Exhaustive Search, Backtracking, and Dynamic Programming

Miles

Consider the 0-1 Knapsack problem

Given:

- n items: each with weight w_i and value v_i
- knapsack max weight: W

Choose k of the items (at most one of each item) to maximize the total value while keeping their weights ≤ W

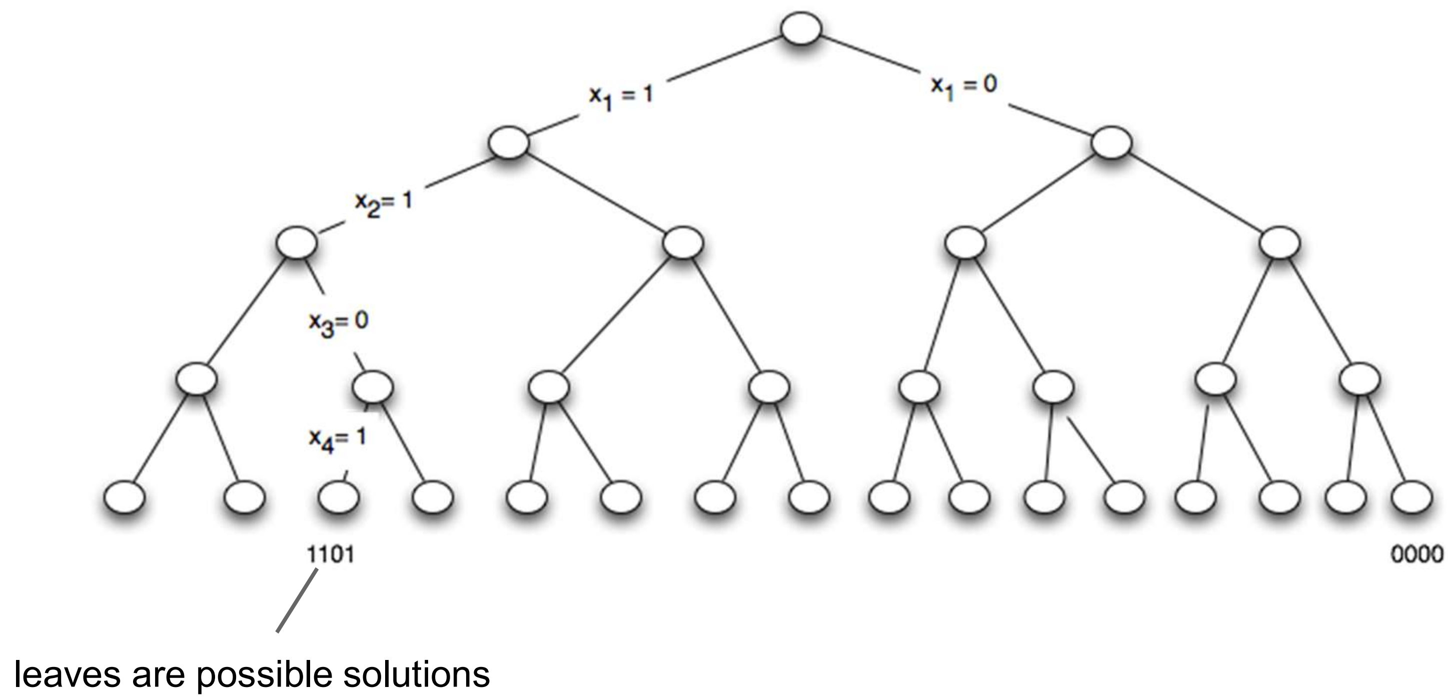
Representing the search space

Let x_i = 1 if item i is chosen, 0 otherwise

Let $X = x_1 x_2 \ldots x_n$

X is a bit pattern

A tree representing the solution space



Naïve approach

Try all (2^n) possible solutions:

1111

1110

1101

1100

…

0000

Smarter approach

What can be ruled out when?

items $(w_i, v_i) = ((3, 5), (8, 5), (10, 13), (6, 6))$

$W = 20$

Smarter approach

Abandon a subtree if it can’t participate in an optimal solution

Smarter approach

If at a node, and weights of items chosen so far is:

equal W: go directly down rightmost path

> W: rule out subtree and go back to parent

Keep track of best total value so far. If current value + possible values down the tree < best, rule out subtree and go back to parent.

Dynamic Programming

Take advantage of subproblems of the same form as original.

