Homework Assignment 4.

1.

(a) $P_{ij} = pP_{i-1,j} + qP_{i,j-1}$

* Consider the term

$pP_{i-1,j} \longrightarrow$ This means India will win

where $p \rightarrow$ Probability that India wins

$P_{i-1,j} \rightarrow$ j is the number of matches, that Pakistan ~~needed~~ ~~to~~ wins

$(i-1)$ is the number of matches needed by India to win.

* Similarly consider

$qP_{i,j-1} \longrightarrow$ This means India will lose

$q \rightarrow$ Probability that Pakistan wins

$P_{i,j-1} \rightarrow$ i is the number of matches India ~~needed~~ ~~to~~ wins

$(j-1)$ is the number of matches needed by Pakistan to win.

* Combining both the terms we get

$$P_{ij} = pP_{i-1,j} + qP_{i,j-1}$$

where P_{ij} is the overall probability that either India wins or loses (i.e) overall summary of the match.

(b) P_{00}

* The first subscript 0, denotes that India needs no victories. The second subscript 0 denotes that Pakistan needs no victories.

* This case where both teams needs no victories occurs only when the match becomes draw or there is a tie

* If there is a tie or draw, the value of $P_{00} = 1$

* If there is no tie or draw condition, then P_{00} cannot be determined.

(d) P_{nn} can be ~~cal~~ calculated by using the equation in ~~@~~ (1) ⓐ that is.

$$P_{ij} = pP_{i-1,j} + qP_{i,j-1}$$

This equation should be solved recursively to get the value of P_{nn}.

(2) Problem 16-1 Page 446-447

(a). * Quarters (q)

$q = \lfloor n/25 \rfloor$ quarters [$n \to$ cents]

This leaves $n_q = n \bmod 25$ cents to make change

* Dimes (d)

$d = \lfloor n_q/10 \rfloor$ dimes, which leaves

$n_d = n_q \bmod 10$ cents to make change

* Nickels

$k = \lfloor n_d/5 \rfloor$ nickels, which leaves $n_k = n_d \bmod 5$ cents to make change.

* Pennies.

$p = n_k$ pennies.

* The problem, we wish to solve is making change for n cents. If $n=0$, the optimal solution is to give no coins. If $n>0$, determine the largest coin whose value is less than or equal to n. Let this coin has value c. Give one such coin and then recursively solve the subproblem of making change for $n-c$ cents.

* We need to show that greedy choice property holds.

Consider some optimal solutions. If it includes a coin of value c, then it will be done. otherwise optimal solution does not include a coin of value c

Consider few cases

- If $1 \leq n \leq 5$, then $c = 1$

  A solution may consist only of pennies & so it must contain the greedy choice.

- If $5 \leq n < 10$, then $c = 5$

  By supposition, this optimal solution does not contain a nickel & so it consists of only pennies. Replace, 5 pennies by 1 nickel, to give a solution with 4 fewer coins.

- If $10 \leq n < 25$, then $c = 10$

  By supposition, this optimal solution does not contain a dime & so it contains only nickels & pennies.

- If $25 \leq n$, then $c = 25$

  By supposition, this optimal solution, does not contain a quarter & so it contains dimes, nickels and pennies

From Above cases, it is proved that, there is always an optimal solution that includes the greedy choice and we can combine this Choice with an optimal solution to the remaining subproblem to produce an optimal solution to our original problem.

$\therefore$ Therefore, the greedy Algorithm produces an optimal solution

(b) when the coin denominations are $c^0, c^1 \dots c^k$, the greedy algorithm to make change for n cents works by finding the denomination c^j, such that $j = \max\{0 \le i \le k : c^i \le n\}$, giving one Coin of denomination c^j & ~~recursing~~ recursing on the subproblem of making change for $n - c^j$ cents.

- To show greedy Algorithm produces an optimal solution, Start by providing a Lemma

Lemma: For $i = 0, 1 \dots k$, Let a_i be the number of coins of Denomination c^i used in an optimal solution, then the problem of making Change for n cents. Then for $i = 0, 1 \dots k-1$, we have $a_i < c$.

Proof: If $a_i \ge c$, for some $0 \le i < k$, then we can improve the solution by using one more coin of denomination c^{i+1} & c fewer coins of denomination c^i. The amount for which we make changes remains the same, but we use $c-1>0$ fewer coins.

To show that greedy solution is optimal, we show that any non-greedy solution is not optimal.

As above,

Let $j = \max\{0 \le i \le k : c^i \le n\}$, so that the greedy solution uses at least one coin of denomination c^j. Consider a non greedy solution, which must use no coins of denomination c^j or higher.

