**CS 325 HW 2 – 30pts**

Problem 1: (6 pts) 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. 𝑇(𝑛) = 𝑇(𝑛 − 2) + 𝑛

**Solution**

Lets consider n=(n-2), then our T(n) becomes

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

for n=(n-4), then our T(n) becomes

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

Therefore T(n) =T(n-2\*k) + (n\*k) –( 2 \* )

If we take the first part of the equation and get T(0), then

n-2\*k = 0

therefore, Value of k can be n/2 . If we substitute this in the second part of the above equation n\*k

n\*k = n\*(n/2) = n2/2

Hence T(n) = θ(n2)

1. 𝑇(𝑛) = 4𝑇(𝑛/ 2 ) + 𝑛3

**Solution**

Using Master theorem

a=4, b= 2

n^ logba = n2 ; f(n) = 𝑛3

Case 3 : f(n) = Ω(n2+є) for є = 1 and

4(n/2)3 <= C 𝑛3 for c=1/2

Therefore T(n) = θ(𝑛3)

1. 𝑇(𝑛) = 9𝑇(𝑛/3 ) + n2

**Solution**

Using Master theorem

a=9, b= 3

n^ logba = n2 ; f(n) = n2

Case 2 : f(n) = θ (n2)

Therefore T(n) = θ (n2lgn)

Problem 2: (4 pts) How many times as a function of n (in θ form), does the following PHP function echo “Print”? Write a recurrence and solve it.

function Algo1( $n ) {

if ($n > 1) {

Algo1($n/3);

Algo1($n/3);

Algo1($n/3);

Algo1($n/3);

for ($i = 1; $i <= $n; $i += 1)

{ echo " Print ".$i." <br> "; }

echo " <br>"; }

else {

return 1;

} }

**Solution** :

Here the problem is divided into 4 parts of n/3 size. The Algo1 function is called recursively 4 times and in each call the echo is called n times.

Hence the Recurrence can be written as

T(n) = 4T(n/3) + O(n)

We can solve the above recurrence using Master Theorem

Using Master theorem

a=4, b= 3

n^ logba = 𝑛log34 will be greater than 1

f(n) = 𝑛

Case 1 : f(n) = O(nlogba -є) for є = 1 and

Therefore T(n) = θ(𝑛log34)

Problem 3: (5 pts) The ternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately onethird.

1. Write pseudo-code a recursive ternary search algorithm

**Solution**

Pseudocode for Ternary search :

Function int ternarySearch( low, high, x)

If(high > = 1)

{

mid1 = low + ( high – low)/3;

mid2 = high – (high – low)/3;

If(a[mid1] == x)

return mid1;

If(a[mid1] == x)

return mid1;

If(x < a[Mid1] )

return ternarySearch(low, mid1-1, x);

else If(x > a[Mid2] )

return ternarySearch(mid2+1, high, x);

else return ternarySearch(mid1+1, mid2-1, x);

}

return 0; }

1. Let T(n) denote the running time of ternary on an array of size n. Write a recurrence relation for T(n).

**Solution -** As we can see from the above pseudocode, the array of n size is divided into size n/3. That is the ternary search function is called only once based on the conditions in order to implement this. Also there is a constant time for the comparison.

Hence the Recurrence relation can be written as

T(n) = n/3 + C

Worst case it can be 2n/3 + C

1. Solve the recurrence relation to obtain the asymptotic running time.

T(n) = T(n/3) + C

We can solve this using Master Theorem

a=1, b = 3

n^(logba) = n0 ; f(n) = C

f(n) = θ (n^(logba)) = θ(n0) = θ(1) -> Case 2

Therefore T(n) = θ (lgn)

Problem 4: (5 pts) The Mergesort3 algorithm is a variation of Mergesort that instead of splitting the list into two halves, splits the list into three thirds. Mergesort3 recursively sorts each third and then merges the thirds together into a sorted list by calling a function named Merge3.

1. Write pseudo-code for Mergesort3 and Merge3

**Mergesort3**

function mergeSort3(a[], low, high, f[])

{

// If array size is 1 then do nothing

if (high - low < 2)

return;

// Splitting array into 3 parts

int mid1 = low + ((high - low) / 3);

int mid2 = low + 2 \* ((high - low) / 3) + 1;

// Sorting 3 arrays recursively

mergeSort3(f, low, mid1, a);

mergeSort3(f, mid1, mid2, a);

mergeSort3(f, mid2, high, a);

// Calling the merge function to merge the sorted arrays

merge(f, low, mid1, mid2, high, a);

}

**Merge3**

function merge(a[],low,m1,m2,high,f[])

{

**int** i = low, j = m1, p = m2, l = low;

**while** ((i < m1) && (j < m2) && (p < high))

{

**if** (a[i] < (a[j])) {

**if** (a[i] < a[p])

f[l++] = a[i++];

**else** f[l++] = a[p++];

}

**else** {

**if** (a[j] < a[p]))

f[l++] = a[j++];

**else** f[l++] = a[p++];

}

}

// comparing first and second ranges

**while** ((i < m1) && (j < m2))

{

**if** (a[i]<a[j]))

f[l++] = a[i++];

**else** f[l++] = a[j++];

}

// comparing second and third ranges

**while** ((j < m2) && (p < high))

{

**if** (a[j]<a[p]))

f[l++] = a[j++];

**else** f[l++] = a[p++];

}

// comparing first and third range

**while** ((i < m1) && (p < high))

{

**if** (a[i]<a[p]))

f[l++] = a[i++];

**else**

f[l++] = a[p++];

}

// copying from first range

**while** (i < m1)

f[l++] = a[i++];

// copying from second range

**while** (j < m2)

f[l++] = a[j++];

// copying from the third range

**while** (p < high)

f[l++] = a[p++];

}

1. Let T(n) denote the running time of Mergesort3 on an array of size n. Write a recurrence relation for T(n).

As we can see from the pseudo code, we can say that the array is splitted into 3 parts of size (n/3). And the merge function takes n times a constant.

