Divide and Conquer Algorithms, and Backtracking

CSE 101: Design and Analysis of Algorithms

Lecture 16

CSE 101: Design and analysis of algorithms

- Divide and conquer algorithms
  - Reading: Chapter 2
- Homework 6 is due today, 11:59 PM
- Quiz 3 is Nov 27, in class
  - Greedy algorithms and divide and conquer algorithms
  - Study guide released tomorrow
- Homework 6 will be assigned Nov 29
  - No homework this week

Divide and conquer example

GREATEST OVERLAP

Greatest overlap

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- An interval [a,b] is a set of integers starting at a and ending at b. For example: [16,23] = {16,17,18,19,20,21,22,23}
- An overlap between two intervals [a,b] and [c,d] is their intersection
- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

Based on slides courtesy of Miles Jones

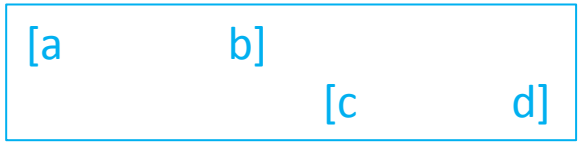
Greatest overlap

- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

procedure overlap([a,b],[c,d]) [assume that a ≤ c]

if b < c:

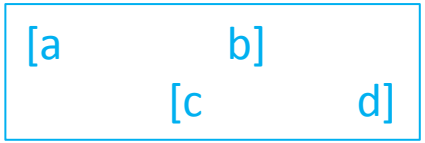
return 0



else:

if b ≤ d:

return b – c + 1



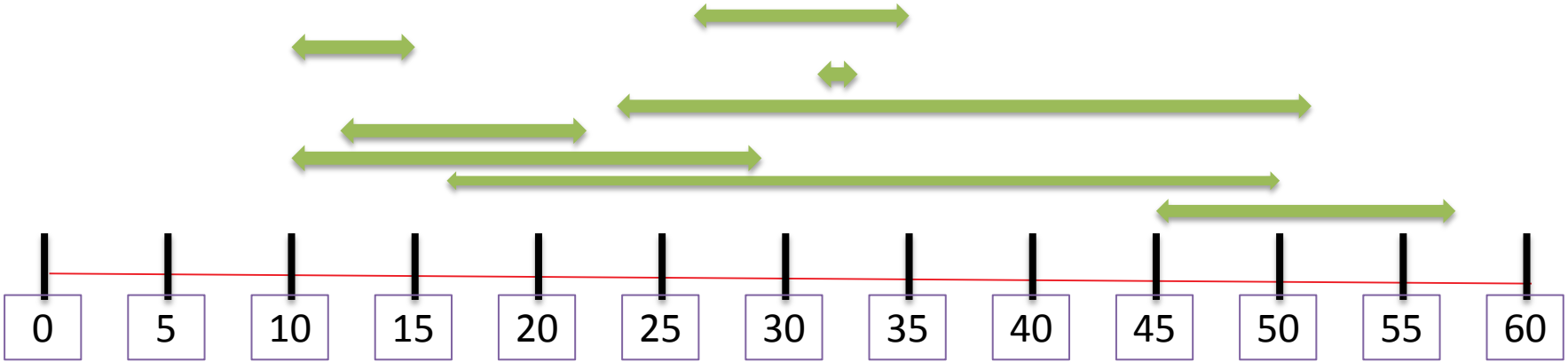
if b > d:

return d – c + 1

Greatest overlap

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Example: What is the greatest overlap of the intervals

  [45,57],[17,50],[10,29],[12,22],[23,51],[31,32],[10,15],[23,35]



Greatest overlap

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Simple solution: compute all overlap pairs and find maximum

```
olap:=0
for i from 1 to n-1
    for j from i+1 to n
        if overlap([a_i,b_i],[a_j,b_j])>olap then
            olap:=overlap([a_i,b_i],[a_j,b_j])
return olap
```

$O(n^2)$ Can we do better?

Greatest overlap

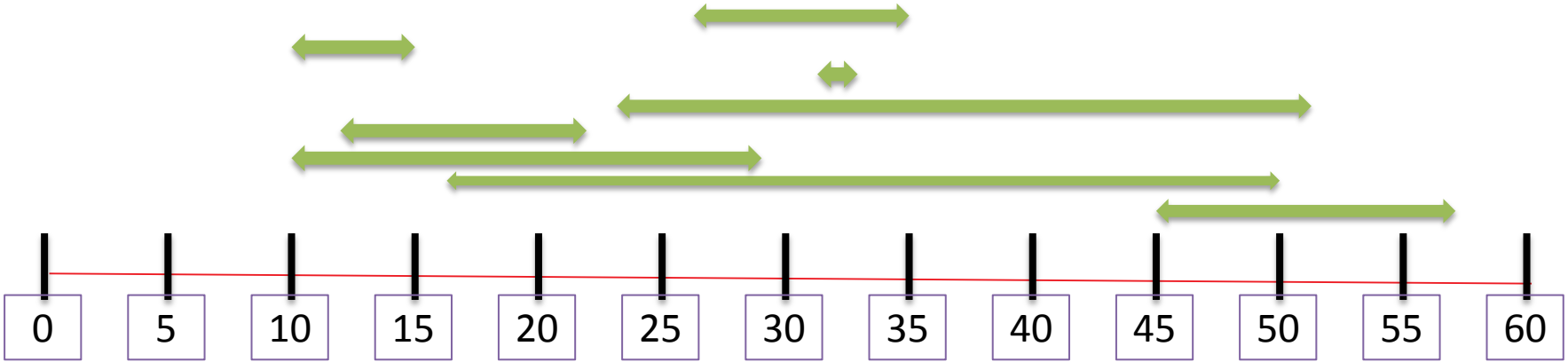
- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
  - Compose your base case
  - Break the problem into smaller pieces
  - Recursively call the algorithm on the smaller pieces
  - Combine the results

Greatest overlap

- Compose your base case
  - What happens if there is only one interval?
    - If n=1, then return 0

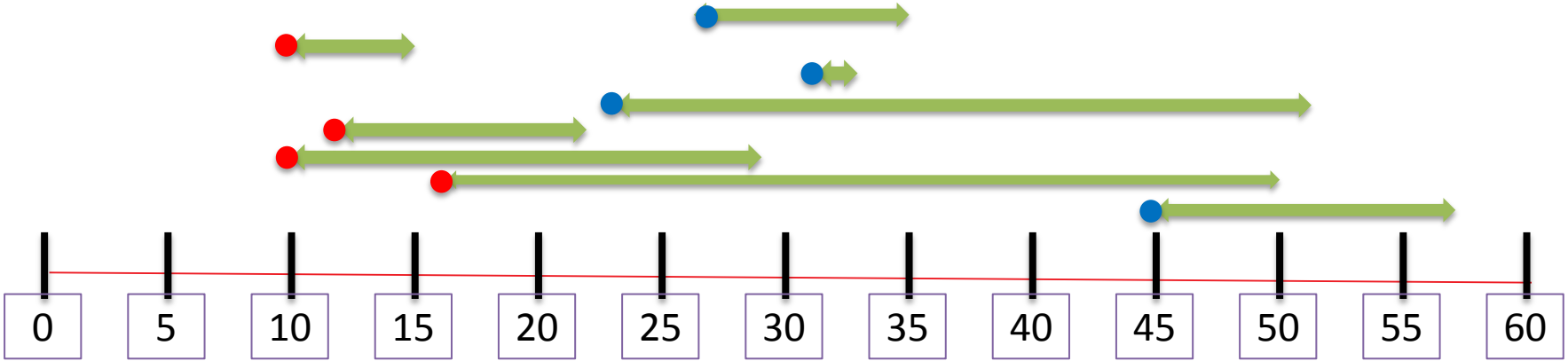
Greatest overlap

- Break the problem into smaller pieces
  - Would knowing the result on smaller problems help with knowing the solution on the original problem?
  - In this stage, let's keep the combine part in mind
  - How would you break the problem into smaller pieces?



Greatest overlap

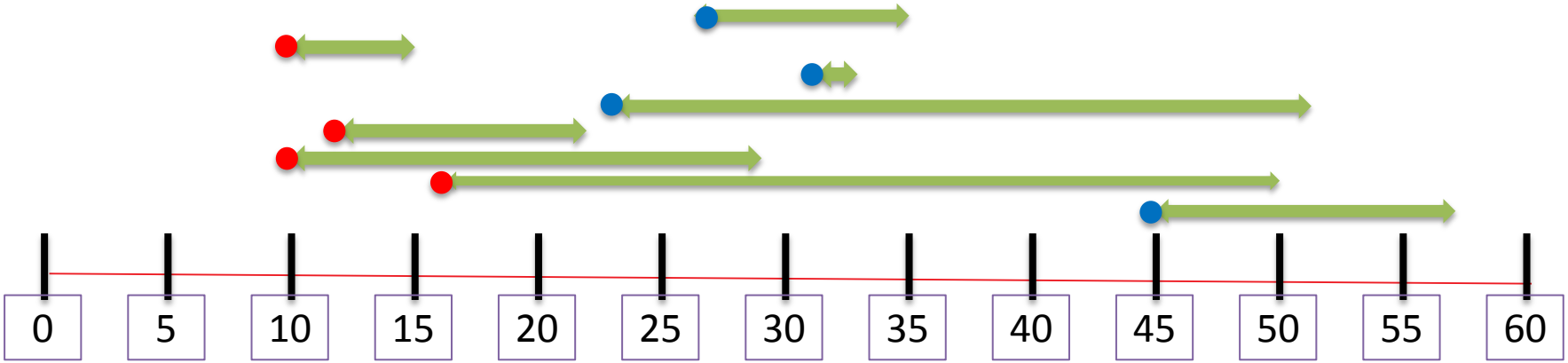
- Break the problem into smaller pieces
  - Would it be helpful to break the problem into two depending on the starting value?



Greatest overlap

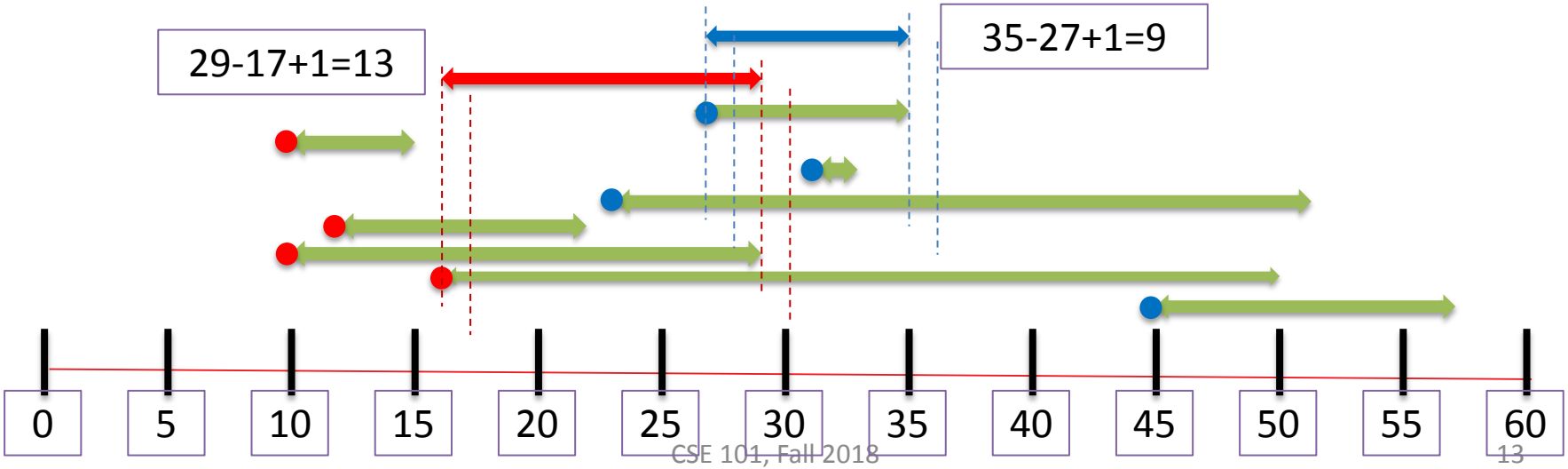
- Break the problem into smaller pieces
  - Sort the list and break it into lists each of size n/2

    [10,15],[10,29],[12,22],[17,50],[23,51],[27,35],[31,32],[45,57]



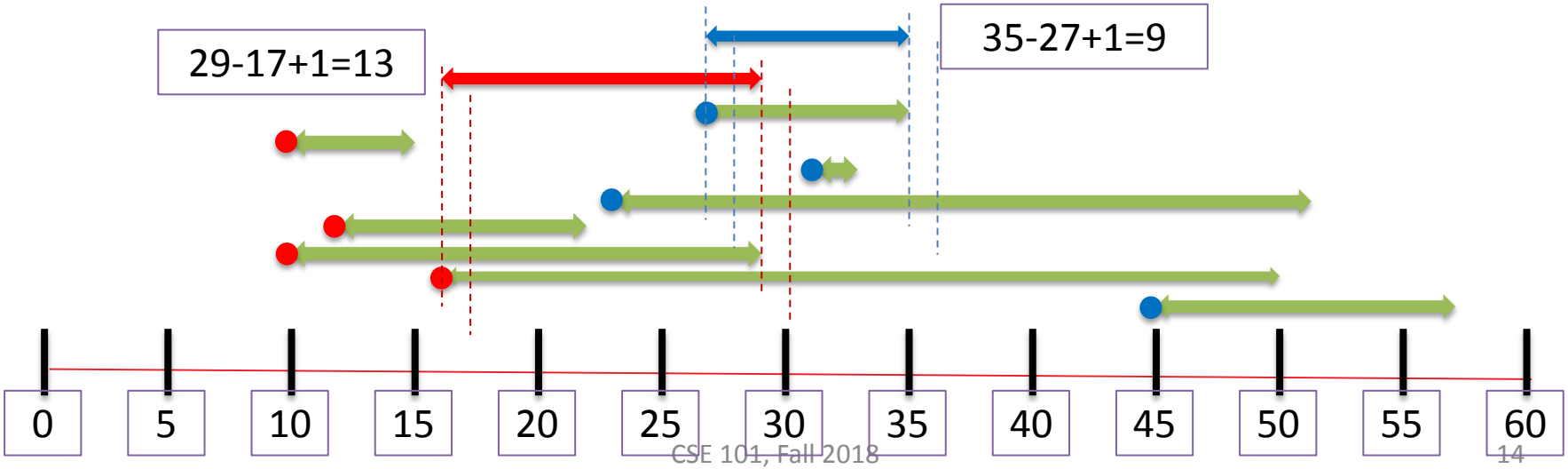
Greatest overlap

- Break the problem into smaller pieces
  - Let's assume we can get a divide and conquer algorithm to work. Then what information would it give us to recursively call each subproblem?
    - overlapDC([10,15],[10,29],[12,22],[17,50])=13
    - overlapDC([23,51],[27,35],[31,32],[45,57])=9



Greatest overlap

- Break the problem into smaller pieces
  - overlapDC([10,15],[10,29],[12,22],[17,50])=13
  - overlapDC([23,51],[27,35],[31,32],[45,57])=9
  - The greatest overlap overall may be contained entirely in one sublist or it may be an overlap of one interval from either side



Greatest overlap

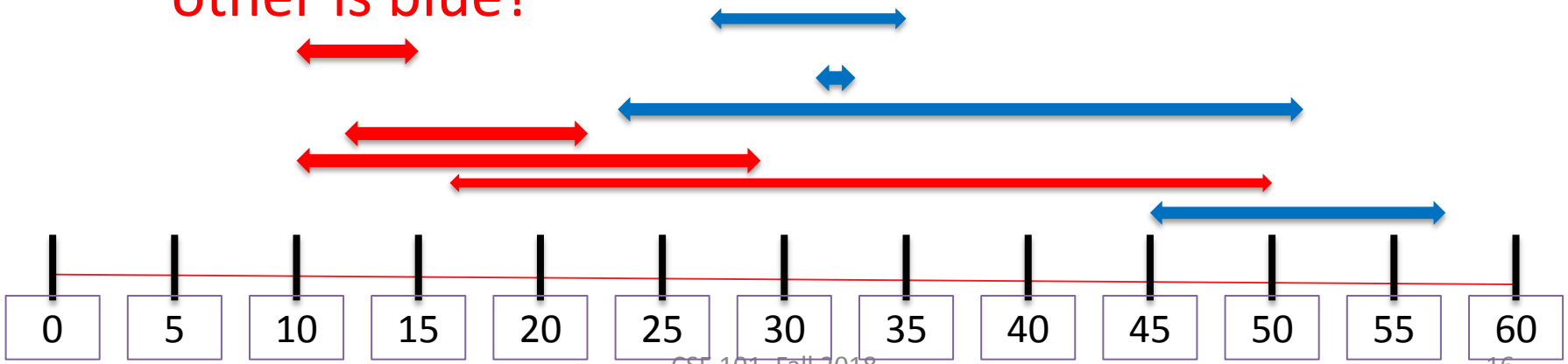
- Combine the results
  - So far, we have split up the set of intervals and recursively called the algorithm on both sides. The runtime of this algorithm satisfies a recurrence that looks something like

$$T(n) = 2T\left(\frac{n}{2}\right) + O(?)$$

  - What goes into the $O(?)$?
  - How long does it take to "combine". In other words, how long does it take to check if there is not a bigger overlap between sublists?

