Class 3 - Python Programming Review II

Guiding Genetic Algorithms with Language Models for Combinatorial Optimization

MISE Research Program - July 2025

Pedro Paredes

Recursive Search

Generating all DNA Strings

Given a number n, generate all DNA strings with n elements, e.g. 2 -> AA, AC, AG, AT, CA, CC, …

```python
def gen_strs(n):
    if n == 0:
        return ['']

    sol = []
    partial = gen_strs(n - 1)
    for base in ['A', 'C', 'G', 'T']:
        for dna in partial:
            sol.append(base + dna)
    return sol
```

Permutations

Given a list, generate all of its permutations, e.g. [1, 2, 3] -> [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]

```
1   def all_perms(items):
2       if len(items) <= 1:
3           return [items]
4       res = []
5       for i in range(len(items)):
6           first = items[i]
7           rest = items[:i] + items[i+1:]
8           for p in all_perms(rest):
9               res.append([first] + p)
10      return res
```

Backtracking

What is backtracking?

Strategy where we enumerate all possible solutions to a problem by **incrementally building** candidates to solutions

Very useful to find solutions to combinatorial problems (we'll see examples)

Alternate solution using backtracking

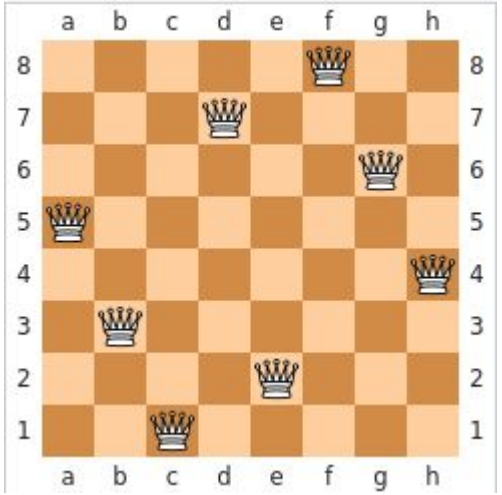
```
1   def gen_strs(current, n, sol):
2       if len(current) == n:
3           sol.append(current)
4           return
5
6       for base in ['A', 'C', 'G', 'T']:
7           gen_strs(base + current, n, sol)
```

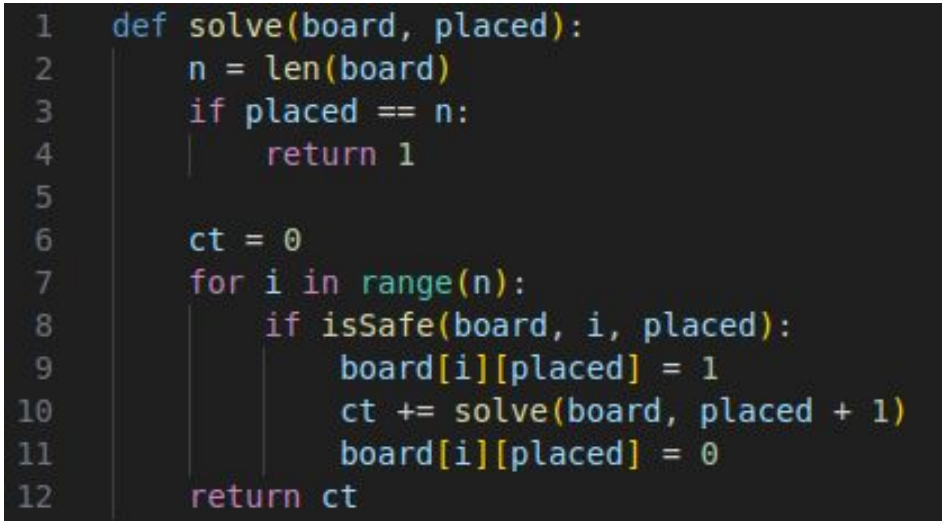
Notice how we build partial solutions (the parameter ‘current’) incrementally

