Algorithms and Data Structures

Spring 2019

Assignment 2

Date: February 20, 2019

Taiyr Begeyev

Problem 2.1 Merge Sort

(a) Implement a variant of Merge Sort that does not divide the problem all the way down to subproblems of size 1. Instead, when reaching subsequences of length k it applies Insertion Sort on these n/k subsequences.

```
/*
    @brief
        Implementation of Insertion Sort Algorithm.
        The array is searched sequentially
        and unsorted items are moved and inserted into the sorted sub-list
    @param array
    @param lenght of the array
*/

void insertionSort(int arr[], int min, int max) {
    int key, j;
    for (int i = min + 1; i <= max; i++) {
        key = arr[i];
        j = i - 1;

        while(j >= min && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

/*
    @brief
        Create two arrays(left and right). Find the length of them and
        copy the values from arr to them. Merge them
    @param array
    @param start index
    @param middle index
    @param end index
*/

void merge(int arr[], int start, int middle, int end) {
    int i, j, k;

    //find the length of the Left Hand array
    int lengthOfLeft = middle - start + 1;
    //find the length of the Right Hand array
    int lengthOfRight = end - middle;

    //create two arrays
    int left[lengthOfLeft], right[lengthOfRight];

    //fill two arrays with elements from arr
    for (i = 0; i < lengthOfLeft; i++)
        left[i] = arr[start + i];
    for (j = 0; j < lengthOfRight; j++)
        right[j] = arr[middle + 1 + j];

    i = 0; j = 0; k = start;
    while(i < lengthOfLeft && j < lengthOfRight) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
```

```
            i++;
        }
        else{
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    while(i < lengthOfLeft) {
        arr[k] = left[i];
        i++;
        k++;
    }

    while(j < lengthOfRight) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

/*
    @brief
        Sorts the elements in the subarray A[start..end]. If start >= end, the
        subarray has at most one element and is therefore already sorted.
        Otherwise, the divide step simply computes an index middle that
        partitions A[start..end] into two subarrays: A[start..middle] and
        A[middle + 1 .. end]

        To sort the entire sequence, we make initial call, where start = 0
        and end = A.length
    @param array
    @param starting index
    @param ending index
    @param length of a certain subsequence
*/

void mergeSort(int arr[], int start, int end, int k) {
    if (end - start + 1 <= k) {
        insertionSort(arr, start, end);
    }
    else {
        int middle = (end + start) / 2;
        mergeSort(arr, start, middle, k);
        mergeSort(arr, middle + 1, end, k);
        merge(arr, start, middle, end);
    }
}
```

(b) Apply it to the different sequences from Problem 1.2 (from Homework 1) for different numbers of k. Add the computation times to the plots you had generated in Problem 1.2

```
    /*
    @brief check different arrays for the same k and for different scenarios
    (AVERAGE CASE, WORST CASE, BEST CASE)
    write n size and the corresponding time to output2.dat and output3.dat
    We will check different arrays for two different k
```

```
@param the name of file we want to write data to
@param k
*/

void differentNforTheSameK(string name, int k) {
ofstream myfile;
myfile.open(name);

for (int i = 100; i < 1000; i += 10) {
    int* arr = new int[i]; //main array
    int* arrAverageCase = new int[i];
    int* arrBestCase = new int[i];
    int* arrWorstCase = new int[i];

    randomlyGenerateArray(arr, i);
    copyArray(arr, arrAverageCase, i);
    copyArray(arr, arrBestCase, i);
    copyArray(arr, arrWorstCase, i);

    //WORST CASE
    //start measuring time
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    mergeSort(arrWorstCase, 0, i - 1, k);
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    // finish
    //convert to microseconds
    auto duration = duration_cast<microseconds>( t2 - t1 ).count();
    //write data to the file
    myfile << i << " " << duration << " ";

    //AVERAGE CASE
    //start measuring time
    t1 = high_resolution_clock::now();
    mergeSort(arrAverageCase, 0, i - 1, k);
    t2 = high_resolution_clock::now();
    // finish
    //convert to microseconds
    duration = duration_cast<microseconds>( t2 - t1 ).count();
    //write data to the file
    myfile << i << " " << duration << " ";

    //BEST CASE
    //start measuring time
    t1 = high_resolution_clock::now();
    mergeSort(arrBestCase, 0, i - 1, k);
    t2 = high_resolution_clock::now();
    // finish
    //convert to microseconds
    duration = duration_cast<microseconds>( t2 - t1 ).count();
    //write data to the file output.dat
    myfile << i << " " << duration << endl;

    delete[] arr;
    delete[] arrAverageCase;
    delete[] arrWorstCase;
    delete[] arrBestCase;
}
myfile.close();
}
```

