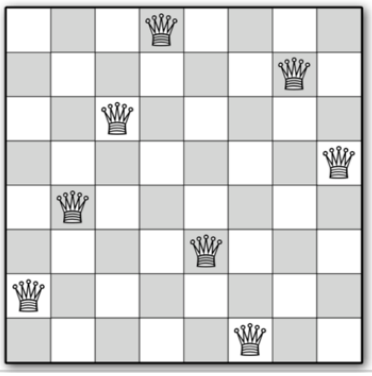
CS 4349.003.19F Lecture 7—September 11, 2019

Main topics for #lecture include #backtracking including #example/n_queens and #example/games_trees and #example/string_segmentation.

n Queens

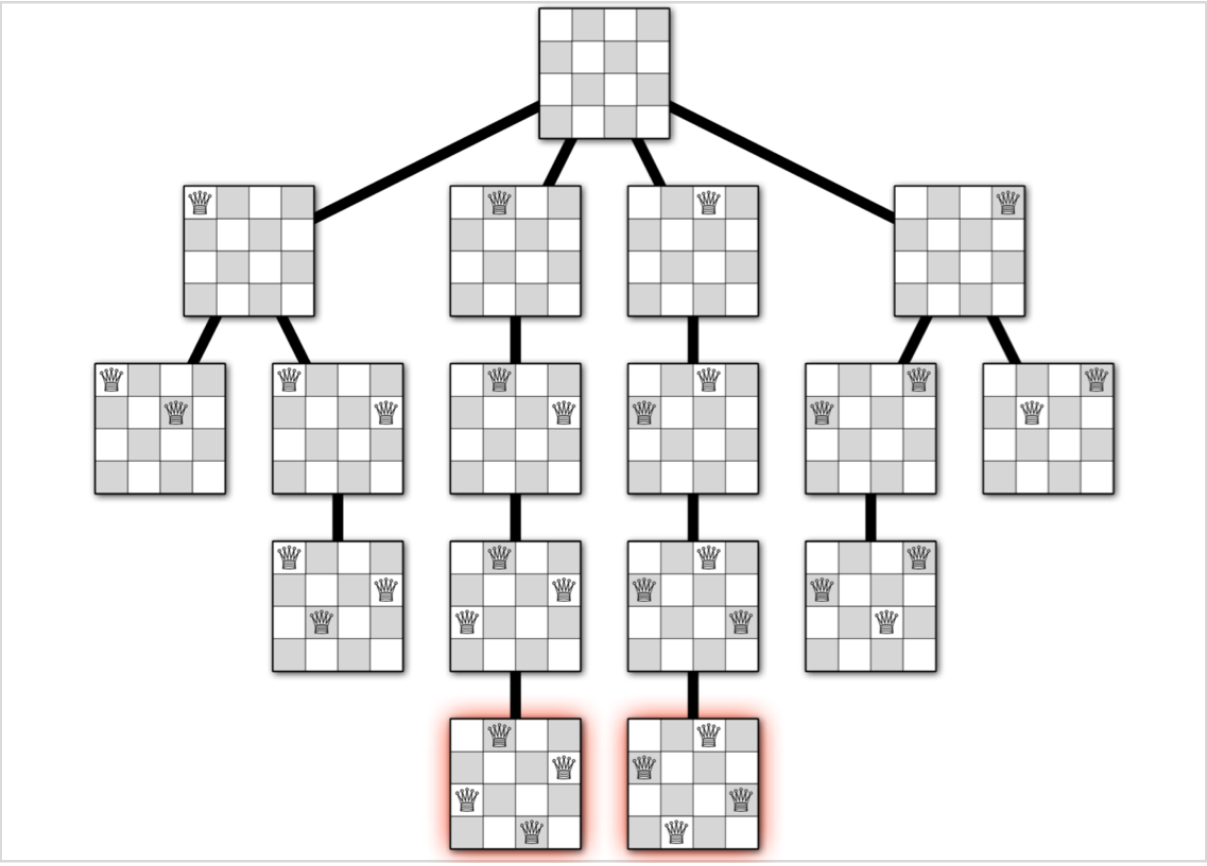
- Today, we're going to discuss another recursive algorithm paradigm called *backtracking*. When presented with several options, some of which may lead to a correct solution, the algorithm tries all options and deals with the consequences recursively.
- The prototypical example is the *n Queens Problem*, proposed by Max Bezzel in 1848 for n=8 and François-Joseph Eustache Lionnet in 1869 for general n.
- The problem is to place n queens on an n x n chessboard so no two queens can attack each other. This means no two queens are in the same row, column, or diagonal. Here is one solution:



- Emmanuel Laquière proposed the following backtracking method for solving the n x n case. We're going to guess possible queen placements and recursively see what we have to do as a consequence of our guesses.
- We'll represent possible solutions using an array Q[1..n] where Q[i] indicates which square in row i contains a queen. We'll place queens one row at a time starting at the top. To place the rth queen, we'll try all n squares in row r from left to right. If the square is already being attacked, we'll ignore it. Otherwise, we'll tentatively place the queen there and recursively look for consistent placements on the remaining rows r + 1 through n. If r > n, then we've placed all the queens and can print our solution.
- Here's pseudocode for this strategy. PlaceQueens(Q[1 .. n], r) prints every placement of queens such that, for $1 \le i \le r - 1$, there is a queen in column Q[i] of row i. To print all placements of queens on the whole board, we call PrintQueens(Q[1 .. n], 1).

```
PLACEQUEENS(Q[1..n], r):
  if r = n + 1
      print Q[1..n]
  else
      for j ← 1 to n
          legal ← TRUE
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← FALSE
          if legal
              Q[r] ← j
              PLACEQUEENS(Q[1..n], r + 1)        ⟨⟨Recursion!⟩⟩
```

- Outside the base case, the outer loop tries all placements in row r. The inner loop checks whether each placement is already being attacked.
- Like all recursive algorithms, you can visualize what's going on using a recursion tree. We'll draw the locations Q[1 .. r - 1] in each node. Leaf nodes mean there is no way to place queens in the next row without them being attacked.



- There are only two solutions for n = 4!

Backtracking

- We just saw an example of a *backtracking* algorithm.
- Backtracking algorithms are used to make a *sequence of decisions* with the goal of building a recursively defined structure satisfying certain conditions. Often, the goal

structure is itself a sequence.
    - For n queens, the goal is a sequence of queen positions, one for each row, such that no two queens attack each other. For each row, the algorithm decides where to place each queen.
- In each recursive call, we make *exactly one* decision, and it must be consistent with previous decisions. For that, the recursive call needs the portion of the data we haven't processed yet *and* a suitable summary of decisions already made. But for efficiency, we want the summary to be as simple as possible.
    - For n queens, we passed in the positions of previously placed queens. We had to remember every past decision to know whether or next placement is consistent with previous ones.
- The hard part with backtracking algorithms is figuring out *in advance* which information will be needed about past decisions *in the middle of* our algorithm. Sometimes this means solving a more general problem than the one you originally set out to solve. Kind of like how we needed to perform selection of the kth smallest element just to find the median element.
- But once you've figured out what to pass along, the algorithm design is relatively easy. Do *recursive brute force*. Try *all* possibilities of the next possible decision, and let the Recursion Fairy deal with the rest based on the information you provide them. No need to be clever. In fact, don't be.

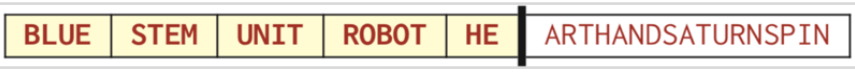
String Segmentation

- For another example problem, suppose you are given a string of letters in some foreign language, but without any spaces or punctuation. You need to break it up into individual constituent words.
- This is a problem that really does come up when trying to read classical Latin or Greek or even when trying to decipher modern languages and scripts like Burmese, Chinese, Japanese, or handwritten homework assignments. There are similar problems too, like trying to segment unpunctuated text into sentences.
- For the purposed of illustration, we'll stick to segmenting sequences of letters from the modern English alphabet into English words.
- For example, suppose we are given the input BOTHEARTHANDSATURNSPIN. It **can** be separated, and in two ways! "BOTH EARTH AND SATURN SPIN" and "BOT HEART HANDS AT URNS PIN". I never said the words had to make a coherent sentence.
- Some text can be segmented multiple ways, so we'll focus on the question, can the text be segmented into English words *at all*?
- I don't have the whole dictionary of English words memorized, so let's suppose we have access to a subroutine IsWord(w) that takes a string w as input and outputs True if w is a

word and False otherwise.

