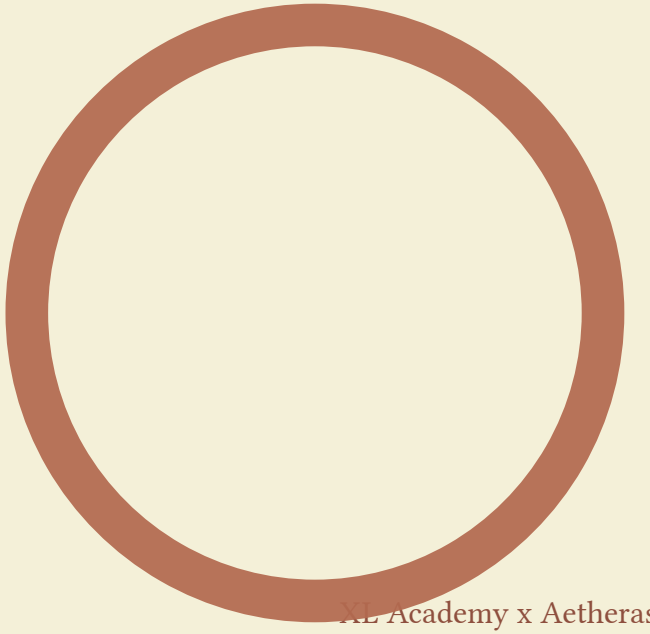
L15

AI Game Development & Computer Science Summer Program

Lecture 15: Algorithms II: Complexity and Tradeoffs

Specifications, runtime analysis, growth families, and algorithm choice.



Keep the contract fixed while comparing cost

Two correct algorithms can make the same promise while using very different amounts of work and memory.

Contract

- Return true when an ID appears more than once.

Pair checking

- Compare every later ID.
- Worst case: quadratic work.

Set strategy

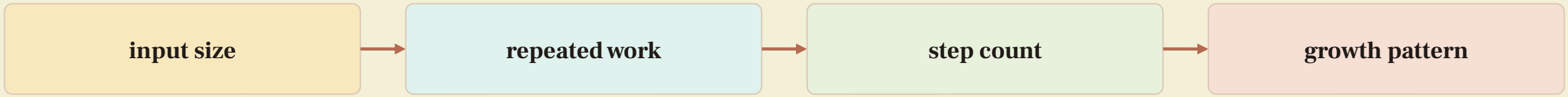
- Remember IDs already seen.
- Expected linear work; extra memory.

Today's question

When does a faster strategy justify its setup and memory cost?

Running time asks how long the algorithm takes

In analysis, running time means steps as a function of input size.



Question to ask

- What operation repeats as the input grows?
- How many times can it happen in the worst case?

A stopwatch measures one run

Step counting explains growth before and beyond one implementation.

Empirical timing

- Run the code.
- Measure elapsed time.
- Useful after implementation.
- Can vary across machines and runs.

Algorithm analysis

- Count steps as input grows.
- Reason before implementation.
- Compare strategies fairly.
- Expose scalability.

Efficiency is a function of input size

Name n first, then count work in terms of n.

Input size	Example	Work unit
n list items	players, names, scores	item comparisons
n characters	a sentence or command	character checks
n nodes	a tree or chain	node visits
n choices	possible move branches	recursive calls or states

Also analyze

- memory usage
- setup cost
- query cost
- precision

Worst case gives a guarantee

Average case is useful only when you know the input distribution.

Case	Meaning	Use
best	lucky input	nice but not guaranteed
average	typical input	needs a real distribution
worst	hardest valid input	sets a performance promise

Default habit

- Start with worst case.
- Then discuss average case if the usage pattern is measured.

Order of growth keeps the dominant term

For large inputs, the leading growth pattern matters most.

Exact count

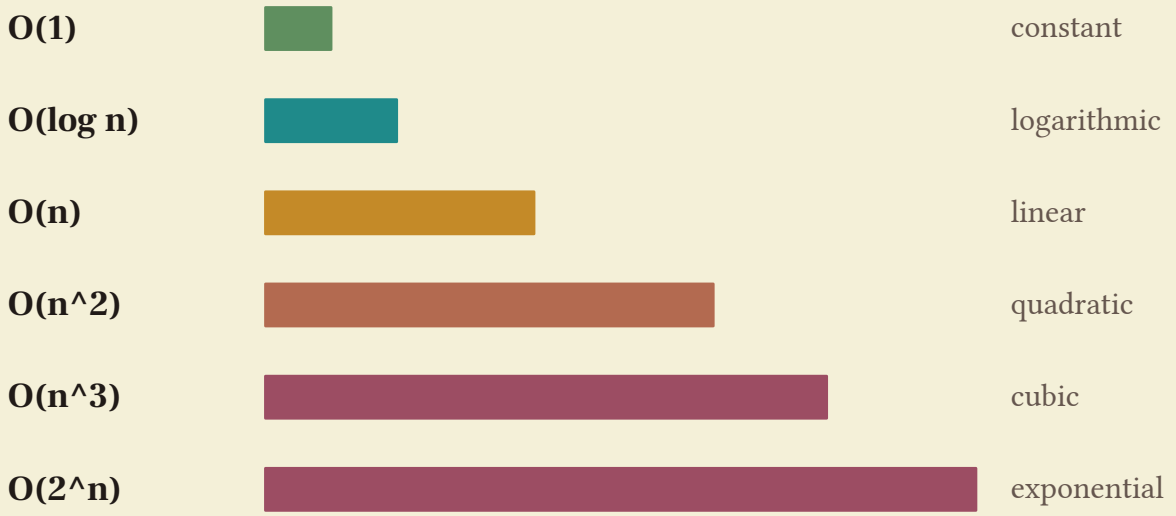
- 10n^2 + 4 log n + n steps

Growth view

- quadratic
- O(n^2)

Why

- The n^2 term controls the shape as n grows.



