Lecture 3: Backtracking

Last updated Sept 17, 2024

References:

- Algorithms, Jeff Erickson, Chapter 2

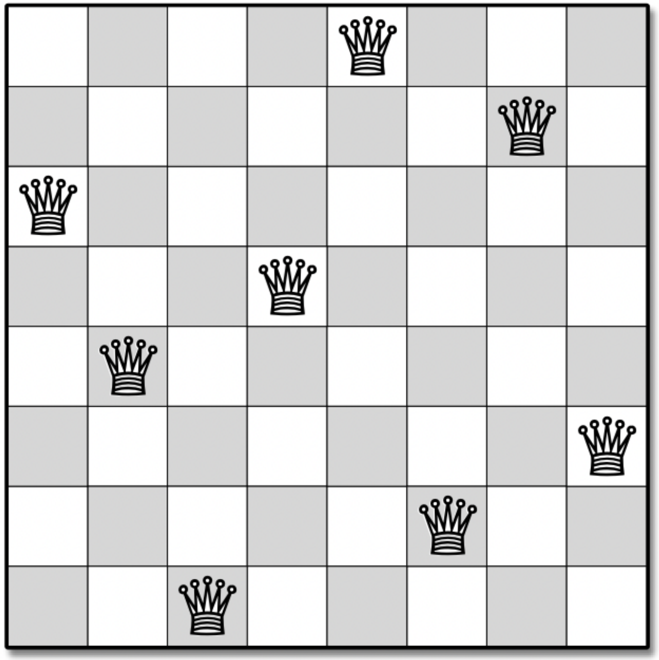
Backtracking

Backtracking is a particular recursive strategy which constructs a slution **incrementally**, one piece at a time.

Think the solution as a vector, and backtracking computes one element at a time.

Whenever the algorithm needs to decide between multiple alternatives, it systematically evaluates **every** alternative and choose the best.

N-Queens



On a 8x8 chessboard, place 8 queens such that no two queens attack each other. (no two queens in the same row, column, or diagonal).

Find one solution, or all the solutions, or count the number of solutions.

Backtracking strategy:

For top row to bottom row, try to place one queen per row at every possible position.

Key data structure: partial solution.

N-Queen Backtracking

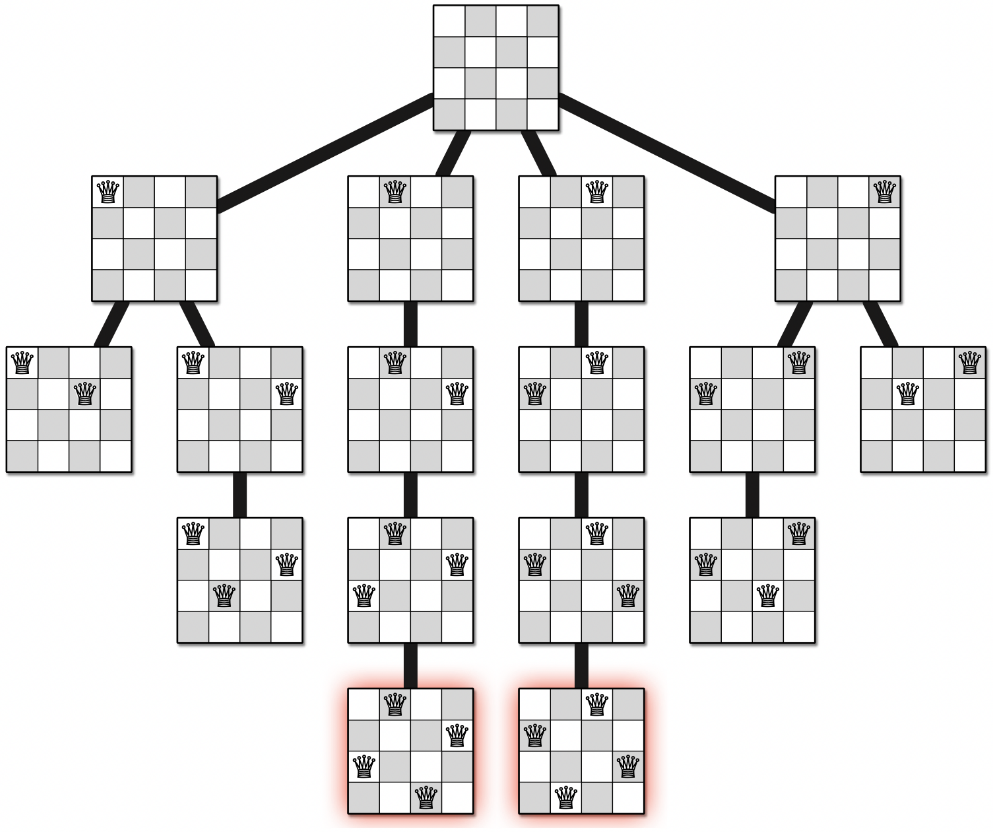
The backtracking algorithm given by Gauss is essentially:

```
def PlaceQueens(Q, r, n):
    if r == n:
        print(Q)  # solution complete, print the board configuration
    else:
        for j in range(n):  # consider position j in row r
            legal = True
            for i in range(r):  # checking if j is legal
                if Q[i] == j or Q[i] == j + r - i or Q[i] == j - r + i:
                    legal = False
                    break
            if legal:
                Q.append(j)  # make move by appending
                PlaceQueens(Q, r + 1, n)  # accept j, recurse!
                Q.pop()  # unmake move by popping the last element


def solve_n_queens(n):
    Q = []  # Initialize an empty list to store queen positions
    PlaceQueens(Q, 0, n)  # Start placing queens from row 0

# Example to solve the 8-queens problem
```

```
solve_n_queens(8)
```