Therefore the recurrence for merge 3 sort can be written as below :

T(n) = 3T(n/3) + Cn

c) Solve the recurrence relation to obtain the asymptotic running time.

T(n) = 3T(n/3) + Cn

We can solve the above recurrence relation using Master Theorem

a=3, b = 3

n^(logba) = n1 ; f(n) = Cn

f(n) = θ (n^(logba)) = θ(n1) = θ(n) -> Case 2

Therefore T(n) = θ (nlgn)

**Problem 5: (10 pts)**

a) Implement the Mergesort3 algorithm to sort an array/vector of integers. You must implement the algorithm in same language you used in homework 1. Name the program “Mergesort3”. Your program should be able to read inputs from a file called “data.txt” where the first value of each line is the number of integers that need to be sorted, followed by the integers. Example values for data.txt: 4 19 2 5 11 8 1 2 3 4 5 6 1 2 The output will be written to files called “merge3.txt”. For the above example the output would be: 2 5 11 19 1 1 2 2 3 4 5 6

Solution – Code has been submitted in Teach and is running in Flip Server.

Path in Flip Server – jayapats/CS325/HW2/Merge3File.java

b) Modify code- Now that you have verified that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from the file data.txt and sorting, you will now generate arrays of size n containing random integer values from 0 to 10,000 to sort. Use the system clock to record the running times of each algorithm for ten different values of n for example: n = 5000, 10000, 15000, 20,000, …, 50,000. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data (do not collect times over a minute). Output the array size n and time to the terminal. Name these new program merge3Time.

**Solution** – Code has been submitted in Teach and is running in Flip Server.

Path in Flip Server – jayapats/CS325/HW2/Merge3Time.java

c) Collect running times - Collect your timing data on the engineering server. You will need at least eight values of t (time) greater than 0. Create a table of running times for each algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value of N** | **Excecution time** | | | |
| **1** | **2** | **3** | **Average** |
| 0 | 760 | 0 | 0 | 253.3333 |
| 10000 | 1410000 | 510000 | 510000 | 810000 |
| 20000 | 1020000 | 1020000 | 1320000 | 1120000 |
| 30000 | 830000 | 730000 | 830000 | 796666.7 |
| 40000 | 940000 | 940000 | 940000 | 940000 |
| 50000 | 1450000 | 1350000 | 1550000 | 1450000 |
| 60000 | 2060000 | 2060000 | 2160000 | 2093333 |
| 70000 | 2570000 | 2470000 | 2770000 | 2603333 |
| 80000 | 2880000 | 3280000 | 2980000 | 3046667 |
| 90000 | 2090000 | 2190000 | 2090000 | 2123333 |
| 100000 | 23100000 | 23100000 | 23100000 | 23100000 |
| 110000 | 36110000 | 35110000 | 26110000 | 32443333 |
| 120000 | 27120000 | 28120000 | 27120000 | 27453333 |
| 130000 | 31130000 | 30130000 | 42130000 | 34463333 |
| 140000 | 39140000 | 33140000 | 33140000 | 35140000 |

1. Plot data and fit a curve - For each algorithm plot the running time data you collected on a graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. What type of curve best fits each data set? Give the equation of the curves that best “fits” the data and draw that curves on the graphs.

d) Compare - Plot the data from Mergesort3 and Mergesort (from HW 1) together on a combined graph. Which algorithm runs faster? How does your experimental running times compare to the theoretical running times of the algorithms?

Merge 2 way Run time :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value of N** | **Excecution time** | | | |
| **1** | **2** | **3** | **Average** |
| 0 | 800 | 0 | 0 | 266.6667 |
| 10000 | 910000 | 910000 | 410000 | 743333.3 |
| 20000 | 720000 | 720000 | 620000 | 686666.7 |
| 30000 | 1330000 | 1030000 | 930000 | 1096667 |
| 40000 | 1240000 | 1440000 | 1040000 | 1240000 |
| 50000 | 1750000 | 1750000 | 1750000 | 1750000 |
| 60000 | 2160000 | 2160000 | 2160000 | 2160000 |
| 70000 | 2770000 | 2870000 | 2570000 | 2736667 |
| 80000 | 2880000 | 3280000 | 3280000 | 3146667 |
| 90000 | 3490000 | 4590000 | 4390000 | 4156667 |
| 100000 | 39100000 | 28100000 | 28100000 | 31766667 |
| 110000 | 28110000 | 29110000 | 39110000 | 32110000 |
| 120000 | 31120000 | 32120000 | 31120000 | 31453333 |
| 130000 | 44130000 | 43130000 | 46130000 | 44463333 |
| 140000 | 45140000 | 50140000 | 48140000 | 47806667 |

Merge 2 way sort Graph :

Merge3 Runtime recurrence:

T(n) = 3T(n/3) + Cn

Merge 2 Runtime recurrence:

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

Combined raph :

Orange – Merge 3 Algorithm

Blue – 2 way merge Algorithm

Comparing the normal merge and merge 3, both runs for the same runtime. Both fits the run time data perfectly.