estimate 289,029, error 0.05% (typical error 1.04/sqrt(m) =
      0.81%)

```

The Bloom filter matches the formula closely. At 8 bits per key with $k = 6$ it measured 2.14% false positives against 2.16% predicted; at 12 bits per key with $k = 8$, 0.28% against 0.31%. Every one of the 10,000 members was found. The whole filter is 10,000 bytes, one byte per key, for keys that are 6 to 9 characters long before any hash table overhead.

The count-min sketch shows the shape of its error. The most common item, seen 20,345 times, was estimated at 20,350, off by 0.02%. Item 500, seen 38 times, was estimated at 46, off by 21%. The worst overcount among all 9,653 items was 96, inside the bound of 200, and nothing was undercounted. It does this in 13,595 counters, whatever the length of the stream.

HyperLogLog estimated 289,162 distinct values to within 1.98% using 1,024 registers and 0.05% using 16,384. The second run was luckier than its typical 0.81%, because the standard error describes a spread, not a limit. An exact count would have to store all 289,162 values, at least 2.3 MB at 8 bytes each before any table overhead, and that grows with the data. The sketch stays at 16 KB, or 12 KB packed, at a billion distinct values.

Choosing, sizing and merging

Have I seen x?	Bloom filter	False positives only, at a rate set by bits per key	OR of the bit arrays
How often has x appeared?	Count-min sketch	Overcounts only, by at most ϵN with probability $1 - \delta$	Adding the counters
How many distinct items?	HyperLogLog	About $\pm 1.04/\sqrt{m}$, in either direction	Maximum of each register

Merging matters as much as memory. Two sketches with the same size and hash functions, built on different machines or on different days, combine into exactly the sketch of the combined data. That's how analytics systems count distinct values across shards without moving the values: BigQuery's `APPROX_COUNT_DISTINCT` and Trino's `approx_distinct` are HyperLogLog. Exact sets or counts would have to be shipped in full.

Size for the peak, not the average. The false-positive formula assumed you knew n . A filter sized for one number of keys and then given twice as many is no longer the filter you designed (exercise 7.1). A sketch sized for today's traffic needs headroom for next year's.

COMMON TRAPS

Treating "probably present" as "present". Use a Bloom filter to skip work, then confirm with the real lookup when the answer matters.

Deleting by clearing bits, which creates false negatives for every other item that shared them.

Overfilling a filter: past its design size, the false-positive rate climbs steeply.

Merging sketches built with different sizes or hash seeds, which silently produces nonsense.

Reading a count-min estimate for a rare item as accurate. The bound is relative to the whole stream.

Using unkeyed hashes on adversarial input. Someone who knows the hash can craft items that set chosen bits or inflate chosen counters, the attack from chapter 2; use a keyed hash with a secret seed.

EXERCISES

EXERCISE 7.1 Sizing a filter for a billion URLs

A crawler wants to skip URLs it has already fetched: a billion of them, with at most 1% false positives. How much memory does the Bloom filter need, and how many hash functions? What happens to the false-positive rate if two billion URLs go into the same filter?

SOLUTION

Bits per key = $-\ln(0.01) / (\ln 2)^2 \approx 9.59$, so the filter needs 9.59 billion bits, about 1.2 GB, with $k = 9.59 \times \ln 2 \approx 6.6$, rounded to 7 hash functions. Storing the URLs themselves, at even 50 bytes each, would take 50 GB. With two billion URLs in the same filter there are only 4.79 bits per key, and the rate becomes $(1 - e^{-7/4.79})^7 \approx 15.7\%$: doubling the load made false positives about sixteen times as common. Size for the largest n you expect, or use a scalable Bloom filter, which adds a new, larger filter whenever the current one fills.

EXERCISE 7.2 Taking things out

Why can't you delete an item from a Bloom filter by clearing its k bits? What can you use instead if you need deletions?

SOLUTION

Bits are shared. In the figure, clearing cat's bits would clear bit 7, which dog also set, and the filter would then report dog as absent: a false negative, which breaks the one guarantee a Bloom filter makes. There are three common alternatives. A **counting Bloom filter** replaces each bit with a small counter, typically 4 bits, incremented on add and decremented on delete, at four times the memory. A **cuckoo filter** stores short fingerprints in a cuckoo hash table and supports deletion, often in less space than a Bloom filter at low false-positive rates, but deleting an item that was never added can remove another item's fingerprint. Or you can avoid deletion entirely by rotating filters: keep one per hour, query the recent ones, and drop whole filters as they expire.

EXERCISE 7.3 Unique visitors this week

You keep one HyperLogLog sketch of visitor IDs per day. How do you get the number of distinct visitors over the last seven days, and why can't you add up the seven daily counts? Can you also estimate how many visitors came on both Monday and Tuesday?

SOLUTION