N-Queen Backtracking

It's equivalent to depth-first search of this tree:



Recursion tree:

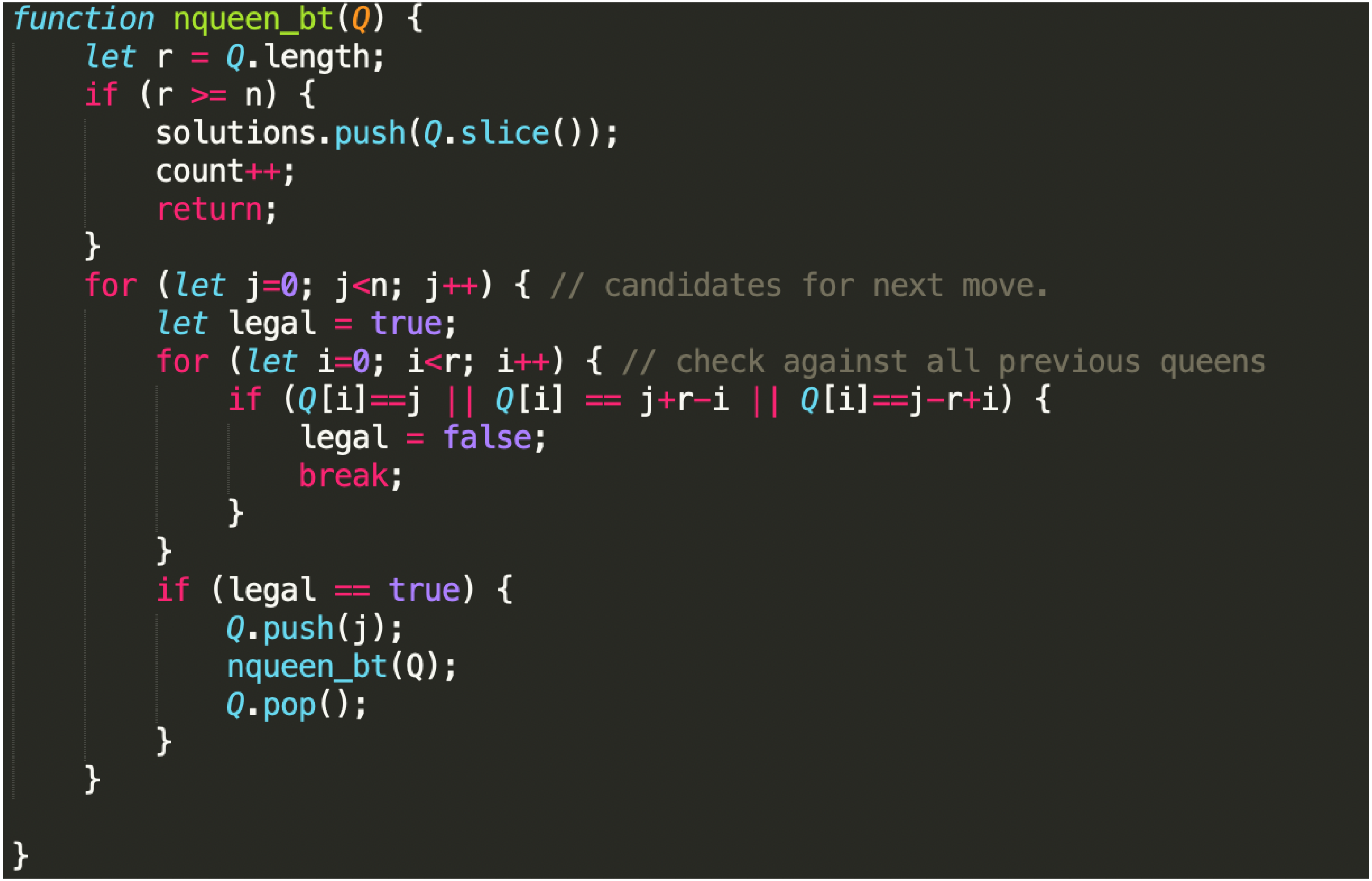
Node: legal partial solution.

Edge: recursive calls

Leaves: partial solutions that cannot be extended (deadend, or complete solution)

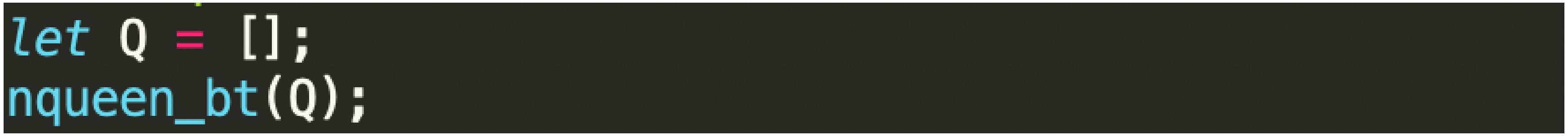
N-Queens Implementation

See the animation page: `http://www2.cs.uh.edu/~panruowu/2021s_cosc3320/nqueen.html`



```javascript
function nqueen_bt(Q) {
    let r = Q.length;
    if (r >= n) {
        solutions.push(Q.slice());
        count++;
        return;
    }
    for (let j=0; j<n; j++) { // candidates for next move.
        let legal = true;
        for (let i=0; i<r; i++) { // check against all previous queens
            if (Q[i]==j || Q[i] == j+r-i || Q[i]==j-r+i) {
                legal = false;
                break;
            }
        }
        if (legal == true) {
            Q.push(j);
            nqueen_bt(Q);
            Q.pop();
        }
    }

}
```

