Correctness

It is ***impossible*** to design an algorithm to analyze if general algorithms are going to halt or not, emphasis on *general*. *Halting Problem*

With specific algorithms or base-cases in such algorithms it is possible to know when and how it will halt.

`TOPIC`: Correctness via induction

Induction Accepted axioms:

- Well-Ordering Principle (WOP):

  $$if X \subseteq \mathbb{N} = 0, 1, \ldots$$

  is nonempty, then X has a least element $\exists m \in X$ s.t. $m \leq x, \forall x \in X$

- Let P be a proposition on $\mathbb{N}$, i.e $\forall\ n \in \mathbb{N}$, then

  $P(n)$ can be true or false if

  1. $P(n_0)$ true
  2. $P(n), P(n_1), P(n_2), \ldots, P(n)$ is true for all $n \geq n_0$, $n \in \mathbb{N}$

  Then $P(n)$ is true $\forall n \in \ltimes$, $n \geq n_0$ *Strong Induction Method*

- P as above:

  1. $P(n_0)$ true
  2. $P(n) \implies P(n+1)$ is true for all $n \geq n_0$, $n \in \mathbb{N}$

*→ "LEAST" IN THIS CASE **DOES NOT** mean it is WITHIN THE BOUNDS. It means there is an element that is SMALLER THAN THE SMALLEST ELEMENT IN X* ***!!!***

Dated -¿ 1/16/2026

`TOPIC`: Inductive Proof Practice

*→**Euler's Characteristic Formula:*** *A relationship for planar graphs, which are graphs capable of being laid out on a table-top with NO crossing edges*

Proof of sum of numbers (by *WOP*):

Assume

$$1 + 2 + 3... + n \neq \frac{n(n+1)}{2}$$

then, $\exists n_0 \geq 0$ s.t.

$$1 + 2... + n_0 \neq \frac{n_0(n_0+1)}{2}$$

Let $\mathbb{X} = n | n \geq 0, n \in \mathbb{N} s.t. 1 + 2 + ... + n \neq \frac{n(n+1)}{2}$

Thus $\mathbb{X} \neq$

Hence $\exists m \in \mathbb{X}$ that is the least element

$--> $ **1-23-26**

Proof to-do list:

- Proof methods:
  - direct proof (assume p, **by xyz**, therefor p)
  - proof by contrapositive (assume **not** p)
  - proof by contradiction (assume p, **but xyz**, therefor p)
- **W**ell **O**rdering **P**rinciple, **W.O.P**
- Proof by Induction (both weak and strong)

- Combinatorial proof
  - number of permutations = $n!$
  - number of r-permutations from n = $n(n-1),\ldots,(n-r+1)$ = $\frac{n!}{r!}$
- pigeonhole principle
- proof of inclusion - exclusion **:** $\hat{(}$
-
-

For these proofs, if i want to progress towards a correct answer, Dr. Liow mentioned that we should **"make up a story"**.

IMPORTANT!!!!

It sounds dumb but since math is already so abstracted, grounding it in a real-world use-case scenario helps to fit the components with logical conclusions so that when i take one step in a direction that is wrong, i can just say it out loud and if it doesnt make any logical sense i can take a step in a different direction to approach a better more logically sound answer.

GRAFFS!!!!! :DDDDD

the same formula for $v - e + f = 2$ correlates to the relationship between the number of `v = vertices` and `e = edges` between a type of graph called a *tree*

Proof. Euler's Handshaking Lemma

Eschewing undirected graphs ***without loops or multi-edges***,

Let G be a graph.

Then, $\sum deg(v) \approxeq \in \mathbb{V} = 2|E|$, where G = (V, E).

For the graph, **python needs to work first ehehehe**

We call the points where an edge touches a node an **incident point**

In the above (a node without a label connets to a node (labeled p) at a point labeled p', from there it connects to another node that has a loop, labeled p")

The number of incident points can be counted in 2 ways:

- Enumerate all edges: $e_1, \ldots, e_i$ Each e_i has 2 incident points, and there are no others.

