James Stallkamp

1

A T(n) = T(n-2)+n. Using Muster Theorem, a=1, b=2, d=1 because n^1 So

B T(n) = 3T(n-1)+1 Using Muster Theorem a=3, b=1, d=0 because n^0=1 So

C Using Master Theorem a=2. b=8 f(n) =4so 4This is case three. c=1/4 is a valid solution. This verifies regularity cond. So

2

1. The stooge sort sorts the input by sorting the first two thirds of the list, then the last two thirds, then the first two thirds again. It does so by recursively divding the list into pieces that are two thirds of the length of the input.
2. If the ceiling function were to be replaced with floor this algorithm would fail on data of certain sizes, 4 is an example. Because of the way the list would be divided one of the value would be “missed” so to speak and the list would not sort properly.
3. T(n) =

3

1. The quaternary search algorithm works by dividing a list into four pieces. Then three elements are obtained, these are the elements that are essentially on the edges between the four pieces. They are obtained using something like where i goes from 1 to 3 and n is the size of the list. The specified value is compared to the k values in order to determine which piece to further search in.

Pseudo code:

quarternarySearch(x,list)

X = desired value

n= size of list

k = floor(n/4)

if(x=list[k])

return(list[k])

if(x=list[2k])

return(list[2k])

if(x=list[3k])

return(list[3k])

if(x=list[3k])

return(list[4k])

if(x<list[k])

quaternary(x,list[0,k]

if(list[k]k<x<list[2k])

quaternary(x,list[k,2k]

if(list[2k]<x<list[3k])

quaternary(x,list[2k,3k]

if(list[3k]<x<list[n])

quaternary(x,list[3k,n]

b The recurrence for the quaternary seach is

c

d The worst case run time of the binary search tree is log(n) whereas the quaternary search is . For small sizes of n the regular binary search will be faster, but eventually with large enough sizes of n the quaternary search does become faster.

4

A divide and conquer algorithm for finding the max and min of a list would work by dividing the list into two even pieces and finding the local max and min of those sub pieces. This process would work recursively to find the local min and max and then comparing the values to find the global max and min for the list. A pseudo code for this algorithm could look like this.

MaxMin(list)

N=sizeof(list)

If(n==1)

Return a[1] as max and a[1] as min

If(n==2)

If(list[1]>list[2])

Return list[1] as max and a[2] as min

If(n>2)

(MaxLeft, minLeft) =MaxMin(list[0,n/2])

(MaxRight, minRight) =MaxMin(list[((n/2) +1), n])

If(MaxLeft>MaxRight)

Max = MaxLeft

Else

Max = MaxRight

If(MinLeft<MinRight)

Min=Minleft

Else

Min = MinRight

Return(Max, Min)

end

b The recurrence is

c Using Master Method. A = 2, b= 2 so Here we compare and f(n) = c = Θ(1) which yields f(n)=O(n) so .

Using induction the result is

5

Solution:

The array is split into two equal pieces and then the function is called twice recursively to find the majority elements of both sub pieces. This is repeated until the sublists are one element long and then they are counted back up.

getMajorityElement(A[0….n])

n = |A|

if n = 1, return a[0]

k = n/2

leftSub = getMajorityElement(a[0…k])

rightSub = getMajorityElement(a[k…n])

if leftSub = rightSub

return leftSub

lCount = getFrequency(a[0…n], leftSub)

rCount = getFrequency(a[0…n], rightSub)

if lCount > k+1

return leftSub

else if rCount > k+1

return rightSub

else

return NO MAJORITY ELEMENT

recurrence is

Using Master Method a =2 b =2 This yields case 2

Solution is