INTRO2CS

Tirgul 7

What is recursion?

- Similar to mathematical **induction**
- A recursive definition is **self-referential**
- A larger, more complex instance of a problem is defined in terms of a smaller, simpler instance of the same problem
- A **base case** must be defined explicitly

When do we use recursion?

- We are given a large problem (say of size n)
- We notice that:
  - There is some simple base case we know how to solve directly (say n=0)
  - The solution to the large problem is composed of solutions to smaller problems of the same type
  - If we could solve a smaller instance of the problem
  - (say n-1), we could use that solution to solve the large problem

How do we use recursion?

- A function may call itself
- Such a function is called **recursive**
- There must be some base case that is handled explicitly, without a recursive call
- The other case has to make sure there is progress towards the base case.
- The recursive function call will use simpler/smaller arguments

The Three Laws of Recursion

1. A recursive algorithm must have a **base case.**
2. A recursive algorithm must change its state and **move toward the base case.**
3. A recursive algorithm must **call itself**, recursively.

Recursive factorial

- $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$
- By definition, $0! = 1$ (base case)
- Recursive definition: $\mathbf{n! = (n-1)! \cdot n}$
- For example:

$\mathbf{4!} =$

$3! \cdot 4 =$

$(2! \cdot 3) \cdot 4 =$

$((1! \cdot 2) \cdot 3) \cdot 4 =$

$(((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 =$

$(((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4 = \mathbf{24}$

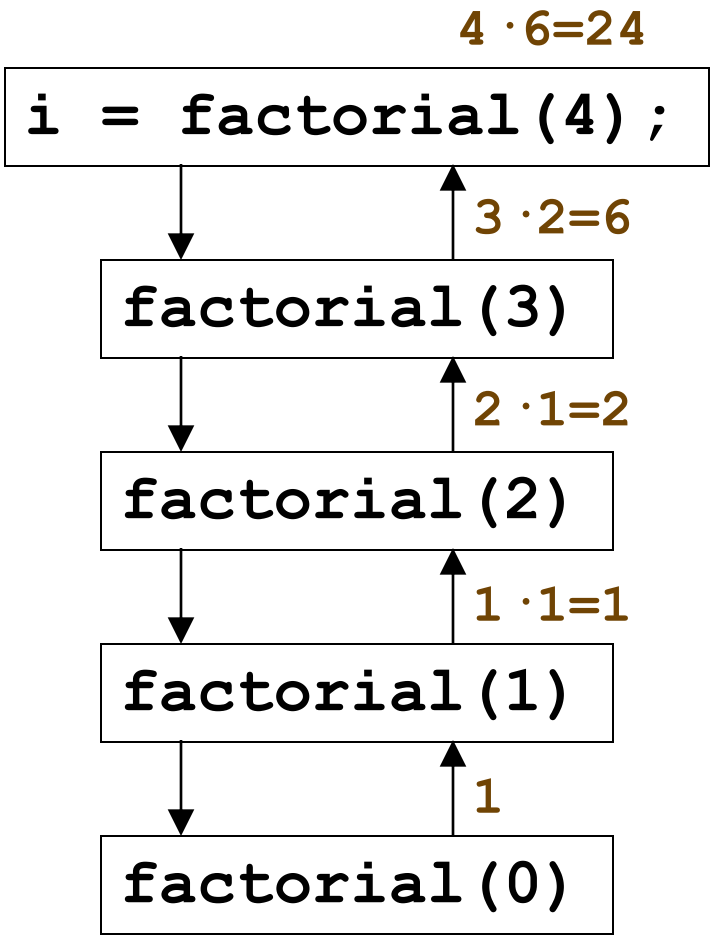
Recursive factorial


```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Base case

recursive call

What’s happening here?




```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Iterative factorial


```
def iterative_factorial(n):
    res == 1:
    for i in range(1,n+1):
        res *= i
    return res
```

Recursion vs. loops

- We could have calculated factorial using a loop
- In general, loops are more efficient than recursion
- However, sometimes recursive solutions are much simpler than iterative ones
- Recursion can be a powerful tool for solving certain types of problems
- Lets see a classic example

Recursive multiplication

- X = 10 *5
- How to solve recursively? Think Recursively!
- What will be the progression of the algorithm?
  - Divide to subproblems:
  - X = 10 *5 = 10+10*4 = 10 + 10 + 3
- What will be our base case?
  - Something that is easy to solve - a mathematical rule maybe?
  - X = X *1!

Recursive multiplication

```
def rmult(n1, n2):
    if n1 == 1:
        return n2
    return n2 + rmult(n1 - 1, n2)
```

Base case

recursive call

```
#rec ...5*10  ==10+(4*10)==10+10+(3*10) ...
```

Is palindrome?

- [1,2,3,4,3,2,1] is a palindrome
- יֶלֶד כּוֹתֵב בְּתוֹךְ דְּלִי is also palindrom
- Why recursion?
- What’s the base case?

