Jonathan Nicolosi

HW2

* 1. T(n) = T(n-2) + 4

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

T(n) = T(n-6) + 12

T(n) = T(n-8) + 16

T(n) = T(n-k) + 2k

Replace k with (n-2)

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

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

T(n) = f(n)

* 1. T(n) = 3T(n-1) + 3

Or **31T(n-1) + 31**

T(n) = 3[3T(n-2)+3] + 3

T(n) = [9T(n-2)] + 9 + 3

T(n) = [9T(n-2)] + 12

Or **32T(n-2) + 31 + 32**

T(n) = 3[9T(n-3) + 12] + 3

T(n) = 27T(n-3) + 39

Or

T(n) = **33T(n-3) + 31 + 32 + 33**

T(n) = f(3n)

* 1. T(n) = 2T() + 4n2

By Master Theorem:

a = 2, b = 8, c = 2

When logba c, T(n) = (nc)

Therefore, T(n) = (n2)

* 1. T(n) = 5T(n/2) + O(n)

a = 5, b = 2, d = 1

d < log(a) = 1 < log2(5)

So O() = O(n2.3219)

* 1. T(n) = 2T(n-1) + O(1)

T(2) = 2T(1)

T(3) = 2T(2) = 2\*2\*T(1)

T(5) = 2\*T(4) = 2\*2\*2\*2\*T(1)

So O(2n)

* 1. T(n) = 9T(n/3) + O(n2)

a = 9, b = 3, d = 2

d = log2a 2 = log39

So O(n2log(n))

I would choose Algorithm A because it grows the slowest.

* 1. We know STOOGESORT sorts its input because we can see that there is a swap based on a comparison between A[0] and A[1].
  2. STOOGESORT would not sort correctly if we replaced k = ceiling(2n/3) with m = floor(2n/3). Depending on what value of n you have, you might divide the array too many times.
  3. Recurrence for STOOGESORT:

T(n) = 3T(n) + (1)

* 1. T(n) = 1 + 3T(n)

T(n) = 1 + 3 + 9T(n)

T(n) = 1 + 3 + 32 + … +

T(n) =

T(n) =

T(n) =

T(n) = ()

T(n) = (n2.71)

* 1. QuaternarySearch(array[], int startOfArray, int endOfArray, searchValue){

if( r 0 ){

**//calculate where each quarter begins and ends**

int firstQuarter = startOfArray + (endofArray – startOfArray)/4

int secondQuarter = firstQuarter + (endofArray – startOfArray)/4

int thirdQuarter = secondQuarter + (endofArray – startOfArray)/4

if(array[firstQuarter] == searchValue)

return firstQuarter; **//value is in the first quarter**

if(array[secondQuarter] == searchValue)

return secondQuarter; **//value is in the second quarter**

if(array[thirdQuarter] == searchValue)

return thirdQuarter; **//value is in the third quarter**

if(array[firstQuarter] > x)

return QuaternarySearch(array, startOfArray, firstQuarter-1, searchValue);

**//recurse backward through first quarter by returning the next lowest value**

if(array[secondQuarter] > x)

return QuaternarySearch(array, firstQuarter, secondQuarter-1, searchValue);

**//recurse backward through second quarter by returning the next lowest value**

if(array[thirdQuarter] > x)

return QuaternarySearch(array, secondQuarter, thirdQuarter-1, searchValue);

**//recurse backward through third quarter by returning the next lowest value**

return QuaternarySearch(arr, thirdQuarter, endOfArray, searchValue);

**//recurse through the fourth quarter by returning the next lowest value**

}

return -1; **//if value is not found**

* 1. Recurrence for quaternary search:

T(n) = T(n/4) + 8

* 1. In binary search, there are 2Log2n + 1 comparisons in worst case. In quaternary search, there are 8log4n + 3 comparisons in worst case.
  2. Using the Master Theorem, a = 8, b = 4 and f(n) = 8

T4(n) = (log(n))

* 1. Pair MaxMin(array, array\_size)

if array\_size = 1

return element as both max and min

else if arry\_size = 2

one comparison to determine max and min

return that pair

else /\* array\_size > 2 \*/

recurse for max and min of left half

recurse for max and min of right half

one comparison determines true max of the two candidates

one comparison determines true min of the two candidates

return the pair of max and min

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

Time complexity is O(n)

* 1. Iterative solution is also O(n)

Solving the problem in O(n log n) time.

Suppose we divide array A into two halves, AL and AR.

Then: A has a majority element x ⇐⇒ ⇒ x appears more than n/2 times in A

⇒x appears more than n/4 times in either AL or AR (or both)

⇐⇒ x is a majority element of either AL or AR (or both)

This suggests a divide-and-conquer algorithm:

*function majority* (A[1 . . . n]){

if n = 1: return A[1]

let AL, AR be the first and second halves of A

ML = majority(AL) and MR = majority(AR)

if ML is a majority element of A:

return ML

if MR is a majority element of A:

return MR

return ‘‘no majority’’

}

Running time: T(n) = 2T(n/2) + O(n) = O(n log n).