ALGORITHM - ASSIGNMENT-4

KAVYA SHAMSUNDARA

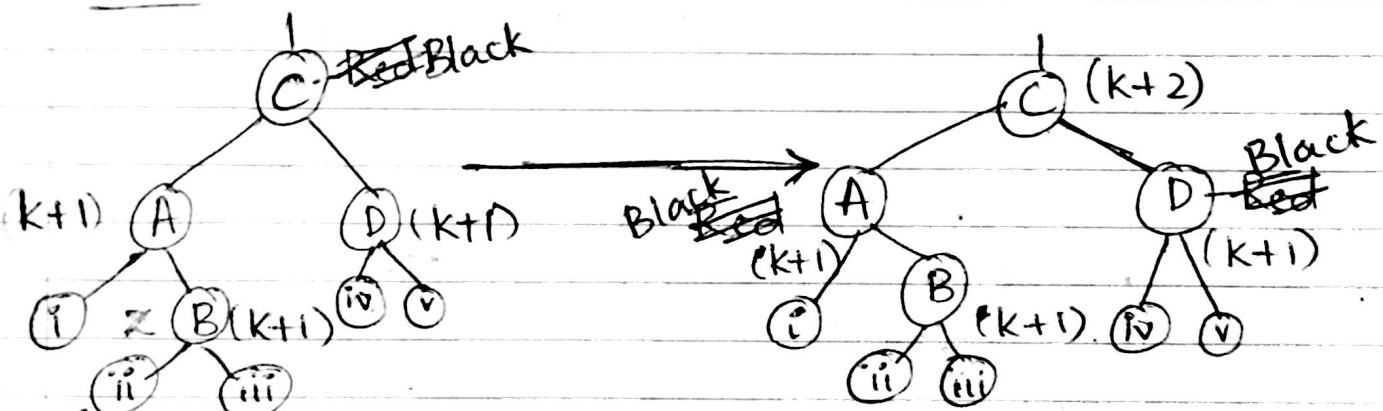
(1) (a) Can the black-height of nodes in a red-black tree be maintained as fields in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not.

Solution: +By Theorem 14.1, it is possible to maintain the black-height

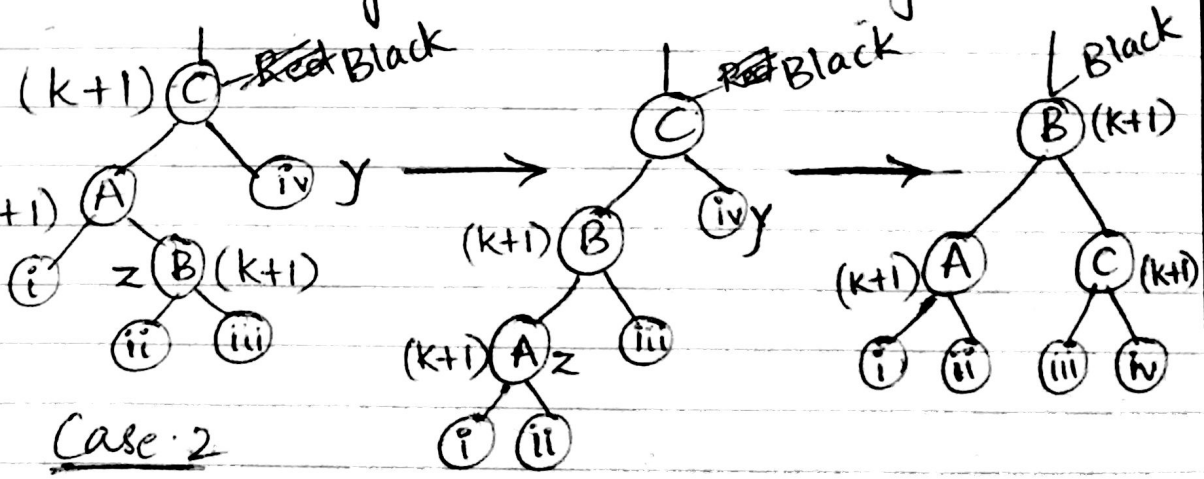
- This is because the black height can be computed from information at the node & its 2 children.
- It can also be computed from just one child's information :- black-height of a red child
  - black height of a black child plus one.
- Consider red black tree operations: RB-INSERT-FIXUP
  RB-DELETE-FIXUP

