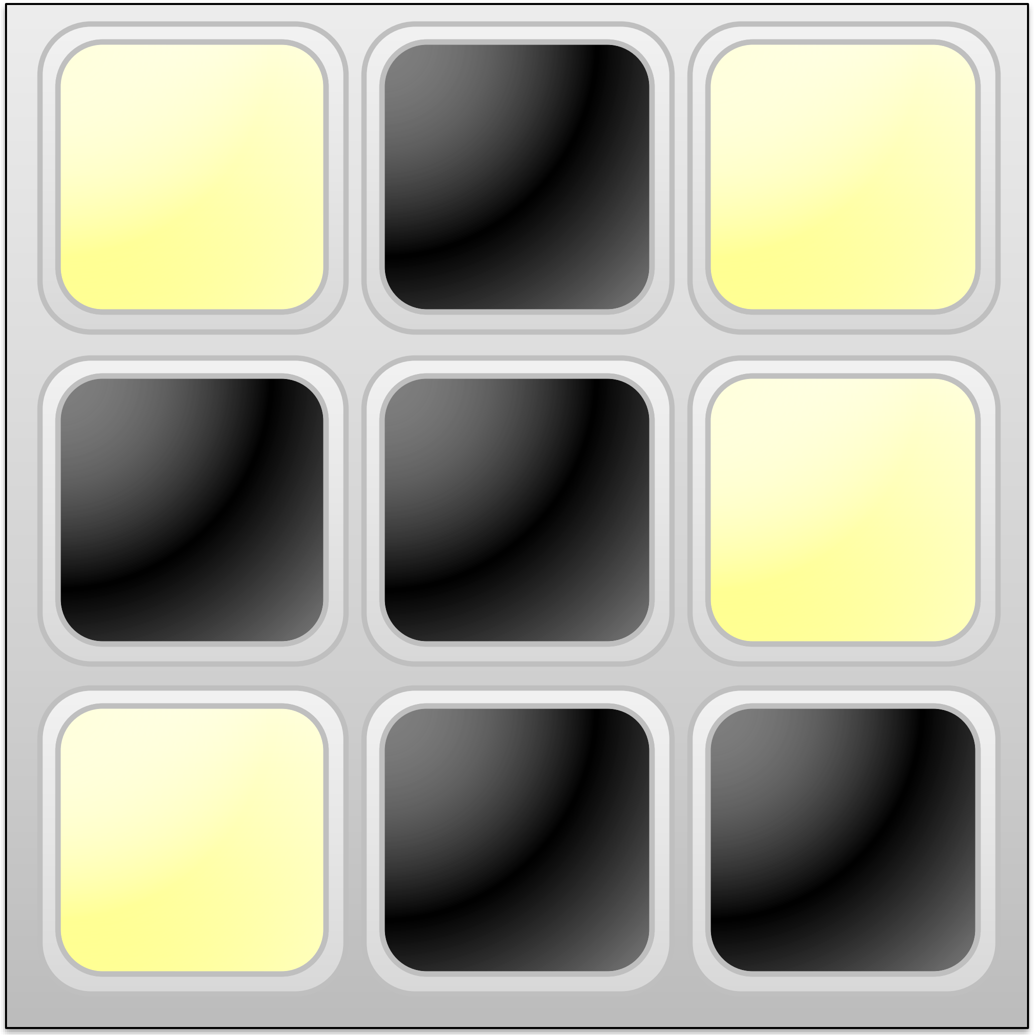
Recall

- In a complete search solution, we need to go through all possibilities
- The possibilities are not always literally given to us in a list
- Sometimes, we need to invent a way to go through all possibilities
- In many cases, we can just use (nested) loops
- In lots of other cases, this requires more work