Counting problems: the n-queens problem

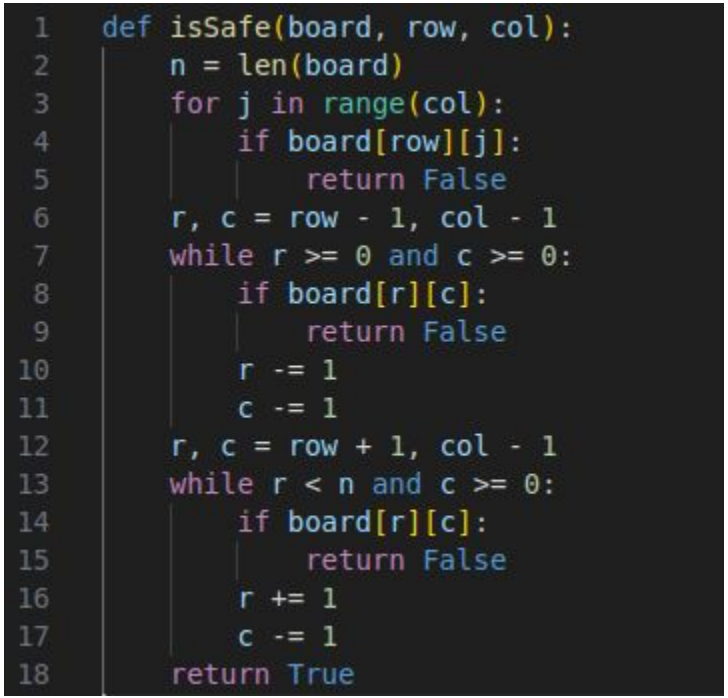
Consider a **n** by **n** chessboard where we want to place **n** queens such that they don't attack other (example on the right)

How many different ways are there to do so?





```
def solve(board, placed):
    n = len(board)
    if placed == n:
        return 1

    ct = 0
    for i in range(n):
        if isSafe(board, i, placed):
            board[i][placed] = 1
            ct += solve(board, placed + 1)
            board[i][placed] = 0
    return ct
```


def isSafe(board, row, col):
n = len(board)
for j in range(col):
if board[row][j]:
return False
r, c = row - 1, col - 1
while r >= 0 and c >= 0:
if board[r][c]:
return False
r -= 1
c -= 1
r, c = row + 1, col - 1
while r < n and c >= 0:
if board[r][c]:
return False
r += 1
c -= 1
return True

Depth First Search

Depth First Search

Search by at every step you choosing one unexplored option, follow it all the way until you can't continue, then backtrack to the last branching point and pick the next option.

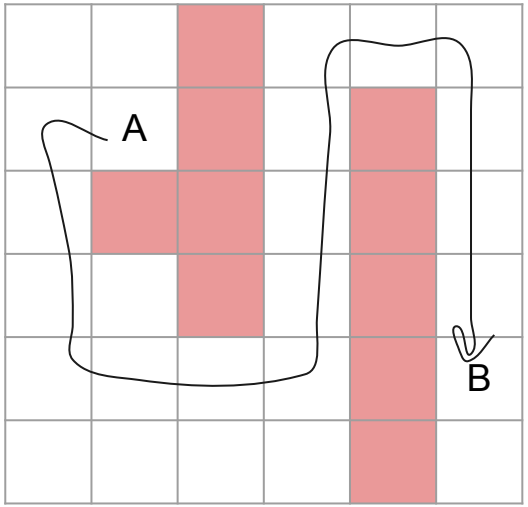
You always finish one complete branch before moving to its sibling branch, giving DFS a natural "deep first, breadth later" behaviour.

When is DFS useful?

- Generating or solving puzzles (Sudoku, mazes, etc).
- Enumerating every possible arrangement or path.

Example: Finding a Path on a Grid

Suppose you have a grid and you want to find a path from point A to point B avoiding obstacles

