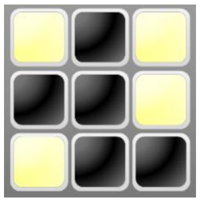
Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy
Grade 9/10 Session 3

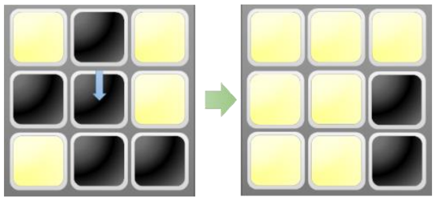
Session 3: Recursive Backtracking

Lights Out

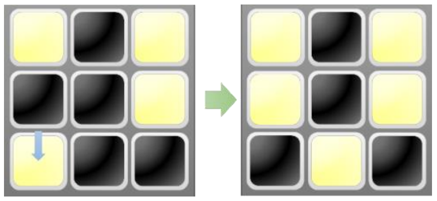


In the puzzle game Lights Out, you're given a grid of lights. Each light can be either on or off. The objective of the game is to turn all the lights off. Above is an example of a 3×3 Lights Out puzzle.

The lights can be manipulated by pressing them. Pressing a light toggles it and its immediate neighbors (directly north, south, east, or west only). All lights previously off will be turned on, and all lights previously on will turn off. Here are two examples.



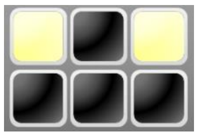
Pressing the center light will toggle it and all its neighbors.



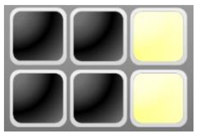
Pressing the lower-left light will toggle the three lights in the corner.

An interesting fact is that not all Lights Out puzzles are solvable. If the puzzle's grid is not a square, there are some starting configurations of the puzzle where it is impossible to turn all the lights off, no matter which lights you press.

Given a 2×3 Lights Out puzzle, you can find configurations which are not solvable.



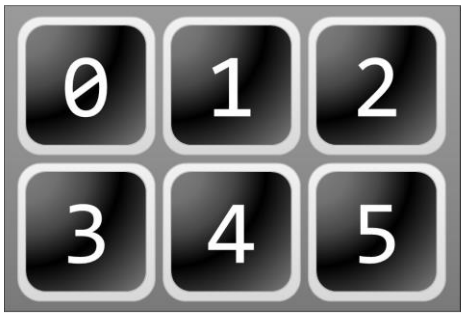
This 2×3 puzzle is **solvable**.



This 2×3 puzzle is **not solvable**.

How would you write a Python program which takes as input a 2×3 Lights Out puzzle, and checks if the puzzle is solvable or not?

To make things easier to follow, let's label the lights from `0` to `5`. Labelling elements of a problem like this can help us reason about the problem more concretely.

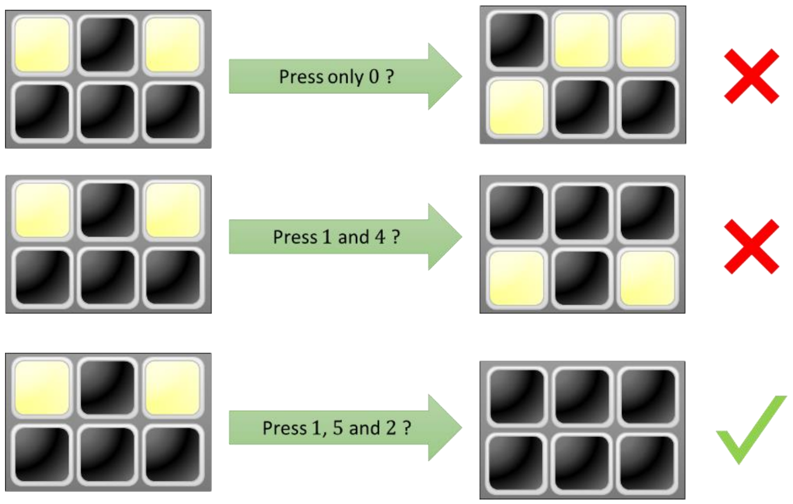


Exercise 1. To solve a Lights Out puzzle, is it necessary to press the same light more than once? Does the order in which you press the lights matter?

Answer.

- Pressing the same light twice is the same as not pressing the light at all, so it is possible to solve a Lights Out puzzle without pressing the same light more than once.
- The order in which you press the lights does not matter. If you press light `0` and then light `1`, it will toggle the lights the same number of times as if you pressed light `1` first and then light `0`.

These observations show us that in order to solve a Lights Out puzzle, we have press some or all of the lights once. We can try different combinations of lights:



Viewing the problem like this, we can phrase the problem in the form "*Out of these possible solutions, which one is correct?*" by saying **Out of all possible combinations of lights 0 to 5, which combination of lights can we press to turn all the lights off?**

A complete search solution would have to go through all possible combinations of lights 0 to 5.

In this code, the variable `light0` represents the number of times we press light 0, the variable `light1` represents the number of times we press light 1, the variable `light2` represents the number of times we press light 2, and so on and so forth.

```
for light0 in "01":
    for light1 in "01":
        for light2 in "01":
            for light3 in "01":
                for light4 in "01":
                    for light5 in "01":
                        check(light0 + light1 + light2 + light3 + light4 + light5)
```

Exercise 2. Explain, in your own words, why the code above correctly tries all possible combinations of lights.

However, this code is rather cumbersome to read and write, since we have so many nested loops. More importantly, this type of solution isn't *easily scalable*.

- What if we wanted to check if a 4×5 Lights Out puzzle is solvable? How many loops would we need to nest?
- What if we didn't know the size of the puzzle in advance, and we wanted to write a program which can handle any $m \times n$ Lights Out puzzle? What if we don't know how many loops we need to write?

Our solution would be easier to generalize if we could *easily modify how many loops are nested*. What we ideally want is some way to write code like the following. We need some sort of "super loop" which can nest code.

```
nest this n times:
  for lights_i in "01"
then do this at the end:
  check(light_0 + light_1 + light_2 + ... + light_n)
```

Rearranging our algorithm

Let's step back a bit and think about what our six nested for loops are doing. Let's try separating each loop from one another and interpreting what each loop does. Each loop makes a decision for only one light by picking either `"0"` or `"1"` then passes control off to the other loops.

