CS 325 - HW 2 Solution

Problem 1: **(3 pts) pse**Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight as possible and justify your answers. **(1 pts each for correct answer)**

a) $\boldsymbol{T(n) = T(n-2) + n}$

$= n + (n-2) + T(n-4)$

$= n + (n-2) + (n-4) + T(n-6)$

$= n + (n-2) + (n-4) + (n-6) + T(n-8)$

….

$= ¼(n(n+2))$

$= \Theta(n^2)$

Or use the Muster Method

b) $\boldsymbol{T(n) = 3T(n-1) + 1}$

a = 3, b = 1, f(n) = $\Theta(n^0)$ so d = 0

$T(n) = \Theta(3^n)$

Using the Muster Method

c) $\boldsymbol{T(n) = 2T\left(\frac{n}{8}\right) + 4n^2}$

a = 2, b = 8, $\log_8(2) = 0.33$ $n^{0.33}$,f(n) = $4n^2$

$n^{0.33}$ = O(f(n) = $4n^2$)

Case 3:

2f(n/8) = 2*$4n^2/8^2$= $n^2/8$ <= $4cn^2$ *which is true* for c = ½ or for any 1/16 < c < 1.

So T(n) = $\Theta(4n^2)$ or $\Theta(n^2)$

CS 325 - HW 2 Solution

Problem 2: Consider the following algorithm for sorting.

```
STOOGESORT(A[0 … n - 1])
    if n = 2 and A[0] > A[1]
        swap A[0] and A[1]
    else if n > 2
        k = ceiling(2n/3)
        STOOGESORT(A[0 … k - 1])
        STOOGESORT(A[n - k … n - 1])
        STOOGESORT(A[0 … k - 1])
```

a) **(1pt)** Explain why the STOOGESORT algorithm sorts its input. (This is not a formal proof).

The base case either has one element or two elements, which are correctly sorted. The three recursive calls overlap by >n/3 elements (by the rounding-up choice). Call these elements the overlap elements, the first n/3 elements the prefix elements and the last n/3 elements the suffix elements. After the first recursive call, the prefix elements are smaller than the overlap elements. After the second recursive call the overlap elements are smaller than the suffix elements; the suffix elements are sorted. So the prefix elements are smaller than the suffix elements. The final recursive call sorts the prefix and overlap elements.

b) **(1pt)** Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)

No. A counterexample should be given. For example, consider the input list [0 3 1 2]. n = 4 and $\lfloor 2n/3 \rfloor = 2$ so A[0 … k − 1] = [0 3] which does not change in the recursive call and A[n− k…n− 1] = [1 2] does not change in the recursive call. The list does not get sorted.

c) **(1pt)**

$T(n)=3T(2n/3) + \Theta(1)$ or $T(n)=3T(2n/3) + c$ or $T(n)=3T(2n/3) + 3$

d) **(1pt)**

Master method a=3, b = 3/2, $\log_{(3/2)}3 = 2.71$

$f(n)=c = O(n^{2.71})$

$T(n) = \Theta(n^{\log_{3/2} 3})$ or $\Theta(n^{2.71})$

Problem 3:

a) Verbal description & Pseudocode (3pts)

```
function quatSearch (A, key, start, end)
    if (start > end)
        return false;
    q = INT(end-start)/4
    q1= start + q;
    q2 = start + 2q;
    q3 = start + 3q;

    if (A[q1] == key)
        return true;
    else if (A[q2] == key)
        return true;
    else if (A[q3] == key)
```

```
        return true;
    else if (key < A[q1])
        return quatSearch (A, key, start, q1-1);
    else if (key > A[q3])
        return quatSearch (A, key, q3+1, end);
    else if (key > A[q2])
        return quatSearch (A, key, q2+1,q3-1);
    else
        return quatSearch (A, key, q1+1,q2-1);
```

b) Recurrence: (2pt) **Correct if matched algorithm even if algorithm is wrong**

$T(n) = T(n/4) + 3$ or $T(n) = T(n/4) + 3k$ or $T(n) = T(n/4) + k$ or $T(n) = T(n/4) + \Theta(1)$ (1 pt)

c) Solution : (1 pt)
Using the master method

$a=1$, $b=4$, $\log_4 1=0$, $n^0 = 1$
compare $f(n) = \Theta(1)$ so case 2.
$T(n) = \Theta(\lg n)$ (1 pt)

d) Comparison – not graded
Both Binary and Quaternary search are $\Theta(\log n)$ (or $\Theta(\lg n)$).

