COMP101 — Computational Thinking

Pattern Recognition | Problem Decomposition | Abstraction | Algorithm Design

UM6P — SASE

January 19, 2026

Plan for today

- Pattern recognition: examples $\rightarrow$ rule $\rightarrow$ proof $\rightarrow$ algorithm

Plan for today

- Pattern recognition: examples $\rightarrow$ rule $\rightarrow$ proof $\rightarrow$ algorithm
- Problem decomposition: split work, split structure, split search space

Plan for today

- Pattern recognition: examples $\rightarrow$ rule $\rightarrow$ proof $\rightarrow$ algorithm
- Problem decomposition: split work, split structure, split search space
- Abstraction: keep what matters, forget what doesn't

Plan for today

- Pattern recognition: examples $\rightarrow$ rule $\rightarrow$ proof $\rightarrow$ algorithm
- Problem decomposition: split work, split structure, split search space
- Abstraction: keep what matters, forget what doesn't
- Algorithm design: turn insights into correct, efficient procedures

Pattern Recognition

Pattern Recognition

Begin with the simplest examples

David Hilbert

Quote

“Man muss immer mit den einfachsten Beispielen anfangen.”

“One must always begin with the simplest examples.”

David Hilbert

A practical workflow

1. Solve a tiny input by hand.

A practical workflow

1. Solve a tiny input by hand.
2. Make a table: input $\rightarrow$ output.

A practical workflow

1. Solve a tiny input by hand.
2. Make a table: input $\to$ output.
3. Make a hypothesis (a real guess).

A practical workflow

1. Solve a tiny input by hand.
2. Make a table: input $\to$ output.
3. Make a hypothesis (a real guess).
4. Try to break it with edge cases.

A practical workflow

1. Solve a tiny input by hand.
2. Make a table: input $\rightarrow$ output.
3. Make a hypothesis (a real guess).
4. Try to break it with edge cases.
5. If it survives: write a lemma $\rightarrow$ code it.

P1 — Trailing zeros of $n!$

Definition

Problem. How many zeros are at the end of $n!$?

Example: $10! = 3628800$ ends with 2 zeros.

- You are not allowed to compute $n!$ directly for large n.

P1 — Trailing zeros of $n!$

Definition

Problem. How many zeros are at the end of $n!$?

Example: $10! = 3628800$ ends with 2 zeros.

- You are not allowed to compute $n!$ directly for large n.
- Goal: a formula / algorithm that works for huge n.

P1 — Work out a few examples

Compute the number of trailing zeros for:

- $5!$

P1 — Work out a few examples

Compute the number of trailing zeros for:

- $5!$
- $10!$

P1 — Work out a few examples

Compute the number of trailing zeros for:

- 5!
- 10!
- 15!

P1 — Work out a few examples

Compute the number of trailing zeros for:

- $5!$
- $10!$
- $15!$
- $25!$

P1 — Think of a rule

- When do we gain a new trailing zero?

P1 — Think of a rule

- When do we gain a new trailing zero?
- Why does 25! jump more than expected?

P1 — Think of a rule

- When do we gain a new trailing zero?
- Why does 25! jump more than expected?
- What should we count instead of multiplying?

P1 — Hints

- A trailing zero means a factor of 10.

P1 — Hints

- A trailing zero means a factor of 10.
- $10 = 2 \cdot 5$.

P1 — Hints

- A trailing zero means a factor of 10.
- $10 = 2 \cdot 5$.
- In $n!$, there are more 2s than 5s.

P1 — Hints

- A trailing zero means a factor of 10.
- $10 = 2 \cdot 5$.
- In $n!$, there are more 2s than 5s.
- So trailing zeros = number of factors of 5 in $n!$.

P1 — Hints

- A trailing zero means a factor of 10.
- $10 = 2 \cdot 5$.
- In $n!$, there are more 2s than 5s.
- So trailing zeros = number of factors of 5 in $n!$.
- Multiples of 25 contribute an extra 5, multiples of 125 contribute another, etc.

