CS325_HW2

Yihong Liu

Problem1.a.

By the Muster Method: T(n) = a T(n - b) + f(n)

T(n) = b T(n - 1) + 1

a = b

b = 1

f(n) = 1 so d = 0

f(n) = theta(n^0)

Due to a being > 1, (b is fixed positive integer greater than 1)

T(n) = theta(n^d a^(n/b))

As a result,

T(n) = theta(n^0 b^(n/1))

Therefore,

T(n) = theta(b^n)

Problem1.b.

By the "formula" for solving recurrences of the form: T(n)=a T(n/b)+f(n) where , where, a ≥ 1, b > 1, and f(n) > 0 , with case 3: if f(n) = omega(n log(b)a +e) for some e > 0, and if a f(n/b) ≤ cf(n) for some c < 1 and all sufficiently large n, then: T(n) = theta(f(n)).

Our recurrence is defined as: T(n) = 3 · T(n/9) + n · log(n)

So, a = 3, b = 9, f(n)=n log(n).
We need to show that f(n) is polynomial larger than n log(b) (a+e), and we get that e=0.5
Now we need to show a f(n/b) ≤ c f(n) for some c < 1 and all sufficiently large n,
We write 3 * n/9 log (n/9) = (3/9) n log n = c f(n) where c = 1/3 and c is smaller than 1
Thus,
T(n)=Θ(n log(n))

Problem2.