The full code available in mergeSort.cpp

(c) How do the different values of k change the best, average, and worstcase asymptotic time complexities for this variant? Explain/prove your answer.

The whole code. The solution for this task is unbder **CHECK THE SAME ARRAY FOR DIFFERENT k AND FOR DIFFERENT SCENARIOS** block

```
/*
Algorithms and Data Structures
Spring 2019
Assignment #2
mergeSort.cpp

Purpose: Implement a variant of Merge Sort that does not divide the problem all the
Insertion Sort on these n/k subsequences.
@author Taiyr Begeyev
@version 2.0 20/02/19
*/

/*
----------           .--
\--     ---/ -----    |--|  ---.---. -------
  |     |    \--  \   |  | <   |   | \-   -- \
  |     |     / -- \- |  |  \---   |   |  | \/
  |----|     (----  / |--|  / ----|   |--|
                  \/         \/
*/



#include <iostream>
#include <ctime>
#include <stdlib.h>
#include <fstream>
#include <chrono>
using namespace std;
using namespace std::chrono;

/* Prototype Declaration */
//void insertionSort(int arr[], int length);
void insertionSort(int arr[], int min, int max);
void merge(int arr[], int start, int middle, int end);
void mergeSort(int arr[], int start, int end, int k);
void printArr(int arr[], int length);
void randomlyGenerateArray(int arr[], int length);
void copyArray(int arr1[], int arr2[], int length);
void reverseOrderOfArray(int arr[], int length);
void differentNforTheSameK(string name, int k);

int main() {
    /* basic setup */
    ofstream myfile1;
    myfile1.open("output1.dat");
    srand(time(NULL));

    int length = rand() % 100 + 1;
    int* arr = new int[length]; //main array
    int* arrAverageCase = new int[length];
    int* arrBestCase = new int[length];
    int* arrWorstCase = new int[length];

    randomlyGenerateArray(arr, length);
```

```
//copy arr to another arrays
copyArray(arr, arrAverageCase, length);
copyArray(arr, arrBestCase, length);
copyArray(arr, arrWorstCase, length);

// Worst Case
// reversed order
reverseOrderOfArray(arrWorstCase, length);

//Best Case
// already sorted array
insertionSort(arrBestCase, 0, length - 1);

cout << "Length of the array: " << length << endl;
cout << "Array: " << endl;
printArr(arr, length);
cout << endl;

cout << "Sorted array: " << endl;
printArr(arrBestCase, length);
cout << endl;

// ---------------------------------------------------------------------------
// CHECK THE SAME ARRAY FOR DIFFERENT k AND FOR DIFFERENT SCENARIOS
// (AVERAGE CASE, WORST CASE, BEST CASE)
// write k and the corresponding time to output1.dat
for (int k = 1; k <= length; k++) {
    //WORST CASE
    //start measuring time
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    mergeSort(arrWorstCase, 0, length - 1, k);
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    // finish
    //convert to microseconds
    auto duration = duration_cast<microseconds>( t2 - t1 ).count();
    //write data to the file output.dat
    myfile1 << k << " " << duration << " ";

    //AVERAGE CASE
    //start measuring time
    t1 = high_resolution_clock::now();
    mergeSort(arrAverageCase, 0, length - 1, k);
    t2 = high_resolution_clock::now();
    // finish
    //convert to microseconds
    duration = duration_cast<microseconds>( t2 - t1 ).count();
    //write data to the file output.dat
    myfile1 << k << " " << duration << " ";

    //BEST CASE
    //start measuring time
    t1 = high_resolution_clock::now();
    mergeSort(arrBestCase, 0, length - 1, k);
    t2 = high_resolution_clock::now();
    // finish
    //convert to microseconds
    duration = duration_cast<microseconds>( t2 - t1 ).count();
    //write data to the file output.dat
    myfile1 << k << " " << duration << endl;
```

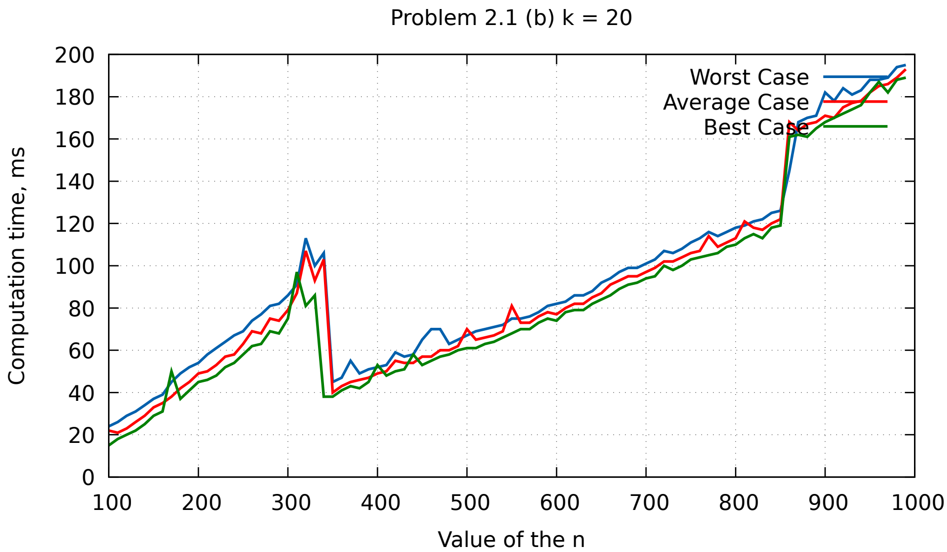
```
        //Since we modified our arrays (sorted it), we need to
        get the initial sequence
        // that's why we copy arr to them
        copyArray(arr, arrAverageCase, length);
        copyArray(arr, arrWorstCase, length);
    }

    delete[] arr;
    delete[] arrAverageCase;
    delete[] arrWorstCase;
    delete[] arrBestCase;

    // ------------------------------------------------------------------------
    // CHECK DIFFERENT ARRAYS FOR THE SAME k AND FOR DIFFERENT SCENARIOS
    // (AVERAGE CASE, WORST CASE, BEST CASE)
    // write n size and the corresponding time to output2.dat and output3.dat

    //We will check different arrays for two different k
    // FIRST
    //k = 20;
    differentNforTheSameK("output2.dat", 20);
    // SECOND
    // k = 50;
    differentNforTheSameK("output3.dat", 50);

    myfile1.close();
    return 0;
}

/*
    @brief
        Implementation of Insertion Sort Algorithm.
        The array is searched sequentially
        and unsorted items are moved and inserted into the sorted sub-list
    @param array
    @param lenght of the array
*/

void insertionSort(int arr[], int min, int max) {
    int key, j;
    for (int i = min + 1; i <= max; i++) {
        key = arr[i];
        j = i - 1;

        while(j >= min && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

/*
    @brief
        Create two arrays(left and right). Find the length of them and
        copy the values from arr to them. Merge them
    @param array
    @param start index
    @param middle index
    @param end index
```