`# Loop 0:`
`for light0 in "01":`
	Decide that `light0` is not pressed, then let loop 1 decide. When loop 1 finishes, decide that `light0` is pressed once, then let loop 1 decide again.

`# Loop 1:`
`for light1 in "01":`
	Decide that `light1` is not pressed, then let loop 2 decide. When loop 2 finishes, decide that `light1` is pressed once, then let loop 2 decide again.

`# Loop 2:`
`for light2 in "01":`
	Decide that `light2` is not pressed, then let loop 3 decide. When loop 3 finishes, decide that `light2` is pressed once, then let loop 3 decide again.

`# Loop 3:`
`for light3 in "01":`
	Decide that `light3` is not pressed, then let loop 4 decide. When loop 4 finishes, decide that `light3` is pressed once, then let loop 4 decide again.

`# Loop 4:`
`for light4 in "01":`
	Decide that `light4` is not pressed, then let loop 5 decide. When loop 5 finishes, decide that `light4` is pressed once, then let loop 5 decide again.

`# Loop 5:`
`for light5 in "01":`
	Decide that `light5` is not pressed, then check if the solution works. Then decide that `light5` is pressed once, then check if the solution works.

Notice that if we express it in this way, each loop only needs to worry about the loop immediately after it. The loops form a "chain", which we can express by each loop as a function.

```
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")
```

Exercise 3. Which function do we have to call to perform the complete search?

Notice that every function except for the last one does essentially the exact same thing. Whenever we see multiple functions which do similar things, we should consider combining them into one.

Exercise 4. How can you rewrite the functions `loop0`, `loop1`, `loop2`, `loop3`, `loop4`, `loop5` so that they are all expressed as a single function `loop(lights, x)`, where x is an integer from 0 to 5?

Solution. Loops 0 to 4 all do very similar things, so we can start by considering just those loops. If we treat x as the loop number, we would write a function that looks like this:

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

Note that we can run loop 0 by calling `loop("", 0)`. We can then consider that loop 5 is different from every other loop. We can handle this using an if statement.

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

Notice that we ended up with a **recursive function**, a function which calls itself. We'll discuss a more systematic way of coming up with recursive functions for complete search later. For now, let's compress our code inside the function back into loops.

```
def solve(lights, x):
    if x == 5:
        for k in "01":
            check(lights + k)
    else:
        for k in "01":
            solve(lights + k, x + 1)
solve("", 0)
```

Exercise 5. Since our solution is recursive, we should try to **blackbox** our understanding of how these functions work. Explain in your own words, what `solve(lights, x)` does, in terms of the integer $\boldsymbol{x}$. Explain the base case and recursive case for the recursive solution.

Possible answer. `solve(x)` checks all combinations of lights from light $\boldsymbol{x}$ to light $\mathbf{5}$, and considers pressing or not pressing each one.

- Base case. To check light 5, we consider not pressing it, and then consider pressing it once.
- Recursive case. To check all combinations of lights from light x to light 5
    - Consider not pressing light x, then consider all combinations of lights from light $x + 1$ to 5.
    - Consider pressing light x once, then consider all combinations of lights from light $x + 1$ to 5.

Exercise 6. Explain in your own words why we perform the complete search by calling `solve("", 0)`.

Exercise 7. Modify the recursive function `solve(lights, x)` to have an additional parameter `n` so that we can perform a complete search on a Lights Out problem with n lights, labelled from 0 to $n - 1$. What function call do we need to make to perform a complete search if there were 16 lights?

Solution. We simply replace `5` with `n - 1`, and also include `n` as an argument in our recursive case.

```
def solve(lights, x, n):
    if x == n - 1:
        for k in "01":
            check(lights + k)
    else:
        for k in "01":
            solve(lights + k, x + 1, n)
```

We would need to call `solve("", 0, 16)` to perform a complete search on 16 lights.

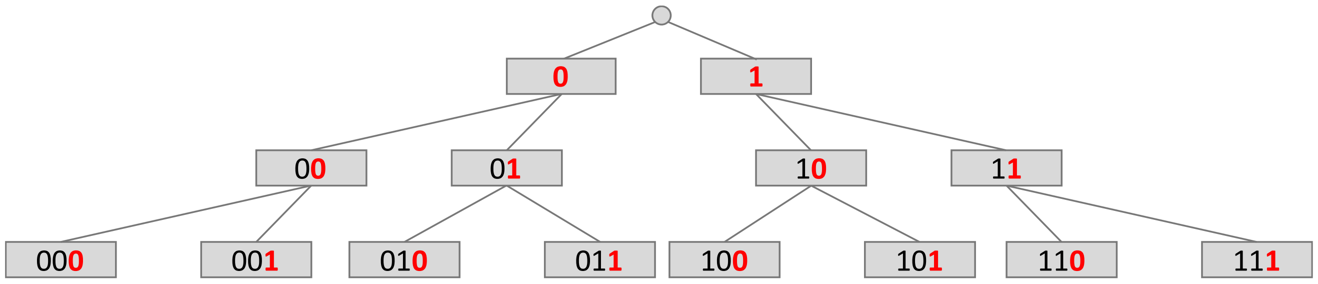
Exercise 8. You can express the recursive solution in a way such that the base case only contains a single `check()` statement. Explain in your own words why the following solution works. Remember that we have $\boldsymbol{n}$ lights, numbered from $\mathbf{0}$ to $\boldsymbol{n} - \mathbf{1}$.

```
def solve(lights, x, n):
    if x == n:
        check(lights)
    else:
        for k in "01":
            solve(lights + k, x + 1, n)
```

Traversing a search space with recursion

We've seen in the previous section that our nested-loop solution could be expressed through recursion, without using nested loops. Using recursion, we also easily handle cases where there are more than 6 lights without having to heavily modify our code. This is unlike our nested loop solution, where we didn't have a way to express the solution if we didn't know in advance how many lights we needed to consider. It seems like solving this problem more generally is done more naturally using recursion.