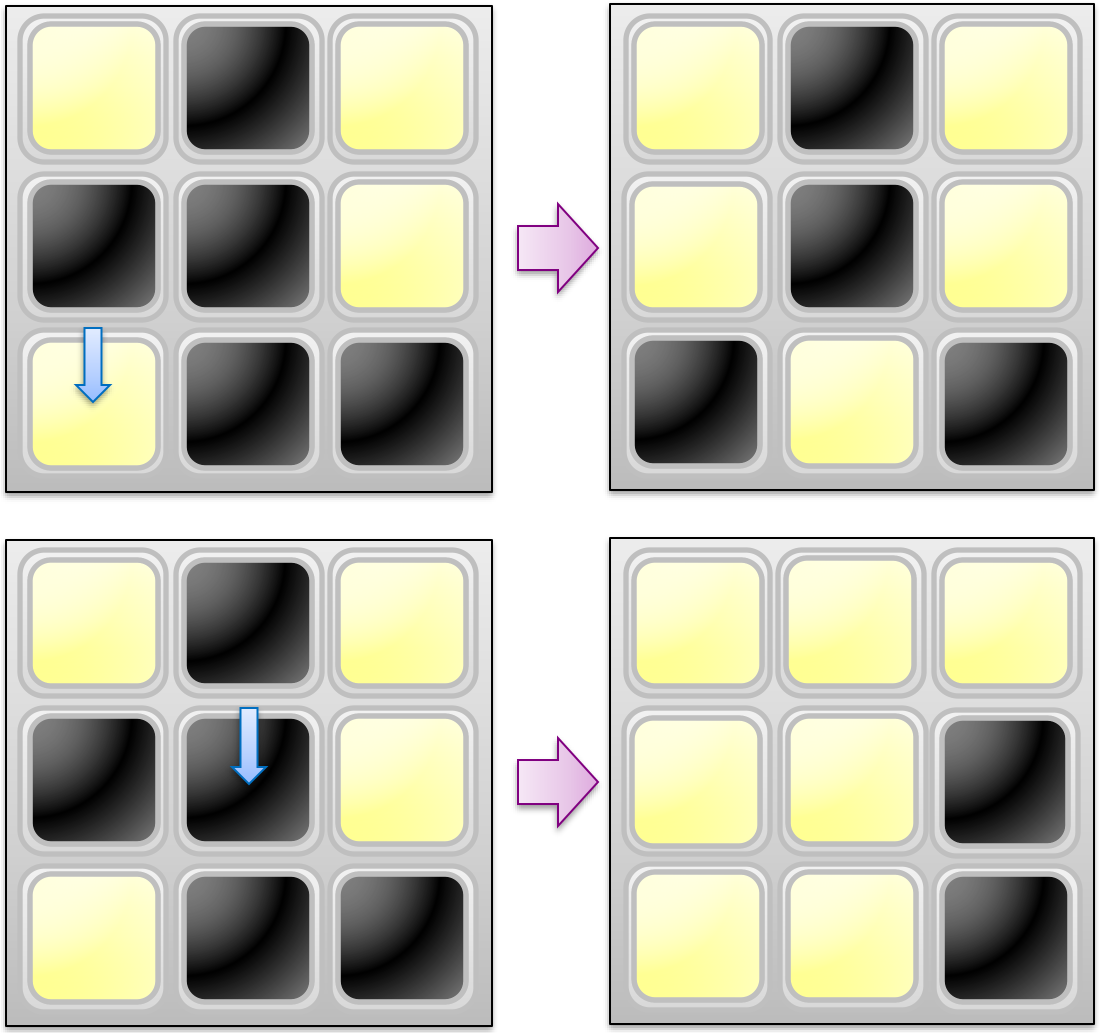
Lights Out

- Puzzle with a grid of lights
- Each light can be on or off
- Objective: turn all lights off



Lights Out

- Light can be switched by pressing on it
- However, switching a light also switches its direct neighbors above, below, to the left, and to the right

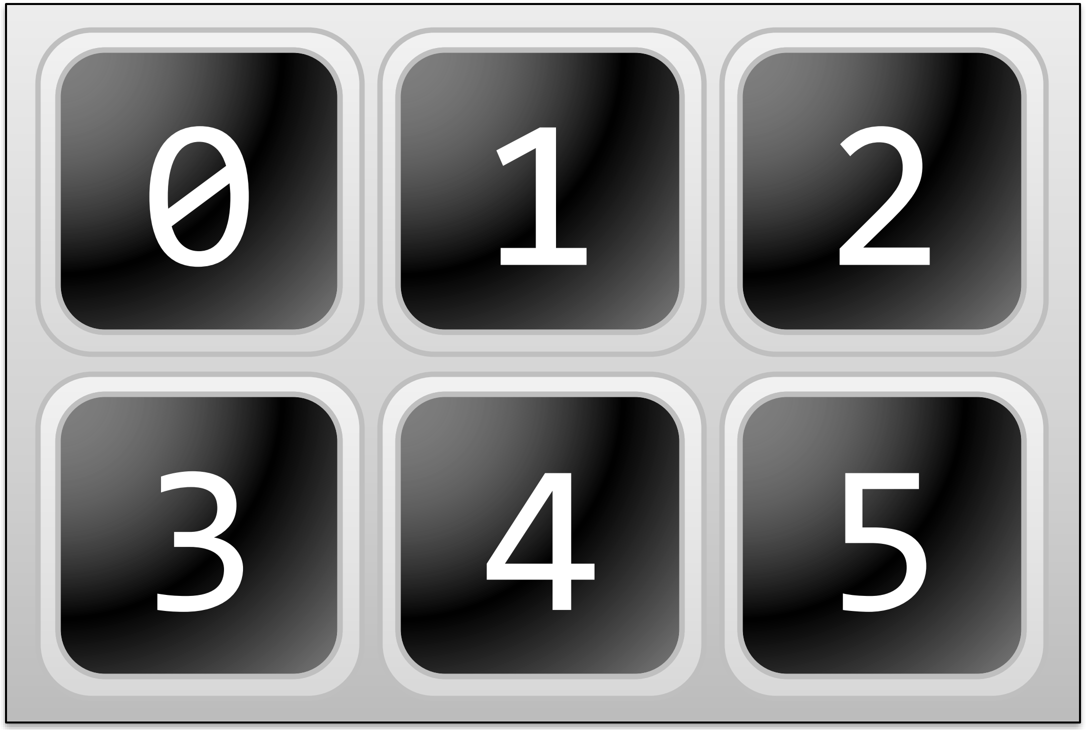


The Problem

- Given the number of rows and columns and the starting grid configuration, determine if the puzzle is solvable or not
  - If yes, output a sequence of presses
- There is a smart solution (try thinking about it later), but first let's think about how to solve it via complete search

