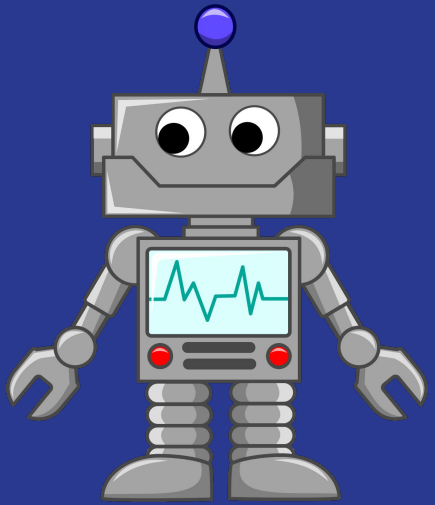
CS 9

Week 6 Problems

Andrew Benson
Ian Tullis



Announcements

- Ian will be in touch soon if you're falling behind on points
- See email from Trudy Gonzalez titled "May 12 - Public Interest Tech Career Fair!"

Want to combine your interest in tech and social impact or tech and policy?

Don't miss this unique opportunity to connect with a range of employers representing Public Interest Technology!

25 employers representing tech nonprofits, government, social enterprises, and for-profit PIT roles will be on hand to tell you about their work and opportunities at their organization or in their field more broadly.

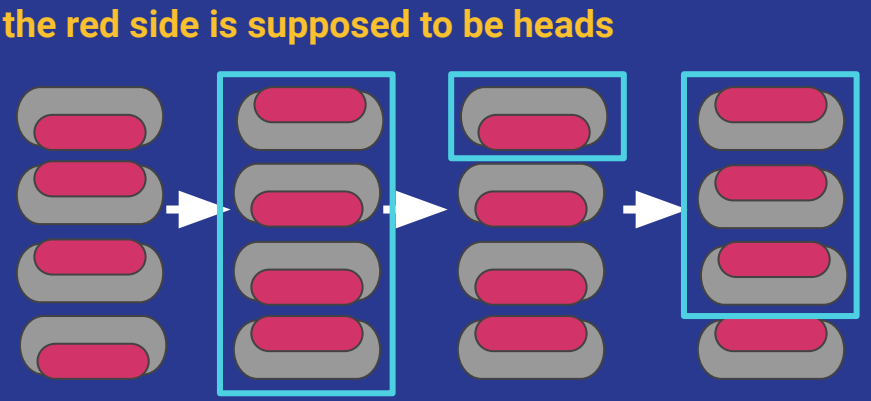
Learn about fellowships, jobs, internships, and career paths!

Sign up in advance for group presentations or 1:1 sessions on Handshake.

⭐⭐ Problem 6-1: Quarter Stack

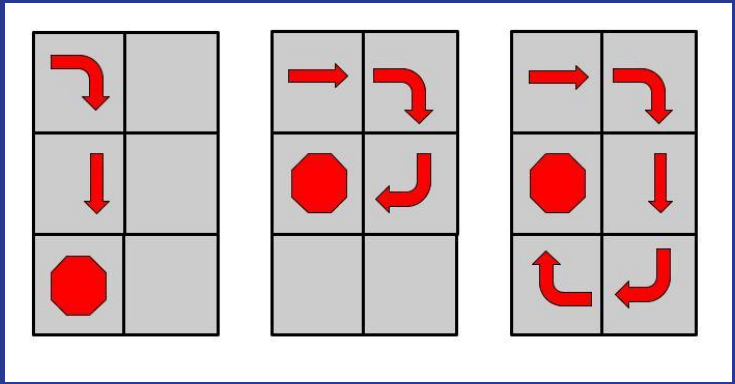
- You have a stack of N quarters ($1 \le N \le 10000$).
- Each quarter might currently be facing up (heads) or down (tails).
- You have only the following move available: pick up some substack of quarters off the top of the stack, flip that entire substack over (i.e. not individually!), then place that back on top of whatever is left of the stack. (It's OK for the substack to be the entire stack.)
- What is the smallest number of moves to get all the quarters heads-up?

- Example:
  - T: Answer 1.
  - HH: Answer 0. (nothing to do!)
  - THHT: Answer 3.
    - e.g., flip the entire stack to get HTTH, then flip the first to get TTTH, then flip the first three to get HHHH.
    - can show this is unbeatable.