For RB-INSERT-FIXUP, consider the cases:

Case 1: z's uncle is red



Case 2: z's uncle y is black & z is a right child



Case 2

Case 3

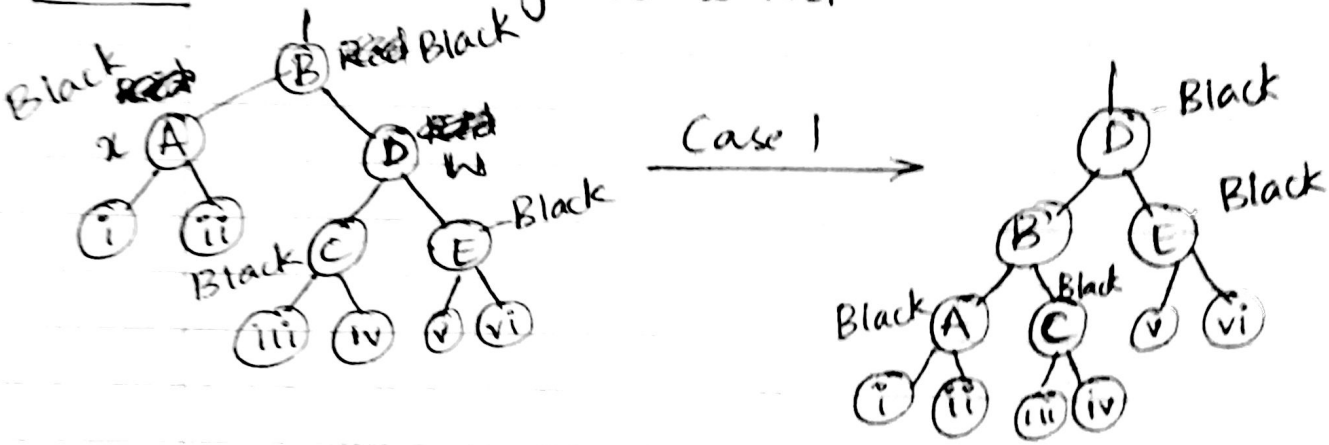
Case 3: z's uncle y is black, z is a left child.

- With subtrees (i), (ii), (iii), (iv), (v) of black height k, we see that even with color changes & rotations, the black-heights of nodes A, B & C remain the same (k+1).

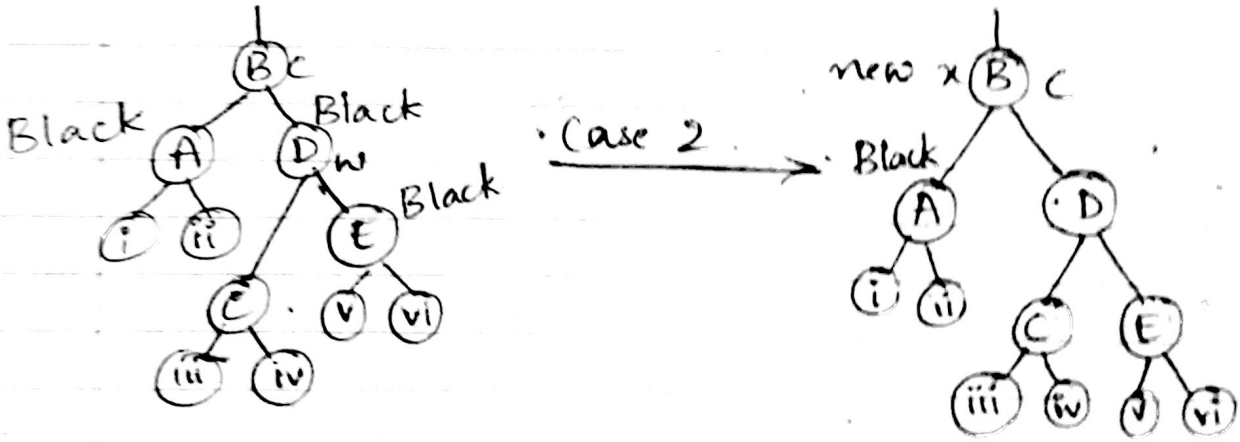
∴ the RB-INSERT-FIXUP maintains its original O(lg n) time.