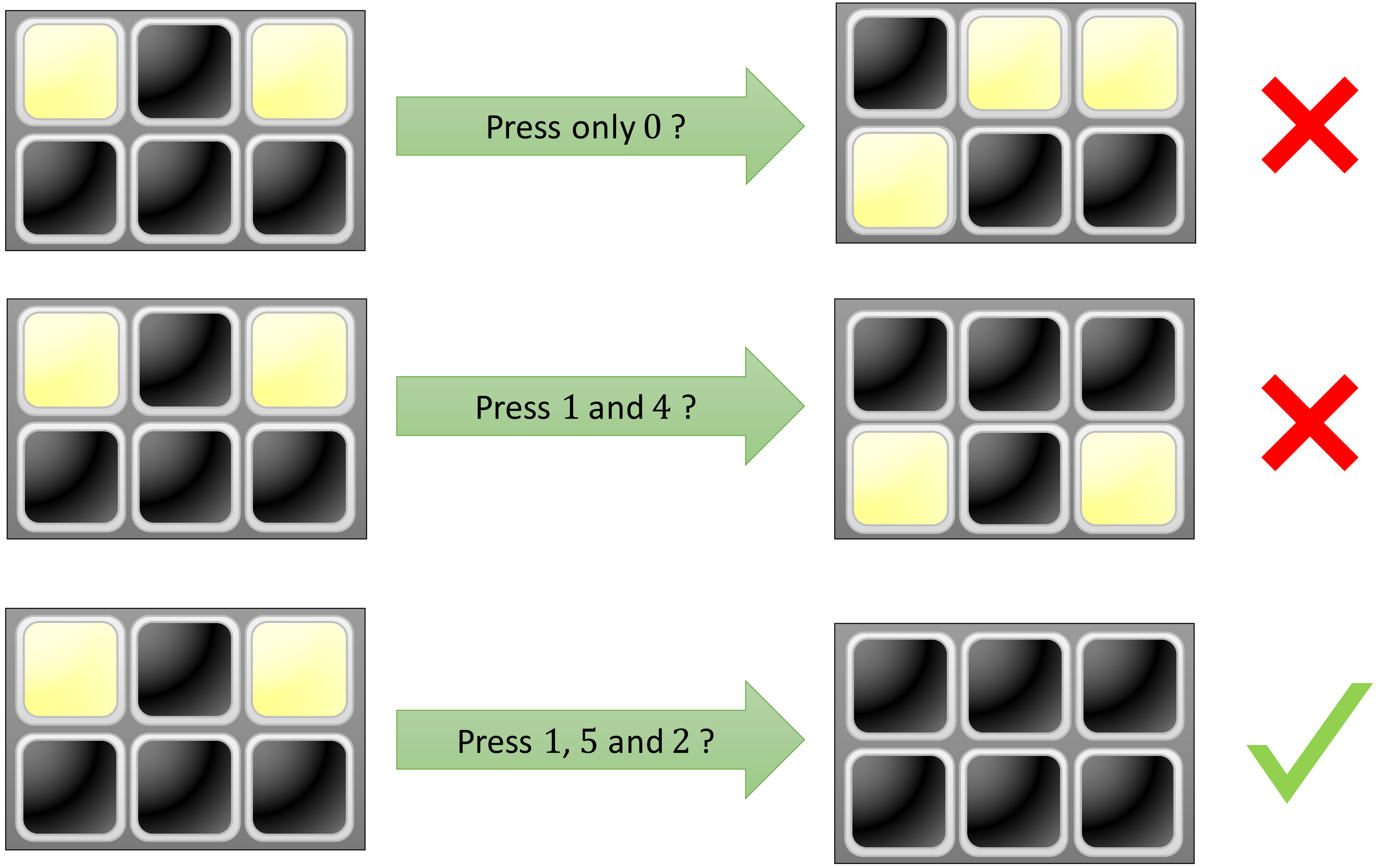
Solving with Complete Search

- Start with a small, fixed-size grid, say 2×3
- Label the lights with integers



Solving with Complete Search

- Is it necessary to press the same light more than once?
  - No
- Does the order in which the lights are pressed matter?
  - No
- Means we just need to decide which among the lights to press
- Which combination should we try?
- All of them!


Press only 0 ?
Press 1 and 4 ?
Press 1, 5 and 2 ?

Solving with Complete Search

- Complete search solution: for each combination of lights, if it solves the puzzle, save it/print it
- But how exactly does "for each combination" work?
- Idea: deciding which lights to press is the same as *independently* deciding for each individual light, whether to press it once or not at all
- For a grid with exactly 6 lights, we can go through all the possibilities using 6 loops

Solving with Complete Search

```
for light0 in [0, 1]: # number of times to press light 0
  for light1 in [0, 1]: # number of times to press light 1
    for light2 in [0, 1]: # number of times to press light 2
      for light3 in [0, 1]: # number of times to press light 3
        for light4 in [0, 1]: # number of times to press light 4
          for light5 in [0, 1]: # number of times to press light 5
            check([light0, light1, light2, light3, light4, light5])
```

Generalizing…

- 4×5 grid? 20 loops
- 10×10 grid? 100 loops
- $n \times m$ grid, where n and m are given as input, not known while you are writing the code
  - It's impossible to prepare the right number of loops in advance

We Need Some Sort of "Super-Loop"