Pseudocode below

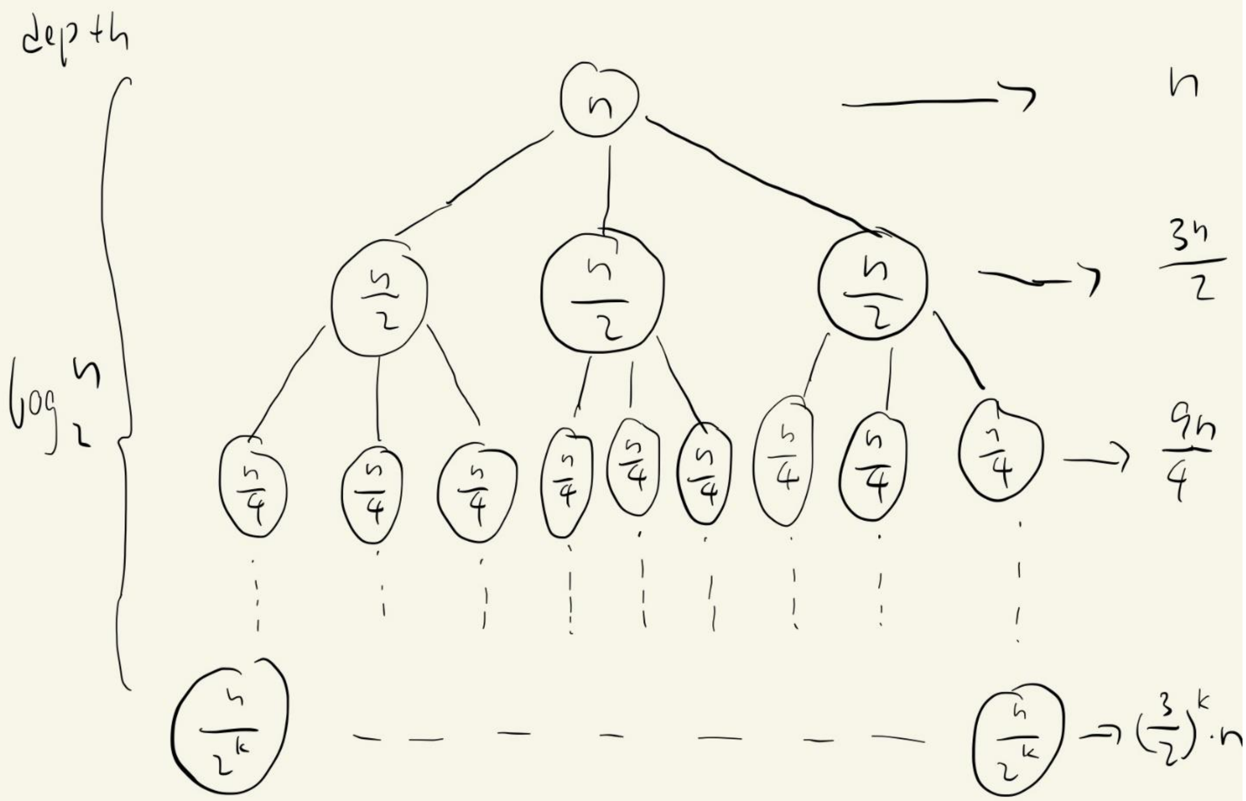
Left = 0

Right = 0

```
Sum = A[low]
Temp_sum = 0
For i = low to high
        Temp_sum = MAX (A[i] , tem_sum + A[i])
                If temp_sum > sum
                        Sum = temp_sum
                        Right = i
                        Left = temp_left
                If temp_sum == A[i]
                        Temp_left = i
Return (Left, Right, Sum)
```

Problem3.a.

$$T(n) = 3 \cdot T(n/2) + n$$



$$T(n) = n + \frac{3}{2}n + \frac{9}{4}n + \left(\frac{3}{2}\right)^{\log_2 n} \cdot n$$

$$= n\left(1 + \frac{3}{2} + \frac{9}{4} + \ldots \left(\frac{3}{2}\right)^{\log_2 n}\right)$$

$$= n \times \left(\frac{1 - \left(\frac{3}{2}\right)^{\log_2 n + 1} - 1}{\frac{3}{2} - 1}\right)$$

After some simplication,

$$T(n) = 3n^{\log_2 3} - 2n = O(n^{\log_2 3}) = O(n^{1.58})$$

Problem3.b.

proof by induction

$$T(n) = 3\ T\left(\frac{n}{2}\right) + n$$

base case : $T(0) = 0$

induction Hypothesis for $n = 2^k$

let's assume for $n = 2^k$

$(a) \leftarrow T(2^k) = (2^k)^{1.58}$ is true

for $n = 2^{k+1}$

$$T(2^{k+1}) = 3 \cdot T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1}$$

$$= 3 \cdot T(2^k) + 2^{k+1}$$

$$= 3 \times (2^k)^{1.58} + 2^{k+1}$$

$$\leq (2^k \cdot 2)^{1.58}$$

$$\leq (2^{k+1})^{1.58} = T(2^{k+1})$$

Hence, it is proved for 2^{k+1}

Problem4.a

It alpha is equal to ½, for n > 1, let's assume n = 2,

If the final array is sorted, then the elements of first half and second half of the given array must be in sorting order and each element in the first half should be less than any element in the second half of the given array.

Consider an unsorted array A and assume that in the first half of the array there is an element A[i] that is bigger than any other element in the array, then apply badsort on A.

Then badsort sorts the first half array, which is A[i] move to A [n/2 -1]

Then badsort sorts the second half of the array, thus all the elements of the second half are in the sorting order.

Again, badsort sorts the already sorted first half array, but the final array is not sorted, as every element in the second half are less than A [n/2-1]

If alpha is less than 1/2 , badsort does not consider few elements to sort, which fails too.

Hence, the badsort does not work properly if alpha is smaller or equal to ½.

Problem4.b.

Assume alpha = ¾

Badsort algorithm recursively calls itself and passes the sub array of size m in the recursive call

When the recursion calls the function with array of size n =3 , then m = 3.

Then , badsort in the first recursive call passes the array A[0...2] , recursive call again passes the array of size 3.

In each recursive call, size of the array must be reduced, and the recursion ends only when the size is reduced to 2.

If array's size is not reduced in each recursion, the recursion will enter infinite loop.

Thus, it cannot work properly also with alpha = ¾.

The way to fix it is to add the following code after m = $\lceil \alpha \cdot n \rceil$,

if   m == n

m = m-1,

this can make sure m is being reduced.

Problem4.c.

T(n) = 3T(alpha * n) + c

Problem4.d

By the Master Method: T(n) = a T(n/b) + f(n)

T(n) = 3T(2n/3) + c

a = 3

b = 3/2

f(n) = c

n^(logb(a)) = n^(log(3/2)(3)) = n^2.71

if f(n) = O(n^logb(a-e)) for some e > 0 then

T(n) = theta((n^logb(a))

Therefore,

T(n) = theta (n^2.71)

Problem5.a.

See teach files

Problem5.b.