* Consider RB-DELETE-FIXUP:

Case 1: x's sibling w is red



Case 2: x's sibling w is black, & both of w's children are black.



- Regardless of the color changes & rotation of the above cases, the black-heights don't change

∴ RB-DELETE-FIXUP maintains its original O(lg n) time.

* Therefore, we ~~consider~~ the black-heights of nodes can be maintained as fields in red-black trees without affecting the asymptotic performance of red-black tree operations.

(1)(b) Define the red depth of a node in a red black tree as the no of red ancestors that the node has. Can the red-depths of nodes in a red-black tree be maintained as fields in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not.

Solution:

- The red depth of any node in the red black tree is the count of the red ancestors that the node has.

-. The red-depths of nodes in a red-black tree cannot be maintained as fields in the nodes of the tree.

- This is because the depth of a node depends on the depth of its parent.
- When the depth of the node changes, the depth of all nodes below it in the tree must be updated.

Updating the root node causes n-1 other nodes to be updated, which would mean that operations on the tree that change node depths might not run in $O(n \lg n)$ time.

(2) Problem 16-1 on pages 446-447,

(a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels & pennies. Prove that your alg yields an optimal soln.

Solution:

* Quarters

- Give $q = \lfloor n/25 \rfloor$ quarters.
- This leaves $n_q = n \bmod 25$ cents to make change.

* Dimes.

- Give $d = \lfloor n_q/10 \rfloor$ dimes
- This leaves $n_d = n_q \bmod 10$ cents to make change.

* Nickels

- Give $k = \lfloor n_d/5 \rfloor$ nickels.
- This leaves $n_k = n_d \bmod 5$ cents to make change.

* Pennies

- Give $p = n_k$ pennies.

- The problem we wish to solve is making change for n cents. If $n = 0$, the optimal solut^n is to give no ~~answer~~ coins
- If $n > 0$, determine the largest coin whose value is less than or equal to n. Let this coin have value c. Give one such coin & then recursively solve the subproblem of making change for n-c cents.
- We need to show that greedy choice property holds
  Consider some optimal solution.
  If this optimal solution includes a coin of value c, then it will be done.
  Otherwise, this optimal soln doesnot include a coin of value c.
  Consides 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 doesnot 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 doesnot contain a dime, & so it contains only nickels & pennies.
- If $25 \leq n$, then $c = 25$
By supposition, this optimal solution doesnot contain a quarter, & so it contains only diners, nickels & pennies.

From above cases, it is proved that there is always an optimal solution that includes the greedy choice., & we can combine this choice with an optimal solution to the remaining subproblem to produce an optimal solution to our original problem.
∴ Therefore, the greedy algorithm produces an optimal solution.

(b) Suppose that the available coins are in the denomination that are powers of c, ie, the denominations are $c^0, c^1, \ldots c^k$ for some integers $c > 1$ & $k \geq 1$. Show that the greedy algorithm always yields an optimal solution

Solution:
- When the coin denominations are $c^0, c^1 \ldots c^k$, the greedy algorithm to make change for n cents works by finding the denomination c^j such that

$j = \max\{0 \leq i \leq k : c^i \leq n\}$, giving one coin of denomination c^j, & recursing on the subproblem of making change for $n - c^j$ cents.