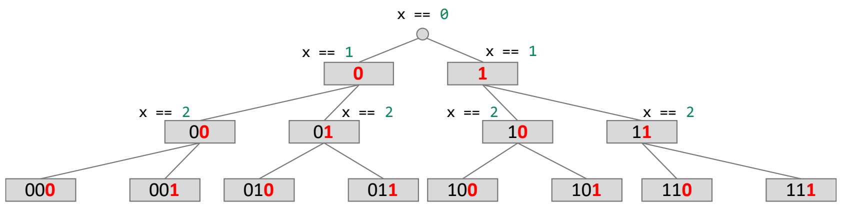
Recall that our problem is **Out of all possible combinations of lights 0 to 5, which combination of lights can we press to turn all the lights off?** To solve this problem using complete search, we have to somehow enumerate all combinations in our search space. We can visualize the search space of the Lights Out problem as a *tree*. We start at the top, the decide if each light is not pressed (`0`) or pressed once (`1`), in order. If we decide that the light is not pressed, then we go left. If we decide that the light is pressed once, we go right.



But we've seen when discussing recursive functions that we can also express traversing a tree like this recursively, branching in the same way at each step. We make one branch for each decision on a single light.

We break the problem of **enumerating a search space** into subproblems where we enumerate smaller and smaller search spaces. To enumerate all combinations of all lights, we make a decision for one light, then consider the smaller problem of enumerating all combinations for the rest of the lights.

```
def solve(lights, x, n):
    if x == n:
        check(lights)
    else:
        for k in "01":
            solve(lights + k, x + 1)
```

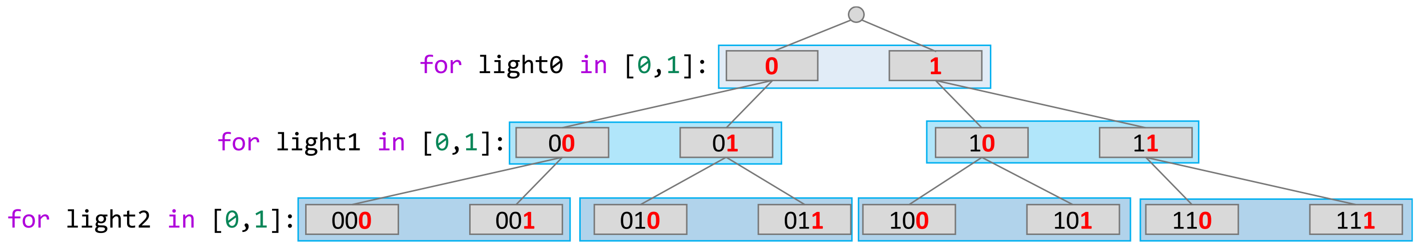


Key insight: Traversing a search space can be seen as making a systematic series of decisions on smaller elements of a problem at a time. If the number of decisions to make is too large or unknown in advance, and *the choices available at each decision step are basically the same*, we can naturally express the decision process with recursion, instead of using lots of nested loops.

Comparing iteration and recursion

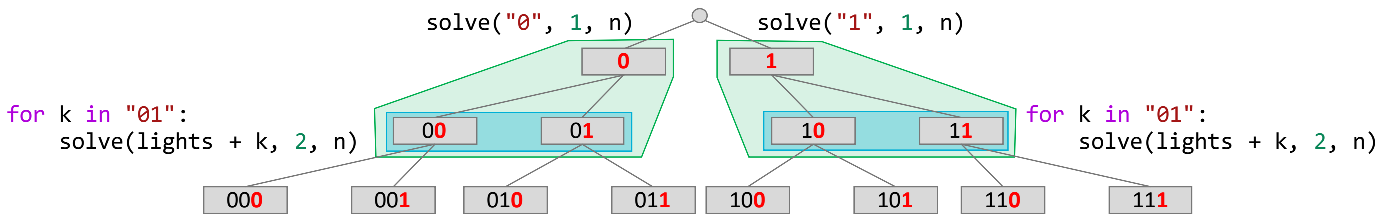
In the nested loop (or iterative) approach, we write one loop for each layer of this tree. When each loop decides the state of one light, it enumerates one layer of this search tree.

```
for light0 in "01":
    for light1 in "01":
        for light2 in "01":
            for light3 in "01":
                for light4 in "01":
                    for light5 in "01":
                        check(light0 + light1 + light2 + light3 + light4 + light5)
```



While we can traverse a tree by nesting loops, it's inconvenient because we have to write one loop for each level of the tree.

- Loops are well suited to moving **horizontally** across this tree, enumerating a fixed search space at each level.
- Recursive function calls are well suited to moving **vertically** down the tree, enumerating a search space of arbitrary depth.



Calculating the answer

So far, our complete search tries all possible solutions but doesn't yet answer the question "Is it possible to solve the Lights Out puzzle?" To complete the solution, we have to write a check function so that it returns `True` if pressing the chosen lights correctly turns all the lights off and returns `False` otherwise.

We would then answer the problem by keeping track of whether or not check returned `True` at least once. In an loop-based solution, we would do something like this:

```
solution_found = False

for light0 in "01":
    for light1 in "01":
        for light2 in "01":
            for light3 in "01":
                for light4 in "01":
                    for light5 in "01":
                        if check(light0 + light1 + light2 + light3 + light4 + light5):
                            solution_found = True

print(solution_found)
```

One way to do this in a recursive solution would be to return `True` if at least one of the decisions made for a light eventually led to a case where check returned `False`.

```
def solve(lights, x):
    if x == 6:
        return check(lights)
    else:
        solution_found = False
        for k in "01":
            if solve(lights + k, x + 1):
                solution_found = True
        return solution_found

solve("", 0, 6)
```

Exercise 9. Let's try to **blackbox** the `solve` function. In your own words, fill in the blanks to the following:

- `solve(lights, x)` returns `True` when _____________.
- `solve(lights, x)` returns `False` when _____________.

Possible answer.