⭐⭐ Problem 6-2: Turnbot

- A robot begins in the upper left cell of a R row by C column (R, C ≤ 25) grid, facing right. The robot only knows how to do two things: **move one cell forward,** or **make a 90 degree clockwise turn and then move one cell in that new direction**. The robot cannot exit the grid or revisit an already-visited cell. Whenever one of the two moves is not possible, the robot must choose the other one. If neither move is possible, the robot's journey ends.
- How many unique journeys are possible? (You just need to find the number.)
- Example:
  - Input: R = 3, C = 2
  - Output: 3



⭐⭐ Problem 6-2: Turnbot

- Hints
  - The robot can only turn right. Is the robot able to simulate a left turn?
  - Do you see any opportunities to model pieces of the problem recursively?

⭐⭐⭐ Problem 6-3: Corner Sum

- You have an NxN grid of integers ($2 \le N \le 1000$) in the range $[-10^9, 10^9]$.
- Find four distinct cells that form the corners of a rectangle, such that the sum of **those four cells** is as large as possible.

- (this is not asking for the sum of that entire subarray! just the corners)

3	-1	1	10
5	-5	6	9
5	2	-2	3
9	13	3	7

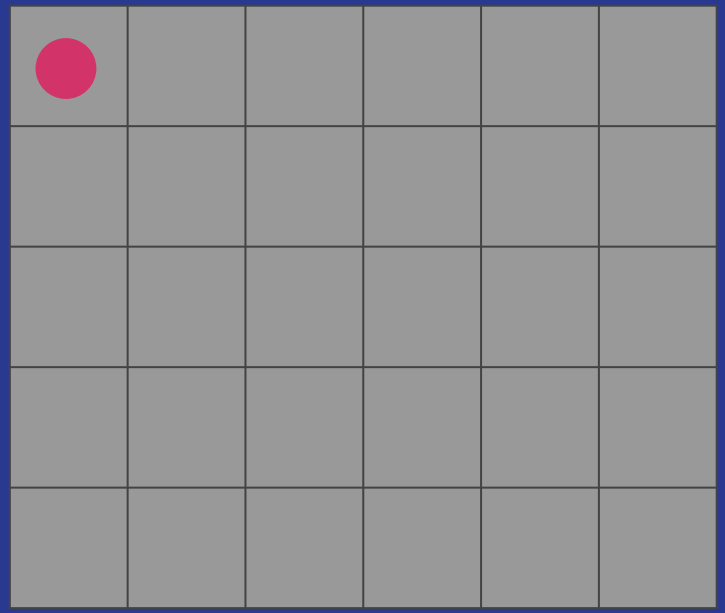
Solutions to 6-2 (discussed in-class)

⭐⭐ Problem 6-2: Turnbot

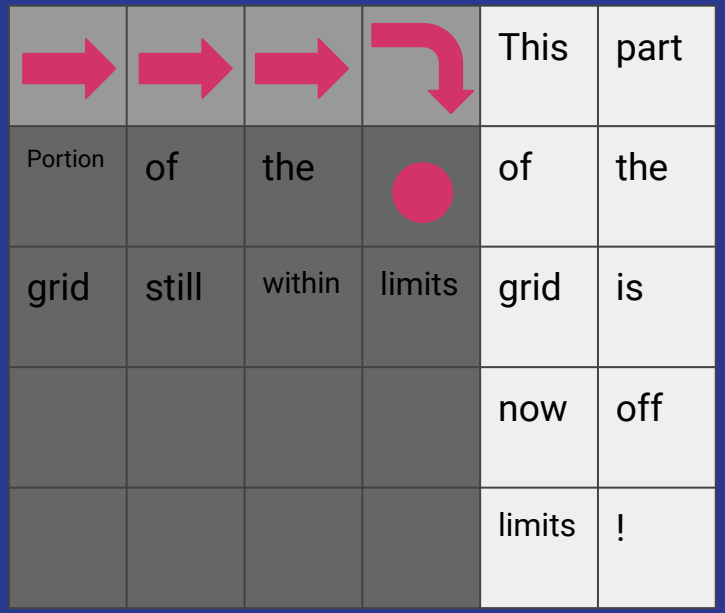
- The number of solutions can get quite large (exponentially so) as the grid size increases.
- So any solution that enumerates / visits them all explicitly will also be exponential!
- How can we do better?