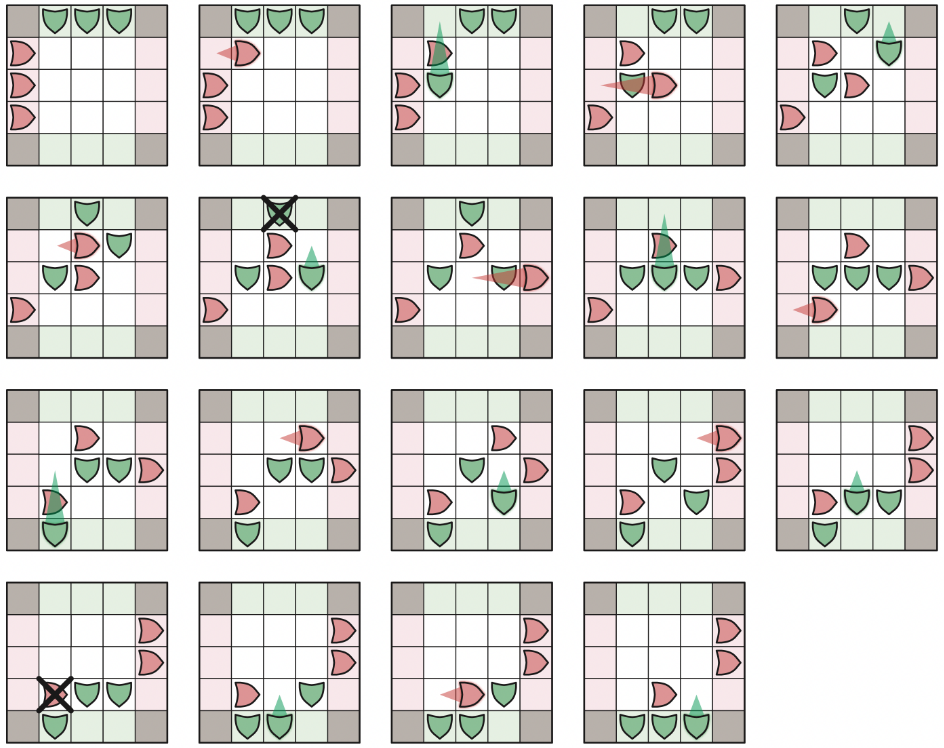
To call the recursive backtracking routine, simply do:



```
let Q = [];
nqueen_bt(Q);
```

Game

Consider a simple two-player game on nxn square grid:



Rules:

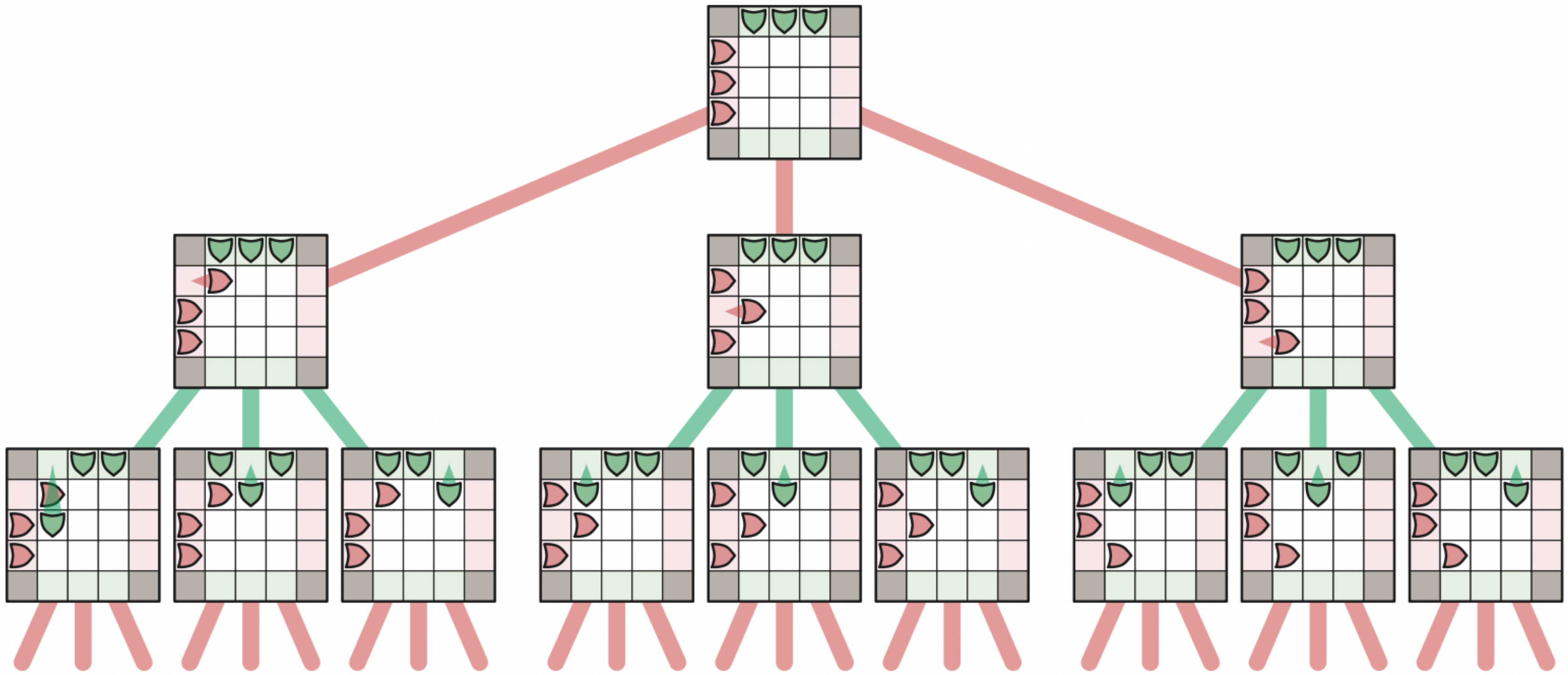
Red/Green player takes turns

Can move only horizontally (red) or vertically (green)

Can move one step, or jump over one.

Place all tokens to the other ends to win.

We can devise algorithm that starts in any game state, it can win the game if it's possible to win another perfect player.



Game Tree

We recursively define a game state to be **good** or **bad** as:

- A game state is **good**, if either the current player already won, or if the currently player can move to a **bad** state of the opposing player.
- A game state is **bad**, if either the current player already lost, or if every available move leads to **good** state of opposing player.

To put it in another way, a non-leaf node in the game tree is good, if one of its children is bad. A non-leaf node is bad, if all its children are good.

If we are in good state, we win even if opposing player is perfect.

If we are in bad state, we lose, unless opposing player makes mistake.

Game Tree: Telling Good from Bad

The recursive definition of **good** and **bad** state automatically suggests a recursive backtracking algorithm to play perfectly:

```
PlayAnyGame($X$, $player$):
    if $player$ won in state $X$
        return good
    if $player$ has lost in state $X$
        return bad
    for all legal moves $X \longrightarrow Y$
        if PlayAnyGame($Y$, $\neg player$) $=$ bad
        return good
    return bad
```

What are the cost?

Subset Sum

Problem: given a set X of positive integers, and a target integer T, is there a subset of X that add up to T?

E.g. if X={8,6,7,5,3,10,9}, and T=15, the answer is yes, because subsets {8,7}, and {7,5,3}, and {6,9}, and {5,10} all add up to T=15.

How to approach this problem? Take any element $x \in X$, the answer is yes iff one of the following is true:

- There is a subset of X that *includes* x that add up to T
- There is a subset of X that *excludes* x that add up to T

Taking each case, we have a recursive algorithm:

```
SubsetSum(X,T):                    #does any subset of X sum to T?
    if T=0 return **true**
    else if T<0 or X=$\emptyset$ return **false**
    else
        x$\leftarrow$any element of X
        with $\leftarrow$ SubsetSum(X$\setminus$\{x\},T-x)        #recurse, case 1
        without $\leftarrow$ SubsetSum(X$\setminus$\{x\},T)       #recurse, case 2
        return with $\wedge$ without
```

The correctness of this algorithm can be proved by induction.

What's the cost?

Backtracking Summary

General patterns: making a sequence of decisions.

- in the N-Queens problem, the goal is a sequence of queen positions, one in each row. For each row, algorithm decides where to place the queen.
- In the game tree problem, the goal is a sequence of legal moves, such that each move is as good as possible for the player. For each game state, the algorithm decides best next move.
- In the SubsetSum problem, the goal is a sequence of input elements that have a particular sum. For each input element, the algorithm decides whether to include it in subset or not. (why is the goal finding a sequence of subset?)

In fact, we can write a generic backtracking algorithm that can serve as template for solving all kinds of problems.

```
def backtrack(a, k, input_data): # a is the partial solution at step k
    if is_solution(a, k, input_data):
        process_solution(a, k, input_data)
    else:
        k += 1
        candidates = construct_candidates(a, k, input_data)
        for c in candidates:  # For each candidate
            a[k] = c  # Make move
            make_move(a, k, input_data)
            # Recursive call for next level
            backtrack(a, k, input_data)
            if finished:  # Early exit if solution found, etc
                return
            unmake_move(a, k, input_data)
```

Backtracking: Key Techniques

Successful design of backtracking often involves designing a partial solution structure that encodes all necessary past decisions. Why? Because current decision depends on past decisions.

- In N-Queens, we must pass all past decisions (positions in all previous rows) in order to make legal moves.
- In game tree problem, we only need the current game state.
- For the SubsetSum problem, we need to pass all the remaining integers, and the remaining target value. Here we don't need the complete history of past decisions (unless we are asked to output a subset or all subsets that add up to T).

We **must** design in advance what information is needed about past decisions. We might need to solve a more general problem.

Backtracking Technique

Also, it's often easier to answer a yes/no question first, and design backtracking algorithm for that.

Modifying the algorithm to obtain more information, or variants of the problems (e.g. how many solutions? All solutions? One solution? The best solution according to some criteria?) can be rather easily done.

Therefore we usually consider only the yes/no variant of a backtracking problem.

Text Segmentation

Problem: given a text string (without punctuation or spaces), segment it into words. For example consider the English string:

`BOTHEARTHANDSATURNSPIN`

which can be segmented as "`BOTH EARTH AND SATURN SPIN`", or "`BOT HEART HANDS AT URN SPIN`".

