b) When would a "child" tree node NOT be promising? 5-coin solution best so far

(best solution so far #coins) ≤ (child's #coins/level)

3. Consider the output of running the backtracking code with pruning (next page) twice with a change amount of 63 cents.

```
Change Amount: 63  Coin types: [1, 5, 10, 25]
Run-time: 0.036 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 1-cent coins is 3
number of 5-cent coins is 0
number of 10-cent coins is 1
number of 25-cent coins is 2
Number of Backtracking Nodes: 4836
```

```
Change Amount: 63  Coin types: [25, 10, 5, 1]
Run-time: 0.003 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 25-cent coins is 2
number of 10-cent coins is 1
number of 5-cent coins is 0
number of 1-cent coins is 3
Number of Backtracking Nodes: 310
```

a) Explain why ordering the coins from largest to smallest produced faster results.

code on left's first solution is 63 coins which is not helpful in pruning

code on right's first solution will be the greedy solution which is much better for pruning

b) For coins of [50, 25, 12, 10, 5, 1] typical timings:

Change Amount	Run-Time (seconds)	Number of Tree Nodes
399	8.88	2,015,539
409	55.17	12,093,221
419	318.56	72,558,646

Why the exponential growth in run-time?

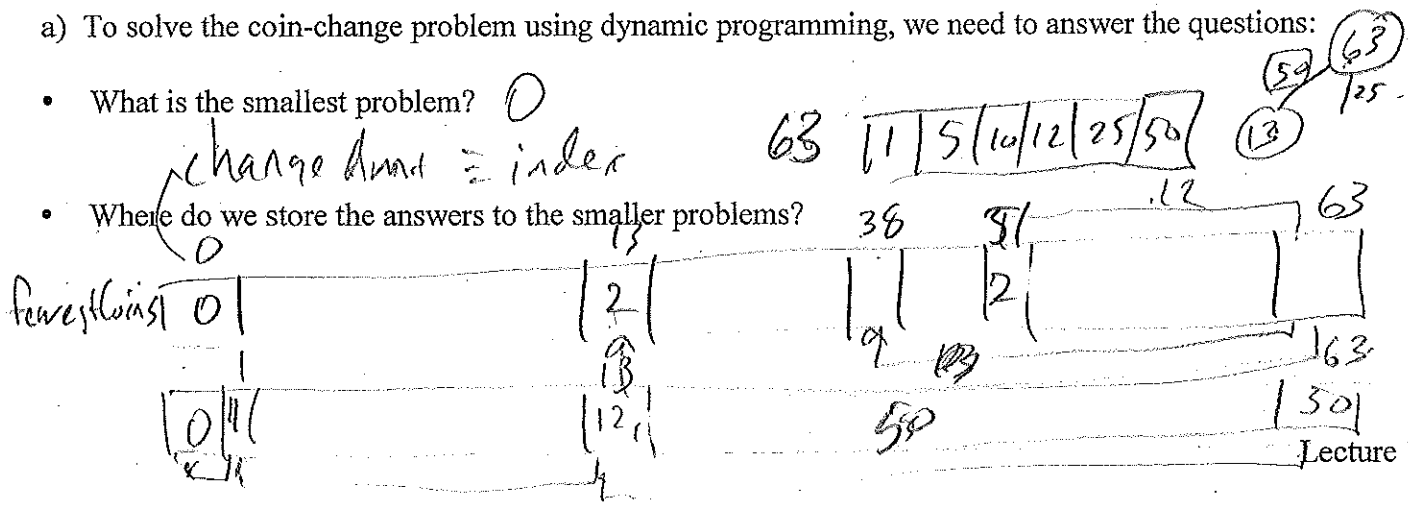
Even with pruning we still have a lot of nodes in the tree that get recalculated, for same change amounts.

4. As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

- Solve smaller problems before larger ones
- store their answers
- look-up answers to smaller problems when solving larger subproblems, so each problem is solved only once

a) To solve the coin-change problem using dynamic programming, we need to answer the questions:

- What is the smallest problem? 0
- Where do we store the answers to the smaller problems? change amt = index



