CS 413 Homework 3

Sterling Jeppson

February 14, 2021

Problem 1

First we note that according to the definition of median provided in the problem, the median of a list of $2n$ elements will be greater than exactly $n-1$ elements and less than exactly n elements. Now let D_1 and D_2 be two databases which we can query. Query the two databases to obtain the median from each of them. Regardless of the parity of n, the median will be in position k when $k = \lceil n/2 \rceil$. Determine whether $D_1(k) < D_2(k)$ or $D_1(k) > D_2(k)$. Since the two scenarios are exactly analogous we will proceed with the explanation in the case that $D_1(k) < D_2(k)$.

We queried the $\lceil n/2 \rceil \geq \lfloor n/2 \rfloor$ element in D_1. Since $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, we can conclude that the first $\lfloor n/2 \rfloor$ elements in D_1 are all less than at least the last $\lceil n/2 \rceil$ elements in D_1. Also, since $D_1(k) < D_2(k)$, and since the kth element in D_2 is less than the last $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ elements in D_2, it follows that the first $\lfloor n/2 \rfloor$ elements in D_1 are all less than at least the last $\lfloor n/2 \rfloor + 1$ elements in D_2. Thus, the first $\lfloor n/2 \rfloor$ elements in D_1 are all less than at least $\lceil n/2 \rceil + \lfloor n/2 \rfloor + 1 = n + 1$ elements in the combined database. Hence the first $\lfloor n/2 \rfloor$ elements in D_1 cannot contain the median as they violate the property that the median is less than exactly n elements.

We queried the $\lceil n/2 \rceil \geq \lfloor n/2 \rfloor$ element in D_2. It is a fact that the last $\lfloor n/2 \rfloor$ elements in D_2 are all at least as large as the first $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ elements in D_2. Also, since $D_2(k) > D_1(k)$, and since the kth element in D_1 is greater than or equal to the first $\lfloor n/2 \rfloor$ elements in D_1, it follows that the last $\lfloor n/2 \rfloor$ elements in D_2 are all greater than at least the first $\lfloor n/2 \rfloor$ elements in D_1. Thus, the last $\lfloor n/2 \rfloor$ elements in D_2 are all greater than at least $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ elements in the combined database. Hence the last $\lfloor n/2 \rfloor$ elements in D_2 cannot contain the median as they violate the property that the median is greater than exactly $n-1$ elements.

Essentially the algorithm is as follows: First, find the median of the two subsets of the two databases and compare them for size. Second, eliminate from the the subset with the lower median the first $\lfloor n/2 \rfloor$ elements and you eliminate from the subset with the greater median the last $\lfloor n/2 \rfloor$ elements. Third, recursively call the function to find the median of the two subsets produced by part 2. Fourth, return the min of medians when the size of the subsets in 1.

This algorithm relies on the fact that the median of the newly produced subsets from step 2 is the same as the median from the original subsets. But this is true because none of the elements eliminated are the median, the same number of elements are eliminated from both of the subsets, and all of the elements eliminated from one subset are less than the median and all of the elements eliminated from the other subset are greater than the median. Here is the algorithm.