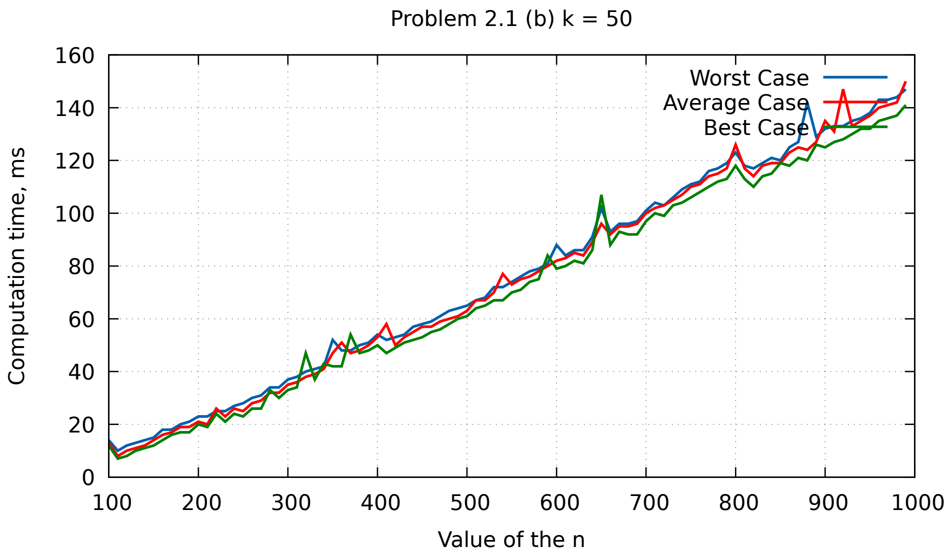
```
*/

void merge(int arr[], int start, int middle, int end) {
    int i, j, k;

    //find the length of the Left Hand array
    int lengthOfLeft = middle - start + 1;
    //find the length of the Right Hand array
    int lengthOfRight = end - middle;

    //create two arrays
    int left[lengthOfLeft], right[lengthOfRight];

    //fill two arrays with elements from arr
    for (i = 0; i < lengthOfLeft; i++)
        left[i] = arr[start + i];
    for (j = 0; j < lengthOfRight; j++)
        right[j] = arr[middle + 1 + j];

    i = 0; j = 0; k = start;
    while(i < lengthOfLeft && j < lengthOfRight) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        }
        else{
            arr[k] = right[j];
            j++;
        }
        k++;
    }


    /* Copy the remaining elements of L[], if there
       are any */
    while(i < lengthOfLeft) {
        arr[k] = left[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while(j < lengthOfRight) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

/*
    @brief
        Sorts the elements in the subarray A[start..end]. If start >= end, the
        subarray has at most one element and is therefore already sorted.
        Otherwise, the divide step simply computes an index middle that
        partitions A[start..end] into two subarrays: A[start..middle] and
        A[middle + 1 .. end]

        To sort the entire sequence, we make initial call, where start = 0
        and end = A.length
```