- `solve(lights, x)` returns `True` when there is a way to press lights x to n to turn off all 6 lights.
- `solve(lights, x)` returns `False` when there is no way to press lights x to n to turn off all 6 lights.

Note about Lights Out: In this section, we showed a complete search solution, which tries many possible solutions. There's actually a smarter way to determine if an $m \times n$ Lights Out puzzle is solvable which does not require complete search.

Writing Recursive Functions to Enumerate a Search Space

Recursive functions and loops

Now that we've seen how recursion can naturally enumerate a search space, let's try practicing writing recursive functions to enumerate different search spaces for a complete search. When we discussed recursion previously, one of the basic examples we used was `count_down(n)`. Here is a slightly different version.

Code:
```python
def count_down(n):
    if n < 0:
        return
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+ str(n))
    count_down(n - 1)

count_down(5)
```

Output:
```
Counting down 5
Counting down 4
Counting down 3
Counting down 2
Counting down 1
Blast off!
```

Notice that the action performed by `count_down(n)` is similar to a loop that goes from `5` to `0`.

Recursion:
```python
def count_down(n):
    if n < 0:
        return
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    count_down(n - 1)

count_down(5)
```

Iteration (loop-based):
```python
n = 5
while n >= 0:
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    n = n - 1
```

Note that when writing a recursive function which works like a loop, we have to write several common parts.

We first think about **the range we want to go through**: in this case, the integers `5,4,3,2,1,0` in that order. We then ask ourselves:

Part:

Where do we **start**?
When writing a recursive function, our starting value is determined by our **initial function call**.

Recursion:
```python
def count_down(n):
    if n < 0:
        return
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    count_down(n - 1)
count_down(5)
```

Iteration (loop-based):
```python
n = 5
while n >= 0:
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    n = n - 1
```

How do we go to the **next element** in the range?

In a loop, we increment or decrement our counter.

When writing a recursive function, the next element in the range is determined by the **recursive function call**.

```python
def count_down(n):
    if n < 0:
        return
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    count_down(n - 1)
count_down(5)
```

```python
n = 5
while n >= 0:
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    n = n - 1
```

When do we **stop?**

In a loop, this is determined by the loop condition.

When writing a recursive function, we simply **don't call the recursive function** whenever we want to stop.

```python
def count_down(n):
    if n < 0:
        return
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    count_down(n - 1)
count_down(5)
```

```python
n = 5
while n >= 0:
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    n = n - 1
```

What do we do at each step?

For a simple example like count_down, this looks essentially the same.

```python
def count_down(n):
    if n < 0:
        return
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    count_down(n - 1)
count_down(5)
```

```python
n = 5
while n >= 0:
    if n == 0:
        print("Blast off!")
    else:
        print("Counting down "+str(n))
    n = n - 1
```

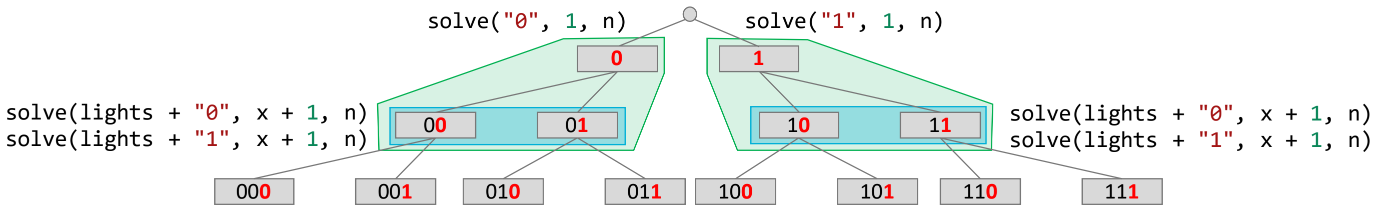
Key insight: Recursion allows you to enumerate a range, just like a loop. But recursion is more flexible than loops since you are able to **branch**.

You can "go to the next element" more than once.

Notice that in our solution to Lights Out, we call `solve(x+1)` **more than once**. There's no easy way to do this with a single loop.

Recursion generates *trees* since they're like a branching loop.

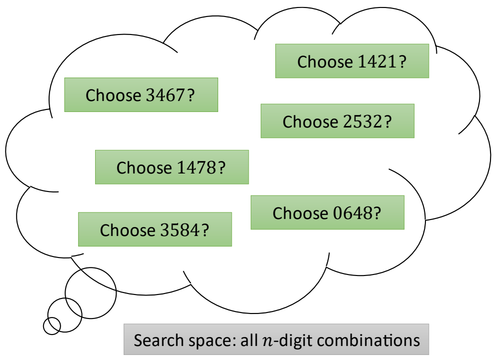
```python
def solve(lights, x, n):
    if x == n:
        check(lights)
    else:
        solve(lights + "0", x + 1, n)
        solve(lights + "1", x + 1, n)
```


solve("0", 1, n)
solve("1", 1, n)
solve(lights + "0", x + 1, n)
solve(lights + "1", x + 1, n)
solve(lights + "0", x + 1, n)
solve(lights + "1", x + 1, n)
0
1
00
01
10
11
000
001
010
011
100
101
110
111

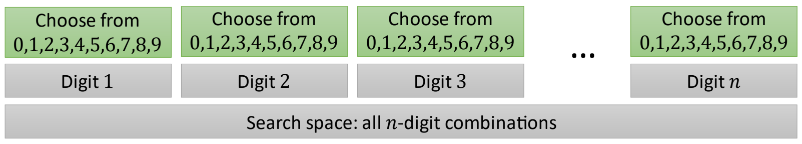
Return of the Combination Lock

With these in mind, we can come up with a general process to enumerate a search space recursively.