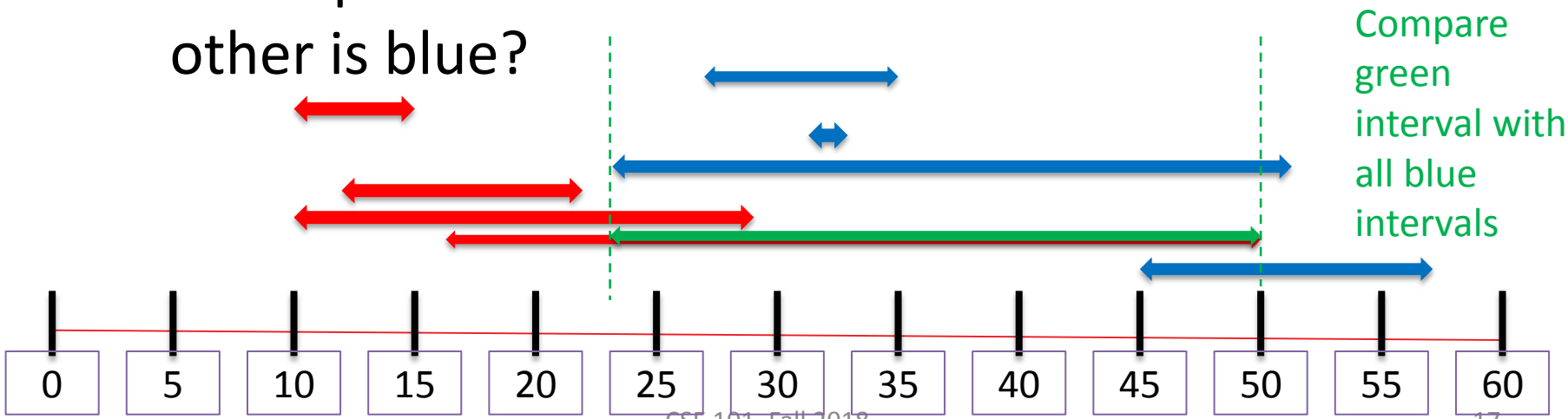
Greatest overlap

- Combine the results
  - What is an efficient way to determine the greatest overlap of intervals where one is red and the other is blue?



Greatest overlap

- Combine the results
  - What is an efficient way to determine the greatest overlap of intervals where one is red and the other is blue?



Greatest overlap between sets

- Let's formalize our algorithm that finds the greatest overlap of two intervals such that they come from different sets sorted by starting point

procedure **overlapbetween** $([[a_1, b_1], \ldots [a_\ell, b_\ell]], [[c_1, d_1], \ldots [c_k, d_k]])$

$(a_1 \leq a_2 \leq \cdots \leq a_\ell \leq c_1 \leq c_2 \leq \cdots \leq c_k)$

if k==0 or $\ell == 0$ then return 0
minc = c_1
maxb = 0
olap = 0
for i from 1 to ℓ:
    if maxb < b_i:
        maxb = b_i

Find the latest ending red. O(n/2)

for j from 1 to k:
    if olap < overlap([minc,maxb],$[c_k, d_k]$):
        olap = overlap([minc,maxb],$[c_k, d_k]$)
return olap

Compare it with blues. O(n/2)

Total runtime O(n)

Greatest overlap

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals

procedure **overlapdc** ($[a_1, b_1]$, … $[a_n, b_n]$)

if n==1 then return 0.
sort($[a_1, b_1]$, … $[a_n, b_n]$) by a values.
mid = $\lfloor n/2 \rfloor$
LS = $[[a_1, b_1]$, … $[a_{mid}, b_{mid}]]$
RS = $[[a_{mid+1}, b_{mid+1}]$, … $[a_n, b_n]]$
olap1 = **overlapdc**(LS)
olap2 = **overlapdc**(RS)
olap3 = **overlapbetween**(LS,RS)
return max(olap1,olap2,olap3)

Greatest overlap, runtime

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals

procedure **overlapdc** $([a_1, b_1], \dots [a_n, b_n])$ $T(n)$

if n==1 then return 0.
sort($[a_1, b_1], \dots [a_n, b_n]$) by a values. $O(n \log n)$
mid = $\lfloor n/2 \rfloor$
LS = $[[a_1, b_1], \dots [a_{mid}, b_{mid}]]$
RS = $[[a_{mid+1}, b_{mid+1}], \dots [a_n, b_n]]$
olap1 = **overlapdc**(LS) $T(n/2)$
olap2 = **overlapdc**(RS) $T(n/2)$
olap3 = **overlapbetween**(LS,RS) $O(n)$
return max(olap1,olap2,olap3)

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

But each recursion sorts intervals that are already sorted

Greatest overlap

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals

Sort first

procedure **overlapdc** (sort($[a_1, b_1], \ldots [a_n, b_n]$))

Do not sort here

if n==1 then return 0.
mid = $\lfloor n/2 \rfloor$
LS = $[[a_1, b_1], \ldots [a_{mid}, b_{mid}]]$
RS = $[[a_{mid+1}, b_{mid+1}], \ldots [a_n, b_n]]$
olap1 = **overlapdc**(LS)
olap2 = **overlapdc**(RS)
olap3 = **overlapbetween**(LS,RS)
return max(olap1,olap2,olap3)

Greatest overlap, runtime

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals

$O(n \log n)$ Sort first

procedure **overlapdc** (sort($[a_1, b_1], \ldots [a_n, b_n]$)) $T(n)$

if n==1 then return 0.
mid = $\lfloor n/2 \rfloor$
LS = $[[a_1, b_1], \ldots [a_{mid}, b_{mid}]]$
RS = $[[a_{mid+1}, b_{mid+1}], \ldots [a_n, b_n]]$
olap1 = **overlapdc**(LS) $T(n/2)$
olap2 = **overlapdc**(RS) $T(n/2)$
olap3 = **overlapbetween**(LS,RS) $O(n)$
return max(olap1,olap2,olap3)

$2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$ by master theorem

$T(n) = O(n \log n) + O(n \log n) = O(n \log n)$

Greatest overlap

- Given a list of intervals [a_1,b_1],…,[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
  - It is clear that the greatest overlap will be from two intervals in the left half, two from the right half, or one from the left and one from the right
  - Our algorithm finds all three of these values and outputs the max

Divide and conquer example

MINIMUM DISTANCE

Minimum distance

- Given a list of coordinates $[(x_1, y_1), \ldots, (x_n, y_n)]$, find the distance between the closest pair
- distance$((x_i, y_i), (x_j, y_j))$ = $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

Minimum distance

- Given a list of coordinates $[(x_1, y_1), \ldots, (x_n, y_n)]$, find the distance between the closest pair
- Brute force solution?

Minimum distance

- Given a list of coordinates $[(x_1, y_1), \ldots, (x_n, y_n)]$, find the distance between the closest pair
- Brute force solution

min = ∞

for i from 1 to n-1:

  for j from i+1 to n:

    d = distance($(x_i, y_i), (x_j, y_j)$)

    if min > d, min = d

return min

Minimum distance

- Given a list of coordinates $[(x_1, y_1), \ldots, (x_n, y_n)]$, find the distance between the closest pair
  - Base case
  - Break the problem up
  - Recursively solve each problem
    - Assume the algorithm works for the subproblems
  - Combine the results

Minimum distance

- Base case
  - What happens if there is only two points?

Minimum distance

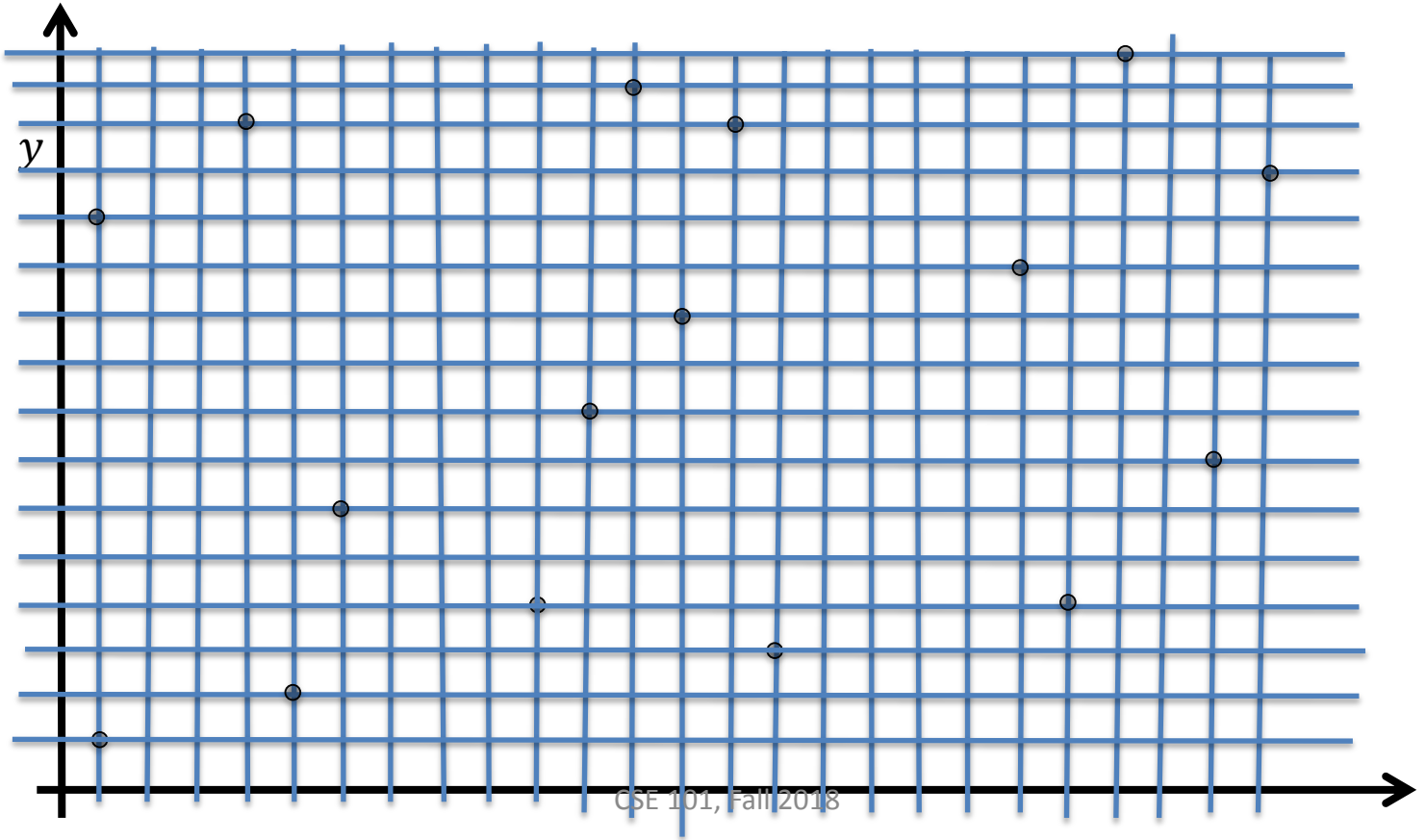
- Base case
  - What happens if there is only two points?
    - If n=2, then return distance($(x_1, y_1), (x_2, y_2)$)

Minimum distance

- Break the problem up
  - How would you break the problem into smaller pieces?

Example

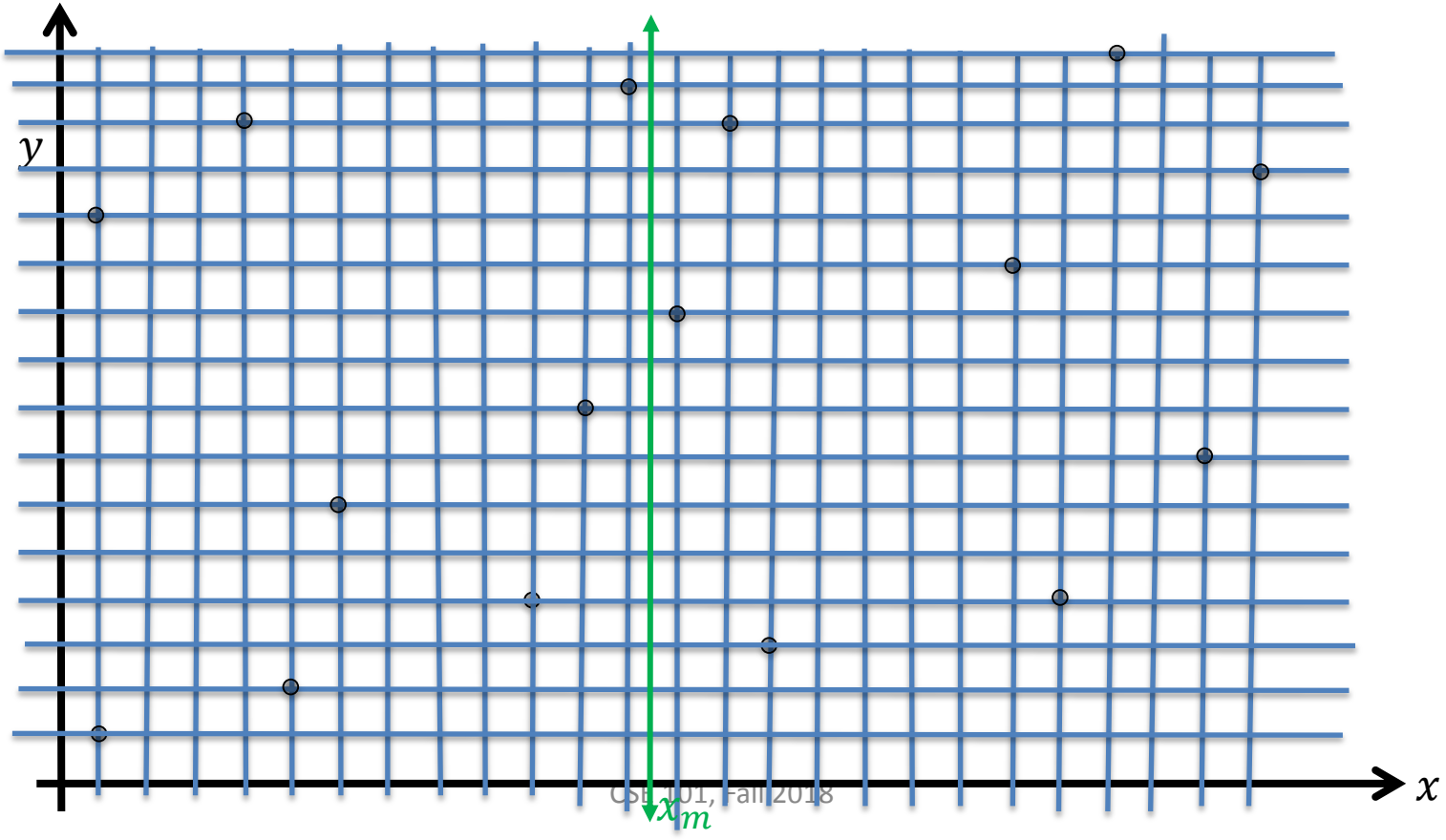
How would you break the problem into smaller pieces?