Take the maximum of each register across the seven sketches. The result is exactly the sketch you would have built from the whole week's stream, so its estimate is the week's distinct count, with the usual error. Adding the daily counts counts a visitor who came on five days five times. For the overlap, inclusion-exclusion gives $|\text{Mon} \cap \text{Tue}| = |\text{Mon}| + |\text{Tue}| - |\text{Mon} \cup \text{Tue}|$, and every term can be estimated. But each term's error is a percentage of a large number, so the error of the difference is a percentage of the union, not of the overlap. A small overlap between two big sets can come out wildly wrong or even negative. For accurate intersections, sketches designed for them, such as MinHash or theta sketches, do much better.

CHECKLIST

- ☐ Can you implement a Bloom filter and size it from n and a target false-positive rate?
- ☐ Can you explain why a Bloom filter has no false negatives, and why you can't delete from it?
- ☐ Can you state the count-min error bound, and say which items it's accurate for?
- ☐ Can you explain HyperLogLog's leading-zero idea and its $1.04/\sqrt{m}$ error?
- ☐ Do you know how each structure merges, and why that matters across machines?

GO DEEPER

- | | |
|--------------|---|
| PAPER | Space/time trade-offs in hash coding with allowable errors Burton H. Bloom, 1970 |
| <hr/> | |
| PAPER | An improved data stream summary: the count-min sketch Cormode & Muthukrishnan |
| <hr/> | |
| PAPER | HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm Flajolet, Fusy, Gandouet & Meunier |

NP-completeness: when no fast algorithm is known

3 exercises

Some problems have no known algorithm much better than trying every possibility, and thousands of them are linked so tightly that a fast algorithm for any one would give a fast algorithm for all. Fifty years of searching hasn't found one. Recognising such a problem is a practical skill. It stops you looking for an algorithm that probably doesn't exist, and points you at what works instead: small inputs, special structure, approximation with a guarantee, measured heuristics, or a solver.

Easy to check, hard to find

Take **subset sum**: given a list of numbers and a target, is there a subset that adds up to exactly the target? If someone hands you a subset, checking it takes a few additions.

Finding one is another matter. The obvious method tries all 2^n subsets, and nobody knows a method that is polynomial in the size of the input for every instance.

That asymmetry defines the class. **P** is the set of problems that can be solved in polynomial time. **NP** is the set of yes-or-no problems whose yes-answers come with a **certificate**, like the subset, that can be checked in polynomial time. The name means "nondeterministic polynomial": a machine that could guess the right certificate would finish in polynomial time. Every problem in **P** is also in **NP**, since you can check an answer by solving the problem yourself. Whether **P** equals **NP** is the most famous open question in computer science. Stephen Cook posed it in 1971, it's one of the Clay Millennium Prize problems, and most researchers believe the answer is no.

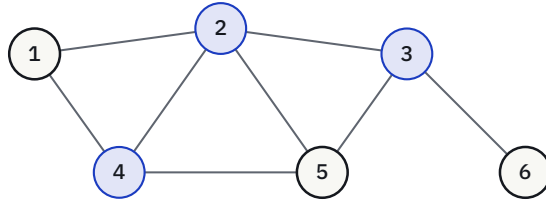
Reductions: why one problem stands for thousands

A **reduction** from problem A to problem B is a polynomial-time translation of any instance of A into an instance of B with the same answer. Given one, a fast algorithm for B would give a fast algorithm for A, so B is at least as hard as A. Cook, and independently Leonid Levin, proved that Boolean satisfiability (SAT) is as hard as everything in **NP**: any **NP** problem can be translated into a formula that's satisfiable exactly when the answer is yes. In 1972 Richard Karp reduced SAT to 21 classic problems, among them vertex cover, clique, Hamiltonian cycle, graph colouring and a version of subset sum. Garey and Johnson's 1979 book catalogued hundreds more, and there are now thousands. A problem is **NP-hard** if everything in **NP** reduces to it, and **NP-complete** if it's also in **NP** itself.



so vertex cover is at least as hard as independent set

THE SAME GRAPH, $n = 6$, $k = 3$



○ {1, 5, 6}: independent set
no edge joins two of them

● {2, 3, 4}: vertex cover
every edge touches one

A reduction translates the question, not the answer. This one is almost free: a set of nodes is independent exactly when the nodes left over touch every edge, so G has an independent set of k nodes exactly when it has a vertex cover of $n - k$. Any vertex-cover solver therefore answers independent-set questions too.

Direction is the classic mistake. To show that your new problem X is hard, reduce a known NP-complete problem *to* X : "if I could solve X quickly, I could solve vertex cover quickly." Reducing X to SAT shows only that X is no harder than SAT. That's useful too, in a different way, because it means a SAT solver can solve X .

Feeling the wall

The example makes the wall concrete. It brute-forces subset sum on instances with no solution, which is the worst case because every subset has to be tried. It checks a certificate, solves a larger instance by dynamic programming, solves a 10-city travelling salesperson problem exactly and with two heuristics, and compares an approximation for vertex cover against the true optimum.

EXAMPLE np.py · exponential search, cheap verification, and three ways around the wall