Theorem

The number of trailing zeros of $n!$ is

$$\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \cdots$$

(stop when the terms become 0).

Algorithm

Algorithm:

- ans = 0, p = 5

P1 — Answer

Theorem

The number of trailing zeros of $n!$ is

$$\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \cdots$$

(stop when the terms become 0).

Algorithm

Algorithm:

- ans = 0, p = 5
- while p <= n: ans += n // p, then p *= 5

P1 — Answer

Theorem

The number of trailing zeros of $n!$ is

$$\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \cdots$$

(stop when the terms become 0).

Algorithm

Algorithm:

- ans = 0, p = 5
- while p <= n: ans += n // p, then p *= 5
- output ans

P1 — Takeaway

- We replaced a huge computation with counting prime factors.

P1 — Takeaway

- We replaced a huge computation with counting prime factors.
- The key question was: what creates a trailing zero?

Definition

Start from $(1, 1)$. You may apply either operation:

- $(a, b) \to (a + b,\ b)$

Given (x, y), decide if it is reachable.

Definition

Start from $(1, 1)$. You may apply either operation:

- $(a, b) \to (a + b,\ b)$
- $(a, b) \to (a,\ a + b)$

Given (x, y), decide if it is reachable.

P2 — Work out a few examples

Try to decide (reachable or not):

- $(2, 1)$

Definition

Tip: generate reachable pairs for 2–3 moves, then stop and look for structure.

P2 — Work out a few examples

Try to decide (reachable or not):

- $(2, 1)$
- $(2, 2)$

Definition

Tip: generate reachable pairs for 2–3 moves, then stop and look for structure.

P2 — Work out a few examples

Try to decide (reachable or not):

- $(2, 1)$
- $(2, 2)$
- $(3, 2)$

Definition

Tip: generate reachable pairs for 2–3 moves, then stop and look for structure.

P2 — Work out a few examples

Try to decide (reachable or not):

- $(2, 1)$
- $(2, 2)$
- $(3, 2)$
- $(3, 6)$

Definition

Tip: generate reachable pairs for 2–3 moves, then stop and look for structure.

P2 — Work out a few examples

Try to decide (reachable or not):

- $(2, 1)$
- $(2, 2)$
- $(3, 2)$
- $(3, 6)$
- $(8, 13)$

Definition

Tip: generate reachable pairs for 2–3 moves, then stop and look for structure.

P2 — Think deeper

- What quantity seems preserved?

P2 — Think deeper

- What quantity seems preserved?
- If you go backwards, what operation would undo a step?

P2 — Think deeper

- What quantity seems preserved?
- If you go backwards, what operation would undo a step?
- What does this remind you of in number theory?

P2 — Hints

- Backwards: from (a, b) you can go to $(a - b, b)$ if $a > b$, or $(a, b - a)$ if $b > a$.

P2 — Hints

- Backwards: from (a, b) you can go to $(a - b, b)$ if $a > b$, or $(a, b - a)$ if $b > a$.
- Check: $\gcd(a + b, b) = \gcd(a, b)$ and $\gcd(a, a + b) = \gcd(a, b)$.

P2 — Hints

- Backwards: from (a, b) you can go to $(a - b, b)$ if $a > b$, or $(a, b - a)$ if $b > a$.
- Check: $\gcd(a + b, b) = \gcd(a, b)$ and $\gcd(a, a + b) = \gcd(a, b)$.
- So $\gcd(a, b)$ is an invariant.

P2 — Answer

Theorem

(x, y) is reachable from $(1, 1)$ iff $\gcd(x, y) = 1$.

Algorithm

Reason (high level):

- Invariant: reachable implies $\gcd(x, y) = \gcd(1, 1) = 1$.

P2 — Answer

Theorem

(x, y) is reachable from $(1, 1)$ iff $\gcd(x, y) = 1$.

Algorithm

Reason (high level):

- Invariant: reachable implies $\gcd(x, y) = \gcd(1, 1) = 1$.
- If $\gcd(x, y) = 1$, the Euclidean algorithm reduces (x, y) to $(1, 1)$ by repeated subtraction.