Let's go back to the problem of enumerating all possible combinations to a combination lock. When we discussed complete search, we considered a 4-digit combination lock. What if we wanted to brute-force an n-digit combination lock? (do not try this at home)



1. Determine the elements in the search space. These are your possible choices for solutions to the problem.


Choose from 0,1,2,3,4,5,6,7,8,9
Digit 1
Choose from 0,1,2,3,4,5,6,7,8,9
Digit 2
Choose from 0,1,2,3,4,5,6,7,8,9
Digit 3
...
Choose from 0,1,2,3,4,5,6,7,8,9
Digit n
Search space: all n-digit combinations

2. Think about **how to build one element in the search space**. Break it down into a series of smaller similar steps.

3. Determine the possible choices for each step.

   Just like when writing a loop, note down where you want to start, where you want to end, and when you want to go to the next step.

   Since you're building an element of the search space, you want to **write what you start building from** and **what you add at each step**. Then, **group together similar steps**.

```
How to build an n-digit combination:

Start at step 1 with an empty string.
Step 1 to step n:
    Choose a digit from 0,1,2,3,4,5,6,7,8,9 and add it to the string.
    Go to the next step
After step n:
    Print out the string
    Stop
```

4. Write a recursive function which makes a choice for **one step**, then calls itself for the next step.

   This recursive function is like a loop which builds an element of the search space from step 1 to step n.

```
# Example: enumerate all 4-digit numbers
def doStep(x , partially_made_string):
    if 1 <= x <= 4:
        for digit in "0123456789":
            doStep(x + 1, partially_made_string + digit)
    elif x > 4:
        print(partially_made_string)
doStep(1, "")
```

You can rearrange the if statements afterwards. For example:

```
def doStep(x, partially_made_string):
    if x > 4:
        print(partially_made_string)
    else: # steps 1 to 4
        for digit in "0123456789":
            doStep(x + 1, partially_made_string + digit)
doStep(1, "")
```

Key insight: When enumerating a search space recursively, you only need to think about **how to build one element of the search space**.

Exercise 10. Modify the following recursive function `doStep(x , partially_made_string):` above to perform the following tasks:

a) Enumerate 4-digit combinations where the only digit allowed is `0`.
b) Enumerate all 6-digit combinations.
c) Enumerate all 6-digit combinations where the first digit from the left is not zero.
d) Enumerate all 6-digit combinations where the second, fourth, and sixth digits from the left are even. (123456 and 222222 are included, but 134680 and 144570 are not.)

Exercise 11. Write a Python program that prints out all 7-letter strings which consist of one or more of the letters A, B, C, D, and E, repetitions allowed.

Solution. In a similar manner, we can write a function like the following. What's different about the solution below is that we count down instead of up, so our counter goes from 7 down to 0.

```
def generate(current_word, n):
    if n == 0:
        print(current_word)
    else:
        for next_letter in "ABCDE":
            generate(current_word + next_letter, n-1)
generate("", 7)
```

Expressing the function like this allows us to blackbox the function (phrasing its action in terms of its arguments): The function `generate(current_word, n)` will generate all strings that can be made by adding `n` letters to `current_word`.

Exercise 12. Modify the code from Exercise 11 from so that strings containing more than one B are not printed out.

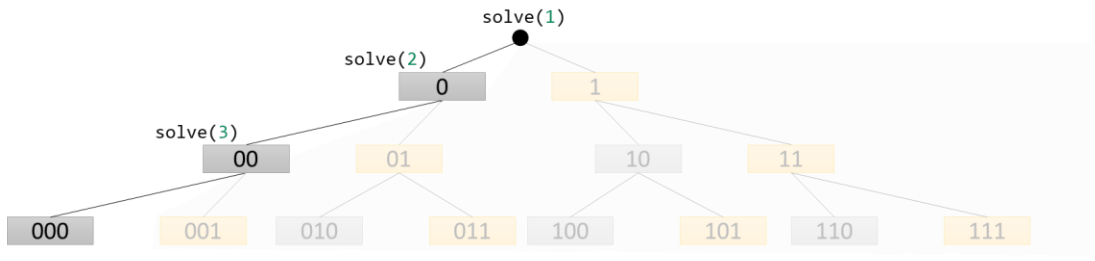
Solution. Depending on the current word so far, we simply change which functions we call to generate the remaining letters.

```
def generate(current_word, n):
    if n == 0:
        print(current_word)
    else:
        if "B" in current_word:
            for next_letter in "ACDE":
                 generate(current_word + next_letter, n-1)
        else:
            for next_letter in "ABCDE":
                generate(current_word + next_letter, n-1)
generate("", 7)
```

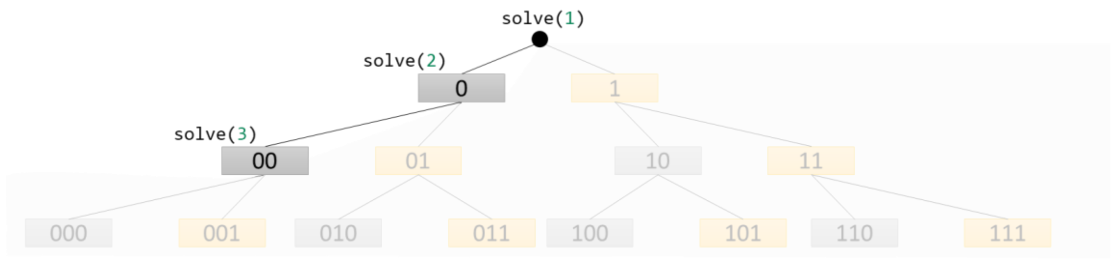
Recursive Backtracking

The method we've discussed which uses recursion to traverse a search space is called **recursive backtracking**, or sometimes simply just **backtracking**.

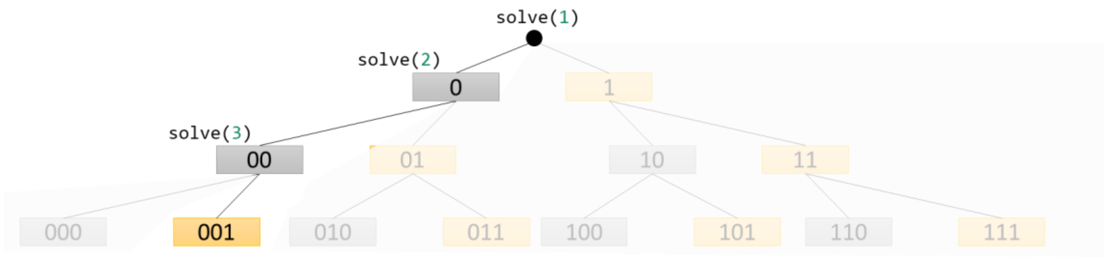
This approach is called backtracking since if we think about the order in which the search tree is traversed, every time we reach a dead end, we *backtrack* (trace our steps backwards) and try again from higher up in the tree.