As usual, we start with a yes/no question: given a string, can it be segmented into English words at all?

To make it concrete, let's say we are given a routine that can tell us whether a string is a "word" or not: IsWord(w).

Text Segmentation

The input are strings of letters, and the output are sequence of words. It's natural to consume and produce from left to right.

Jumping into the middle of the segmentation process, we might have something like:

`BLUE`	`STEM`	`UNIT`	`ROBOT`	`HEARTHANDSATURNSPIN`

Here the yellow bars are the past decisions—spliting 17 letters into 4 words. Now it's time to decide: *where do the next work end*?

There are 4 tentative possibilities:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN
BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN
BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN
BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

In the first case, we *tentatively* accepts `HE` as our next word, and recurse into the remaining text. If it returns **true**, we are done; if it returns **false**, we try the next case, and so on.

In this particular problem, our past decisions are not needed for making the current decision at all.

Text Segmentation

```
# returns true if the string A can be splitted into words in word_dict
def splittable(A, word_dict):
    # Base case: if the string is empty, it is considered splittable
    if len(A) == 0:
        return True

    # Iterate over the string and try to split it
    for i in range(1, len(A) + 1):
        # Check if the prefix A[0:i] is a valid word
        if A[:i] in word_dict:
            # If the prefix is a word, recursively
            # check the rest of the string
            if splittable(A[i:], word_dict):
                return True

    # If no valid split found, return False
    return False
```

```
# Example usage
word_dict = {"can", "the", "string", "be", "segmented"}  # Example
dictionary of valid words
A = "canthestringbe"
print(splittable(A, word_dict))
```

Note that the partial solution does not need to be maintained.

Longest Increasing Subsequence

For any sequence S, a **subsequence** of S is one that's obtained by deleting zero or more elements, without changing the order. The elements in the subsequence need not be contiguous in S.

In contrast, a **substring** of S is a contigous subsequence of S.

For example, `MASHER` and `LAUGHTER` are subsequences of `MANSLAUGHTER`, but only `LAUGHTER` is also a substring.

Now the LIS problem is: given a sequence of integers, find the longest subsequence whose elements are in increasing order.

LIS

Again, we are dealing with sequence, might as well start from left to right, and decide to include A[j] or not in our subsequence.

Here's a possible middle of decision sequence:

3 1 4 1 5 9 2 6 5 3 **5**$^?$ 8 9 7 9 3 2 3 8 4 6 2 6

We are deciding whether to include the number 5. But we cannot because it make our current subsequence not increasing. Next one.

3 1 4 1 5 9 2 6 5 3 5 **8**$^?$ 9 7 9 3 2 3 8 4 6 2 6

Now we have two choices to explore:

1. We include 8, and recurse into the rest
2. We exclude 8, and recurse into the rest

LIS

Now the important question: what information is needed from our past decisions? It appears that we only need one last included number to make current decision (we have the increasing order constraint).

Our base case is that when we at the end (j=n), we return 0, which is the length of the LIS (there is only one).

```python
# returns the size of longest increasing subsequence in A
# in which all elements are larger than prev
def LIS(prev, A):
    # Base case: if the array is empty, the LIS length is 0
    if len(A) == 0:
        return 0

    # If the current element cannot be part of the subsequence
    if A[0] <= prev:
        return LIS(prev, A[1:])  # Skip A[0] and recurse on the rest

    # Option 1: Skip the current element and continue
    skip = LIS(prev, A[1:])
    # Option 2: Include the current element and continue with it as the new 'prev'
    take = LIS(A[0], A[1:]) + 1

    # Return the maximum of skipping or taking the current element
    return max(skip, take)

# Example usage
A = [3, 10, 2, 1, 20]  # Sample input array
print(LIS(float('-inf'), A))  # Start with the smallest possible 'prev'
```

LIS Backtracking

Proof of correctness of the backtracking LIS algorithm:

Sufficient to provide the recurrence: let $L(i,j)$ denote the length of the LIS of A[j..n], with every element bigger than A[i].

Our recursion gives the recurrence:

$$L(i,j) = \begin{cases} 0 & \text{if } j > n \\ L(i,j+1) & \text{if } A[i] > A[j] \\ \max\left\{ \begin{smallmatrix} L(i,j+1) \\ 1 + L(j,j+1) \end{smallmatrix} \right. & \text{otherwise} \end{cases}$$

To show the correctness of this recurrence, it's important to note what $L(i,j)$ means, and based on that, argue the recurrence makes sense.

What's the cost of this recursive algorithm?

$$T(n) = 2T(n-1) + 1 \quad \Rightarrow \quad T(n) = 2^n - 1$$

Worked Example: Sudoku

Rules:

1. Each of the digits 1-9 must occur exactly once in each row;

2. Each of the digits 1-9 must occur exactly once in each column;

3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes (with dark borders)

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

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

Now let's try to solve this. We can represent the 9x9 grid by a 2d array, with elements being "123456789" or 0 if it's not marked.

We are given a 2d array which represents a Sudoku puzzle. Our job is to find one solution. Backtracking lends itself nicely to this task.

```
Sudoku(A[1..9][1..9]):
    if grid is full:
        output solution and return
    find next open position i,j                    #next position?
    for num in 1..9 that can fill A[i][j]          #next fill-in?
        A[i][j] = num                              #attempt to fill in
        Sudoku(A[1..9][1..9])                      #recurse!
        A[i][j] = 0                                #dead end; backtrack
```

Note that this backtracking algorithm will systematically try **all** possible ways fill-in the grid until a solution is found.

