Assignment 3

Virpobe Paireepinart

Algorithm Design and Analysis, Fall 2009, Texas State University

R 3.7 Chapter 3, page 212   Explain why performing a rotation in an n-node binary tree represented using a sequence takes Omega(n) time.

When a binary tree is represented as a sequence, it is represented linearly. Typically there is an equation that can be used to perform in-order traversal on the sequence without having to seek at all.

Omega(n) is the largest that the height of a binary tree with "n" nodes can be. Because of the way that the sequences are laid out, a rotation only takes constant time. The elements who are in in-order traversal with each other can be accessed in constant time by just offsetting current location. So the time it takes to perform a rotation is Omega(n) to traverse the tree and find the location to rotate and then a constant factor to perform the rotation.

R 5.11 Chapter 5, page 282   Let S = {a, b, c, d, e, f, g} be a collection of objects with benefit-weight values as follows: a: (12, 4), b: (10, 6), c: (8, 5), d: (11, 7), e: (14, 3), f: (7, 1), g: (9, 6). What is an optimal solution to the 0/1 knapsack problem for S assuming we have a sack that can hold objects with total weight 18? Show your work.

First we initialize our list to be all 0's.

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Then we bring K from 1 to 7.

When k is 1, we need to calculate for w from 18 to 4 (4 is the weight of our first item).

We know that B[w] is going to be 0, and B[w-wk] + bk will be 0 + 12 (the benefit of the first item), so all items from 4 to 18 (inclusive) will be updated to 12.

[0, 0, 0, 0, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12]

For the next iteration, K is 2, which is still less than or equal to 7, so we keep going...

We need to calculate for w from 18 to 6. For w from 18 to 11, we will be indexing back 6 items and we will still get 12's. after that we will start getting the 0's. (because 11 – 6 = 5. at 10, 10-6 = 4, which is where the 0's start.) so from 18 to 1 we will have 12 + 10 = 22, and past that we will start indexing the 0's and we will get 0 + 10 = 10. 10 is less than 12 so those are left as 12's.

[0, 0, 0, 0, 12, 12, 12, 12, 12, 12, 22, 22, 22, 22, 22, 22, 22, 22]

now K is 3, which is still less than or equal to 7, so we keep going...

We need to calculate for w from 18 to 5

From w = 18 to 16 we will get 22 + 8 = 30. From w = 15 to 10 we will get 12 + 8 = 20. All these locations except w=10 are 22 already so they won't be changed, B[10] will be changed from 12 to 20. From w = 9 to 5 we will get 0 + 8 = 8, which is less than 12 so we preserve 12 at these locations.

[0, 0, 0, 0, 12, 12, 12, 12, 12, 20, 22, 22, 22, 22, 22, 30, 30, 30]

now K is 4, which is still less than or equal to 7, so we keep going...

We need to calculate for w from 18 to 7

at w = 18 we will get 22 + 11 = 33. at w = 17 we will get 20 + 11 = 31. at w = 16 through 12, we will get 12 + 11 = 23. At w = 11 through 7 we will get 0 + 11 = 11. 11 is less than 12 and 20 and 22 so nothing in w=11 to 7 will be updated.

[0, 0, 0, 0, 12, 12, 12, 12, 12, 20, 22, 23, 23, 23, 23, 30, 31, 33]

now K is 5, which is still less than or equal to 7, so we keep going...

We need to calculate for w from 18 to 4. From w = 18 to w = 15 it will be getting 23's so 23 + 14 = 37 which is bigger so they all will be updated. w=15 will be 22 + 14 = 36, w=14 will be 20+14 = 36, which is bigger than the 23 at that location so it will be updated too. So will w=13, 20+14 = 34 > 23. from w = 12 to 8 we will get 12 + 14 = 26 which will be updated for all of them. from w = 7 to 4 we will get w = 0 + 14 and they will all be updated as well.

[0, 0, 0, 14, 14, 14, 14, 26, 26, 26, 26, 26, 34, 36, 37, 37, 37, 37]

now K is 6, which is still less than or equal to 7, so we keep going...

