Flipping Game

Read the following Codeforces problem: Problem - 327A - Codeforces. Take a minute to think about how you would solve the problem. Proceed reading only when you are done thinking.

The problem boils down to the following question: "which segment to flip?" Intuitively, we want to flip as many zeros as possible and as few ones as possible. One natural greedy approach would be to find the longest segment containing only zeros and flip it, but this turns out to be wrong (see if you can find a counterexample)! Now you can try to come up with several different greedy approaches, but if you do this, you'll find that most of them fail.

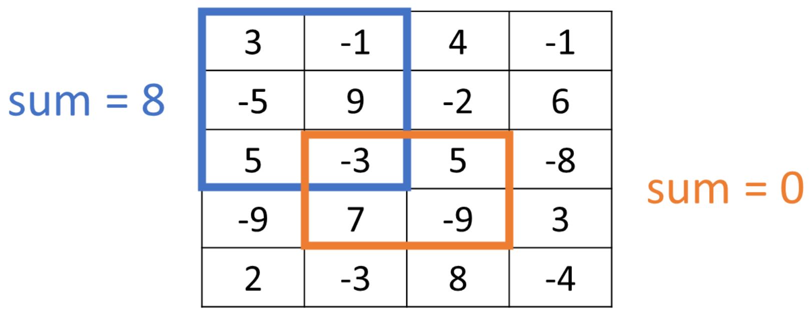
On the other hand, there is an easy solution, and it's very easy to see why it always gives the correct answer:

- Just try all possible segments to flip.
- For each segment, count the number of ones obtained.
- Keep the maximum over all the segments considered.

```
ans = 0
for i in range(n):
    for j in range(i, n):
        ans = max(ans, score_if_flip_segment(i, j))
```

The same strategy works for all problems! Examples:

1. How many numbers between 525,600 and 1,048,576 are divisible by 7?
   - Just go through all numbers! Increment a counter every time we see a number divisible by 7
2. Given a grid of $n \times m$ integers (may be negative), draw a rectangle so that the sum of the numbers inside the rectangle is maximized. The rectangle must have sides parallel or perpendicular to the lines of the grid.



3	-1	4	-1
-5	9	-2	6
5	-3	5	-8
-9	7	-9	3
2	-3	8	-4

   - Just try all possible rectangles! Keep track of the maximum sum seen so far.

The general strategy is to just try all the possible answers, and do the appropriate action on each one (if it satisfies the *constraints* of the problem):

- For *decision* (i.e. yes/no) problems, set a Boolean flag.
- For *optimization* problems, update the maximum or minimum.
- For *counting* problems, increment a counter.
- To simply enumerate all the solutions, put a candidate solution in a list, or print it out.

We call this strategy **complete search** or **brute force** or **trial-and-error** or **guess-and-check**. In the context of algorithms, the first two are the most common names.

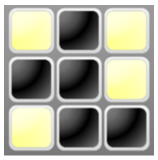
```
for candidate in possibilities:
    if valid_solution(candidate):
        # choose the appropriate action
        ans = True
        ans = min(ans, score(candidate))
        ans = max(ans, score(candidate))
        ans += 1
        correct_answers.append(candidate)
        print(candidate)
```

The idea is that computers are really fast, and won't get tired considering all the possibilities, so we can let them do all the work. Of course, the issue is that computers aren't infinitely fast, and this approach doesn't scale well. For many problems, Complete Search will be too slow. On the other hand, unlike a greedy algorithm, it's easier to guarantee the correctness of a Complete Search algorithm. Many problems do not even have greedy solutions, and it seems like we have no choice but Complete Search for these problems. Even if a greedy algorithm exists and would be faster, if the input size is small enough, going with a Complete Search algorithm saves us from worrying about correctness and needing proofs.

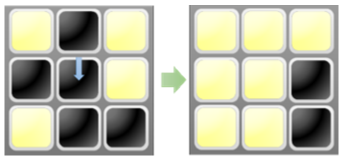
Let's call the set of all possibilities the **search space** of a problem. Often, the search space is not literally given to us as a list called `possibilities`, and we need to invent ways to *generate* the search space. In each example above, it was easy to generate the search space: just use loops. Sometimes, generating the search space requires more work.

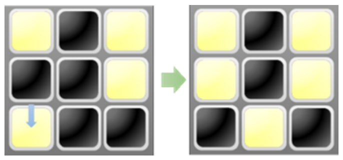
Lights Out

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. Below is an example of a 3 × 3 Lights Out puzzle.

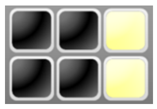


The lights can be manipulated by pressing them. Pressing a light toggles it and its immediate neighbors (directly north, south, east, or west only). All lights previously off will be turned on, and all lights previously on will turn off. Here are two examples.