- To show greedy algorithm produces an optimal solution, start by proving a lemma.

Lemma: For $i = 0, 1 \ldots k$ let a_i be the no of coins of denomination c^i used in an optimal solution to the problem of making change for n cents. Then for $i = 0, 1, \ldots k-1$, we have $a_i < c$.

Proof: If $a_i \geq c$ for some $0 \leq i < k$, 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 change remains the same, but we use $c-1 > 0$ fewer coins.

To show that greedy ~~algorithm~~ solution is optimal, we show that any non-greedy solution is not optimal. As above,

Let $j = \max\{0 \leq i \leq k : c^i \leq n\}$, 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 $n \geq c^j$, we have $\sum_{i=0}^{j-1} a_i c^i \geq c^j$

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

Thus, $\sum_{i=0}^{j-1} a_i c^i \leq \sum_{i=0}^{j-1} (c-1) c^i$

$$= (c-1) \sum_{i=0}^{j-1} c^i$$

$$= (c-1) \cdot \frac{c^j - 1}{c-1}$$

$$= c^j - 1$$

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

∴ We conclude non-greedy solution is not optimal

Hence, greedy ~~solution~~ algo provides optimal solution

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Solution:
- With actual US coins, we can use coins of denomination 1, 10 & 25. When n = 30 cents, the greedy solution gives one quarter & 5 pennies, for a total of 6 coins. The non-greedy solution of 3 dimes is better.
- The smallest integer numbers we can use 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) Give an $O(nk)$-time algo that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Solution:
- 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 $d_1\ d_2 \ldots d_k$. Since one of the coins is a penny, there is a way to 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 $c[j] = 1 + c[j-d]$. As base case, we have $c[j] = 0$ for all $j \leq 0$.
- To develop recursive formulation, we have to check all denominations, giving

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

```
COMPUTE-CHANGE(n,d,k)
for j ← 1 to n
    do c[j] ← ∞
       for i ← 1 to k
           do if j ≥ di and 1+c[j-di] < c[j]
              then c[j] ← 1+c[j-di]
                   denom[j] ← di
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 1st parameter decreases in each recursive call, this procedure runs in O(n) time.

(d) (i) Page 447, but use dynamic programming in its recursive formulation.

Solution:

- The problem has the 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 donot 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

$$n[i,j] = n[i-1,j] \text{ or } n[i,j] = n[i, j-c_i] + 1$$

in $O(k+n)$ time.

(d) (ii) Page 447, but use dynamic programming in its iterative formulation.

Solution

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) Analyse the time required.

Solution:

- The CHANGE procedure runs in $\Theta(nk)$ due to the nested loops & it uses $\Theta(n)$ additional space in the form of the $C[\cdot]$ and $S[\cdot]$ arrays.
- The MAKE CHANGE procedure runs in time $O(n)$ since the parameter 'n' is reduced by atleast 1 (the minimum coin denomination value) in each pass through the while loop.

-It uses no additional space beyond the inputs given. Thus, the total running time is $\Theta(nk)$ & the total space requirement is $\Theta(n)$.

(3)

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

* Consider the term:
  $pP_{i-1,j}$ → this means India will win.

where p → probability that India wins.
$P_{i-1,j}$ → j is the number of matches that Pak win
$(i-1)$ is the number of matches needed by India to win

* Similarly consider
  $qP_{i,j-1}$ → This means India will lose.

where q → probability that Pakistan wins
$P_{i,j-1}$ → i is the number of matches that India 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.
ie., the overall summary of the match.

(3)(b) P_{00}.

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

* This case where both the team needs no victories occur only when the match becomes draw. or there is a tie.

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

* If there is tie or draw condition, then P_{00} cannot be determined.

(3)(c) P_{nn} can be calculated by using the equation in (3)@ 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}.