Minimum distance

- Break the problem up
  - Usually the smaller pieces are each of size n/2
  - We will break the problem in half, sort the points by their x values, then find a value x_m such that half of the x values are on the left and half are on the right

Example



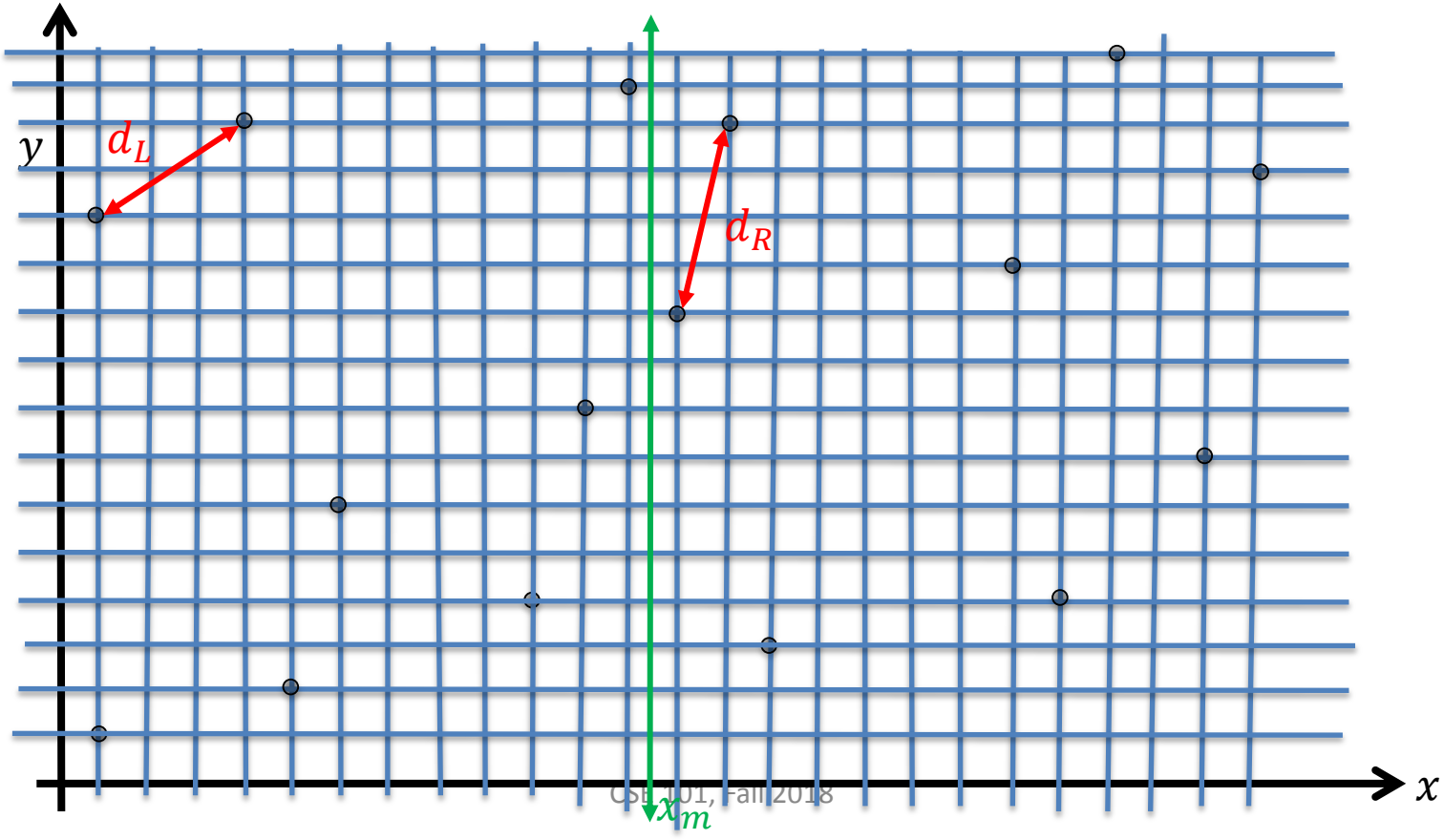
Minimum distance

- Break the problem up
  - Usually the smaller pieces are each of size n/2
  - We will break the problem in half, sort the points by their x values, then find a value x_m such that half of the x values are on the left and half are on the right
  - Perform the algorithm on each side
    - Assume the algorithm works for the subproblems (have faith)
  - What does that give us?

Minimum distance

- Break the problem up
  - Usually the smaller pieces are each of size n/2
  - We will break the problem in half, sort the points by their x values, then find a value x_m such that half of the x values are on the left and half are on the right
  - Perform the algorithm on each side
    - Assume the algorithm works for the subproblems (have faith)
  - What does that give us?
  - It gives us the distance of the closest pair on the left and on the right; let's call them d_L and d_R

Example



Minimum distance

- Combine the results
  - How will we use this information to find the distance of the closest pair in the whole set?

Minimum distance

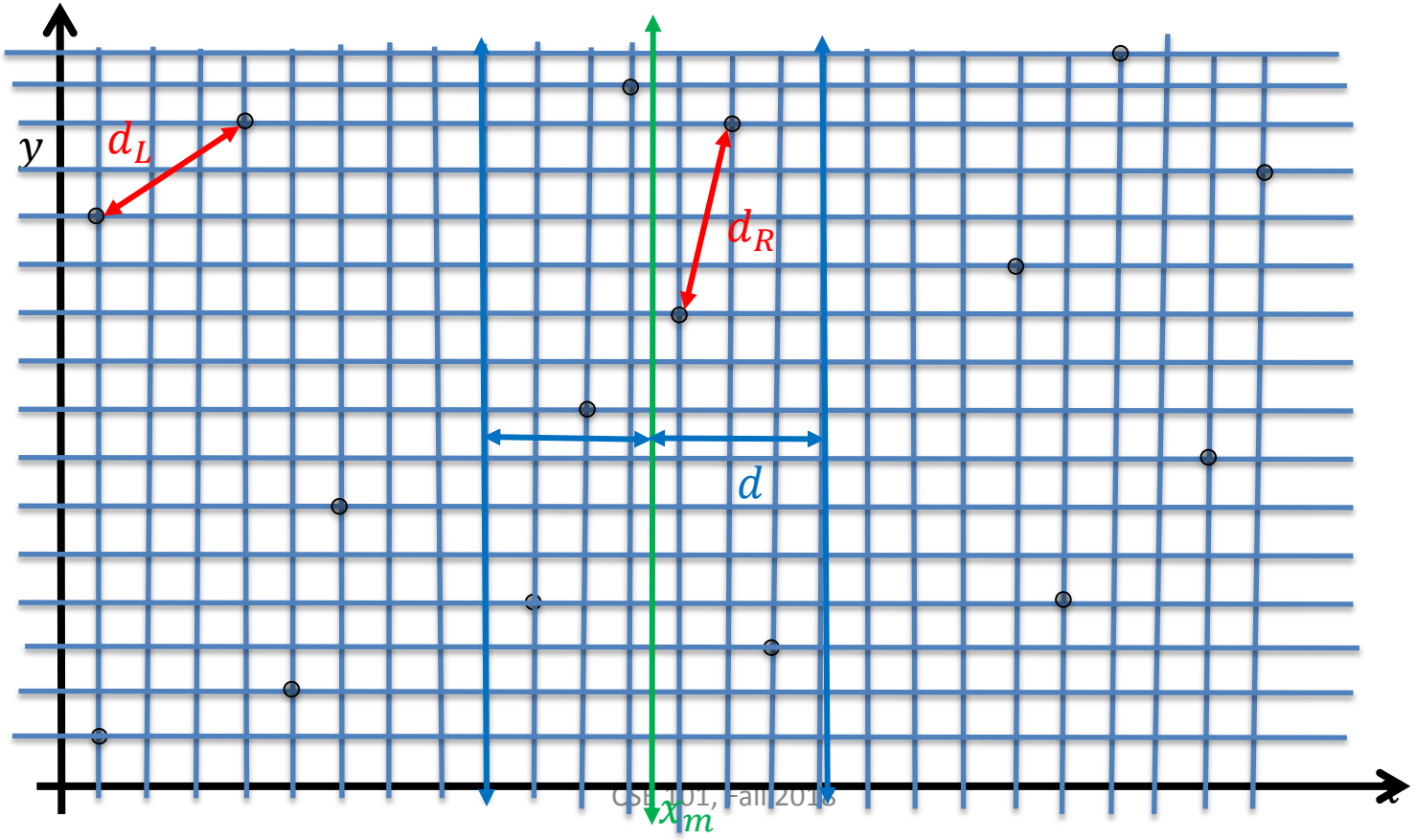
- Combine the results
  - How will we use this information to find the distance of the closest pair in the whole set?
  - We must consider if there is a closest pair where one point is in the left half and one is in the right half
  - How do we do this?

Minimum distance

- Combine the results
  - How will we use this information to find the distance of the closest pair in the whole set?
  - We must consider if there is a closest pair where one point is in the left half and one is in the right half
  - How do we do this?
  - Let $d = \min(d_L, d_R)$ and compare only the points (x_i, y_i) such that $x_m - d \leq x_i$ and $x_i \leq x_m + d$
  - Worst case, how many points could this be?

Example

Worst case, how many points could this be?



Minimum distance

- Combine the results
  - Let P_m be the set of points within d of x_m
  - Then, P_m may contain as many as n different points
  - So, to compare all the points in P_m with each other would take $\binom{n}{2}$ many comparisons
- So the runtime recursion is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$$

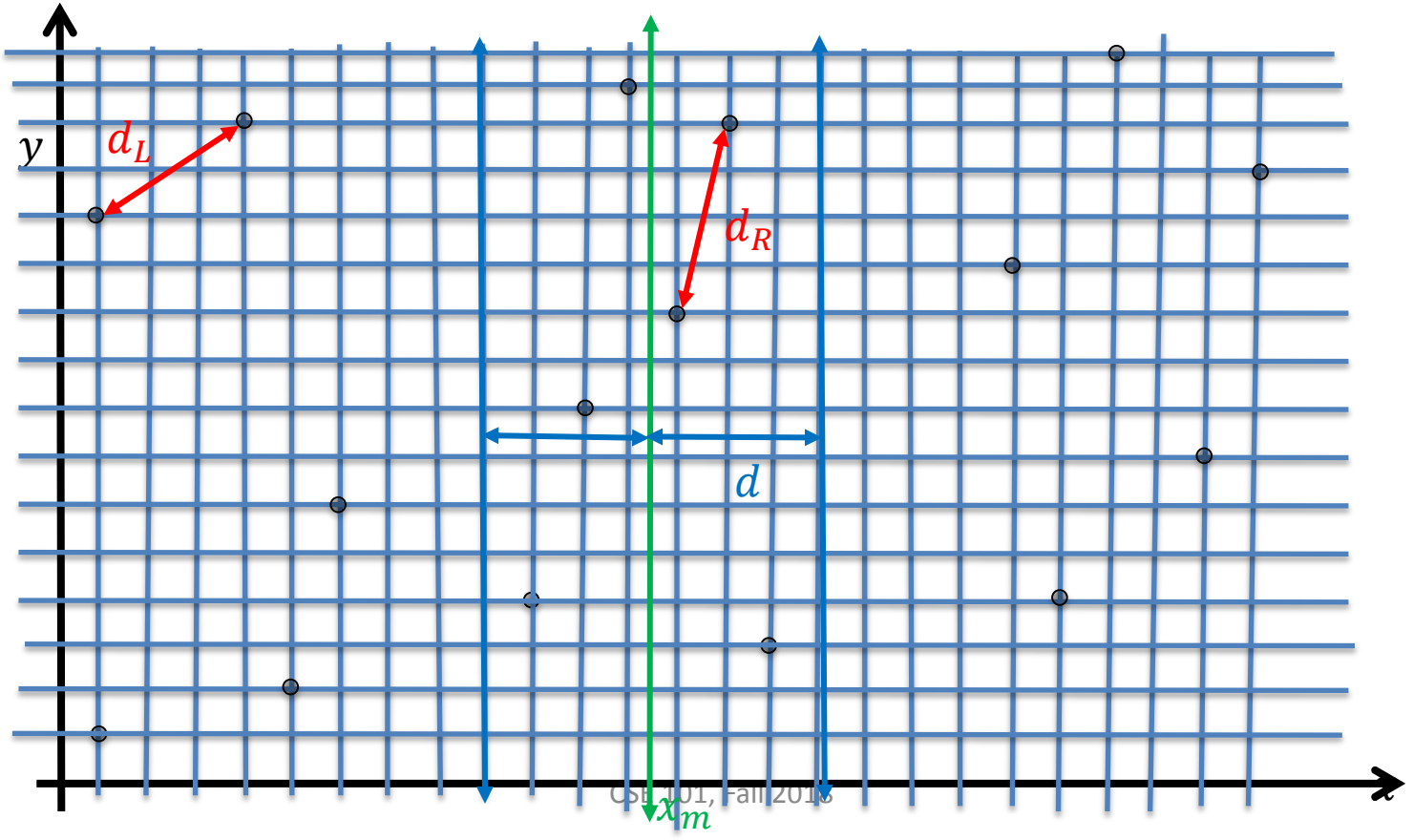
$$T(n) = O(n^2) \text{ by master theorem}$$

- Can we improve the combine term?

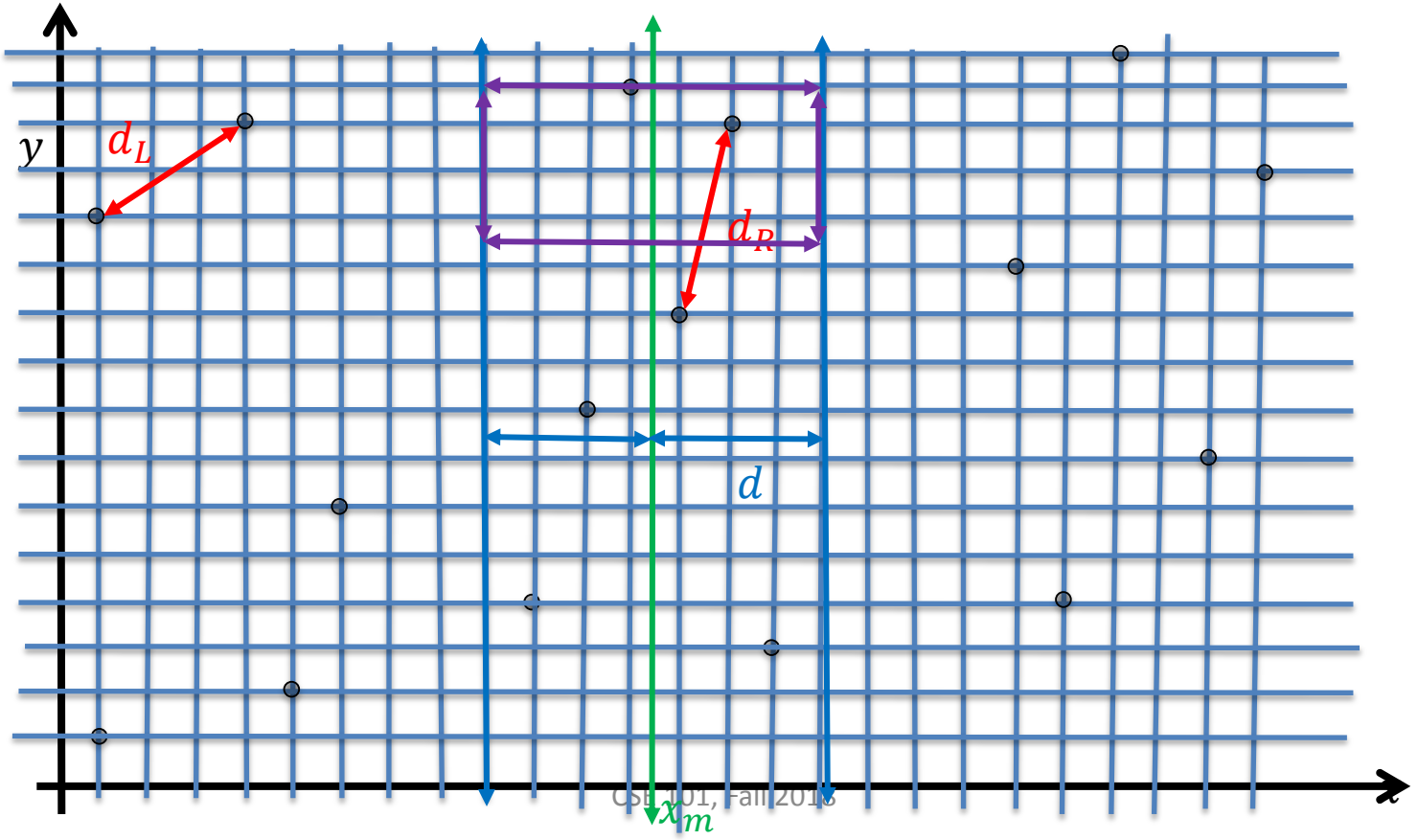
Minimum distance

- Combine the results
  - Let P_m be the set of points within d of x_m
  - Then, P_m may contain as many as n different points
  - Lets only compare the points i, j in P_m with each other if $|y_i - y_j| \leq d$
  - How many such comparisons are there?
  - How can we find all such pairs?

