Solutions to Problem Set 2

Solution to Problem 1  This problem is similar to the Partitioning problem used in Quick Sort. Our idea is to maintain an interval of red items, followed by an interval of white items, followed by an interval of blue items, followed by an interval of unseen items. Depending on the color of the current item, we will either leave the item alone, or swap it with one or two previous items. Here is code written in (Turbo) Pascal:

```pascal
first_white := 1;
first_blue := 1;
for current := 1 to n do
  case A[current].color of
    white:
    begin
      swap(A[first_blue], A[current]);
      inc(first_blue)
    end;
    red:
    begin
      swap(A[first_blue], A[current]);
      swap(A[first_white], A[first_blue]);
      inc(first_blue);
      inc(first_white)
    end
  end
```

Because each iteration of the for loop takes constant time, the running time is $\Theta(n)$. Only a constant number of extra variables are used.

Solution to Problem 2  Define the signature of a word to be the alphabetical ordering of its letters. (For example, the signature of “dopiest” is “deiopst”. Note that two words are anagrams if and only if they have the same signature.

First, we will compute the signature of each word. We will use a batch version of bucket sort for this step. Specifically, we will have 26 buckets labelled from ‘a’ to ‘z’. For each word in the dictionary, and for each letter of that word, put a pointer to that word in the appropriate bucket. Then by making one scan of the buckets, we can recover the signature of each word. (Doing a bucket sort simultaneously on all the words reduces the overhead of scanning the buckets.) This first step takes time proportional to the sum of the word lengths (plus 26).

Second, we need to sort the signatures. Group the words according to their lengths. Then use radix sort on the words of a particular length, letter by letter.
This second step also takes time proportional to the sum of the word lengths, because the overhead of scanning the buckets is dominated by the actual work.

Finally, we just make one pass through the list, putting all words with the same signature on the same line of output. Trivially, this step takes time proportional to the sum of the word lengths.

So, all together, the time is proportional to the sum of the word lengths.

I borrowed this algorithm from Zhe Yang, one of our TAs, who used ideas from a paper by Jiazhen Cai and Robert Paige.

This problem was posed by Jon Bentley (1986), *Programming Pearls*, Addison-Wesley, Column 2. He suggested using Insertion Sort in the first step, but Bucket Sort seems better.

**Solution to Problem 3** We will first design a (recursive) procedure that computes two things: the smallest item, and a small set of candidate items (one of which is the second-smallest item).

```pascal
procedure find_min (const A[first..last];
    var smallest;
    var candidates);
var
    mid;
    left_smallest, right_smallest;
    left_candidates, right_candidates;
begin
    if first = last then
        begin
            smallest := A[first];
            candidates := empty_set
        end
    else
        begin
            mid := (first + last) div 2;
            find_min(A[first..mid], left_smallest, left_candidates);
            find_min(A[mid+1..last], right_smallest, right_candidates);
            if left_smallest <= right_smallest then
                begin
                    smallest := left_smallest;
                    candidates := left_candidates union {right_smallest}
                end
            else
                begin
                    smallest := right_smallest;
                    candidates := right_candidates union {left_smallest}
                end
        end
end;
```
It is not hard to prove (by induction on \( n \), the number of items) two properties:

1. The number of item comparisons made by the procedure is exactly \( n - 1 \).
2. The number of candidates is at most \( \lceil \lg n \rceil \).

After we call the procedure, we just need to compute the minimum of the candidates. By Property 2, this last step uses at most \( \lceil \lg n \rceil - 1 \) comparisons. Hence, by Property 1, the total number of comparisons is at most \( n + \lceil \lg n \rceil - 2 \).

**Solution to Problem 4** First find the median value; we can compute the median in worst-case time \( \Theta(n) \) as described in Section 10.3 of Cormen et al. We claim that if there exists a majority value, then it must be the median value. Why? Well, suppose there exists a majority value \( M \). By definition of majority, fewer than \( n/2 \) items are different from \( M \). In particular, fewer than \( n/2 \) items are less than \( M \), and fewer than \( n/2 \) items are greater than \( M \). Hence \( M \) must be the median value.

To check whether the median value really is the majority value, just make one scan of the array. The scan takes time \( \Theta(n) \), so the total time is \( \Theta(n) \).

Although the algorithm above is asymptotically optimal, the constant factors are large. We present a second algorithm that is more practical. It has the advantage of relying only on equality checks between items, not comparisons. We first find a candidate for the majority value.

```plaintext
excess := 0;
for current := 1 to n do
    begin
        if excess = 0 then
            candidate := A[current];
            if A[current] = candidate then
                inc(excess)
            else
                dec(excess)
        end

This code takes time \( \Theta(n) \). It is not hard to prove that if there exists a majority value, then it must be \( \text{candidate} \). Checking whether \( \text{candidate} \) really is the majority takes time \( \Theta(n) \) as before, so the total time is again \( \Theta(n) \).

I copied this second algorithm from Yu-Sung Chang, a student in our class. See also the following two papers:
