FIT2004

Algorithms and Data Structures

Weeks 1 – 3 Study Notes

★ PT1 Examinable

Sections marked ★ PT1 Examinable are examined in Pass Test 1.

PT1 question types:

```
(A) Derive a recurrence relation from pseudocode
(B) Solve a recurrence relation (telescope + verify)
(C) Prove a loop invariant (initialisation + maintenance)
(D) Run an algorithm by hand (e.g. Dutch National Flag partition)
(E) Perform sorting by hand (Counting Sort or Radix Sort)
```

Week 1 · Analysis of Algorithms

Algorithm analysis answers two questions: is the algorithm correct, and how many resources does it use?

1.1 Program Verification

Correctness requires proving two things:

- The algorithm always produces the correct result when it terminates.
- The algorithm always terminates.

The primary tool for proving correctness is the loop invariant — a statement about the algorithm's state that is true before, during, and after a loop.

Three stages of a loop invariant proof

Initialisation: Prove the invariant holds before the first iteration of the loop.
Maintenance: Prove that if the invariant holds at the start of iteration i, it still holds at the start of iteration i + 1. Use a fresh letter (e.g. i) — never use n or k which already appear elsewhere.
Termination: Prove that when the loop exits, the invariant implies the result is correct.

★ PT1 Examinable

nly. Termination only if the

Pages: 17 · Words: 4,490
Last page (End of Document) removed

Common errors (from the feedback guide):

- Taking the variable value at the wrong point (beginning vs end of iteration).
- Not comparing the computed value to what the invariant says it should be.
- Writing 'Assume the invariant holds at iteration n' — must use a fresh variable like i.
- Not explicitly invoking the induction hypothesis in the maintenance step.
- Indexing errors (off-by-one on array bounds).

Example: Binary Search

```
function BINARY_SEARCH(array[1..n], key)
  lo = 1 and hi = n + 1
  while lo < hi - 1 do
    mid = floor((lo + hi) / 2)
    if key >= array[mid] then lo = mid
    else hi = mid
  if array[lo] = key then return lo
  else return null
```

Invariant: Binary Search

If key is in array, then at each iteration:

1. array[lo] ≤ key
2. if hi ≠ n + 1, then array[hi] > key

Together these imply key (if it exists) lies within array[lo..hi−1].

Proof sketch:

Initialisation: lo = 1, hi = n+1. array[lo] is the minimum element so ≤ key. hi = n+1 so condition 2 is vacuously true.

Maintenance: if key ≥ array[mid], setting lo = mid preserves condition 1. If key < array[mid], setting hi = mid preserves condition 2.

Termination: loop exits when lo ≥ hi−1, combined with the invariant forces lo = hi−1, so key (if present) must be at array[lo].

Terminates because: lo < mid < hi always holds when lo < hi−1, so the interval [lo..hi) shrinks by at least 1 each iteration.

Case	Time	Aux. Space
Best	O(1)	O(1)
Worst	O(log n)	O(1)
Average	O(log n)	O(1)

1.2 Complexity Analysis — Recurrence Relations

The time complexity T(n) of a recursive algorithm is expressed as a recurrence relation. We solve it by telescoping.

★ PT1 Examinable

(A) Deriving a recurrence: state T(n) = [cost of recursive calls] + [non-recursive work], with base case.

- Use T, not the function name.
- Coefficient = number of recursive calls. Argument = size passed to each call.
- Non-recursive work: constant c if no loop; cn if there is a loop over n elements.
- Specify that c is a constant (e.g. 'where c is a positive constant').
- Don't forget the base case — and get the condition right (≤ vs <, = 1 vs ≤ 1).

(B) Solving a recurrence (telescoping method):

Step 1 Write T(n) = … then expand T(n-1) or T(n/2) etc. two or three times.

Step 2 Identify the pattern and write the general form T(n) = T(n-k) + f(k).

Step 3 Choose k to reach the base case (e.g. k=n, or $k=\log_2 n$).

Step 4 Substitute T(base) = b to get a closed-form expression.

Step 5 Verify: substitute back into the original recurrence for both base and general case.

Common errors: algebraic errors in step 1; stopping before eliminating T; declaring closed form when T still appears; no verification; verifying only one case.

1.2.1 Common Recurrences You Must Know

Recurrence	Closed form	Complexity	Example algorithm
T(n)=T(n/2)+c, T(1)=b	$a \log_2 n + b$	$\Theta(\log n)$	Binary search
T(n)=T(n−1)+c, T(0)=b	$an + b$	$\Theta(n)$	Linear search
T(n)=2T(n/2)+cn, T(1)=b	$an \log n + bn$	$\Theta(n \log n)$	Merge sort
T(n)=T(n−1)+cn, T(0)=b	$c \cdot n(n+1)/2 + b$	$\Theta(n^2)$	Selection sort (worst)
T(n)=2T(n−1)+a, T(0)=b	$(a+b) \cdot 2^n - a$	$\Theta(2^n)$	Naive Fibonacci
T(n)=3T(n/2)+cn, T(1)=b	$O(n^{\log_2 3})$	$\Theta(n^{1.585})$	Karatsuba

Worked example — solving T(n) = 2T(n/2) + cn

Telescope:

$$T(n) = 2T(n/2) + cn$$
$$= 4T(n/4) + cn + cn = 4T(n/4) + 2cn$$
$$= 8T(n/8) + 3cn$$
$$= 2^k T(n/2^k) + k \cdot cn$$

Set $k = \log_2 n$ to reach $T(1) = b$:

$$T(n) = 2^{\wedge}(\log_2 n) \cdot b + cn \cdot \log_2 n = nb + cn \log n$$

Verify base case (n=1): $b + 0 = b = T(1)$. ✓

Verify general case: $2T(n/2) + cn = 2[(n/2)b + c(n/2)\log(n/2)] + cn = nb + cn(\log n - 1) + cn = nb + cn \log n = T(n)$. ✓

1.3 Asymptotic Notation

Big-O (upper bound)

$f(n) = O(g(n))$ iff $\exists$ constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Informally: f grows no faster than g. Big-O bounds can be overestimates.

Example: $2n + 1 = O(n)$ and also $2n + 1 = O(n^3)$ (overestimate but still valid).

Big-Ω (lower bound)

$f(n) = \Omega(g(n))$ iff $\exists$ constants $c > 0$ and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Informally: f grows at least as fast as g. Equivalent to $g(n) = O(f(n))$.

Big-Θ (tight bound)

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$.

Informally: f and g have the same order of magnitude.

Example: $n^2 + 3n + 5 = \Theta(n^2)$. But $n^2 + 3n + 5 \neq \Theta(n^3)$ and $\neq \Theta(n)$.

1.4 Measures of Complexity

Best, worst, and average case

Best case: fewest operations over all inputs of size n. Worst case: most operations. Average case: average over all inputs (usually the hardest to compute). Unless stated otherwise, worst case is assumed.

Space complexity vs auxiliary space complexity

Space complexity: total memory used by the algorithm including the input.

Auxiliary space complexity: memory used excluding the input.

In-place algorithm

An algorithm is in-place if its auxiliary space complexity is O(1).

It may only use a constant amount of extra memory beyond the input itself.

Note: some books have a slightly looser definition — this course uses the O(1) definition.

1.5 Selection Sort

```
function SELECTION_SORT(array[1..n])
  for i = 1 to n do
    min = i
    for j = i+1 to n do
      if array[j] < array[min] then min = j
    swap(array[i], array[min])
```

Worked example

Trace on array [5, 3, 8, 1, 4]:

5	3	8	1	4

i=1: min found at index 4 (value 1). Swap arr[1] and arr[4]:

1	3	8	5	4

i=2: min found at index 2 (value 3). Already in place (no swap):

1	**3**	8	5	4

i=3: min found at index 5 (value 4). Swap arr[3] and arr[5]:

1	3	**4**	5	8

i=4: min found at index 4 (value 5). Already in place. Final sorted array:

1	3	4	5	8

★ PT1 Examinable

(C) Selection Sort Invariant:

```
   At the beginning of iteration i, the following two conditions
both hold:
   1.  array[1..i-1] is sorted.
   2.  Every element of array[1..i-1] is ≤ every element of
array[i..n].
```

Initialisation (i=1): array[1..0] is empty — both conditions hold trivially.

Maintenance: At iteration i, we find min = index of smallest element in array[i..n].

```
   After swap(array[i], array[min]):
   • array[i] = min of array[i..n].  Since array[i-1] ≤ all of
old array[i..n] (by IH), and
     array[i] is the minimum of that range, array[i-1] ≤
array[i], so array[1..i] is sorted.
```

• array[i] ≤ all of array[i+1..n] (since it was the minimum), so condition 2 holds.

Both invariants hold for i+1.

Case	Time	Aux. Space	Stable?	In-place?
Best	$O(n^2)$	$O(1)$	No	Yes
Average	$O(n^2)$	$O(1)$	No	Yes
Worst	$O(n^2)$	$O(1)$	No	Yes

Selection sort: not stable (equal elements can be reordered by swaps), not online.

1.5 Insertion Sort

```
function INSERTION_SORT(array[1..n])
  for i = 2 to n do
    key = array[i]
    j = i - 1
    while j > 0 and array[j] > key do
      array[j+1] = array[j]
      j = j - 1
    array[j+1] = key
```

Worked example

Trace on array [5, 3, 8, 1, 4]:

5	3	8	1	4

i=2: key=3. Shift 5 right, insert 3 at position 1:

3	**5**	8	1	4

i=3: key=8. 8 > 5, no shifts needed:

3	5	**8**	1	4

i=4: key=1. Shift 8, 5, 3 right, insert 1 at position 1:

1	**3**	**5**	**8**	4

i=5: key=4. Shift 8, 5 right, insert 4:

1	3	4	5	8

★ PT1 Examinable

(C) Insertion Sort Invariant (weaker than selection sort — only one condition):

At the beginning of iteration i, array[1..i−1] is sorted.

Initialisation (i=2): array[1..1] is a single element — trivially sorted.

Maintenance: At iteration i, key = array[i]. The while loop shifts elements rightward until array[j] ≤ key or j = 0. Since array[1..i−1] was sorted, after the while loop:

- array[1..j] is sorted (all ≤ key).
- array[j+1..i] was sorted and all > key, so placing key at j+1 keeps array[1..i] sorted.

The invariant holds for i+1.

Case	Time	Aux. Space	Stable?	In-place?
Best	O(n)	O(1)	Yes	Yes
Average	O(n²)	O(1)	Yes	Yes
Worst	O(n²)	O(1)	Yes	Yes

Best case O(n): when array is already sorted, the inner while loop never executes.

Insertion sort is: stable ✓, online ✓ (can process new elements one at a time), in-place ✓.

Stable sorting

Stable sort

A sorting algorithm is stable if elements that compare equal retain their original relative order.

Example: sorting [('Bob',75),('Alice',75),('Dan',90)] by score stably gives [('Bob',75),('Alice',75),('Dan',90)].

An unstable sort might swap Bob and Alice even though they have the same score.

Stable sorting is important when sorting by multiple criteria: sort by the secondary key first, then stable-sort by the primary key to combine them correctly.

Online vs offline

Online algorithm: can process input sequentially without seeing it all upfront. Insertion sort is online. Selection sort is not (needs to scan the whole unsorted section).

Comparison: Selection Sort vs Insertion Sort

Property	Selection Sort	Insertion Sort
Best case	O(n²)	O(n)
Worst case	O(n²)	O(n²)
Stable	No	Yes
Online	No	Yes
Invariant strength	Strong (2 conditions)	Weak (1 condition)

Week 2 · Divide and Conquer & Fast Sorting (Part 1)

Divide and conquer works in three steps: (1) Divide the problem into smaller subproblems, (2) Conquer each subproblem recursively, (3) Combine the solutions. Analysis involves solving a recurrence.

2.1 Karatsuba's Multiplication Algorithm

Grade school multiplication of two n-digit numbers takes $O(n^2)$. Can we do better?

The idea

Split x and y into high/low halves: $x = xM \cdot 10^{(n/2)} + xL$, $y = yM \cdot 10^{(n/2)} + yL$

Then $x \cdot y = xM \cdot yM \cdot 10^n + (xM \cdot yL + xL \cdot yM) \cdot 10^{(n/2)} + xL \cdot yL$

Naively: 4 recursive calls (xM·yM, xM·yL, xL·yM, xL·yL) → recurrence $T(n)=4T(n/2)+cn$ → $O(n^2)$ still!

Karatsuba's trick: compute only 3 calls:

```
(1) a = xM · yM
(2) b = xL · yL
(3) c = (xM+xL) · (yM+yL)
Then  xM·yL + xL·yM  =  c − a − b   (no extra recursion needed!)
```

This trick traces back to Gauss's method for multiplying complex numbers with 3 real multiplications.

```
function KARATSUBA(x, y)   // x, y are n-digit positive numbers
  if n = 1 then
    return x * y            // base case: single-digit multiplication
  let xM, xL = high/low halves of x
  let yM, yL = high/low halves of y
  a = KARATSUBA(xM, yM)
  b = KARATSUBA(xL, yL)
  c = KARATSUBA(xM + xL, yM + yL)
  z = c - a - b
  return a * 10^n + z * 10^(n/2) + b
```

Recurrence: $T(n) = 3T(n/2) + cn \quad \rightarrow \quad T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$

This is the algorithm Python uses internally for large integer multiplication.

2.2 Merge Sort (Revision)

Divide: split array in half. Conquer: recursively sort each half. Combine: merge the two sorted halves.

```
function MERGE_SORT(array[lo..hi])
  if hi > lo then
    mid = floor((lo + hi) / 2)
```

```
    MERGE_SORT(array[lo..mid])
    MERGE_SORT(array[mid+1..hi])
    array[lo..hi] = MERGE(array[lo..mid], array[mid+1..hi])

function MERGE(A[i..n1], B[j..n2])
  result = empty array
  while i <= n1 or j <= n2 do
    if j > n2 or (i <= n1 and A[i] <= B[j]) then   // A[i] <= B[j] makes it
stable
      result.append(A[i]);  i += 1
    else
      result.append(B[j]);  j += 1
  return result
```

Worked trace on [5, 2, 8, 1, 9, 3]

5	2	8	1	9	3

Split → [5,2,8] and [1,9,3]

[5,2,8] → [5,2] and [8] → [2,5] and [8] → merge → [2,5,8]

[1,9,3] → [1,9] and [3] → [1,9] and [3] → merge → [1,3,9]

Final merge of [2,5,8] and [1,3,9]:

1	2	3	5	8	9

Recurrence: $T(n) = 2T(n/2) + cn,\ T(1) = b \ \rightarrow \ T(n) = nb + cn \log n = \Theta(n \log n)$

Case	Time	Aux. Space	Stable?	In-place?
Best	O(n log n)	O(n)	Yes	No
Average	O(n log n)	O(n)	Yes	No
Worst	O(n log n)	O(n)	Yes	No

Merge sort is stable because the MERGE routine uses $A[i] \leq B[j]$ (not strict <), so equal elements from the left half come first.

Not in-place because MERGE creates a new output array of size n.

2.3 Counting Inversions

Problem: Counting Inversions

Input: An array A of n distinct integers.

Output: Count of pairs (i,j) where $i < j$ and $A[i] > A[j]$.

Example: [3,1,2] has 2 inversions: (3,1) and (3,2).

Brute force: $O(n^2)$. Target: O(n log n).

Key idea

Adapt Merge Sort: each recursive call sorts its half AND returns the inversion count within that half.

Split inversions (i in left half, j in right half) are counted during the merge step.

Counting split inversions during merge:

When picking from the RIGHT subarray (B[j] < A[i]):

→ all remaining elements of A are greater than B[j], so each forms an inversion with B[j].

→ add (n1 − i + 1) to the split inversion count, where n1 is the last index of A.

```
function SORT_AND_COUNTINV(array[lo..hi])
  if lo = hi then return (array[lo], 0)
  mid = floor((lo+hi)/2)
  (array[lo..mid],   InvL) = SORT_AND_COUNTINV(array[lo..mid])
  (array[mid+1..hi], InvH) = SORT_AND_COUNTINV(array[mid+1..hi])
  (array[lo..hi],    InvS) = MERGE_AND_COUNTSPLITINV(array[lo..mid],
array[mid+1..hi])
  return (array[lo..hi], InvL + InvH + InvS)

function MERGE_AND_COUNTSPLITINV(A[i..n1], B[j..n2])
  result = []; splitInv = 0
  while i <= n1 or j <= n2 do
    if j > n2 or (i <= n1 and A[i] <= B[j]) then
      result.append(A[i]);  i += 1
    else
      result.append(B[j]);  j += 1
      splitInv = splitInv + n1 - i + 1    // remaining left elements all invert
with B[j]
  return (result, splitInv)
```

Time complexity: O(n log n) — same recurrence as Merge Sort.

Week 3 · Fast Sorting Algorithms

This week covers Heapsort, Quicksort (with three partition schemes), the comparison-based sorting lower bound, Counting Sort, and Radix Sort.

3.1 Heapsort

Binary Heap

A complete binary tree (all levels full except possibly the last, filled left to right).

Max-heap property: every node is ≥ both its children → maximum element is at the root.

Stored as a flat array: root at index 1; children of node i are at 2i and 2i+1.

Operations:

HEAPIFY(array): Build a max-heap from an unsorted array in O(n) time.

```
   INSERT(array, x):    Insert x and RISE it to its correct
position. O(log n).
   EXTRACT_MAX(array):  Swap root with last element, shrink
heap, FALL root. O(log n).
   RISE(array, i):      Swap node i upward until heap property
is satisfied.
   FALL(array, i):      Swap node i downward until heap property
is satisfied.
```

```
function HEAPSORT(array[1..n])
  HEAPIFY(array[1..n])
  for i = n to 1 do
    array[i] = EXTRACT_MAX(array[1..i])

function HEAPIFY(array[1..n])
  for i = floor(n/2) to 1 do   // start from last non-leaf node
    FALL(array[1..n], i)

function EXTRACT_MAX(array[1..n])
  swap(array[1], array[n])
  n = n - 1
  FALL(array[1..n], 1)
  return array.pop_back()

function RISE(array[1..n], i)
  parent = floor(i/2)
  while parent >= 1 do
    if array[parent] < array[i] then
      swap(array[parent], array[i]);  i = parent;  parent = floor(i/2)
    else break

function FALL(array[1..n], i)
  child = 2*i
  while child <= n do
    if child < n and array[child+1] > array[child] then child += 1
    if array[i] < array[child] then
      swap(array[i], array[child]);  i = child;  child = 2*i
    else break
```

Worked example — HEAPIFY then EXTRACT

Array [3,1,4,1,5,9,2]. After HEAPIFY:

9	5	4	1	1	3	2

EXTRACT_MAX (i=7): swap arr[1] and arr[7] = [2,5,4,1,1,3,9], FALL(arr[1..6], 1):

5	2	4	1	1	3	**9**

EXTRACT_MAX (i=6): swap arr[1] and arr[6] = [3,2,4,1,1,5,9], FALL → [4,2,3,1,1,5,9]:

1	1	2	3	4	5	9

Case	Time	Aux. Space	Stable?	In-place?

Best	O(n)	O(1)	No	Yes
Average	O(n log n)	O(1)	No	Yes
Worst	O(n log n)	O(1)	No	Yes

Best case O(n) occurs only when all elements are identical (EXTRACT_MAX then requires no swaps).

3.2 Quicksort

Divide-and-conquer sort. Choose a pivot, partition the array so all elements < pivot are left and all > pivot are right, then recursively sort both sides.

```
function QUICKSORT(array[lo..hi])
  if hi > lo then
    pivot = array[lo]
    mid = PARTITION(array[lo..hi], pivot)
    QUICKSORT(array[lo..mid-1])
    QUICKSORT(array[mid+1..hi])
```

Partition Scheme 1 — Naive 3-way (out-of-place, stable)

Create three temporary arrays: LEFT (< pivot), PIVOTS (= pivot), RIGHT (> pivot). Concatenate and copy back. O(n) time, O(n) space. Stable.

Partition Scheme 2 — Hoare (in-place, not stable)

Two pointers start at lo+1 and hi. Move inward swapping elements on the wrong side. When they cross, swap pivot to its final position. O(n) time, O(1) space. Not stable. Can perform badly when many elements equal the pivot.

Partition Scheme 3 — Dutch National Flag (in-place, not stable)

★ PT1 Examinable

(D) Running the DNF Algorithm — the most commonly examined PT1 'run by hand' question.

Invariant at all times:

```
   array[1..lo-1]   contains RED   items  (<  pivot)    ← sorted section
   array[lo..mid-1] contains WHITE items  (=  pivot)    ← sorted section
   array[mid..hi]   contains UNKNOWN items              ← not yet processed
   array[hi+1..n]   contains BLUE  items  (>  pivot)    ← sorted section
```

At each step, examine array[mid]:

```
   array[mid] < pivot  (Red):   swap(array[mid], array[lo]), lo++,  mid++
```

```
   array[mid] = pivot  (White): mid++          (already in right
place)
   array[mid] > pivot  (Blue):  swap(array[mid], array[hi]),
hi--   (mid stays!)
```

Loop ends when mid > hi. Return lo, mid (boundaries of the = section).

Common errors:

- Incrementing mid after a Blue swap (wrong — array[mid] is now unknown).
- Printing array at the wrong line (print immediately after each swap/step).
- Using space allocated to multiple attempts for a single attempt.

```
function PARTITION(array[1..n], pivot)
  lo = 1,  mid = 1,  hi = n
  while mid <= hi do
    if array[mid] < pivot then          // Red case
      swap(array[mid], array[lo])
      lo += 1;  mid += 1
    else if array[mid] = pivot then     // White case
      mid += 1
    else                                // Blue case  (array[mid] > pivot)
      swap(array[mid], array[hi])
      hi -= 1                           // mid does NOT increment here!
  return lo, mid
```

Worked example — DNF with pivot = 4

Array [3, 9, 1, 4, 5, 4, 2], pivot = 4

3	9	1	4	5	4	2

lo=1, mid=1, hi=7

mid=1: arr[1]=3 < 4 → Red → swap(arr[1],arr[1])=[no change], lo=2, mid=2

3	9	1	4	5	4	2

mid=2: arr[2]=9 > 4 → Blue → swap(arr[2],arr[7])=[3,2,1,4,5,4,9], hi=6

3	**2**	1	4	5	**4**	9

mid=2: arr[2]=2 < 4 → Red → swap(arr[2],arr[2])=[no change], lo=3, mid=3

3	**2**	1	4	5	4	9

mid=3: arr[3]=1 < 4 → Red → swap(arr[3],arr[3])=[no change], lo=4, mid=4

3	2	**1**	4	5	4	9

mid=4: arr[4]=4 = 4 → White → mid=5

3	2	1	**4**	5	4	9

mid=5: arr[5]=5 > 4 → Blue → swap(arr[5],arr[6])=[3,2,1,4,4,5,9], hi=5

3	2	1	4	**4**	**5**	9

mid=5: arr[5]=4 = 4 → White → mid=6. mid(6) > hi(5) → loop ends.

3	2	1	4	4	5	9

Result: [3,2,1 | 4,4 | 5,9] — elements < 4 on left, = 4 in middle, > 4 on right. ✓

Quicksort time complexity

Case	Time	Aux. Space	Why
Best	`O(n log n)`	`O(log n)`	`Pivot always the median → balanced O(log n)-deep recursion`
Average	`O(n log n)`	`O(log n)`	`Expected O(log n) depth with random pivots (proved by coin-flip argument)`
Worst	$O(n^2)$	`O(log n)`	`Pivot always min or max → O(n) levels of recursion`

Worst-case recurrence: $T(n) = T(n-1) + cn \rightarrow \Theta(n^2)$.

Best-case recurrence: $T(n) = 2T(n/2) + cn \rightarrow \Theta(n \log n)$.

Average-case: O(n log n). Proof: define 'good pivot' as anything in the middle 50%. Good pivots occur with probability 1/2, doubling the depth to 2 log_{4/3}(n) = O(log n).

Quicksort is not in-place (by the strict O(1) definition)

In-place partitioning (Hoare or DNF) uses O(1) extra space for the partition step.
However, Quicksort requires O(log n) stack space in the best/average case for recursion,
and O(n) in the worst case. So Quicksort is not in-place by the O(1) definition.
Using the looser definition (no additional data structures beyond O(1) per level), it is in-place.

3.3 Complexity Lower Bound for Comparison-Based Sorting

Theorem: Lower bound

Any comparison-based sorting algorithm takes Ω(n log n) in the worst case.
In other words, no comparison-based sorting algorithm can be faster than O(n log n) worst-case.

Proof sketch (not examinable in Semester 1, 2026):

Model the algorithm as a decision tree where each internal node is a comparison and each leaf is one of the n! possible sorted orderings. Any correct sorting algorithm must reach every leaf, so the tree has at least n! leaves. A binary tree of height h has at most 2^h leaves. Therefore:

$$n! \leq 2^h \implies h \geq \log_2(n!)$$

Using the fact that $\log_2(n!) \geq (n/2) \cdot \log_2(n/2) = \Omega(n \log n)$, the height is at least $\Omega(n \log n)$.

3.4 Counting Sort

Assumes all elements are integers in a known range [0, u−1]. Does not compare elements, so the Ω(n log n) lower bound does not apply.

Algorithm

1. Count occurrences: counter[x] = number of times x appears in the input.
2. Compute starting positions: position[0] = 1; position[v] = position[v−1] + counter[v−1].
3. Place elements: scan input left-to-right; place array[i] at output[position[array[i]]], then increment position[array[i]].

Scanning left-to-right in step 3 is what makes the sort stable.

```
function COUNTING_SORT(array[1..n], u)
  counter[0..u-1] = [0, 0, ...]
  for i = 1 to n do
    counter[array[i]] += 1
  position[0..u-1] = [1, 0, ...]     // position[0]=1, rest=0 initially
  for v = 1 to u-1 do
    position[v] = position[v-1] + counter[v-1]
  temp[1..n] = [0, 0, ...]
  for i = 1 to n do
    temp[position[array[i]]] = array[i]
    position[array[i]] += 1
  swap(array, temp)
```

★ PT1 Examinable

(E) Counting Sort worked example — array [3,1,3,7,5,3,7,8], u=9

Step 1 Count:

```
counter: [0,1,0,3,0,1,0,2,1]    (index = value)
```

Step 2 Positions (position[v] = position[v-1] + counter[v-1]):

```
position: [1,1,2,2,5,5,6,6,8]
```

Step 3 Place left-to-right:

```
(3,a): output[position[3]=2] = (3,a),  position[3]→3
(1,p): output[position[1]=1] = (1,p),  position[1]→2
(3,c): output[position[3]=3] = (3,c),  position[3]→4
(7,f): output[position[7]=6] = (7,f),  position[7]→7
(5,g): output[position[5]=5] = (5,g),  position[5]→6
```

```
(3,b): output[position[3]=4] = (3,b),  position[3]→5
(7,d): output[position[7]=7] = (7,d),  position[7]→8
(8,w): output[position[8]=8] = (8,w),  position[8]→9
```

Result: [(1,p),(3,a),(3,c),(3,b),(5,g),(7,f),(7,d),(8,w)] — stable ✓

Case	Time	Aux. Space	Stable?	In-place?
All cases	O(n + u)	O(n + u)	Yes	No

O(n + u) time is linear in n only when u = O(n). If u is very large (e.g. sorting 2 numbers in range [0, 10^9]), counting sort is very inefficient.

3.4 Radix Sort

Sorts integers digit by digit, from least significant to most significant, using a stable sort (counting sort) for each pass.

Key idea

If we process digits from least significant (LSD) to most significant (MSD), and each pass is stable,

then elements sorted by earlier (less significant) digits retain their order when later passes tie.

If we process MSD to LSD, this does NOT work without recursive subgroups.

If each pass is NOT stable, the sort gives wrong results because earlier digit orders get destroyed.

```
function RADIX_SORT(array[1..n], base, digits)
  for digit = 1 to digits do    // 1 = least significant digit
    RADIX_PASS(array[1..n], base, digit)

function RADIX_PASS(array[1..n], base, digit)
  counter[0..base-1] = [0, 0, ...]
  for i = 1 to n do
    counter[GET_DIGIT(array[i], base, digit)] += 1
  position[0..base-1] = [1, 0, ...]
  for v = 1 to base-1 do
    position[v] = position[v-1] + counter[v-1]
  temp[1..n] = [0, 0, ...]
  for i = 1 to n do
    d = GET_DIGIT(array[i], base, digit)
    temp[position[d]] = array[i]
    position[d] += 1
  swap(array, temp)
```

★ PT1 Examinable

(E) Radix Sort worked example — array [114, 849, 909, 6034, 73, 2477, 9, 5834], base=10

Sorted edges (UNITS digit, digit=1):

```
   Bucket 3: [73]          Bucket 4: [114, 6034, 5834]
   Bucket 7: [2477]        Bucket 9: [849, 909, 9]
   Result (stable):  [73, 114, 6034, 5834, 2477, 849, 909, 9]
```

After 2nd pass (TENS digit, digit=2):

```
   Tens digits: 73→7, 114→1, 6034→3, 5834→3, 2477→7, 849→4,
909→0, 9→0
   Bucket 0: [909, 9]  Bucket 1: [114]  Bucket 3: [6034, 5834]
   Bucket 4: [849]     Bucket 7: [73, 2477]
   Result:  [909, 9, 114, 6034, 5834, 849, 73, 2477]
```

Key: stability is critical — 909 comes before 9 (both tens=0) because 909 was before 9 in the previous pass.

Key: process LSD first (digit=1) not MSD — processing MSD first gives wrong results.

Complexity of Radix Sort

Suppose we sort n integers with k digits in base b. We do k passes of counting sort, each on a universe of size b:

Time: $O(k \cdot (n + b))$

Space: $O(n + b)$

Choosing the right base: for w-bit integers, k = w/log(b). Setting b = n (base-n radix sort):

Time: $O(w/\log(n) \cdot n)$

For integers with at most w = O(log n) bits (i.e. values up to O(n)), this is O(n) — linear!

Case	Time	Aux. Space	Stable?	In-place?
All cases	O(k(n + b))	O(n + b)	Yes†	No

† Stable only if counting sort (or another stable sort) is used for each pass.

Complete Sorting Algorithm Reference

Algorithm	Best	Worst	Stable?	In-place?	Aux Space
Selection Sort	$O(n^2)$	$O(n^2)$	No	Yes	O(1)
Insertion Sort	O(n)	$O(n^2)$	Yes	Yes	O(1)
Merge Sort	O(n log n)	O(n log n)	Yes	No	O(n)
Heapsort	O(n)	O(n log n)	No	Yes	O(1)
Quicksort	O(n log n)	$O(n^2)$	No	No*	O(log n)
Counting Sort	O(n+u)	O(n+u)	Yes	No	O(n+u)
Radix Sort	O(k(n+b))	O(k(n+b))	Yes†	No	O(n+b)

* Quicksort uses O(log n) stack space for recursion — not in-place by the O(1) definition.