Is palindrome?

```
def is_pal(s):
    if len(s) <= 1:
            return True
    else:
       return (s[0] == s[-1]) and
            is_pal(s[1:-1])
```

however…

```
def is_pal2(s):
     return s == s[::-1]
```

Pascal Triangle

- Why recursion?
- Let’s say we are interested on the n line in the triangle (pascal(n))
- What will be the base case?
- How to progress?

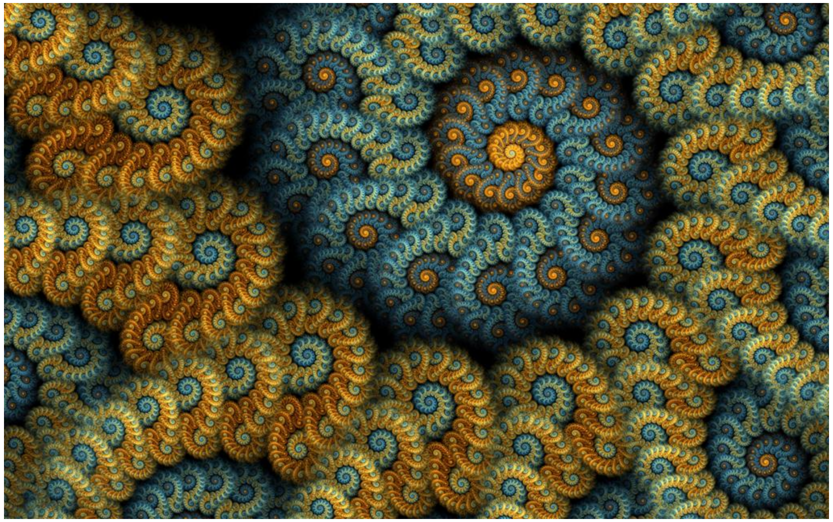
$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

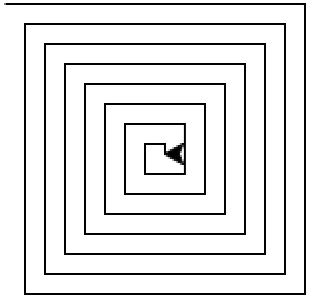
Pascal Triangle

```
import sys

def pascal(n):
        if n == 1:
                return [1]
        else:
                line = [1]
                previous_line = pascal(n-1)
                for i in range(len(previous_line)-1):
                                line.append(previous_line[i] +
                                    previous_line[i+1])
                line += [1]
        return line

print(pascal(int(sys.argv[1])))
```

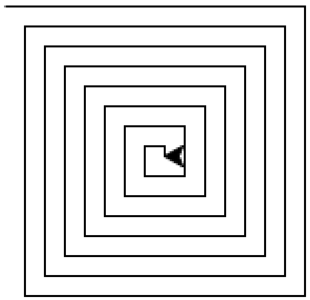
Fractals

A **fractal** is a never-ending pattern. **Fractals** are infinitely complex patterns that are self-similar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop.





Spiral

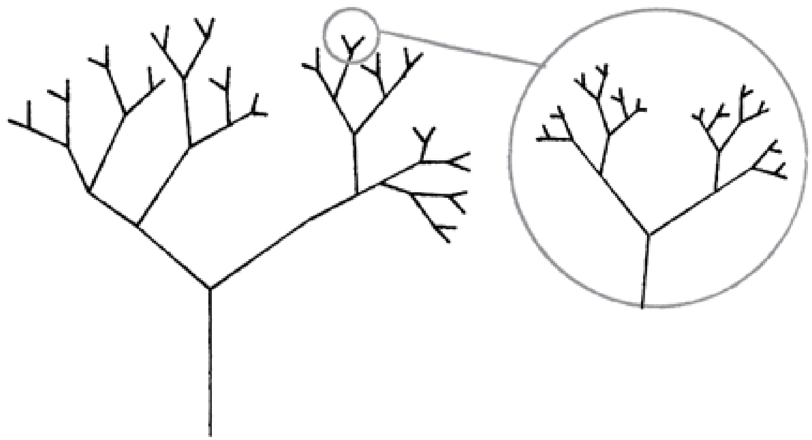


```
import turtle

def draw_spiral(tur, line_len):
    if line_len > 0:
            tur.forward(line_len)
            tur.right(90)
            draw_spiral(tur, line_len-5)

tur = turtle.Turtle()
draw_spiral(tur,100)
```

- Where's the base case?
- Where's the progress?

Fractal trees

Draw a fractal tree:



the shape of this branch resembles the tree itself. This is known as *self-similarity*, each part is a "reduced-size copy of the whole."