P2 — Takeaway

- The pattern was an invariant.

P2 — Takeaway

- The pattern was an invariant.
- Going backwards exposed the structure.

P2 — Takeaway

- The pattern was an invariant.
- Going backwards exposed the structure.
- Many reachability problems hide a known algorithm (here: Euclid).

Definition

Digital root: repeatedly replace n by the sum of its digits until one digit remains.

Example: $38 \to 11 \to 2$, so $\mathrm{dr}(38) = 2$.

- Goal: compute $\mathrm{dr}(n)$ instantly for huge n.

P3 — Digital root (Problem)

Definition

Digital root: repeatedly replace n by the sum of its digits until one digit remains.

Example: $38 \to 11 \to 2$, so $\mathrm{dr}(38) = 2$.

- Goal: compute $\mathrm{dr}(n)$ instantly for huge n.
- No repeated digit-summing loops for million-digit integers.

Compute digital roots:

- $\mathrm{dr}(5)$

P3 — Work out examples

Compute digital roots:

- $\mathrm{dr}(5)$
- $\mathrm{dr}(9)$

Compute digital roots:

- $\mathrm{dr}(5)$
- $\mathrm{dr}(9)$
- $\mathrm{dr}(10)$

P3 — Work out examples

Compute digital roots:

- dr(5)
- dr(9)
- dr(10)
- dr(19)

P3 — Work out examples

Compute digital roots:

- $\mathrm{dr}(5)$
- $\mathrm{dr}(9)$
- $\mathrm{dr}(10)$
- $\mathrm{dr}(19)$
- $\mathrm{dr}(38)$

P3 — Work out examples

Compute digital roots:

- $\mathrm{dr}(5)$
- $\mathrm{dr}(9)$
- $\mathrm{dr}(10)$
- $\mathrm{dr}(19)$
- $\mathrm{dr}(38)$
- $\mathrm{dr}(999)$

- Which numbers map to 9?

- Which numbers map to 9?
- Do numbers with the same remainder mod 9 have the same digital root?

P3 — Think of a rule

- Which numbers map to 9?
- Do numbers with the same remainder mod 9 have the same digital root?
- Why should digit-sum preserve something modulo a number?

P3 — Hint

- $10 \equiv 1 \pmod{9}$.

P3 — Hint

- $10 \equiv 1 \pmod 9$.
- So $10^k \equiv 1 \pmod 9$.

P3 — Hint

- $10 \equiv 1 \pmod{9}$.
- So $10^k \equiv 1 \pmod{9}$.
- If $n = \sum d_k\, 10^k$, then $n \equiv \sum d_k \pmod{9}$.

P3 — Answer

Theorem

$$\operatorname{dr}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 9 & \text{if } n \neq 0 \text{ and } n \equiv 0 \pmod 9, \\ n \bmod 9 & \text{otherwise.} \end{cases}$$

P3 — Takeaway

- The right abstraction was: keep only $n \bmod 9$.

P3 — Takeaway

- The right abstraction was: keep only $n \bmod 9$.
- Pattern recognition often becomes: “what is preserved?”

Definition

n people stand in a circle. Starting from person 1, eliminate every 2nd person. Who survives?

- Start small.

P4 — Josephus (Problem)

Definition

n people stand in a circle. Starting from person 1, eliminate every 2nd person. Who survives?

- Start small.
- Write down the survivor index.

P4 — Josephus (Problem)

Definition

n people stand in a circle. Starting from person 1, eliminate every 2nd person. Who survives?

- Start small.
- Write down the survivor index.
- Then look for structure.

P4 — Work out small cases

Compute the survivor for:

- $n = 1, 2, 3, 4, 5$

Definition

Write the sequence: $J(1), J(2), \ldots$

P4 — Work out small cases

Compute the survivor for:

- $n = 1, 2, 3, 4, 5$
- $n = 6, 7, 8$

Definition

Write the sequence: $J(1), J(2), \dots$

P4 — Work out small cases

Compute the survivor for:

- $n = 1, 2, 3, 4, 5$
- $n = 6, 7, 8$
- $n = 9, 10, 11, 12$

Definition

Write the sequence: $J(1), J(2), \ldots$

- What happens at $n = 2, 4, 8, 16, \ldots$?