⭐⭐ Problem 6-2: Turnbot

- Key Insight: Once the robot makes a right turn, it cuts off a huge fraction of the grid from future traversal.
- In fact, the portion of the grid left to be traversed looks rather familiar...

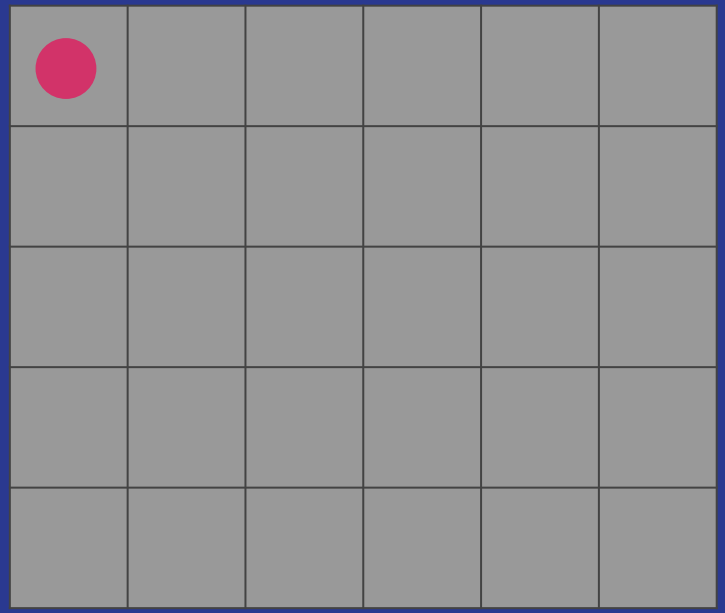


4 moves later

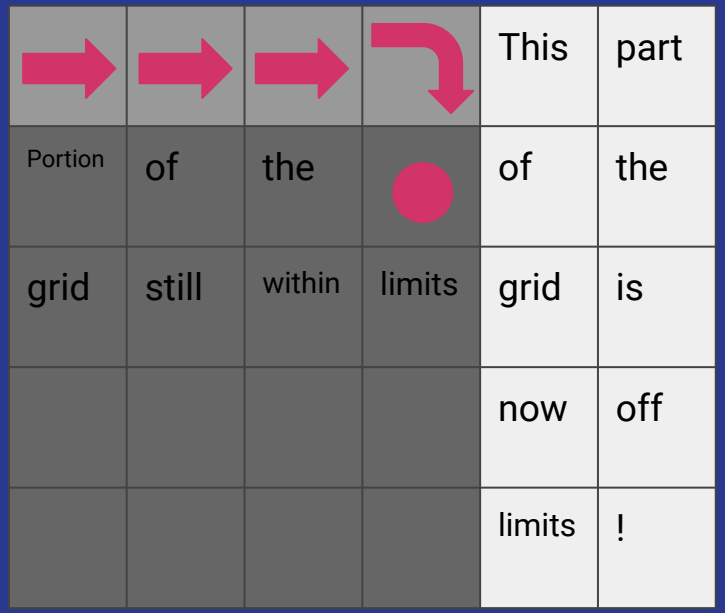


⭐⭐ Problem 6-2: Turnbot

- The dark gray portion of the grid is where the robot can keep traveling, but this acts functionally the same as the original problem, with a smaller grid, rotated 90 degrees.
- numJourneys(5, 6) after 3 forwards and 1 right turn = numJourneys(4, 4)

