Recursive Backtracking

This reading is relatively new, and your feedback will help us improve it! If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch during office hours or open lab hours, or via e-mail at `6.101-help@mit.edu`.

Table of Contents

1) Introduction

Many problems can be solved by a trial-and-error method. Bertrand Russell, a famous mathematical logician of the 20th century, suggested that a bunch of monkeys banging away randomly at typewriters in the basement of the British Museum might eventually reproduce the text of Shakespeare's Hamlet if they work long enough. Computer scientists thus talk about the *British Museum Algorithm* as a baseline way to solve any problem for which all possible solutions can be enumerated: just try them all until one succeeds.

Computers are thought of as being really good at this kind of drudgery, but even their speed is daunted by large problems. For example, Hamlet contains 29,551 words, according to Wikipedia, and Shakespeare's known writings contain about 31,000 unique words. So, to generate Hamlet, our computer might use the following strategy to type a text that turns out to be Hamlet:

- While the story is shorter than Hamlet (in word count), try extending it by each one of the words in Shakespeare's vocabulary.
- If the story is equal to Hamlet, stop and declare success.

Unfortunately, the expected running time of this process is proportional to the number of possible stories with the same number of words as Hamlet --- which would be 31000^{29551}, or roughly 10^{132724} possible stories. No super-fast computer running for the lifetime of the universe could complete this task. For more realistic approaches, constraints along the way in examining possible solutions can dramatically cut down on such giant problems by cutting off lines of exploration as soon as they can be shown not to lead to solutions.

For example, if we had a perfect grammar checker for Elizabethan English, we could stop exploring a story as soon as we hit an ungrammatical patch in its construction.

In this reading, we explore methods to solve complex problems using recursion to reduce a larger problem to a number of smaller ones by trying each possible next step toward a solution and seeing if the resulting subproblem is solvable.

2) The Mixtape

The compact cassette tape, developed by Philips in the early 1960s, democratized music recording by allowing anyone with a relatively inexpensive cassette recorder to record or copy music and to share it with friends. A popular use of these tapes was to create a *mixtape*, copying songs perhaps from a variety of sources onto a collection that had special meaning. Gifts of mixtapes were common, so a tape of love songs could musically express feelings that a person might find difficult to share otherwise.

Here is an example of a tape one of the instructors made a long time ago:

Each tape, unlike today's almost infinite digital repositories, had a limited capacity (60 minutes in the image, which means 30 minutes on each side of the tape), and we were driven to include as much music as possible on a tape. So, the mixtape problem was defined as:

- from a collection of songs, choose a set that comes closest to filling a tape.

We use a variant of this problem in this reading, namely choosing a set of songs for a mixtape that exactly fills the capacity of (one side of) the tape. We will represent the collection of songs by a `dict` whose keys are the song titles and whose values are the lengths of the songs (in minutes). The function `mixtape`, then, is called with a song collection and the target duration:

```python
def mixtape(songs, target_duration):
    """
    Given a dictionary of songs (mapping titles to durations), as well as a
    total target duration, return a set of song titles such that the sum of
    those songs' durations equals the target_duration.

    If no such set exists, return None instead.

    >>> songs = {'A': 5, 'B': 10, 'C': 6, 'D': 2}
    >>> mixtape(songs, 21) == {'A', 'B', 'C'}
    True
    >>> mixtape(songs, 1000) is None
    True
    """
    raise NotImplementedError


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

This code is included in `mixtape.zip`, which you can use to follow along if you're interested. This file also includes several test cases and a `test.py` file.

Note the doctest, which specifies that we should be able to fill 21 minutes using songs `'A'`, `'B'` and `'C'`, but that we cannot fill a tape with a duration of 1000 minutes (there just aren't enough songs in our collection).

A note on the doctest: Normally, we might write the first test as