Names we use

- constant
- logarithmic
- linear
- quadratic
- cubic
- exponential

Linear search checks one item at a time

An unsorted list gives no shortcut.

```
type Profile = { id: string; name: string };

const hasIdLinear = (profiles: readonly Profile[], targetId: string): boolean => {
  for (const profile of profiles) {
    if (profile.id === targetId) return true;
  }
  return false;
};
```

Worst case

- Target is last.
- Target is missing.
- You may inspect all n profiles.

Growth

- Worst-case running time is linear.

Check: unsorted search runtime

Worst-case running time as a function of list size?

A.

- constant

B.

- logarithmic

C.

- linear

D.

- quadratic

E.

- exponential

After you choose

- Linear.
- In the worst case, the algorithm may compare the target with every profile.

Worked example: binary search halves a sorted array

The sorted-array promise makes the middle meaningful.

```
const binaryHasId = (ids: readonly string[], targetId: string): boolean => {
  let low = 0;
  let high = ids.length - 1;
  while (low <= high) {
    const mid = Math.floor((low + high) / 2);
    if (ids[mid] === targetId) return true;
    if (ids[mid] < targetId) low = mid + 1;
    else high = mid - 1;
  }
  return false;
};
```

Requirement	Why
sorted array	halves are meaningful
middle access	jump without walking
bounds update	problem gets smaller

Worst-case steps

- 16 items -> at most 5 checks.
- 1,024 items -> at most 11 checks.
- About log2(n) checks.

Check: binary-search runtime

Worst-case running time as a function of array size?

A.

- constant

B.

- logarithmic

C.

- linear

D.

- quadratic

E.

- exponential

After you choose

- Logarithmic.
- Each comparison can discard about half of the remaining array.

Can lookup be better than logarithmic?

If the data is prepared in advance, memory can buy faster queries.

Linear

- No setup.
- Check one by one.
- Worst case: n checks.

Logarithmic

- Keep a sorted array.
- Throw away half each time.
- Worst case: about log2(n).

Near constant

- Precompute a Set or Map.
- Ask membership by key.
- Extra memory and setup.

How expected constant lookup gets paid for

Expected constant lookup is not free; it moves work into setup and memory.

Strategy	Setup	Query	Memory
scan array	none	linear	low
sort then search	sort once	logarithmic	low
Set lookup	build Set	expected constant	higher

Decision

- Use a Set when repeated queries justify setup and memory.

AI assist: ask for setup and query costs separately, then verify both with the actual data shape.

Pair checking is quadratic

When every item can be paired with many later items, work grows like n squared.

```
const hasSharedBirthday = (dates: readonly string[]): boolean => {
  for (let i = 0; i < dates.length; i++) {
    for (let j = i + 1; j < dates.length; j++) {
      if (dates[i] === dates[j]) return true;
    }
  }
  return false;
};
```

Worst-case pairs

- 10 people -> 45 comparisons.
- 100 people -> 4,950 comparisons.
- 1,000 people -> 499,500 comparisons.

Check: shared-birthday runtime

Worst-case running time for pair checking?

A.

- constant

B.

- logarithmic

C.

- linear

D.

- quadratic

E.

- exponential

After you choose

- Quadratic.
- The nested loops compare pairs, and the number of pairs grows roughly like n squared.

The same spec can have a better algorithm

A Set changes the shared-birthday strategy from pair checking to remembered values.

```
const hasSharedBirthdayFast = (dates: readonly string[]): boolean => {
  const seen = new Set<string>();
  for (const date of dates) {
    if (seen.has(date)) return true;
    seen.add(date);
  }
  return false;
};
```

Tradeoff

- One pass through the list.
- Extra memory for the Set.
- Near constant membership checks.

Subsets are exponential

If you need every subset, each extra item doubles the number of outputs.

Items	Subsets
3	8
10	1,024
20	1,048,576
30	1,073,741,824

Design questions

- Do you really need every combination?
- Can you stop early?
- Can you prune branches?
- Can an approximation be acceptable?

Check: subset generation runtime

Worst-case running time if the output is all subsets?

A.

- constant

B.

- logarithmic

C.

- linear

D.

- quadratic

E.

- exponential

After you choose

- Exponential.
- The output itself has 2^n subsets, so any algorithm that lists them all must do exponential-size work.

Algorithm choice is a tradeoff

Correctness is required; efficiency depends on the context.

Factor	Question
simplicity	can you implement and debug it confidently?
running time	does it finish fast enough for the largest real input?
memory	does faster lookup justify another structure?
precision	is an approximation acceptable?
setup	does the work happen once or before every query?

Decision habit

- State the spec.
- Prove or test correctness.
- Estimate growth.
- Name the tradeoff.

Use Big-O as a comparison tool

The notation is useful only when attached to an input and an operation.

Weak claim	Better claim
This is O(n).	It compares the target to each item, so worst-case comparisons grow linearly.
This is faster.	It spends memory building a Set, then makes repeated membership checks cheaper.
This is too slow.	At 10,000 items, the pair check has about 50 million comparisons.

AI assist: ask for the plain-English reason, not just the notation.

Apply the tradeoff to a real constraint

Choose a strategy only after naming the input size, query pattern, and memory limit.

Small, one-time check

- 12 unsorted IDs
- one duplicate check
- favor simple pair checking

Sorted lookup table

- 1,000,000 sorted IDs
- many lookups, little memory
- use binary search

Repeated membership

- 100,000 IDs
- many membership checks
- use a Set when memory is available

Defend one choice

State the same spec, the repeated operation, its growth label, and one tradeoff.