CS2020 Problem Set 2 Solution

School of Computing - National University of Singapore

February 1, 2011

Problem 4

Problem 4.a

You are required to implement the insertion sort. Validation on the input is a must and be careful with the base case.

```
private void InsertionSortWords(int begin, int end) throws Exception
{
    // Check inputs
    if ((m_WordList==null) || (m_FileWordCount==0)){
        throw new Exception("Failed in InsertionSortWords: no words to sort.");
    }

    if ((begin>=end) || (begin<0) || (end>m FileWordCount)){
        throw new Exception("Invalid Input.")
    }

    int index = begin;
    String SortString = "";

    //iterate from begin to end.
    for (int iMaxSorted = begin; iMaxSorted<end; iMaxSorted++){

        SortString = m_WordList[iMaxSorted+1];
        index = iMaxSorted+1;
        while (index > begin && SortString.compareTo(m_WordList[index-1]) < 0)
            {
                m_WordList[index] = m_WordList[index-1];
                index--;
            }

        m_WordList[index] = SortString;
    }

    m_Sorted = true;
}


// for the change in main java file
        if (NumWords <= MinMergeSize)
                {
                        InsertionSortWords(Begin,End);                          //
                        return;                                                 //
                }
```

Problem 4.b

You are supposed to fix the value of **MinMergeort** to improve the overall performance of merge sort.

Problem 4.c

Here you are required to run profiling with TPTP to identify the best value of **MinMergeSort**. In this case, we expect you to present the test cases and give a little explanation on how you decide the size of test cases.

In general, the optimal value lies in range [100..200].

Problem 4.d

Let n be the number of elements to sort and k be the **MinMergeSort**. Every time we run insertion sort on k items, it takes tim $O(k^2)$. Approximately insertion sort will be called for $\frac{n}{k}$ times and asymptotic cost sums up to be $O(nk)$.

Problem 4.e

We are going to compute the asymptotic cost of improved MergeSort.

For N elements, the worst case of MergeSort is $O(N \log N)$. Since we are going to have $\frac{N}{k}$ sublists with length of k, the worst case to merge those $\frac{N}{k}$ sublists can be evaluated through a tree analysis: there are $\log \frac{N}{k}$ levels with constant cost N for merging on each, $N \times \log \frac{N}{k}$.

In total, the asymptotic cost of modified MergeSort is equal to the sum of insertion sort and merge sort: $O(NK) + O(N \log \frac{N}{k})$ or $O(NK) + O(N \log N)$. Since we are analysing asymptotic cost, we can get the improvement in time if $k \leq c \log N$, where c is a constant. $k = \log N$ is an acceptable answer, however, since we are doing asymptotic analysis, $c \log N$ turns out to be a better answer.

Problem 5

Problem 5.a

Prof Acker has a large pile of N papers on which we can perform mergesort/quicksort. No extra space is required.

After the array V is sorted, we can iterate through the array V once and check if any algorithm have more than $\frac{N}{2}$ members. The asymptotic cost of sorting is $O(N \log N)$ and the check takes $O(N)$, hence the total cost is the same as the sorting, $O(N \log N)$.

Instead of checking all elements in the sorted array, we can check the middle elements in the V. If there exists a majority algorithm X, X must take the place in $V[\frac{N}{2}]$. Hence we can perform the check from the mid-point of V. Asymptotic analysis remains the same.

Problem 5.b

In this case, Prof Ack cannot perform his favourite sorting and he decides to take advantage of Divide-and-Conquer approach. Before Prof Ack started coding, he had the following observation to share with his students:

> If there is a majority element in the array, then one of the two halves has a majority element.

Then he designs the algorithm based on his observation. For an array P:

1. Find a majority element A in the left half (if any)
2. Find a majority element B in the right half (if any)

3. Count how many times of A element in the entire array
4. Count how many times of B element in the entire array
5. if A is the majority, return A as array P's majority element.
6. if B is the majority, return B as array P's majority element.

For step 1 and step 2, Prof Ack follows the idea of Divide-and-Conquer, which divides the original problem to two sub-problems. The pseudocode for this algorithm:

```
FindMajority(begin,end,M){
  //find the majority element in M[begin...end]

  //handle the range with single element only
  //base case
  if(begin == end)
    return M[begin];

  //calculate the midpoint of array
  int mid = (begin+end)/2;

  //recursively divide-and-conquer
  KeyA = FindMajority(begin,mid,M);
  KeyB = FindMajority(mid+1,end,M);

  //count the number of A and B in M
  freqA = countfreq(KeyA,begin,mid);
  freqB = countfreq(KeyB,begin,end);

  //return the majority element if any
  if(freqA  > freqB)
    return A; //A is not necessarily to be the majority
  else
    return B; //B is not necessarily to be the majority
}
```

For asymptotic analysis, $T(N) = 2T(\frac{N}{2}) + O(N)$, similar to MergeSort, in total it takes $O(N \log N)$.

As suggested in the problem, there is a non-Div-and-Conquer way to solve the problem. Basically, we start out with "candidate = first element of the array" and set count=1.Then we iterate through the array (starting with element 2). If the element is the same as candidate, increment the count. If the element is different from candidate, decrement the count. If count==0, then we start with the next element in the array. At the end, the last candidate is the only possible majority element. This method is called *Moore's Voting Algorithm*. Please note that if you happen to Google this solution, you have to cite the reference.

A less elegant solution is that you can iterate the whole array and count each element. Obviously it takes $O(n^2)$ time and Prof Ack won't like it even though it works.

Problem 5.c

Prof Ack likes his Divide-and-Conquer solution to Problem 5.b and now he is going to extend the idea to find the element with at least k appearances. Actually, he got another observation:

> An array of size n has at most k elements that appear at least $\frac{n}{k}$ times.
>
> If an element appears $\frac{n}{k}$ times in an array, then it appears at least $\frac{n}{2k}$ times in either the right half or the left half of the array. (If it appears $< \frac{n}{2k}$ times in both halves, then it appears $\frac{n}{k}$ times in the combined array.)

Thus Prof Ack comes up with the following routine which returns EVERY element that appears at least $\frac{n}{k}$ times in an array.

```
find-frequent(A[1..n], n, k)
  if (n < k) then
       Count how many times each element in A appears.
       Return every element that appears at least once.

  L = find-frequent(A[1..n/2], n/2, k)
  R = find-frequent(A[n/2+1..n], n/2, k)
  Count how many times each element in L and R appear in A.
  Return every element that appears at least n/k times.
```

The base case here is an array of size k, and is clearly correct since we count how many times each item appears. (Note: this uses $O(k)$ space.) For the recursive step, we know that L contains every item that appears at least n/(2k) times in L, and R contains every item that appears at least n/(2k) times in R. Thus the combined set $L \cup R$ contains every element that appears at least n/k times in A. (Also, note that $L \cup R$ is size at most 2k, so the space needed is $O(k)$ here too.) We check each of the items in $L \cup R$, so we know that we return every element that appears at least n/k times in A.

The running time is $T(n) = 2T(\frac{n}{2}) + O(kn)$ (since it costs $O(n)$ to check each of the $O(k)$ elements in $L \cup R$), and hence the total running time is $O(knlogn)$. The space usage is $S(n) = S(\frac{n}{2}) + O(k) = O(k \log n)$. This solution accomplishes this without writing to the array.