Algorithms - Spring '25

Induction

Pseudocode +

Runtime

Recap

- Reading due by 8am, every day we have class
  ↳ no excuses, but I'll drop lowest 3 to allow for illness / forgetting / travel
- HW 0: due next Wed
- No class or office hour Monday
- Extra office hour on Tuesday from 3-5pm

Last time:

- Big-O
- Identities & Summations
- Induction

④ Induction:

There is a template!

Base case: Prove statement for small value

Ind hypothesis: Assume true for values $\leq k$

Ind. step: Prove true for next value $k+1$

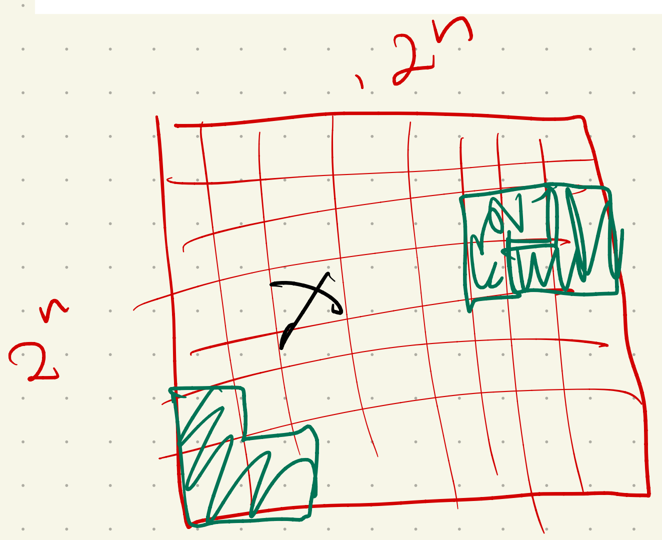
Think of this as "automating" a proof.

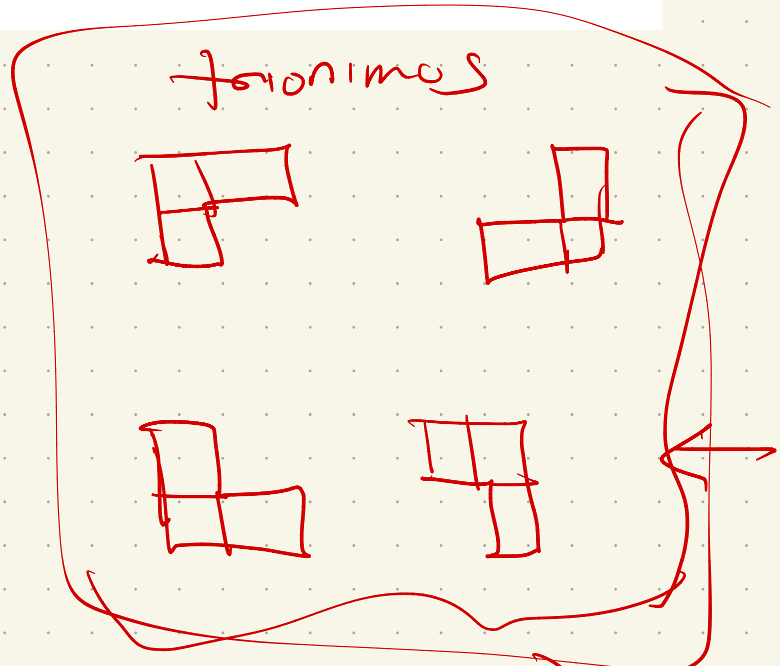
- $P(1)$
- $\forall k \geq 1$, $\overbrace{\text{if } P(k)}^{\text{IH}}$ then $\underbrace{P(k+1)}_{\text{IS}}$

Learn it, use it, + love it!

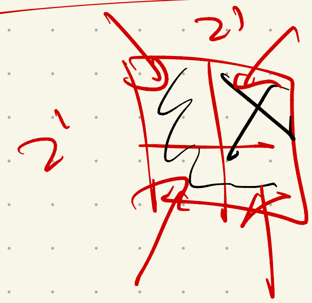
"Structural" induction:

Let n be a positive integer. Show that every $2^n \times 2^n$ checkerboard with one square removed can be tiled using right triominoes, where these pieces cover three squares at a time, as shown in Figure 4.