- Nest this n times:

```
for light_i in [0, 1]:
```

- Then do this at the end:

```
check([light_0, light_1, ..., light_n])
```

Step Back and Think What Each Loop is Doing

`for light0 in [0, 1]:`	Decide `light0` is not pressed, then let loop 1 decide. When loop 1 finishes, decide `light0` is pressed once, then let loop 1 decide again.
`for light1 in [0, 1]:`	Decide `light1` is not pressed, then let loop 2 decide. When loop 2 finishes, decide `light1` is pressed once, then let loop 2 decide again.
`for light2 in [0, 1]:`	Decide `light2` is not pressed, then let loop 3 decide. When loop 3 finishes, decide `light2` is pressed once, then let loop 3 decide again.
`for light3 in [0, 1]:`	Decide `light3` is not pressed, then let loop 4 decide. When loop 4 finishes, decide `light3` is pressed once, then let loop 4 decide again.
`for light4 in [0, 1]:`	Decide `light4` is not pressed, then let loop 5 decide. When loop 5 finishes, decide `light4` is pressed once, then let loop 5 decide again.
`for light5 in [0, 1]:`	Decide `light5` is not pressed, then check if the solution works. Then decide `light5` is pressed once, then check if the solution works.

Step Back and Think What Each Loop is Doing

- Thinking about it this way, each loop only needs to worry about the loop immediately after it
- The succeeding steps can be *blackboxed* away
- We can express this with functions

Writing Nested Loops with Functions

```python
def loop0():
    loop1([0])
    loop1([1])


def loop1(lights):
    loop2(lights + [0])
    loop2(lights + [1])


def loop2(lights):
    loop3(lights + [0])
    loop3(lights + [1])

def loop3(lights):
    loop4(lights + [0])
    loop4(lights + [1])


def loop4(lights):
    loop5(lights + [0])
    loop5(lights + [1])


def loop5(lights):
    check(lights + [0])
    check(lights + [1])
```

Loops 0 to 4 Are Basically the Same, So We Can Combine Them

```
def loop0():
    loop(0, [])
```

```
def loop(x, lights):
    loop(x + 1, lights + [0])
    loop(x + 1, lights + [1])
```

```
def loop5(lights):
    check(lights + [0])
    check(lights + [1])
```

aspc ateneo summer programming camp

Loop 5 is Slightly Different, But We Can Include It in the Same Function with a Conditional

```
def loop0():
    loop(0, [])


def loop(x, lights):
  if x == 5:
      check(lights + [0])
      check(lights + [1])
  else:
      loop(x + 1, lights + [0])
      loop(x + 1, lights + [1])
```

- Notice how we naturally end up with a recursive function!
- Think about it:
  - How to make this work for any given number of lights?
  - Can we simplify this so that there's only one check call in the base case?

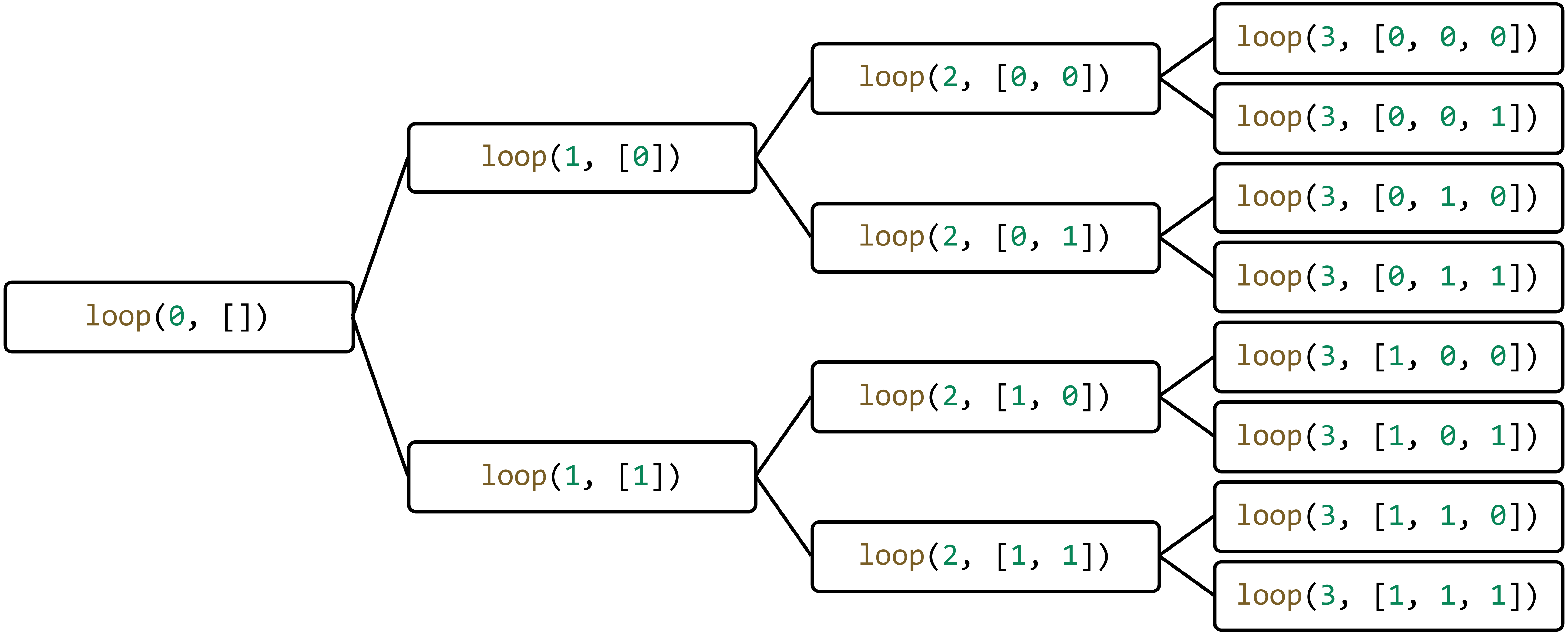
Generalizing

```
def loop(x, lights):
    if x == L - 1: # if L is the number of lights
        check(lights + [0])
        check(lights + [1])
    else:
        loop(x + 1, lights + [0])
        loop(x + 1, lights + [1])
```