For each item:

- Use it (assume is part of the final solution)
- Lose it (assume is *not* part of the final solution)

Dynamic Programming

S((5, 3), (5, 8), (10, 13), (6, 6)), W=20)

Use it: 3 items left, and 17 remaining capacity:
  value = 5 + S((5, 8), (10, 13), (6, 6), W=17)
lose it: 3 items left and 20 remaining capacity:
  value = 0 + S((5, 8), (10, 13), (6, 6), W=20)
So, take the minimum

Problem: multiple recomputations

S((10, 13), (6, 6), W=15) is computed more than once (as are the recursive sub-calls)

Solution: store results in a 2D best_value table
Dimension 1: # of items to consider (1..N)
Dimension 2: knapsack capacity (0..W)

Filling in the table from the bottom-up

((5, 3), (5, 8), (10, 13), (6, 6)), W=20

Remaining weight

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1..4	**4**																					
2..4	**3**																					
3,4	**2**																					
4,4	**1**	0	0	0	0	0	0															
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Backtracking

Find all of the 3x3 magic squares:

A 3x3 magic square is a square of the numbers 1,2,3,…,9 such that each number appears exactly once and each row, column, and diagonal sums to the same value.

Backtracking

Find all of the 3x3 magic squares:

How many arrangements would you have to try using exhaustive search?

A	B	C
D	E	F
G	H	I

Backtracking

Find all of the 3x3 magic squares:

How many arrangements would you have to try using exhaustive search?

9!=362,880

A	B	C
D	E	F
G	H	I

Backtracking

Find all of the 3x3 magic squares:

A	B	C
D	E	F
G	H	I

How would we use backtracking?

Let's do some math first to eliminate some possibilities.

The common sum must be (1+2+3+...+9)/3=15.

Backtracking

Find all of the 3x3 magic squares:

A	B	C
D	E	F
G	H	I

So we know that (A+E+I)+(B+E+H)+(C+E+G)+(D+E+F)=4(15)

so 3(E)+A+B+C+D+E+F+G+H+I = 3(E)+3(15)

Therefore 3(E)=15 and E must be 5!!!!

Backtracking

Find all of the 3x3 magic squares:

A	B	C
D	5	F
G	H	I

So now, how many arrangements are there?

Backtracking

Find all of the 3x3 magic squares:

A	B	C
D	5	F
G	H	I

So now, how many arrangements are there?

8! = 40320

Backtracking

Find all of the 3x3 magic squares:

A	B	C
D	5	F
G	H	I

What else do we know?

Backtracking

Find all of the 3x3 magic squares:

A	B	C
D	5	F
G	H	I

What else do we know?

The pairs (1,9), (2,8), (3,7), (4,6) must be opposite each other.

Backtracking

Find all of the 3x3 magic squares:

Up to rotation we could have:

1		
	5	
		9

	1	
	5	
	9	

Backtracking

Find all of the 3x3 magic squares:

Up to rotation we could have:

	1	
	5	
	9	

Backtracking

Find all of the 3x3 magic squares:

Up to rotation we could have:

	1	
	5	
	9	

Dynamic Programming

Come up with a recursive algorithm to solve the problem

Create a table to store the results

Fill in table:

- Top-down (using recursion, but checking table first)
- Bottom-up (iterating through the table)

But we said knapsack was infeasible!

It is. If W doubles in length, that causes an exponential increase in time (because when W double in length, the number itself increases exponentially).

This approach only works for small W.

Dynamic Programming

Like divide and conquer, DP solves a family of subproblems.

Instead of dividing the input in half, we traverse the input and keep track of the solution of the subproblems as we go along. Then we use the previous solution to compute the current solution.

Biggest Difference

Describe an algorithm that takes a list of positive integers $a_1, \ldots a_n$ as an input and outputs the maximum difference $a_i - a_j$ with $i<j$

Biggest Difference

Descrive an algorithm that takes a list of positive integers a_1,…a_n as an input and outputs the maximum difference a_i-a_j with i<j

Naïve solution:

max:=0

for i from 1 to n-1

for j from i+1 to n

if max<a_i-a_j then max:=a_i-a_j

return max

Dynamic Programming (Biggest difference)

What we are going to do is loop from j=2 to n and keep track of a value bd[j] which will be the biggest difference within the list

a_1,…,a_j

If we know bd[j], can we use it to find bd[j+1]?

Dynamic Programming (Biggest difference)

What we are going to do is loop from $j=2$ to n and keep track of a value bd[j] which will be the biggest difference within the list

$a_1,\ldots,a_j$

If we know bd[j], can we use it to find bd[j+1]?

Along with bd[j], let's keep track of another value:
m[j]=max value in the list $a_1,\ldots,a_j$

Dynamic Programming (Biggest difference)

Given bd[j], m[j] and a_(j+1)

can you tell me what is bd[j+1]?

