COMP 251

Algorithms & Data Structures (Winter 2022)

Algorithm Paradigms – Complete Search

School of Computer Science
McGill University

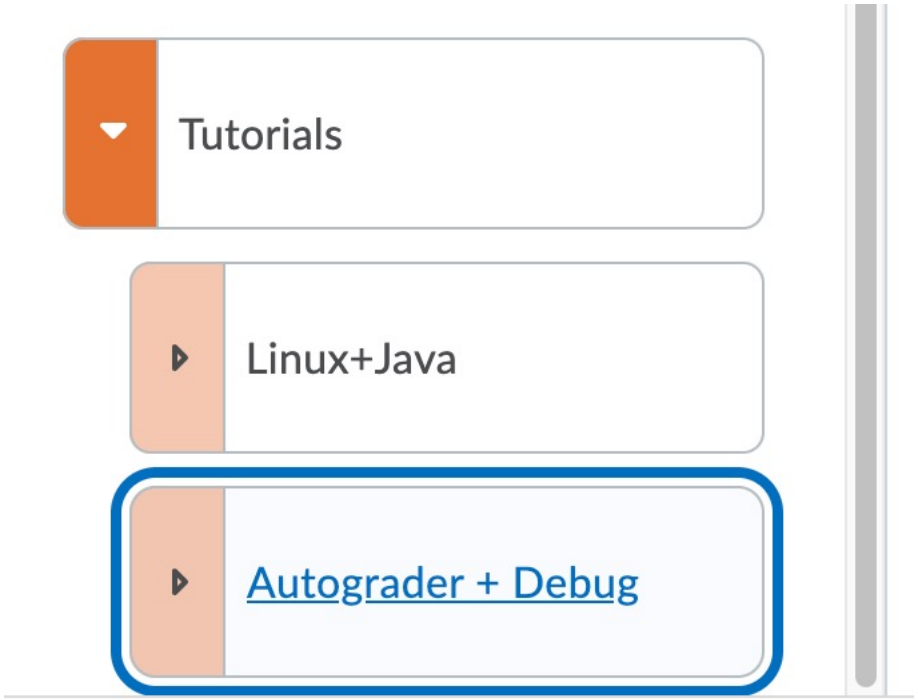
Slides of (Comp321 ,2021), Slides of (Scottm-cs314), Erickson (Algorithms-ebook)

Announcements

OFFICE HOURS (Jan 31-Feb 4)

	Monday	Tuesday	Wednesday	Thursday	Friday
9am - 10am					
10am - 11am		Shishir-OH	Rui-OH	Jennifer-OH	T.A meeting
11am - 12pm	David-OH	Shishir-OH	Yaxuan Li-OH	Yaxuan Li (extra)	Zedian Xiao-OH
12pm - 1pm					
1pm - 2pm		Itai-OH		Yaxuan Li (extra)	Ade-OH
2pm - 3pm	TUTORIAL	Lecture	Yaxuan Li (extra)	Lecture	James-OH
3pm - 4pm	James-OH	Lecture	David-OH	Lecture	
4pm - 5pm		Shael-OH		Alvin-OH	Parham-OH
5pm - 6pm	Gabriel-OH				Rui (extra)
6pm - 7pm	Rui (extra)				Rui (extra)
7pm - 8pm					
8pm - 9pm					
9 pm - 10pm					

Announcements



Algorithmic Paradigms

- General approaches to the construction of *correct* and *efficient* solutions to problems.
- Such methods are of interest because:
  - They provide templates suited to solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most high-level languages.
  - The temporal and spatial requirements of the algorithms can be precisely analyzed.
- Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

Algorithmic Paradigms

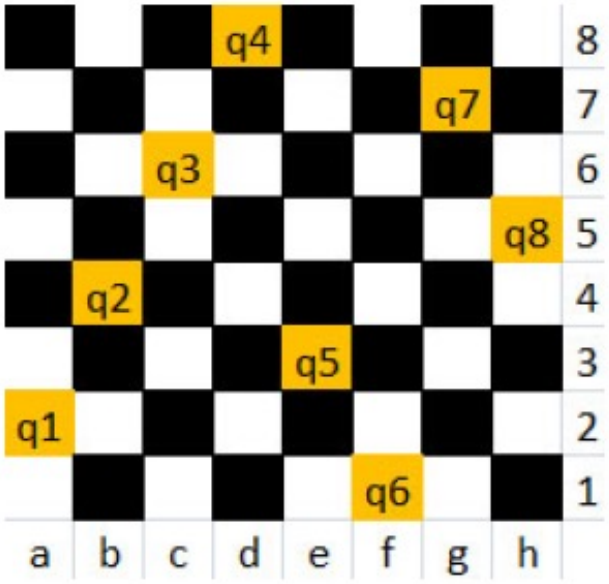
- Complete Search.
- Divide and Conquer.
- Dynamic Programming.
- Greedy

Complete search

- Also know (related) as recursive backtracking or brute force.
  - Backtracking is a sort of refined brute force
- It is a method for solving a problem by searching (up to) the entire search space to obtain the required solution.
  - The search space can be large but finite.
  - Does not exist an algorithm that uses a method other than exhaustive search.
    - Need for developing techniques of searching, with the hope of cutting down the search space to possibly a much smaller space. Backtracking can be described as an organized exhaustive search which often avoids searching all possibilities.

Recursive Backtracking – example

- Problem: In chess (with a standard 8x8 board), it is possible to place eight queens on the board so that no queen can be taken by any other. Write a program that will determine all such possible arrangements.



Recursive Backtracking – example

- Solution 1: slow
  - Use a solution vector where a_i is true iff there is a queen on the i^{th} square.
  - There are $2^{64} \approx 1.84 \times 10^{19}$ different true/false vectors for 8X8 board. huge

then count the solutions

- Solution 2: slow but better
  - Have the i^{th} element of a solution vector explicitly list the square where the i^{th} queen resides. a_i will be an integer from 1 to n^2.
  - There are $64^8 \approx 2.81 \times 10^{14}$ vectors.

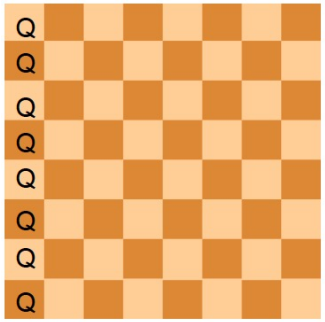
then count solutions

Recursive Backtracking – example

- Solution 3: much better - still slow
  - Prune solution 2 by removing symmetries. Ensure that the queen in a_i sits on a higher number square that the queen in a_{i-1}
  - How many ways can you choose k things from a set of n items?
    - In this case there are 64 squares and we want to choose 8 of them to put queens on.