Simplifying the Base Case (Think Why This Works)

```
def loop(x, lights):
    if x == L: # if L is the number of lights
        check(lights)
    else:
        loop(x + 1, lights + [0])
        loop(x + 1, lights + [1])
```

We Can Visualize with a Tree Diagram



The Main Idea

- "Go through all possibilities" can be broken down into steps, where each step consists of making an *independent* decision
- If the number of decisions to make is too large or unknown in advance, and *the choices available are basically the same for all steps*, we can naturally express the process with recursion, instead of lots of nested for loops
- The recursive logic is almost always the same for going through all sorts of combinations: to enumerate all combinations, make a decision for one item, and then recursively enumerate all combinations for the rest of the items

Running Time Analysis

- If there are L lights…
- Root of recursion tree has 1 node
- Which branches into 2 nodes at level 1
- Which branch into 4 nodes at level 2
- Which branch into 8 nodes at level 3
- Which branch into 2^L nodes at level L

Running Time Analysis

- Level i of the recursion tree has 2^i nodes
- The total is $2^0 + 2^1 + 2^2 + \cdots + 2^L = O(2^{L+1})$
  - The left-hand side is $L + 1$ ones in binary
  - The right-hand side is 1 followed by $L + 1$ zeros in binary
  - These two numbers only differ by 1
  - More simply, you can say that since every level branches into 2 and there are L levels, then there are roughly 2^L nodes
    - This isn't exactly correct, but as a rough estimate, it's good enough

Running Time Analysis

- Level i of the recursion tree has 2^i nodes
- The total is $2^0 + 2^1 + 2^2 + \cdots + 2^L = O(2^{L+1})$
- We spend $O(L)$ time at each node
  - For the recursive case, $O(L)$ to create a new list with the new element
  - For the base case, $O(L)$ to check if the chosen lights solve the puzzle
- Total time $O(2^{L+1}L)$

If You Removed Loops From Python, Can You Still Solve the Same Problems?

- Yes!
  - Assuming we removed recursion limits from Python

Sum of First n Positive Integers

```python
ans = 0
i = 1
while i <= n:
    ans += i
    i += 1

print(ans)
```

Sum of First n Positive Integers

```python
ans = 0
i = 1
while i <= n:
    ans += i
    i += 1

print(ans)
```

```python
def sum(ans, i):
  if i <= n:
    return sum(ans + i, i + 1)
  else:
    return ans

print(sum(0, 1))
```

Writing Loops Without Writing Loops

- Any loop can be transformed into recursion like this
- The transformations are even somewhat "mechanical"
- Notice how every piece of logic on the left has a corresponding piece on the right
- If you know about keyword arguments in Python, writing the recursive version with keyword arguments makes the correspondence even more obvious

Initial Values

```python
ans = 0
i = 1
while i <= n:
    ans += i
    i += 1

print(ans)
```

```python
def sum(ans, i):
  if i <= n:
    return sum(ans + i, i + 1)
  else:
    return ans

print(sum(0, 1))
```

Update Step

```
ans = 0
i = 1
while i <= n:
    ans += i
    i += 1

print(ans)
```

```
def sum(ans, i):
  if i <= n:
    return sum(ans + i, i + 1)
  else:
    return ans

print(sum(0, 1))
```

Terminating / Continuing Condition

```python
ans = 0
i = 1
while i <= n:
    ans += i
    i += 1

print(ans)
```

```python
def sum(ans, i):
  if i <= n:
    return sum(ans + i, i + 1)
  else:
    return ans

print(sum(0, 1))
```

This Seems Familiar…

Iteration

1. Look at the goal
2. **Believe** that the loop *already works before you've written anything*, for steps 1 to $i - 1$
3. Using the values *you already have* for $i - 1$, figure out a way to make your variables contain the correct values for step i
4. Do the easy parts

Recursion

1. Look at the goal
2. **Believe** that the function *already exists* and will give you the correct answer on some smaller version of the problem
3. Using the answers *you already have* for the smaller version, build the answer for the original version
4. Do the easy parts

This Seems Familiar...

Iteration

- Initial variable values
- Loop step
- Terminating condition

Recursion

- Initial function call
- Recursive call
- Base case

There's a Different Way...

```
def sum(n):
    if n == 0:
        return 0
    else:
        return n + sum(n - 1)
```

There's a Different Way...