Let the non-greedy solution use a_i coins of denomination c^i, for $i = 0, 1, \ldots j-1$; thus we have

$\sum_{i=0}^{j-1} a_i c^i = n$. Since $n \ge c^j$, we have

$$\sum_{i=0}^{j-1} a_i c^i \ge c^j$$

Now suppose that the non-greedy solution is optimal. By above Lemma, $a_i \le c-1$ for $i = 0, 1, \cdots j-1$

Thus
$$\sum_{i=0}^{j-1} a_i \cdot c^i \le \sum_{i=0}^{j-1} (c-1) c^i$$
$$= (c-1) \sum_{i=0}^{j-1} c^i$$
$$= (c-1) \cdot \frac{c^j - 1}{c-1} = c^j - 1$$

$c^{j-1} < c^j$, which contradicts our earlier assertion that $\sum_{i=0}^{j-1} a_i c^i \leq c^j$

$\therefore$ We conclude, non greedy solution is not optimal. Hence greedy algorithm provides optimal solution.

(C) * With U.S. coins, we can use denomination of 1, 10 and 25. When $n = 30$ cents, the greedy solution gives one quarter and 5 pennies for a total of 6 coins. The non greedy solution of 3 dimes is better.

* The smallest integer numbers we can choose are 1, 3 & 4. When $n = 6$ cents, the greedy solution gives one 4 cent coin & two 1-cent coins, for a total of 3 coins. The non greedy solution of 2 3-cent coins is better.

(d)

* Since we have optimal substructure, dynamic programming might apply.
* Define $C[j]$ to be the minimum number of coins, we need to make change for j cents. Let the coin denominations be $d_1, d_2 \cdots d_k$. Since one of the coins is a penny. We can make

change for any amount $j \geq 1$.

* Because of optimal substructure, if we knew that an optimal solution for the problem of making change for j cents used a coin of denomination d, we would have $C[j] = 1 + C[j - d_i]$. As base case, we have $C[j] = 0$ for all $j \leq 0$.

- To develop recursive formulations, we have to check all denominations, giving,

$$C[j] = \begin{cases} 0 & \text{if } j \leq 0 \\ 1 + \min\limits_{1 \leq i \leq k} \{ C[j - d_i] \} & \text{if } j > 1 \end{cases}$$

```
CALCULATE_CHANGE (n, d, k)

for j ← 1 to n
    do C[j] ← ∞
        for i ← 1 to k
            do if j ≥ d_i and 1 + C[j - d_i] < C[j]
                then C[j] ← 1 + C[j - d_i]
                    denom[j] ← d_i
return C and denom.
```

This procedure runs in $O(nk)$ time.

```
GIVE_CHANGE (j, denom)
    if j > 0
        then give one coin of denomination denom[j]
            GIVE_CHANGE (j - denom[j], denom)
```

- Initial call is GIVE-CHANGE (n, denom). Since the value of 1^{st} parameter decreases in each recursive call, this procedure runs in $O(n)$ time.

d (i) The problem has optimal substructure property. So, we formulate a dynamic programming recursion

* Let $n[i,j]$ represent the minimum number of coins required to make a change for j cents using coins with denomination no greater than c_i ($c_0 = 1$). Then $n[0,j] = j$ &

$$n[i,j] = \min(n[i-1,j], n[i, j-c_i]+1)$$

[Either do not use coin c_i or use the minimum number of coins to make change for $j - c_i$ cents plus 1 for c_i]. We can calculate the value of $n[i,j]$ in $O(nk)$ time. To Reconstruct the values of each coin in the change set by checking whether $n[i,j] = n[i-1,j]$ or $n[i,j] = n[i, j-c_i]+1$ in $O(k+n)$ time d (ii). Consider the following piece of pseudocode where d is the array of denomination values, k is the number of denominations & n is the Amount for which change is to be made.

```
CHANGE (d, k, n)
C[0] ← 0
  for p ← 1 to n
      min ← ∞
      for i ← 1 to k
          if d[i] ≤ p, then
             if 1 + C[p - d[i]] < min, then
                min ← 1 + C[p - d[i]]
                coin ← i
          C[p] ← min
          S[p] ← coin
   return C and S.
```

d (iii) The CHANGE procedure runs in $\Theta(nk)$ due to the nested Loops & it uses $\Theta(n)$ additional space in the form of the $C[\cdot]$ & $S[\cdot]$ arrays. The MAKE-CHANGE procedure runs in $O(n)$ time, since the parameter n is decreased by atleast 1 in each pass through the while loop. It uses no additional space beyond the inputs given. ∴ Total running time is $\Theta(nk)$

and the total space requirement is $\Theta(n)$.

(e) Since each denomination, can be used just once, for each denomination ~~d~~ k in d is 1. Substituting $k = 1$ in d(ii) algorithm, will make the inner for loop running only once. Therefore the Algorithm will run in $\Theta(n)$ time.

3: Exercise 17.4-3 on Page 471 of CLRS3.