- So how should we go about solving this problem?
- The input is a sequence of letters, so we might try to consume the input letters in order from left to right. We should return TRUE if we can output a sequence of words, so should consider a process that produces the output words in order from left to right.
- So if we were in the middle of this process, our progress may look like this. We chose the words, blue, stem, unit, and robot. We have hearthandsaturnspin left to work with.

- At this point, we have to make a decision: which word appears next in our output sequence. We have four choices, he, here, heart, and hearth.
- We don't know which choices will lead to successfully segmenting the rest of the string! We could try to be smart with our choice, but one advantage of backtracking is that it encourages us to just try everything and let the Recursion Fairy deal with our mess.
- So we try he and recurse.



- We try hear, and recurse.

- etc.
- As long as the Recursion Fairy reports success at least once, we can report success. If the Recursion Fairy never reports success, then there is no first word we can safely choose, and we report failure.
- So what do we actually need to pass into a recursive call? For text segmentation, the earlier decisions have *no effect* on what choices we make for the suffix. Therefore, we can just ignore everything before the suffix.
- So the whole strategy is: select the first output word, and recursively segment the rest of the input string.
- We still need a base case for the recursive algorithm. The only time we can't recurse on a smaller string is length 0. Fortunately, the empty string has a segmentation into zero words!

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Index Formulation

- In practice, though, you probably don't want to pass around a whole array to recursive subproblems. In fact, for the purposes of designing an algorithm, it's often *better* to treat the input array as a global variable and pass around indices instead.
- Since we always need a suffix A[i .. n] of the input array, it suffices just to pass in a single index i and describe our recursive subproblem as "Given an index i, find a segmentation of the suffix A[i .. n]."
- To use the index, we need to define two boolean functions.
  - IsWord(i, j) = True if and only if A[i .. j] is a word (this is essentially given to us).
  - Splittable(i) = True if and only if the suffix A[i .. n] can be split into words.
- To describe our algorithm, we just need to describe how to implement Splittable(i). Our original problem is solved by returning the value of Splittable(1).
- The convenient thing about our new index based formulation is that we can describe our recursive strategy using a simple recurrence.

$$Splittable(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(IsWord(i,j) \wedge Splittable(j+1)\big) & \text{otherwise} \end{cases}$$

- Our algorithm is simply to evaluate Splittable(1) using this recurrence. And I would consider the recurrence plus initial call a full description of the algorithm for homework. In pseudocode, it looks almost exactly the same as the earlier algorithm.

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

- It may look like a trivial notation difference, but using index notation instead of array notation will speed up backtracking algorithms in practice, and it will be useful for speeding this algorithm up using a process called *dynamic programming* which we'll discuss on Monday.

Subset Sum

- In the unlikely event we have extra time, we'll consider another example problem called SubsetSum.
- We're given a set X of *positive integers* and a *target* integer T. Is there a subset of X that

adds up to T?

- For example if X = {2, 5, 8} and T = 10, the answer is TRUE, because 2 + 8 = 10. If X = {2, 5, 8} and T = 11, the answer is FALSE.
- There are two easy cases:
  - If T = 0, then the answer is TRUE, because the empty subset of numbers sum to 0.
  - If T < 0 or T ≠0 and X is empty, the answer is FALSE. We have positive integers, so they can't sum to a negative number. Also, we can't create a positive number if we have no integers.
- So what about the general case? Let x be an arbitrary value in the non-empty set X. If there is a subset summing to T, then that subset either includes x or it doesn't.
  - If it does include x, then the remaining members of the subset come from X \ {x} and they sum to T - x.
  - If it doesn't include x, then all of the subset members come from X \ {x} and they sum to T.
- Either way, we can reduce to the SubsetSum problem on a smaller subset X \ {x}.

```
⟨⟨Does any subset of X sum to T?⟩⟩
SUBSETSUM(X, T):
  if T = 0
      return TRUE
  else if T < 0 or X = ∅
      return FALSE
  else
      x ← any element of X
      with ← SUBSETSUM(X \ {x}, T − x)    ⟨⟨Recurse!⟩⟩
      wout ← SUBSETSUM(X \ {x}, T)        ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

- In summary, we already discussed why the base cases are correct. If we aren't in a base case, then if there is a subset summing to T, it either contains x or it doesn't. If so, we'll the Recursion Fairy will find the rest of the subset during the first recursive call. Otherwise, they'll find the whole subset during the second call.
- Like text segmentation, and unlike n Queens, we don't need to remember all our past decisions, so we don't pass them along to recursive calls. All we need to remember is what subset of elements are remaining and a target value.