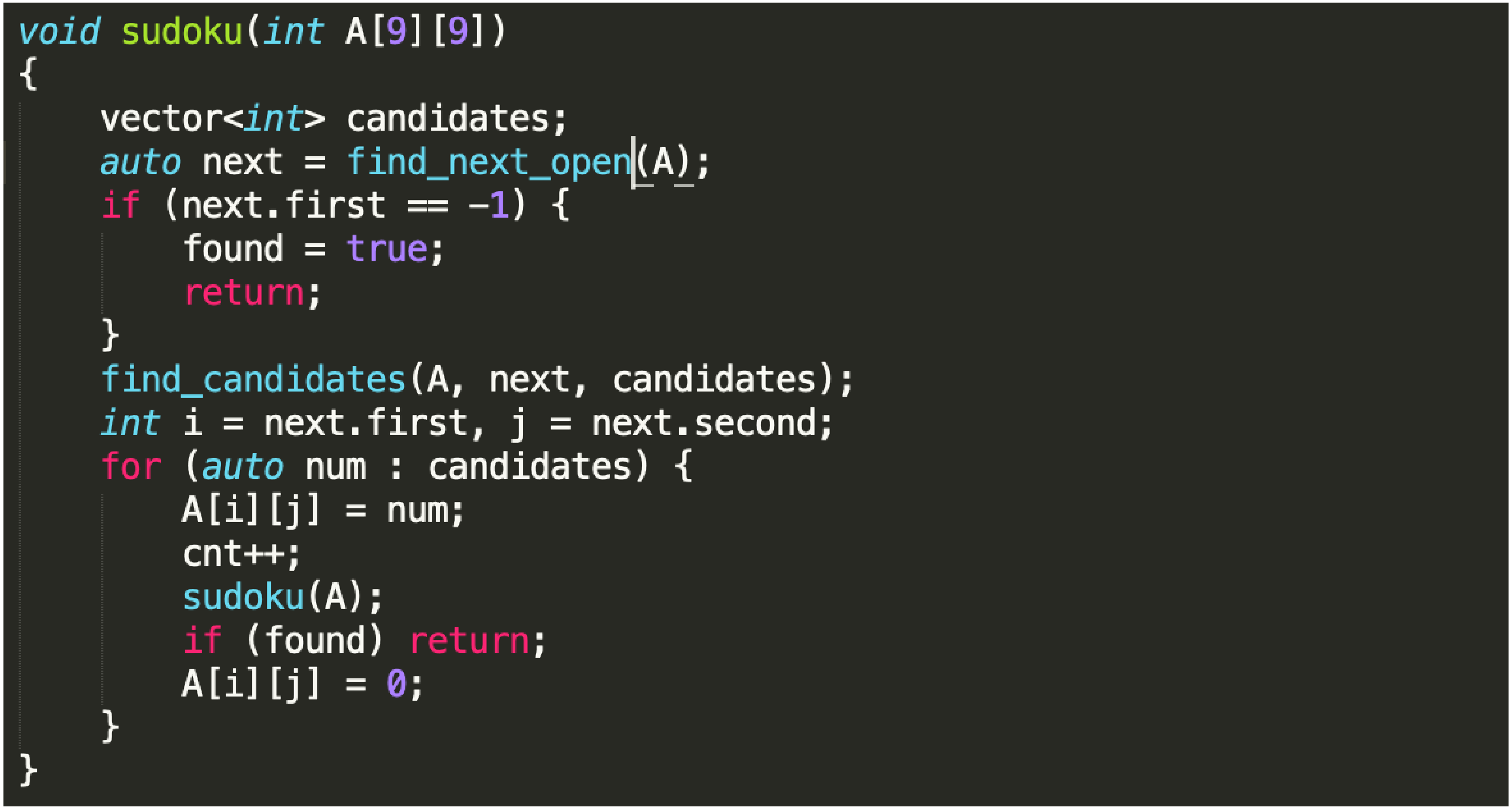
Constructing the candidates for the next solution position involves first picking an open square we want to fill, and then indentifying which number are candidates to fill the square. Two reasonable way to pick the first open square:

- Arbitrary Square Selection: pick the first one, the last one, or a random one; All are correct and it's not clear which one is better.
- Most constrained Square Selection: check each of the open squares, and pick the one that has fewest candidate numbers.

Both work correctly, but speed might be different. It's intuitive that the second heuristic may lead to solution faster. Why? Because if our next position has only two candidate numbers, we have 1/2 possibility of guessing right. Whereas if we pick open position that has 9 candidate numbers, we have 1/9 possibility to guess right.