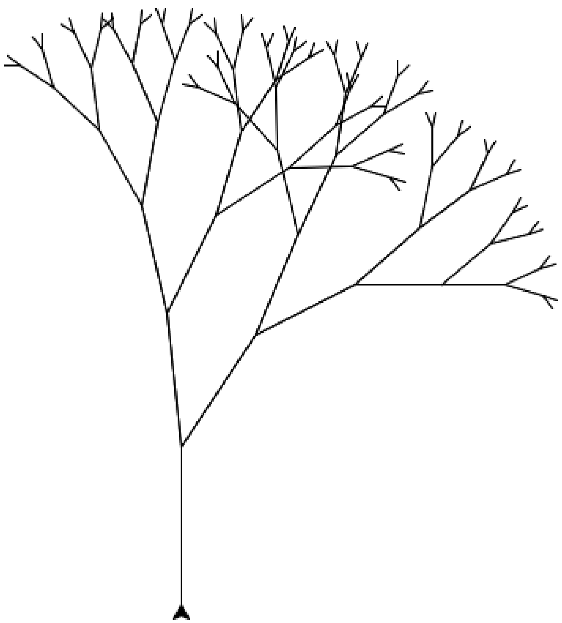
Fractal trees

```
import turtle

def tree(branch_len, tur):
  if branch_len > 5:
      trtle.forward(branch_len)
      trtle.right(20)
      tree(branch_len-15, trtle)
      trtle.left(40)
      tree(branch_len-15, trtle)
      trtle.right(20)
      trtle.backward(branch_len)
```

```
def main():
      t = turtle.Turtle()
      t.left(90)
      t.up()
      t.backward(250)
      t.down()
      tree(t, 100)

main()
```

Probabilistic trees

```
import turtle
import random

def prob_tree(branch_len, trtle):
  deg = random.uniform(0, 40)
  if branch_len > 5:
      trtle.forward(branch_len)
      trtle.right(deg)
      prob_tree(branch_len-15, trtle)
      trtle.left(40)
      prob_tree(branch_len-15, trtle)
      trtle.right(40-deg)
      trtle.backward(branch_len)
```



Exploring all states using recursion Backtracking

- We can use recursion to go over many options, and do something for each case.

- Example:
- **printing all subsets of the set S = {0,…,n-1} (printing the power set of S).**
- Difficult to do with loops (but possible).
- Much simpler with recursion.

Power Set - The basic idea

- Lets decompose the problem to two smaller problems of the same type.

- The recursive decomposition:
  - Print all subsets that contain an item,
  - Then print all the subsets that do not contain it.

- Keep track of our current “state”.
  - items that are in the current subset,
  - items not in the current subset,
  - items we did not decide about yet.

Power Set – python code

```
def power_set(n):

    cur_set = [False]*n

    power_set_helper(cur_set, 0)
```

This is not the recursive function. It calls the recursive function that does the real work.

Holds the subset we are currently building.

The recursive function

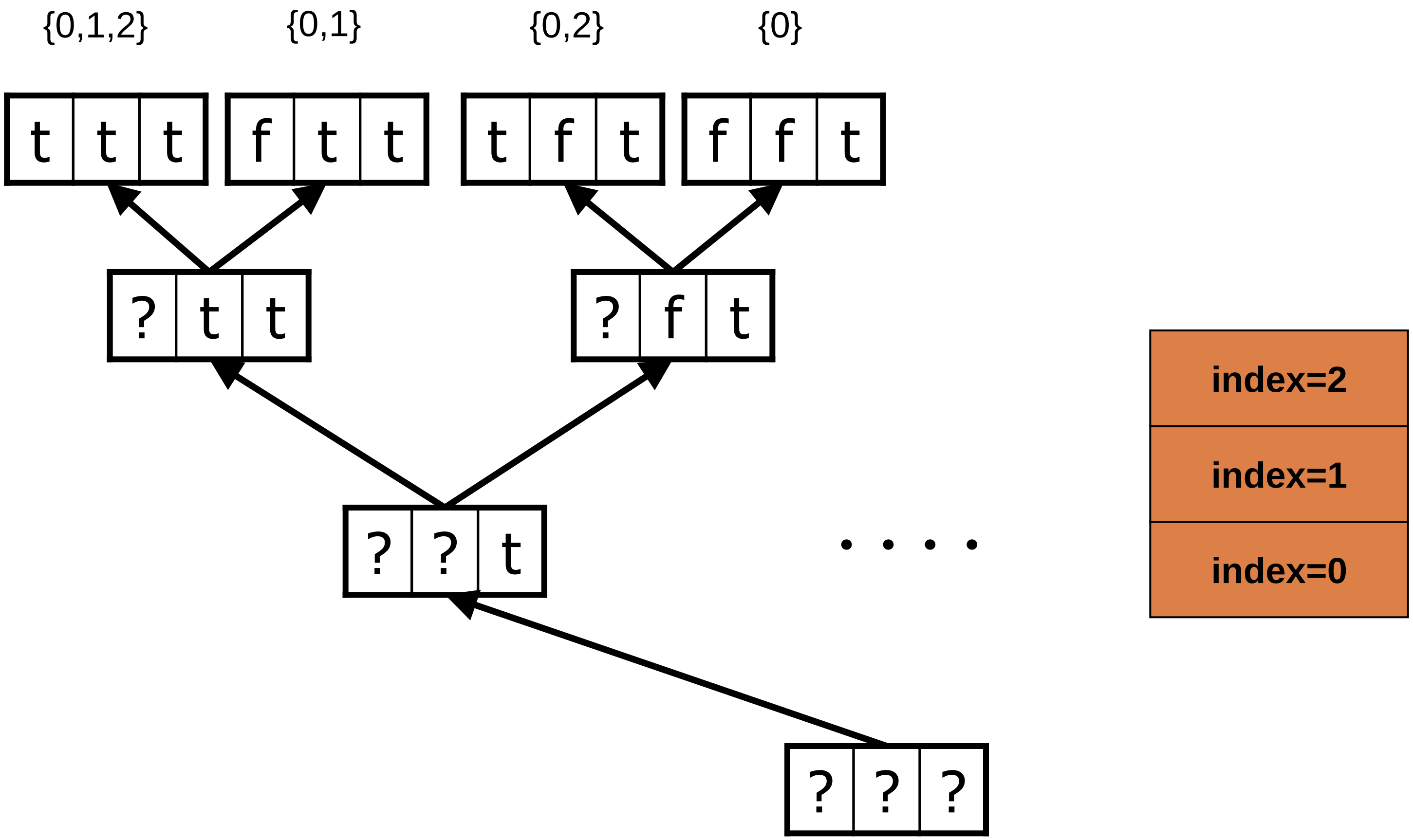
Power Set – python code

```
def power_set_helper(cur_set, index):

    #base:we picked out all the items in the set
    if index==len(cur_set):
        print_power_set(cur_set)
        return

    #runs on all sets that include this index
    cur_set[index] = True
    power_set_helper(cur_set, index+1)

    #runs on all sets that does not include index
    cur_set[index] = False
    power_set_helper(cur_set, index+1)
```