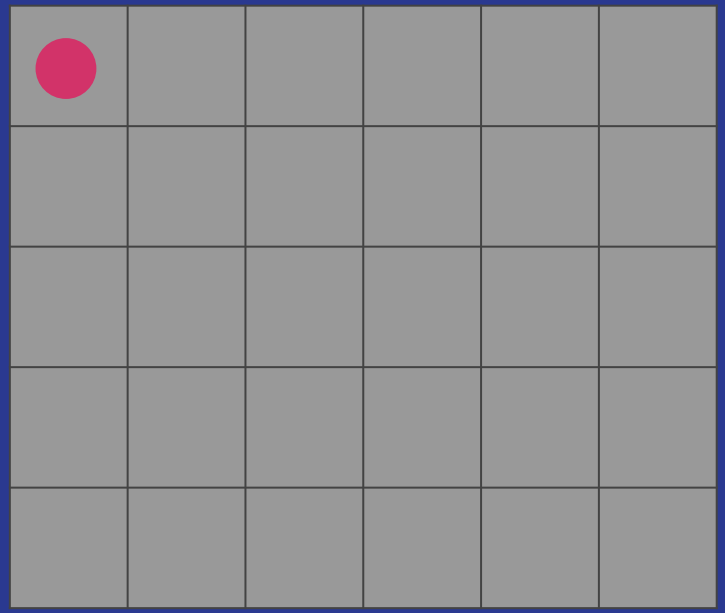


4 moves later

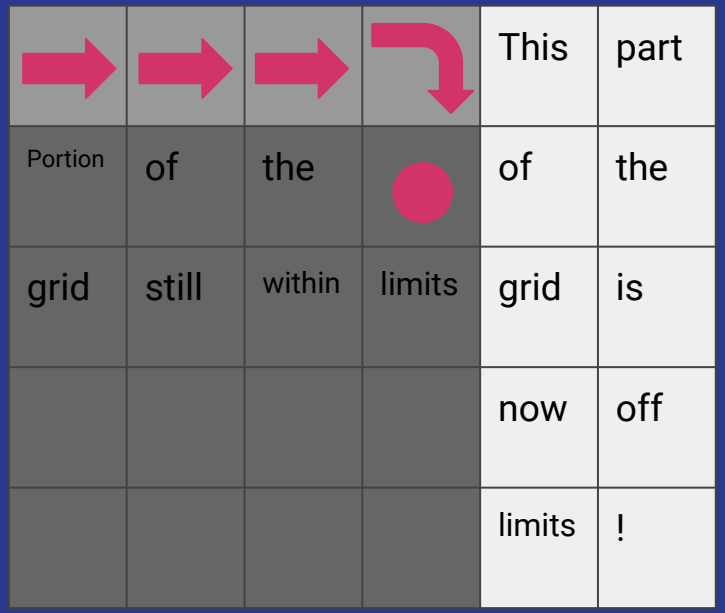


⭐⭐ Problem 6-2: Turnbot

- More generally, from the starting position, we move forward 0+ steps, then turn in one of the columns. Suppose we 1-index the columns from 1 to C.
- If we turn in the i-th column, the remainder grid is a (R-1) by i grid. If we rotate it to make our current position the top left, that grid is i by (R-1).



4 moves later



⭐⭐ Problem 6-2: Turnbot

- We can write a recurrence relation for numJourneys().
- Recursive Case:
  - numJourneys(R, C) = sum(i = 1 to C) of numJourneys(i, R-1)
- Base Case:
  - For any C >= 1, numJourneys(1, C) = 1
    - We have exactly one row, so cannot turn, so we must move to the end
- We can turn this relation directly to code.

⭐⭐ Problem 6-2: Turnbot

- We can write a recurrence relation for numJourneys().
- Recursive Case:
  - numJourneys(R, C) = sum(i = 1 to C) of numJourneys(i, R-1)
- Base Case:
  - For any C >= 1, numJourneys(1, C) = 1
    - We have exactly one row, so cannot turn, so we must move to the end
- We can turn this relation directly to code.

```
# Assume r, c >= 1
def numJourneys(r, c):
    if r == 1:
        return 1
    return sum([numJourneys(i, r-1) for i in range(1, c+1)])
```

⭐⭐ Problem 6-2: Turnbot

- Don’t forget to memoize!

```
# Assume r, c >= 1
def numJourneysMemoized(r, c, table):
    if r == 1:
        return 1
    if (r, c) in table:
        return table[(r, c)]
    journeys = sum([numJourneysMemoized(i, r-1, table) for i in
range(1, c+1)])
    table[(r, c)] = journeys
    return journeys

def numJourneys(r, c):
    return numJourneysMemoized(r, c, dict())
```

⭐⭐ Problem 6-2: Turnbot

- With memoization, we end up creating a O(RC) size table, so our complexity is at least that.
- For each entry, we end up doing O(C) work since we need to sum up the contributions of C terms.
- Final complexity: O(RC^2). Can we do better, perhaps by eliminating the sum?