Solution:

Suppose that i^{th} operation is TABLE_DELETE

Consider the value of Load factor α:

$$\alpha = \frac{(\text{no of Entries in Table after iteration } i)}{(\text{Size of Table after iteration } i)}$$

$$= num_i / size_i$$

Case 1: if $\alpha_{i-1} = 1/2$, $\alpha_i < 1/2$

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1}$$

$$= 1 + (size_i - 2num_i) - (2num_{i-1} - size_{i-1})$$

$$= 3 + 2\, size_{i-1} - 4\alpha_{i-1}\, size_{i-1}$$

$$\leq -1 + 2\, size_{i-1} - 4/2\, size_{i-1}$$

$$= 3.$$

Case 2:

If $\frac{1}{3} \leq \alpha_{i-1} < \frac{1}{2}$, $\alpha_i \leq \frac{1}{2}$ [i^{th} operation would not cause shrinkage]

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1}$$

$$= 1 + (size_i - 2\,num_i) - (size_{i-1} - 2\,num_{i-1})$$

$$= 1 + (size_{i-1} - 2(num_{i-1} - 1)) - (size_{i-1} - 2\,num_{i-1})$$

$$= 3$$

Case 3: If $\alpha_{i-1} = \frac{1}{3}$, $\alpha_i \leq \frac{1}{2}$ [i^{th} operation would not have caused shrinkage]

$$\hat{c} = c_i + \phi_i - \phi_{i-1}$$

$$= num_i + 1 + (size_i - 2\,num_i) - (size_{i-1} - 2\,num_{i-1})$$

$$= num_{i-1} - 1 + 1 + (\tfrac{2}{3} size_{i-1} - 2(num_{i-1} - 1)) - (size_{i-1} - 2\,num_{i-1})$$

$$= num_{i-1} - \tfrac{1}{3} size_{i-1} + 2$$

$$= 2.$$

$\therefore$ Thus the Amortized cost of TABLE-DELETE is bounded by 3.

4

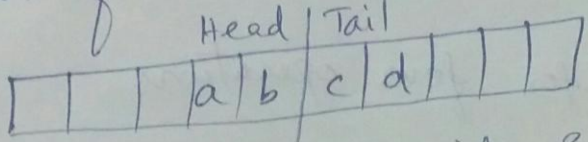
(a) The Deque has 2 stacks HEAD & TAIL.

It places these stacks back-to-back, so that operations are fast at either end.

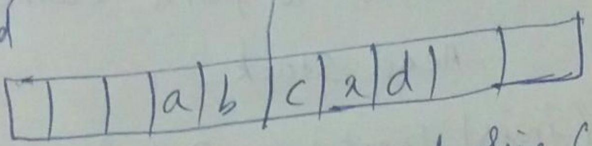
The total number of elements, n = Head.size() + Tail.size().

```
∴ int size () {
      return Head.size () + Tail.size ();
  }
```

∴ Insertion of an element (Insert Rear)

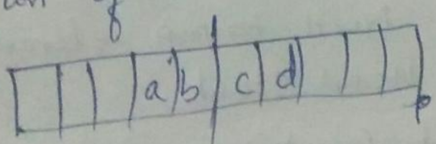


Add (3, x): where 3 is the index & 'x' is the element to be inserted

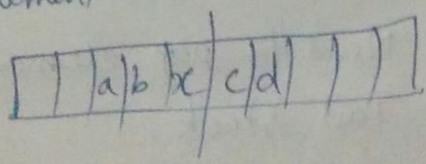


this is because if $i <$ Head.size(), then it corresponds to element of Head at position Head.size() $- i - 1$

- Insertion of an element (Insert front)



Add element x



(b) If Both Stacks, (Head & Tail) are not empty, we can extract the topmost element (pop() operation) from the head stack or Tail stack. This will be used as Front Delete & Rear Delete.

* If either of the two stacks, Head & Tail is empty, we need to split the Non Empty stack using the Temp stack and later push the elements on to the Queue.

(c). Worst case for four operations.

- Insert front:- Need to push element in Head Stack.

$$T(\text{insert Front}) = O(1)$$

- Insert Rear:- Need to push element in Tail stack

$$T(\text{Insert Rear}) = O(1)$$

- Delete front:- Need to pop element from Head Stack.

$$T(\text{Delete front}) = O(1)$$

- Delete Rear:- Need to pop element out from Tail stack.

$$T(\text{Delete Rear}) = O(1).$$

(d). Given potential function proportional to

$||Head| - |Tail||$

For Insert Front ~~& Insert Rear~~.

(i) Amortized cost = Actual cost + $\Phi_i - \Phi_{i-1}$ (Potential function)

$$\hat{c} = (Head + tail) + |(Head + 1 - \cancel{tail}) - (Head - \cancel{tail})|$$

$$\hat{c} = Head + tail + 1$$

$$\therefore \hat{c} = O(1)$$

(ii) ~~For Delete~~ For Insert Rear. [Element inserted at Tail]

$$\hat{c} = (Head + Tail) + |(Head - (tail+1)) - ((Head) - (tail))|$$

$$\therefore \hat{c} = O(1).$$

(iii) For Delete front. [Element Deleted at Head]

$$\hat{c} = (Head + Tail) + |(Head - 1) - tail| - |(Head) - tail|$$

$$\therefore \hat{c} = O(1)$$

(iv) For Delete Rear. [Element Deleted at Tail]

$$\hat{c} = (Head + tail) + ((|Head| - |Tail| - 1) - (|Head| - (Tail)))$$

$$\therefore \hat{c} = O(1)$$

$\therefore$ Amortize time for four operations is $O(1)$