Example

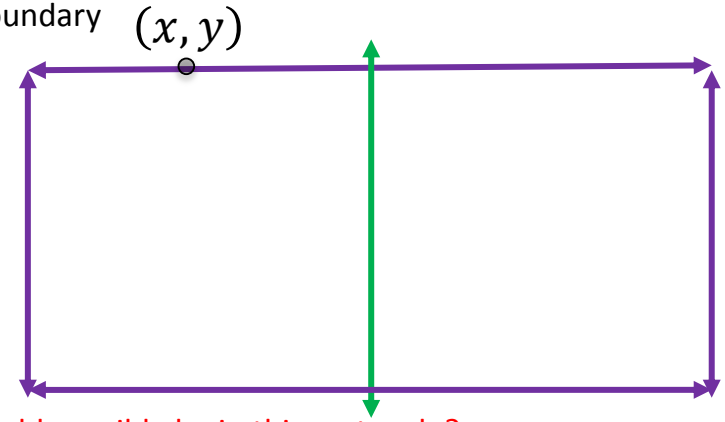


Example



Minimum distance

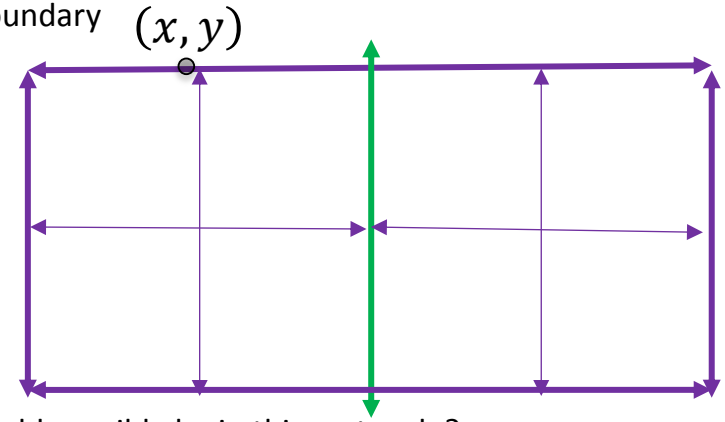
- Combine the results
  - Given a point $(x, y) \in P_m$, all the points we would need to compare it to are in a $2d \times d$ rectangle with that point at its upper boundary



  - How many points could possibly be in this rectangle?

Minimum distance

- Combine the results
  - Given a point $(x, y) \in P_m$, all the points we would need to compare it to are in a $2d \times d$ rectangle with that point at its upper boundary



  - How many points could possibly be in this rectangle?
  - There could not be more than 8 points total because if we divide the rectangle into eight $\frac{d}{2} \times \frac{d}{2}$ squares then there can never be more than one point per square

To learn more about this, investigate geometric point packing bound

Minimum distance

- Combine the results
  - So instead of comparing (x, y) with every other point in P_m we only have to compare it with at most 7 points lower than it (smaller y)
  - To gain quick access to these points, let's sort the points in P_m by y values
  - The points above must be in the 7 points before our current point in this sorted list
  - Now, if there are k vertices in P_m we have to sort the vertices in $O(k\log k)$ time and make at most $7k$ comparisons in $O(k)$ time for a total combine step of $O(k \log k)$
  - But, we said in the worst case, there are n vertices in P_m and so worst case, the combine step takes $O(n \log n)$ time
- So the runtime recursion is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

BACKTRACKING

Backtracking

- Backtracking is a generic method that can be applied to many problems that have a solution set that is exponentially large, e.g., search and optimization problems
- Backtracking can often be a first step towards finding a greedy or dynamic programming algorithm
- Backtracking often gives a more efficient runtime over exhaustive search or brute force but may not result in a polynomial time algorithm, and is usually an improved exponential time
- On the other hand, it applies even to NP-complete problems, where we do not expect to find less than exponential time algorithms
- Often, they are better on typical inputs than their worst-case

Search and optimization problems

- Many problems involve finding the best solution from among a large space of possibilities
- Instance: what does the input look like?
- Solution format: what does an output look like?
- Constraints: what properties must a solution have?
- Objective function: what makes a solution better or worse?

Global search vs local searchs

- Like greedy algorithms, backtracking algorithms break the massive global search for a solution into a series of simpler local searches for part of the solution (Which edge do we take first? Then second? …)
- While greedy algorithms guess the “best” local choice and only consider this possibility, in backtracking we use exhaustive search to try out all combinations of local decisions

Global search vs local searchs

- However, we can often use the **constraints** of the problem to **prune** cases that are dead ends. Applying this recursively, we get a substantial savings over exhaustive search.
- This might take a long time to do. What are some other ideas in general?

Search and optimization problems

- Many problems involve finding the best solution from among a large space of possibilities
- Instance: what does the input look like?
- Solution format: what does an output look like?
- Constraints: what properties must a solution have?
- Objective function: what makes a solution better or worse?

Search and optimization problems

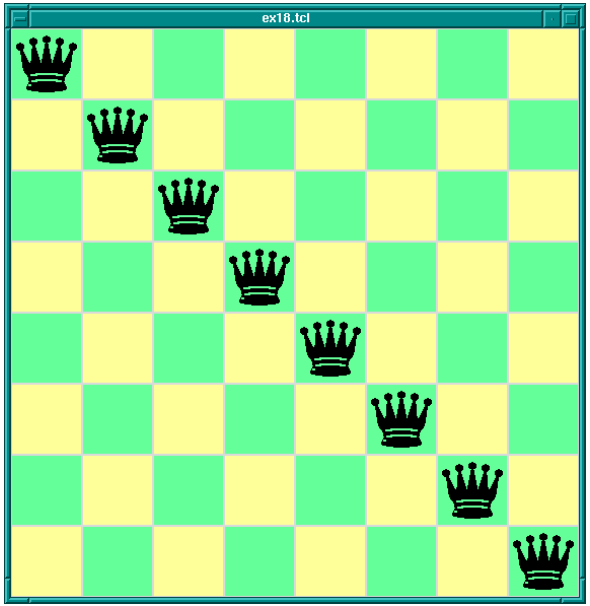
- Many problems involve finding the best solution from among a large space of possibilities
- Instance: what does the input look like?
- Solution format: exhaustive search generally loops through all possibilities that satisfy the solution format
- Constraints: backtracking uses the constraint to eliminate impossible solutions (usually early on in their development)
- Objective function: what makes a solution better or worse?

Backtracking example

8 QUEENS PUZZLE

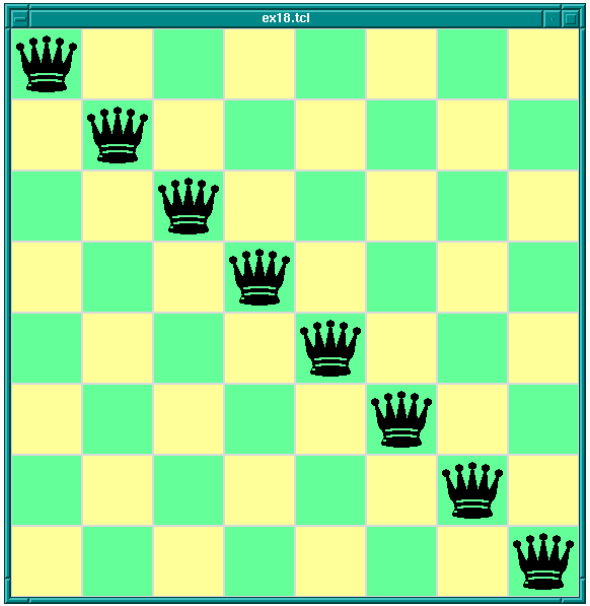
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- Solution format: what does an output look like?
- Constraints: what properties must a solution have?



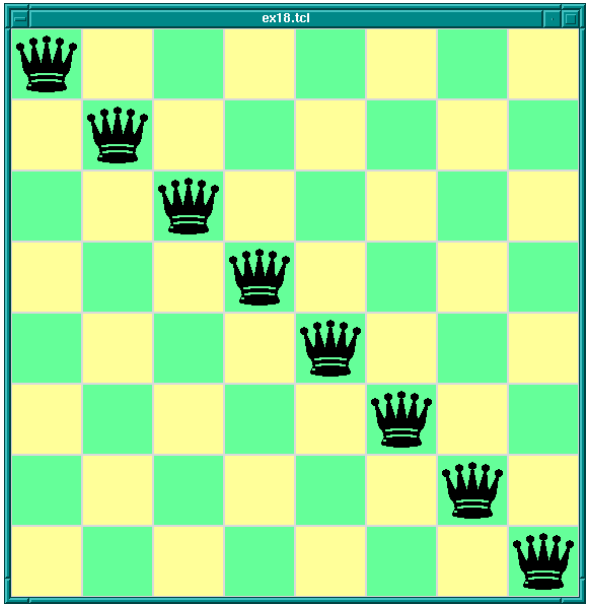
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- Solution format: Arrangement of 8 queens on chessboard
- Constraints: No two queens are attacking



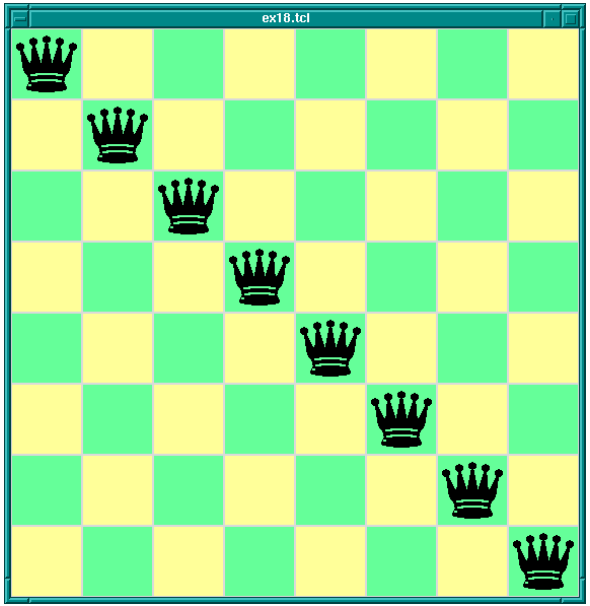
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- Brute force
  - Put all possible arrangements of queens on the chessboard
  - $8^{64} = 281474976710656$



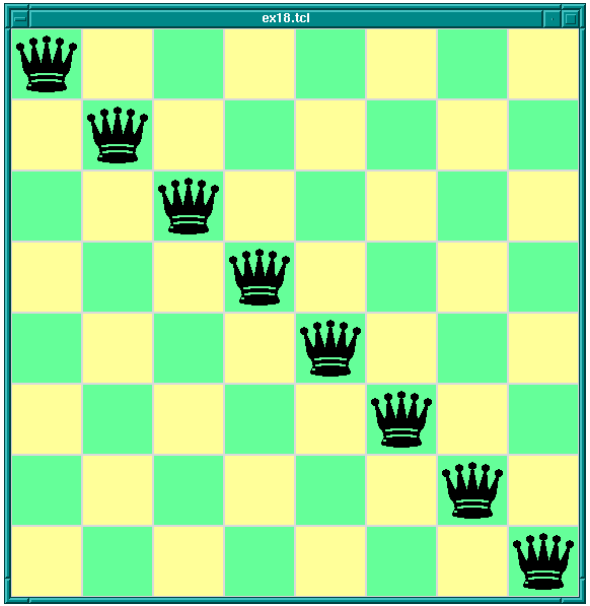
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- Brute force (better)
  - Put all possible arrangements of queens on the chessboard such that no two queens occupy the *same square*
  - $\binom{64}{8} = 4426165368$



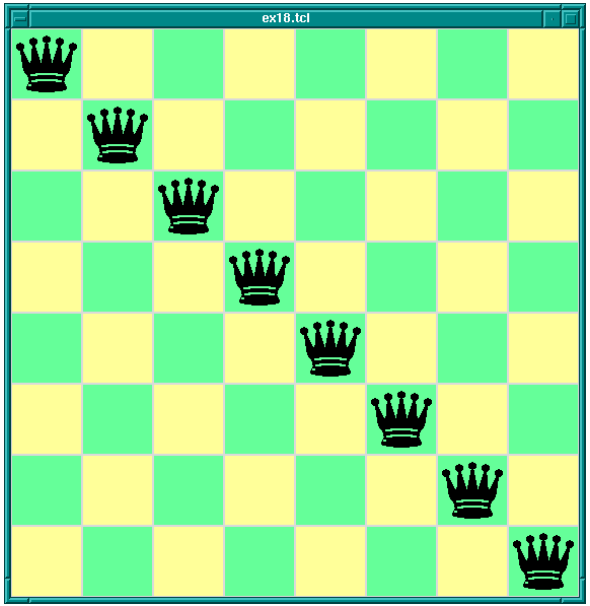
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- Brute force (better++)
  - Put all possible arrangements of queens on the chessboard such that no two queens occupy the *same row*
  - $8^8 = 16777216$



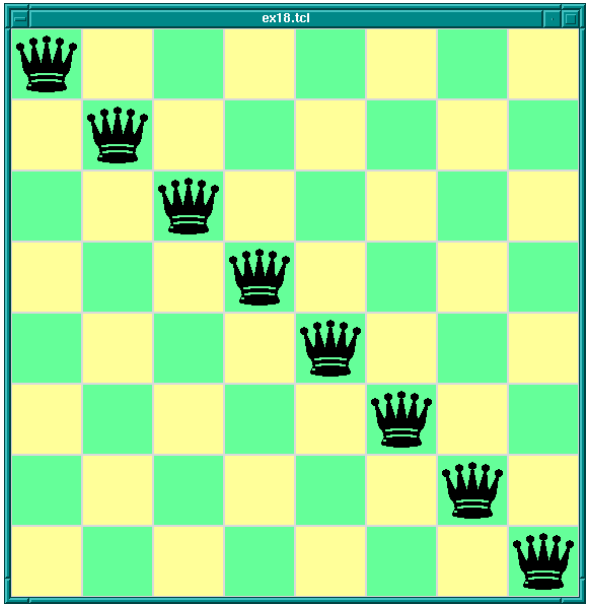
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- Brute force (better+++)
  - Put all possible arrangements of queens on the chessboard such that no two queens occupy the *same row or column*
  - $8! = 40320$



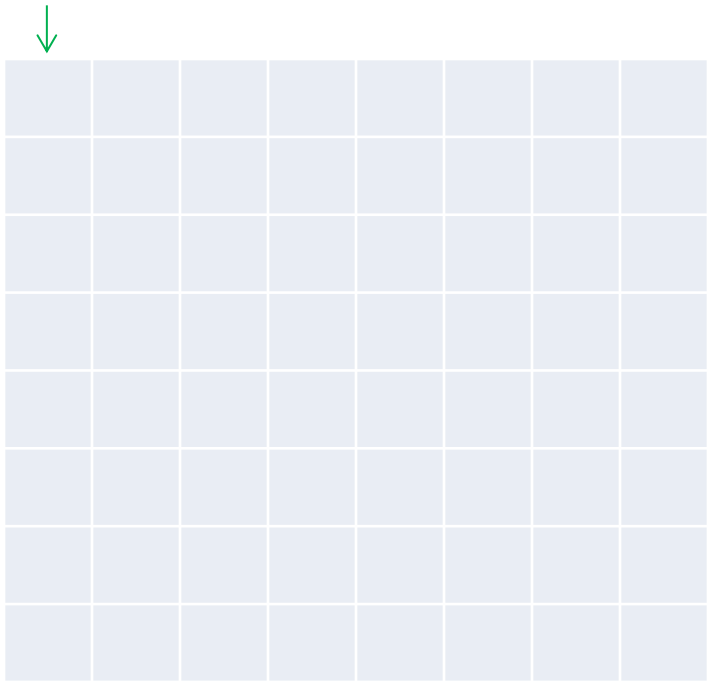
8 queens puzzle

- Put 8 queens on a chessboard such that no two are attacking
- The backtracking method attacks the problem by considering one row at a time, eliminating or “pruning” possible non-solution (dead-end) chessboard positions early in their construction
  - Using this method, there are only 15720 possible queen placements