P4 — Think deeper

- What happens at $n = 2, 4, 8, 16, \dots$?
- Between two powers of two, what pattern do you see?

P4 — Think deeper

- What happens at $n = 2, 4, 8, 16, \ldots$?
- Between two powers of two, what pattern do you see?
- If $n = 2^k + r$, can you express $J(n)$ using r?

- If n is a power of two, the survivor is 1.

P4 — Hints

- If n is a power of two, the survivor is 1.
- As n increases past a power of two, survivors are odd numbers in order.

P4 — Hints

- If n is a power of two, the survivor is 1.
- As n increases past a power of two, survivors are odd numbers in order.
- There is a clean closed form using the largest power of two $\leq n$.

Theorem

Let p be the largest power of two with $p \leq n$. Write $n = p + r$ with $0 \leq r < p$. Then:

$$J(n) = 2r + 1.$$

Algorithm

Algorithm:

- Find $p = 2^{\lfloor \log_2 n \rfloor}$.

Theorem

Let p be the largest power of two with $p \leq n$. Write $n = p + r$ with $0 \leq r < p$. Then:

$$J(n) = 2r + 1.$$

Algorithm

Algorithm:

- Find $p = 2^{\lfloor \log_2 n \rfloor}$.
- Compute $r = n - p$.

Theorem

Let p be the largest power of two with $p \leq n$. Write $n = p + r$ with $0 \leq r < p$. Then:

$$J(n) = 2r + 1.$$

Algorithm

Algorithm:

- Find $p = 2^{\lfloor \log_2 n \rfloor}$.
- Compute $r = n - p$.
- Output $2r + 1$.

Pattern recognition recap

- P1: prime factor counting.

Pattern recognition recap

- P1: prime factor counting.
- P2: invariants + reversing the process.

Pattern recognition recap

- P1: prime factor counting.
- P2: invariants + reversing the process.
- P3: modular preservation.

Pattern recognition recap

- P1: prime factor counting.
- P2: invariants + reversing the process.
- P3: modular preservation.
- P4: structure around powers of two.

Problem Decomposition

Problem Decomposition

What is problem decomposition?

Definition

Decomposition is breaking a problem into smaller parts that are easier to solve and test.

In algorithms, it usually looks like:

- preprocess once + answer many queries,

What is problem decomposition?

Definition

Decomposition is breaking a problem into smaller parts that are easier to solve and test.

In algorithms, it usually looks like:

- preprocess once + answer many queries,
- split structure (left/right, segments) + merge,

What is problem decomposition?

Definition

Decomposition is breaking a problem into smaller parts that are easier to solve and test.

In algorithms, it usually looks like:

- preprocess once + answer many queries,
- split structure (left/right, segments) + merge,
- split search space (meet-in-the-middle).

Decomposition steps

1. Understand the main goal.

Decomposition steps

1. Understand the main goal.
2. Break it into major subproblems.

Decomposition steps

1. Understand the main goal.
2. Break it into major subproblems.
3. Break each subproblem until it is directly solvable.

Decomposition steps

1. Understand the main goal.
2. Break it into major subproblems.
3. Break each subproblem until it is directly solvable.
4. Solve subproblems (bottom-up).

Decomposition steps

1. Understand the main goal.
2. Break it into major subproblems.
3. Break each subproblem until it is directly solvable.
4. Solve subproblems (bottom-up).
5. Integrate + test.

Static range sum queries

Definition

Given an array $a[1..n]$ and many queries (l, r), return

$$\sum_{i=l}^{r} a[i].$$

- Naive: sum each query directly $\Rightarrow$ too slow.