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1} = \frac{n!}{k!(n-k)!} \quad \text{if } 0 \leq k \leq n$$

    - This change will reduce the search space to $\binom{64}{8} = 4.426 \times 10^9$
  - Includes lots of set ups with multiple queens in the same column.



Recursive Backtracking – example

- Solution 4:
  - Note that there must be exactly one queen per column for a n-queens solution. Then you can limit the candidates of the i^{th} queen to the eight squares on the ith column.
  - There are $8^8 \approx 1.67 \times 10^7$ vectors for 8X8 board.

- Solution 5:
  - Since no two queens can share the row/column, we know that the n columns of a complete solution must form a permutation of n.
  - We reduce then our search space to just 8! = 40320 vectors.

Recursive Backtracking – example

- Solution 6:
  - You can improve solution 5 by knowing that no two queens can share any of the two diagonals.

- Solution 7:
  - Identify the 12 unique (or fundamental) solutions. You can utilize this fact by only generating the 12 unique solutions and, if needed, generate the whole 92 by rotating and reflecting these 12 unique solutions

Brute force = useful
need to find the best searchspace to brute force

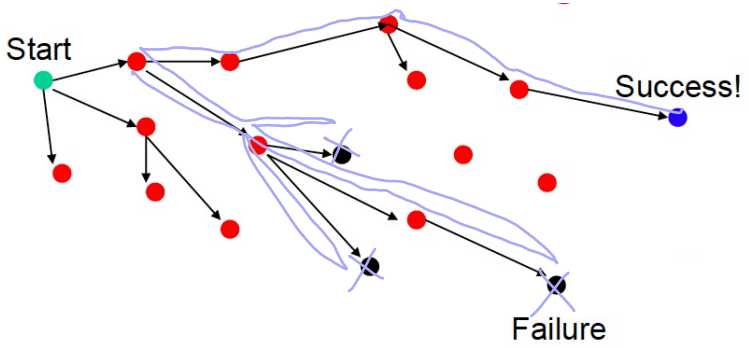
Recursive Backtracking – Solution 4

- Solution 4:
  - There are $8^8 \approx 1.67 \times 10^7$ vectors for 8X8 board.
  - If number of queens is fixed and I realize there can't be more than one queen per column I can iterate through the rows for each column.

```
for(int r0 = 0; r0 < 8; r0++){
      board[r0][0] = 'q';
       for(int r1 = 0; r1 < 8; r1++){
            board[r1][1] = 'q';
            for(int r2 = 0; r2 < 8; r2++){
                  board[r2][2] = 'q';
                  // a little later
                  for(int r7 = 0; r7 < 8; r7++){
                        board[r7][7] = 'q';
                        if( queensAreSafe(board) )
                              printSolution(board);
                        board[r7][7] = ' '; //pick up queen
                  }
                  board[r6][6] = ' '; // pick up queen
```

Recursive Backtracking – Solution 4

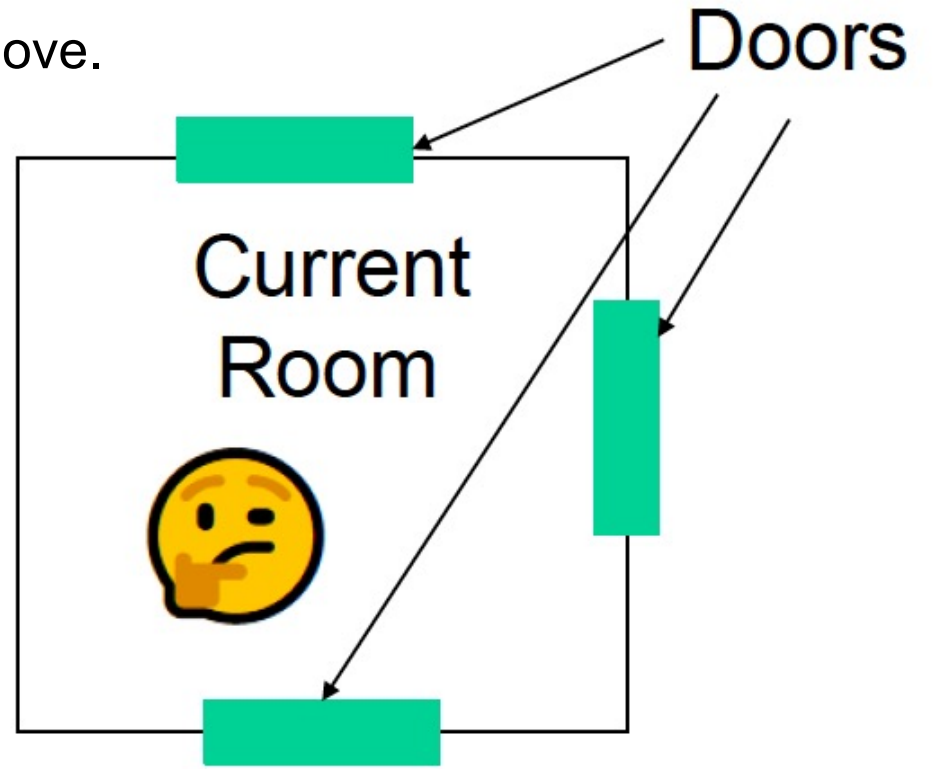
- What about if you do not have 8, but N queens:
  - You do not know how many *for* loops to write.
- Do the problem recursively - backtracking.
  - Recursive because later versions of the problem are just slightly simpler versions of the original.
    - You have n – i queens to add.
  - Backtracking because we may have to try different alternatives.
    - Problem space consists of states (nodes) [chess board] and actions (paths that lead to new states) [placing a queen].
    - If a node only leads to failure go back to its "parent" node. Try other alternatives