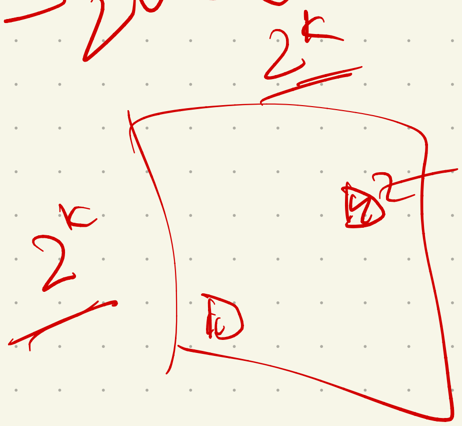


Base case: $2^1 \times 2^1$ board

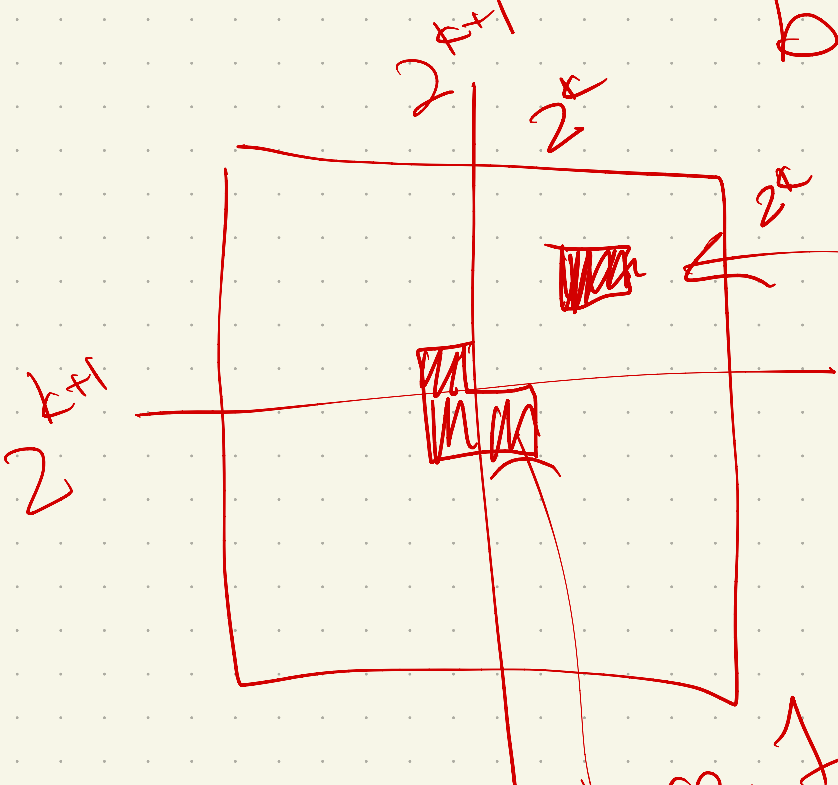


No matter which square is removed, I can cover the other 3

IH: Assume I can tile any $2^k \times 2^k$ board with any 1 square removed



IS: Show I can do a $2^{k+1} \times 2^{k+1}$ board: (w/ one cell removed)



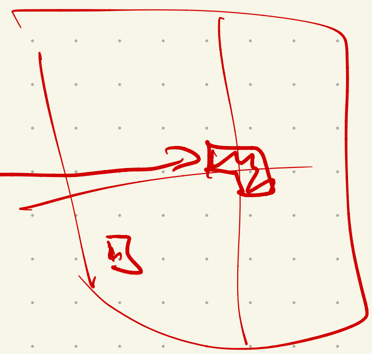
I can tile this smaller square (by IH)

place 1 triomino

Cut $2^{k+1} \times 2^{k+1}$ board into 4 smaller ones, each $2^k \times 2^k$.