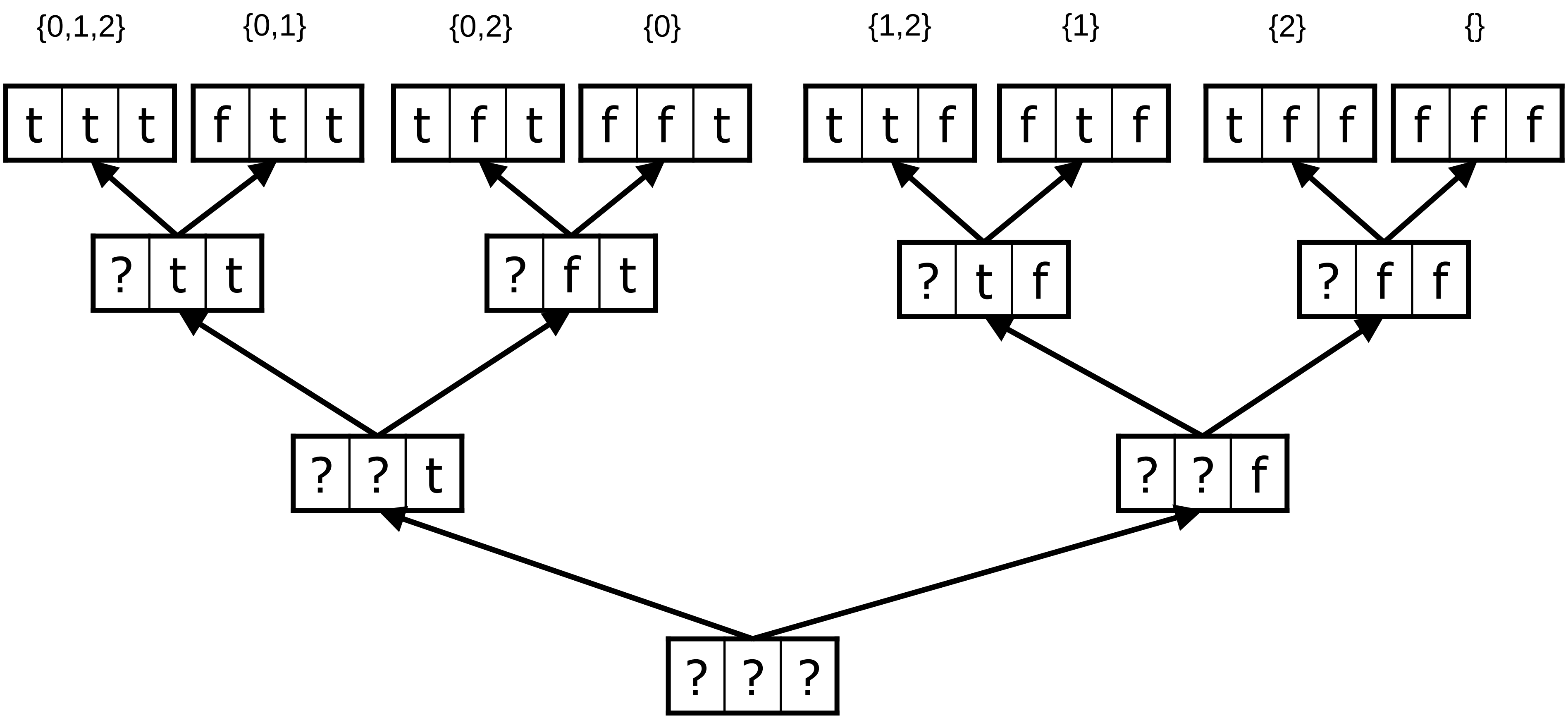
Power Set – python code

```python
def print_power_set(cur_set):

    print('{', end=' ')
    for (idx, in_cur_set) in enumerate(cur_set):
        if in_cur_set:
            print(idx, end=' ')
    print('}')
```