The first branch that our solution to Lights Out tries is the left-most branch.



Then, we *backtrack*...

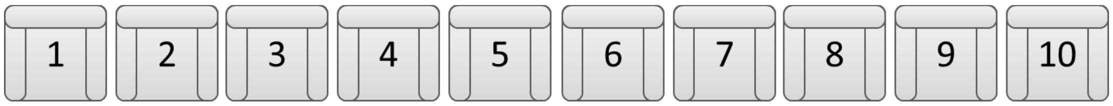


...and try the next branch.

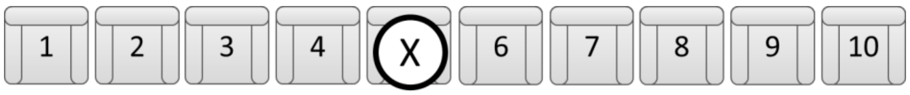
Now that we've discussed how to enumerate a search space using recursion in-depth, let's try solving recursive backtracking problems, applying what we know about complete search.

Enumerating seating arrangements

There are 3 groups of students on a field trip: Group A, Group B and Group C. All the students are about to watch a show. They have to be seated in a row containing 8 seats, numbered from 1 to 10. As one of the point persons for the field trip, you need to find a seating arrangement for the three groups of students. Only one person can sit in each seat.



One seat, x, is already taken by a stranger watching the show. In this example, $x = 5$.



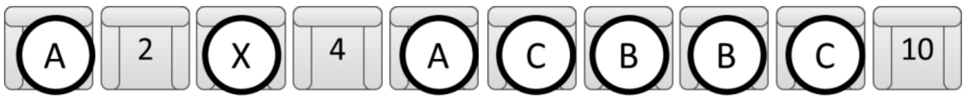
Suppose a, b, c are the numbers of students in Group A, Group B and Group C, respectively. There are at least 2 students in each group.

You need to find if there a way to place all $a + b + c$ students in seats such that:

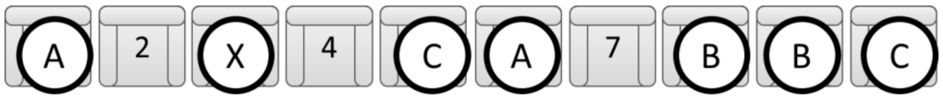
- Students in Group A are not next to any other student from Group A.
- Students in Group B are always next to at least one student from Group B.
- No student is next to a stranger.

Exercise 13. Write a Python program which takes in integers $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x}$ as input and prints all valid seating arrangements. (a seating arrangement is valid if it satisfies all the conditions)

For example, if $a = 2, b = 2, c = 2, x = 3$, that means Groups A, B and C have 2 students each, and the stranger is in seat 3. One valid seating arrangement is:

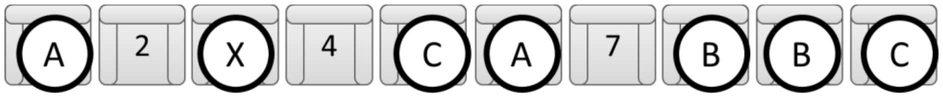


This is not the only valid seating arrangement. Another valid seating arrangement is:



Suppose we wanted to solve this problem using complete search. Our first question is: <u>What is our search space?</u> Our search spaces can be as large as we want, as long as they contain all possible correct solutions. In the previous sections, we learned how to enumerate strings, so we should try to express our search space as a set of strings.

We can represent a seating arrangement as a string with 10 letters, each letter corresponding to one seat. For example, the seating arrangement:



can be represented by the string `ANXNCANBBC`, where `N` represents an empty seat. Let's consider the search space: the set of all strings of length 10 which contain the letters A, B, C, N, or X.

First, we get the input.

```
a = int(input())
b = int(input())
c = int(input())
x = int(input())
```

Now, our search space is a set of possible seating arrangements. We can build one element of this search space by **deciding on one seat at a time.** For each of the ten seats from left to right, we do one of the following:

- Place a student from group A (add `"A"` to the string)
- Place a student from group B (add `"B"` to the string)
- Place a student from group C (add `"C"` to the string)
- Place a stranger (add `"X"` to the string)
- Leave the seat empty (add `"N"` to the string)

Similar to Exercise 10, we can write code like the following:

```
def generate(x, partial_seating_arrangement):
    if x > 10:
        print(partial_seating_arrangement)
    else:
        for letter in "ABCXN":
            generate(x + 1, partial_seating_arrangement + letter)
generate(1, "")
```

If you try running this code, you'll enumerate all possible seating arrangements. We need to perform some check so we only print out all <u>valid seating arrangements</u>. We need to make sure that:

- The number of students from Groups A, B and C are all correct (the integers a, b, c respectively).
- The stranger is in the correct position (the stranger is in seat x).
- Students in Group A are not next to any other student from Group A.
- Students in Group B are always next to at least one student from Group B.
- No student is next to a stranger.

