1.
(a) Explain why Pij = pPi−1,j + qPi,j−1.
The probability that India will win is p and the probability that Pakistan will win is q. So, to get the probability that India will win the series, P_{ij}, you have to multiply the probability p with the probability that India will need one less victory, $P_{i-1, j,}$ and multiply the probability q with the probability that Pakistan will need one less victory, $P_{i, j-1}$, then take their sum.

(b) What is the value of P00?
This is the probability that both countries will be tied as winners in the tournament because the 0,0 represents the probability that neither India nor Pakistan will need any more victories. This is impossible since there are 2n-1 matches which means that there aren't enough matches for the game to end in a tie. Therefore, $P_{0,0}$ is meaningless.

(c) Devise and analyze an unmemoized dynamic programming algorithm that calculates Pnn, the probability that India will win the series.

```
-----------------------------------
Function Prob(n, n)
if n==0
        return 0
end if
for I = n to 0
        return (pProb(i-1, i) + qProb(i, i-1))
end for
-----------------------------------------
```

Since this function has only one for loop iteration it has time complexity O(n)

(d) Devise and analyze analyze a memoized O(n 2)-time dynamic programming algorithm that calculates Pnn.

```
-----------------------------------------------
Function memorizedProb(n, n)
if n==0
        return 0
else
        if Pij[n] exists
                return Pij[n]
        else
                return Pij[n] = (pProb(i-1, i) + qProb(i, i-1))
        end if
end if
-------------------------------------------------
```

Since we are recursively passing the values into the Prob function which also has a for loop then this will take time complexity O(n^2).
Citations: http://jeffe.cs.illinois.edu/teaching/algorithms/2009/hwex/all-hwex.pdf
Author Jeff Erickson, Title: All HW Exams, Date: 2/1/2001

2. Problem 16-1 on pages 446–447

(a) The greedy algorithm is to always use the greatest value coins for the existing amount and to use as many of those coins as possible without exceeding the existing amount. After deducting this sum from the existing amount we use the remainder as the new existing amount and repeat the process. So we apply the best solution for the current step without regard for the optimal solution. In this case the greedy algorithm will lead to the optimal solution because locally optimal solutions will lead to globally optimal solutions.

Proof: Only use quarters until adding another would exceed the desired value n, then do the same with dimes until no more can be added and the same with nickels then pennies until the desired value n is reached. This will use the least amount of coins to reach n. Therefore, the greedy algorithm provides the optimal solution for this set of coin denomination.

(b) This coin system is canonical. So we can prove by contradiction that $\sum_{k=0}^{GR}\{c^k\} = \sum_{k=0}^{OPT}\{c^k\}$. First assume it is wrong, so we have $\sum_{i=0}^{GR}\{c^i\} > \sum_{i=0}^{OPT}\{c^i\}$ and let the i-th item, c^i, be a mistake. Removing c^i from GR(greedy) will decrease OPT(optimal) by the same amount. Therefore, c^i is not a mistake, and the value per size of OPT is equal to the value per size of GR. Therefore, our assumption that $\sum_{i=0}^{GR}\{c^i\} > \sum_{i=0}^{OPT}\{c^i\}$ is wrong, proving that $\sum_{i=0}^{GR}\{c^i\} = \sum_{i=0}^{OPT}\{c^i\}$.

(c) consider the set of coins {1,3,4} if we try to make change for 6 cents the solution from the greedy algorithm would yield 1X4+2X1 = 6 cents. The total number of coins is 3. But a better solution is 2X3 = 6 cent which is optimal and uses only 2 coins.

(d) The following function takes O(nk) time by looping through all k coin denominations n times
n = amount in cents

```
Function greedy(n)
If (n ==0)
        Return 0
For (k = 0 to n)
        k+(k+1)
End
```

i. As given on page 447, but use dynamic programming in its recursive formulation

```
recur(n)
for (k = 1 to n)
        k + recur(k+1)
end
```

ii. As given on page 447, but use dynamic programming in its iterative formulation

```
iterative(n)
for (k=1 to n )
        k+(k+1)
end
```

iii. Analyze the time required.
Both iterative and recursive take time O(nk)

(e) Suppose that, in part (d), we add the restriction that each denomination can be used just once. Modify your algorithm to determine if making change for n cents is possible.

```
Memoize(n)
If mem[n]  exists
        Return mem[n]
Else
        Return mem[n] = n + mem[n+1]
```