make smaller problems

Recursive Backtracking

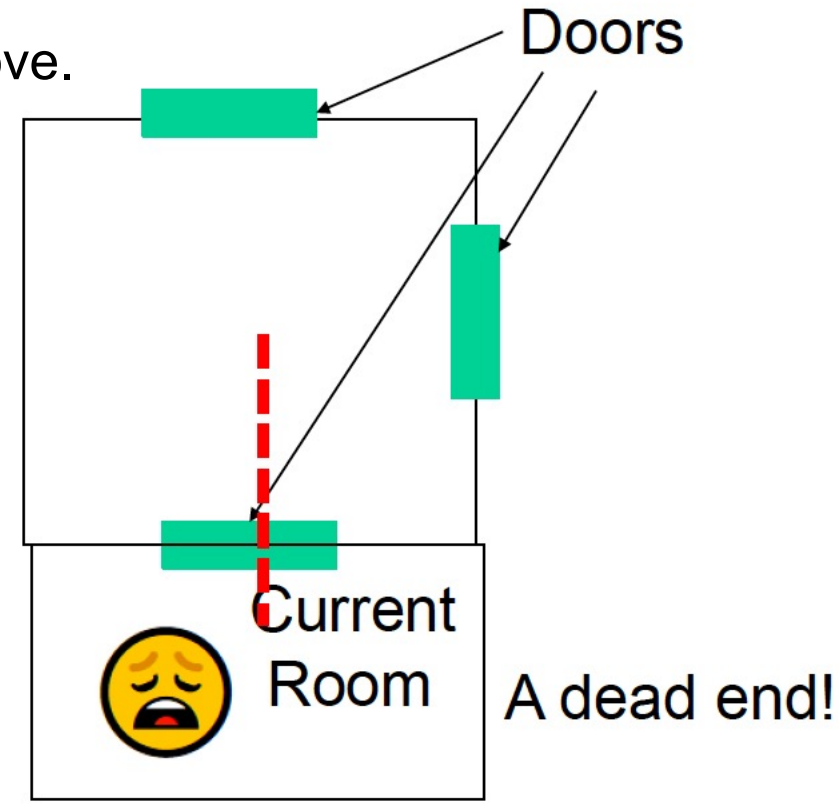
- Escaping a Maze.
  - A view form above.



  - Exit out there, some where to the south!

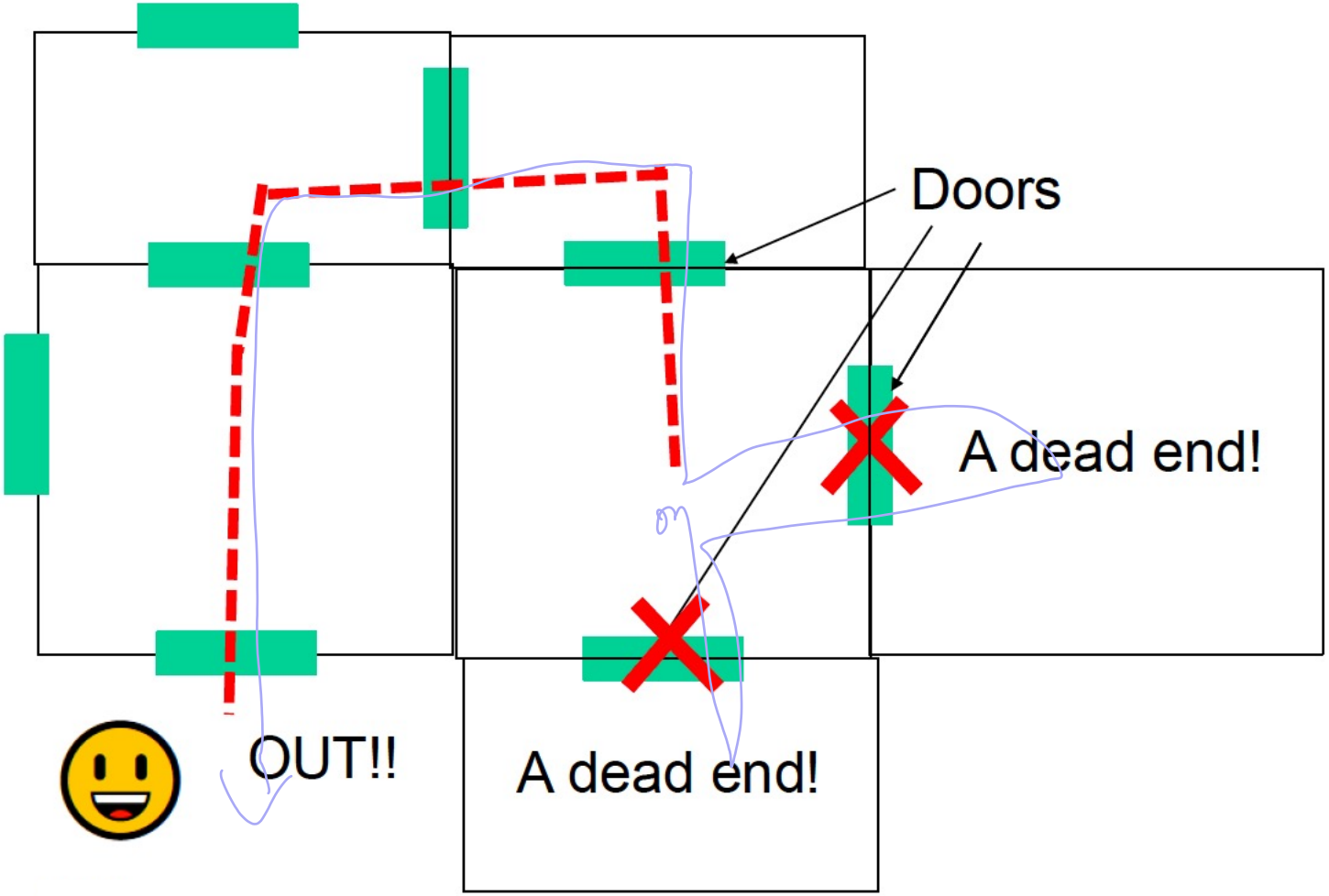
Recursive Backtracking

- Escaping a Maze.
  - A view form above.



  - Exit out there, some where to the south!

Recursive Backtracking



- Exit out there, some where to the south!