Since many of these conditions rely on getting the **neighbors** of a student (the seats next to the student, we can first create a function `get_neighbors(arrangement, k)` which takes a seating arrangement and a seat number (k) then returns a string containing the seats next to seat k.

```
def get_neighbors(arrangement, k):
    if k == 0:
        return arrangement[1]
    elif k == 9:
        return arrangement[8]
    else:
        return arrangement[k-1] + arrangement[k+1]
```

Then, we can write a function `is_valid(seating_arrangement)` which returns `True` if a seating arrangement is valid, and `False` otherwise.

```
def is_valid(arrangement):
    global a, b, c, x
    # check number of students in each group
    if not (arrangement.count("A") == a):
        return False
    if not (arrangement.count("B") == b):
        return False
    if not (arrangement.count("C") == c):
        return False

    # check if there is only one stranger and they are in the the correct position
    # remember that the seats are numbered from 1 to 10
    if not (arrangement.count("X") == 1):
        return False
    if not (arrangement[x-1] == "X"):
        return False

    for i in range(10):
        # a student from group A cannot be next to another student from group A
        if arrangement[i] == 'A':
            if "A" in get_neighbors(arrangement, i):
                return False
        # a student from group B must be next to another student from group B
        if arrangement[i] == 'B':
            if "B" not in get_neighbors(arrangement, i):
                return False
        # no student must be next to a stranger
        if (arrangement[i] in "ABC") and ("X" in get_neighbors(arrangement, i)):
            return False

    # If all conditions are satisfied, return True
    return True
```

Finally, we can use our `is_valid` function in our recursive complete search so that we only output valid seating arrangements.

```
def generate(x, partial_seating_arrangement):
    if x > 10:
        if is_valid(partial_seating_arrangement):
            print(partial_seating_arrangement)
    else:
        for letter in "ABCXN":
            generate(x + 1, partial_seating_arrangement + letter)
generate(1, "")
```

Exercise 14. Write a Python program which takes in integers a, b, c, x as input and outputs a single integer: how many valid seating arrangements are there?

We can solve this problem by modifying `generate` so that instead of returning a Boolean, we return `1` if an arrangement is valid and `0` otherwise. Then, we can simply **add up** the total value of all calls to `generate`.

```
def generate(x, partial_seating_arrangement):
    if x > 10:
        if is_valid(partial_seating_arrangement):
            return 1
        else:
            return 0
    else:
        total_ways = 0
        for letter in "ABCXN":
            total_ways += generate(x + 1, partial_seating_arrangement + letter)
        return total_ways
generate(1, "")
```

Exercise 15. Write a Python program which takes in integers a, b, c, x as input and prints **`True`** if there is a valid seating arrangement, and **`False`** otherwise.

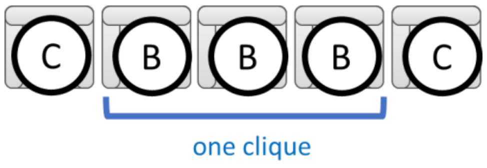
Exercise 16. Write a Python program which takes in integers a, b, c, x, y as input and outputs a string showing one possible valid seating arrangement. If there is no valid seating arrangement, output `IMPOSSIBLE`.

For this exercise, once we find a valid seating arrangement, we can assign it to a global variable.

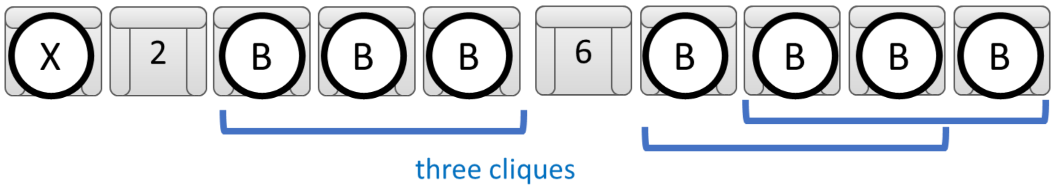
```python
answer = "IMPOSSIBLE"
def generate(x, partial_seating_arrangement):
    global answer
    if x > 10:
        if is_valid(partial_seating_arrangement):
            answer = partial_seating_arrangement
    else:
        for letter in "ABCXN":
            generate(x + 1, partial_seating_arrangement + letter)
generate(1, "")

print(answer)
```

Exercise 17. Let's say if three students from Group B sit in consecutive seats, we call those three seats a clique. Write a Python program which takes in integers a, b, c, x, y as input and outputs a string showing a valid arrangement. For this exercise, your seating arrangement must contain as few cliques as possible.



Here is an example of a seating arrangement with three cliques.



This problem is different since it asks us to find a "best" answer according to some criteria. Since our criteria has to do with the number of cliques, we leave it to you to write a function `count_cliques(arrangement)` which returns an integer containing the number of cliques.

Then, when we see many valid seating arrangements, we only keep ones that have a lower number of cliques.

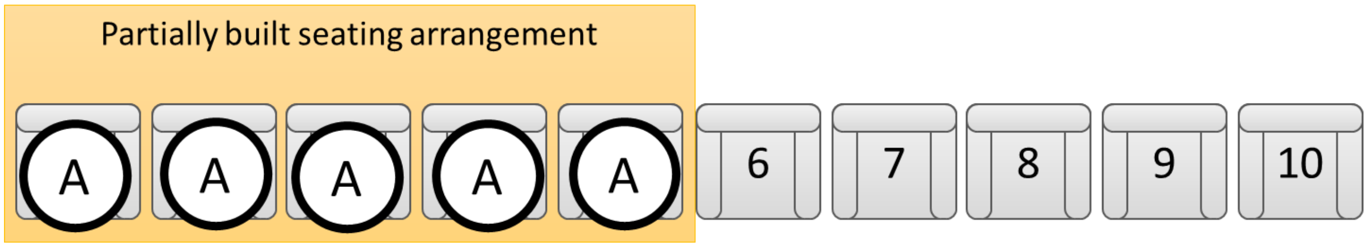
```python
answer = "IMPOSSIBLE"
def generate(x, partial_arrangement):
    global answer
    if x > 10:
        if is_valid(partial_arrangement):
            if answer == "IMPOSSIBLE"
                answer = partial_arrangement
            elif count_cliques(partial_arrangement) < count_cliques(partial_arrangement):
                answer = partial_arrangement
    else:
        for letter in "ABCXN":
            generate(x + 1, partial_arrangement + letter)
generate(1, "")
print(answer)
```

Exercise 18. Similar to Exercise 17, let's call three students from Group B sit in consecutive seats a clique. Write a Python program which takes in integers a, b, c, x, y as input and outputs an integer: the minimum number of cliques possible across all valid seating arrangements. If there is no valid seating arrangement, output `IMPOSSIBLE`.