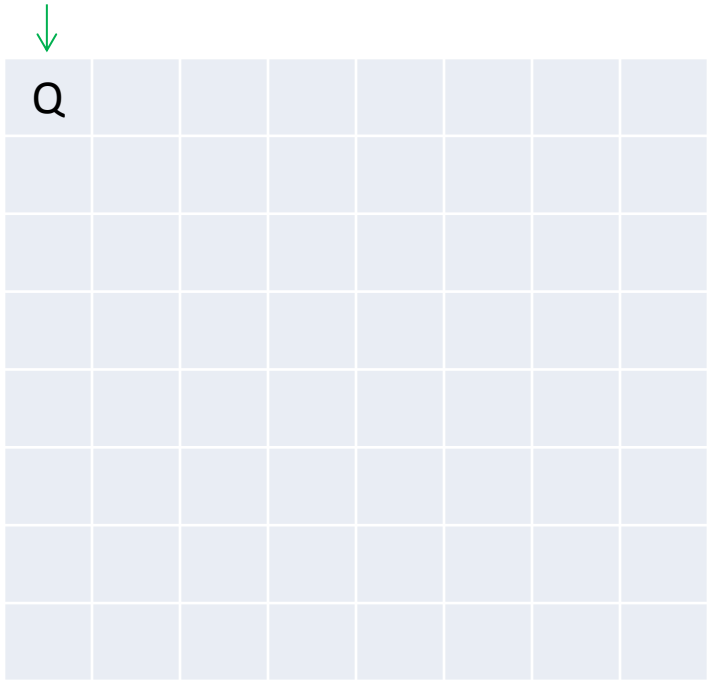


8 queens puzzle

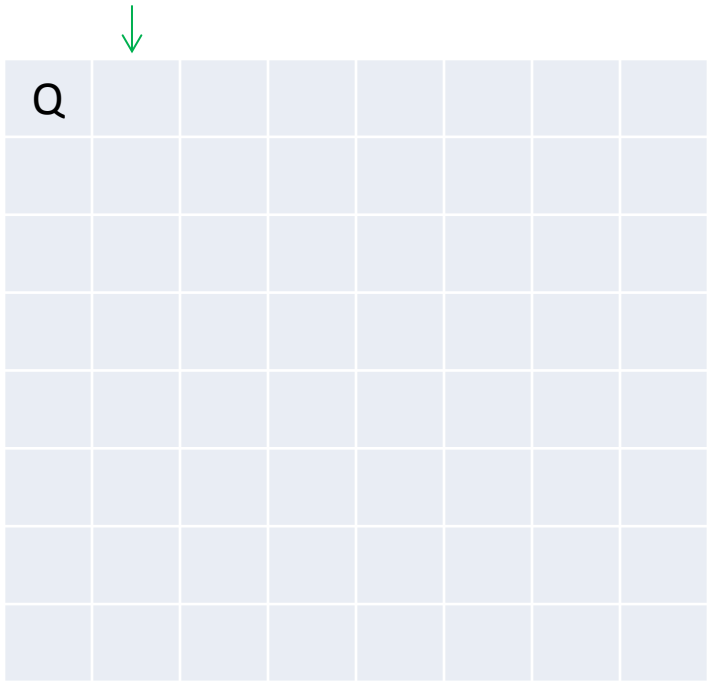
8 queens puzzle



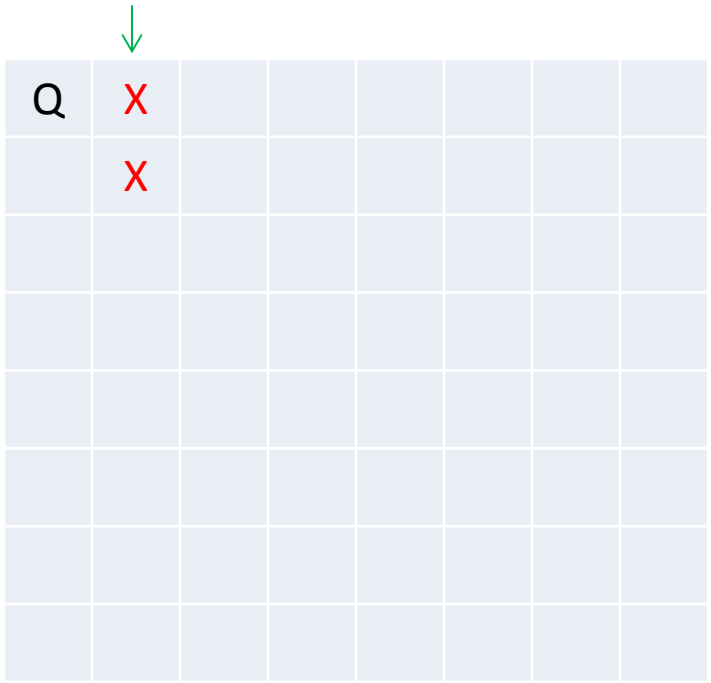
8 queens puzzle



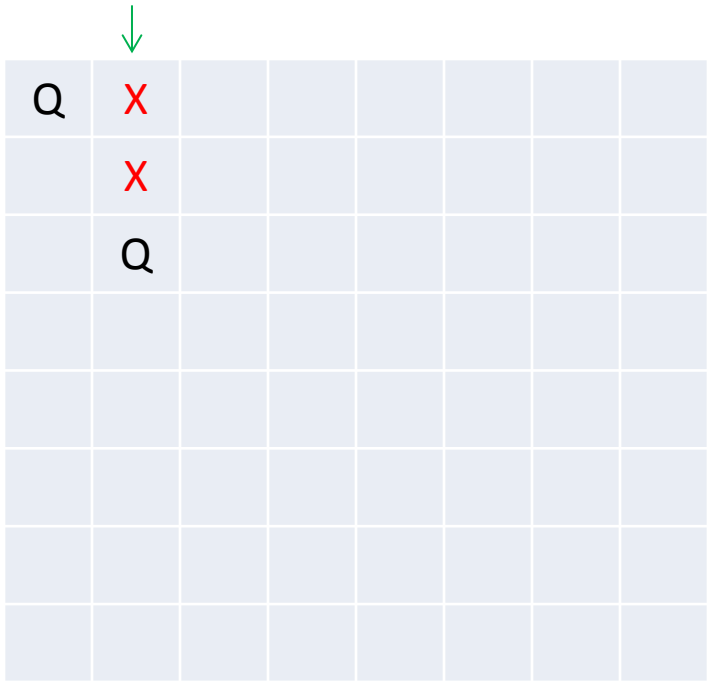
8 queens puzzle



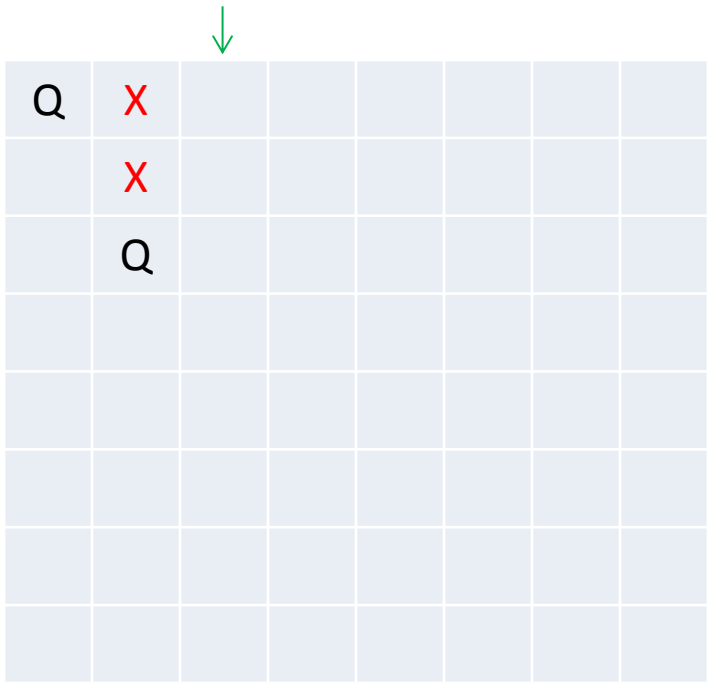
8 queens puzzle



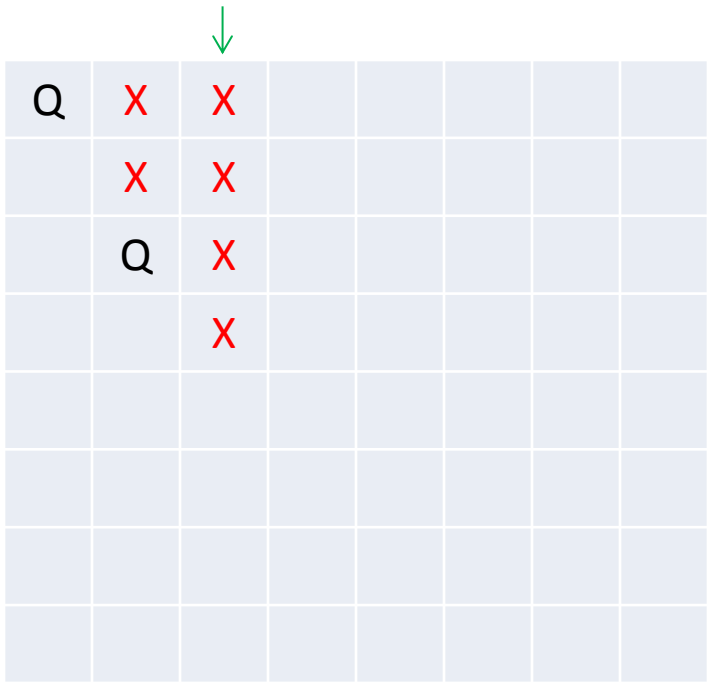
8 queens puzzle



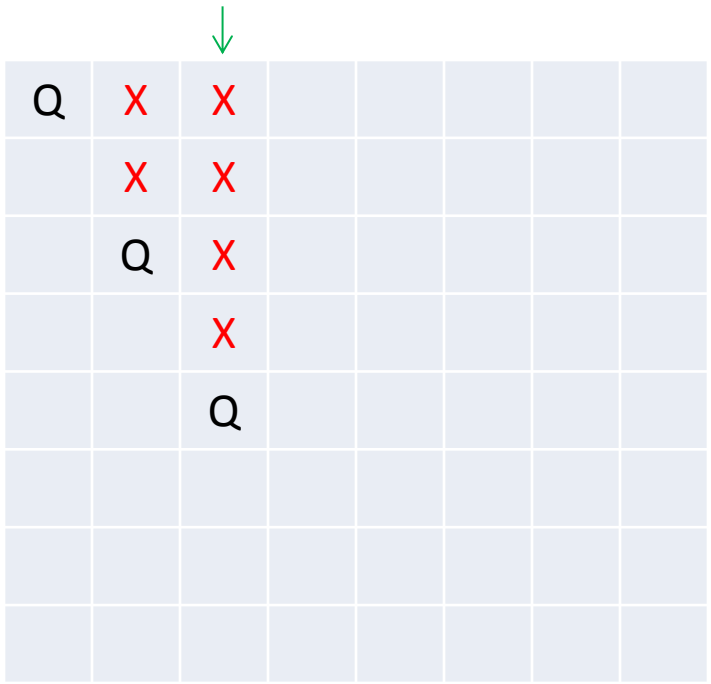
8 queens puzzle



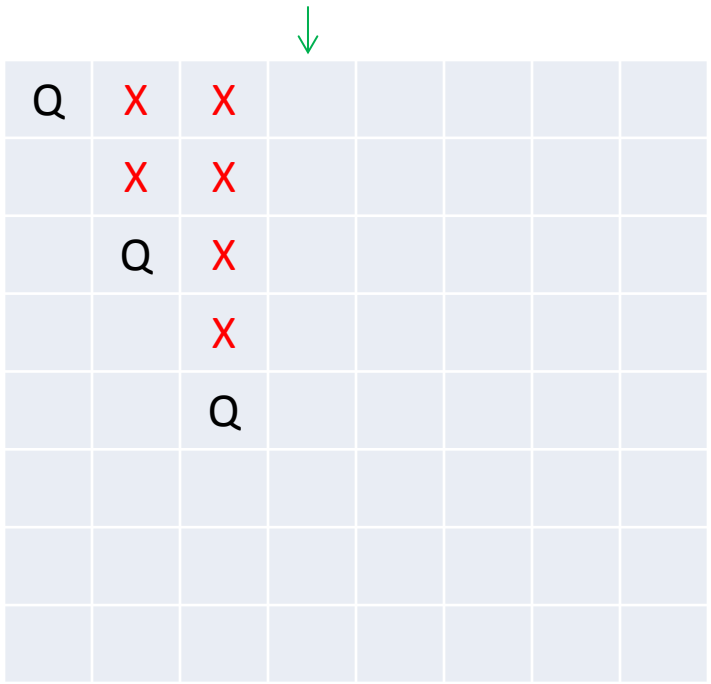
8 queens puzzle



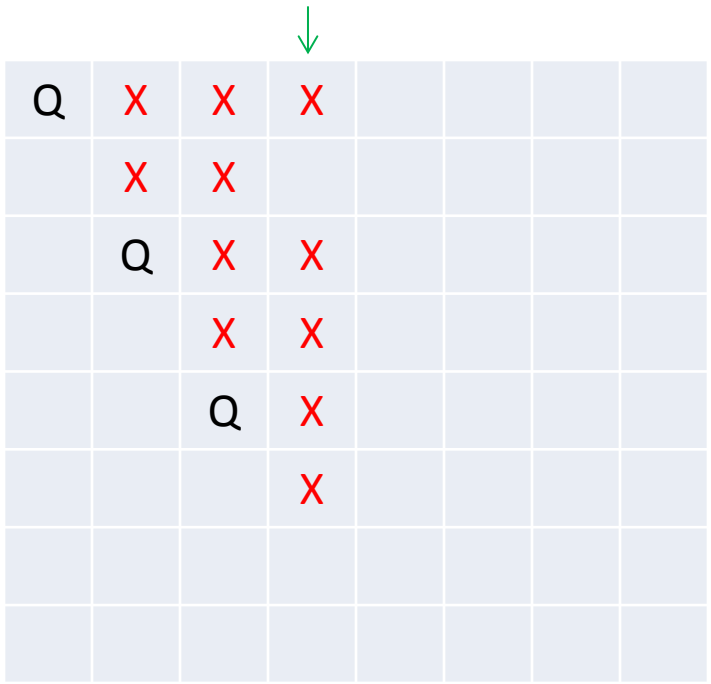
8 queens puzzle



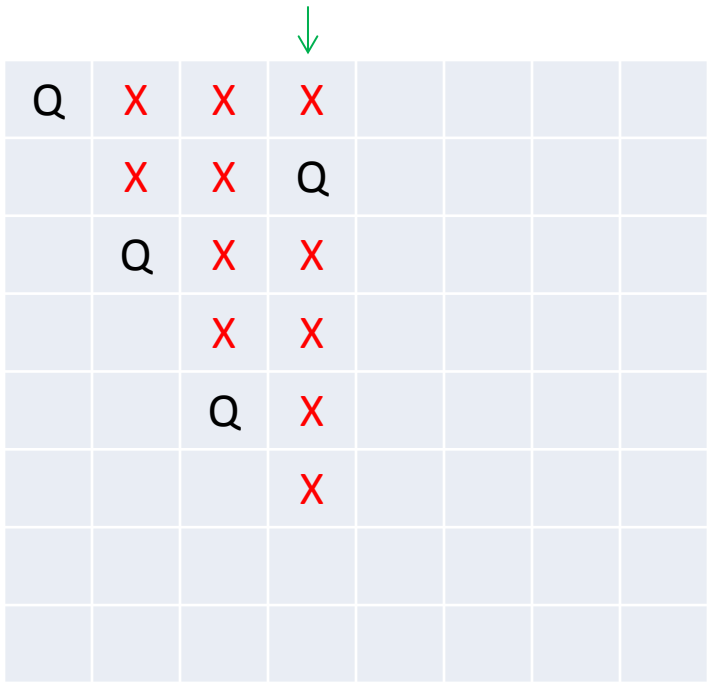
8 queens puzzle



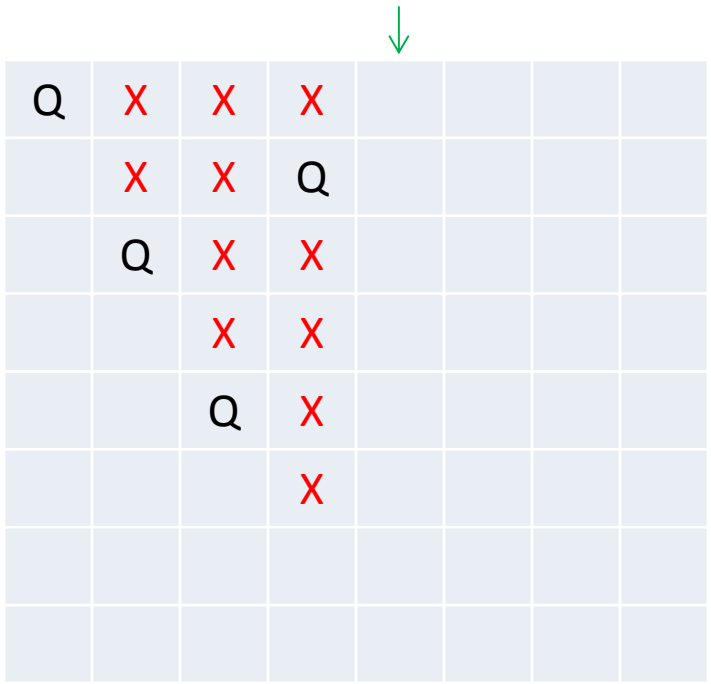
8 queens puzzle



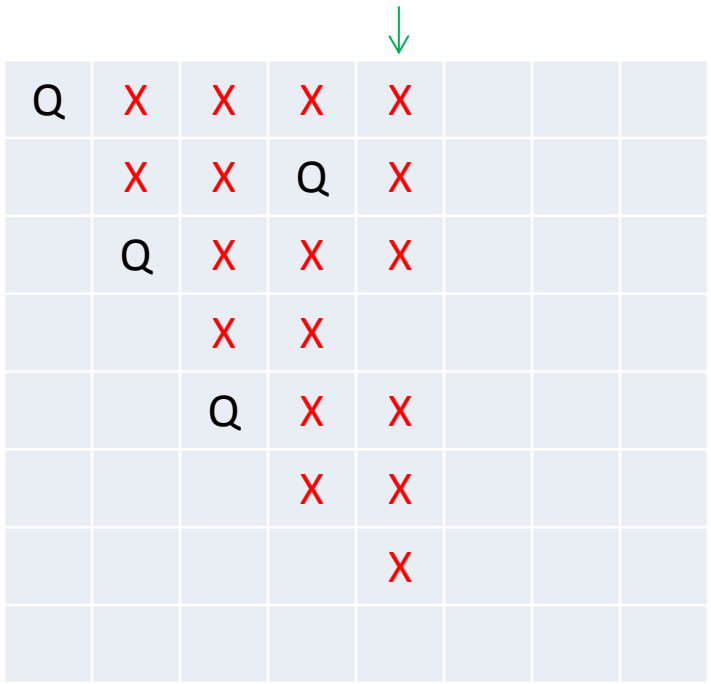