Please enter a fraction or decimal value for alpha < 1:

3/4

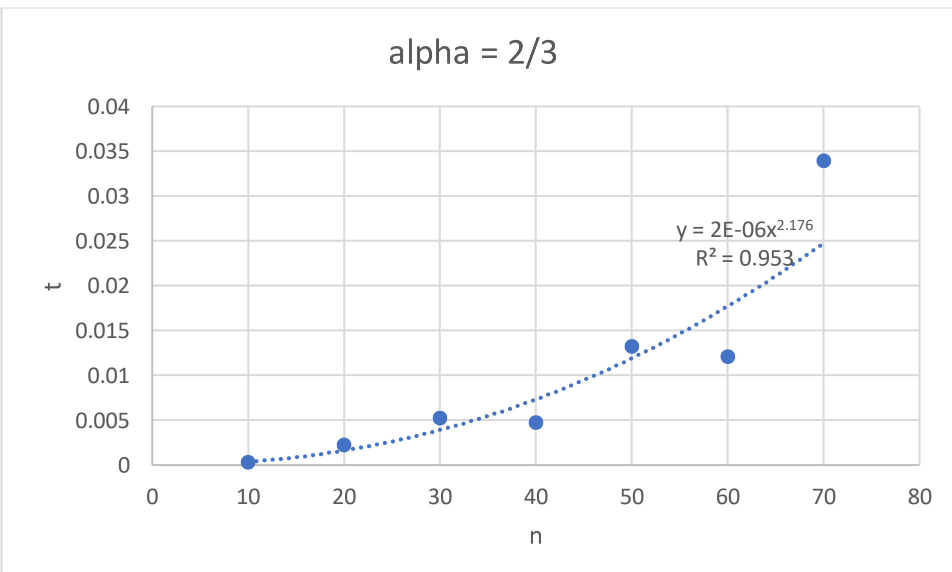
Size	Running time
10	0.0008089542388916016
20	0.015100955963134766
30	0.09955406188964844
40	0.2715120315551758
50	0.2786870002746582
60	0.7923452854156494
70	2.396939754486084

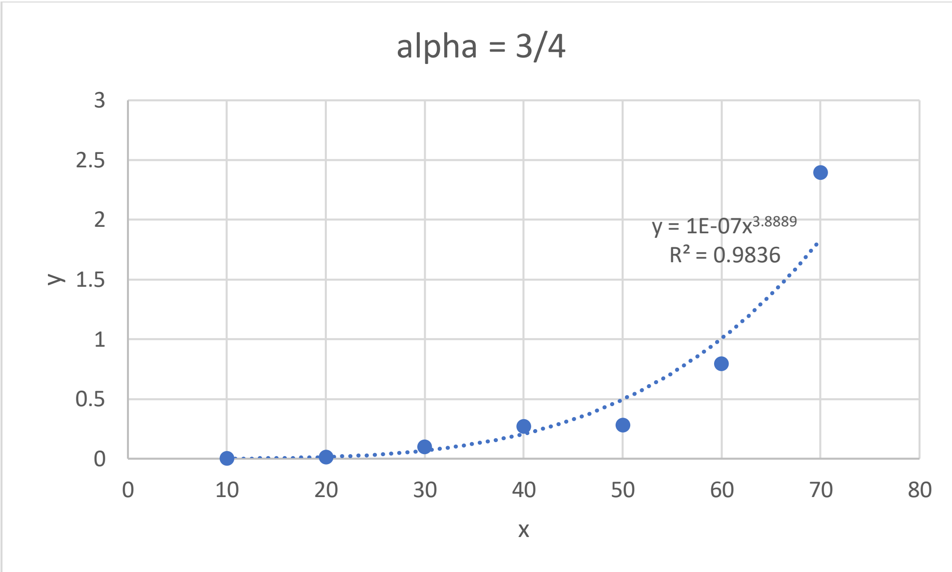
Please enter a fraction or decimal value for alpha < 1:

2/3

Size	Running time
10	0.00029206275939941406
20	0.002185821533203125
30	0.005240201950073242
40	0.004729747772216797
50	0.013213872909545898
60	0.012063026428222656
70	0.033876895904541016

Problem5.c.





A power curve best fits the badSort data set. The equation of the curve that best "fits" the data is y = x^2.175. Comparing the experimental running time to that of the theoretical running time of the Stooge Sort algorithm, they are close. The experimental was y = x^2.175 and the theoretical was y = x^2.71 in problem 4d.

Problem5.d

When alpha = 2/3, it performs better. It's much faster than alpha = ¾.