One of them is missing a cell. The other 3 share a common center which can be covered by a trionimo:

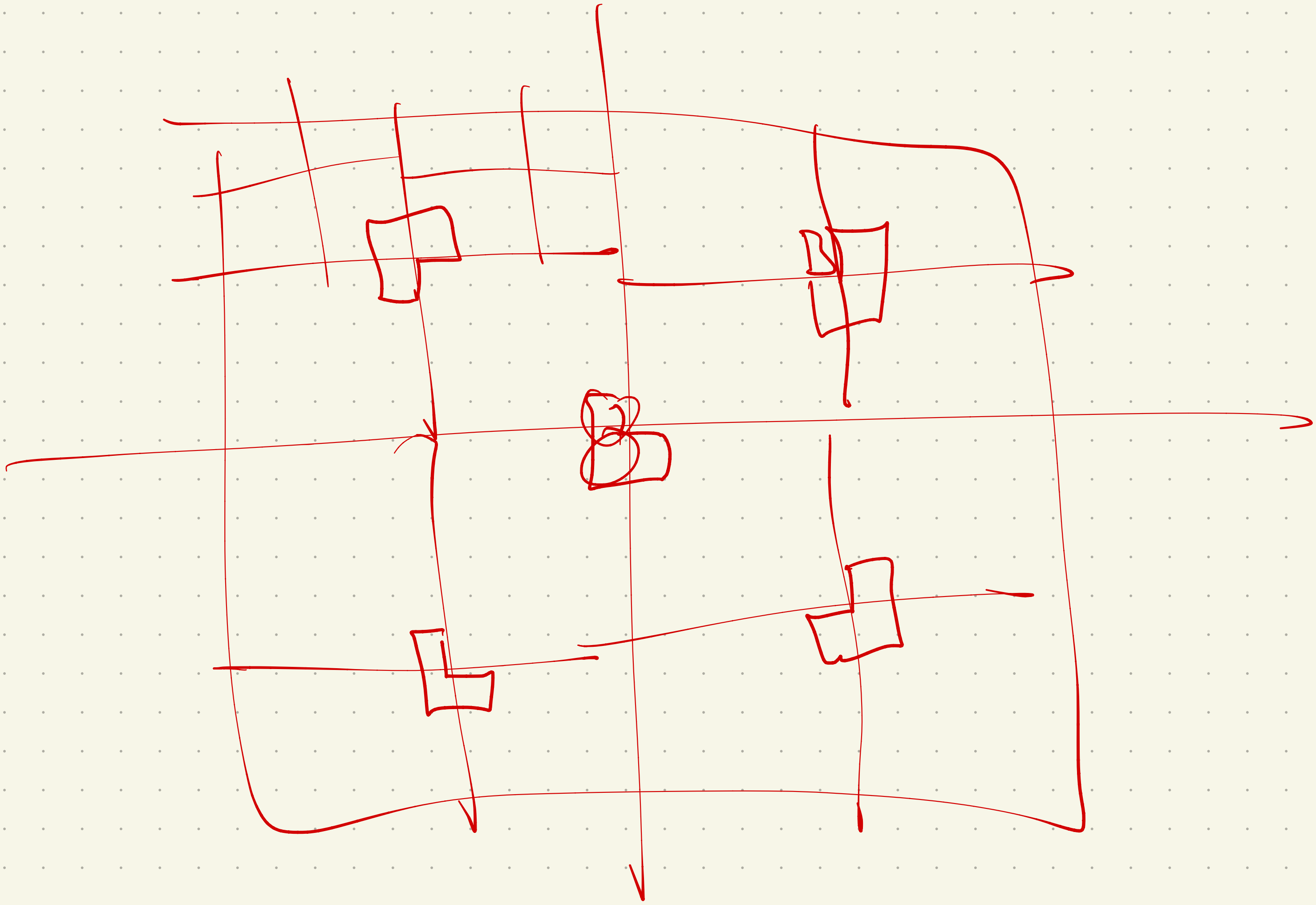
Remove those 3 center cells



Now have 4 $2^k \times 2^k$ boards, each missing a square

↳ tile by IH.