Algorithm to find median of two independent databases

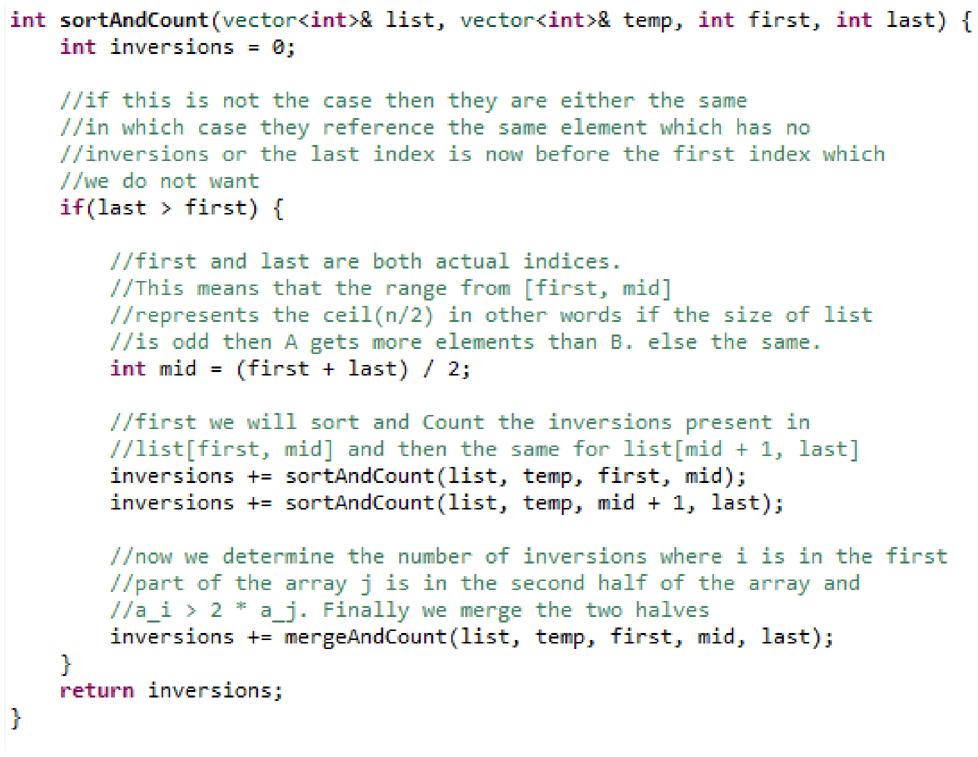
```
function MEDIAN($n$, $i$, $j$)
    $k = \lceil n/2 \rceil$
    if $n == 1$ then
        return MIN($D_1(i+1), D_2(j+1)$)
    end if
    if $D_1(i+k) < D_2(j+k)$ then
        return MEDIAN($k$, $i + \lfloor n/2 \rfloor$, j)
    else
        return MEDIAN($k$, $i$, $j + \lfloor n/2 \rfloor$)
    end if
end function
```

In the above algorithm, n is the number of elements in each of the database subsets, i is 1 less than the minimum index of the D_1 which is the first index of the subset of D_1 passed to the function. j has an analogous purpose except it applies to D_2. For example, when we initially call the function, suppose that $i = 0$ and $n = 5$. Suppose that we eliminate the first $\lfloor n/2 \rfloor$ of D_1 after the first function call. Then in the second function call, $i = 2$. This means that the first 2 entries of D_1 are no longer possible locations for the median. Also remember that in pseudo-code the indices start at 1 not 0 and so i is exclusive not inclusive. From this explanation it is clear the way that the algorithm eliminates the smaller $\lfloor n/2 \rfloor$ entries. It is not as clear how the larger $\lfloor n/2 \rfloor$ are eliminated. This is done through the calculation of k in the beginning of every function call. Since $k = \lceil n/2 \rceil$ and since $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, by starting from an exclusive min index and adding $\lceil n/2 \rceil$, the top $\lfloor n/2 \rfloor$ will be excluded. The reason that we return the min of the two medians when the size of the database subsets is 1 is because the problem defined the median to be the element that is greater $n-1$ elements. So if there are only 2 elements in the combined database it is necessary to choose the first one which is smaller. To find the median of the two databases you would run the algorithm with input MEDIAN$(n, 0, 0)$.

Finally we need to establish a run-time bound on the algorithm. Every time the function is called the problem is reduced into a version of itself that is $\lceil n/2 \rceil$ times as large as the initial call. Also, in every function call 2 queries are performed. It follows that the number of queries $Q(n) \leq Q(\lceil n/2 \rceil) + 2$ when $n > 1$, and $Q(1) = 2$. The function requires $\lceil log_2 n \rceil$ recursive calls to reduce n to 1 and does 2 queries in every call. So $Q(n) = 2 \cdot \lceil \log_2 n \rceil + 2$. This is $O(\log n)$.

Problem 2

This problem is very similar to the example given in section 5.3. We implement the algorithm with the same two functions with a slight modifications on the Merge-and-Count procedure. The Sort-and-Count procedure is given as follows:



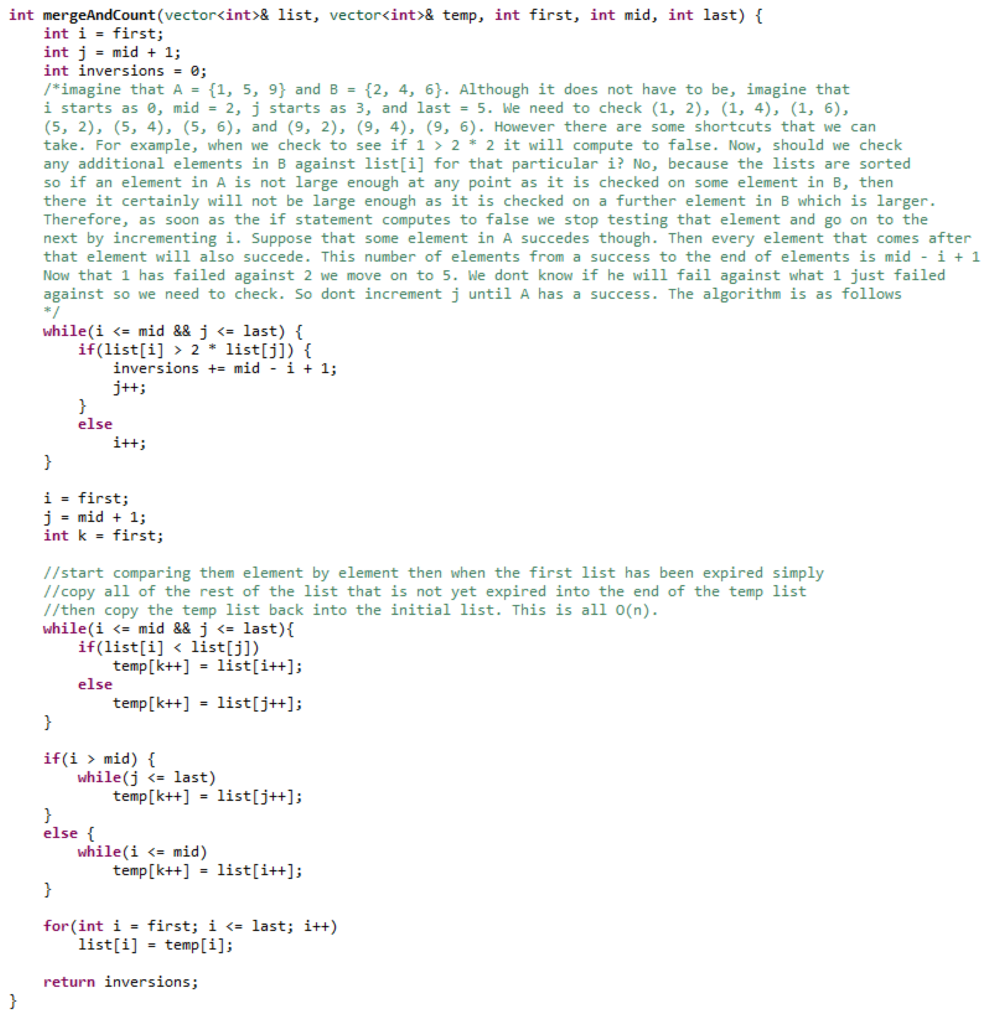
```
int sortAndCount(vector<int>& list, vector<int>& temp, int first, int last) {
    int inversions = 0;

    //if this is not the case then they are either the same
    //in which case they reference the same element which has no
    //inversions or the last index is now before the first index which
    //we do not want
    if(last > first) {

        //first and last are both actual indices.
        //This means that the range from [first, mid]
        //represents the ceil(n/2) in other words if the size of list
        //is odd then A gets more elements than B. else the same.
        int mid = (first + last) / 2;

        //first we will sort and Count the inversions present in
        //list[first, mid] and then the same for list[mid + 1, last]
        inversions += sortAndCount(list, temp, first, mid);
        inversions += sortAndCount(list, temp, mid + 1, last);

        //now we determine the number of inversions where i is in the first
        //part of the array j is in the second half of the array and
        //a_i > 2 * a_j. Finally we merge the two halves
        inversions += mergeAndCount(list, temp, first, mid, last);
    }
    return inversions;
}
```

Figure 1: Sort-and-Count Procedure Implemented in C++

The explanation to this function is essentially the same as the explanation to the books explanation to the problem presented in 5.3. In order to find the significant inversions in an array or vector, we first divide the list roughly in half, calculate the number of inversions in the two sub-vectors with a recursive call, and finally merge the two sorted sub-vectors and count the number of inversions from the left sub-vector to the right sub-vector. This function has two recursive calls where the work done in each call besides the function Merge-and-Count is in constant time. Thus far this function has run time (assuming an even input) is