Dynamic Programming (Biggest difference)

Given bd[j], m[j] and a_(j+1)

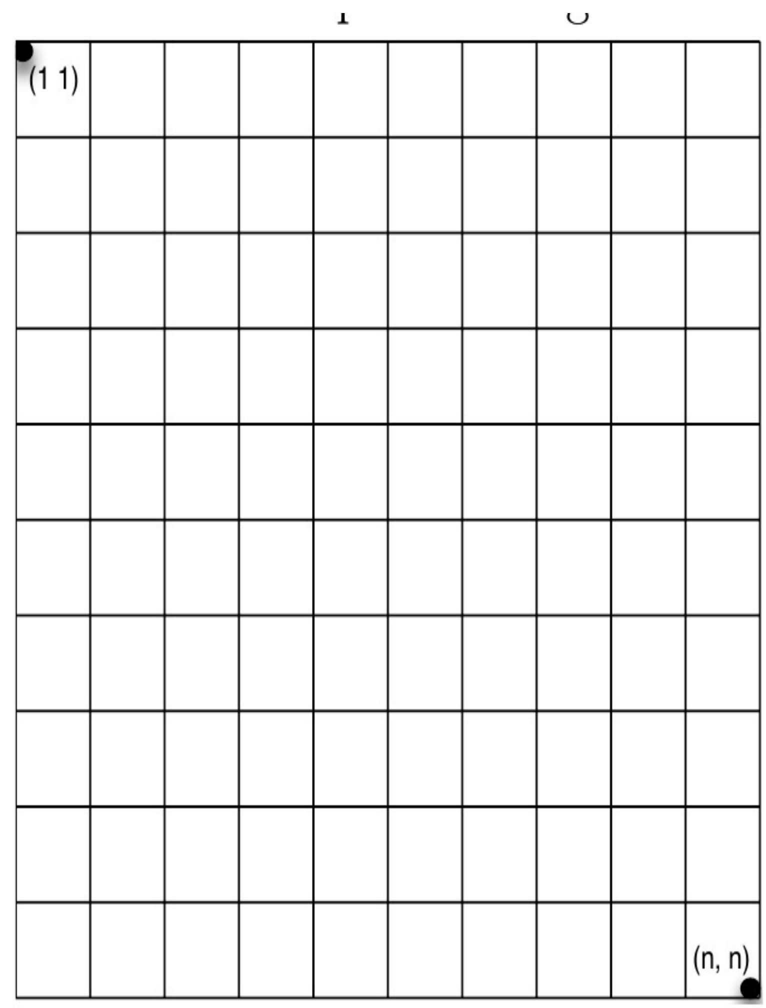
can you tell me what is bd[j+1]?

bd[j+1]=max(bd[j],m[j]-a_(j+1))

nxn grid

Given an nxn grid, Determine how many ways there are to travel from the top left to the bottom right.

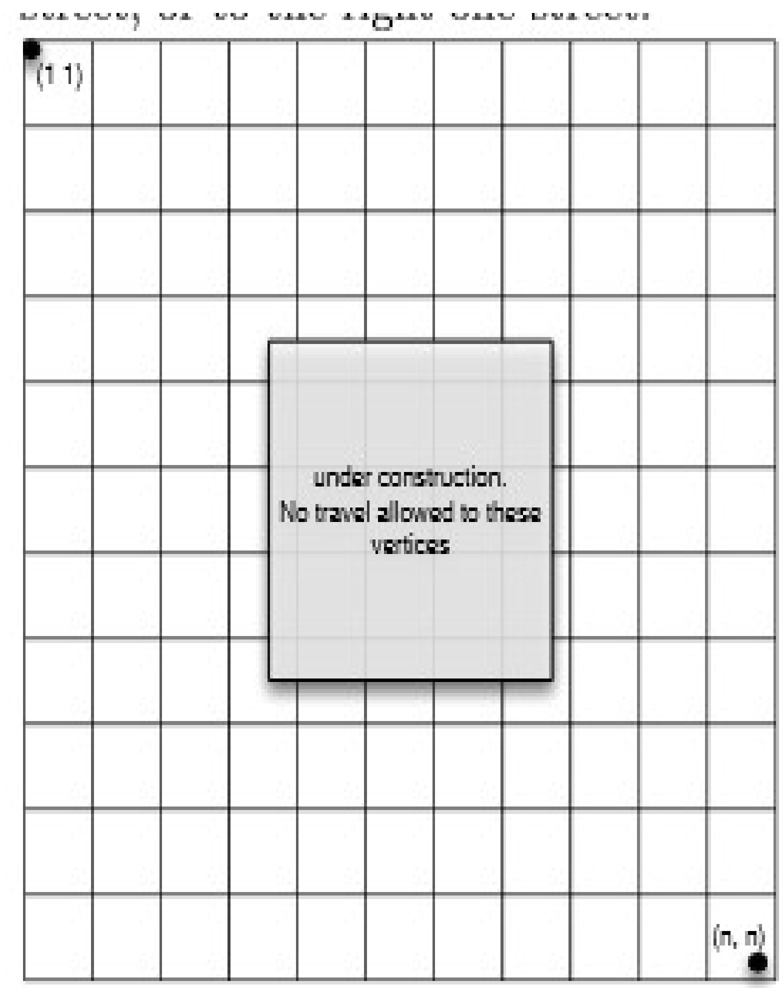
How would you figure this out?



nxn grid

Given an nxn grid with part of it off limits,
Determine how many ways there are to travel from the top left to the bottom right.

How would you figure this out?



The icosian Game

Suppose you are on a road trip and you want to visit 20 locations arranged in the picture. Is there a way to visit every location exactly once and then return to your home?