```

import itertools, math, random

random.seed(8)
# 1. Subset sum: checking an answer is fast; finding one, in general, means trying subsets.
def brute_force(nums, target):
    """Sum of every subset, built incrementally: 2^n subsets, each one addition."""
    sums = [0] * (1 << len(nums))
    for mask in range(1, 1 << len(nums)):
        low = mask & -mask
        sums[mask] = sums[mask ^ low] + nums[low.bit_length() - 1]
        if sums[mask] == target:
            return mask, mask
    return None, len(sums) - 1

print("subset sum, no subset hits the target (the worst case), subsets checked:")
for n in (10, 14, 18, 20):
    nums = [random.randrange(1, 10**9) * 2 for _ in range(n)] # even numbers can never sum to
    an odd target
    found, checked = brute_force(nums, 1)
  
```

```

    print(f"    n = {n:2}: {checked:>17,} subsets examined, answer found: {found is not None}")
print(f"    n = 40: {2**40 - 1:>17,} subsets: each extra number doubles the work")

nums = [random.randrange(1, 1000) for _ in range(30)]
certificate = random.sample(range(30), 12)
target = sum(nums[i] for i in certificate)
print(f"verifying a claimed answer for n = 30 takes {len(certificate)} additions: "
      f"{sum(nums[i] for i in certificate) == target} (brute force could need 2^30 = {2**30:,} subsets)")

reachable, work = {0}, 0
for x in nums:
    sums, not subsets
    work += len(reachable)
    reachable |= {s + x for s in reachable if s + x <= target}
print(f"dynamic programming with small numbers (all below 1,000): target {target:,} reachable: "
      f"{target in reachable}, {work:,} steps (bound: 30 x {target + 1:,} sums = {30 * (target + 1):,})")
print(f"the same idea with 40-bit numbers would need tables of about 2^40 = {2**40:,} cells")

# 2. Travelling salesperson: exact by brute force, then two fast heuristics.
pts = [(random.random(), random.random()) for _ in range(10)]
def length(tour): return sum(math.dist(pts[tour[i]], pts[tour[i - 1]]) for i in
    range(len(tour)))
best = min(((0,) + p for p in itertools.permutations(range(1, 10))), key=length)
nn, left = [0], set(range(1, 10))
while left:
    to the closest city
    nxt = min(left, key=lambda c: math.dist(pts[nn[-1]], pts[c])); nn.append(nxt);
    left.remove(nxt)
tour, improved = nn[:], True
while improved:
    that shortens the tour
    improved = False
    for i in range(1, len(tour) - 1):
        for j in range(i + 1, len(tour)):
            new = tour[:i] + tour[i:j + 1][::-1] + tour[j + 1:]
            if length(new) < length(tour) - 1e-12:
                tour, improved = new, True
print(f"\ntravelling salesperson, 10 cities ({math.factorial(9):,} tours tried for the exact answer):")
print(f"    optimal {length(best):.3f}    nearest neighbour {length(nn):.3f} ({length(nn) / length(best) - 1:.0%}) "
      f"after 2-opt {length(tour):.3f} ({length(tour) / length(best) - 1:.0%}")

# 3. Vertex cover: an approximation with a guarantee.
nodes = range(14)
edges = random.sample([(a, b) for a in nodes for b in nodes if a < b], 25)
def covers(s): return all(a in s or b in s for a, b in edges)
exact = min((set(c) for r in range(15) for c in itertools.combinations(nodes, r) if
    covers(set(c))), key=len)
approx = set()
for a, b in edges:
    still uncovered
    if a not in approx and b not in approx:
        approx |= {a, b}
print(f"\nvertex cover, 14 nodes and 25 edges: optimal {len(exact)} nodes (found by checking all 2^14 = {2**14:,} subsets)")
print(f"    matching-based approximation: {len(approx)} nodes, covers every edge: {covers(approx)}, "
      f"within 2x of optimal: {len(approx) <= 2 * len(exact)}")

```

OUTPUT

subset sum, no subset hits the target (the worst case), subsets checked:

```

n = 10:          1,023 subsets examined, answer found: False
n = 14:          16,383 subsets examined, answer found: False
n = 18:          262,143 subsets examined, answer found: False
n = 20:          1,048,575 subsets examined, answer found: False
n = 40: 1,099,511,627,775 subsets: each extra number doubles the work
verifying a claimed answer for n = 30 takes 12 additions: True (brute force could need  $2^{30} = 1,073,741,824$  subsets)
dynamic programming with small numbers (all below 1,000): target 6,575 reachable: True, 102,145
steps (bound:  $30 \times 6,576$  sums = 197,280)
the same idea with 40-bit numbers would need tables of about  $2^{40} = 1,099,511,627,776$  cells

travelling salesperson, 10 cities (362,880 tours tried for the exact answer):
optimal 3.071 nearest neighbour 3.201 (+4%) after 2-opt 3.108 (+1%)