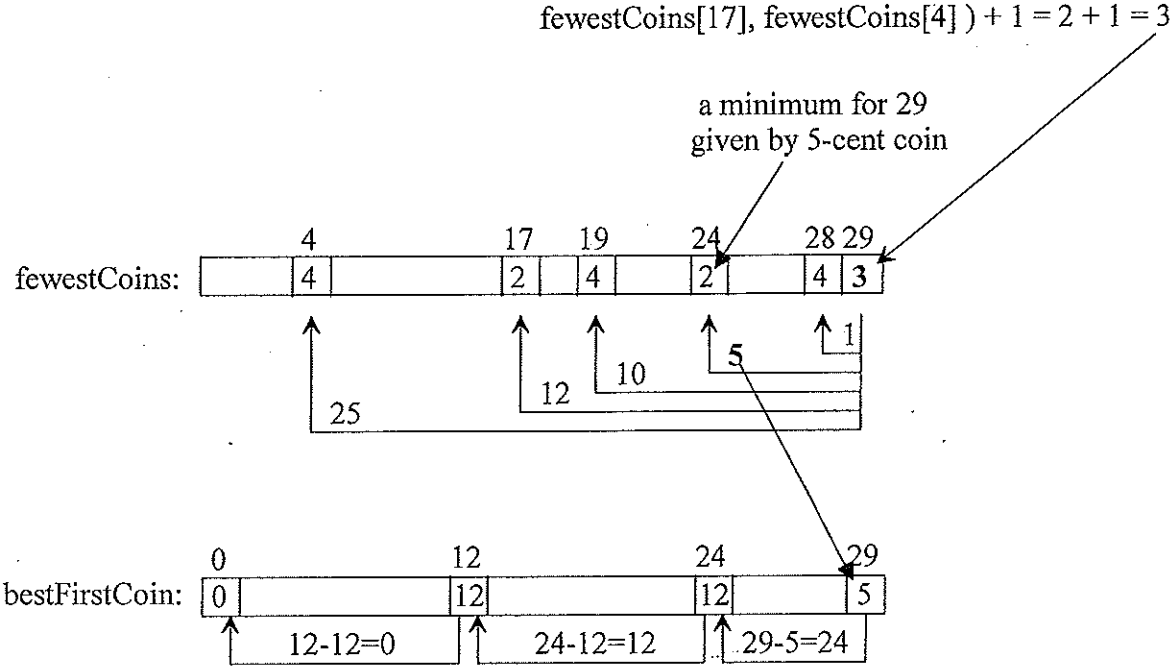
Dynamic Programming Coin-change Algorithm:

I. Fills an array fewestCoins from 0 to the amount of change. An element of fewestCoins stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types {1, 5, 10, 12, 25, 50}, the dynamic programming algorithm would have previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

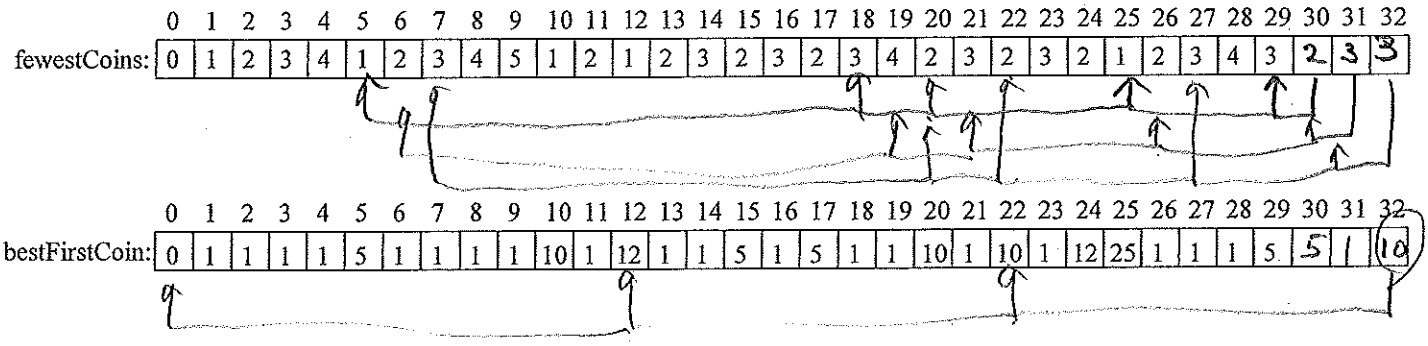
II. If we record the best, first coin to return for each change amount (found in the "minimum" calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

fewestCoins[29] = minimum(fewestCoins[28], fewestCoins[24], fewestCoins[19], fewestCoins[17], fewestCoins[4]) + 1 = 2 + 1 = 3



Extract the coins in the solution for 29-cents from bestFirstCoin[29], bestFirstCoin[24], and bestFirstCoin[12]

b) Extend the lists through 32-cents.