Problem 4: Design and analyze a divide and conquer algorithm that determines the minimum and maximum value in an unsorted list (array). Write pseudo-code for the min_and_max algorithm, give the recurrence and determine the asymptotic complexity of the algorithm. Compare the running time of the recursive min_and_max algorithm to that of an iterative algorithm for finding the minimum and maximum values of an array.

a) Verbal description & Pseudocode (3pts)

MIN-MAX(A)

```
If |A|=1 then return min=max=A[0]
Divide A into two equal subsets A1 and A2

$(min_1, max_1)$ = MIN-MAX($A_1$)
$(min_2, max_2)$ = MIN-MAX($A_2$)

If $min_1 <= min_2$ then min = $min_1$
        Else min = $min_2$
If $max_1 >= max_2$ then max = $max_1$
        Else max = $max_2$
Return (min,max)
```

b) Recurrence (2 pts) **Correct if matched algorithm even if algorithm is wrong**
$T(n) = 2T(n/2) + 2$ **or** $T(n) = 2T(n/2) + c$ (1pt)

c) Solution (1 pt) : $T(n)$ is $\Theta(n)$ (1pt)

d) Iterative is also $\Theta(n)$ **not graded**

CS 325 - HW 2 Solution

Problem 5: Majority Element **Sample solution answers may vary**

A Θ(n log n) running time divide and conquer algorithm:

a) Verbal Description (2 pt) Note the elements cannot be sorted so you cannot use merge sort

The algorithm begins by splitting the array in half repeatedly and calling itself on each half. This is similar to what is done in the merge sort algorithm. When we get down to single elements, that single element is returned as the majority of its (1-element) array. At every other level, it will get return values from its two recursive calls.

The key to this algorithm is the fact that if there is a majority element in the combined array, then that element must be the majority element in either the left half of the array, or in the right half of the array. There are 4 scenarios.

i. Both return "no majority." Then neither half of the array has a majority element, and the combined array cannot have a majority element. Therefore, the call returns "no majority."
ii. The right side is a majority, and the left isn't. The only possible majority for this level is with the value that formed a majority on the right half, therefore, just compare every element in the combined array and count the number of elements that are equal to this value. If it is a majority element then return that element, else return "no majority."
iii. Same as above, but with the left returning a majority, and the right returning "no majority."
iv. Both sub-calls return a majority element. Count the number of elements equal to both of the candidates for majority element. If either is a majority element in the combined array, then return it. Otherwise, return "no majority."

The top level simply returns either a majority element or that no majority element exists in the same way.

b) Pseudocode (2pt) There may be multiple right answers

```
GetMajorityElement(A[1...n]) {
// Input: Array A of objects
// Output: Majority element of a
    if n = 1{
        return A[1]
    }
    k = floor(n/2)
    left = GetMajorityElement(A[1...k])
    right = GetMajorityElement(A[k+1...n]
    if left = right {
        return left
    }
    lcount = GetFrequency(A[1...n],left)
    rcount = GetFrequency(A[1...n],right)
    if lcount > k+1 {
        return left
    }else if rcount > k+1{
        return right
    }else
        return NO-MAJORITY-ELEMENT
}
```

CS 325 - HW 2 Solution

GetFrequency computes the number of times an element (left or right) appears in the given array A[1...n]. Two calls to GetFrequency is $\Theta(n)$. After that comparisons are done to validate the existence of majority element. GetFrequency is the linear time equality operation.

c) Recurrence: (2pt) **Correct if matched algorithm even if algorithm is wrong**

$T(n) = 2T(n/2) + 2n$ or $T(n) = 2T(n/2) + cn$ or $T(n) = 2T(n/2) + \Theta(n)$

d) Solution: (1pt)

By Case 2 of the Master Theorem: $T(n)$ is $\Theta(n\log n)$