- But it looks less like the loop
- Let's call the first way *loopy recursion*

How to Write "Super-Loops"

- A loopy recursion lets us do one thing after another *in series*

```
def loop(args):
    loop(update(args))
```

How to Write “Super-Loops”

- Adding more recursive calls allows the computation to split into *parallel* branches
- Each recursive call can be slightly different to reflect the fact that a different decision is made in each branch

```
def do(args):
    do(update1(args))
    do(update2(args))
    do(update3(args))
```

AKA Recursive Backtracking

- In reality, the function calls aren't executed in parallel
- Each branch of the recursion tree will be explored in full before the next branch is explored
- After a branch is fully explored, the computer *backtracks* to the deepest node in the recursion tree with yet unexplored branches and takes a different decision
- Because of how the code is executed, this style of solving problems is called **recursive backtracking**

AKA Recursive Backtracking

- However, no need to think about the backtracking process while coming up with the code
- Much easier to imagine:
  - The computer literally spawns parallel processes at each recursive step
  - A recursive backtracking solution is literally trying all decisions in parallel

Recursive Backtracking: A More Powerful Way to Brute Force

- The recursion tree of the "loopy recursion" `sum` above would be just a straight line
- Loops are enough when the search can be naturally represented as "straight lines"
- For search that is more naturally represented with trees, recursive backtracking is the way

Exercise

- Write code that prints all n-letter strings that can be formed from A, B, and C
- For example, all the 3-letter strings are AAA, AAB, AAC, ABA, ABB, ABC, ACA, ACB, ACC, BAA, BAB, BAC, BBA, BBB, BBC, BCA, BCB, BCC, CAA, CAB, CAC, CBA, CBB, CBC, CCA, CCB, CCC

Possible Answer

```python
def generate(string_so_far, n):
    if len(string_so_far) == n:
        print(string_so_far)
    else:
        generate(string_so_far + 'A', n)
        generate(string_so_far + 'B', n)
        generate(string_so_far + 'C', n)

generate('', int(input()))
```

Exercise

- Write code that prints all n-letter strings that can be formed from the 26 capital English letters
  - Recall that `ord` returns the integer value of a single-letter string and that `chr` returns a single-letter string whose integer value is given, hence that `chr(ord('A') + i)` returns the ith capital English letter (counting from 0)
  - In C++, just do `char('A' + i)`

Possible Answer

```
def generate(so_far, n):
    if len(so_far) == n:
        print(so_far)
    else:
        for i in range(26):
            generate(so_far + chr(ord('A') + i), n)

generate('', int(input()))
```

Exercise

- Write code that prints the number of n-letter strings that can be formed from the 26 capital English letters and which do not have "ABC" as a substring

Possible Answer

```
ways = 0

def generate(so_far, n):
    global ways
    if len(so_far) == n:
        ways += 1 if 'ABC' not in so_far else 0
    else:
        for i in range(26):
            generate(so_far + chr(ord('A') + i), n)

generate('', int(input()))
print(ways)
```

Another Possible Answer

```python
def ways(so_far, n):
    if len(so_far) == n:
        return 1 if 'ABC' not in so_far else 0
    else:
        return sum(
            ways(so_far + chr(ord('A') + i), n)
            for i in range(26)
        )

print(ways('', int(input())))
```

Exercise

- Assume there is a function `check` which takes an n-letter string as input and returns True if it is a correct "password" or False otherwise
- Write code that prints the minimum number of A's in a string among all strings which are correct passwords
- Assume there is at least one correct password

Possible Answer

```python
answer = float('inf')

def generate(so_far, n):
    global answer
    if len(so_far) == n:
        if check(so_far):
            answer = min(answer, so_far.count('A'))
    else:
        for i in range(26):
            generate(so_far + chr(ord('A') + i), n)

generate('', int(input()))
print(answer)
```

Another Possible Answer

```
def best(so_far, num_As, n):
    if len(so_far) == n:
        return num_As if check(so_far) else float('inf')
    else:
        return min(
            best(
                so_far + chr(ord('A') + i),
                num_As + (1 if i == 0 else 0),
                n
            )
            for i in range(26)
        )

print(best('', 0, int(input())))
```