vertex cover, 14 nodes and 25 edges: optimal 8 nodes (found by checking all  $2^{14} = 16,384$ 
subsets)
matching-based approximation: 10 nodes, covers every edge: True, within 2x of optimal: True

```

Each extra number doubles the brute-force work. Twenty numbers mean a million subsets, which is nothing. Forty mean a trillion, which compiled code can still grind through. Sixty mean a million times more again. Checking a claimed answer for 30 numbers took 12 additions, against a search space of more than a billion subsets. That gap between finding and checking is NP in one line of output.

What to do instead

Check the size first. Exponential doesn't automatically mean too slow. If your instances have 20 items, brute force over a million subsets is the simplest correct solution. With pruning (backtracking, branch and bound), brute force often reaches much further in practice than its worst case suggests.

Exploit small numbers. The dynamic program tracks which sums are reachable instead of which subsets exist. For 30 numbers below 1,000 it decided reachability of 6,575 in 102,145 steps. That's polynomial in the *value* of the target, not in its length in bits, which makes it **pseudo-polynomial**. With 40-bit numbers the table would need about 2^{40} cells and the trick is gone. Problems like subset sum and knapsack are called weakly NP-complete for this reason: they're practical whenever the numbers involved are small.

Exploit structure. NP-hardness is a statement about the worst instances, and yours may avoid them. Vertex cover is easy on trees and on bipartite graphs, where by König's theorem it equals a maximum matching. 2-SAT is solvable in linear time though 3-SAT is NP-complete. Colouring an interval graph, such as assigning meetings to rooms, is solved by a greedy algorithm. Some problems are hard only in one parameter: a vertex cover of size k can be found in about $2^k \cdot n$ steps by branching on the two ends of an uncovered edge, which is fast for a huge graph with a small cover.

Approximate with a guarantee. The vertex-cover algorithm in the example takes both ends of any uncovered edge. It found 10 nodes where the optimum is 8, and it can never be worse than twice the optimum (exercise 8.3). For travelling salesperson with real distances, which obey the triangle inequality, Christofides' algorithm guarantees a tour within 1.5 times the

optimum, an approximation ratio improved in 2020 by a famously tiny margin. Some problems resist even this: without the triangle inequality, no polynomial algorithm can approximate travelling salesperson within any constant factor unless $P = NP$.

Use heuristics, and measure them. Nearest neighbour built a tour 4% longer than the optimum; 2-opt, which reverses any segment that shortens the tour, brought it to 1%. Neither guarantees anything in general, but local search of this kind routinely gets within a few percent on real instances. Measure a heuristic against exact answers on small instances, or against a lower bound on large ones, before you trust it.

Hand it to a solver. SAT solvers and mixed-integer programming solvers routinely solve industrial instances with millions of variables, in hardware verification, scheduling and logistics. Their worst case is still exponential, but real instances have structure that decades of solver engineering exploit. Translating your problem into SAT or an integer program, which is itself a reduction, and running Google's OR-Tools, the open-source HiGHS or a commercial solver like Gurobi often beats a hand-written algorithm by a wide margin.

Hard problems in disguise

NP-hard problems rarely announce themselves. Choosing one version of each package so that every dependency constraint holds is NP-complete: Russ Cox showed how to encode 3-SAT in it. That's why Go modules chose minimal version selection, which sidesteps the general problem by design, while Dart's pub and Python's uv use PubGrub, a resolver built on SAT-solving techniques. Scheduling exams so no student has a clash is graph colouring, and so is register allocation in a compiler. Placing containers onto servers is bin packing. Choosing which features fit a release budget is knapsack. Planning a delivery route is travelling salesperson. When a problem asks you to choose a subset, an assignment or an ordering that satisfies constraints or optimises a total, suspect NP-hardness before you spend a week looking for a clever polynomial algorithm.

COMMON TRAPS

Reading NP as "not polynomial". It means nondeterministic polynomial, and every problem in P is in NP.

Reducing in the wrong direction: to prove X hard, reduce a known hard problem to X, not X to it.

Calling a pseudo-polynomial algorithm polynomial because its running time is a product of two input numbers.

Giving up because a problem is NP-hard, when your instances are small, structured, or well within a solver's reach.

Presenting a heuristic's answer as optimal, or reporting it without a bound or a comparison.