c) What coins are in the solution for 32-cents? 12 10 10

1. Consider the following sequential search (linear search) code:

Textbook's Listing 5.1	Faster sequential search code

def sequentialSearch(alist, item):
  """ Sequential search of unorder list """
  pos = 0
  found = False
 	
  while pos < len(alist) and not found:
  if alist[pos] == item:
  found = True
  else:
  pos = pos+1
 	
  return found
	def linearSearch(aList, target):
	  """Returns the index of target in aList
	  or -1 if target is not in aList"""
	  for position in range(len(aList)):
	  if target == aList[position]:
	  return position
	  return -1

a) What is the *basic operation* of a search? compare two items

b) For the following `aList` value, which `target` value causes `linearSearch` to loop the fewest ("best case") number of times? 10

	0	1	2	3	4	5	6	7	8	9	10
aList:	10	15	28	42	60	69	75	88	90	93	97

c) For the above `aList` value, which `target` value causes `linearSearch` to loop the most ("worst case") number of times? 97 or unsuccessful search

d) For a *successful search* (i.e., `target` value in `aList`), what is the "average" number of loops? ("n" = len(aList))

n/2

Textbook's Listing 5.2	Faster sequential search code

def orderedSequentialSearch(alist, item):
  """ Sequential search of order list """
  pos = 0
  found = False
  stop = False
  while pos < len(alist) and not found and not stop:
  if alist[pos] == item:
  found = True
  else:
  if alist[pos] > item:
  stop = True
  else:
  pos = pos+1
 	
  return found
	def linearSearchOfSortedList(target, aList):
	  """Returns the index position of target in
	  sorted aList or -1 if target is not in aList"""
	  breakOut = False
	  for position in range(len(aList)):
	  if target <= aList[position]:
	  breakOut = True
	  break
	
	  if not breakOut:
	  return -1
	  elif target == aList[position]:
	  return position
	  else:
	  return -1

e) The above version of linear search assumes that `aList` is sorted in ascending order. When would this version perform better than the original `linearSearch` at the top of the page?

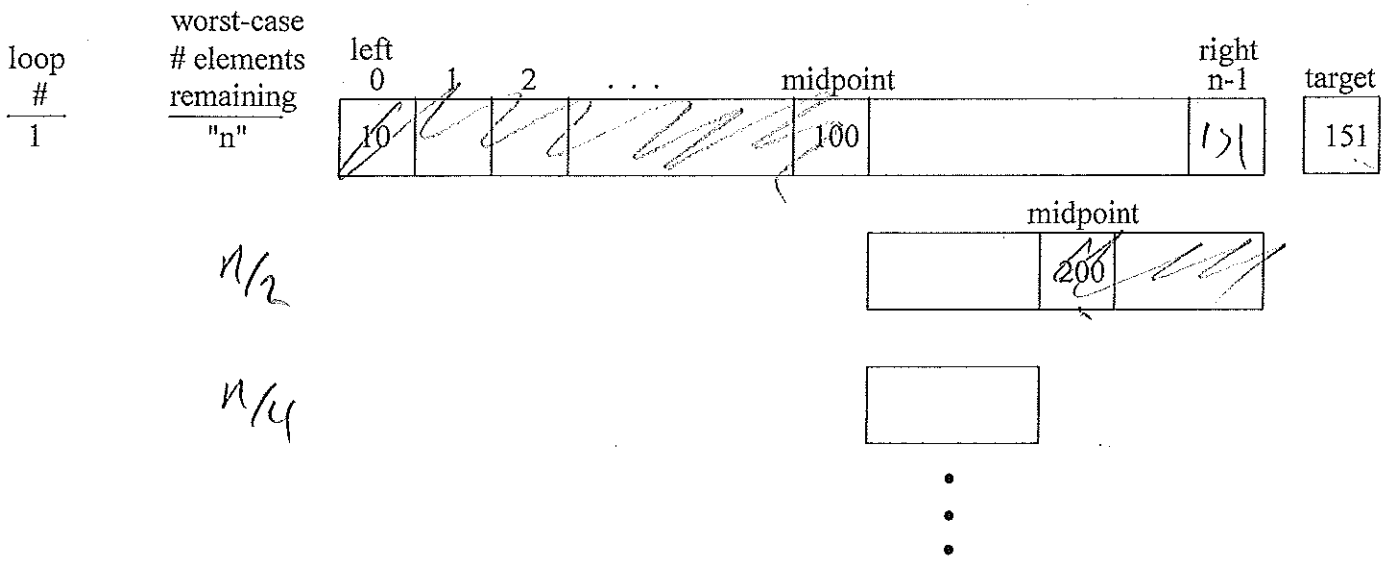
When it can stop early by finding a list value that's ≥ the target.