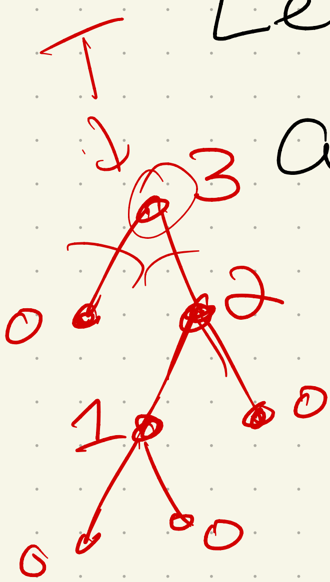
⇒ Tile the larger board: use the IH 4 solution, plus center trionimo. ∎



$$A(n) = 4A\left(\frac{n}{4}\right) + 1$$

Induction on graphs / tree

Let $h(T)$ = height of a full binary tree



full → every node has 0 or 2 children

$$h(T) = \begin{cases} 0 & \text{if } T \text{ is a single node} \\ 1 + \max(h(T\text{'s root's children})) \end{cases}$$

Claim: The number of nodes in a full binary tree is

$$\le 2^{h(T)} + 1$$

$$n \le 2^h + 1 \Rightarrow h = O(\log n)$$

Claim: The number of nodes in a full binary tree is $\leq 2^{h(T)} + 1$.

mistake: should be -1

Proof: induction on size of tree, n

Base case: $n = 1$

height $= 0$ (by dfn)

and $2^0 + 1 = 2$

$$1 \leq 2$$

n $\rightarrow 1$, $2 \leftarrow 2^h + 1$

IH: If we have $k < n$ vertices, then

$$k \leq 2^{h(T)} + 1$$

Ind step:

Consider n nodes:

$$h(T) = \max\{h(T_1), h(T_2)\} + 1$$

n nodes: root

2 subtrees T_1 & T_2

nodes in $T_1 < n$

nodes in $T_2 < n$

Use IH:

nodes in $T_1 \le 2^{h(T_1)} + 1$

nodes in $T_2 \le 2^{h(T_2)} + 1$

$$n = 1 + (\text{nodes in } T_1) + (\text{nodes in } T_2)$$

$$\le 1 + (2^{h(T_1)} + 1) + (2^{h(T_2)} + 1)$$

$$\le 1 + 2\left(2^{\max\{h(T_1), h(T_2)\}} + 1\right)$$

$$= 1 + 2^{\max\{h(T_1), h(T_2)\}+1} + 2$$

$$= 3 + 2^{h(T)}$$

stay tuned...

$$2^{h(T)} - 1 \checkmark$$

5) Pseudo code + runtime: Discrete math examples (from Rosen textbook)

ALGORITHM 1 Finding the Maximum Element in a Finite Sequence.

procedure *max*($a_1, a_2, \ldots, a_n$: integers)
max := a_1
for $i := 2$ **to** n
    if *max* $< a_i$ **then** *max* $:= a_i$
return *max*{*max* is the largest element}

variable assignment

Pascal-like

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \le n$ and $x \ne a_i$)
    $i := i + 1$
if $i \le n$ **then** *location* $:= i$
else *location* $:= 0$
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

Pseudocode conventions here:

Variable assignment: $x \leftarrow 2$

↑ arrow

Boolean comparison:

if $(x = 5)$ ← T or F

Arrays: $A[0..n-1]$ ← n is size of array

- each $A[i]$

value of a set type (same for array)

Loops: for $i \leftarrow 1$ to 100

$A[i] \leftarrow i$

Assume "basic data structures"

Again, takes practice! Read intro chapter & then give HW0/HW1 a try.

Pseudocode format:

In a pinch, pretend you're in Python/Ruby.

High level & readable.

I realize this is not a "definition" – that is the point!

It's about effective communication.

Initially:

- lots of examples
- lots of practice
- reach out if you have questions
- in a pinch – peer evaluation!

Example (& tie to runtimes):

Mulitplication:

Input: 2 numbers ↳ in decimal

$X[0 .. m-1]$, $Y[0 .. n-1]$ ← m digit, ← n digit

& $X = \sum_{i=0}^{m-1} X[i] \cdot 10^i$, $Y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j$