⭐⭐ Problem 6-2: Turnbot

- Yes! Look back at our recurrence:
  - numJourneys(R, C) = sum(i = 1 to C) of numJourneys(i, R-1)
- The partial sum from (i = 1 to C-1) is just numJourneys(R, C-1).
- We can eliminate the sum by adjusting our recurrence:
  - numJourneys(R, C) = numJourneys(R, C-1) + numJourneys(C, R-1)
- Base Cases:
  - [From before] For any C >= 1, numJourneys(1, C) = 1
  - [New] For any R >= 1, numJourneys(R, 1) = numJourneys(1, R-1) = 1
    - This follows from the sum expression above, but also it's just a single column, so we have to turn immediately and move to the end
- This is now actually O(RC).

⭐⭐ Problem 6-2: Turnbot

```
# Assume r, c >= 1
def numJourneysMemoized(r, c, table):
    if r == 1 or c == 1:
        return 1
    if (r, c) in table:
        return table[(r, c)]
    journeys = numJourneysMemoized(r, c-1, table) +
        numJourneysMemoized(c, r-1, table)
    table[(r, c)] = journeys
    return journeys

def numJourneys(r, c):
    return numJourneysMemoized(r, c, dict())
```

Don't let Ian forget to say the password!!!!!!

Corner Sum: Solution

- Ian came up with this problem last night (it's probably been done before) and Andrew solved it way better!
- First of all, the negative numbers might look scary, but they don't really matter.
  - Why not? The final answer will use exactly four of the numbers. So even if we imagined transforming the grid to be non-negative by taking the most negative number and subtracting that off from every number of the grid, we wouldn't change which four cells the final answer uses.

Corner Sum: Solution

- **Brutest force:** try all possible sets of four cells in the grid, see which form rectangles, take the best sum from those. $O(n^8)$. 😱
- **Slightly better:** for each cell in the grid, try all pairs of (other cell in its row, other cell in its column). Once you choose a set of three cells like that, the location of the fourth cell is uniquely determined. $O(n^6)$.
- **Better still:** Actually, any *two* cells (with the first strictly above and to the left of the second) determine a rectangle. This cuts it down to $O(n^4)$.
- Can we do **Even Better**?

Corner Sum: Solution

- Andrew did!

- For each pair of rows, look at the sum of the 1st elements of both, then the sum of the 2nd elements of both, and so on. Keep track of the best such sum and the second best sum. This can be done in O(n) time per pair of rows, and the sum of those is the best we can do with this pair of rows.

- Since we do this for all $O(n^2)$ pairs of rows, this is $O(n^3)$ overall.

Corner Sum: Solution

```
def solve(grid):
    # find the minimum grid element. Not reeeally necessary but only O(n^2)
    smallest = min([min(r) for r in grid])
    answer = smallest * 4
    n = len(grid)
    for i in range(n):
        for j in range(i+1, n):
            best_in_row = smallest * 2
            second_best_in_row = smallest * 2
            for k in range(n):
                sm = grid[i][k] + grid[j][k]
                if sm >= best_in_row:
                    second_best_in_row = best_in_row
                    best_in_row = sm
                elif sm > second_best_in_row:
                    second_best_in_row = sm
            answer = max(answer, best_in_row + second_best_in_row)
    return answer
```

Corner Sum: Solution

- Can we do **Even Even Better?** Maybe? IDK
  - certainly we can't do better than $O(n^2)$ since we need to at least look at every cell of the grid. (What if we don't look at one, and the number is a quazillion and we should have used it)
  - If you think of something better than $O(n^3)$, let us know on Ed!

- Oh yeah, and there is one more obnoxious point…

One last subtlety

- Did you think about rectangles like this?
- (It can be worth asking: "do you mean grid-aligned rectangles?")

3	1	10	1
5	4	2	10
10	3	-10	8
9	10	9	3

- Checking these adds a smaller, 45-degree-tilted version of the original problem, so it doesn't change the big-O runtime.