8 queens puzzle



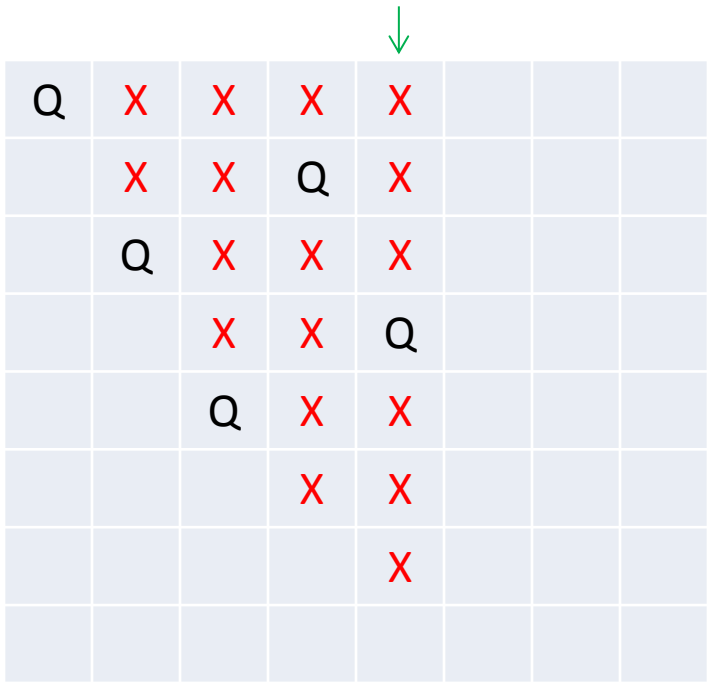
8 queens puzzle



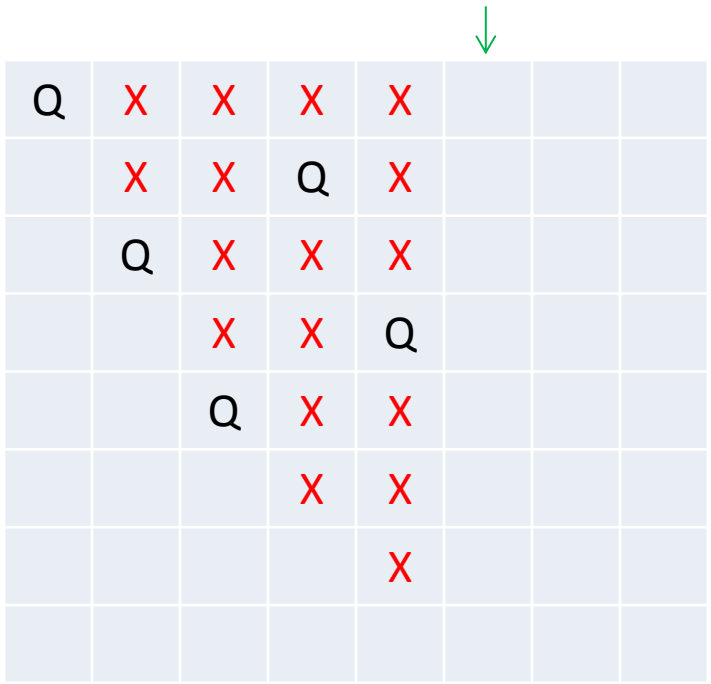
8 queens puzzle



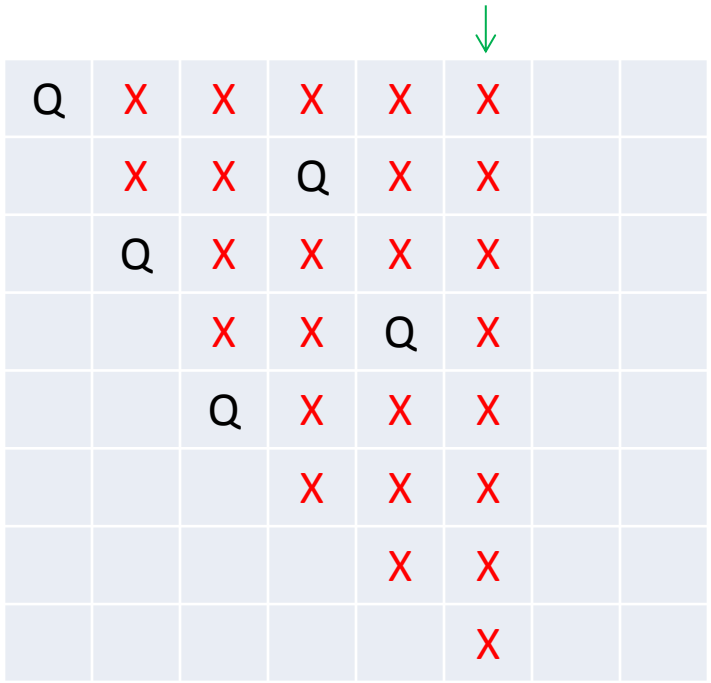
8 queens puzzle



8 queens puzzle

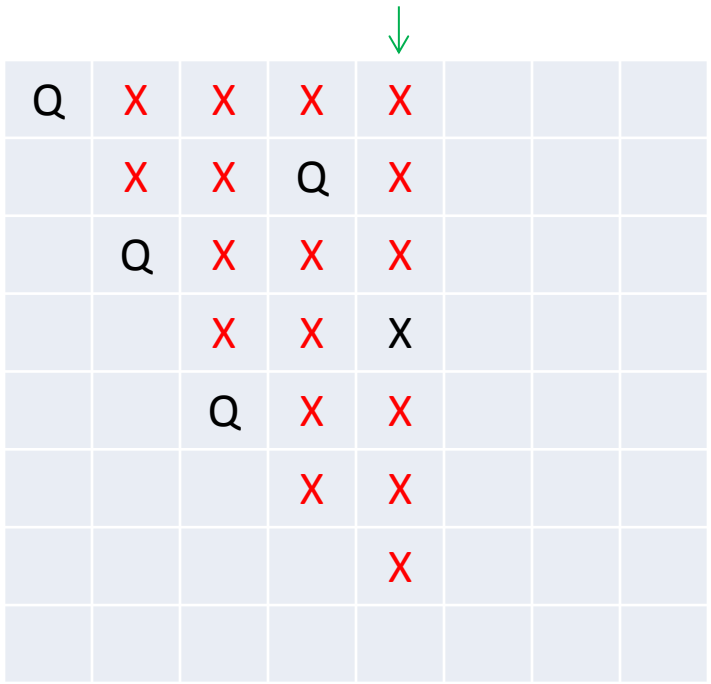


8 queens puzzle



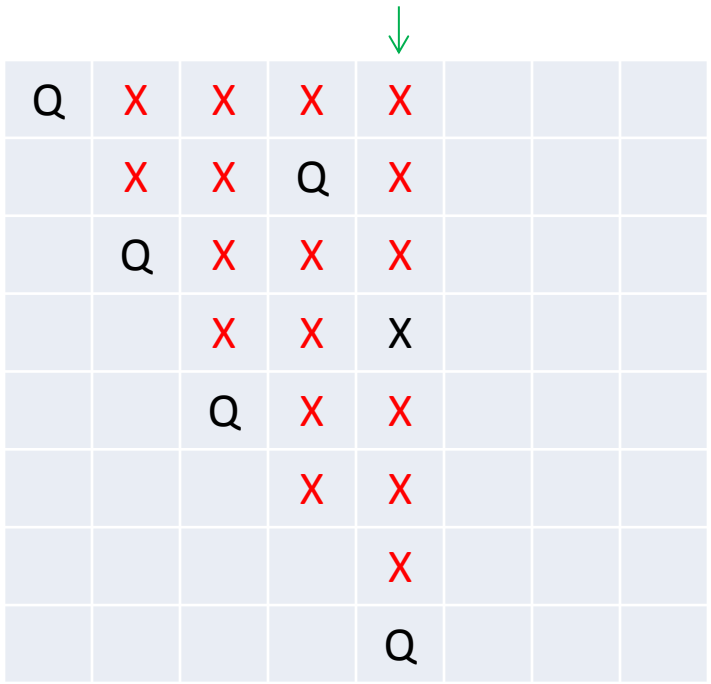
Cannot possibly be completed to a valid solution

8 queens puzzle

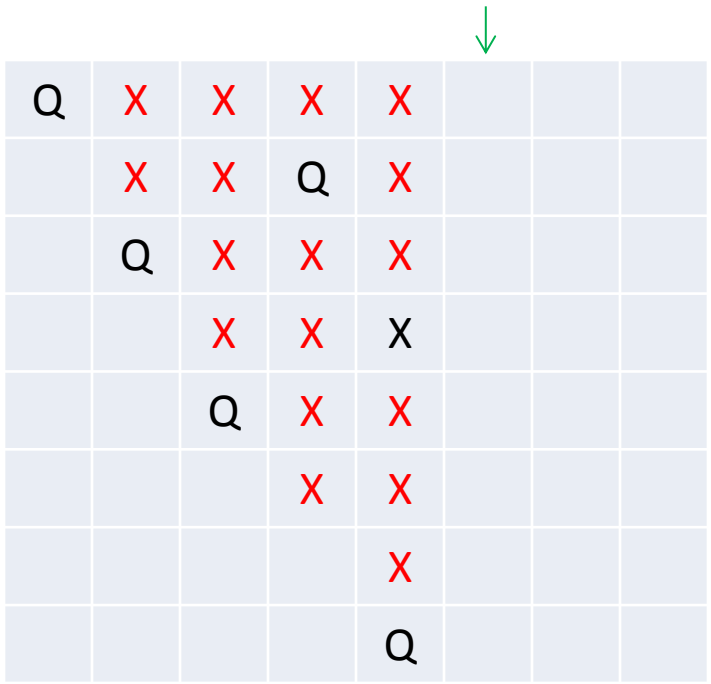


Backtrack

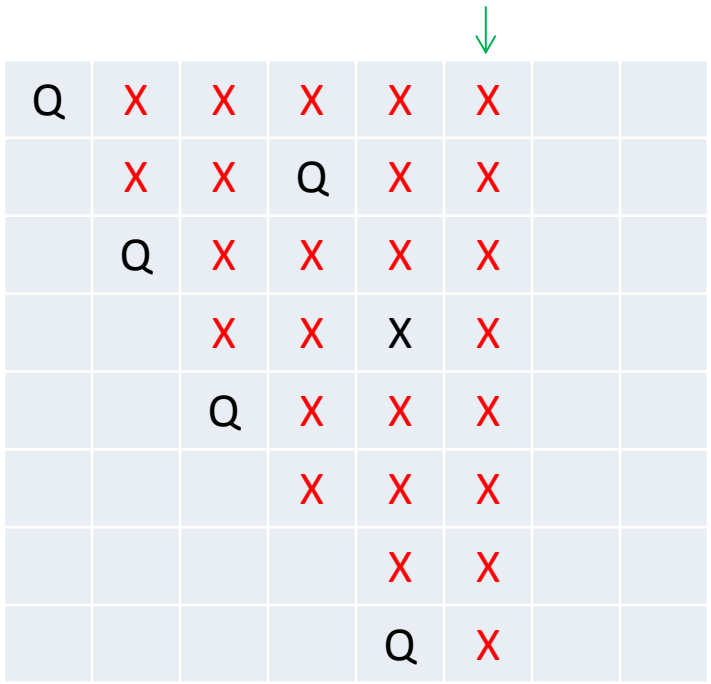
8 queens puzzle



8 queens puzzle

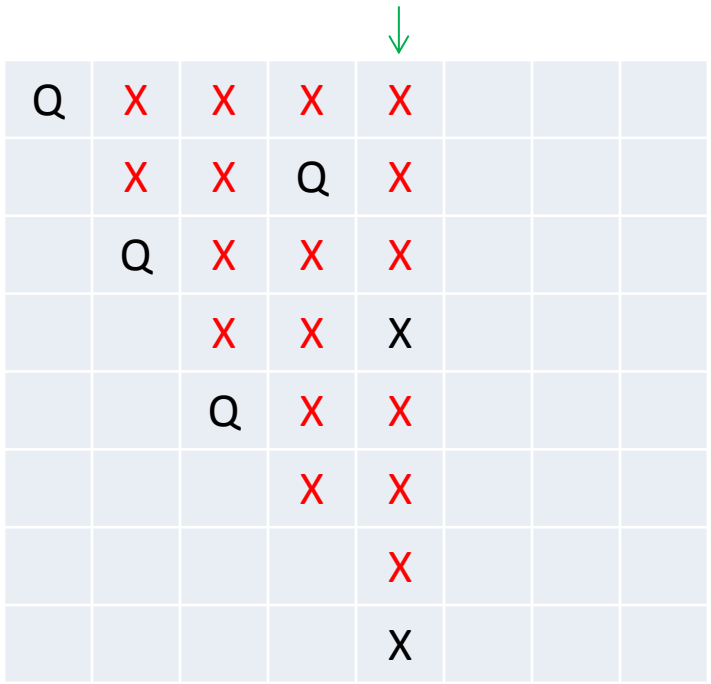


8 queens puzzle



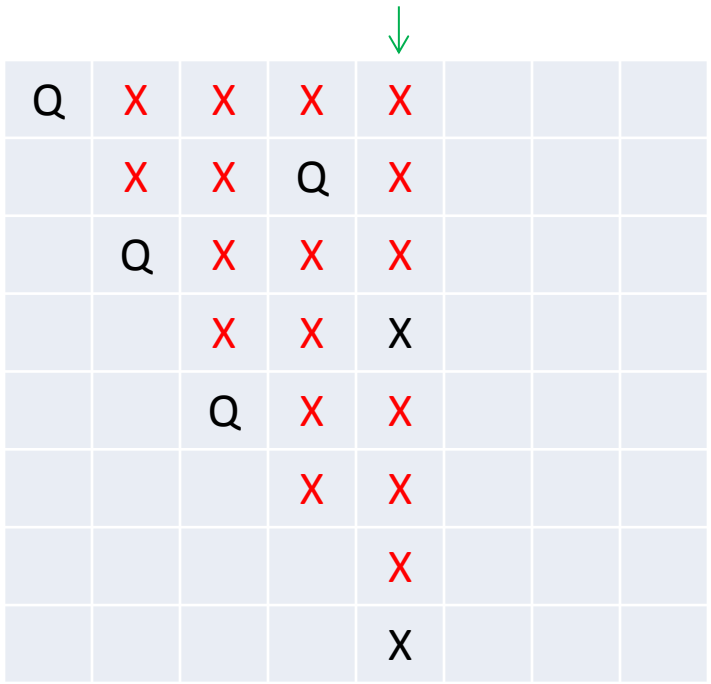
Cannot possibly be completed to a valid solution

