0.1 Stack overflow

A stack overflow error most commonly occurs when a recursive function makes an exceedingly large numbers of calls to itself (usually because of not properly setting up a base case), causing the memory allocated for a function's call stack to overflow. Python prevents this from happening by throwing a RecursionError when a certain number of recursive calls are made (this number is system-dependent)

0.2 Binary search

A stack overflow error most commonly occurs when a recursive function makes an exceedingly large numbers of calls to itself (usually because of not properly setting up a base case), causing the memory allocated for a function's call stack to overflow. Python prevents this from happening by throwing a RecursionError when a certain number of recursive calls are made (this number is system-dependent)

```python
def binary_search(data, key):
    """
    Examples
    --------
    >>> binary_search([1, 7, 35, 45, 67], 3)
    False
    >>> binary_search([1, 7, 35, 45, 67], 7)
    True
    """
    if len(data) == 1:
        return data[0] == key

    mid = len(data)//2
    if key < data[mid]:
        return binary_search(data[:mid], key)
    else:
        return binary_search(data[mid:], key)
```

Fibonacci Recursion: Time Complexity: $O(2^n)$
Space Complexity: $O(n)$

```python
  def f(n):
    if n == 1 or n == 2:
        return 1
    else:
        return f(n-1) + f(n-2)
```

0.3 Linked-list

A stack overflow error most commonly occurs when a recursive function makes an exceedingly large numbers of calls to itself (usually because of not properly setting up a base case), causing the memory allocated for a function's call stack to overflow. Python prevents this from happening by throwing a RecursionError when a certain number of recursive calls are made (this number is system-dependent)

0.4 Binary Tree

```python
class BinaryTree:

    def __init__(self, item):
        self.item = item
        self.left = None # type = BinaryTree
        self.right = None # type = BinaryTree

    def insert(self, item):
        # pick a random side to insert on
        left = np.random.rand() < 0.5
        if item == left:
            return
        if left:
            if self.left is None:
                self.left = BinaryTree(item)
            else:
                self.left.insert(item)
        else:
            if self.right is None:
                self.right = BinaryTree(item)
            else:
                self.right.insert(item)

    def contains(self, item):
        if self.item == item:
            return True

        if self.left is not None:
            if self.left.contains(item):
                return True

        if self.right is not None:
            if self.right.contains(item):
                return True

        return False

    def print_tree(self, level=0, prefix="Root: "):
        """Recursively prints the tree structure."""
        print(" " * level + prefix + str(self.item))
        if self.left:
            self.left.print_tree(level + 1, prefix="L--- ")
        if self.right:
            self.right.print_tree(level + 1, prefix="R--- ")

    # We would want some more functions here, e.g. to add/remove
      things from the tree.
```

nearest neighbour: time complexity for n points in k dimensions $O(nk)$

0.5 K-D Trees

But, as we've seen with trees and hash tables, sometime we speed things up with better data structures. One of the classic ways to speed up nearest neighbours is a data structure call the k-d tree.
Basic idea:
In each recursive step, there is a certain number of datapoints. If there's only one, we're done.
Otherwise, for one of the two dimensions (we alternate back and forth), find the median value along the dimension.
Split the data into two subsets based on being above or below that median, and build a (sub)tree for each of those subsets.
Starting from the full dataset, you will create a tree where each leaf is a datapoint.
You can find an approximate nearest neighbour by traversing the down the tree using the same decision points as were used to original split the data; the final leaf is the desired neighbour.

0.6 Timing experiments

```python
n_sizes = [100, 1000, 10_000, 100_000]

results = defaultdict(list)
results["n"] = n_sizes

d = 10

for n in n_sizes:
    print('n: ', n)
    X = np.random.rand(n, d)
    query = np.random.rand(1, d)

    print(" KDTree")
    time = %timeit -q -o -r 3 sklearn.neighbors.KDTree(X)
    results["KDTree init"].append(time.average)
    KDT = sklearn.neighbors.KDTree(X)

    time = %timeit -q -o -r 3 KDT.query(query)
    results["KDTree query"].append(time.average)

    print(" Brute force")
    time = %timeit -q -o -r 3 nearest_neighbour(X, query)
    results["Brute force"].append(time.average)
```

Other nearest neighbour approaches
Note: there are other nearest neighbour approaches besides k-d trees, including some very fast approximate algorithms.
In general, you can often do something faster if the result can be slightly wrong.
There are approaches based on hashing instead of trees.

0.7 amortization of hash table growth

Growth is slow, but only occurs rarely, and so the cost "averages out" because after adding n elements you've spent $O(n)$ time on growth, for an average of $O(1)$ per insertion.