Recursive Backtracking – N Queen

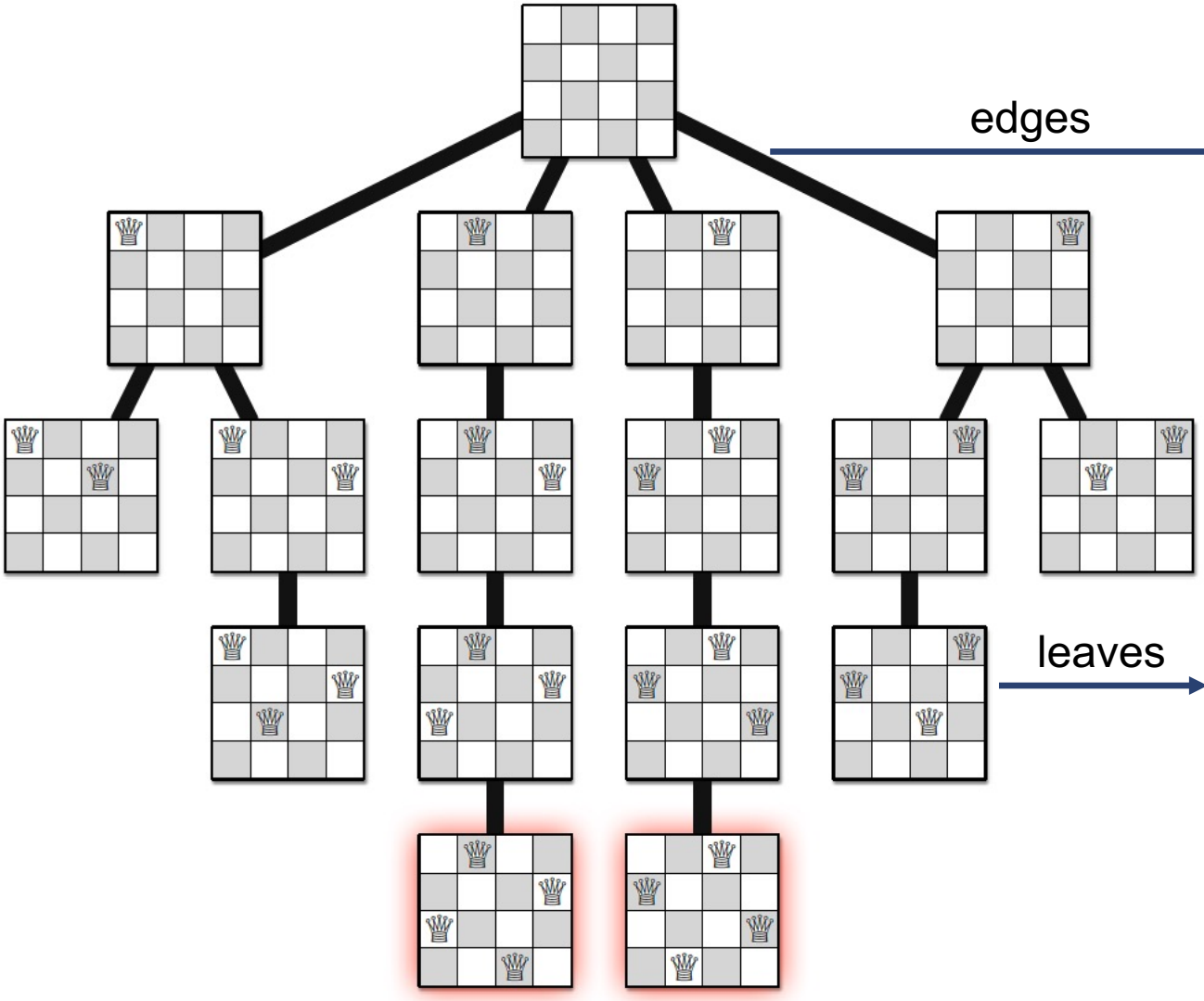
- We represent the positions of the queens using an array `Q[1 .. n]`, where `Q[i]` indicates which square in row `i` contains a queen.
- The index `r` is the index of the first empty row.
- The prefix `Q[1 ..r-1]` contains the positions of the first r -1 queens.

```
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!⟩⟩
```

All possible placements of a queen on row r

Checks whether a placement is consistent with the queens already on the first r-1 rows

N Queens – recursion tree

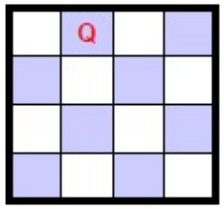


edges → Recursion calls

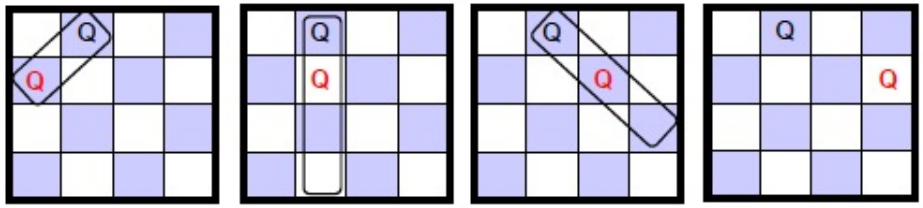
leaves → Solutions that can not be further extended

N Queens – recursion tree

- row 0:

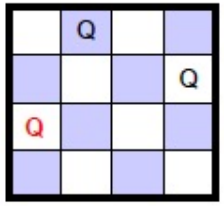


- row 1:

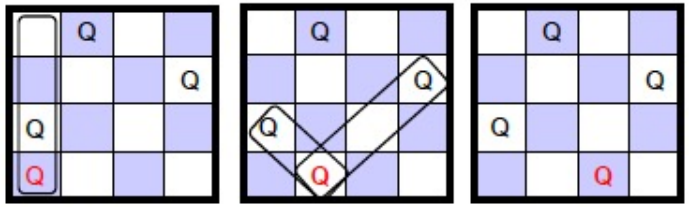


brute force = try all

- row 2:



- row 3:



A solution!