8 queens puzzle



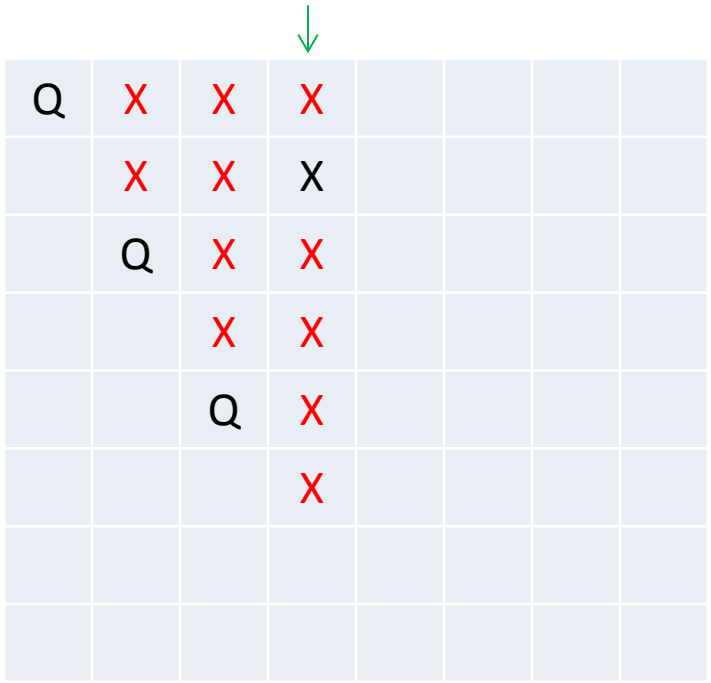
Backtrack

8 queens puzzle



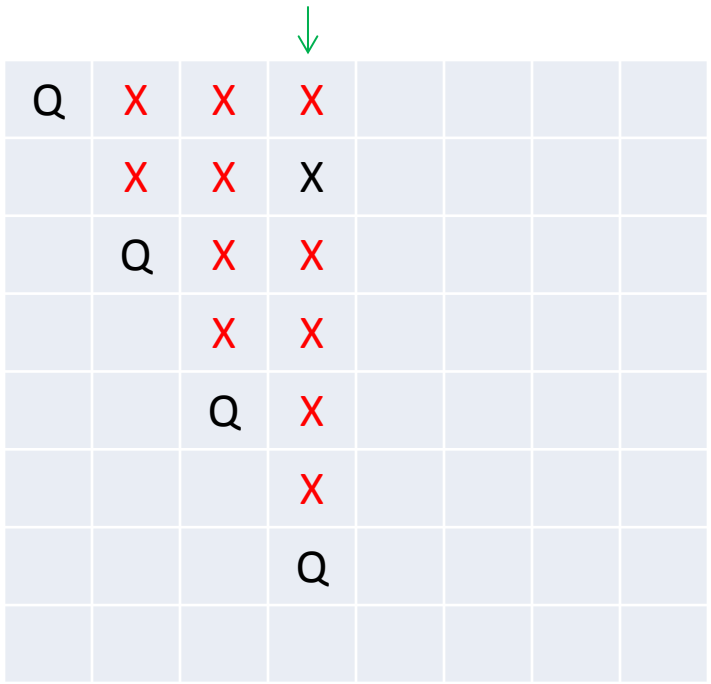
Cannot possibly be completed to a valid solution

8 queens puzzle

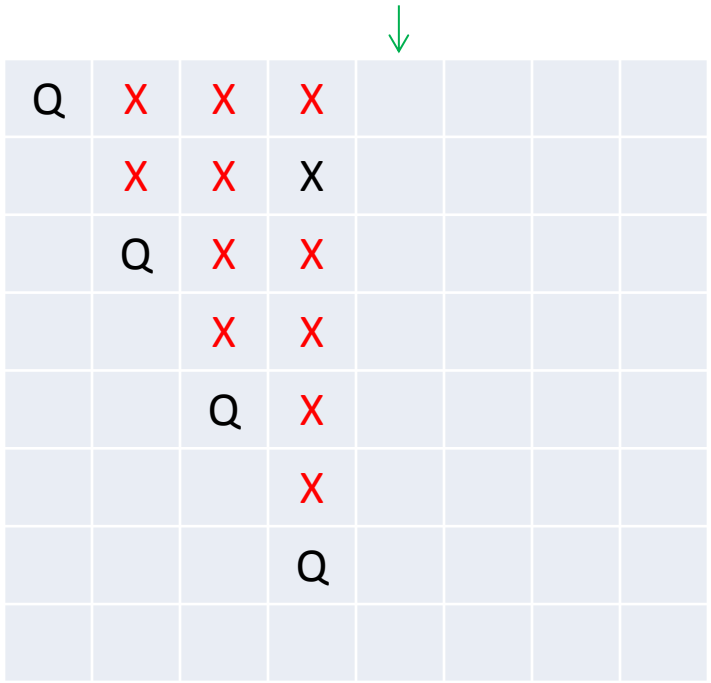


Backtrack

8 queens puzzle



8 queens puzzle



And so on

https://youtu.be/V4qSux-M8N4

Backtracking example

SUDOKU

Search and optimization problems

- Many problems involve finding the best solution from among a large space of possibilities
- Instance: what does the input look like?
- Solution format: what does an output look like?
- Constraints: what properties must a solution have?
- Objective function: what makes a solution better or worse?

Sudoku

- Instance: a partially filled in puzzle
- Solution format: a grid with all squares filled with the numbers 1 through 9
- Constraint: there can be no repeats of numbers in each sub-square, row or column
- Objective: find a solution with the constraint

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

Sudoku

- Backtracking strategy: fill in the first available cell with the least possible number and recurse until there is a cell that cannot be filled in
- Go back to the last decision point and try the next biggest possible number if available

MAXIMAL INDEPENDENT SET

Maximal independent set

- Given a graph with nodes representing people, and there is an edge between A in B if A and B are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are connected with an edge.

Maximal independent set

- Given a graph with nodes representing people, and there is an edge between A in B if A and B are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are connected with an edge.
- Instance: what does the input look like?
- Solution format: what does an output look like?
- Constraints: what properties must a solution have?
- Objective function: what makes a solution better or worse?

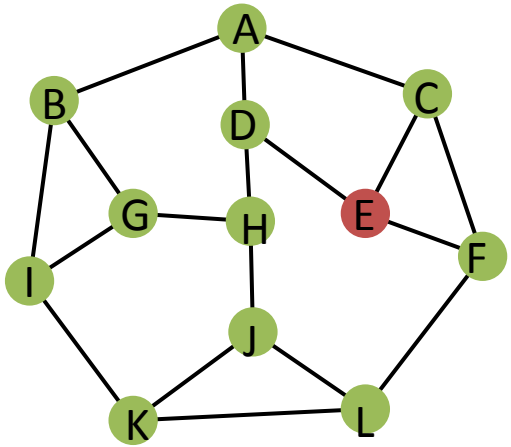
Maximal independent set

- Given a graph with nodes representing people, and there is an edge between A in B if A and B are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are connected with an edge.
- Instance: an undirected graph
- Solution format: set of vertices S
- Constraints: $u, v \in S \Rightarrow (u, v) \notin E$
- Objective function: maximize |S|

Maximal independent set

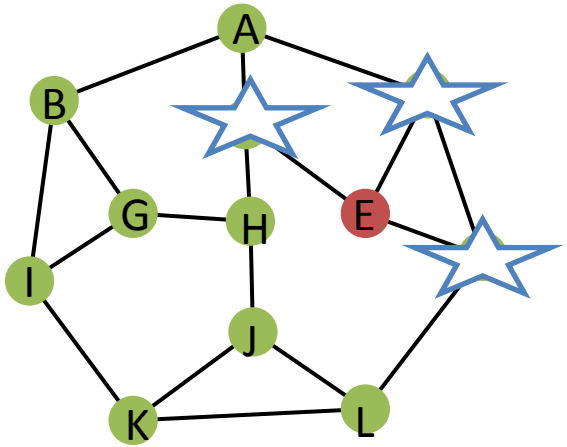
- Greedy approaches?
- One may be tempted to choose the person with the fewest enemies, remove all of his enemies and recurse on the remaining graph
- This is fast, however it does not always find the best solution

Example



Greedy: all degree 3, pick any, say E

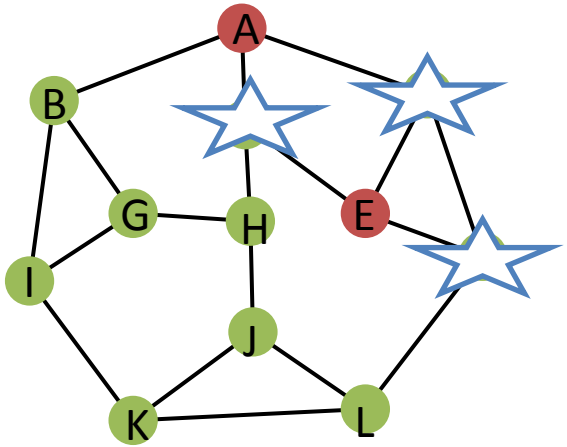
Example



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

Example

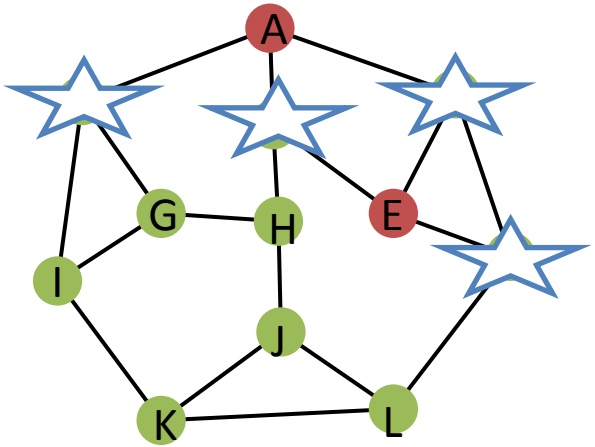


Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

A now is lowest degree

Example



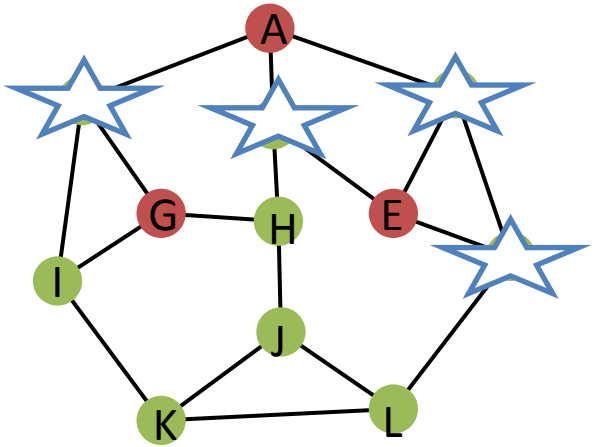
Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

A now is lowest degree

Neighbors (enemies) of A forced out of set

Example



Greedy: all degree 3, pick any, say E

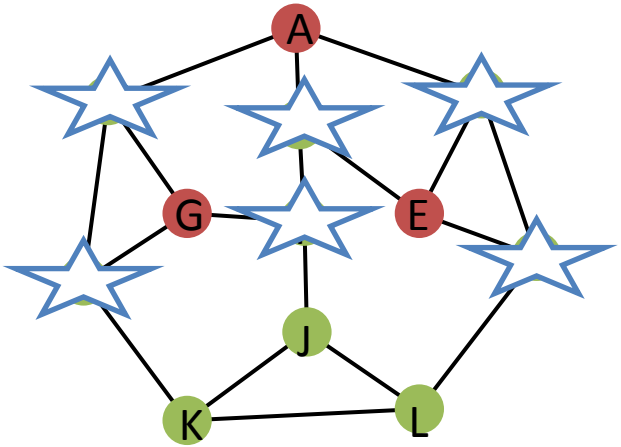
Neighbors (enemies) of E forced out of set

A now is lowest degree

Neighbors (enemies) of A forced out of set

Multiple degree 2, pick any, say G

Example



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

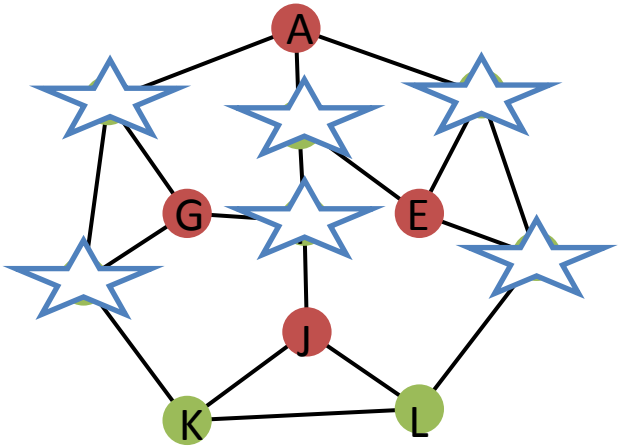
A now is lowest degree

Neighbors (enemies) of A forced out of set

Multiple degree 2, pick any, say G

Neighbors (enemies) of G forced out of set

Example



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

A now is lowest degree

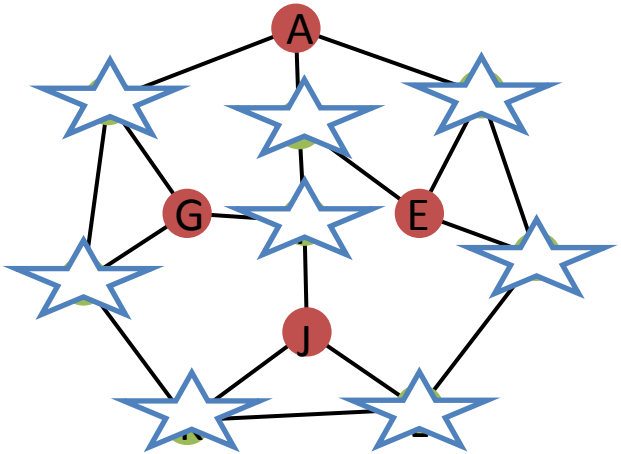
Neighbors (enemies) of A forced out of set

Multiple degree 2, pick any, say G

Neighbors (enemies) of G forced out of set

All degree 2, pick any, say J

Example



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

A now is lowest degree

Neighbors (enemies) of A forced out of set

Multiple degree 2, pick any, say G

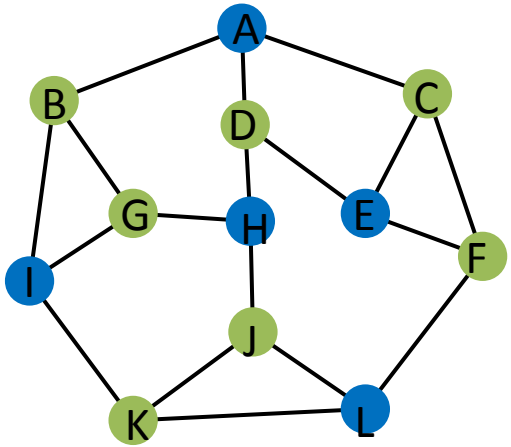
Neighbors (enemies) of G forced out of set

All degree 2, pick any, say J

Neighbors (enemies) of J forced out of set

Solution found by greedy is size 4

Example



Better solution: size 5

Maximal independent set

- Given a graph with nodes representing people, and there is an edge between A in B if A and B are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are connected with an edge.
- Instance: an undirected graph
- Solution format: set of vertices S
- Constraints: $u, v \in S \Rightarrow (u, v) \notin E$
- Objective function: maximize |S|

Maximal independent set

- Given a graph with nodes representing people, and there is an edge between A in B if A and B are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are connected with an edge.
- What is the solution space?
- How much is exhaustive search?
- What are the constraints?

- What is the objective?

Maximal independent set

- Given a graph with nodes representing people, and there is an edge between A in B if A and B are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are connected with an edge.
- What is the solution space? all subsets S of V
- How much is exhaustive search? $2^{|V|}$
- What are the constraints? for each edge e={u,v}, if u is in S, v is not in S
- What is the objective? maximize |S|

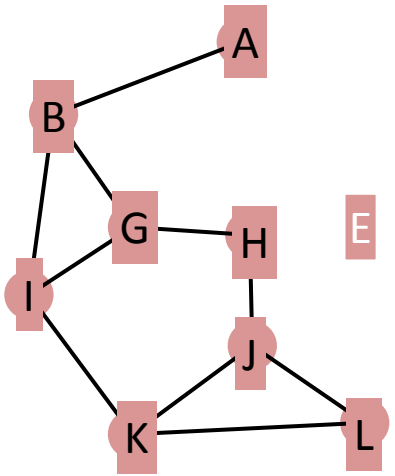
Maximal independent set

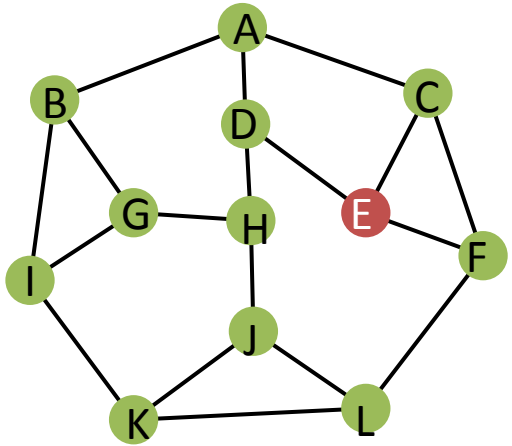
- Backtracking: Do exhaustive search locally, but use constraints to simplify problem as we go
- What is a local decision? Do we pick vertex E or not?
- What are the possible answers to this decision? Yes or no
- How do the answers affect the problem to be solved in the future?
  - If we pick E, then recurse on subgraph $Gr - E \cup \{E\text{'s neighbors}\}$ (and add 1)
  - If we do not pick E, then recurse on subgraph $Gr - E$

Example

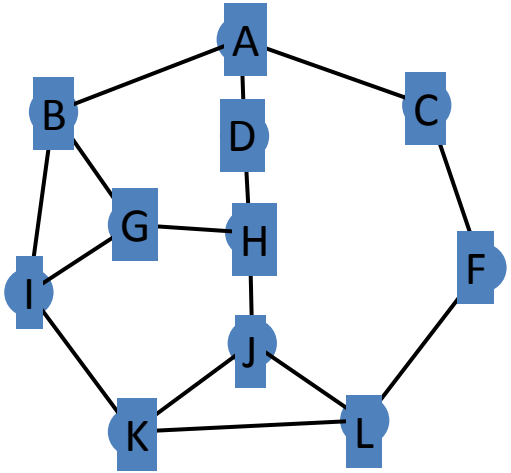
Local decision: do we pick vertex E or not?