$$T(n) \leq 2T(n/2) + O(?)$$

Therefore it is critical to determine if the modifications made to the Merge-and-Count procedure have kept the procedure $O(n)$ time or not. The modified procedure is given below.



```
int mergeAndCount(vector<int>& list, vector<int>& temp, int first, int mid, int last) {
    int i = first;
    int j = mid + 1;
    int inversions = 0;
    /*imagine that A = {1, 5, 9} and B = {2, 4, 6}. Although it does not have to be, imagine that
    i starts as 0, mid = 2, j starts as 3, and last = 5. We need to check (1, 2), (1, 4), (1, 6),
    (5, 2), (5, 4), (5, 6), and (9, 2), (9, 4), (9, 6). However there are some shortcuts that we can
    take. For example, when we check to see if 1 > 2 * 2 it will compute to false. Now, should we check
    any additional elements in B against list[i] for that particular i? No, because the lists are sorted
    so if an element in A is not large enough at any point as it is checked on some element in B, then
    there it certainly will not be large enough as it is checked on a further element in B which is larger.
    Therefore, as soon as the if statement computes to false we stop testing that element and go on to the
    next by incrementing i. Suppose that some element in A succedes though. Then every element that comes after
    that element will also succede. This number of elements from a success to the end of elements is mid - i + 1
    Now that 1 has failed against 2 we move on to 5. We dont know if he will fail against what 1 just failed
    against so we need to check. So dont increment j until A has a success. The algorithm is as follows
    */
    while(i <= mid && j <= last) {
        if(list[i] > 2 * list[j]) {
            inversions += mid - i + 1;
            j++;
        }
        else
            i++;
    }

    i = first;
    j = mid + 1;
    int k = first;

    //start comparing them element by element then when the first list has been expired simply
    //copy all of the rest of the list that is not yet expired into the end of the temp list
    //then copy the temp list back into the initial list. This is all O(n).
    while(i <= mid && j <= last){
        if(list[i] < list[j])
            temp[k++] = list[i++];
        else
            temp[k++] = list[j++];
    }

    if(i > mid) {
        while(j <= last)
            temp[k++] = list[j++];
    }
    else {
        while(i <= mid)
            temp[k++] = list[i++];
    }

    for(int i = first; i <= last; i++)
        list[i] = temp[i];

    return inversions;
}
```

Figure 2: Merge-and-Count Procedure Implemented in C++

The difference between this function and the function given in 5.3 is that this function counts the inversions and merges the sorted lists separately. I can't precisely state why this is, but I can offer some statements. In 5.3, inversion $\implies$ disorder and disorder $\implies$ inversion. This means that whenever we encounter disorder, we immediately know the number of inversions and so we can continue to place the numbers into the temp array in $O(n)$ time. However in this problem, inversion $\implies$ disorder, but disorder $\not\implies$ inversion. But we are not presented with inversion but with disorder. This means that when we encounter disorder we cannot continue to place at least 1 sorted element into the temp array. Here is a concrete example of what happens when we try to apply the technique from 5.3 to this problem. Say that the left-hand array is 1, 6, 9, and the right-hand array is 2, 3, 4. We see no inversions with 1 and 2 so we put 1 into the temp array and increment the left-hand array pointer to 6. Now there is an inversion so we record 1 for 6 and 1 for 9. Now we put 2 into the temp array so that it now contains 1 and 2. Next we increment the pointer in the right-hand array to point to 3. This is where the problem occurs. What can we do? There is disorder but there is not an inversion. We can't put 3 into the temp array because there may be inversions with 3 that we have not yet considered. It happens that there are, for example 9 and 3. But we can't put 6 into the temp array because it does not go before 3. We would need to leave 6 and check 3 for inversions on indices of the left-hand array greater than the index which contains 6. We would need to do this until we found an inversion with 3 or until we ran out of elements in the left-hand array. Then we can finally place 3. But now we need to return to 6 to test 6 for size comparison. And if we found another element in the right-hand side array then we would need to again repeat the procedure that we performed for 3 before we could finally place 6. Clearly this is a terrible situation. But what I am not certain about is whether this is simply an annoying situation to program around or is it more than that? If it is the case that such a technique necessarily produces a nonlinear running time, then the algorithm as a whole would fail to be $O(n \log n)$.

The solution to this problem is to calculate the number of inversions and merge the sub-vectors in sorted order in two separate $O(n)$ steps. I did not prove the correctness of the first part of this algorithm but the comments in the above function provide a convincing explanation as to how the while loop counts inversions. One thing that should be proved however is that the while loop terminates in $O(n)$ time. Observe that in each iteration of the while loop, either i or j will be incremented by 1. Since the condition for the while loop to terminate is that $i >$ mid or $j >$ last, the loop can only iterate a max of mid - first + last - (mid + 1) + 1 = $\lfloor (n-1)/2 \rfloor - 0 + (n-1) - (\lfloor (n-1)/2 \rfloor + 1) + 1 = n-1$ times. Hence the counting of the inversions is in $O(n)$ time where n is the length of the combined sub-vectors. The merge portion of the function is no different than the standard merge sort algorithm and runs in $O(n)$ time. It follows that the entire Merge-and-Count function runs in $O(n)$ time.