Recursive Backtracking

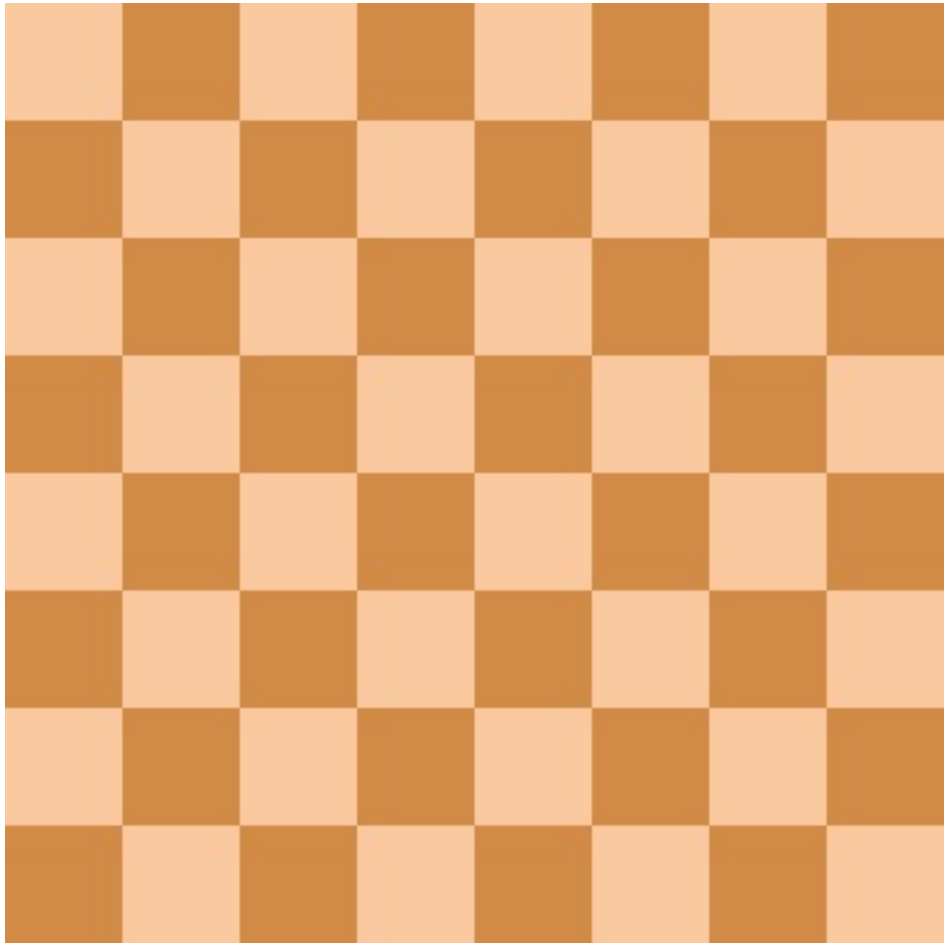


Image Credit: [wikipedia]

- You must practice!!!
- Learn to recognize problems that fit the pattern
- Clearly define your search space.
  - Correctness
  - Efficiency
- Be aware of your resources.
  - Time.
  - Memory.
- All solutions or a solution?
  - Find a path to success
  - Find all paths to success
  - Find the best path to success

Recursive Backtracking - template

If at a solution, report success

for (every possible choice from current state / node)

- Make that choice and take one step along path
- Use recursion to try to solve the problem for the new node / state
- If the recursive call succeeds, report the success to the previous level
- Back out of the current choice to restore the state at the beginning of the loop.

Report failure

Recursive Backtracking - Sudoku

- Sudoku
- 9 by 9 matrix with some numbers filled in
- all numbers must be between 1 and 9
- Goal: Each row, each column, and each mini matrix must contain the numbers between 1 and 9 once each
  - no duplicates in rows, columns, or mini matrices

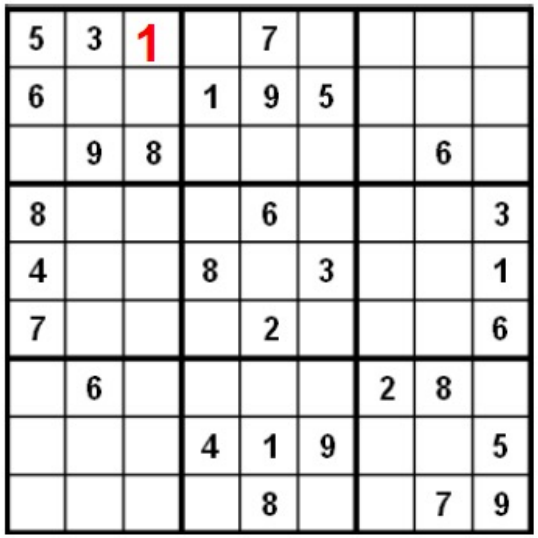
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

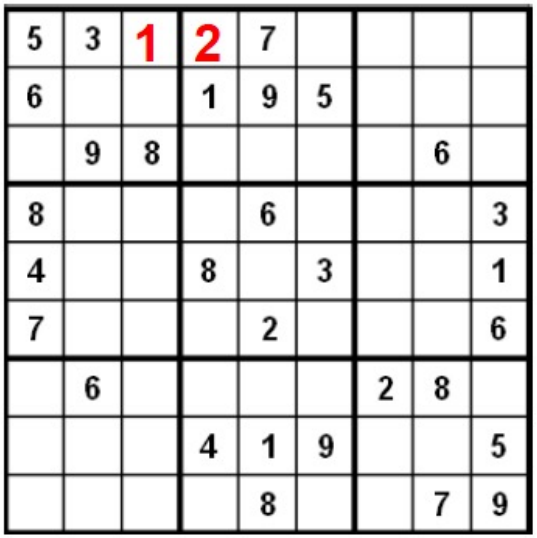
Recursive Backtracking - Sudoku

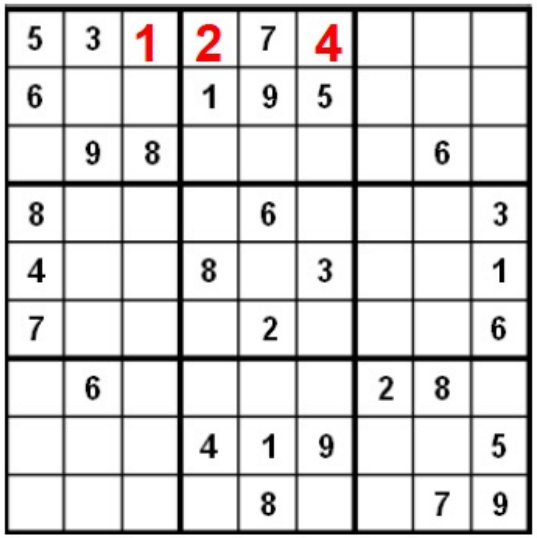
- Brute force Sudoku
  - if not open cells, solved
  - scan cells from left to right, top to bottom for first open cell
  - When an open cell is found start cycling through digits 1 to 9.
  - When a digit is placed check that the set up is legal
  - now solve the board

