Vlad Predovic

CS325

Due 4/10/2016

Homework 2

1.

a) 𝑇(𝑛)=𝑇(𝑛−2)+𝑛: T(n-2) = T(n-4) + n - 2

T(n) = T(n-4) +n - 2 + n base case k = n-1

T(n) = T(n-k) + n – k

= n-n+1 + \*n – n +1

= 🡪 **T(n) = O(n2)**

Also, using Muster’s theorem:

For d=1, a=1, F(n)=n since F(n) = O(n)

T(n) = O(nd+1) = O(n1+1) **= O(n2)**

b) 𝑇(𝑛)=𝑇(𝑛−1)+3: **Roll pattern out:**

T(n-1) = T(n-2) + 3

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

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

To generalize: T(n) = T(n-k) + 3k

**Substitute in:**

Guess that k = n

T(n) = n-n + 3n

T(n) = 3n 🡪 O(n)

c) 𝑇(𝑛)=2𝑇()+𝑛: n = Ω(nlog42-E) = Ω (n1/2) and 2() ≤ cn so case 3

T(n)=Θ(n) of the master method.

d) 𝑇(𝑛)=4𝑇()+𝑛2√𝑛: 𝑛2√𝑛 = Ω (nlog24-E) = Ω (n2) and 4()2.5 ≤ c 𝑛2√𝑛

so T(n) = Θ(n2.5) by the master method

2.

|  |  |
| --- | --- |
| a. T(n) = 5T() + n | master case 1 /w a=5 b=2 f(n)=n:  n = O(nlog25) =n2  **T(n)= O(n2.33)** |
| b. T(n) = 2T(n-1) + c  define T(0) = constant | First expand using iteration method:  T(n) = 2T(n-1) + c  = 2(2T(n-2) + c) + c  = 2(2(2T(n-3) + c) + c) + c  T(n) = 2kT(n-k) + c(2k-1)  Guess that T(n) = 2n so then  T(n) ≤ d2n-b where d and b are const.  Base Case: T(1) = 1 ≤ d21 - b  Which is true when  T(n) = 2T(n-1) + c  = 2(d2n-1 - b) + c  = d2n - 2b + c d2n-b  Which is true as long as b > c  **Therefore T(n) = O(2n)** |
| c. T(n) = 9T() + n2 | Master case 2 /w a=9, b=3 f(n)=n2  n2 = Θ (nlog 39 + E)  **T(n)= O(n2lg(n))** |

Option c is the best as it has the lowest order exponent: **O(n2lg(n))**

3.

Each call calls foo four times on half the previous input n and then prints n times.

T(n) = 4T() + n 🡪 a=4 b=2 f(n)=n; f(n) = O(nlog24 – E) = O(n2)

So due to master theorem case 1: T(n) = O(n2)

4.

Binary: T(n) ≤ T() + c 🡪 T(n) is O(logn)

T(n) = T() + c

T(n) = T( + c + c + c

T(n) = T() + kc stop when n / 3k = 1 🡪 log3n = klog33 🡪k = log3n

T(n) = T() + (log3n)c = T(1) + (log3n)c

For my Ternary Search T(n) is also O(logn) as the algorithm will take at most logn time. It can bottom out early as well. However in each recursive call there are six comparisons as opposed to three for the binary algorithm. This is double the amount and would prove very costly in the long run.

TernarySearch(array A, query x) {

If length(A) = 0

Return false

Else

If x = middle element of A

Return true

else

I = length of A / 3

If x < A[I]

B = first third of A

Elseif x < A[I+I]

B = second third of A

Else

B = last third of A

Return TernarySearch(array B, x)

}

5.

1. T(n2) = n + n (merge 1 and 2) T(n3) = 2n + n (merge 3)

T(nk) = (k-1)n + n 🡪 T(n) = 2n + … + (k-1)n + n = n(2+3+4+…(k-1))

T(n) = (k2+K)/2 - 1 (POWER SERIES YAH)= **O(k2n)**

1. Take k sorted arrays of size n and merge the kth array with each (k+1) array until you are left with k/2 arrays. Repeat the process until you are left with 1 final array.