```
    >>> mixtape(songs, 21)
    {'A', 'B', 'C'}
```

but because Python does not guarantee to print the elements of a set in any particular order, we instead test for set equality and expect the result to print as `True`.

We approach this problem using recursion, which the previous readings covered. In such an approach, we need to identify any base case(s), which can be solved without further recursion, and the recursive case(s), where we can solve a smaller version of the problem to build up the solution to the original.

Check Yourself:

What base case(s) do you recognize in the mixtape problem?

How would recursion allow the solution of a subproblem to build the solution to the original problem?

Let's first focus on the recursive problem, using the trial-and-error approach. The basic idea is to pick one of the songs in the song list and assume that it will be part of the mixtape. That leaves us the problem of filling the rest of the tape (the remaining duration after subtracting the length of the chosen song) with the remaining songs.

Our first approach to implementing this idea in code might look as follows:

```
    song = list(songs.keys())[0] # pick an arbitrary song
    duration = songs[song]
    return {song} | mixtape({k: v for k, v in songs.items() if k != song},
                            target_duration - duration)
```

The `|` operator is being used here to take the union of two sets.

So, the answer to the recursive case is a set of songs, including the chosen song and the set chosen for the subproblem of filling the remaining duration. You will see that we'll need to tweak this a bit, but it's the right basic idea.

To construct the `dict` of songs remaining for the subproblem, we used a dictionary comprehension, iterating over each of the key/value pairs in `songs` and including all the ones except the chosen song. In case that format is new, it's worth mentioning that the following is an alternative that accomplishes the same goal:

```
    new_songs = dict(songs)
    del new_songs[song]
    return {song} | mixtape(new_songs, target_duration - duration)
```

This code avoids the dict comprehension, but it's critical to copy `songs`, because `del` mutates the dict it works on. Moving forward, we'll stick with the dictionary-comprehension-based approach since it makes it clearer that a copy is being made.

Observing that the target duration decreases on each recursive call gives us a clue about what a base case might be. If the `target_duration == 0`, then we have solved the problem and need to add no more songs to the solution set, so we can return the empty set. With this in mind, the body of the mixtape function would look like this:

```
    if target_duration == 0:
        return set()

    song = list(songs.keys())[0]
    duration = songs[song]
    return {song} | mixtape({k: v for k, v in songs.items() if k != song},
                            target_duration - duration)
```

You could try to run this version of the program with the doctest, but you will see that, although the first example succeeded, the second failed:

```
Failed example:
    mixtape(songs, 1000) is None
```

You'll see an error message saying that `IndexError: list index out of range` in the line `song = list(songs.keys())[0]`.

What is the core issue that leads to this error?

◯ `list` makes a new list containing `song.keys()` as a single element. We should use `[songs.keys()]` instead.

◯ `songs.keys()` can't be converted to a list.

◯ `songs` is empty.

◯ Indexing into `songs.keys()` is impossible.

This suggests a second base case: if we run out of songs when we still have `target_duration` to fill.

```
    if not songs:
        return _______ # < -- something
```

Given the specification of `mixtape` from its docstring, what should go in the blank above?

Now that we have changed our base case(s), we need to adjust our recursive case as well. Previously, we had been acting under the assumption that our recursive result would always be a set. But now that's no longer true! So let's adjust our code to try to account for this:

```
    recursive_result = mixtape({k: v for k, v in songs.items() if k != song},
                               target_duration - duration)
    if recursive_result:
        return {song} | recursive_result
    else:
        return None
```

After making this change, what will happen when running the first doctest, `mixtape(songs, 21)`?

--

After having made the change recommended in the explanation of the exercise just above, the code now passes both test cases. So are we done? Well, from the tone of this reading, you might well guess: No!

Passing two test cases is great, and we shouldn't discount the progress we've made so far. However, passing those two test cases (in fact, passing *any* number of test cases) isn't really a perfect guarantee that our code is working as expected. So we've got some more work to do.

Check Yourself:

Despite passing the two tests in the docstring, our code will still fail in some cases. Can you construct a different test case that would show a remaining error?

Show/Hide

Our algorithm, so far, actually does not explore all possibilities. When we chose `list(songs.keys())[0]` as the next song to try to use, our recursive call assumed that that song was part of the solution. So we're failing to consider any solution that doesn't include that first song!

So in some sense, it worked for our previous example only because we got lucky (because the first three songs chosen happened to fill the tape). In reality, we need to consider a couple of different possibilities in our recursive case(s):

1. The first song chosen is part of a solution, and we can solve the subproblem of filling the rest of the time.
2. That song *is not* part of a solution. In this case, rather than assuming that there is no solution, we ought to consider possibilities *that don't include that song*.

This method shows the essence of *backtracking*: try something, and if it does not work, try another alternative. This should resonate with what you remember from the recursion reading about searching trees of possibilities.

We implement this as follows:

```
def mixtape(songs, target_duration):
    """
    Given a dictionary of songs (mapping titles to durations), as well as a
    total target duration, return a set of song titles such that the sum of
    those songs' durations equals the target_duration.