Assuming a problem is hard because it looks hard. Shortest paths, minimum spanning trees, bipartite matching, 2-SAT and linear programming are all polynomial.

EXERCISES

EXERCISE 8.1 Exams without clashes

Your university wants to know whether all its exams fit into k time slots so that no student has two exams at once. Show that this problem is NP-hard, using the fact that deciding whether a graph can be coloured with k colours is NP-complete. Which way does the reduction go?

SOLUTION

Reduce graph colouring, the known hard problem, to exam scheduling, the new one. Given any graph, make one exam per node and, for each edge, one student who takes the two exams at its ends. A clash-free timetable with k slots assigns each exam a slot so that the two ends of every edge get different slots, which is exactly a k -colouring, and every k -colouring gives such a timetable. The translation takes time proportional to the size of the graph. So a fast exam scheduler would colour arbitrary graphs fast, and exam scheduling is at least as hard as colouring. Going the other way, turning timetables into graphs, would prove only that scheduling is no harder than colouring. That's true, and useful for using a colouring heuristic, but it proves nothing about hardness.

EXERCISE 8.2 A polynomial algorithm for an NP-complete problem?

The subset-sum dynamic program runs in $O(n \cdot T)$ time, where T is the target. Subset sum is NP-complete. Why isn't this a proof that $P = NP$?

SOLUTION

"Polynomial" means polynomial in the size of the input, measured in bits. A target T takes only about $\log_2 T$ bits to write down, so with b -bit numbers T can be as large as 2^b , and $O(n \cdot T)$ is $O(n \cdot 2^b)$: exponential in the input length. Adding one bit to every number doubles the running time while barely growing the input. The algorithm is pseudo-polynomial, fast only when the numbers are small. Some problems stay NP-hard even when all their numbers are small, such as bin packing and 3-partition; these are called strongly NP-hard, and no pseudo-polynomial algorithm for them exists unless $P = NP$.

EXERCISE 8.3 Why the approximation is never more than twice optimal

The vertex-cover approximation walks the edges and, whenever an edge has neither end in the cover, adds both ends. Prove that the result is a vertex cover, and that it's at most twice the size of the smallest one.

SOLUTION

It's a cover because every edge is examined, and after an edge is examined at least one of its ends is in the set: either it already was, or the algorithm just added both. For the bound, look at the edges that triggered additions. No two of them share an end, because once an edge's ends are added, any later edge touching either of them is already covered. They form a **matching** M , and the algorithm's cover has exactly $2|M|$ nodes. Any vertex cover, including the optimal one, must contain at least one end of each edge in M , and since those edges share no ends, those are $|M|$ different nodes. So the optimum is at least $|M|$, and the algorithm's cover is at most twice the optimum. In the example, the algorithm matched 5 edges and took 10 nodes, so the optimum had to be at least 5; it was 8.

CHECKLIST

- ☐ Can you define P, NP, NP-hard and NP-complete, and say what a certificate is?
- ☐ Do you know which direction a reduction goes to prove a problem hard?
- ☐ Can you spot common NP-hard problems in disguise?
- ☐ Can you explain why the subset-sum dynamic program isn't polynomial?
- ☐ Can you name the ways to cope: small inputs, structure, approximation, heuristics and solvers?

GO DEEPER

BOOK Algorithms, chapter 12: NP-hardness Jeff Erickson

ARTICLE Version SAT Russ Cox

COURSE 6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs Erik Demaine, MIT

THE LIBRARY CHECKLIST

The stage's deliverable is a library implementing every structure from scratch, with tests, and with each method's complexity documented. This is the list. Build each structure, test it against Python's built-in equivalent on random input as the examples in this volume do, and write the cost of each method in its docstring, including what makes the worst case happen. The last column says where each one is built, in this volume or in Sorted Again.

STRUCTURE	OPERATIONS AND COST	BUILT IN
Dynamic array	append $O(1)$ amortised · get/set by index $O(1)$ · insert or delete in the middle $O(n)$	Ch. 1
Ring buffer and deque	push and pop at either end $O(1)$, amortised if it grows	Ch. 1
Linked list	insert or delete at a known node $O(1)$ · search $O(n)$	Sorted Again
Stack and queue	push, pop, enqueue, dequeue $O(1)$	Ch. 1, Sorted Again
Hash map and hash set	get, put, delete $O(1)$ expected, $O(n)$ worst · resize $O(n)$, amortised $O(1)$	Ch. 2
Binary heap	push and pop $O(\log n)$ · peek $O(1)$ · build from n items $O(n)$	Ch. 3
Balanced search tree (AVL or red-black)	search, insert, delete, min, max, successor $O(\log n)$ · in-order walk $O(n)$	Ch. 4
Trie	insert, search, prefix search $O(\text{length of the key})$	Sorted Again