  Therefore the number of incident points must be **2**$|E|$

- Enumerate all nodes: $v_1, \ldots, v_m$

  Each v_i has deg(v_i) incident points, and there are no others.

Hence $\sum deg(\gtrapprox) \gtrapprox \in \mathbb{V}$ = the total number of incident points

which then = 2$|E|$!!! **yay!!!**

110: Search Algorithms

TOPIC: Main idea

A pertinent idea Liow mentions is the *partial solutions* that make up a search algorithm. We describe algorithms in stages, where we perform one option such as sorting, then another like mathematical operations: but they are clearly distinct from one another. Though distinct, they help construct the piece of the puzzle the next leg of the algorithm will focus on. This is very similar to the issue of sub-problems he mentions that we deal with in recursion or general problem solving.

110.1: Backtrack

TOPIC: Decision-making

The idea of search algorithms is to make decisions. We want the decision to be correct as much as possible, but when it isn't we are left with the last possible option: return from where we came and make another (maybe educated) decision in order to try something that may lead us to a correct solution.

This is exemplified best by **depth-first searches**, where we exhaust a chain of decisions quickly to find a proper answer. Honestly reading this back as I write it, its starting to make a lot of sense. The idea for backtracking is like

"i went this way for a little bit, i didnt quite find what i was looking for but if i go back a few paces, maybe the turns back behind me lead to the correct solution"

at least i hope thats what he was getting at

$-->$ **1-22-26**

110.2: Pseudocode

So after reading some more, theres a **very** important point i want to jot down for later:

backtracking should distinguish between "forward" and "backward"

What does that mean? **"forward"** attempts to build an entire solution from the larger subset of smaller *"partial solutions"*. Sometimes its right, sometimes its wrong. if its right, then awesome we found a solution.

BUT, if its *wrong*, we now have to define **"backward"**. Backward in this case is trying to go back to a point in time where we had more *potentially* correct options, more potential **partial solutions** we could use to build towards making a whole solution.

`TOPIC`: Emphasis on efficiency

The above bit of text is explaining the general, bare-bones idea of a backtracking search solution. But to make it efficient there cant be too many items that comprise the solution.

For example, if we know that a solution has options `op_0, op_1, ... op_n`, and we start building a concrete solution with options `op_0, op_5, op_10`, we wouldnt include them the next time we have to backtrack ALL the way back to that first iteration of finding a solution.

On top of that, (hehe foreshadowing puns) it would be more prudent to *add* the current working partial solution *on top of* the existing solution instead of making separate objects for each and every stage of a partial solution fitting our overall algorithm.

110.2: Knight's tour

my solution for the 3×4 knight's tour question:

9	6	1	4
0	3	8	11
7	10	5	2

I mean i guess its one solution, but whatever.

Anyways, i read from pg 20 to like pg 50 which is mostly a demo of some code for finding the knights tour of a $n \times m$ board, but it showed off only 3 × 3 and 5 × 5.

I was reading it and the basic synopsis of how i should approach backtracking (for this problem at least) was like this:

- Have a starting point (in this case the top left-most cell in the chessboard)
- continue building the solution from the last valid starting point, lets call it p
- if all options in that "tree" past p are failures, then go up past p and choose $p + 1$ or whatever other iteration of p is next
- repeat the process until a solution is found or there is no possible solution and all other options have been exhausted

Essentially what i just said but in code form:

```
ALGORITHM: NONRECURSIVE-BACKTRACK

let s be an empty stack
push the empty solution onto s
perform_goal_test = FALSE

direction = "down"

while s is not empty:

      if perform_goal_test:
         if s is solution:
            return s
         perform_goal_test = FALSE

      let t be the top of stack s

      if direction is "down":
         if t has a child:
            push child of t onto s
            perform_goal_test = TRUE
      else:
         direction = "right"
```

and this is the code i mentioned earlier:

110.10: Magic Square

110.11: Sudoku Problem