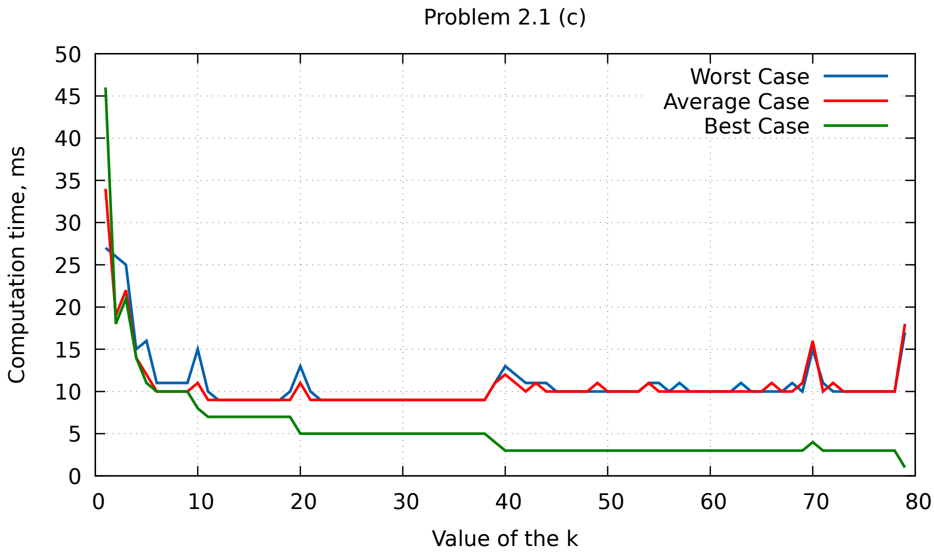
```
    @param array
    @param starting index
    @param ending index
    @param length of a certain subsequence
*/

void mergeSort(int arr[], int start, int end, int k) {
    if (end - start + 1 <= k) {
        insertionSort(arr, start, end);
    }
    else {
        int middle = (end + start) / 2;
        mergeSort(arr, start, middle, k);
        mergeSort(arr, middle + 1, end, k);
        merge(arr, start, middle, end);
    }
}

/*
    @brief print an array
    @param array
    @param a length of the array
*/

void printArr(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

/*
    @brief generates array of random numbers in a certain range
    @param array and its length
    @returns randomized array
*/

void randomlyGenerateArray(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i] = rand() % 500 + 1;
    }
}

/*
    @brief copies elements from Arr1 to Arr 2
    @param arr1
    @param arr2
    @param length of both arrays
    @returns the array where elements from arr1 were copied to
*/

void copyArray(int arr1[], int arr2[], int length) {
    for (int i = 0; i < length; i++) {
        arr2[i] = arr1[i];
    }
}

/*
    @brief make a reversed order of an array for the worst case
    @param array and its length
```

```
    @returns reversed array
*/

void reverseOrderOfArray(int arr[], int length) {
    int positionOfMax, tmp;

    //go through all elements
    for (int i = 0; i < length - 1; i++) {
        //make current element min and remember its position
        positionOfMax = i;
        //find tshe min in unsorted part
        for (int j = i + 1; j < length; j++) {
            if (arr[j] > arr[positionOfMax]) {
                positionOfMax = j;
            }
        }

        //swap min with current element if min is not current element
        if (positionOfMax != i) {
            tmp = arr[i];
            arr[i] = arr[positionOfMax];
            arr[positionOfMax] = tmp;
        }
    }
}

/*
    @brief CHECK DIFFERENT ARRAYS FOR THE SAME k AND FOR DIFFERENT SCENARIOS
           (AVERAGE CASE, WORST CASE, BEST CASE)
           write n size and the corresponding time to output2.dat and output3.dat
           We will check different arrays for two different k
    @param the name of file we want to write data to
    @param k
*/

void differentNforTheSameK(string name, int k) {
    ofstream myfile;
    myfile.open(name);

    for (int i = 100; i < 1000; i += 10) {
        int* arr = new int[i]; //main array
        int* arrAverageCase = new int[i];
        int* arrBestCase = new int[i];
        int* arrWorstCase = new int[i];

        randomlyGenerateArray(arr, i);
        copyArray(arr, arrAverageCase, i);
        copyArray(arr, arrBestCase, i);
        copyArray(arr, arrWorstCase, i);

        //WORST CASE
        //start measuring time
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        mergeSort(arrWorstCase, 0, i - 1, k);
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        // finish
        //convert to microseconds
        auto duration = duration_cast<microseconds>( t2 - t1 ).count();
        //write data to the file
        myfile << i << " " << duration << " ";
```

```
        //AVERAGE CASE
        //start measuring time
        t1 = high_resolution_clock::now();
        mergeSort(arrAverageCase, 0, i - 1, k);
        t2 = high_resolution_clock::now();
        // finish
        //convert to microseconds
        duration = duration_cast<microseconds>( t2 - t1 ).count();
        //write data to the file
        myfile << i << " " << duration << " ";

        //BEST CASE
        //start measuring time
        t1 = high_resolution_clock::now();
        mergeSort(arrBestCase, 0, i - 1, k);
        t2 = high_resolution_clock::now();
        // finish
        //convert to microseconds
        duration = duration_cast<microseconds>( t2 - t1 ).count();
        //write data to the file output.dat
        myfile << i << " " << duration << endl;

        delete[] arr;
        delete[] arrAverageCase;
        delete[] arrWorstCase;
        delete[] arrBestCase;
    }
    myfile.close();
}
```







