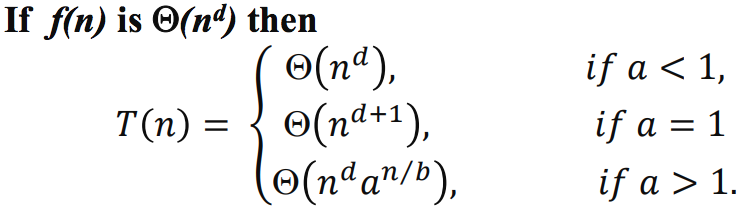
Problem 1:

By using Muster Method Version 2:

𝑇(𝑛) = 3𝑇(𝑛 −1) +1 | a = 3, b = 1, f(n) = 1 | for a > 0, b > 0, d >= 0

1 = O(nd) 🡪 O(n0)



Since a > 1, we plug in the values into Θ() for solving the problem.

Θ() 🡪 Θ()

Answer: T(n) = Θ(3n), as 3n is the asymptotic bounds for T(n).

Problem 2:

a). Pseudocode for Ternary Search Tree:

//Linked list will be used for the TST

ternarySearch(root, value){

if not root //If the root is NULL, return 0

return false;

if value equals to root->data //If the value is the same as the root value

return true;

//If the value is less than the node, then we should search the left of TST recursively

else if value less than root->data

then return ternarySearch(root->left, value);

else if value greater root->data)//If the value is greater, then go the right

then return ternarySearch(root->right, value);

else

then return ternarySearch(root->middle, value); //Else go to the middle

}

b). Recurrence:

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

c). Solving Recurrence:

By using the Master Method,

a=1, b=3, f(n) = Θ. f(n) is a constant, so it’s also Θ(1).

log31 = 0 🡪 n0 🡪 Θ(1). In this comparison with f(n), we can use case 2.

T(n) = Θ(lgn)

Comparison to the run time of binary search algorithm, they both tend to have the time complexity of Θ(lgn).

Problem 3:

a). Pseudocode:

**maxMin**(array){

if array size = 1

return array[0];

//Takes the array and divide it by 2 for two separate arrays

left\_array = first half of the array;

right\_array = second half of the array;

//Finds the max and min from the left array

(left\_min, left\_max) = maxMin(left\_array);

//Finds the max and min from the right array

(right\_min, right\_max) = maxMin(right\_array);

//Compare the max and min from the left array with the right array to determine the actual max and min values

If left\_min <= right\_min, then actual\_min = left\_min;

Else actual\_min = right\_min;

If left\_max <= right\_max, then actual\_max = right\_max;

Else actual\_max = left\_max;

Return (actual\_min, actual\_max);

}

b). Recurrence:

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

c). Solving the recurrence:

By using the Master Method,

a=2, b=2, f(n) = Θ(1) .

Log22 = 1, nlog22 = n, case 1 since n > Θ(1)

T(n) = Θ(n).

By using recursion, the time is linear. With iterative algorithm, it will still going to Θ(n).

Problem 4:

a). Pseudocode for 4-way merge sort

mergeSort(array, left, right){

if left value is less than right value{

middle = the middle of the whole array

left\_middle = the middle of the first half array

right\_middle = the middle of the second half array

//Since it’s a 4 way merge sort, calling the function 4 times with different range pass in

mergeSort(array, first left array)

mergeSort(array, second left array)

mergeSort(array, first right array)

mergeSort(array, second right array)

//calling the merging function to merge the arrays

merging(array, all the middle points)

}

}

b). Recurrence and Solution

Recurrence: T(n) = 4\*T(n/4)+n

Solution: By using the Master Method,

a=4,b=4,f(n)=n;

log44 = 1, nlog44 = n, case 2

T(n) = Θ(n lgn).

Problem 5:

c). 4 Way Merge Sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | Run Time 1 | Run Time 2 | Run Time 3 | Run Time 4 |
| 50000 | 0.01 | 0.01 | 0.01 | 0.01 |
| 100000 | 0.03 | 0.03 | 0.03 | 0.03 |
| 150000 | 0.06 | 0.06 | 0.06 | 0.06 |
| 200000 | 0.08 | 0.07 | 0.07 | 0.07 |
| 250000 | 0.08 | 0.08 | 0.08 | 0.08 |
| 300000 | 0.11 | 0.1 | 0.11 | 0.11 |
| 350000 | 0.13 | 0.13 | 0.13 | 0.13 |
| 400000 | 0.15 | 0.15 | 0.15 | 0.15 |
| 450000 | 0.12 | 0.17 | 0.18 | 0.12 |
| 500000 | 0.15 | 0.15 | 0.2 | 0.15 |

d).

The type of curve best fits to this data set would be a supposed linear curve despite the complexity is O(n log4 n). Since this is presented in a large data scale, the n will conquer the log4 n. The line of best fit is labelled in the graph.

e).

4 way merge sorting is faster than both insertion and merge sort because 4 way merge breaks the array into four parts and sort them instead of just breaking down into two like the regular merge sort.

Modified Timing Code

//Reference to my previous merge sort program

//Merge sort reference:

//https://www.geeksforgeeks.org/merge-sort/

int main(){

srand (time(NULL));

clock\_t t;

int array\_size;

int temp\_var;

int n =0;

int random;

for(int i=0;i<10;i++){

n = n + 50000;

int array[1000000];

for(int j=0;j<n;j++){

random = rand()%100001;

array[j] = random;

}

t = clock();

// Calling the merge function

mergeSortFourWays(array, n-1);

t = clock() - t;

cout<<"Array Size: "<<n<<endl;

cout<<"Time: "<<((float)t)/CLOCKS\_PER\_SEC << " seconds"<<endl;

cout<<endl;

}

return 0;

}