Now we assume an input array with an even number of elements and we can conclude that the running time of the entire algorithm is

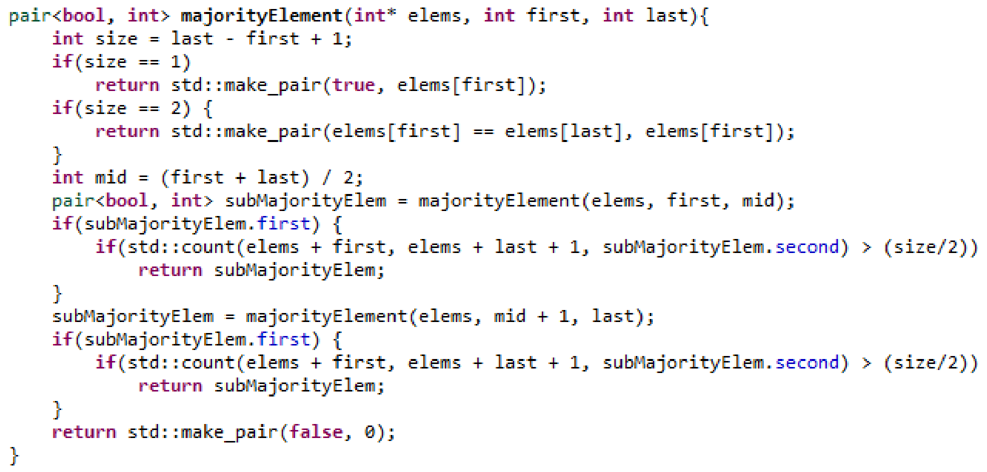
$$T(n) \leq 2T(n/2) + O(n)$$

Hence $T(n)$ satisfies recurrence 5.1 and so by 5.2, the Sort-and-Count algorithm runs in time $O(n \log n)$.

Problem 3

For this problem observe that if a list of n values has a majority value x such that x is the value of more than $n/2$ elements, then x must be the majority value of the first $n/2$ values or the last $n/2$ values. Suppose not. Then count(x) in the first $n/2$ values is less than or equal to $n/4$ and count(x) in the last $n/2$ values is less than or equal to $n/4$. It follows that count(x) from [1, $n/2$] + count(x)

from $[n/2 + 1,\ n]$ is less than or equal to $n/2$ which is a contradiction. Here is how the algorithm will work. Divide the input list into two equal sub-lists and determine if there is a possible majority element in the first sub-list with a recursive call. If there is a possible majority element, then do a linear check of that element on all of the elements to see if there are greater than $n/2$ instances. If there are then we return that element as a majority element. If there is no possible majority element or if the linear search fails to find a majority element, do a recursive call on the second half of the list. Once again check if there is a possible majority element and run a linear search on that element if there is. If the linear search fails then return no element. If it succeeds then return the majority element. The base case will be $n = 1$ in which case you return that there is a majority element, namely the element, and $n = 2$ in which case you compare the two elements and return one of them if they are the same or none if they are different. The algorithm is given below.



```
pair<bool, int> majorityElement(int* elems, int first, int last){
    int size = last - first + 1;
    if(size == 1)
        return std::make_pair(true, elems[first]);
    if(size == 2) {
        return std::make_pair(elems[first] == elems[last], elems[first]);
    }
    int mid = (first + last) / 2;
    pair<bool, int> subMajorityElem = majorityElement(elems, first, mid);
    if(subMajorityElem.first) {
        if(std::count(elems + first, elems + last + 1, subMajorityElem.second) > (size/2))
            return subMajorityElem;
    }
    subMajorityElem = majorityElement(elems, mid + 1, last);
    if(subMajorityElem.first) {
        if(std::count(elems + first, elems + last + 1, subMajorityElem.second) > (size/2))
            return subMajorityElem;
    }
    return std::make_pair(false, 0);
}
```

Figure 3: Majority-Element Procedure Implemented in C++

In the above algorithm there are a maximum of two recursive calls and each call does $O(2n) = O(n)$ work: $O(n)$ work for searching the list for a majority element if there is a possible majority element on the left sub-array, and $O(n)$ work for searching the list for a majority element if there is a possible majority element in the right sub-array. Hence the running time of the algorithm $T(n)$ satisfies the following recurrence relation:

$$T(n) \leq 2T(n/2) + 2n$$

It now follows by 5.2 that $T(n)$ is $O(n \log n)$.