COT 5407: Introduction to Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/5407S19.html

Approach to DP Problems

- Write down a recursive solution
- Use recursive solution to identify list of **subproblems** to solve (there must be overlapping subproblems for effective DP)
- Decide a data structure to store solutions to subproblems (**MEMOIZATION**)
- Write down **Recurrence relation** for solutions of subproblems
- Identify a **hierarchy/order** for subproblems
- Write down non-recursive solution/algorithm

DP Problems

- Find a recursive solution
  - For what purpose?
  - To **reduce** the problem to one or more **simpler** problems
    - reduce the size of the input by imposing conditions
    - e.g., if we know something about last item in input or
    - e.g., if we know how to break up the problem/solution

Because of "Optimal Substructure Property"

Car removal problem

1. Either the last one is removed …
   - We now have a subproblem with only N-1 cars.
     - Problem with cars 1, 2, … N-1
2. Or it stays …
   - We retain last car, and get a constrained subproblem as we know that the second to last must match last car.
     - Problem with cars 1, 2, … K where K is last car matching car N

List of Subproblems

- This will become clear if we follow the recursion one or two more steps
- In this case:
  - Problems on cars 1, 2, …, k for different values of k

List of Subproblems

May be refined later

- The inputs to the subproblems are:

$L_1 = \{c_1\}$

$L_2 = \{c_1, c_2\}$

$L_3 = \{c_1, c_2, c_3\}$,

…,

L_n = set of all cars

- Memoization is thus obvious:

A[1] = solution to L_1

A[2] = solution to L_2

A[3] = solution to L_3

…

A[n] = solution to L_n

A[j] = least number of cars to be removed when the input is L_j

Recurrence Relation for A[j]

Incorrect Solution

1. Either car j is removed …
   - We now have a subproblem with only j-1 cars.
     - Problem with cars 1, 2, … j-1
     - A[j] = 1 + A[j-1]
2. Or it stays …
   - We retain last car, and get a constrained subproblem as we know that the second to last must match last car.
     - Problem with cars 1, 2, … K where K is last car matching car j
     - A[j] = (j-K-1) + A[K]
     - A[j] = (j-K-1) + A[K] → $A[j] = \min_K \{ (j-K-1) + A[K] \}$

Why is the solution incorrect?

- We don't know whether A[j] refers to a solution that includes car j or not. This will dictate what car can be appended at the end of the solution to this subproblem
- For e.g., if input is
  - (1,2), (2,3), (3, 4), (2,5), (5,6), (6,7)

Minor change in Memoization

- A[j] = least number of cars to be removed when the input is L_j and car j is included
- B[j] = least number of cars to be removed when the input is L_j and car j is not included

Recurrence Relation for A[j], B[j]

1. Either car j is removed …
   - We now have a subproblem with only j-1 cars.
     - Problem with cars 1, 2, … j-1
     - B[j] = 1 + min{ A[j-1], B[j-1] }
2. Or it stays …
   - We retain last car, and get a constrained subproblem as we know that the second to last must match last car.
     - Problem with cars 1, 2, … K where K is last car matching car j
     - A[j] = min{ (j-K-1) + A[K] }