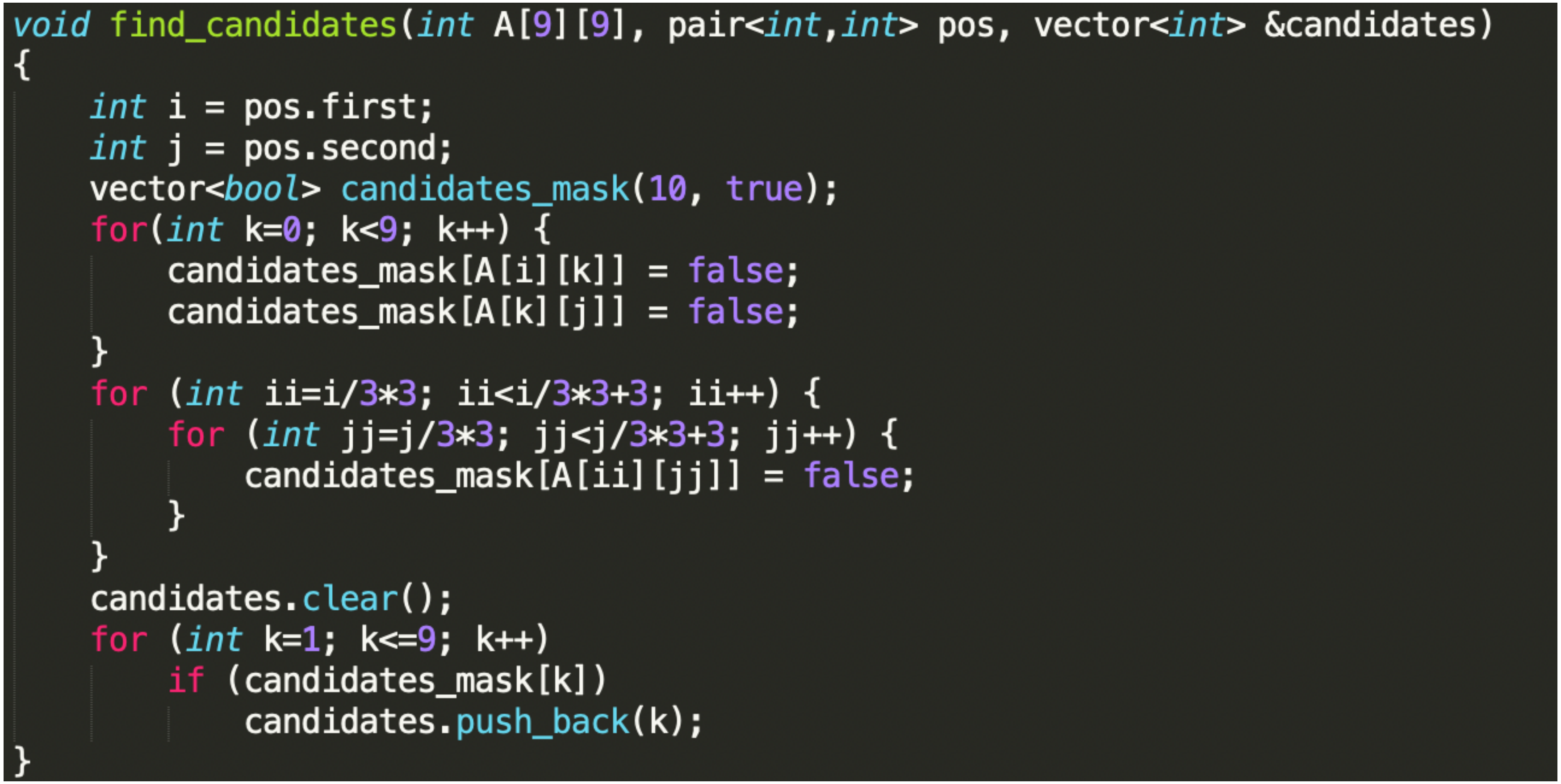
Let's put it into C++ code. Some test puzzle files can be downloaded from http://lipas.uwasa.fi/~timan/sudoku/

First we have the `sudoku` function



```
void sudoku(int A[9][9])
{
    vector<int> candidates;
    auto next = find_next_open(A);
    if (next.first == -1) {
        found = true;
        return;
    }
    find_candidates(A, next, candidates);
    int i = next.first, j = next.second;
    for (auto num : candidates) {
        A[i][j] = num;
        cnt++;
        sudoku(A);
        if (found) return;
        A[i][j] = 0;
    }
}
```