`power_set` and the stack



`power_set` and the stack



Sort using recursion - Quicksort

- A very efficient sorting algorithm
- A probabilistic algorithm:
- On average, the algorithm takes O(n log n) comparisons to sort n items.
- In the worst case, it makes O(n2) comparisons, though this behavior is rare.

Quick Sort

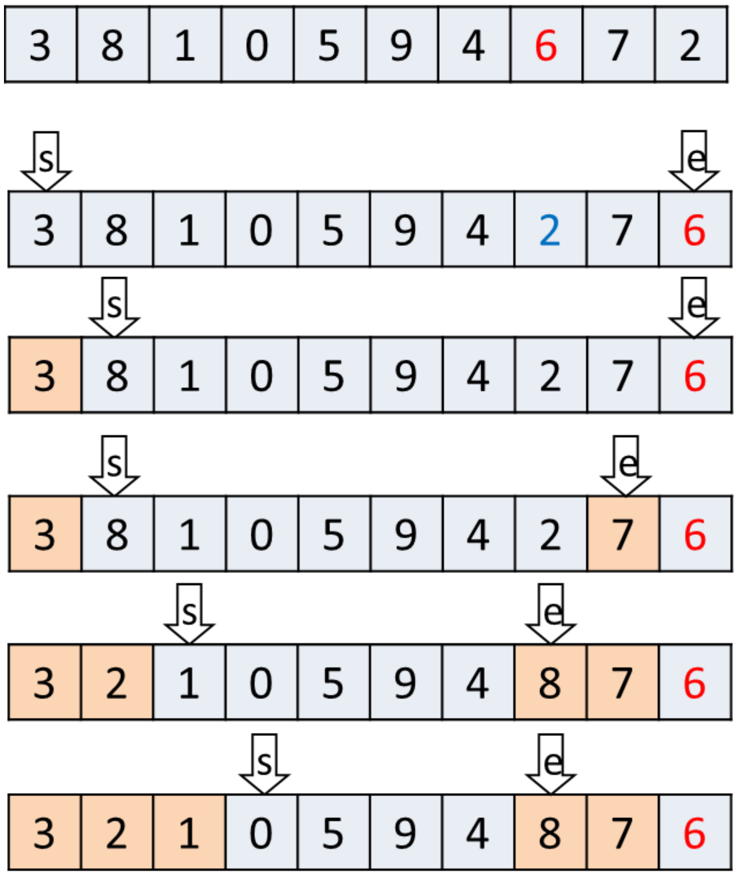
- Choose an element from the list called *pivot*
- Partition the list:
  - All elements < *pivot* will be on the left
  - All elements ≥ *pivot* will be on the right
- Recursively call the *quicksort* function on each part of the list

Quick Sort - implementation


```
def quicksort(data):
    quicksort_helper(data, 0 , len(data))

def quicksort_helper(data, start, end):
    if(start < end-1):
        pivot_idx = partition(data, start, end)
        quicksort_helper(data, start, pivot_idx)
        quicksort_helper(data, pivot_idx+1, end)
```

Quick Sort – implementation (II)