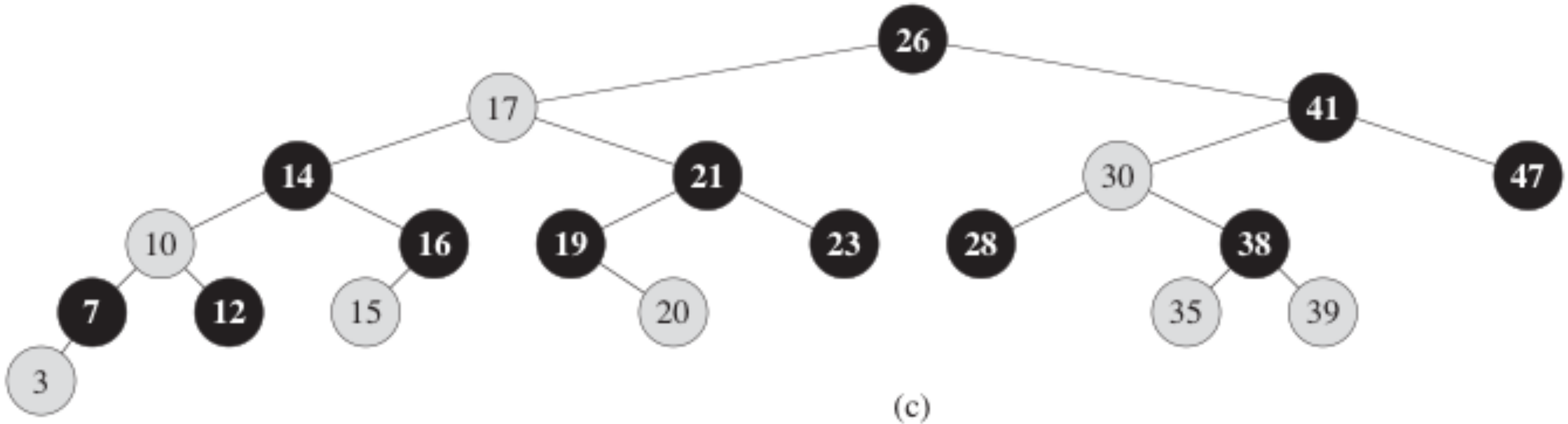
What to return?

- Min { A[n], B[n] }

Time Complexity

- $O(n^2)$

RB-Trees



(c)

OS-Rank

```
OS-RANK(x,y)
// Different from text (recursive version)
// Find the rank of x in the subtree rooted at y
  r = size[left[y]] + 1
  if x = y then return r
  else if ( key[x] < key[y] ) then
      return OS-RANK(x,left[y])
  else return r + OS-RANK(x,right[y] )
```

Time Complexity O(log n)

How to augment data structures

1. choose an underlying data structure
2. determine additional information to be maintained in the underlying data structure,
3. develop new operations,
4. verify that the additional information can be maintained for the modifying operations on the underlying data structure.

Augmenting RB-Trees

Theorem 14.1, page 309

Let f be a field that augments a red-black tree T with n nodes, and f(x) can be computed using only the information in nodes x, left[x], and right[x], including f[left[x]] and f[right[x]].

Then, we can maintain f(x) during insertion and deletion without asymptotically affecting the O(log n) performance of these operations.

For example,

size[x] = size[left[x]] + size[right[x]] + 1

rank[x] = ?

Rank cannot be maintained because of this theorem.

Augmenting information for RB-Trees

- Parent
- Height
- Any associative function on all previous values or all succeeding values.
- Next
- Previous

Augmented Info

- OddSize[v]
  - Number of odd valued nodes in subtree rooted at v
- It can be maintained because:
  - OddSize[v] =
    OddSize[Left[v]]
    + OddSize[Right[v]]
    + (key[v] % 2)

OS-SoOdd

```
OS-SoOdd(x,y)
// Different from text (recursive version)
// Find the rank of x in the subtree rooted at y
  r = OddSize[left[y]] + key[x] % 2
  if x = y then return r
 else if ( key[x] < key[y] ) then
     return OS-SoOdd (x, left[y])
 else return r + OS-SoOdd (x, right[y])
```

Time Complexity O(log n)

More Dynamic Operations

	Search	Insert	Delete	Comments
Unsorted Arrays	O(N)	O(1)	O(N)	
Sorted Arrays	O(log N)	O(N)	O(N)	
Unsorted Linked Lists	O(N)	O(1)	O(N)	
Sorted Linked Lists	O(N)	O(N)	O(N)	
Binary Search Trees	O(H)	O(H)	O(H)	H = O(N)
Balanced BSTs	O(log N)	O(log N)	O(log N)	As H = O(log N)

	Se/In/De	Rank	Select	Comments
Balanced BSTs	O(log N)	O(N)	O(N)	
Augmented BBSTs	O(log N)	O(log N)	O(log N)	