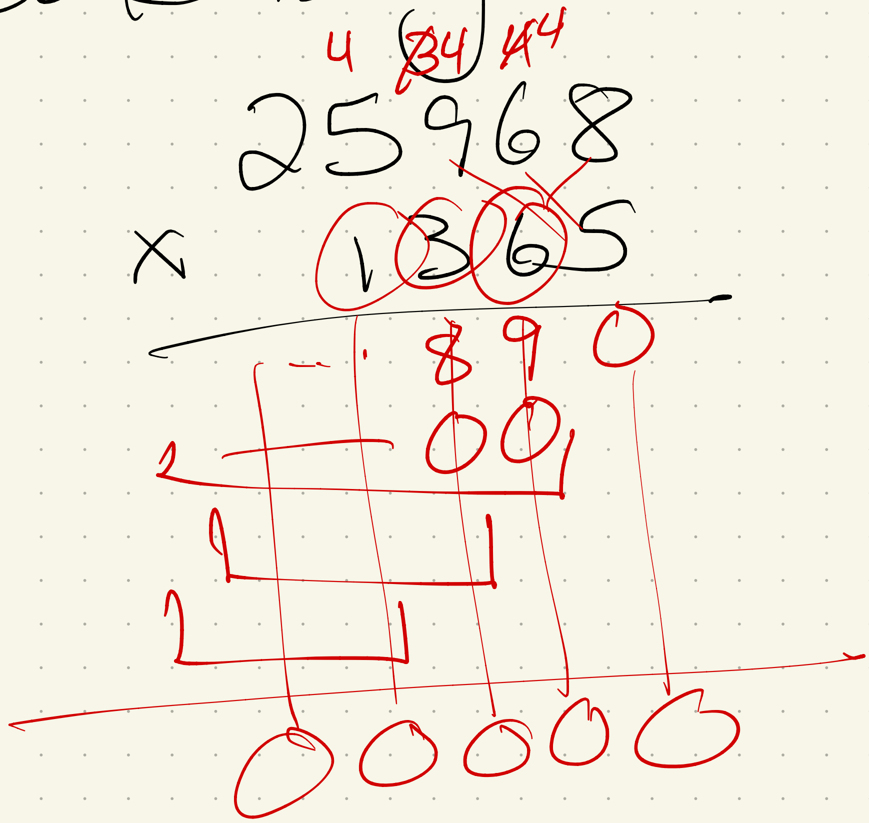
Example: $X = 25968$

$Y = 1365$

↳ $X = 8 \cdot 10^0 + 6 \cdot 10^1 + 9 \cdot 10^2 + 5 \cdot 10^3 + 2 \cdot 10^4$

$X[0 .. 4] = [\underset{0}{8}, \underset{1}{6}, \underset{2}{9}, \underset{3}{5}, \underset{4}{2}]$

Back to grade school:



Abstract:

Find all digits:

+ how many powers of 10 they get:

$$X \cdot Y = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1}$$

Another view: Instead of just adding all, search for all that land in one spot k:

> **FIBONACCIMULTIPLY**$(X[0..m-1], Y[0..n-1])$:
> $hold \leftarrow 0$
> for $k \leftarrow 0$ to $n+m-1$
>   for all i and j such that $i+j=k$
> $hold \leftarrow hold + X[i] \cdot Y[j]$
> $Z[k] \leftarrow hold \bmod 10$
> $hold \leftarrow \lfloor hold/10 \rfloor$
> return $Z[0..m+n-1]$

Space & runtime:

More complex: recursion!

Algorithm 1 Quicksort

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
procedure PARTITION(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1 do
        if A[j] < x then
            i = i + 1
            exchange A[i] with A[j]
        end if
        exchange A[i] with A[r]
    end for
end procedure
```

QuickSort Pseudocode Example

Any function which calls itself (on a smaller size)

Multiplication: How?

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

Why? Proof by cases:

if x is even:

if x is odd:

Note: historical name! Not a commentary..

```
PEASANTMULTIPLY(x, y):
    if x = 0
        return 0
    else
        x' ← ⌊x/2⌋
        y' ← y + y
        prod ← PEASANTMULTIPLY(x', y')    ⟨⟨Recurse!⟩⟩
        if x is odd
            prod ← prod + y
        return prod
```

Runtime:

Correctness

Recursive Algorithms : Chapter 1

1st half:

Setup, plus (hopefully) familiar examples:

- Towers of Hanoi
- Merge sort

2nd half:

- Recap of recurrences & "Master theorem"
- Linear time selection
- Multiplication (again)
- Exponentiation