Kristin Schaefer

HW2 : Recursion, Recurrence Relations and Divide & Conquer

Sources:

Problem 1

1. https://stackoverflow.com/questions/306316/determine-if-two-rectangles-overlap-each-other

Problem 2

1. https://www.youtube.com/watch?v=eijmg5iscl8

Problem 3

1. https://stackoverflow.com/questions/57789350/divide-and-conquer-find-the-majority-of-element-in-array#57789505

2. https://github.com/antmarakis/Algorithms/blob/master/Divide%20and%20Conquer/FindMajorityElement.py

3. http://users.eecs.northwestern.edu/~dda902/336/hw4-sol.pdf

4. http://anh.cs.luc.edu/363/handouts/MajorityProblem.pdf

1. 2D boxes

I did not come up with a recursive solution to test the intersection of two rectangles.

The non-recursive solution I found consists of 4 statements:

```
doBoxesOverlap(box1, box2)
if (box1_x1 < box2_x2) and (box2_x1 < box1_x2) and (box1_y1 < box2_y2) and (box2_y1 < box1_y2):
        return True
else:
        return False
```

I tried to reconfigure the above pseudocode in two ways to make a recursive function. Both methods proved unsuccessful.

Method 1: Test smaller scale versions of box1 and box2.

This method did not make sense as the test for the original scale rectangles already yields the result.

Method 2: As box1 and box2 are both lists composed of 4 points [x1, y1, x2, y2], try comparing the last values in the arrays recursively for n times.

This method did not work, as the values to be compared are not in sequential order in the arrays.

(continue to next page)

```
doBoxesOverlap(box1, box2)
if (box1_x1 < box2_x2) and (box2_x1 < box1_x2) and (box1_y1 < box2_y2) and (box2_y1 < box1_y2):
        return True
else:
        return False
```

where:

box1 = [box1_x1, box1_y1, box1_x2, box1_y2] =>

box1_x1	box1_y1	box1_x2	box1_y2
0	1	2	3

box2 = [box2_x1, box2_y1, box2_x2, box2_y2] =>

box2_x1	box2_y1	box2_x2	box2_y2
0	1	2	3

plugging this into the pseudocode:

```
doBoxesOverlap(box1, box2)
if (box1[0] < box2[2] and (box2[0] < box1[2]) and (box1[1] < box2[3]) and (box2[1] < box1[3]):
        return True
else:
        return False
```

with the indices rewritten in terms of n where box1[0...n] and box2 [0...n] :

```
doBoxesOverlap(box1, box2)
if (box1[n-3] < box2[n-1] and (box2[n-3] < box1[n-1]) and (box1[n-2] < box2[n]) and (box2[n-2] < box1[n]):
        return True
else:
        return False
```

There isn’t a clear sequential way to test the four conditions by decrementing the index of each array for each recursive call. Even if the conditions are rearranged, the conditions still require flipping of the equality sign or alternating box1 and box2 as input 1 and 2 for each recursive call of the doBoxesOverlap() function. Thus, writing this function recursively would prove to be too unclear and confusing.

Kristin Schaefer

HW2 : Recursion, Recurrence Relations and Divide & Conquer

2a. Recurrence relation for power2(x,n)

```
power2(x,n):
    if n==0:                                          constant time
        return 1
    if n==1:                                          constant time
        return x
    if (n%2)==0:
        return power2(x, n/2) * power2(x,n/2)
    else:                                             2T(n/2)
        return power2(x, n/2) * power2(x,n/2) * x
```

recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n) = 2T(n/2) + \Theta(1) & n > 1 \end{cases} \quad \Rightarrow \quad T(n) = \begin{cases} c & n = 1 \\ T(n) = 2T(n/2) + c & n > 1 \end{cases}$$

(where c is some constant)

substitution method:

1. $T(n) = 2T(n/2) + 1$

   $T(n/2) = 2T(n/4) + 1$

   $T(n) = 2[2T(n/4) + 1] + 1$

2. $T(n) = 2^2T(n/4) + 2 + 1$

   $T(n/4) = 2T(n/8) + 1$

   $T(n) = 2^2[2T(n/8) + 1] + 2 + 1$

3. $T(n) = 2^3T(n/8) + 2^2 + 2 + 1$

k^{th} term $T(n) = 2^kT(n/2^k) + 2^{k-1} + 2^{k-2} + ... + 2^1 + 2^0$

$T(1) = 1$ $\quad$ $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$2^0 + 2^1 + 2^2 + ... + 2^{k-1} + 2^k = 2^{k+1} - 1 \quad \Rightarrow \quad 2^0 + 2^1 + 2^2 + ... + 2^{k-1} = 2^{k+1} - 2^k - 1$
$= 2^k - 1$

