1) *Quick sort* general idea is as follows.

- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (called *partitioning*) the unsorted items such that:

Pivot Index

All items < to Pivot	Pivot Item	All items >= to Pivot

- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

a) Given the following `partition` function which returns the index of the pivot after this rearrangement, complete the recursive `quicksortHelper` function.

```
def partition(lyst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) / 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in xrange(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot item and the boundary item
    temp = lyst[boundary]
    lyst[boundary] = lyst[right]
    lyst[right] = temp
    return boundary
```

```
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
```

b) For the list below, trace the first call to partition and determine the resulting list, and value returned.

	0	1	2	3	4	5	6	7	8	left	right	index	boundary
lyst:	54	26	93	17	77	31	44	55	20	0	8		

b) What initial arrangement of the list would cause partition to perform the most amount of work?

c) Let "n" be the number of items between left and right. What is the worst-case *O*( ) for partition?

d) What would be the overall, worst-case *O*( ) for Quick Sort?

e) Why does the partition code select the middle item of the list to be the pivot?

f) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

g) What would be the big-oh for Quick Sort in the average case?

2) Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the **fewest number** of coins for this amount of change.

a) What "greedy" algorithm would you use to solve this problem with US coin types of {1, 5, 10, 25, 50} and a change amount of 29-cents?

b) Do you get the correct solution if you use this algorithm for coin types of {1, 5, 10, 12, 25, 50} and a change amount of 29-cents?

3) One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms.
Solve a problem by:

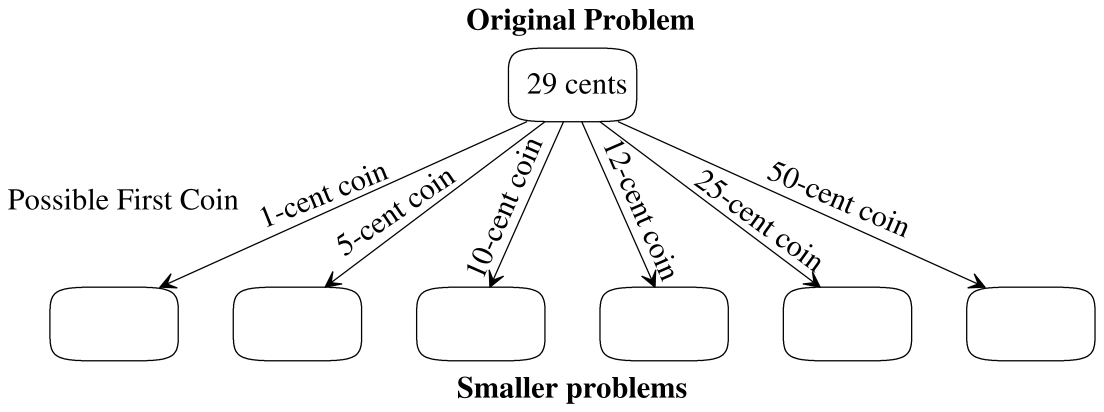
- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

a) For the coin-change problem, what determines the size of the problem?

b) How could we divide the coin-change problem for 29-cents into smaller problems?

c) If we knew the solution to these smaller problems, how would be able to solve the original problem?

4) After we give back the first coin, which smaller amounts of change do we have?

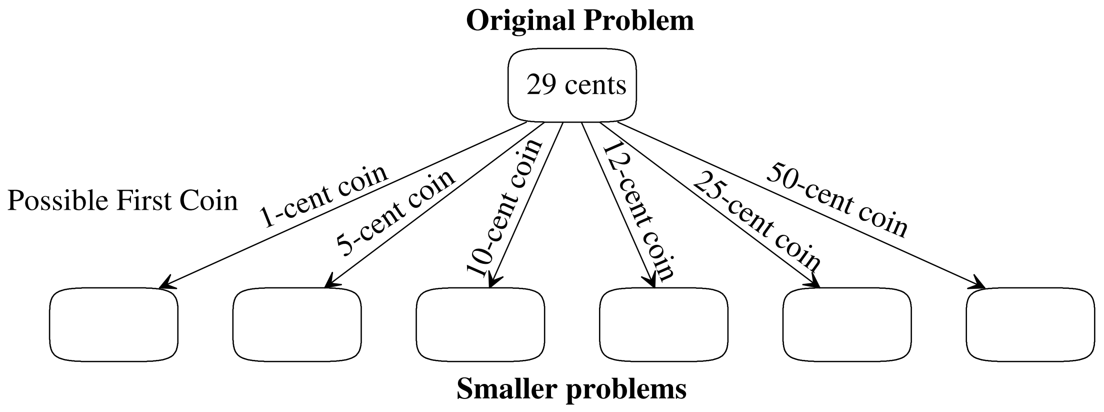


5) If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

6) Complete a recursive relationship for the fewest number of coins.

$$\text{FewestCoins(change)} = \begin{cases} \min\limits_{\text{coin} \in \text{CoinSet}}(\text{ FewestCoins}(\qquad\qquad\qquad)\,) + & \text{if change} \notin \text{CoinSet} \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

7) Complete a couple levels of the recursion tree for 29-cents change using the set of coins {1, 5, 10, 12, 25, 50}.


Original Problem
29 cents
Possible First Coin
1-cent coin
5-cent coin
10-cent coin
12-cent coin
25-cent coin
50-cent coin
Smaller problems