    If no such set exists, return None instead.

    >>> songs = {'A': 5, 'B': 10, 'C': 6, 'D': 2}
    >>> mixtape(songs, 21) == {'A', 'B', 'C'}
    True
    >>> mixtape(songs, 1000) is None
    True
    >>> mixtape(songs, 10)
    {B}
    """
    if target_duration == 0:
        return set()

    if not songs:
        return None

    song = list(songs.keys())[0]
    songs_rest = {k: v for k, v in songs.items() if k != song}
    duration = songs[song]

    # if the first song is part of a solution, we can fill the remaining space
    # on the tape with the other songs
    recursive_result1 = mixtape(songs_rest, target_duration - duration)
    if recursive_result1 is not None:
        return {song} | recursive_result1

    # if the first song is not part of the solution, we should try to fill the
    # _whole_ duration with the other songs
    recursive_result2 = mixtape(songs_rest, target_duration)
    if recursive_result2 is not None:
        return recursive_result2

    # if there is no solution with the first song, and no solution without it,
    # then there must be no solution.  it's only now that we can safely return
    # None to indicate failure.
    return None
```

The first recursive case tests if we can solve the problem by including the selected song, and the second case tests if we can solve it by excluding that song.

This version does now solve the problem, though additional minor changes could improve it. For example, once the `target_duration` becomes negative, no solution on that path is possible, so we could adjust the failure case as follows:

```
    if target_duration < 0 or not songs:
        return None
```

Without this change, the code would *theoretically* work, but finding a failure case could take a *really* long time for cases where there is no solution.

3) An Alternative Approach

This latest version is a pure recursive solution, but some mixture of recursion and iteration is also possible. For example, instead of recursing on the "exclude this song" case, we could iterate over which song to include:

```
def mixtape(songs, target_duration):
    """
    Given a dictionary of songs (mapping titles to durations), as well as a
    total target duration, return a set of song titles such that the sum of
    those songs' durations equals the target_duration.

    If no such set exists, return None instead.

    >>> songs = {'A': 5, 'B': 10, 'C': 6, 'D': 2}
    >>> mixtape(songs, 21) == {'A', 'B', 'C'}
    True
    >>> mixtape(songs, 1000) is None
    True
    >>> mixtape(songs, 10)
    {B}
    """
    if target_duration == 0:
        return set()

    if not songs:
        return None

    for song, duration in songs.items():
        recursive_result = mixtape({k: v for k, v in songs.items() if k != song},
                                   target_duration - duration)
        if recursive_result is not None:
            return {song} | recursive_result
    return None
```

Here, we try each song in turn and return success if we can recursively succeed in filling the tape with others. Instead of failing if the first song does not work, we continue to try others, and we only fail if we have run through all the songs and none of them worked. Note that, strictly speaking, the `return None` at the end is not needed since any Python function that "falls through" its end automatically returns `None`, but in this case it is helpful to the reader to be explicit.

It is an often-used but dangerous assumption by programmers that a program that passes a handful of test cases must surely be correct. Passing more test cases can increase your confidence, especially if the test cases were crafted well to test a variety of situations, but it always remains possible to fail to anticipate a case on which the program might fail. There are techniques of proving program correctness under all possible cases, but that is a more advanced topic, for a later subject in your curriculum.

We have included 25 other, larger test cases in the code distribution, which you can run by invoking

```
pytest test.py
```

4) Yet Another Approach

We've seen a couple of alternative approaches already, but let's look at another couple of ways we could solve this problem before we move on to another example.

First, we focus on a method that you saw in some of our earlier code on recursion. In our solution above, we dive down into subproblems that, if solved successfully, return part of the answer (starting with an empty set by default, which could just as well be an immutable `frozenset`) and build up the entire answer as they return to higher levels of the recursion. Here we will consider an alternative in which the possible answer is constructed on the way down instead.

We accumulate in the set `so_far` the songs we have chosen to that point. The success base case is if their total duration equals the target, and the failure case is if that total exceeds the target. We follow the approach of iterating over all the songs and seeing if we can solve the recursive problem by adding that song to the `so_far` set in the recursion.