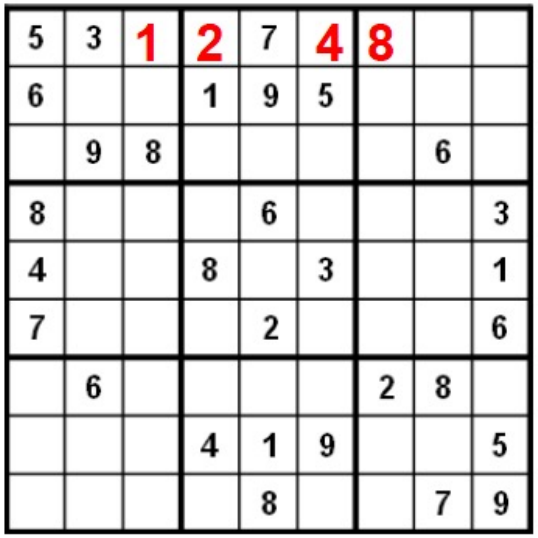
5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

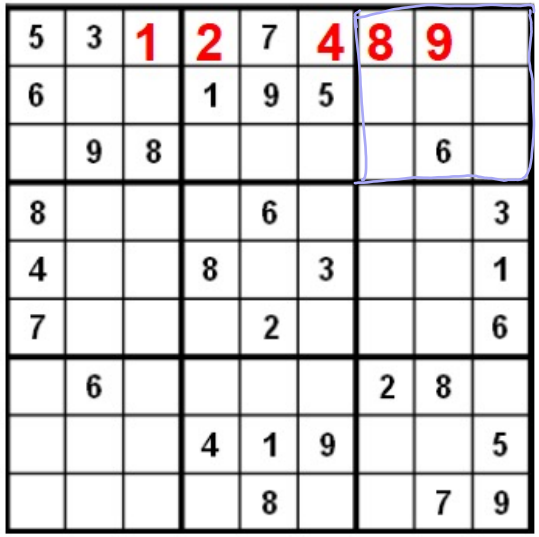
Recursive Backtracking - Sudoku











uh oh!

Recursive Backtracking - Sudoku

- When the search reaches a dead end it ***backs up*** to the previous cell it was trying to fill and goes onto to the next digit.
- We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again
- so the algorithm needs to remember what digit to try next
- Now in the cell with the 8. We try and 9 and move forward again.

5	3	1	2	7	4	8	9	
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

5	3	1	2	7	4	9		
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

Recursive Backtracking - Sudoku

- When the search reaches a dead end it ***backs up*** to the previous cell it was trying to fill and goes onto to the next digit.
- We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again
- so the algorithm needs to remember what digit to try next
- Now in the cell with the 8. We try and 9 and move forward again.

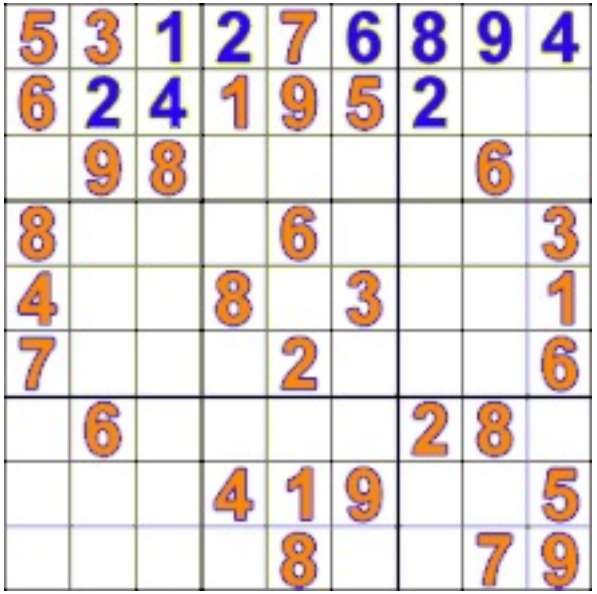


Image Credit: [wikipedia]

Recursive Backtracing - Conclusions

- Generating versus Filtering:
  - Programs that generate lots of candidate solutions and then choose the ones that are correct are called 'filters' - recall the naive 8-queens solvers.
  - Those that hone in exactly to the correct answer without any false starts are called 'generators' – recall the improved 8-queens solver with 12 solutions.
  - Generally, filters are easier to code but run slower. Do the math to see if a filter is good enough
- Prune Infeasible Search Space Early:
- Utilize Symmetries.
  - In the 8-queens problem, there are 92 solutions but there are only 12 unique (or fundamental) solutions as there are rotations and reflections symmetries in this problem. You can utilize this fact by only generating the 12 unique solutions and, if needed, generate the whole 92 by rotating and reflecting these 12 unique solutions.

Recursive Backtracing - Conclusions

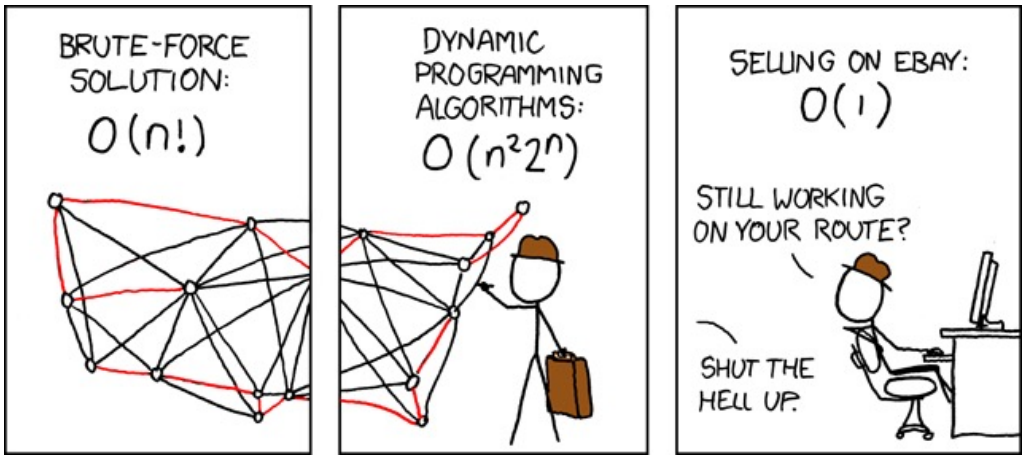
- Backtracking algorithms are commonly used to make a sequence of decisions, with the goal of building a recursively defined structure satisfying certain constraints.
  - In the n-queens problem, the goal is a sequence of queen positions, one in each row, such that no two queens attack each other. For each row, the algorithm decides where to place the queen.
- In each recursive call to the backtracking algorithm, we need to make exactly one decision, and our choice must be consistent with all previous decisions.
  - For the n-queens problem, we must pass in not only the number of empty rows, but the positions of all previously placed queens.
- Finally, once we've figured out what recursive problem we really need to solve, we solve that problem by recursive brute force:
  - Try all possibilities for the next decision that are consistent with past decisions, and let the recursion worry about the rest.