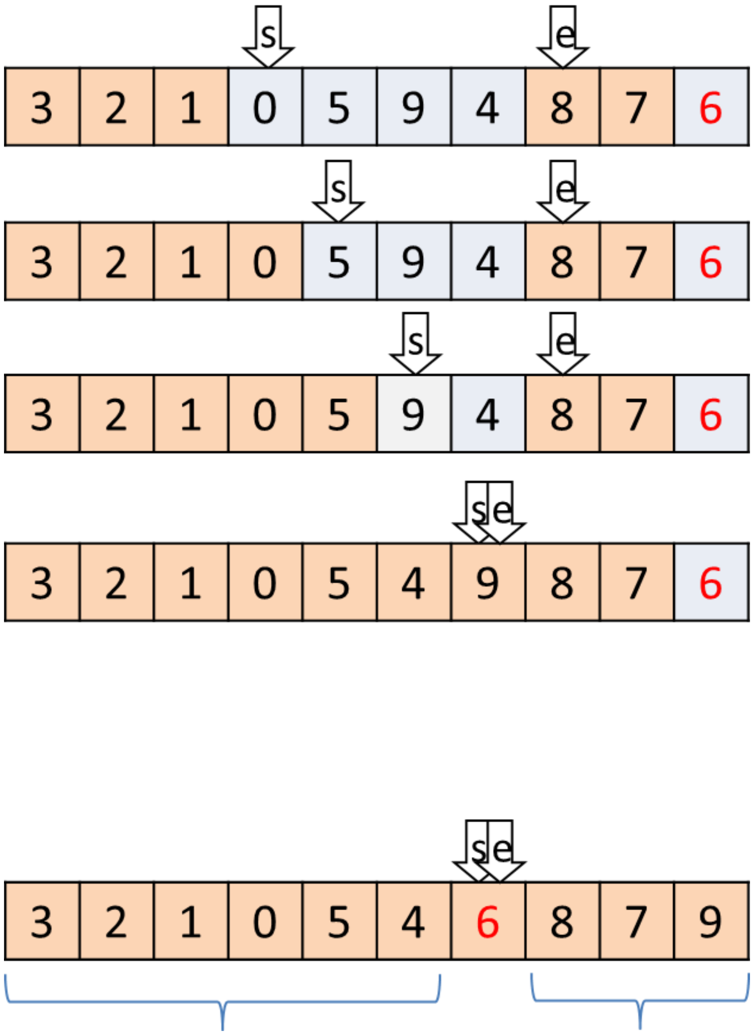

```python
def partition(data, start, end):
    pivot_idx = random.randint(start, end-1)
    pivot = data[pivot_idx]
    swap(data,pivot_idx, end-1)
    pivot_idx = end-1
    end -= 1
    while(start < end):
        if(data[start] < pivot):
            start += 1
        elif(data[end-1] >= pivot):
            end -= 1
        else:
            swap(data, start, end-1)
            start += 1
            end -= 1
    swap(data,pivot_idx, start)
    return start
```




```python
def swap(data,ind1,ind2):
    data[ind1],data[ind2] = data[ind2],data[ind1]
```

Quick Sort – implementation (III)

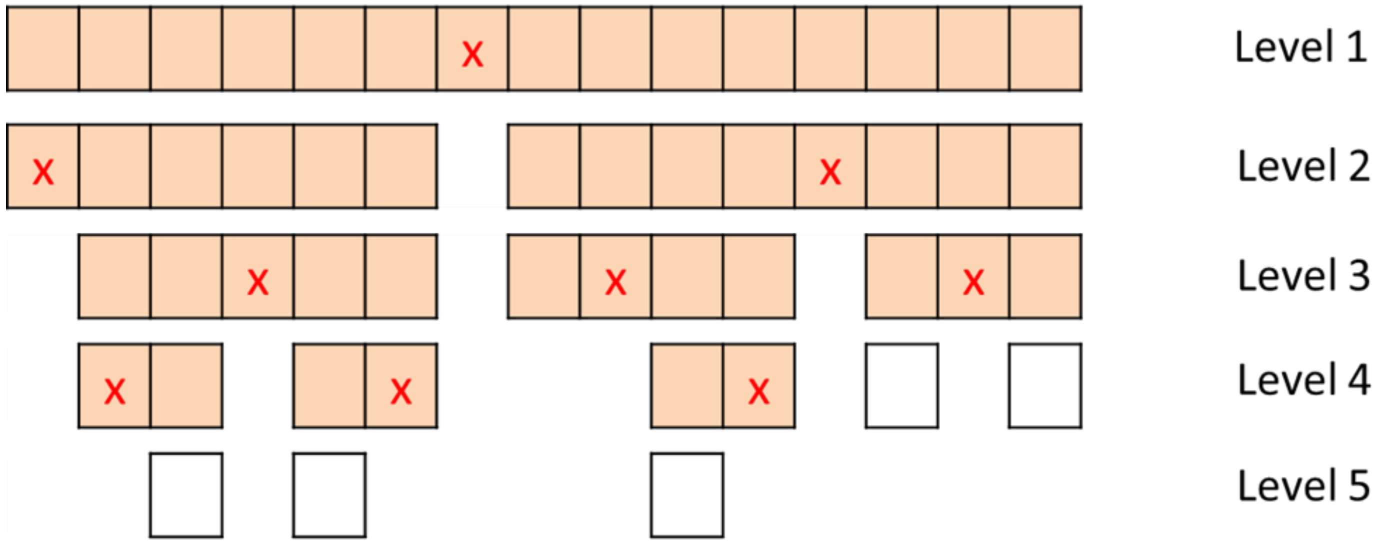

```
def partition(data, start, end):
    pivot_idx = random.randint(start, end-1)
    pivot = data[pivot_idx]
    swap(data,pivot_idx, end-1)
    pivot_idx = end-1
    end -= 1
    while(start < end):
        if(data[start] < pivot):
            start += 1
        elif(data[end-1] >= pivot):
            end -= 1
        else:
            swap(data, start, end-1)
            start += 1
            end -= 1
    swap(data,pivot_idx, start)
    return start
```