Yes





No



Recurse on subgraph $Gr - E \cup \{E\text{'s neighbors}\}$ (and add 1 for E)

Recurse on subgraph $Gr - E$

Case analysis as recursion

MIS1(G= (V,E))

IF |V|=0 return the empty set

Pick vertex v:

S_1 := MIS1(G-v-N(v)) Case 1: optimal solution with v

S_2 := MIS1(G-v) Case 2: optimal solution without v

IF $|S_2| > |S_1|$ return S_2, else return S_1

Correctness

- Strong induction:
- n=0. MIS1 correctly returns empty set.
- Otherwise, use strong induction:
  - S_1 is maximal independent set containing v
  - S_2 is maximal independent set not containing v
  - Better of two is maximal independent set in G

Time analysis

MIS1(G= (V,E)) $T(n)$

IF $|V|$=0 return the empty set

Pick vertex v:

S_1 := MIS1(G-v-N(v)) worst case $T(n-1)$

S_2 := MIS1(G-v) $T(n-1)$

IF $|S_2| > |S_1|$ return S_2, else return S_1

$O(1)$

$T(n) \leq 2T(n-1) + O(1)$ reduce and conquer

$T(n) \leq O(2^n)$

Maximal independent set

- What is the worse case for MIS1?

Maximal independent set

- What is the worse case for MIS1?
  - An empty graph with no edges, i.e., the whole graph is an independent set
  - But then we should just return all vertices without trying cases
  - More generally, if a vertex has no neighbors, the case when we include it v+MIS(G-v) is always better than the case when we do not include it, MIS(G-v)

Getting rid of that worst case

MIS2(G= (V,E))

IF |V|=0 return the empty set

Pick vertex v:

S_1 := MIS2(G-v-N(v))

IF deg(v) = 0 return S_1

S_2 := MIS2(G-v)

IF $|S_2| > |S_1|$ return S_2, else return S_1

Correctness: if deg(v) =0, then $|S_2| < |S_1|$, so we'd return S_1 anyway. So, does same thing as MIS1

Maximal independent set

- What is the worse case for MIS2?

Maximal independent set

- What is the worse case for MIS2?
  - One line. If we always pick the end, then we recurse on one line of size n-1 and one of size n-2

$T(n) = T(n-1) + T(n-2) + poly(n)$
$T(n) = O(Fib(n)) = O(2^{.7n})$
Still exponential but for medium sized n, makes huge difference
n=80: 2^{56} = minute of computer time, 2^{80} = 16 million minutes

Maximal independent set

- Can we do better?
- In the example, we argued that we should add vertices of degree 1 as well
- Modify-the-solution proof:
  - If v has one neighbor u, let S_2 be the largest independent set with v not in S_2. Let S'= S_2 - {u}+{v}. S' is an independent set, is at least as big as S_2, and contains v. Thus, S_1 is at least as big as S', which is at least as big as S_2. So don't bother computing S_2 in this case.

Improved algorithm

MIS3(G= (V,E))

IF |V|=0 return the empty set

Pick vertex v:

S_1 := MIS3(G-v-N(v))

IF deg(v) = 0 or 1 return S_1

S_2 := MIS3(G-v)

IF $|S_2| > |S_1|$ return S_2, else return S_1

Correctness: if deg(v) =0 or 1, then $|S_2| \leq |S_1|$, so we'd return S_1 anyway. So, does same thing as MIS1

Time analysis

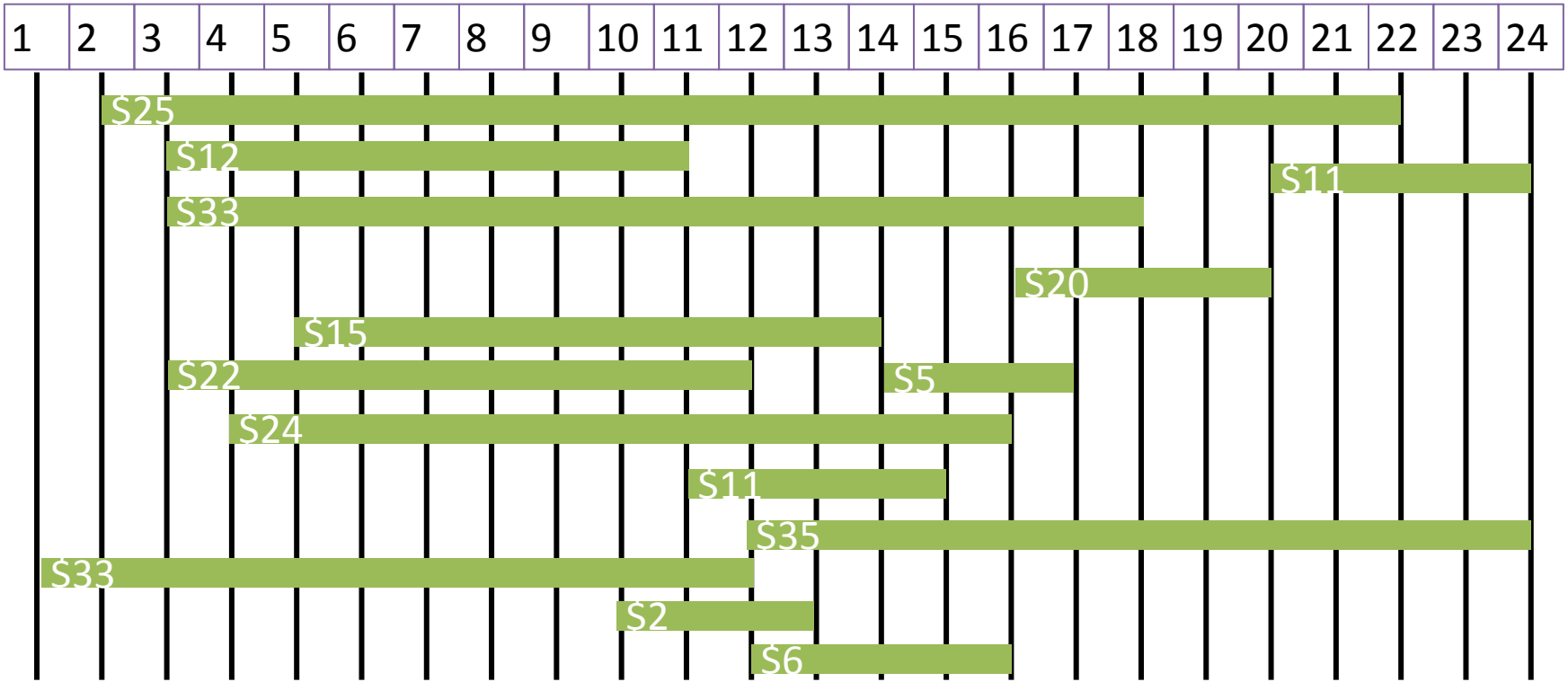
- T(n) is at most T(n-2) + T(n-1) + small amount
- Similar to Fibonacci numbers, but a bit better, about $2^{.6n}$ rather than $2^{.7n}$
- n=80: $2^{.6n}=2^{48}$, less than a second.
- n=100: 2^{60} = 16 minutes, 2^{70}=16,000 minutes
- So while still exponential, big win for moderate n

Is this tight?

- I do not know whether there is any graph where MIS3 is that bad
- Best known MIS algorithm around $2^{n/4}$, by Robson, building on Tarjan and Trojanowski. Does much more elaborate case analysis for small degree vertices.
- Interesting research question: is there a limit to improvements?
- This question leads to Exponential Time Hypothesis, which has interesting ramifications whether true or false

WEIGHTED EVENT SCHEDULING

Weighted event scheduling



Weighted event scheduling

- Instance: what does the input look like?
- Solution format: what does an output look like?
- Constraints: what properties must a solution have?
- Objective function: what makes a solution better or worse?

Weighted event scheduling

- Instance: list of n intervals I= (s,f), with associated values v
- Solution format: subset of intervals
- Constraints: cannot pick intersecting intervals
- Objective function: maximize total value of intervals chosen

Weighted event scheduling

- No known greedy algorithm
  - In fact, Borodin, Nielsen, and Rackoff formally prove no greedy algorithm even approximates
- Brute force?

Weighted event scheduling

- No known greedy algorithm
  - In fact, Borodin, Nielsen, and Rackoff formally prove no greedy algorithm even approximates
- Brute force? 2^n subsets

Backtracking

- Sort events by start time
- Pick first to start. I_1 not necessarily good to include, so we will try both possibilities:
  - Case 1: we exclude I_1, recurse on $[I_2, \ldots, I_n]$
  - Case 2: we include I_1, recurse on the set of all intervals that **do not** conflict with I_1
    - Is there a better way to describe this set?

Backtracking

- Sort events by start time
- Pick first to start. I_1 not necessarily good to include, so we will try both possibilities:
  - Case 1: we exclude I_1, recurse on $[I_2, \ldots, I_n]$
  - Case 2: we include I_1, recurse on the set of all intervals that **do not** conflict with I_1
    - Is there a better way to describe this set? All events that start after I_1 finished, $[I_J, \ldots, I_n]$ for some J

Backtracking

- Sort events by start time
- Pick first to start. I_1 not necessarily good to include, so we will try both possibilities: exclude I_1 and include I_1

BTWES ($I_1 \ldots I_n$): in order of start times

If n=0 return 0

If n=1 return V_1

Exclude:= BTWES($I_2 .. I_n$)

J:=2

Until (J > n or s_J > f_1) do J++

Include:= V_1 +BTWES($I_J .. I_n$)

return Max(Include, Exclude)

Backtracking, runtime

- Sort events by start time
- Pick first to start. I_1 not necessarily good to include, so we will try both possibilities: exclude I_1 and include I_1

BTWES ($I_1 \ldots I_n$): in order of start times $T(n)$

If n=0 return 0

If n=1 return V_1

Exclude:= BTWES($I_2 .. I_n$) $T(n-1)$

J:=2

Until (J > n or s_J > f_1) do J++

Include:= V_1 +BTWES($I_J .. I_n$) $T(n-J)$

return Max(Include, Exclude)

$$T(n) = T(n-1) + T(n-J) + O(\text{poly})$$
$$T(n) = O(2^n)$$

Backtracking, runtime

- $O(2^n)$ worst case time, same as exhaustive search
- We could try to improve or use dynamic programming

Example

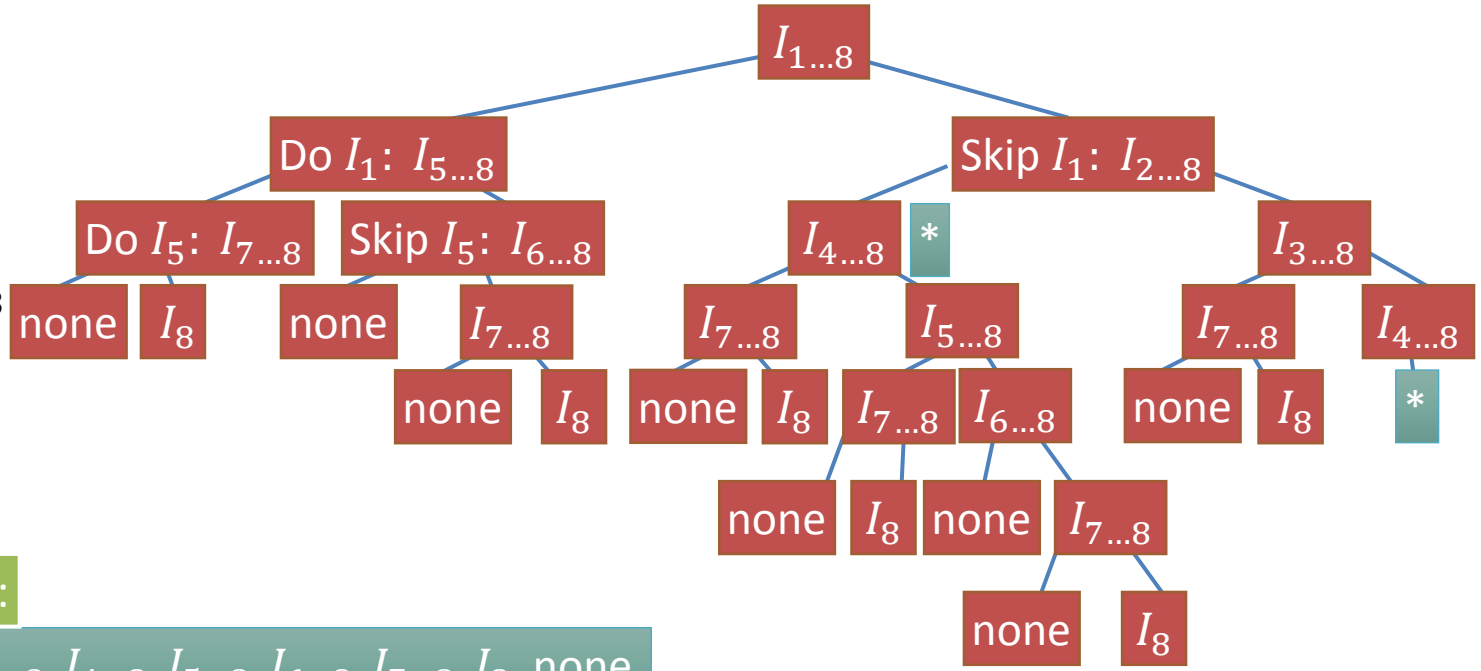
- $I_1 = (1,5), V_1 = 4$
- $I_2 = (2,4), V_2 = 3$
- $I_3 = (3,7), V_3 = 5$
- $I_4 = (4,9), V_4 = 6$
- $I_5 = (5,8), V_5 = 3$
- $I_6 = (6,11), V_6 = 4$
- $I_7 = (9,13), V_7 = 5$
- $I_8 = (10,12), V_8 = 3$

Total number of calls vs number of distinct calls

- We make up to 2^n recursive calls in our algorithm
- But every recursive call has the form $I_J \ldots I_n$
- Thus, there are at most $n + 1$ different calls throughout
- Memorization: Store and reuse the answers, do not recompute

Example

$I_1 = (1,5), V_1 = 4$
$I_2 = (2,4), V_2 = 3$
$I_3 = (3,7), V_3 = 5$
$I_4 = (4,9), V_4 = 6$
$I_5 = (5,8), V_5 = 3$
$I_6 = (6,11), V_6 = 4$
$I_7 = (9,13), V_7 = 5$
$I_8 = (10,12), V_8 = 3$



Distinct calls:

$I_{1\ldots8}, I_{2\ldots8}, I_{3\ldots8}, I_{4\ldots8}, I_{5\ldots8}, I_{6\ldots8}, I_{7\ldots8}, I_8$, none

Characterize calls made

- All of the recursive calls BTWES makes are to arrays of the form $I_{K \ldots n}$ or empty with K=1…n
- So, of the 2^n recursive calls we might make, only $n + 1$ distinct calls are made
- Just like Fibonacci numbers: many calls made exponentially often
- Solution same: create array to store and re-use answers, rather than repeatedly solving them

Dynamic programming steps

- Step 1: Define sub-problems and corresponding array
- Step 2: What are the base cases
- Step 3: Give recursion for sub-problems
- Step 4: Find bottom-up order
- Step 5: What is the final output?
- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

- For analysis:
- Step 7: correctness proof
- Step 8: runtime analysis

Next lecture

- Dynamic programming
  - Reading: Chapter 6