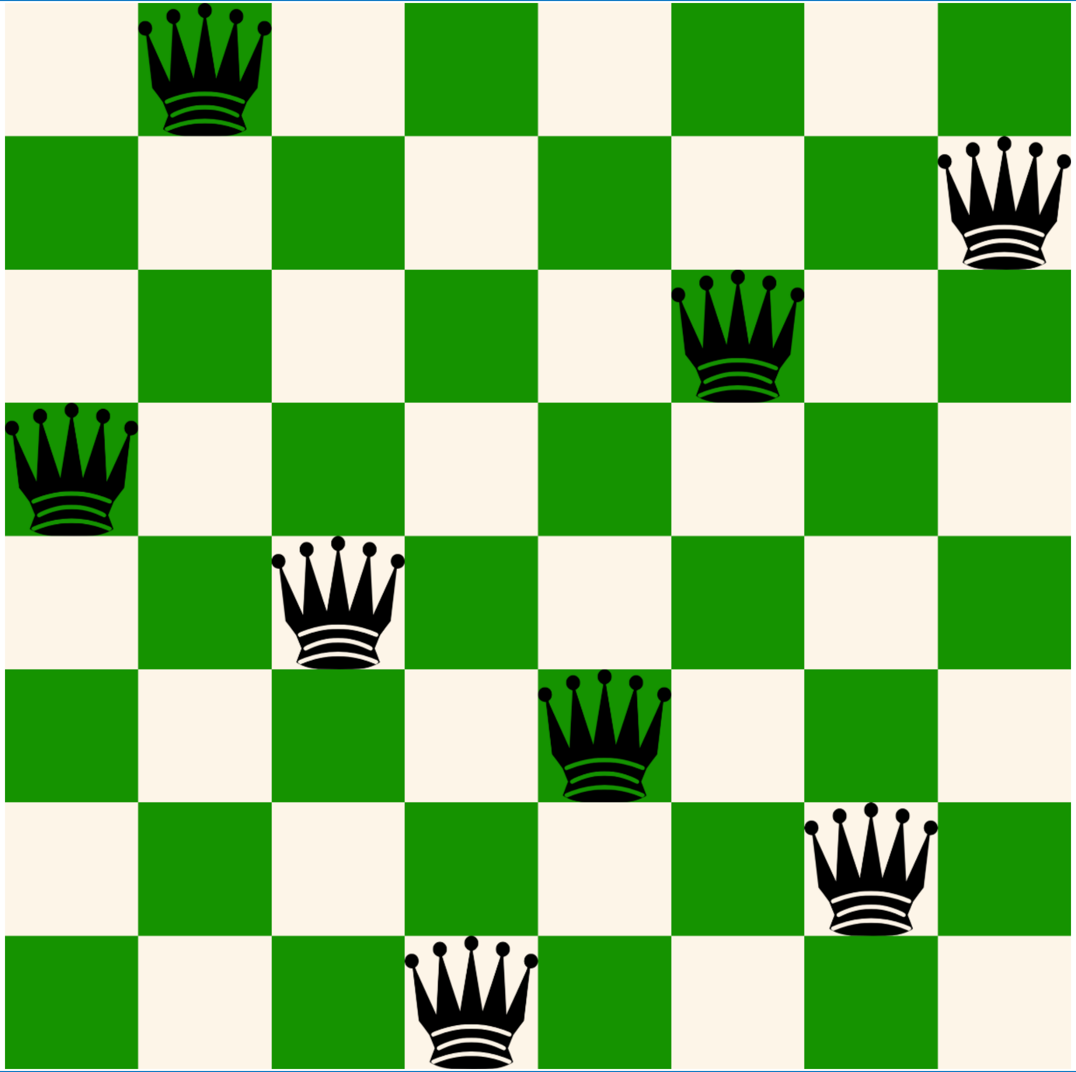
Yet Another Possible Answer

```python
def best(so_far, n):
    if len(so_far) == n:
        return 0 if check(so_far) else float('inf')
    else:
        return min(
            (1 if i == 0 else 0) +
            best(so_far + chr(ord('A') + i), n)
            for i in range(26)
        )

print(best('', int(input())))
```

N-Queens Chess Puzzle

- Put N queens on a chessboard, so that no two queens attack each other
- Let's try for fixed N first, say 8



Naïve Solution

- Each queen has 64 possible positions
- Use 8 nested for loops

```
for q0 in range(64):
  for q1 in range(64):
    for q2 in range(64):
      ...
      # Put queens on squares q0, …, q7 and check
```

- Takes $64^8 = 28{,}147{,}497{,}610{,}656$ iterations

An Observation

- Makes no sense to put two queens on same row
- Can narrow down search by only deciding columns

```
for q0 in range(8):
    for q1 in range(8):
        for q2 in range(8):
            ...
            # Put queen 0 on row 0 column q0,
            #     queen 1 on row 1 column q1, …
            #     queen 7 on row 7 column q7
```

- Takes $8^8 = 16{,}777{,}216$ iterations

Generalizing: A Recursive Solution

- This is basically the same as the "print all n-letter strings," except, instead of choosing from A-Z for the ith choice, choose from a column from 0 to $n - 1$
- The position (column) of the ith queen is like the ith letter of the string

Generalizing: A Recursive Solution

```
def solve(row, board):
  if row < n:
    for col in range(n):
      solve(row + 1, add_queen(board, row, col))
  else:
    if correct(board):
      print(board)
```

Representing the Chessboard

- The most obvious way is with an $n \times n$ grid
- There is a way to do it with a list of size n
  - How? Left as an exercise
  - Hint: all we need is to be able to check at the end if queens attack each other
- The details are not important for now
- But notice: no matter how you represent the board, each recursive call needs its own copy
  - So that decisions in one branch don't "pollute" a different branch

Sharing a Global Board

- Making copies uses a lot of memory and slows the solution down
- Don't want to make an already slow complete search solution even slower
- Instead, we can let all calls share a global chessboard
- To solve the "pollution" problem, before exploring a new branch, reset the board to whatever it was before we explored the previous branch
  - Can be done by undoing the most recent move after every recursive call (think about why)

Sharing a Global Board

```
board = make_global_board()

def solve(row):
    global board
    if row < n:
        for col in range(n):
            board.add_queen(row, col)
            solve(row + 1)
            board.remove_queen(row, col)
    else:
        if correct(board):
            board.print()
```

Notice the Difference

- `add_queen(board, row, col)` in the previous solution means create a new chessboard with the queen added, without changing the board passed to it
- `board.add_queen(row, col)` in the current solution means directly modify the chessboard
- Not only is this more efficient, but also easier to code
- Even if efficiency were not an issue, you might prefer to always do this