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

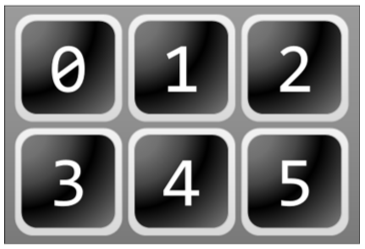

Pressing the lower-left light will toggle the three lights in the corner.

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.



This 2 × 3 puzzle is not solvable.

There is in fact a "greedy" solution for Lights Out (try thinking about it later), but let us consider how to solve it with Complete Search. Let's start with a fixed-size grid, say 2 × 3. To make things easier to follow, let's label the lights from 0 to 5. Labeling elements of a problem like this can help us reason about the problem more concretely.

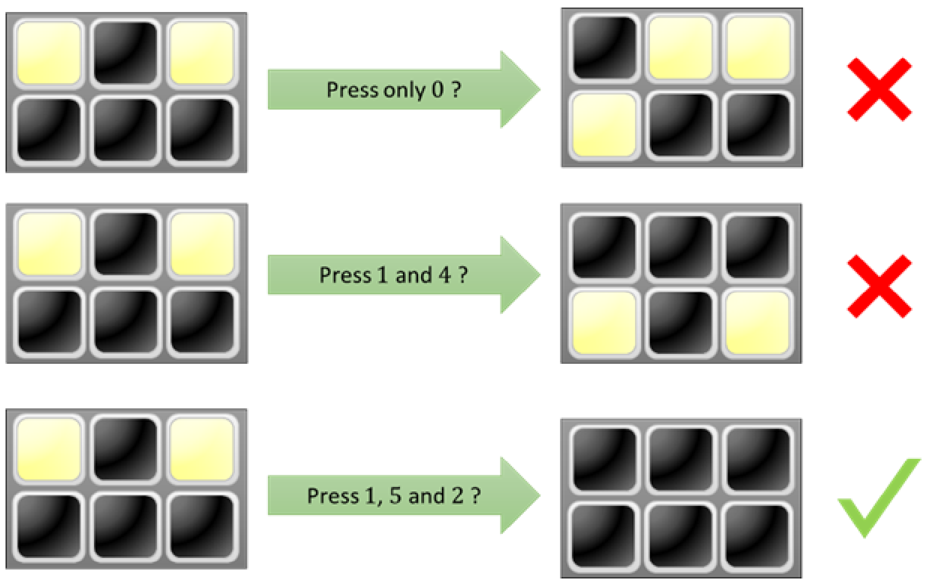


Exercise. To solve a Lights Out puzzle, is it necessary to press the same light more than once? Does it matter in which order you press the lights?

Answer. (highlight to reveal; optionally paste elsewhere and remove the highlight color)

[redacted]

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



A complete search solution would go through all possible combinations of lights.

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 [0, 1]:
  for light1 in [0, 1]:
    for light2 in [0, 1]:
      for light3 in [0, 1]:
        for light4 in [0, 1]:
          for light5 in [0, 1]:
            check([light0, light1, light2, light3, light4, light5])
```

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 that can handle any m × 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 light_i in [0, 1]:
```

then do this at the end:

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

Let's step back a bit and think about what our six nested for loops are doing. Let's try separating each loop from 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.

`for light0 in [0, 1]:`	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.
`for light1 in [0, 1]:`	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.
`for light2 in [0, 1]:`	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.
`for light3 in [0, 1]:`	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.
`for light4 in [0, 1]:`	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.
`for light5 in [0, 1]:`	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 writing 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])
```

Take a moment to convince yourself that these functions accomplish the same thing as the nested loops above. Verify by replacing `check` with `print` in both versions.

Exercise. 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.

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

loop(0, [])
```

Notice that we ended up with a **recursive function**.

Exercise. Explain in your own words, what `loop(x, lights)` does, in terms of the integer `x`.

Possible answer. [redacted]

Exercise. Modify the recursive function `loop(x, lights)` to solve the general problem with n lights.

Possible answer. [redacted]

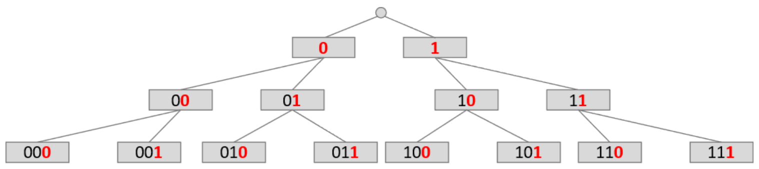
[redacted]

Exercise. Express the recursive solution in a way such that the base case only contains a single check statement.

Answer.

[redacted]

We can visualize the search space of the Lights Out problem as a *tree*. We start at the top, then 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.



We break the problem of **generating a search space** into subproblems where we generate 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.

Key insight: Generating a search space can be broken down into a series of decision points, separately trying all possible decisions at each point. If the number of decisions to make is too large or unknown in advance, and *the choices available at each point are basically the same across all decision points*, we can naturally express the process with recursion, instead of using lots of nested loops.