```
def mixtape(songs, target_duration, so_far = frozenset()):
    so_far_duration = sum(songs[s] for s in so_far)
    if so_far_duration == target_duration:
        return ______  # < -- what should go here?
    if so_far_duration > target_duration:
        return None

    for song, duration in songs.items():
        recursive_result = mixtape(songs, target_duration, so_far | {song})
        if recursive_result is not None:
            return recursive_result
    return None
```

Notice that we have equivalent checks to our base cases from our earlier version of the code here. Which parts match up? How are they different and how are they similar?

What should go in the blank above?

Sadly, even with the right value in the blank, testing this on the doctests (not shown here to save space) fails with a `RecursionError`. Why? When we pass `songs` down to our recursive cases, the iteration will always start by trying to add the same first song to `so_far`. Since adding the same element to a set does not change the set, each recursive subproblem is just identical to the previous problem, so this would lead to an infinite regress. When it reaches Python's recursion limit, Python cuts it off and raises `RecursionError`.

This problem is relatively easy to fix by adding a test to the iteration so we skip over songs that are already in `so_far`:

```
        if song in so_far:
            continue
```

Remember that `continue` tells Python just to go on to the next iteration of the `for` loop.

This version now solves the problem.

Check Yourself:

Unlike in our previous solution, we used no base case to check if we are out of other songs to try. Why does this code work despite not having that piece of code represented explicitly?

Show/Hide

5) An Iterative Mixtape Version

In our reading on graph search, we saw that it is possible, and often helpful, to maintain an agenda that keeps track of the program's state and its plans for what paths to search in the future. Here, we will develop a version of `mixtape` that takes just this approach. Then, we'll use this as a jumping-off point for a discussion of the similarities and differences between the recursive and iterative solutions to this problem.

Check Yourself:

Try to think about how we can use the agenda-based approach to construct a solution to this problem. Can you think of how the code might be structured? What will the elements in our agenda represent?

Show/Hide

Here is a working implementation of this idea (omitting the comment and doctests to save space):

```python
def mixtape(songs, target_duration):
    agenda = [set()]
    while agenda:
        this_mixtape = agenda.pop()
        duration = sum(songs[s] for s in this_mixtape)
        if duration == target_duration:
            return this_mixtape
        if duration > target_duration:
            continue
        for song in songs:
            if song not in this_mixtape:
                agenda.append(this_mixtape | {song})
    return None
```

Each element in our agenda represents a possible solution to the problem, in the form of a set of songs. As we pull possibilities from the agenda, we check the length of each:

- If it's the right length, we just return it (this is equivalent to the "success" base case in our recursive version).
- If it's too long, then we skip it, since neither it nor its descendents can possibly solve the problem (this is equivalent to at least part of our "failure" base case from above).
- If it's too short, we skip it but we add related sets to the agenda, each containing one more song.

When we run out of possibilities to try, we return `None`, signalling failure.

Is the code above doing a depth-first search, a breadth-first search, or something else?

BFS

5.1) A note on efficiency

Looking at the agenda while the program is solving a simple problem, we see the following successive states, showing the search tree that it is exploring:

```
[set()]
[{'A'}, {'B'}, {'C'}, {'D'}]
[{'A'}, {'B'}, {'C'}, {'A', 'D'}, {'B', 'D'}, {'C', 'D'}]
[{'A'}, {'B'}, {'C'}, {'A', 'D'}, {'B', 'D'}, {'A', 'C', 'D'}, {'B', 'C', 'D'}]
[{'A'}, {'B'}, {'C'}, {'A', 'D'}, {'B', 'D'}, {'A', 'C', 'D'}]
...
```

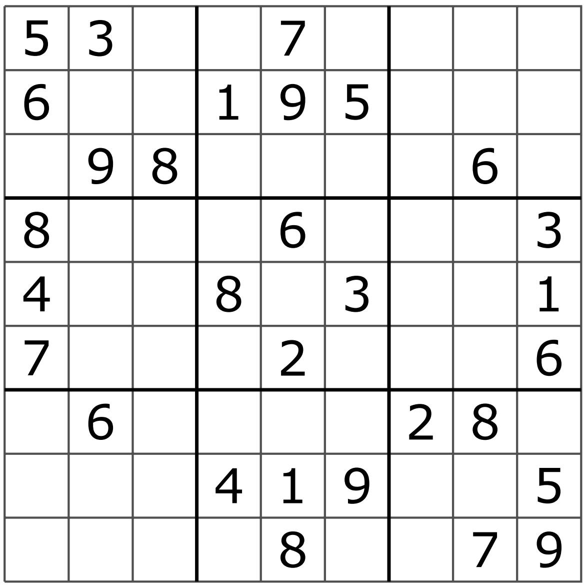
For the impossible task with `target_duration` = 1000, we iterate through 65 versions of the agenda before failing. It explores all combinations of the four songs, some of them multiple times, before failing. Keeping a set of already-tested mixtapes that failed (a visited set!) would allow us to reduce this redundancy.

6) Sudoku

To study another example that involves search and backtracking, we will build a solver for the popular Sudoku game. Sudoku is the name given to a variant of a digit-placement game seen already in French newspapers in the 19th century but popularized by a Japanese puzzle company in the 1980s. The goal is to fill every cell of a 9x9 board with a digit 1-9 under the following constraints:

- No row may contain any digit more than once.
- No column may contain any digit more than once.
- None of the 3-3 blocks that make up the 9x9 grid may contain any digit more than once.

Each of these regions holds a permutation of the digits 1 through 9, but of course they mutually constrain each other. In addition, some of the 81 cells are filled in with particular digits that must remain part of the solution. The filled-in squares do not themselves violate any of the above constraints, but their choices (how many are filled in, in which parts of the grid, and with what digits) determine the difficulty of any particular instance of the puzzle. It has been estimated that there are 6×10^{21} possible valid Sudoku grids. Here is just one example:



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

In this section, we'll walk through a process for designing a program to solve Sudoku puzzles. A fundamental question when starting to think about this problem is: "How do we represent the grid?"

We will begin with a familiar scheme, representing the board as a list of lists (i.e., a typical Python two-dimensional array), each grid cell holding an assigned digit. We'll use the digit `0` to indicate that a cell has not yet been filled. The solution to a Sudoku puzzle is just such a grid but one in which no `0` entries remain and all the constraints are satisfied.

Such puzzles are typically hard enough that they cannot be solved by inspection and require trial-and-error technique using backtracking to solve them.

Here is the overall strategy we can use:

1. Find a `0` in the board.
2. Try to fill it with one of the digits `1-9`.
3. Recursively try to solve that new board; if successful, we have our answer.
4. If not, iterate over the other digits.
5. If none of the digits work, then this path fails; return `None`.

Here is some code that attempts to implement this strategy:

```python
def solve_sudoku(board):
    """
    Given a sudoku board (as a list-of-lists of numbers, where 0 represents an
    empty square), return a solved version of the puzzle.
    """
    for row in range(9):
        for col in range(9):
            if board[row][col] != 0:
                continue
            # Found a 0; try filling it
            for trial in range(1,10):
                new_board = [[trial if (r, c) == (row, col) else board[r][c]
                              for c in range(9)]
                             for r in range(9)]
                if valid_board(new_board):
                    result = solve_sudoku(new_board)
                    if result is not None:
                        return result
            return None
    return board
```

Note that the outer iteration over `row` and `col` is used just to find the first open spot on the grid, not every open spot. Remaining ones will be filled by recursive calls to `solve_sudoku`. If there aren't any open spots, then we have succeeded and return the board.

Having found an open spot, we copy the existing board but with one of the 1-9 digits replacing the open spot. We then check if that new board satisfies the constraints of Sudoku. If it does, then we recursively try to solve that new board. If that works, we succeed and return the solved board. If it fails, we move on to a different trial digit. If we've run through all the digits, then there is no solution to the current board, so we fail by returning None.

We do need to implement a utility `valid_board` to check if the new board violates any of the constraints. This is straightforward, using another utility, `violation`, that sees if a list of numbers contains any duplicates (other than `0`).

Here are some examples to test the program:

```python
def format_sudoku(board):
    """
    Format a sudoku board to be printed to the screen
    """
    if not board:
        return 'Failed'
    _divider = '+'+''.join('-+' if i%3==2 else '-' for i in range(9))
    lines = []
    for i in range(9):
        if i % 3 == 0:
            lines.append(_divider)
        line = '|'
        for j in range(9):
            line += ' ' if board[i][j] == 0 else str(board[i][j])
            if j % 3 == 2:
                line += '|'
        lines.append(line)
    lines.append(_divider)
    return '\n'.join(lines)
```

```
grid1 = [[5,1,7,6,0,0,0,3,4],
         [2,8,9,0,0,4,0,0,0],
         [3,4,6,2,0,5,0,9,0],
         [6,0,2,0,0,0,0,1,0],
         [0,3,8,0,0,6,0,4,7],
         [0,0,0,0,0,0,0,0,0],
         [0,9,0,0,0,0,0,7,8],
         [7,0,3,4,0,0,5,6,0],
         [0,0,0,0,0,0,0,0,0]]

grid2 = [[5,1,7,6,0,0,0,3,4],
         [0,8,9,0,0,4,0,0,0],
         [3,0,6,2,0,5,0,9,0],
         [6,0,0,0,0,0,0,1,0],
         [0,3,0,0,0,6,0,4,7],
         [0,0,0,0,0,0,0,0,0],
         [0,9,0,0,0,0,0,7,8],
         [7,0,3,4,0,0,5,6,0],
         [0,0,0,0,0,0,0,0,0]]

grid3 = [[0,0,1,0,0,9,0,0,3],  # http://www.extremesudoku.info/sudoku.html
         [0,8,0,0,2,0,0,9,0],
         [9,0,0,1,0,0,8,0,0],
         [1,0,0,5,0,0,4,0,0],
         [0,7,0,0,3,0,0,5,0],
         [0,0,6,0,0,4,0,0,7],
         [0,0,8,0,0,5,0,0,6],
         [0,3,0,0,7,0,0,4,0],
         [2,0,0,3,0,0,9,0,0]]

import time
for grid in [grid1, grid2, grid3]:
    print(format_sudoku(grid))
    t = time.time()
    res = solve_sudoku(grid)
    elapsed = time.time() - t
    if res:
        print(format_sudoku(res))
    else:
        print('Failed')
    print(elapsed, 'seconds')
    print()
```

When run, these produce the following output:

```
+---+---+---+
|517|6  | 34|
|289|  4|   |
|346|2 5| 9 |
+---+---+---+
|6 2|   | 1 |
| 38|  6| 47|
|   |   |   |
+---+---+---+
| 9 |   | 78|
|7 3|4  |56 |
|   |   |   |
+---+---+---+
+---+---+---+
|517|698|234|
|289|134|756|
|346|275|891|
+---+---+---+
|672|849|315|
|138|526|947|
|954|713|682|
+---+---+---+
|495|362|178|
|723|481|569|
|861|957|423|
+---+---+---+
0.12484216690063477 seconds

+---+---+---+
|517|6  | 34|
| 89|  4|   |
|3 6|2 5| 9 |
+---+---+---+
|6  |   | 1 |
| 3 |  6| 47|
|   |   |   |
+---+---+---+
| 9 |   | 78|
|7 3|4  |56 |
|   |   |   |
+---+---+---+
+---+---+---+
|517|698|234|
|289|134|756|
|346|275|891|
+---+---+---+
|672|849|315|
|138|526|947|
|954|713|682|
+---+---+---+
|495|362|178|
|723|481|569|
|861|957|423|
+---+---+---+
1.297372817993164 seconds