Graph (adjacency list)	BFS, DFS, topological sort $O(V + E)$ · Dijkstra $O((V + E) \log V)$ · A* depends on the heuristic	Sorted Again, Ch. 3
Union-find	find and union with path compression and union by rank, $O(\alpha(n))$ amortised, effectively constant	Sorted Again
Comparison sorts	insertion $O(n^2)$ · merge $O(n \log n)$ · quick, three-way, $O(n \log n)$ expected · heap $O(n \log n)$	Ch. 5
Counting and radix sort	counting $O(n + k)$ for k possible keys · LSD radix $O(d(n + b))$ for d digits in base b	Ch. 6

Bloom filter	add, query $O(k) \cdot \text{size from } n$ and the target false-positive rate	Ch. 7
Count-min sketch	add, estimate $O(d) \cdot \text{size from } \varepsilon$ and δ	Ch. 7
HyperLogLog	add $O(1) \cdot \text{count}$ and merge $O(m)$	Ch. 7

Then solve the problems. Sorted Again's 150 problems, with their tests, are the second half of the deliverable; for each one you fail, write down which pattern you missed, as the stage asks.

THE EIGHT RULES

One per chapter, each linked back to it. Read them before you choose a data structure for anything that will grow, and again when something that was fast in testing is slow in production.

- 01 Grow by a factor, never by a constant, and shrink later than you grow; that is what makes appending amortised constant time.
 - 02 A hash table is only as fast as its hash and its load factor; resize before it fills, and use a keyed hash when other people choose the keys.
 - 03 When you only need the next item in order, use a heap, build it in one pass, and give A* an estimate that never overestimates.
 - 04 An unbalanced search tree is a linked list waiting for sorted input; balance it, or use a library that does.
 - 05 Use the library sort, know whether it's stable, and don't sort everything to find the top few.
 - 06 Linear-time sorts buy their speed with assumptions about the keys; check the range and the distribution before you rely on them.
 - 07 When exact answers cost too much memory, choose the error you can live with, size the sketch for your peak, and confirm a "probably" before acting on it.
 - 08 Before hunting for a fast algorithm, ask whether the problem is NP-hard; if it is, use small inputs, structure, approximation, measured heuristics or a solver.
-

READING LIST

The books, courses and papers behind this volume, and the chapters each one deepens. If you read only one, make it Skiena's *Algorithm Design Manual*, which is also the stage's main book. If you do only one thing, build the library above and test it against the built-ins, because a structure you have built is one you never have to take on trust.

WORK	AUTHOR	DEEPENS
Sorted Again	this series	The patterns
The Algorithm Design Manual, 3rd edition	Steven Skiena	Ch. 1–8
MIT 6.006 Introduction to Algorithms	Demaine, Ku & Solomon	Ch. 1–6
Introduction to Algorithms, 4th edition	Cormen, Leiserson, Rivest & Stein	Ch. 1–6, 8
Algorithms, 4th edition	Sedgewick & Wayne	Ch. 1–6
Algorithms	Jeff Erickson	Ch. 5, 8
Algorithms Illuminated (the Stanford course)	Tim Roughgarden	Ch. 2–3, 5, 7–8
Computers and Intractability	Garey & Johnson	Ch. 8
NeetCode 150	NeetCode	Practice

Compiled October 2026 as the supplement to stage 02, Data structures and algorithms, of The Engineer's Syllabus, alongside *The Engineer's Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel*, *Foundry*, *Exchange* and *Ledger*, and as the second half of a pair with *Sorted Again*, which teaches the problem-solving patterns. It draws on the books in the reading list; on the implementations themselves, CPython's list and dict, Tim Peters' notes on Timsort, PEP 456 and Abseil's Swiss tables; and on the original papers: Hart, Nilsson and Raphael on A* (1968), Pugh on skip lists (1990), Bloom (1970), Kirsch and Mitzenmacher on double hashing in Bloom filters (2006), Cormode and Muthukrishnan (2005), Flajolet, Fussy, Gandouet and Meunier (2007), Cook (1971), Karp (1972), Maggs and Sitaraman on content delivery (2015), and Russ Cox on version selection. Adoption details, such as which library uses which sort or hash, are late-2026 facts used to teach method, not facts to rely on.

Every example was executed before publication, in pure-standard-library Python, and every one counts operations rather than time, so it prints the same numbers on every machine. Re-run in October 2026 with Python 3.14.7, every example printed exactly the output shown. Tested here: dynamic array growth policies, shrinking thresholds and a ring buffer; chaining and linear probing against Knuth's formulas, tombstones and a hash-flooding attack; heap construction, Dijkstra against A* on an open grid and a maze; unbalanced and AVL trees on sorted and random input; seven comparison sorts on four inputs, with the $n \log n$ floor and a stability test; counting and LSD radix sort against merge sort; a Bloom filter, a count-min sketch and HyperLogLog against exact answers; and subset sum, travelling salesperson and vertex cover, exactly and approximately.