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.

Answer.

[redacted]

[redacted]

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).

Answer.

[redacted]

[redacted]

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.

Answer.

[redacted]

[redacted]

[redacted]

Alternative Answer.

[redacted]

[redacted]

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.

Answer.

[redacted]

[redacted]

[redacted]

Alternative Answer.

[redacted]

[redacted]

Another Alternative Answer.

[redacted]

[redacted]

[redacted]

Make sure you understand why all the answers above work before proceeding.

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

Yes! (Assuming we removed recursion limits from Python.) Here is a program that prints the sum of the first n positive integers, using loops:

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

print(ans)
```

Here is a program that does the same thing without loops:

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

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

Any loop can be transformed into recursion like this. The transformations you need to perform are somewhat mechanical even. Here's a side-by-side comparison. Notice how every piece of logic specified on the left has a corresponding piece on the right.

Initial Values	**`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)`**`)`

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	`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))`

Here’s another way to view what we were doing earlier. This “loopy recursion” we just wrote (and loops in general) lets us do one thing after another in series.

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

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 was made in each branch.

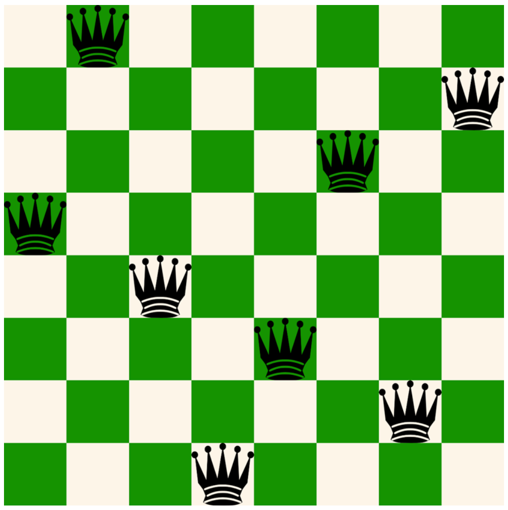
```
def do(args):
    do(update1(args))
    do(update2(args))
    do(update3(args))
```

In reality, these function calls aren't executed in parallel. Each branch of the recursion tree will be explored in full before the next branch is explored, like how DFS works. 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 a piece of code like this gets executed on normal computers, this style of solving problems is called **recursive backtracking**. However, you don't need to think about the backtracking process while coming up with the code. It is much easier to imagine that the computer literally spawns parallel processes at each recursive step and that a Complete Search algorithm written like this is literally trying all decisions in parallel.

If you drew the recursion tree of the "loopy recursion" above, you get a linked list. Loops are enough to generate search spaces that can be naturally represented as linked lists. For search spaces that are more naturally represented as trees, recursive backtracking is the way.

N-Queens

The problem: put N queens on an N-by-N chessboard so that no two queens attack each other. Here is a solution for N = 8.



Let's solve this for fixed N first, say N = 8. Each queen has 64 possible positions, so a complete search solution would require 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
```

We can speed this up with the following observation: it makes no sense to put two queens on the same row, so we can narrow down our search space by just deciding which column to put each queen in.

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

Now let's try to generalize with recursive backtracking:

```
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)
```

We skip the details of how exactly to represent the board, how to generate a new board with an added queen, and how to check if a board is correct. You can try filling in the missing details as an exercise. One thing you'll notice when you do this is that however you choose to represent the board, each recursive call needs its own copy of `board`. This is so that decisions in one branch don't "pollute" a different branch. However, making these copies is memory-intensive, and due to the way CPU caches work, actually significantly slows the solution down. We don't want to make an already slow complete search solution even slower. Fortunately, there is a simple trick to reduce the memory requirement. We can go ahead and let all calls share a global board. To solve the problem of "pollution," before exploring a new branch, we reset the board to whatever it was before we explored the previous branch. This can be done by undoing the most recent move after every recursive call (think about why this works).

```
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()
```

This “do-recur-undo” trick can be done on any backtracking solution. If a backtracking solution that does not share data structures across calls looks like this:

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

Then one that does 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)
```

Even when memory and running time are not of concern, this trick is still useful. Sometimes, it is easier to write code that directly modifies the data structure at each decision point, rather than making new copies.

We can speed up the N-queens solution with another observation: if some queens on the board already attack each other, it makes no sense to continue putting more queens. So before we place a new queen on the board, we can first check if it already attacks others. If it does, we can *prune* (skip) the corresponding branch. Expressed differently, we only make the recursive call if the new queen does not attack any others when placed at the currently considered position. The word **pruning** comes from gardening: to prune is to cut off ugly branches from a plant.

```
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()
```

Exercise. Why is the `if correct(board)` check no longer necessary before printing?

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)
```

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.