Quick Sort – Runtime Analysis (I)

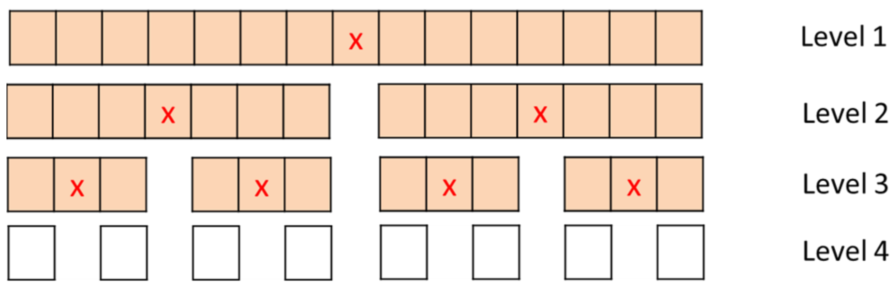
- On each level of the recursion, we go over lists that contain total of n elements:

  About n steps at each level



Quick Sort – Runtime Analysis (II)

- How many levels are there?
- It depends on the pivot value:
  - Lets say we choose each time the median value
  - Each time the list is divided by half:
    - ⇒ n/2
    - ⇒ n/4
    - ⇒ …
    - ⇒ 1

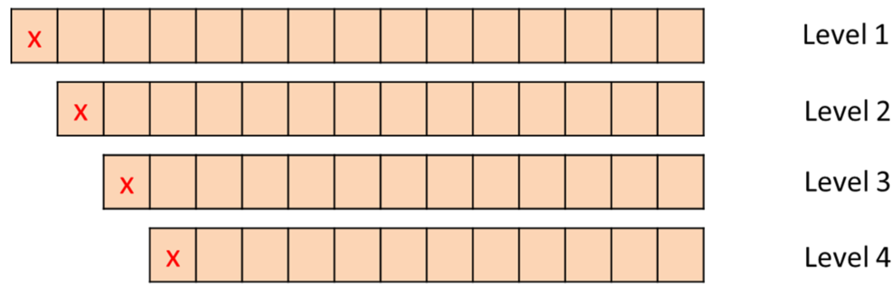


  - There will be log(n) levels, and each takes n steps

⇒ ***It would take about nlog(n) steps***

Quick Sort – Runtime Analysis (III)

- Lets say we choose each time an extreme value (smallest or largest) – it is unlikely
- Each time we get one list of size 1 and one of size n-1:
  - ⇒ n-1
  - ⇒ n-2
  - ⇒ ...
  - ⇒ 1



- There will be n levels, and each takes n steps

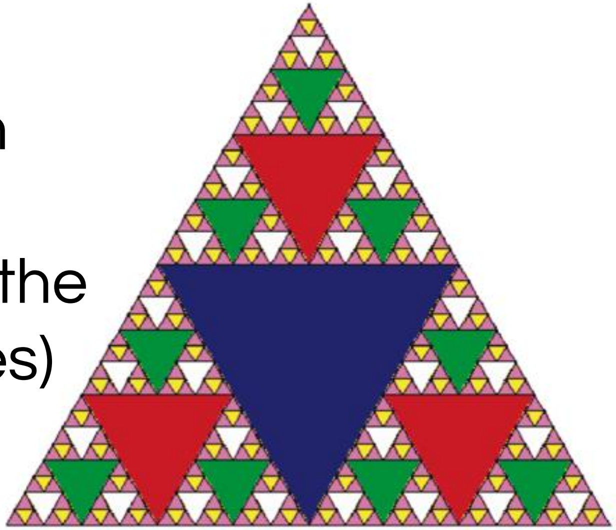
⇒ ***It would take about n^2 steps***

- The efficiency is depended on the pivot choice!