Citations: 1) http://www.jade-cheng.com/uh/coursework/ics-311/homework/homework-06.pdf
Author Jade Yu Cheng, Title: ICS 311 HW6, Date: Sep. 15, 2008
2) https://carlstrom.com/stanford/cs161/ps5sol.pdf
Author: Brian D. Carlstrom, Title:Problme Set #5 Solutions, Date: summer 2004
3) Recitation notes Feb. 26, 2016 Taeho Jung

3. Exercise 17.4-3 on page 471 of CLRS3.
Let c_i denote the actual cost of the ith operation, $\hat{c}_i$ its amortized cost and n_i , s_i and Φ_i the number of items stored in the table, the size of the table and the potential after the ith operation, respectively. The potential Φ_i cannot get negative values and we have $\Phi_0 = 0$. Therefore $\Phi_i \geq \Phi_0$ for all i and the total amortized cost provides an upper bound on the actual cost. Now if the ith operation is TABLE-DELETE, then $n_i = n_{i-1} - 1$. Consider first the case when the load factor does not drop below 1/3, i.e. $n_i/s_{i-1} \geq 1/3$.
Then the table isn't contracted and $s_i = s_{i-1}$. We pay $c_i = 1$ for deleting one item. Thus

$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$
$= 1 + |2n_i - s_i| - |2n_{i-1} - s_{i-1}|$
$= 1 + |2n_{i-1} - s_{i-1} - 2| - |2n_{i-1} - s_{i-1}|$ $n_i = n_{i-1} - 1$, s_i
$= s_{i-1} \leq 1 + |(2n_{i-1} - s_{i-1} - 2) - (2n_{i-1} - s_{i-1})|$ (reverse) triangle inequality
$= 1 + 2 = 3$

Now consider the case when the load factor drops below 1/3, i.e.
$n_i/s_{i-1} < 1/3 \leq n_{i-1}/s_{i-1} \Rightarrow 2n_i < 2/3 * s_{i-1} \leq 2n_i + 2.$ (1)
Now we contract the table and have
$s_i = \text{floor}(2/3 \cdot s_{i-1}) \Rightarrow 2/3 * s_{i-1} - 1 \leq s_i \leq 2\ 3\ s_{i-1}.$ (2)
By combining (1) and (2) we get
$2n_{i-1} 1 \leq s_i \leq 2n_i + 2$ and thus
$|2n_i - s_i| \leq 2.$
Furthermore, from (1) we get $s_{i-1} > 3n_i$ and thus
$|2n_{i-1} - s_{i-1}| = -2n_{i-1} + s_{i-1} \geq n_{i-1}.$
We pay $c_i = n_{i+1}$ for deleting one item and moving the remaining n_i items into the contracted table. Then, $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} - 1 = (n_{i+1}) + |2n_i - s_i| - |2n_{i-1} - s_{i-1}| \leq (n_i + 1) + 2 - n_i = 3$
In both cases the amortized cost is at most 3 and thus bounded above by a constant.

Citation: https://notebookbft.files.wordpress.com/2015/10/clrs-solution-collection.pdf Author: Cormen, Lee, Lin Title: Instructor manual, Date: unknown

4. (a)Inserting from the front will take O(1) time, because you are inserting (pushing) to the top of the Head stack, which does not require searching through anything.
inserting to the rear is the same as pushing the element to the top of the Tail stack, which also takes O(1) time.

(b) We would need to count the number of elements in the non-empty stack, then divide that by 2. Let the non-empty stack have n elements, then starting from the element n at the top of the stack to the first element at the bottom, then each element will be popped off the stack and pushed onto the temp stack.
Once done, the element that was at the bottom of the non-empty stack will now be at the top of the temp stack. Begin popping elements off the temp stack and pushing them onto the head stack until you've pushed the (n/2)th element onto it.
The rest of the elements should be popped off the temp stack and pushed onto the tail stack. This way, the contents of the non-empty stack are divided among the head and tail stacks, and should still be in their proper places at the front and rear of the deque.

(c) What is the worst-case cost of each of the four operations?

Insert to front: O(1)
Insert to rear: O(1)
Delete from front: O(1)
Delete from rear: O(1)

(d) Using the potential function we can calculate amortized time = actual time + |Head| - |Tail|, or am = ac + |Head| - |Tail|

Insert from Front: am = 1 + (n+1) – n = 2

Insert from Rear: am = 1 + n – (n+1) = 0

Delete from Front: am = 1 + (n-1) – n = 0

Delete from Rear: am = 1 + n – (n-1) = 2

Since all of the operations result in either 2 or 0 which are constants, then the amortized time is bounded above by the constant 2, which is indeed in O(1) time.

Citation: textbook chapter 17