2. Consider the following binary search code:

Textbook's Listing 5.3	Faster binary search code

def binarySearch(alist, item):
  first = 0
  last = len(alist)-1
  found = False
 	
  while first<=last and not found:
  midpoint = (first + last)//2
  if alist[midpoint] == item:
  found = True
  else:
  if item < alist[midpoint]:
  last = midpoint-1
  else:
  first = midpoint+1
 	
  return found
	def binarySearch(target, lyst):
	  """Returns the position of the target
	  item if found, or -1 otherwise."""
	  left = 0
	  right = len(lyst) - 1
	  while left <= right:
	  midpoint = (left + right) // 2
	  if target == lyst[midpoint]:
	  return midpoint
	  elif target < lyst[midpoint]:
	  right = midpoint - 1
	  else:
	  left = midpoint + 1
	  return -1

a) "Trace" binary search to determine the worst-case basic total number of comparisons?



b) What is the worst-case big-oh for binary search? $O(\log_2 n)$

c) What is the best-case big-oh for binary search? $O(1)$

d) What is the average-case (expected) big-oh for binary search? $O(\log n)$

$2^{10} = 1024$ $2^{20} =$

e) If the list size is 1,000,000, then what is the maximun number of comparisons of list items on a *successful search*?

~20

f) If the list size is 1,000,000, then how many comparisons would you expect on an *unsuccessful search*?

~20

3. Hashing Motivation and Terminology:

a) Sequential search of an array or linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.

If *n* is the number of items being searched, what is the average and worst case big-oh notation for a sequential search?

average case $O(\ n/2\ )$

worst case $O(\ n\ )$

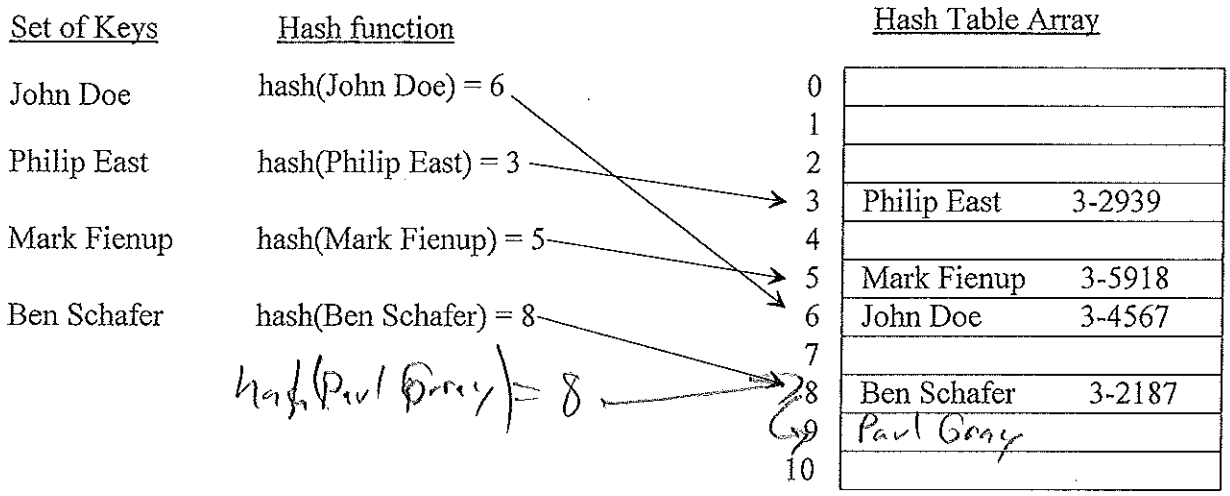
b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.

If *n* is the number of items being searched, what is the average and worst case big-oh notation for a search?

average case $O(\ \log n\ )$

worst case $O(\ \log n\ )$

Hashing tries to achieve average constant time (i.e., $O(1)$) searching by using the target's value to calculate where in the array/Python list (called the *hash table*) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target's *home address*) is the job of the *hash function*. A *perfect hash function* would take your set of target values and map each to a unique array index.



a) If *n* is the number of items being searched and we had a perfect hash function, what is the average and worst case big-oh notation for a search?

average case $O(\ 1\ )$

worst case $O(\ 1\ )$

4. Unfortunately, perfect hash functions are a rarity, so in general many target values might get mapped to the same hash-table index, called a *collision*.

Collisions are handled by two approaches:

- *open-address* with some *rehashing* strategy: Each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.
- *chaining, closed-address*, or *external chaining*: all target values hashed to the same home address are stored in a data structure (called a *bucket*) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)