Pruning

When we previously discussed complete search, we discussed how smaller search spaces are better. One way to reduce the size of a search space is to **check the correctness of partial solutions**.

Since we fill in the seats from left to right, without any conditions, it's possible that we might consider solutions which have the first five seats already filled with students from Group A.

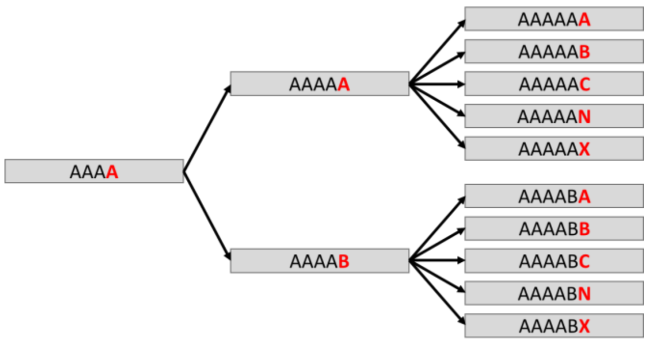


But what if we know that there are only 4 students in Group A? No matter how we complete this seating arrangement, we will not get a valid seating arrangement.

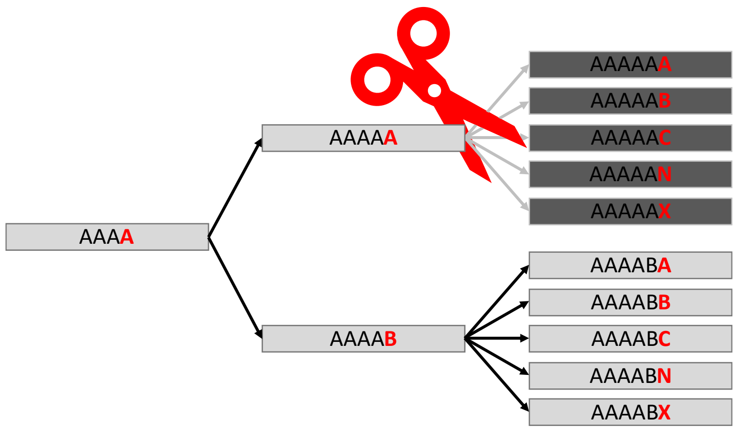
What happens if we return from our function immediately if there are more than the required number of students from Group A?

```
def generate(seat, partial_seating_arrangement):
    global a
    if seating_arrangement.count("A") > a:
        return
    if seat > 10:
        if is_valid(partial_seating_arrangement):
            print(partial_seating_arrangement)
    else:
        for letter in "ABCXN":
            generate(seat + 1, partial_seating_arrangement + letter)
generate(1, "")
```

So when our program tries to go through all possible seating arrangements...



If we know that $a = 4$, then our function will return if our current seating arrangement has 5 or more A's. This will **cut off** the search, and reduce the size of the search space!



Checking conditions early like this to reduce the size of the search space is called **pruning**. (In gardening, pruning means cutting off branches of a plant.)

Exercise 19. If we generate all valid seating arrangements with pruning based on the number of A's like in the function below:

```
def generate(seat, partial_seating_arrangement):
    global a
    if seating_arrangement.count("A") > a:
        return
    if seat > 10:
        if is_valid(partial_seating_arrangement):
            print(partial_seating_arrangement)
    else:
        for letter in "ABCXN":
            generate(seat + 1, partial_seating_arrangement + letter)
generate(1, "")
```

Do we still need to include

```
if not (arrangement.count("A") == a):
        return False
```

in our `is_valid(arrangement)` function?

Answer.

Yes, since our pruning does not remove cases with **less than** the required number of A's.

Exercise 20. Modify the `generate(seat, partial_seating_arrangement)` function so that we only consider seating arrangements where there is only one X, and it is in the correct location.

Solution. We can modify our code so it only adds "X" to the string at the correct location. Notice that we omitted X from the string "ABCXN" as well.

```
def generate(seat, partial_seating_arrangement):
    global a, x
    if seating_arrangement.count("A") > a:
        return
    if len(partial_seating_arrangement) >= x and partial_seating_arrangement[x - 1] != "X":
        return
    if seat > 10:
        if is_valid(partial_seating_arrangement):
            print(partial_seating_arrangement)
    else:
        for letter in "ABCN":
            generate(seat + 1, partial_seating_arrangement + letter)
generate(1, "")
```

Exercise 21. Given the generate function in Exercise 20, do we still need to include

```python
    # check if there is only one stranger and they are in the the correct position
    # remember that the seats are numbered from 1 to 10
    if not (arrangement.count("X") == 1):
        return False
    if not (arrangement[x-1] == "X"):
        return False
```

in our `is_valid(arrangement)` function?

Answer. No, we don't need this check anymore, since it is handled completely by our pruning.

Key insight: Checking conditions early while generating a search space can reduce the size of the search space.

Conclusion

As we've seen, recursive backtracking is a "more powerful" brute force, since it allows us to more finely generate possible solutions in the search space. However, we're still doing a complete search. The basic principles from the first session still hold:

- To apply complete search, express the problem in the form "*Out of these possible solutions, which one is correct.*" We need to define a search space.
- Instead of finding a smart way to solve the problem, we're relying on computational power to brute-force, so we only need to think about
  - How do we traverse the search space?
    - Break down "building an element of the search space" into smaller choices, then implement using loops or recursion.
  - How do we check if a solution is correct?
- We can easily adapt a complete search algorithm to solve other types of problems, like
  - How many correct solutions are there?
  - Which solution is the best (according to some criteria)?
- We can reduce the search space by checking partial solutions.

Recursive backtracking is just a new, more powerful tool used to **traverse a search space**. From the first session, one of the main reasons we would want to traverse the search space differently is to **reduce the size of the search space**, which in this session we also called **pruning the search space**.

Reducing the size of the search space is important because the smaller the search space is, the faster our programs can solve the problem. But how can we tell how fast is fast enough? Our next session will discuss **how to talk about the running time of programs**, which will help answer this question and eventually lead you to write faster, more efficient programs.