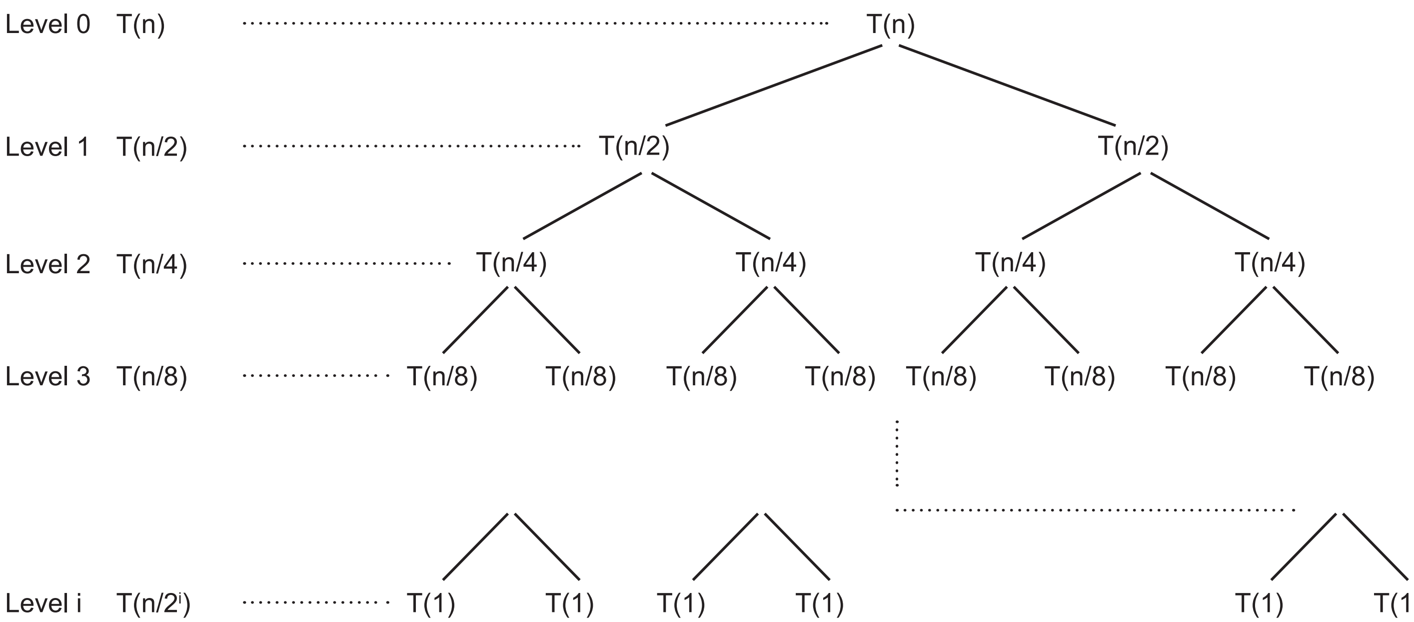
$T(n) = n \cdot T(1) + 2^k - 1$
$T(n) = n \cdot 1 + n -1$
$T(n) = 2n -1$

$T(n) = \Theta(n)$

Kristin Schaefer

HW2 : Recursion, Recurrence Relations and Divide & Conquer

recursion tree method:



$i = \log_2 n$

$$\sum_{i=0}^{\log_2 n + 1} 2^i \quad \Rightarrow s_n = \frac{a(1-r^n)}{(1-r)} = \frac{1(1-2^{\log(n)+1})}{(1-2)} = 2n-1$$

$\Rightarrow\ T(n) = \Theta(n)$

master theorem method:

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n) > 0$

$T(n) = 2T(n/2) + 1 \quad \Rightarrow n^{\log_b a} = n^1 = n$
$a = 2$, $b = 2$

$f(n) = 1$, so f(n) grows polynomially slower than $n^{\log_b a}$, which indicates Case 1

The solution for the recurrence relation is then:
$T(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n)$

2b. Asymptotic bounds for T(n)

a)

master theorem method:

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n) > 0$

T(n) = 4T(n/2) + n => $n^{\log_b a} = n^{\log_2 4} = n^2$
$a = 4$, $b = 2$, $f(n) = n$

$f(n) = n$, so f(n) grows polynomially slower than $n^{\log_b a}$, which indicates Case 1

The solution for the recurrence relation is then:
$T(n) = \Theta(n^{\log_b a})$ => $T(n) = \Theta(n^2)$

b)

master theorem method:

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n) > 0$

$T(n) = 2T(n/4) + n^2$ => $n^{\log_b a} = n^{\log_4 2} = n^{1/2}$
$a = 2$, $b = 4$, $f(n) = n^2$

$f(n) = n^2$, so f(n) grows polynomially faster than $n^{\log_b a}$, which indicates Case 3

The solution for the recurrence relation is then:
$T(n) = \Theta(f(n))$ => $T(n) = \Theta(n^2)$

Kristin Schaefer

HW2 : Recursion, Recurrence Relations and Divide & Conquer

3. Divide and Conquer

a)

Pseudocode:

```
MajorityBirthdays(days[1...n]):
    if n = 1:
        return days[1]
    m = n // 2
    days_left = MajorityBirthdays(days[1...m])
    days_right = MajorityBirthdays(days[m+1...n])
    if days_left = days_right:
        return days_left
    left_count = GetCount(days[0...n], days_left)
    right_count = GetCount(days[0...n], days_right)
    if left_count > m:
        return days_left
    else if right_count > m:
        return days_right
    else:
        return None
```

Helper function:

```
GetCount(days, day_index):
    sum = 0
    for day in days:
        if day = day_index
        sum = sum + 1
    return sum
```

b)

Proof of correctness:

Note:

- An element is a majority element of the array days[1...n] if the element occurs more than half of the length of days[1...n] times.

Recursion invariant: At each recursive call, MajorityBirthdays(days) returns the majority element of two subarrays days[1...n/2] and days[(n/2)+1...n], if a majority element exists. If no majority element is found, the recursive call returns None.

Initialization (Base Case): When n = 1, MajorityBirthdays(days) returns days[1], the single element of the array, which is correct.

Maintenance (Inductive Case):

Assume that the recursion invariant holds true for k recursive calls, so days[k] holds the majority element of A[1...k/2] and A[(k/2)+1...k] subarrays. In the $(k+1)^{th}$ call if the subarrays days[1...(k+1)/2] and days[(k+2)/2...k+1] contain an element with a greater count than the majority element of days[k] then the majority element of days[k+1] becomes the majority element.

Thus, the maintenance step holds.

Termination: At the top level of the recursive call, MajorityBirthdays(days) returns the majority element of days[1...n] (if a majority element exists), which is the solution to the problem. If no majority element is found in days[1...n], MajorityBirthdays(days) returns None.