Gnuplot script

gnuplot plot1.p

```
set terminal pdf
set output "plot1.pdf"

# x and y labels
set xlabel "Value of the k"
set ylabel "Computation time, ms"
set title "Problem 2.1 (c)"

# Line styles
set style line 1 \
    linecolor rgb "#0060ad" \
    linetype 1 linewidth 2 \
    pointtype 7 pointsize 1.5

set style line 2 \
    linecolor rgb "#ff0000" \
    linetype 2 linewidth 2 \
    pointtype 7 pointsize 1.5

set style line 3 \
    linecolor rgb "#008000" \
```

```
    linetype 3 linewidth 2 \
    pointtype 7 pointsize 1.5

set grid

plot "output1.dat" using 1:2 title "Worst Case"  with lines linestyle 1,\
     "output1.dat" using 3:4 title "Average Case" with lines linestyle 2,\
     "output1.dat" using 5:6 title "Best Case" with lines linestyle 3


set output "plot2.pdf"

set xlabel "Value of the n"
set ylabel "Computation time, ms"
set title "Problem 2.1 (b) k = 20"

plot "output2.dat" using 1:2 title "Worst Case"  with lines linestyle 1,\
     "output2.dat" using 3:4 title "Average Case" with lines linestyle 2,\
     "output2.dat" using 5:6 title "Best Case" with lines linestyle 3

set output "plot3.pdf"

set xlabel "Value of the n"
set ylabel "Computation time, ms"
set title "Problem 2.1 (b) k = 50"

plot "output3.dat" using 1:2 title "Worst Case"  with lines linestyle 1,\
     "output3.dat" using 3:4 title "Average Case" with lines linestyle 2,\
     "output3.dat" using 5:6 title "Best Case" with lines linestyle 3
```

(c) As you can see in the last picture, the best case running time is decreasing when k is increasing. The Worst Case running time is increasing as k is increased. The Average case reaches its minimum when k is around 15. Suppose we have random k. This meas we can just start using the usual merging procedure, except starting it at the level in which each array has size at most k. This means that the depth of the merge tree is $lg(n)lg(k) = lg(n/k)$. Each level of merging is still time cn, so putting it together, the merging takes time $\Theta(nlg(n/k))$

(d) We are dealing with constant factors, so k should be a value that is the largest in a list, but beats it still gives the result of Insertion Sort beating Merge Sort. In practice, k should be the largest list length on which insertion sort is faster than merge sort.

Problem 2.2 Recurrences

Use the substitution method, the recursion tree, or the master method to derive upper and lower bounds for $T(n)$ in each of the following recurrences. Make the bounds as tight as possible. Assume that $T(n)$ is constant for $n \leq 2$.

(a) $T(n) = 36T(n/6) + 2n$
1. Extract $a, b, f(n)$

$$a = 36, b = 6, f(n) = 2n$$

2. Determine $n^{\log_b a}$

$$n^{\log_b a} = n^{log_6 36} = n^2$$

3. Compare $f(n)$ and $n^{\log_b a}$

Since $n^2 > 2n$, then $n^{log_b a} > f(n)$
4. Determine the appropriate case and apply it
Thus case 1:

$$f(n) = O(n^{2-\epsilon}) \implies$$

where $\epsilon = 1$

$$T(n) = \Theta(n^2)$$

(b) $T(n) = 5T(n/3) + 17n^{1.2}$
1. Extract $a, b, f(n)$

$$a = 5, b = 3, f(n) = 17n^{1.2}$$

2. Determine $n^{\log_b a}$

$$n^{\log_b a} = n^{log_3 5}$$

3. Compare $f(n)$ and $n^{\log_b a}$
Since $n^{log_3 5} > 17n^{1.2}$, then $n^{log_b a} > f(n)$
Remark: $log_3 5 \approx 1.46$
4. Determine the appropriate case and apply it
Thus case 1:

$$f(n) = O(n^{log_3 5-\epsilon}) \implies$$

where $\epsilon \approx 0.26$

$$T(n) = \Theta(n^{log_3 5})$$

(c) $T(n) = 12T(n/2) + n^2 lgn$
1. Extract $a, b, f(n)$

$$a = 12, b = 2, f(n) = n^2 lgn$$

2. Determine $n^{\log_b a}$

$$n^{\log_b a} = n^{log_2 12} = n^{2+\log_2 3} = n^2 \cdot n^{\log_2 3}$$

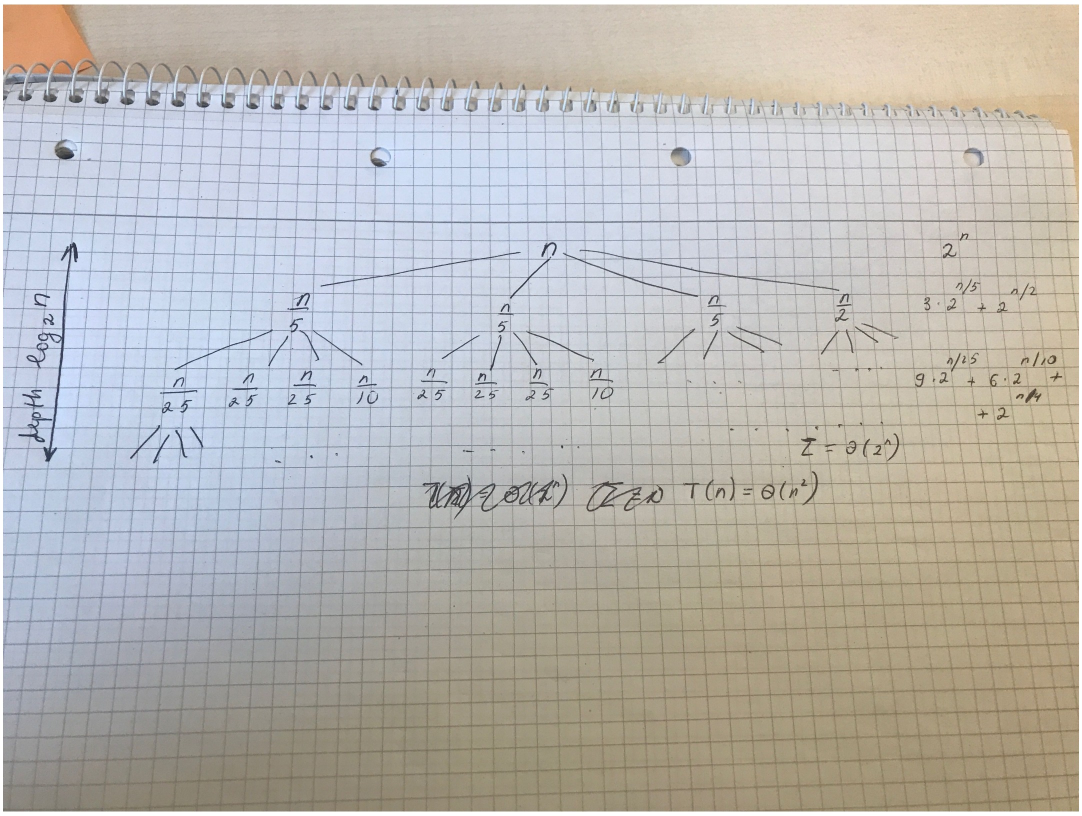
3. Compare $f(n)$ and $n^{\log_b a}$
Since $n^{log_2 12} > n^2 lgn$, then $n^{log_b a} > f(n)$
Remark: $log_2 3 \approx 1.58496$
4. Determine the appropriate case and apply it
Thus case 1:

$$f(n) = O(n^{log_2 3-\epsilon}) \implies$$

where $\epsilon \approx 0.58496$

$$T(n) = \Theta(n^{log_2 12})$$

(d) $T(n) = 3T(n/5) + T(n/2) + 2^n$
Let's solve it by the Recursion Tree Method
2^n dominates the rest of the terms in the finding sum of the tree $(2^{1/2})^n$, $(2^{1/5})^n$ Therefore, the result is $\Theta(n^2)$



(e) $T(n) = T(2n/5) + T(3n/5) + \Theta(n)$
Let's solve it by the Recursion Tree Method
From Recursion Tree Method, the height of the tree is lgn. The sum of nodes at each level equals cn. It follows that $T(n) = \Theta(nlgn)$