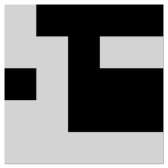
Search

- a challenging topic
- <u>all</u> of you can do it
  - you have laid the foundation for this all term
- <u>all</u> of you will struggle with it some
  - the world has more difficult design problems than easy ones
- approach this topic knowing that you can do it

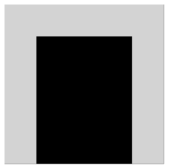
Search

- You know how to:
  - represent information of different forms
    - we have been doing information that “seemed clearly real”
  - design functions that do backtracking search on a tree
  - design functions that use generative recursion
- One big difference today is we will represent “made up” information
  - a synthetic tree
  - generative recursion
  - backtracking search

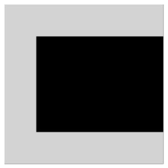
Solving Mazes



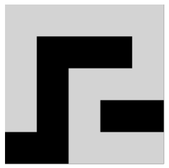
1



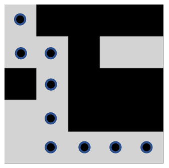
2



3

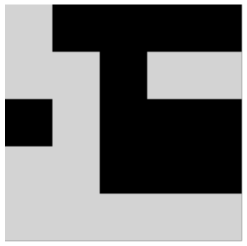


4



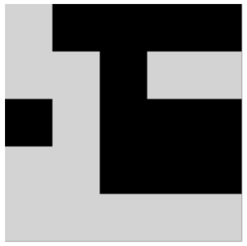
1

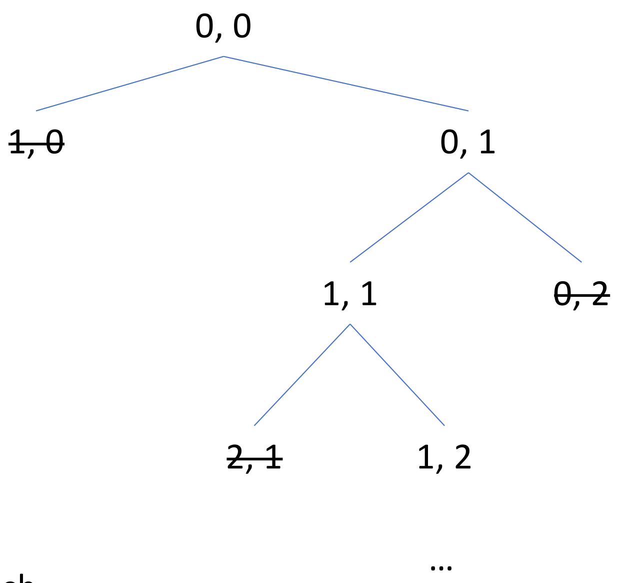
This maze is solveable, so will eventually reach 4, 4. Yay!



This maze is solveable, so will eventually reach 4, 4. Yay!

Tree of x,y positions moving through this maze



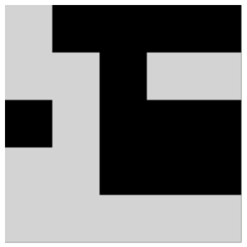


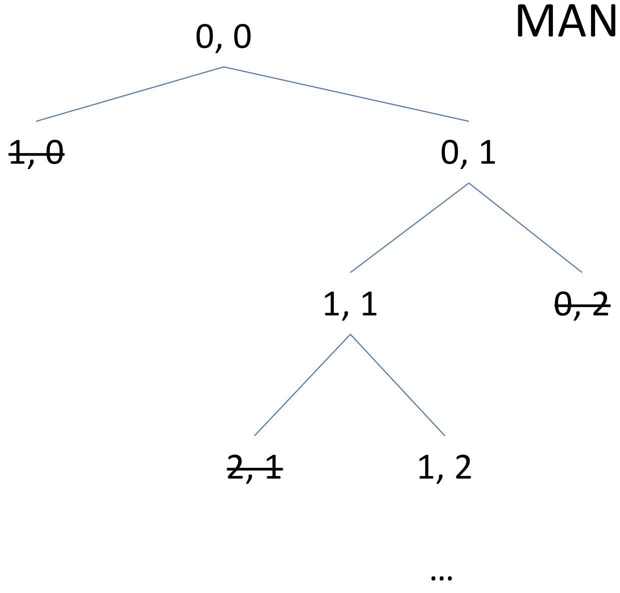
In this module, we restrict the problem so that at each step it is only possible to move right (x+1) or down (y+1).

But sometimes either of those may be invalid because they run into a wall or off the edge of the maze.

Do not assume each position can have only one valid next position. In general it is an arbitrary-arity tree.

MANDATORY search tree





3 KEY questions:

What is changing search state? current x,y position

How to form next next search search states? down and right, UNLESS hit wall or edges

How to tell when done? solved if reach lower right corner
can also run out of moves

```
;; genrec
(define (genrec d)
  (cond [(trivial? d) (trivial-answer d)]
        [else
         [... d
              (genrec (next-problem d))]]))


;; arb-tree  (of Pos) -> Pos and (listof Pos)
(define (fn-for-pos p)
  (... p
       (fn-for-lop (POS-SUBS p)))) ;pos-subs doesn't really exist!!!


(define (fn-for-lop lop)
  (cond [(empty? lop) (...)]
        [else
         (... (fn-for-pos (first lop))
              (fn-for-lop (rest lop)))]))


;; try-catch
     (local [(define try <one-option>)]
       (if (not (false? try))
           try
           <other-option>))
```

```
(define (fn-for-pos p)
  (cond [(trivial? p) (trivial-answer p)]
        [else
         (... p
              (fn-for-lop (next-problem p)))]))


(define (fn-for-lop lop)
  (cond [(empty? lop) false]
        [else
         (local [(define try (fn-for-pos (first lop)))]
           (if (not (false? try))
               try
               (fn-for-lop (rest lop))))]))
```

```
(define (fn-for-pos p)
  (cond [(solved? p) true]
        [else
         (fn-for-lop (valid-next-positions p))]))


(define (fn-for-lop lop)
  (cond [(empty? lop) false]
        [else
         (local [(define try (fn-for-pos (first lop)))]
           (if (not (false? try))
               try
               (fn-for-lop (rest lop))))]))
```

```
;; genrec
(define (genrec d)
  (cond [(trivial? d) (trivial-answer d)]
        [else
         [... d
              (genrec (next-problem d))]]))
```

genrec

```
;; arb-tree  (of Pos) -> Pos and (listof Pos)
(define (fn-for-pos p)
  (... p
       (fn-for-lop (POS-SUBS p)))) ;pos-subs doesn't really exist!!!
```

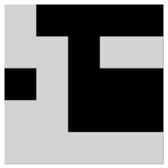
```
(define (fn-for-lop lop)
  (cond [(empty? lop) (...)]
        [else
         (... (fn-for-p (first lop))
              (fn-for-lop (rest lop)))]))
```

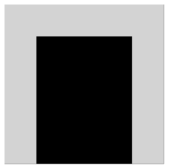
arb-tree

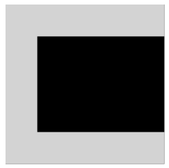
```
;; try-catch
    (local [(define try <one-option>)]
      (if (not (false? <one-option>))
          <one-option>
          <other-option>))
```

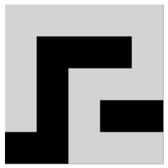
try-catch

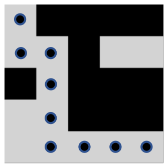
Solving Mazes


1


2


3


4


1