Definition

Given an array $a[1..n]$ and many queries (l, r), return

$$\sum_{i=l}^{r} a[i].$$

- Naive: sum each query directly $\Rightarrow$ too slow.
- Decompose into preprocessing + queries.

Work a tiny example

Array: $[3, 1, 4, 1, 5]$.

Queries:

- sum(2,4)

Definition

What single precomputed array would let you answer every query in O(1)?

Work a tiny example

Array: $[3, 1, 4, 1, 5]$.

Queries:

- sum(2,4)
- sum(1,5)

Definition

What single precomputed array would let you answer every query in O(1)?

Work a tiny example

Array: $[3, 1, 4, 1, 5]$.

Queries:

- sum(2,4)
- sum(1,5)
- sum(3,3)

Definition

What single precomputed array would let you answer every query in O(1)?

Decomposition: prefix sums

Algorithm

Subproblem A (preprocess): Build $P[i] = a[1] + \cdots + a[i]$.

Subproblem B (query): Answer with $P[r] - P[l-1]$.

Theorem

One-time work: $O(n)$. Each query: $O(1)$.

$\#$ of subarrays divisible by K (Problem)

Definition

Given an array $a[1..n]$ and an integer K, count subarrays (l, r) such that

$$\sum_{i=l}^{r} a[i] \equiv 0 \pmod{K}.$$

Decomposition: turn ranges into a prefix condition

- Let $P[i] = a[1] + \cdots + a[i]$.

Decomposition: turn ranges into a prefix condition

- Let $P[i] = a[1] + \cdots + a[i]$.
- Then $\mathrm{sum}(l..r) = P[r] - P[l-1]$.

Decomposition: turn ranges into a prefix condition

- Let $P[i] = a[1] + \cdots + a[i]$.
- Then $\text{sum}(l..r) = P[r] - P[l-1]$.
- Divisible by K means:

$$P[r] \equiv P[l-1] \pmod{K}.$$

Answer: count frequencies

Algorithm

Algorithm:

- Compute remainders $R[i] = P[i] \bmod K$.

Answer: count frequencies

Algorithm

Algorithm:

- Compute remainders $R[i] = P[i] \bmod K$.
- For each remainder value with count c, add $\binom{c}{2} = c(c-1)/2$.

Karatsuba multiplication (Problem)

Definition

Multiply two very large integers faster than grade-school multiplication.

- Grade-school: 4 multiplications of half-size parts.

Karatsuba multiplication (Problem)

Definition

Multiply two very large integers faster than grade-school multiplication.

- Grade-school: 4 multiplications of half-size parts.
- Karatsuba: reduce to 3 multiplications.

Decomposition: split numbers into halves

Let x and y be big numbers. Split each into high/low parts:

$$x = x_1 \cdot 10^m + x_0, \qquad y = y_1 \cdot 10^m + y_0.$$

- Grade-school expands into 4 products: $x_1y_1, x_1y_0, x_0y_1, x_0y_0$.

Decomposition: split numbers into halves

Let x and y be big numbers. Split each into high/low parts:

$$x = x_1 \cdot 10^m + x_0, \qquad y = y_1 \cdot 10^m + y_0.$$

- Grade-school expands into 4 products: $x_1 y_1, x_1 y_0, x_0 y_1, x_0 y_0$.
- Karatsuba reduces this to 3 products using algebra.

Key trick: 3 multiplications

Compute:

$$A = x_1 y_1, \quad B = x_0 y_0, \quad C = (x_1 + x_0)(y_1 + y_0).$$

Then the cross term is:

$$x_1 y_0 + x_0 y_1 = C - A - B.$$

Theorem

Decomposition + recombination: same answer, fewer expensive multiplications.

Meet in the middle (warm-up): $n = 20$

Definition

Subset sum: given n numbers, is there a subset with sum S?

- If $n = 20$, brute force over all subsets: $2^{20} \approx 10^6$ (often OK).

Meet in the middle (warm-up): $n = 20$

Definition

Subset sum: given n numbers, is there a subset with sum S?