In order to merge each array begin at the lowest index of both arrays and create the new array by adding on the lowest value in either array until you run out of elements.

Each comparison to create a merged array takes O(1) time iterating through n elements. Therefore each merge takes O(n) time. Additionally each time a round of merges is completed the amount of merges left is halved.

**So ultimately it will take O(knlog(k))**

6.

1. With k elements insertion sort will use Θ(k2) time. However since there are n/k sub-lists, there ultimately have to be n/k sub-lists of k elements sorted: Θ(k2)

Θ(k~~2~~) = Θ(nk)

1. Mergesort usually takes Θ(nlg(n)) to sort n elements because merge sort keeps halving the arrays until you are left with one element in each. However, since the sublists are already sorted, Mergesort does not have to iterate further. Hence n/k ‘elements’ are considered instead. Substituting n/k elements for n elements results in the following recurrence: where d = n/k

T(n) = 2T(d/2) + dk

Applying the Master Theorem case 2 with a = 2, b = 2, f(n) = dk = ~~k~~ = n

f(n) = dk = Θ(nlog22 –E) = Θ(d) and therefore

T(n) = Θ(nlog22-Elgd) = Θ(dklgd) = Θ(~~k~~lgd) = **Θ(nlg())**

1. Since T(nk+nlg()) = Θ(nlg(n)) 🡪 we know ~~n~~k ≤ ~~n~~lg(n) and ~~n~~lg() ≤ ~~n~~lg(n) 🡪

k ≤ lg(n) and ~~lgn~~ – lgk ≤ ~~lgn~~ 🡪 1 ≤ k 🡪 **1 ≤ k ≤ lg(n)**

If we substitute k for the max value lg(n):

nlg(n)+nlg() 🡪 nlg(n) + nlg(n) - nlg(lgn) 🡪 2nlg(n) – nlg(lg(n))

nlg(lg(n)) is extremely small compared to nlg(n). So therefore

**2nlg(n) – nlg(lg(n)) = Θ(nlg(n))**

1. By comparing the equations with constants dependent on the implementations chosen. Find at what point Insertion sort is faster than Mergesort. You could write a function that loops, comparing the times of both until Mergesort becomes faster than Insertion sort.

7)

Note: Cannot return two variables at once. That is up to the users implementations (for example in c you could use a struct). The first variable returned is considered the min and the second the max.

minmaxCursive(array A) {

If length(A) = 1

Return A[0], A[0]

If length(A) = 2

Return A[0], A[1]

Else

Aleft = A[0…length/2-1]

Aright = A[length/2…length-1]

maxr, minr = minmaxCursive(Aright)

maxl, minl = minmaxCursive(Aleft)

If maxl < maxr

Max = maxr

Else

Max = maxl

If minl < minr

Min = minl

Else

Min = minr

Return(min, max)

}

T(n) = T(n/2) + T(n/2) + c = 2T(n/2) + c

As opposed to direct Mergesort where the recurrence is T(n) = 2T(n/2) + n,

The items do not have to be merged together.

Using the master theorem case 1:

a=2, b=2, f(n) = c 🡪 c = O(nlog22 –E) = O(n) **🡪 T(n) = O(n)**

Iterative Implementation:

minmaxSearch(array A) {

If length(A) = 0

Return false

Else

Array MinMax[2]

MinMax[0] = A[0]

MinMax[1] = A[0]

For value in A:

If value > MinMax[0]: MinMax[0] = value

Else if value < MinMax[1]: MinMax[1] = value

Return MinMax

}

The time complexity of this algorithm is T(n) = O(n) since it loops once over all items in the array comparing it to the current max/min.

Both the recursive and iterative versions of the algorithm show to have the same worst case time complexity of O(n). However, the iterative version has two comparisions that occur n times(size of the array). The recursive implementation has 4 comparisons that occur making it the slower choice