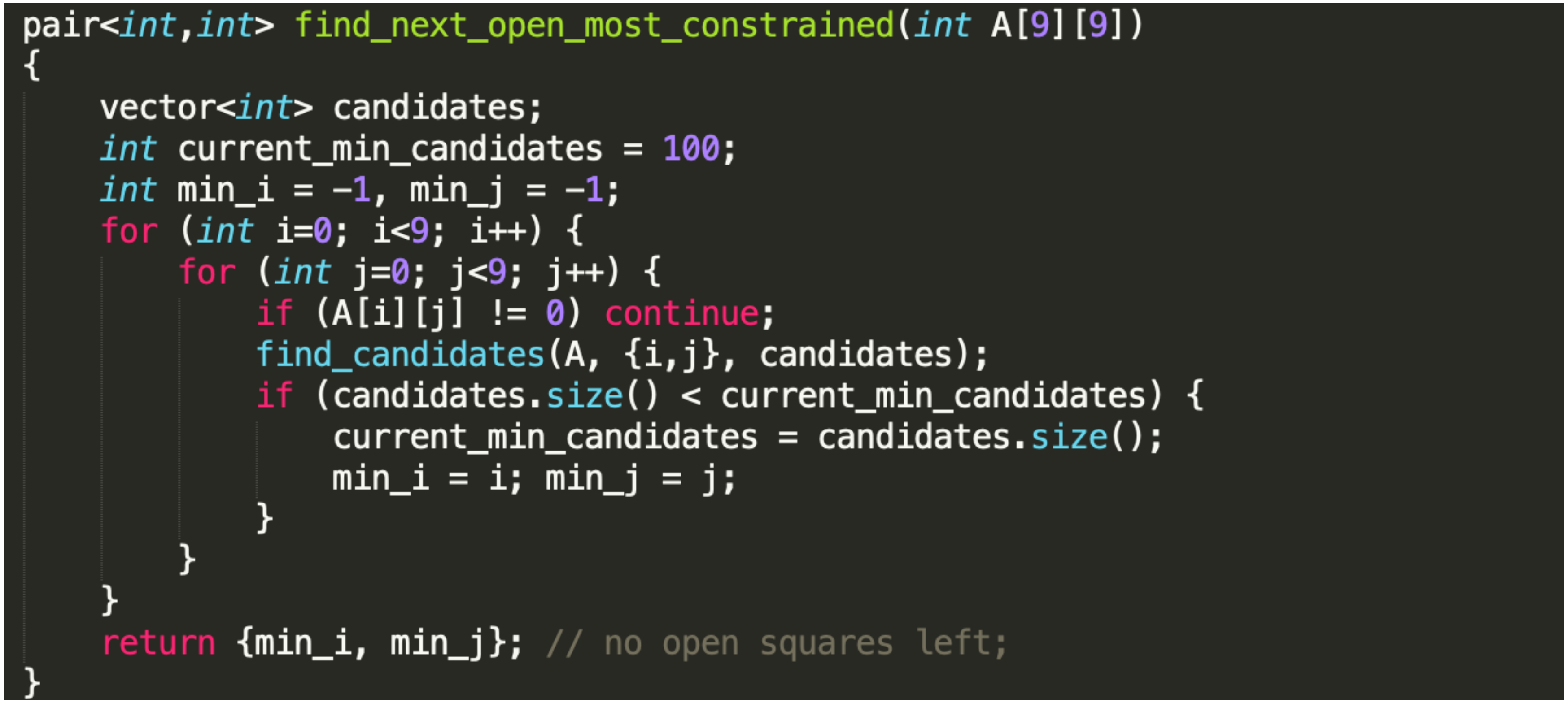
next let's see the `find_candidates()` which gives a list of legal fill-ins for a given open square:



```
void find_candidates(int A[9][9], pair<int,int> pos, vector<int> &candidates)
{
    int i = pos.first;
    int j = pos.second;
    vector<bool> candidates_mask(10, true);
    for(int k=0; k<9; k++) {
        candidates_mask[A[i][k]] = false;
        candidates_mask[A[k][j]] = false;
    }
    for (int ii=i/3*3; ii<i/3*3+3; ii++) {
        for (int jj=j/3*3; jj<j/3*3+3; jj++) {
            candidates_mask[A[ii][jj]] = false;
        }
    }
    candidates.clear();
    for (int k=1; k<=9; k++)
        if (candidates_mask[k])
            candidates.push_back(k);
}
```

And for the `find_next_open()` function that chooses next open square to fill-in, we implement three heuristics: find the first open (left-right, top-bottom order), find a random open square, and find the most constrained open square.

Here we only show the `find_next_open_most_constrained()`



```cpp
pair<int,int> find_next_open_most_constrained(int A[9][9])
{
    vector<int> candidates;
    int current_min_candidates = 100;
    int min_i = -1, min_j = -1;
    for (int i=0; i<9; i++) {
        for (int j=0; j<9; j++) {
            if (A[i][j] != 0) continue;
            find_candidates(A, {i,j}, candidates);
            if (candidates.size() < current_min_candidates) {
                current_min_candidates = candidates.size();
                min_i = i; min_j = j;
            }
        }
    }
    return {min_i, min_j}; // no open squares left;
}
```

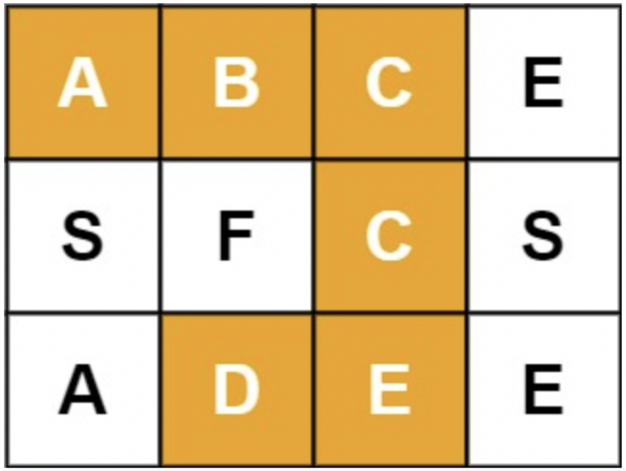
Some runtime performance for the very hard case in previous picture.

find_next_open	runtime(ms)	#recursion calls
first open	1,079	6,943,195
random	?	?
most constrained	68	10,373

It seems that the most-constrained heuristic is very helpful in handling the hard case.

Word Search

(Leetcode 79). Given an $m \times n$ `board` and a `word`, find if the word exists in the grid. The word can be reconstructed from letters sequentially adjacent cells, where “adjacent” means horizontally or vertically neighboring. The same letter cannot be used more than once.



```
Input: board = [["A","B","C","E"],["S","F","C","S"],
["A","D","E","E"]], word = "ABCCED"
Output: true
```

Word Ladder

(leetcode 126)

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

Note:

- Return an empty list if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.