- If $n = 20$, brute force over all subsets: $2^{20} \approx 10^6$ (often OK).
- If $n = 40$, brute force: $2^{40} \approx 10^{12}$ (not OK).

Algorithm

Decompose the search space:

- Split the numbers into two halves of size about 20.

Algorithm

Decompose the search space:

- Split the numbers into two halves of size about 20.
- Enumerate all subset sums on the left: list L.

Meet in the middle: decomposition for $n \approx 40$

Algorithm

Decompose the search space:

- Split the numbers into two halves of size about 20.
- Enumerate all subset sums on the left: list L.
- Enumerate all subset sums on the right: list R.

Meet in the middle: decomposition for $n \approx 40$

Algorithm

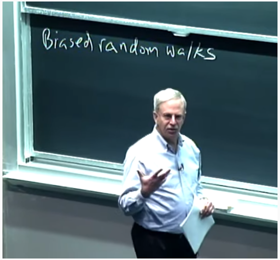
Decompose the search space:

- Split the numbers into two halves of size about 20.
- Enumerate all subset sums on the left: list L.
- Enumerate all subset sums on the right: list R.
- Sort one list. For each $x \in L$, check if $S - x$ exists in R.

Abstraction

Abstraction

A quote on abstraction



John V.Guttag

Quote

"The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context."

John V. Guttag

Abstraction in computational thinking

Definition

Abstraction is choosing a representation that makes the problem solvable.

- Keep only the state you truly need.

Definition

Abstraction is choosing a representation that makes the problem solvable.

- Keep only the state you truly need.
- Convert a story into a clean mathematical model.

Abstraction in computational thinking

Definition

Abstraction is choosing a representation that makes the problem solvable.

- Keep only the state you truly need.
- Convert a story into a clean mathematical model.
- Make correctness easy to argue.

Balanced parentheses (Problem)

Definition

Given a string like "(()())", decide if it is balanced.

- You want a one-pass algorithm.

Balanced parentheses (Problem)

Definition

Given a string like ”(()())”, decide if it is balanced.

- You want a one-pass algorithm.
- You want a rule you can prove.

Abstraction: map to +1 and -1

- Map '(' to $+1$ and ')' to -1.

Theorem

Balanced means:

Abstraction: map to +1 and -1

- Map '(' to $+1$ and ')' to -1.
- Keep one counter c.

Theorem

Balanced means:

Abstraction: map to +1 and -1

- Map '(' to +1 and ')' to -1.
- Keep one counter c.

Theorem

Balanced means:

- c never goes negative,

Abstraction: map to +1 and -1

- Map '(' to +1 and ')' to -1.
- Keep one counter c.

Theorem

Balanced means:

- c never goes negative,
- and ends at 0.

Answer + algorithm

```
def balanced(s: str) -> bool:
    c = 0
    for ch in s:
        c += 1 if ch == '(' else -1
        if c < 0:
            return False
    return c == 0
```

Definition

We ignored irrelevant details and kept the necessary state: one counter.

Definition

How many ways to tile a $1 \times n$ row using 1×1 tiles and 2×1 tiles?

- Compute small n.

Tiling a row (Problem)

Definition

How many ways to tile a $1 \times n$ row using 1×1 tiles and 2×1 tiles?

- Compute small n.
- Look for a recurrence.

Work out small cases

Let $f(n)$ be the number of tilings.

- $f(0)$ (empty row)

Work out small cases

Let $f(n)$ be the number of tilings.

- $f(0)$ (empty row)
- $f(1)$

Work out small cases

Let $f(n)$ be the number of tilings.

- $f(0)$ (empty row)
- $f(1)$
- $f(2)$

Work out small cases

Let $f(n)$ be the number of tilings.

- $f(0)$ (empty row)
- $f(1)$
- $f(2)$
- $f(3)$

Work out small cases

Let $f(n)$ be the number of tilings.

- $f(0)$ (empty row)
- $f(1)$
- $f(2)$
- $f(3)$
- $f(4)$

Abstraction: focus on the last tile

- If the last tile is 1×1, the rest is a tiling of $n - 1$.