In General, Recursive Backtracking with Copies Looks Like This

```
def solve(p):
    if still_has_moves(p):
        for move in valid_moves:
            solve(apply(move, p))
    else:
        process(p)
```

While Recursive Backtracking with "Do-Recur-Undo" Trick Looks Like This

```
def solve(progress_counter):
    if has_more_steps(progress_counter):
        for move in valid_moves:
            global_object.do(move)
            solve(progress_counter + 1)
            global_object.undo(move)
    else:
        process(global_object)
```

Why This Works

- Remember, the computer executes each branch completely before moving to the next branch
  - Think about the actual backtracking process once to convince yourself it works
  - Then, you can go back to forgetting about it and just thinking about "parallel" processes
  - The do-undo steps become a template you "just follow" and trust that it works and makes things faster

Running Time Analysis

- Each queen has N possible independent positions
- Root of recursion tree has $1 = N^0$ node
- Queen 0 level has N^1 nodes
- Queen 1 level has N^2 nodes
- Queen 2 level has N^3 nodes
- Queen $N - 1$ level has N^N nodes

Running Time Analysis

- Each queen has N possible independent positions
- Level i of the recursion tree has N^i nodes
- The total is $N^0 + N^1 + N^2 + \cdots + N^N = O(N^N)$

Why $N^0 + N^1 + N^2 + \cdots + N^N = O(N^N)$

Expression	Number in base N
$N^0 + N^1 + \cdots + N^N$	$111 \ldots N+1$ times $\ldots 111$
$(N^0 + N^1 + \cdots + N^N)(N-1)$	(Letting $d = N-1$) $ddd \ldots N+1$ times $\ldots ddd$
$(N^0 + N^1 + \cdots + N^N)(N-1) + 1$	$1000 \ldots N+1$ times $\ldots 000$ also known as N^{N+1}

- So $(N^0 + N^1 + N^2 + \cdots + N^N)(N-1) + 1 = N^{N+1}$
- So $N^0 + N^1 + N^2 + \cdots + N^N = \frac{N^{N+1}-1}{N-1} = O(N^N)$

Running Time Analysis

- Each queen has N possible independent positions
- Level i of the recursion tree has N^i nodes
- The total is $N^0 + N^1 + N^2 + \cdots + N^N = O(N^N)$
- Without board sharing, spend $O(N^2)$ time per node to copy
- Total is $O(N^N) \cdot O(N^2) = O(N^{N+2})$

Running Time Analysis

- With board sharing, only $O(1)$ on nodes at levels $< N$
- $N^0 + N^1 + N^2 + \cdots + N^{N-1} = O(N^{N-1})$ such nodes
- Total for levels $< N$ is $O(N^{N-1})$
- Spend $O(N^2)$ at the last level
- Total for last level is $N^N \cdot O(N^2) = O(N^{N+2})$
- Total is $O(N^{N+2})$
  - The speed up isn't reflected by the Big-Oh, just because there are too many nodes in the last level and most of the processing happens there, but the speed up is there

Even Simpler Running Time Analysis

- There are N steps and each step multiplies the number of candidates by N, so there are N^N candidates
- We spend $O(N^2)$ at every candidate
- The total time is $N^N \cdot O(N^2) = O(N^{N+2})$
- This isn't always exact
  - Could be off by a linear factor or 2, especially if different amount of work done at last level vs. previous levels
- But the exponential is the most important anyway
  - Analyze further only if near time limit

Analyzing Recursive Backtracking Solutions

- In general, draw the tree, count how much work is done at each node, then take the sum
- The amount of work done for lots of nodes will be the same, so no need to add one-by-one, just multiply
- Even simpler:
  - Backtracking solutions will typically be "perform S steps, splitting into B branches at each step"
  - Means $\approx B^S$ nodes
  - Typically, work done at each node is roughly the same except for a linear factor, so total $\approx$ work per node $\times$ number of nodes

Speeding It Up Further: An Observation

- It makes no sense to continue putting more queens if some queens already attack each other
- Before placing a new queen, first check if it attacks others
- If it does, **prune** (skip) the corresponding branch
- Expressed differently: only make the recursive call if the new queen does not attack any others when placed at the currently considered position

N-Queens, with Pruning

```
def solve(row):
    if row < n:
        for col in range(n):
            if not has_attacking(board, row, col):
                board.add_queen(row, col)
                solve(row + 1)
                board.remove_queen(row, col)
    else:
        board.print()
```

Something to Think About

- Why is `if correct(board)` no longer needed before printing?

Pruning

- Word comes from gardening
- To prune is to cut off ugly branches from a plant

In General, We Can Write

```
def solve(progress_counter):
    if has_more_steps(progress_counter):
        for move in valid_moves:
            if not sure_fail(move):
                global_object.do(move)
                solve(progress_counter + 1)
                global_object.undo(move)
    else:
        process(global_object)
```

Analysis

- With pruning, the recursion tree becomes complicated, making it hard to properly analyze the running time
- In theory, we need to analyze every new problem on-the-fly, and solve a difficult counting problem to figure out the number of configurations processed
- In practice, we can just benchmark or intuitively feel that we skip a lot of configurations by pruning, to conclude it is fast enough

Practice Problems

- https://progvar.fun/problemsets/complete-search-recursive-backtracking

Thanks!