We need to calculate for w from 18 to 1.

at w = 18 thru 16, 37 + 7 = 44 which will update w. at w=15, 36 + 7 = 43, which updates. at w=14, 34 + 7 = 41, which updates. at w = 13 thru 9, 26 + 7 = 33, which updates all except w=13. At w=8 thru 5 we get 14 + 7 = 21 which updates all except for w=8. then w=4 to 1 we get 0 + 7 = 7 which updates all but w=4. Our result after these updates is:

[0, 7, 7, 14, 21, 21, 21, 26, 33, 33, 33, 33, 34, 41, 43, 44, 44, 44]

now K is 7 (finally!)

We need to calculate for w from 18 to 6. If you look at the previous list, we know that our benefit we're adding is 9. Well every item 4 before the end item is more than 9 from the end so none of our items are going to get updated.

Now k is > 7 so we need to stop.

Our final resulting array is:

[0, 7, 7, 14, 21, 21, 21, 26, 33, 33, 33, 33, 34, 41, 43, 44, 44, 44]

Our final result is that 44 is the best benefit we can have. The algorithm we followed does not relate which items are the ones that get our optimal solution. I can tell by looking that items a: (12, 4), b: (10, 6), c: (8, 5), and e: (14, 3) add up to a benefit of 44 and a weight of 18. If we wanted the algorithm to keep track of the chosen elements we would have to modify it.

R 5.9 Chapter 5, page 282 What is the best way to multiply a chain of matrices with dimensions that are 10x5, 5x2, 2x20, 20x12, 12x4, and 4x60? Show your work.

Define sequence S as the matrices 1x5, 5x2, 2x20, 20x12, 12x4 and 4x60.

for I = 1 to 6,

set Ni,i = 0.

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

… ☹ …

C-5.5 Chapter 5, page 283

In the **art gallery guarding** problem we are given a line L that represents a long hallway in an art gallery. We are also given a set X = {x0, x1, …, xn-1} of real numbers that specify the positions of paintings in this hallway. Suppose that a single guard can protect all the paintings within distance at most 1 of his or her position (on both sides). Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings with positions in X.

Suppose you have paintings such as

[0, 2, 4.3, 21, .4, 3, 2.3, 2.5]

The first thing you would want to do is sort the paintings (assuming the order is not given to you sorted.)

[0, 0.4, 2, 2.3, 2.5, 3, 4.3, 21]

::Algorithm::

Then placement of the first item is obvious, it should be items[0] + 1. The rationale is that, because we know the paintings are sorted, the optimal distance from the first painting is for it to be fully to the left of the first guard (none of a guard's cover area should extend beyond the left boundary or it's useless. After that, we just place a guard at i+1 anytime we encounter a painting that is unguarded.

::End of algorithm::

Therefore, you can immediately place a guard at location 1.

So our current_guard = 1.

Now we start at the second item of the list.

0.4 is covered by the guard at position 1 because 0 <= 0.4 <= 2.

2 is covered by the guard at position 1 as well because 0 <= 2 <= 2.

2.3 is not covered by a guard, so we add a guard at 2.3 + 1 or 3.3 (we KNOW there are no items between 2 and 2.3 so there is no reason for this area to be covered by a guard.).

now we have a guard at 3.3 so current_guard = 3.3.

2.3 <= 2.5 <= 4.3, so 2.5 is covered.

2.3 <= 3 <= 4.3, so 3 is covered.

2.3 <= 4.3 <= 4.3, so 4.3 is covered.

Then we move to 21. It is not covered so we add a guard at 22.

::Logical argument::

This is optimal because it makes the assumption that the items are in order, and every time it places an item it places it at the optimal location.

envision on a number line

[0----1--@--2----3----4]

This item at 1.5 is run into first. If we place the guard directly on top of the item, our coverage area is

[0--[--1--@--2--]--3--4]

However, we know that there are no items between 0.5 and 1.5 because it is our first item encountered (and the items are in order). Therefore it follows that the optimal position for this guard would be at location 1.5 + 1. That way there is no wasted coverage area.

For all subsequent guards this is true as well. If the item is not already covered by a guard, then there needs to be another guard at i+1. That makes the guard's distributions optimal.