Abstraction: focus on the last tile

- If the last tile is 1×1, the rest is a tiling of $n - 1$.
- If the last tile is 2×1, the rest is a tiling of $n - 2$.

Abstraction: Conclusion

Theorem

Recurrence:

$$f(n) = f(n-1) + f(n-2).$$

Answer + algorithm

Definition

Several piles of stones. On your turn, pick one pile and remove any positive number.

Player who takes the last stone wins.

- Which positions are losing?

Nim (Problem)

Definition

Several piles of stones. On your turn, pick one pile and remove any positive number.

Player who takes the last stone wins.

- Which positions are losing?
- What is the winning move when possible?

Work out tiny examples

Try two piles (a, b):

- $(1, 1)$

Definition

Look for a simple rule that predicts losing positions.

Work out tiny examples

Try two piles (a, b):

- $(1, 1)$
- $(1, 2)$

Definition

Look for a simple rule that predicts losing positions.

Work out tiny examples

Try two piles (a, b):

- $(1, 1)$
- $(1, 2)$
- $(2, 2)$

Definition

Look for a simple rule that predicts losing positions.

Work out tiny examples

Try two piles (a, b):

- $(1, 1)$
- $(1, 2)$
- $(2, 2)$
- $(1, 3)$

Definition

Look for a simple rule that predicts losing positions.

Work out tiny examples

Try two piles (a, b):

- $(1, 1)$
- $(1, 2)$
- $(2, 2)$
- $(1, 3)$
- $(2, 3)$

Definition

Look for a simple rule that predicts losing positions.

Abstraction: nim-sum

- Represent the state by one number: XOR of pile sizes.

Abstraction: nim-sum

- Represent the state by one number: XOR of pile sizes.
- Call it the nim-sum.

Answer (decision)

```python
def winning(piles):
    x = 0
    for p in piles:
        x ^= p
    return x != 0
```

Definition

Abstraction move: many piles → one number (XOR).

Algorithm Design & Summary

Algorithm Design & Summary

Definition

Algorithm design is turning insights into a procedure that is:

- correct (always gives the right answer),

Definition

Algorithm design is turning insights into a procedure that is:

- correct (always gives the right answer),
- efficient (meets the constraints),

Algorithm design: from idea to solution

Definition

Algorithm design is turning insights into a procedure that is:

- correct (always gives the right answer),
- efficient (meets the constraints),
- implementable (clear steps).

A computational thinking checklist

1. Solve a tiny case.

A computational thinking checklist

1. Solve a tiny case.
2. Write the naive algorithm.

A computational thinking checklist

1. Solve a tiny case.
2. Write the naive algorithm.
3. Identify the bottleneck.

A computational thinking checklist

1. Solve a tiny case.
2. Write the naive algorithm.
3. Identify the bottleneck.
4. Choose a tool: prefix sums, gcd invariant, modulo, divide and conquer, hashing, etc.

A computational thinking checklist

1. Solve a tiny case.
2. Write the naive algorithm.
3. Identify the bottleneck.
4. Choose a tool: prefix sums, gcd invariant, modulo, divide and conquer, hashing, etc.
5. Prove the key step.

A computational thinking checklist

1. Solve a tiny case.
2. Write the naive algorithm.
3. Identify the bottleneck.
4. Choose a tool: prefix sums, gcd invariant, modulo, divide and conquer, hashing, etc.
5. Prove the key step.
6. Implement and test edge cases.

Summary

- Pattern recognition turns examples into rules.

Summary

- Pattern recognition turns examples into rules.
- Decomposition turns impossible into manageable.

Summary

- Pattern recognition turns examples into rules.
- Decomposition turns impossible into manageable.
- Abstraction turns messy stories into clean models.

Summary

- Pattern recognition turns examples into rules.
- Decomposition turns impossible into manageable.
- Abstraction turns messy stories into clean models.
- Algorithm design turns models into correct, fast procedures.

End

Questions?