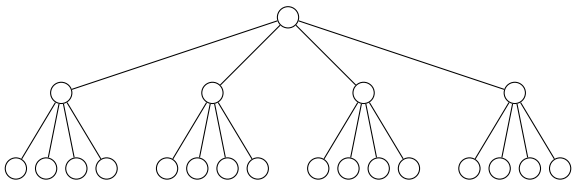
2. a)



c) k^d leaves. Why? Well, there is 1 node if d is 0, and for each increment of d by 1, each leaf spawns k new leaves, so the number of leaves increases by a factor of k. So for any fixed positive k, we can prove this formula is correct by induction on d.

3. Here is the output from `rmaze.py` with input `m10.txt`. For each time-step interval (i.e., between the printing of 2 successive diagrams), explain what search operations occur.

   E.g.: in the first interval, the algorithm picks the left neighbour (how do we know this?) of the starting cell, finds that cell empty, so goes there.

```
X  X  X  X  X  X  X
X        +  !     X
X  X  X  X  X  X  X
```

```
  from the code, we see that the 4 neighboring cells of the starting
  cell have been examined in random order.  if a wall is encountered,
  nothing happens, and another location is examined.
  if the target ! is discovered, the search halts.
  we can see from the next maze output below that the search
  did not halt, so it must be that the first non-wall cell
  that was examined was to the left of the start,
  where the ? is.
```

```
X  X  X  X  X  X  X
X     ?  +  !     X
X  X  X  X  X  X  X
```

```
  again, the location of ? below tells us that we looked left
  before looking right
```

```
X  X  X  X  X  X  X
X  ?     +  !     X
X  X  X  X  X  X  X
```

```
  here there is only 1 (non-wall) way to look, so eventually,
  after hitting a wall an unknown number of times, we looked right
```

```
X  X  X  X  X  X  X
X     ?  +  !     X
X  X  X  X  X  X  X
```

```
  since no ? appears in the diagram below, it must be that
  we are at the start again.  so search continues from there.
```

```
X  X  X  X  X  X  X
X        +  !     X
X  X  X  X  X  X  X

  since there are no further diagrams, we must have gone right
  from the start, discovered the target !, and ended the search

finish at location (1, 4)
```

4. Execution alternates continually between the following two states, call them A and B. This is because the order in which we consider neighbors is fixed: we always try them in the order shown below, which corresponds to left (same row, previous column), right (same row, next column), up (previous row, same column), down (next row, same column). So when we are A we will try to go left and succeed, and go to B. When we are at B, we will try to go left (fail, there is a wall) and then try to go right, and succeed, and be back at A.

```
X  X  X  X  X  X  X
X     ?  +  !     X
X  X  X  X  X  X  X

X  X  X  X  X  X  X
X  ?     +  !     X
X  X  X  X  X  X  X

nbr_offsets = [(0,-1), (0,1), (-1,0), (1,0)]
```

5. The first element of the tuple is the row index, the second element is the column index. So this means *same row, previous column*. Since we are reading our mazes in from left to right, *previous column* means *column immediately to the left*. So keeping the row the same and moving to the previous column corresponds to moving left.

   Again, we read mazes in from top to bottow, so *moving down* would be *keep the column the same, increase the row index*, so (1,0).

6. The first version is iterative depth first search, where the list behaves like a stack: last in, first out. Neighbors are examined in order left, right, top, down. So, after the first four neighbors of the start have been placed in the list, the first neighbor removed will be the last added, namely down. So 1 will be directly below the start. After the cell labelled 1 is examined, its neighbors up left, right, up (already seen, so ignored) and down, so 2 will be the down-neighbor of 1. The next cells added to the list are 2's left- right- up- (ignored, already seen) down- (ignored, a wall) neighbors, so the next cell removed from the list is 2's right-neighbor. From 3, the target is found, so search ends.