Recursive Backtracing - Conclusions

- Useful to solve decision, optimization and enumeration problems.
- A lot of applications.
  - Puzzles
  - Combinatorial optimization
  - Logic programming
  - Constraint satisfaction problems
  - Huge in artificial intelligence.
    - Smart agent searching an $O(2^{64})$ space VS naïve agent searching an O(12) space.
    - Machine learning (deep learning, reinforce learning)

Recursive Backtracing - Conclusions

- Problems that are solved by backtracking can usually use the same recursive strategy to solve many different variants of the same problem.
- Brute-Force is slow.
  - Prune your search space.
  - Is it the only way to solve it?
  - What type of solutions do I need (all, any, the optimal?)



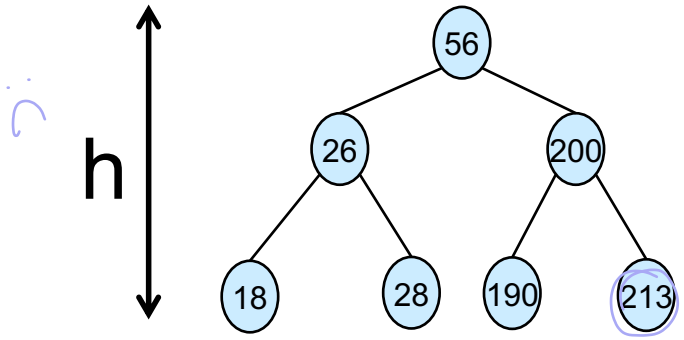
Taken from https://www.explainxkcd.com

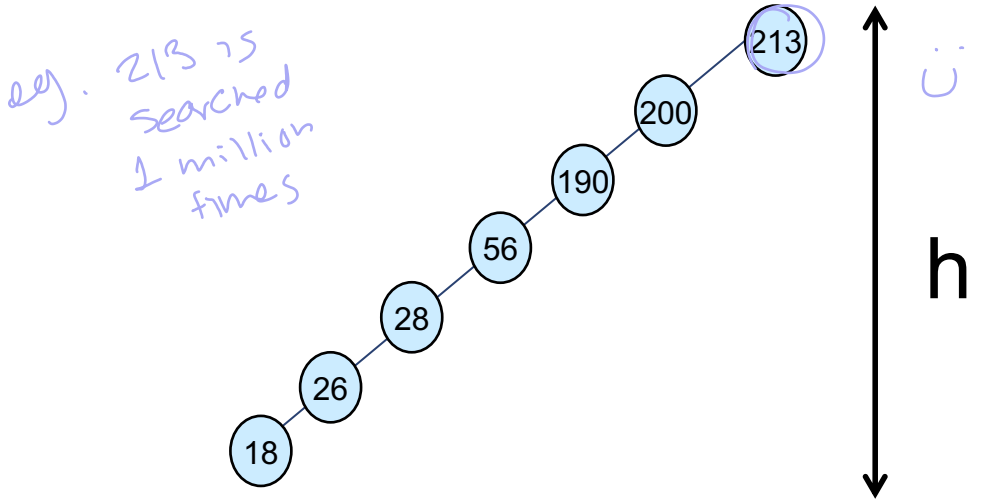
Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
- Recall that:
  - The running time for a successful search in a binary search tree is proportional to the depth of the tree (i.e., the number of ancestors of the target node).
  - To minimize the worst-case search time, the height of the tree should be as small as possible.
    - The ideal tree is perfectly balanced (remember AVL and red-black trees).

Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
- However:
  - In many BST applications, it is more important to minimize the total cost of several searches rather than the worst-case cost of a single search.
    - If x is a more frequent search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall depth of the tree.





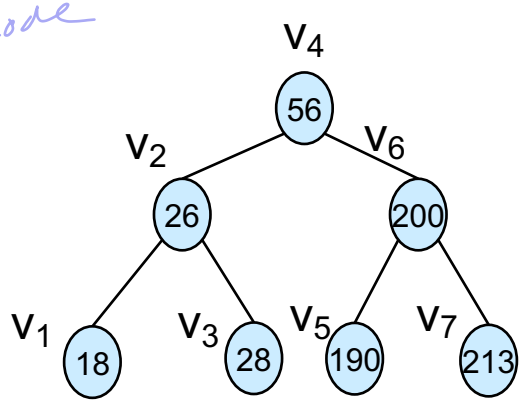
Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
  - Given an array of **sorted** keys `A[1 .. n]` and an array of corresponding access frequencies `f[1 .. n]`. Our task is to 'build' the binary search tree that minimizes the total search time, assuming that there will be exactly `f[i]` searches for each key `A[i]`.
  - Let `T` be a BST. Let $v_1, v_2, \ldots, v_n$ be the nodes of `T` such that each node v_i stores the corresponding key `A[i]`.
  - The total cost (ignoring constant factors) of performing all the searches is given by:

$$Cost(T, f[1..n]) := \sum_{i=1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } T$$

number of times searching for a specific node

need to return a tree that given a set of frequencies will minimize search time



A = [18,26,28,56,190,200,213]

Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
  - Now suppose v_r is the root of `T`.
    - v_r is an ancestor of every node in `T`.
    - If `i < r`, then all ancestors of v_i (except the root) are in the left subtree of `T`.
    - If `i > r`, then all ancestors of v_i (except the root) are in the right subtree of `T`.
    - Then, we can partition the cost function into three parts:

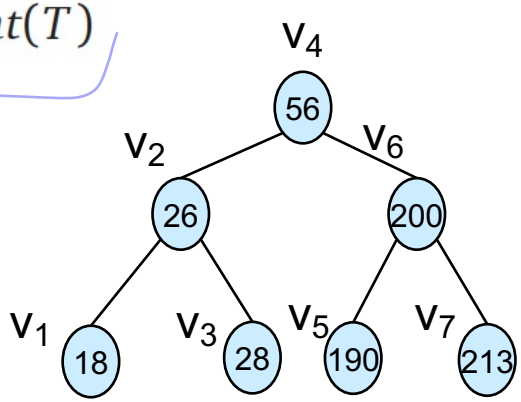
$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } left(T)$$

$$+ \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } right(T)$$

left children

number of times through root

right children



Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
  - So far, we have two ways to express the cost.

$$Cost(T, f[1..n]) := \sum_{i=1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } T$$

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } left(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } right(T)$$

  - Note that the second and third summations of the second equation look exactly like our first equation.

Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
  - Simple substitution give us a recurrence for the cost.

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + Cost(left(T), f[1..r-1]) + Cost(right(T), f[r+1..n])$$

  - The task now is to compute the tree T_{opt} that minimize this cost function.
    - Once we choose the correct key to store at the root, the recursion (i.e., recursive backtracking) will construct the rest of the optimal tree.
      - The left subtree `left(`T_{opt}`)` must be the optimal BST for the keys `A[1..r-1]` and access frequencies `f[1..r-1].`
      - The right subtree `right(`T_{opt}`)` must be the optimal BST for the keys `A[r+1..n]` and access frequencies `f[r+1..n].`

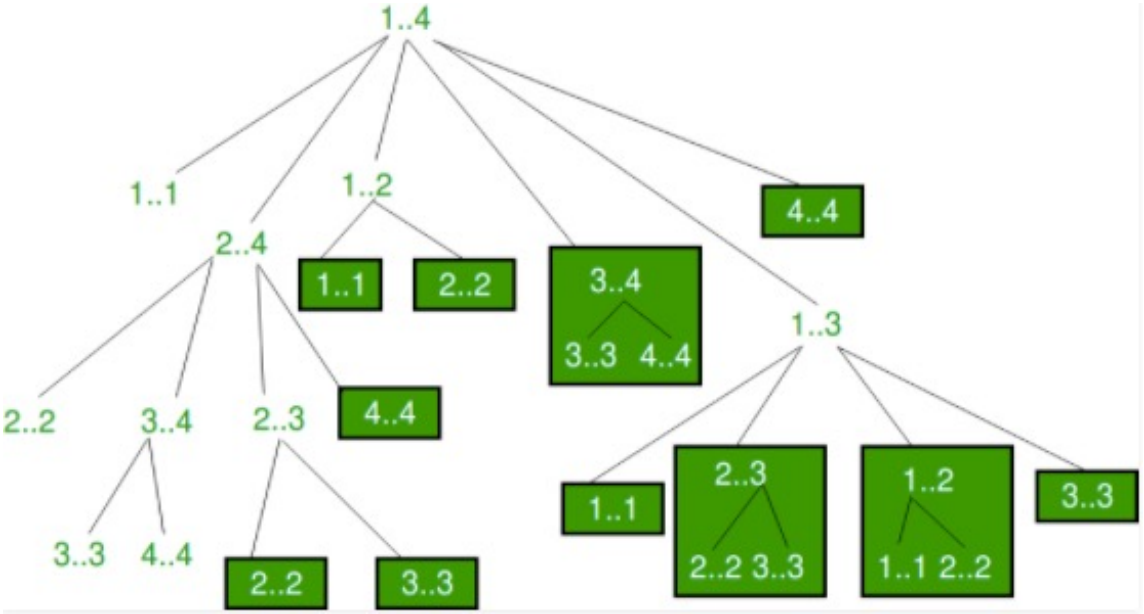
Recursive Backtracking + Divide and Conquer

- Optimal Binary Search Trees.
  - More generally, let `OptCost(i,k)` denote the total cost of the optimal search tree for the interval of frequencies f[i..k].

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

  - This recursive definition can be translated into a recursive backtracking algorithm to compute `OptCost(1,n)`.
    - We one by one try all nodes as root (`r` varies from `i` to `k` in second term)

Recursive Backtracking + Divide and Conquer



Picture from https://www.geeksforgeeks.org/

- The running time satisfies the recurrence:

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) \; + \; O(n)$$

Recursive Backtracking + Divide and Conquer

- The running time satisfies the recurrence:

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) \; + \; O(n)$$

Recursive Backtracking + Divide and Conquer

- The running time satisfies the recurrence (supplemental):

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) + O(n)$$

k	left	right
1	T(0)	T(7)
2	T(1)	T(6)
3	T(2)	T(5)
4	T(3)	T(4)
5	T(4)	T(3)
6	T(5)	T(2)
7	T(6)	T(1)
8	T(7)	T(0)

$$T(n) = 2\sum_{k=0}^{n-1} T(k) + \alpha n$$

Recursive Backtracking + Divide and Conquer

- The running time satisfies the recurrence:

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) + O(n)$$

- By solving the recurrence (supplemental material).

$$T(n) = 2\sum_{k=0}^{n-1} T(k) + \alpha n$$

Replacing O() with a constant

$$T(n-1) = 2\sum_{k=0}^{n-2} T(k) + \alpha(n-1)$$

Recurrence for T(n-1)

$$T(n) - T(n-1) = 2T(n-1) + \alpha$$

Substracting the recurrences

$$T(n) = 3T(n-1) + \alpha$$

Simplifying the recurrence

$$\boldsymbol{T(n) = O(3^n)}$$

Recursion tree method (topic of next class)

Recursive Backtracking + Divide and Conquer

- The running time satisfies the recurrence:

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) + O(n)$$

$$\boldsymbol{T(n) = O(3^n)}$$

- The number of binary search trees with n vertices satisfies the recurrence:

$$N(n) = \sum_{r=1}^{n-1} \big(N(r-1) \cdot N(n-r)\big)$$

$$N(n) = \Theta(4^n/\sqrt{n})$$

Note: No obvious:
https://math.stackexchange.com/questions/1986247/asymptotic-approximation-of-catalan-numbers

Recursive Backtracking + Divide and Conquer

- The running time satisfies the recurrence: $T(n) = O(3^n)$
- The number of binary search trees with n vertices satisfies the recurrence: $N(n) = \Theta(4^n/\sqrt{n})$
- The analysis implies:
  - Our recursive algorithm does **not** examine all possible BST.
  - Our algorithm saves considerable time by searching independently for the optimal left and right subtrees for each root.
    - A full enumeration of binary search trees would consider all possible pairs of left and right subtrees (hence the product in the recurrence for N(n))

Divide and Conquer

Algorithmic Paradigms

- Complete Search.
- Divide and Conquer.
- Dynamic Programming.
- Greedy

Divide and Conquer

- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem


time for questions