+---+---+---+
|  1|  9|  3|
| 8 | 2 | 9 |
|9  |1  |8  |
+---+---+---+
|1  |5  |4  |
| 7 | 3 | 5 |
|  6|  4|  7|
+---+---+---+
|  8|  5|  6|
| 3 | 7 | 4 |
|2  |3  |9  |
+---+---+---+
+---+---+---+
|641|789|523|
|385|426|791|
|927|153|864|
+---+---+---+
|193|567|482|
|472|831|659|
|856|294|317|
+---+---+---+
|718|945|236|
|539|672|148|
|264|318|975|
+---+---+---+
1.5451231002807617 seconds
```

The exact timings will vary from one computer to the next, of course.

6.1) Improving Efficiency

Now, with a satisfied smile, we may look over the program and rejoice. However, no program is really ever done, because it is (almost) always possible to find improvements. For example, although our program works, it seems inelegant that when we find an empty spot, we blindly create boards for all possible digits at that spot and test each to see if they are valid. People solving Sudoku puzzles can look at an empty spot and immediately see that the possible choices are constrained by other digits already on the grid. For the first example above, we found a first empty spot at coordinates `(0, 4)` (next to the `6` in the first row). By looking at the values already filled in in that first row, fifth column, and the 3×3 subgrid, we can tell that the digit to fill in cannot be any of `5`, `1`, `7`, `6`, `3`, `4`, or `2`. Thus, we should try only the remaining digits, `8` or `9`, and avoid trying obviously bad choices that will then be rejected by `valid_board`.

So let's look at writing a version of this function that implements this strategy. A big part of doing so will involve first determining what numbers *cannot* go in a certain cell. To this end, we'll define some utility functions:

- `values_in_row(board, r)` returns the nonzero integers present in row `r` in the given board (`r=0` represents the top-most row). For example, in the example board above, `values_in_row(board, 1)` should return some collection containing `6`, `1`, `9`, and `5`.
- `values_in_column(board, c)` returns the nonzero integers present in column `c` in the given board (`c=0` represents the left-most column). For example, in the example board above, `values_in_column(board, 3)` should return a collection containing `1`, `8`, and `4`.
- `values_in_subgrid(board, sr, sc)` returns the nonzero integers in the subgrid identified by `sr` and `sc`. These are the coordinates of the subgrid in the 3x3 arrangement of subgrids in the overall grid. `sr` and `sc` are both in the range 0-2, i.e., `range(0, 3)`. `(sr=0, sc=0)` represents the top-left-most subgrid, and `(sr=2, sc=2)` represents the bottom-right-most subgrid. For example, in the example board above, `values_in_subgrid(board, 1, 2)` should return a collection containing `3`, `1`, and `6`.

In the box below, fill in the definitions for these functions. The return type is up to you, but it must be a collection and it must contain the correct numbers.

```python
def values_in_row(board, r):
    pass


def values_in_column(board, c):
    pass


def values_in_subgrid(board, sr, sc):
    pass

```

With these helpers in hand, we can figure out which values are allowed for a given grid and only try those values. This avoids the need to write the function that checks for a board's validity, which simplifies the code dramatically.

Instead, we define `valid_moves` to compute what digits are valid at that spot based on the same constraints that `valid_board` had used, but in a much simpler way:

```python
def solve_sudoku(board):
    """
    Given a sudoku board (as a list-of-lists of numbers, where 0 represents an
    empty square), return a solved version of the puzzle.
    """
    for row in range(9):
        for col in range(9):
            if board[row][col] != 0:
                continue
            # Found a 0; try filling it
            for trial in valid_moves(board, row, col):
                new_board = [[trial if (r, c) == (row, col) else board[r][c] for c in range(9)]
                              for r in range(9)]
                # no need to check for victory; new_board is valid by construction!
                result = solve_sudoku(new_board)
                if result is not None:
                    return result
            return None
    return board
```

Check Yourself:

This code relies on a function called `valid_moves` to tell us what numbers can go in a given spot. Using your functions from above, how should that function look?

Show/Hide

Here is one possible solution, assuming that the helpers above are returning sets:

```
def valid_moves(board, row, col):
    return (
        set(range(1, 10))  # all possible moves
        - values_in_row(board, row)  # exclude values already in the row
        - values_in_column(board, col)  # exclude values already in the column
        - values_in_subgrid(board, row//3, col//3)  # exclude values already
in the subgrid
    )
```

Of course, it's possible to make this work even if you chose a different type to return from your helper functions; but the choice of sets here really puts us in a good position to write `valid_moves` in a very concise way by using the built in `-` operation on sets (which implements a "set difference").

Although the resulting solutions differ a bit from what we had earlier, this is understandable. Iterating over a set likely puts digits in a different order than when we iterated over a range. Because these puzzles often have multiple valid solutions, that ordering can affect which solution we find. Nevertheless, in all three examples, the running time of this improved version is more than an order of magnitude less than of the original. And, not only is this solution more efficient than what we saw earlier, arguably an even bigger benefit is that it is simpler and more concise as well!

6.2) Improving Efficiency: Mutation

Following up the "no program is really ever done" theme, you might be troubled by all the boards we are still copying (though fewer than before). Could we instead reuse the same board over and over, mutating its values as needed? The answer is "yes", though one should undertake such an approach with great caution. Mutating data structures is fraught with opportunities for aliasing-related bugs, and such errors are often extremely difficult to debug. Nevertheless, we proceed!

Instead of constructing `new_board`, we will just modify `board`, but then we have to remember to "unmodify" it if filling in none of the trial values lead to success.

Show/Hide Line Numbers

```
def solve_sudoku(board):
    """
    Given a sudoku board (as a list-of-lists of numbers, where 0 represents an
    empty square), return a solved version of the puzzle.
    """
    for row in range(9):
        for col in range(9):
            if board[row][col] != 0:
                continue
            # Found a 0; try filling it
            for trial in valid_moves(board, row, col):
```

```
12                  board[row][col] = trial
13                  result = solve_sudoku(board)
14                  if result is not None:
15                      return result
16              board[row][col] = 0
17              return None
18      return board
```

This looks similar to our code from above, but line 12 mutates `board` instead of making a new board, and we have a new addition. Line 16 is not equivalent to anything from our earlier code; why is it even there?

This is a subtle point, but that line is *critical*. The high-level idea is that, each step along the way, we replace a `0` from the original board with some new number n. Then, recursive calls that we make (and *their* recursive calls and so on, all of which share the same `board`) assume that n is necessarily in that spot! If we eventually decide that there is no solution from that recursive call, we need to make sure to *undo* our action of putting n in that spot (so that other recursive calls no longer assume that n must be in that spot). Without setting that value back to `0`, we would get incorrect results (often, our code would indicate that there is no solution, even for boards that *can* be solved).

Again, there's a lot of danger here. So why bother? Well, in many cases, we might actually recommend *not* bothering, since the version that only uses immutable data types is generally much safer. But one reason that we might risk those dangers is that we get another substantial speed improvement by avoiding the repeated copying of our game board: in this case, around 3x!

Here is a table summarizing timing on our three examples with our original solution, the `valid_moves` change, and the mutating method:

	Original Code	**Only Valid Moves**	**Mutate Instead of Copy**
`grid1`	0.125	0.014	0.004
`grid2`	1.297	0.123	0.044
`grid3`	1.545	0.149	0.050

7) Summary

In this reading, our focus has been on a particular kind of recursive structure that we refer to as *backtracking*. This kind of structure typically contains a few pieces:

- A base case signifying a trivial "success."
- A base case signifying a "failure."
- Recursive case(s) that distinguish between those cases to try different possibilities for one small part of a subproblem (e.g., adding a single song to the mixtape or placing a single number in Sudoku). If that change leads to a failure in a recursive call, we try the next possibility; and only when all possibilities have failed do we signal failure.

We saw this approach applied to two different problems in this reading, and we saw that it is actually implementing a depth-first search, with Python's normal mechanism for keeping track of stack frames serving the role that the "agenda" served in our iterative graph-search programs.