Bonus Slides

Sierpinski Triangle

- A fractal that exhibits the property of self-similarity is the Sierpinski triangle
- Algorithm:
  - Start with a single large triangle
  - Divide this large triangle into four new triangles by connecting the midpoint of each side.
  - Ignore the middle triangle that you just created
  - apply the same procedure to each of the three corner triangles
  - The base is defined as the level of the triangle (how many inner triangeles)



Sierpinski Triangles

```
def draw_triangle(points, color, tur):
  tur.fillcolor(color)
  tur.up()
  tur.goto(points[0][0],points[0][1])
  tur.down()
  tur.begin_fill()
  tur.goto(points[1][0], points[1][1])
  tur.goto(points[2][0], points[2][1])
  tur.goto(points[0][0], points[0][1])
  tur.end_fill()

def get_mid(p1, p2):
  return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)
```

Sierpinski Triangles

```
def sierpinski(points, degree, tur):
    colormap = ['blue', 'red', 'green', 'white', 'yellow', 'violet', 'orange']
    draw_triangle(points, colormap[degree], tur)
    if degree > 0:
    sierpinski([points[0], get_mid(points[0], points[1]),
                        get_mid(points[0], points[2])],
                        degree-1, tur)
    sierpinski([points[1], get_mid(points[0], points[1]),
                                get_mid(points[1], points[2])],
                          degree-1, tur)
    sierpinski([points[2], get_mid(points[2], points[1]),
                                get_mid(points[0], points[2])],
                          degree-1, tur)
```

middle triangle

triangle 1

triangle 2

triangle 3

Understanding the Traceback

```
# in file t.py:
def a(L):
    return b(L)

def b(L):
    return L.len() #should have been len(L)
# in the python shell we try
a(L)
```

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    a(L)
NameError: name 'L' is not defined
```

Understanding the Traceback

```
# in file t.py:
def a(L):
    return b(L)

def b(L):
    return L.len() #should have been len(L)
# in the python shell we try
a([1,2,3])
```

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    a(L)
  File ”…/t.py", line 2, in a
    return b(L)
  File ”…/t.py", line 6, in b
    return L.len()
```

AttributeError: 'list' object has no attribute 'len'

Understanding the Traceback

```
# in file t.py:
def c(L):
    print((L[0])
    print("bye")
# in the python shell we try
a([])
```

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    c([])
  File "…\t.py", line 10, in c
    print(L[0])
```

IndexError: list index out of range

Understanding the Traceback

```
# in file t.py:
def c(L):
    print((L(0))
    print("bye")
# in the python shell we try
c([1,2,3])
```

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    c([1,2,3])
  File "…\t.py", line 9, in c
    print(L(0))
```

TypeError: 'list' object is not callable

Understanding the Traceback

```
# in file t.py:
def c(L):
    print((L[0])
    print("bye")
```

invalid syntax (but the next line is marked)
or **unexpected EOF while parsing** if this is the last line in the

Tips

- Pay attention to indentation (and other idle formatting issues) – it might imply on bugs
- Make sure you are in the right range when working with containers
- Adding printouts might be helpful
- You can use Google with the error name (e.g. **TypeError: 'list' object is not callable)**

Exploring all states using backtracking

- A backtracking alg. can be used to find a solution (or all solutions) to a combinatorial problem.

- Solutions are constructed incrementally

- If there are several options to advance incrementally, the algorithm will try one option, then backtrack and try more options.

- If you reach a state where you know the path will not lead you to the solution, backtrack!

N-Queens

- The problem:
  - On an NxN chess board, place N queens so that no queen threatens the other (no other queen allowed in same row,col or diagonal).
  - Print only one such board.
- Simplifying step:
  - Place 1 queen somewhere in an available column then solve the problem of placing all other queens.
- Base case:
  - All queens have been placed.

The N-Queen Problem - helper functions

```
def illegal_placement(board, row, col):
    #Note: it is enough to look for threatening queens in lower columns
    for delta in range(1,col+1):
        #Check for queen in the same row or in upper diagonal or in lower diagonal
        if (board[row][col-delta] or
            (row-delta>=0 and board[row-delta][col-delta]) or
            (row+delta<len(board) and board[row+delta][col-delta])):
            return True
    return False

def print_board(board):
    for row in board:
        for q in row:
            print('Q',end=' ') if q else print('-',end=' ')
        print()
```

The N-Queen Problem - the recursion function

```
def place_queen_at_col(board, col):
    #Base case: we have passed the last column
    if col == len(board[0]):
        return True
    #Iterate over rows until it is okay to place a queen
    for row in range(len(board)):
        if illegal_placement(board, row, col):
            continue
        #place the queen
        board[row][col] = True
        #Check if we can fill up the remaining columns
        if place_queen_at_col(board, col+1):
            return True
        #If not, remove the queen and keep iterating
        board[row][col] = False
    #If no placement works, give up
    return False
```

The N-Queen Problem - calling the recursive function

```
#This function uses a recursive helper method that really does
   the work
def place_queens(board_size):
    board = []
    for i in range (board_size):
        board.append([])
        for j in range (board_size):
            board[i].append(False)
if place_queen_at_col(board, 0):
        print_board(board)
    else:
        print("No Placement Found!")
```

```
# what would happen
if we were trying to
do it using:
    # board =
[[False]*board_size]
*board_size
```

Output of N-Queens

```
Q - - - - - - -
- - - - - - Q -
- - - - Q - - -
- - - - - - - Q
- Q - - - - - -
- - - Q - - - -
- - - - - Q - -
- - Q - - - - -
```