## 4 c)

|  |  |  |
| --- | --- | --- |
| Value of MinMergeSort | time in Sorting (s) |  |
| 0 | 3.858233 | (just merge sort) |
| 2 | 0.10273 |  |
| 50 | 0.048814 |  |
| 90 | 0.038675 |  |
| 100 | 0.035966 |  |
| 110 | 0.035634 |  |
| 125 | 0.033003 |  |
| 150 | 0.032559 |  |
| 175 | 0.032922 |  |
| 200 | 0.034055 |  |
| 250 | 0.039215 |  |

As we can see, the best value for MinMergeSort is around 150.

## 4 d)

The asymptotic cost of running InsertionSort once on k items is O(k2),

it's executed around n/k times (since the whole data set would be divided into lots of cases, and each case would contain k elements at most; while the average number of elements in each case is near k)

Thus, the asymptotic cost of all the calls to InsertionSort is O(n\*k)

## 4 e)

suppose the time of ***n*** input cases is time T(n), thus:

T(n) = 2T(n/2) + O(n) = 4T(n/4) + O(n) + 2O(n/2) = ...

This can be viewed as a tree, and if it's substituted until T(1), the height of the tree is h1 = log2n;

However, for this modified MergeSort, it's substituted as T(a) until a<k. The height uncounted is: h2 = log2k.

Thus, the height of the tree of this modified MergeSort is h3 = h1 - h2 = log2n - log2k = log2(n/k)

Therefore, in the end, for this new method:

Sub in h3 and T(k) = O(k2), can get:

Thus, as long as k is less than log(n), the modified MergeSort would still be O(n\*log(n)).

i.e., the maximum of k is log(n).

However, since big-O notation is just a asymptotic notation, it’s also reasonable to take k < ***c***\* log(n), where ***c*** is a constant.

## 5 a)

Firstly, order the options: 1, 2, 3, ... , m. Then sort the votes according to the order, in ascending order.

After sorting, he can look for the data on the 1/4, 1/2 and 3/4 position of the whole set. Count the number of appearance of these data, and if any of them exceed n/2, then this is the winner; if not, then no winner.

## 5 b)

1. Outline of the algorithm:
   1. Basic idea: if one choice appears more than half of the total times, then its frequency minus the total frequency of the rest of the elements must be positive; and that element is what we want.
   2. Storage used: I use two storages in this algorithm: int order, int counter. order is the order of the algorithms (can treat it as “name”), and counter a variable for counting.
   3. Steps: it’s consisted of two loops.
      1. Initialize the counter to be zero and the order is nil.
      2. First scan through the votes, if counter equals to zero, then set the order to be that vote scanned. If counter > 0, there’re two possibilities:
         1. if the order is that vote scanned, then increment that vote by one;
         2. if not, then decrease that vote by one
      3. After this loop,
         1. if the counter is greater than 0, then the order stores the possibly algorithms with the most votes (and possibly not). Then in another loop to check if this algorithm has votes more than half;
         2. if counter is 0, then no algorithm is voted more than half. Thus, none is the winner.
2. Pseudo code for the algorithm:

// assumption: ***votes*** is the array of the votes, and the value is the order of the selected

// algorithms, and ***n*** is the total number of the votes we’ve known.

// initialization

counter = 0, order = nil;

// loop 1

foreach i in votes:

if (counter == 0) order=i;

else

if(order == i) counter++;

else counter--;

end of foreach

if(counter == 0) return false;

// loop 2 to check

counter=0;

foreach i in votes:

if(order==i) counter++

end of foreach

if(counter\*2>n) return order;

else return false;

1. Prove the correctness of the algorithm:
   1. clearly, if the total votes is less than 4, then the prof. can simply look at these votes and decide the winner. Thus the storage can be used is more than ***st = log2(4) = 2***.
   2. we prove loop 1 first.
      1. to prove: if counter > 0, then order is the only possible choice for the frequency to be larger than half.
      2. for the first element, it holds. (counter = 1, order is that element)
      3. suppose it holds for the ***k***th element, then for the ***(k+1)***th element:
         1. if (counter==0): order = the order of the k+1 element.

there is no element exceed half. by adding in this new element, the only possible element to exceed half if this ***(k+1)***th element.

* + - 1. if(counter > 0):
         1. order == the order of the ***(k+1)***th element: counter++

since this order is already the only possibility more than half, by adding in this new ***(k+1)***th element, it’s still the only possible choice.

* + - * 1. order != the order of the ***(k+1)***th element: counter—

by adding in this new element, it makes the total size larger. while since there has already been cumulated a number of “counter” more elements of ***order***, thus, for an element to become a new only possible choice, it has to eliminate all the excess of other elements cumulated before.

Since this new element has made the pool larger, the counter need to decrement by 1.

* + 1. thus, the loop 1 can give us the only possible choice for the winner.
  1. then we look at loop 2:
     1. loop 2 is just to check if that selected algorithm in loop one is the winner, it’s simply counting to check the basic condition if counter > ***n***/2.
     2. This is intuitively correct.
  2. thus, the correctness of the algorithm has been proved.

1. an analysis of the asymptotic performance: since the time spent in both loops are all linear, the total time spent is O(n).

### In addition, an in-efficient divide-and-conquer algorithm:

// assume nil is a special value, can be casted to class Node

class Node{

int order; // the order of the element

int frequency; // the times of appearance of the element

Node(int order, int frequency){

this.order=order; this.frequency = frequency;

}

}

Node count-votes(int start, int end){ // return value: node

if(start +1 >= end) return votes[start]

middle = (start + end) /2;

left = count-votes(start,middle);// store the largest frequency in the left part, and the order

right = count-votes(middle+1, end);

if(left!=nil){

if(right!=nil)

if(left.order==right.order)

return new Node(left.order, left.frequency+right.frequency);

left.frequency += count-frequency(middle+1,right,left.order);

if(left.frequency>(end-start+1)/2) return left;)

}

if(right!=nil){

right.frequency += count-frequency(start,middle,right.order);

if(right.frequency> (end-start+1)/2) return right;

}

return nil; // no element's frequency exceed half

}

// to count the frequency of the value "order" from start to end in the array votes.

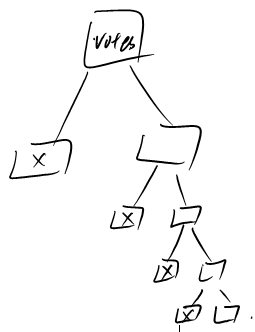
int count-frequency(int start,int end,int order){

int i,count=0;

for(i=start;i<=end;i++) if(votes[i]==order) count++;

return count;

}

Storage used: (as shown in the picture, a cross means its sub-tree has already been checked, and the value is stored in the place of that cross) at most, it would occupy at one node on each level of the tree. Thus the memory use at most is log(n) (the height of the tree).

asymptotic performance of this algorithm: for each level of the tree, it needs to scan all the elements in the set “votes”; and the height of the tree is log(n). Thus, this algorithm’s worst case needs O(n\*log(n)) time. It’s much less efficient than the first algorithm given.

## 5 c)

I don’t have any good idea on how to solve this so far…

(It’s really good to start doing the assignment earlier…)

Reference:

1. I learnt the idea of counting from Boyer–Moore’s vote algorithm, URL:

<http://www.cs.utexas.edu/~moore/best-ideas/mjrty/index.html>

A Linear Time Majority Vote Algorithm

1. I checked my answer for Question 4c with ***Jiao Jingping*** and ***Ding Mingzhe***, and they helped me to improve my solution to question 4c. In addition, ***Ding Mingzhe*** also helped me a little bit with question 5 b.