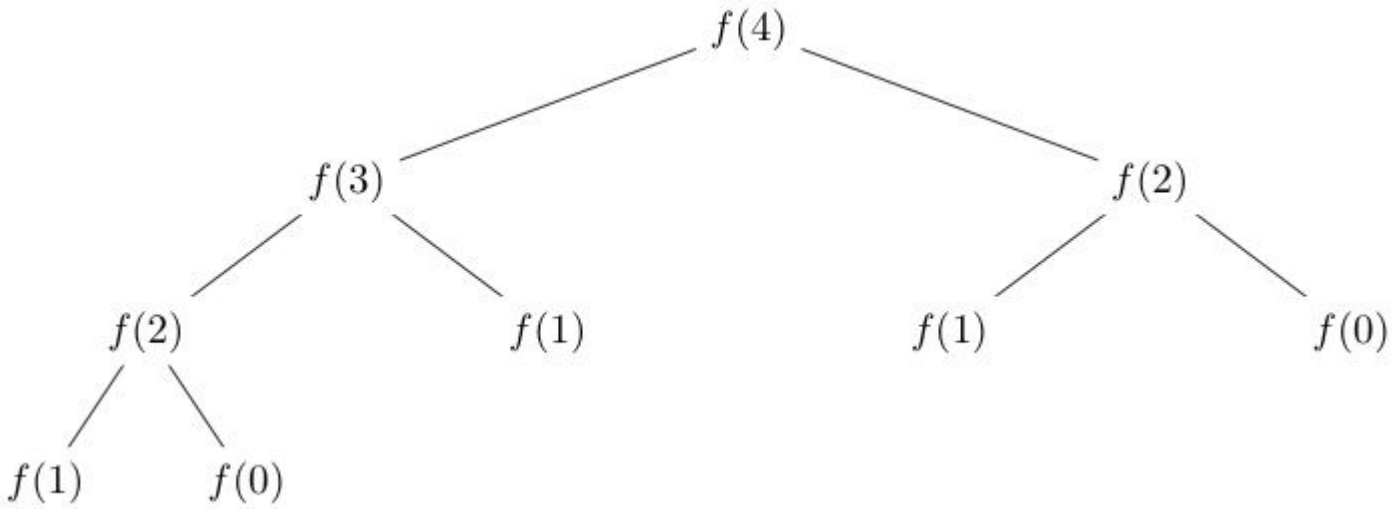


Code

```python
def dfs_paths(r, c, path, visited):
    # reached goal?  record path
    if (r, c) == (len(grid)-1, len(grid[0])-1):
        all_paths.append(path[:])
        return
    # explore four directions
    for dr, dc in ((1,0), (-1,0), (0,1), (0,-1)):
        nr, nc = r + dr, c + dc
        inside = 0 <= nr < len(grid) and 0 <= nc < len(grid[0])
        if inside and grid[nr][nc] == 0 and (nr, nc) not in visited:
            visited.add((nr, nc))
            path.append((nr, nc))
            dfs_paths(nr, nc, path, visited)
            path.pop()          # back-track
            visited.remove((nr, nc))
```

Dynamic Programming (optional)

Avoiding repeating actions



Avoiding repeating actions

- When we write recursive code we subdivide a problem into smaller subproblems
- Often there are a lot of repeated subproblems (like in the previous example)
- We can avoid having to recompute the solution to subproblems by storing it
- This is called Dynamic Programming

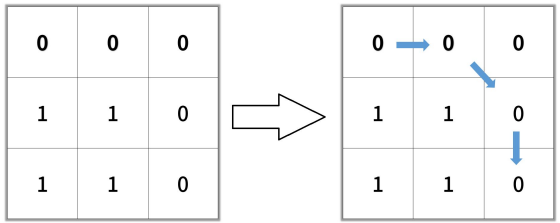
Back to the Fibonacci example

```python
dp = [-1 for i in range(10)]

def fib(n, dp):
    if dp[n] != -1:
        return dp[n]
    if n <= 1:
        dp[n] = n
    else:
        dp[n] = fib(n - 1, dp) + fib(n - 2, dp)
    return dp[n]

print(fib(9, dp))
print(dp)
```

```
34
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Challenge - Paths on a grid

- Let's suppose we have a **n** by **n** grid with integers
- We start at the top left corner of the grid and we want to go to the lower right
- We can move down or to the right
- Everytime we step on a grid cell we pay a cost equal to the cell's value
- What is the minimum cost path?



Complete the following

```
def path(grid, row, col):
    n = len(grid)
    if row == n and col == n:
        return grid[row - 1][col - 1]

    return grid[row - 1][col - 1] + min(path(???), path(???))
```

Solution

```
1  def path(grid, row, col):
2      n = len(grid)
3      if row == n and col == n:
4          return grid[row - 1][col - 1]
5
6      sol = 10000000000
7      if row < n:
8          sol = min(sol, path(grid, row + 1, col))
9      if col < n:
10         sol = min(sol, path(grid, row, col + 1))
11     return sol + grid[row - 1][col - 1]
```

this is correct, but … what's the problem with code?

How do we store repeated computation?

```
1  dp = [[-1 for i in range(n)] for j in range(n)]
2
3  def path(grid, row, col):
4      n = len(grid)
5      if row == n and col == n:
6          return grid[row - 1][col - 1]
7      if dp[row - 1][col - 1] != -1:
8          return dp[row - 1][col - 1]
9
10     sol = 10000000000
11     if row < n:
12         sol = min(sol, path(grid, row + 1, col))
13     if col < n:
14         sol = min(sol, path(grid, row, col + 1))
15     dp[row - 1][col - 1] = sol + grid[row - 1][col - 1]
16     return dp[row - 1][col - 1]
```

What's next?

Next class saturday

Another assignment sheet will be added to the website shortly

Class 4: Greedy Heuristics and Hill Climbing