   The next version is breadth first search, which is easier to analyze since the order in which cells are removed is the same in which they are added. So the order in which the tilde symbols appear will be the same order in which the dots appear. So, 1,2,3,4 are respectively the left,right,up,down neighbors of the start, 5,6 are the left,up (not right, because we have already seen it) neighbors of 1.

```
X X X X X X X          X X X X X X X
X . . . . . X          X . . . . . X
X . . . . . X          X . 6 3 . . X
X . . + . . X          X 5 1 + 2 . X
X . . 1 . . X          X . . 4 . . X
X . . 2 3 ! X          X . . . . ! X
X X X X X X X          X X X X X X X
```

7. This is now function `wander2` in program `simple/stile/rmaze.py` of the class code repository.

```
def wander(self,psn):
  num_iterations = 0
  while True:
    num_iterations += 1
    if self.char_at(psn) == empt_ch:
      self.mark_location(psn,curr_ch)
    self.showpretty()
    shuffle(nbr_offsets)
    for shift in nbr_offsets:
      new_psn = psn[0]+shift[0],psn[1]+shift[1]
      new_ch = self.char_at(new_psn)
      if new_ch == dest_ch:
        print(num_iterations,'looks')
        return new_psn
      elif new_ch != wall_ch:
        if self.char_at(psn) == curr_ch:
          self.mark_location(psn,empt_ch)
        psn = new_psn
        break
```

8.
```
A     B--C
|     |  |
D     E--F
|     |
G--H--I
```

9. i) `f b e i a c j k g h d`

   ii) `f b a e c i j k g h d`

   iii) The order in which nodes are first seen is `f b e i j k g h d a c`. (the order in which nodes are popped off the stack is `f i j k g h d e a b c`)

10. There is more than one way to do this.

```
4xx  4xx  4xx  _xx  x_x  xx_  xxx  xxx  xxx  _xx  x_x  xx_  xx1
1x_  1_x  _1x  41x  41x  41x  41_  4_1  _41  x41  x41  x41  x4_

xx1  x_1  x1_  x14  x14  x_4  _x4  xx4  xx4  x_4  x4_  x4x  x4x
x_4  xx4  xx4  xx_  x_x  x1x  x1x  _1x  1_x  1xx  1xx  1x_  1_x

x_x  _xx  1xx  1xx  1xx
14x  14x  _4x  4_x  4x_
```

11. a)

```
    213          *23         *25         25*         *35
    4*5          415         431         431         241
 L   U   R     D   R     D   R     L   D     D   R
213 2*3 213   423 2*3   425 2*5   2*5 251   235 3*5
*45 415 45*   *15 415   *31 431   431 43*   *41 241
```

b) Here are the first two levels ...

```
           473
           25*
           186
  L          U          D
 473        47*        473
 2*5        253        256
 186        186        18*
L R U D    L   D      L   U
...
```

12. a) there is a 1-1 correspondence between states and permutations of the 9 tiles, so 9! = 362 880.

b) there is a 1-1 correspondence between solvable states and unsolvable (they have the same intial sequence, with the last two tiles exchanged), so exactly half of all possible states are solvable, so 12!/2 = 239 500 800.

c) about .94 hours. 70600 iterations per second, so (12!/2)/70.6 seconds = 3392 seconds, about 57 minutes.

d) about 4.7 years.

(16!/2)/70.600 seconds =

((16!/2)/70.600)/60 minutes =

(((16!/2)/70.600)/60)/60 hours =

((((16!/2)/70.600)/60)/60)/24 days =

(((((16!/2)/70.600)/60)/60)/24)/365 years

13. (a) up 9, down 7, left 7, right 7

(b) S is not solvable: odd number of columns, odd number of inversions T is solvable: even number of columns, odd number of inversions, blank's row is in an even row (counting from bottom).

(c) The first row will be positions 1 through 9, the next row 10 through 18, the next row 19 through 27, the next row 28 through 36, and the middle element of the next row — the middle of the puzzle — will be in position 41.

Left and right slides will not change the number of inversions. Sliding up or down, the relative position of exactly 8 pairs of tiles will change, so the number of inversions will change by a total of at most 8.