Sample Solution to Problem Set 2

1. (15 points) Computing forces in particle physics

Chapter 5, Exercise 4, page 247.

Answer: This can be solved using a convolution. Define one vector to be $a = (q_1, q_2, \ldots, q_n)$ and the other vector to be $b = (1/n^2, 1/(n-1)^2, \ldots, 1/4, 1, 0, -1, -1/4, \ldots, -1/(n-1)^2, -1/n^2)$. Now, for each j, the convolution of a and b will contain an entry of the form

$$\sum_{i<j} \frac{q_i}{(i-j)^2} + \sum_{i>j} \frac{-q_i}{(i-j)^2}.$$

We simply have to multiply the above term by Cq_j to get the desired net force F_j.

Setting up the convolution vectors takes $O(n)$ time (note that the first vector has n entries and the second $2n+1$), finding the convolution $O(n \log n)$ time, and obtaining the forces an additional $O(n)$ time. So the total time is $O(n \log n)$.

2. (15 points) Hidden surface removal in computer graphics

Chapter 5, Exercise 5, page 248.

Answer: We first label the lines in order of increasing slope, and then use divide and conquer. If $n \leq 3$, then we can easily solve it in constant time. The first and third lines will always be visible, the second will be visible if and only if it meets the first line to the left of where the third line meets the first line. We can compute the intersection points and compare to obtain the visible lines.

Let $m = \lceil n/2 \rceil$. We first recursively compute the sequence of visible lines among $L_1, \ldots, L_m$. Let the output of the recursion be $\mathcal{L} = \{L_{i_1}, \ldots, L_{i_p}\}$ in order of increasing slope. We also compute, in this recursive call, the sequence of points $a_1, \ldots, a_{p-1}$ where a_k is the intersection of line L_{i_k} with $L_{i_{k-1}}$. Notice that $a_1, \ldots, a_{p-1}$ will have increasing x-coordinates; for if two lines are both visible, the region in which the line of smaller slope is uppermost lies to the left of the region in which the line of larger slope is uppermost. Similarly, we recursively compute the sequence $\mathcal{L}' = \{L_{j_1}, \ldots, L_{j_q}\}$ of visible lines among $L_{m+1}, \ldots, L_n$, together with the sequence of intersection points $b_k = L_{j_k} \cap L_{j_{k-1}}$, for $k = 1, \ldots, q-1$.

To complete the algorithm, we need to design the conquer step. That is, we must show how to determine the visible lines in $\mathcal{L} \cup \mathcal{L}'$, together with the corresponding intersection points, in $O(n)$ time. (Note that $p + q \leq n$, so if we run in linear time, we are done.) We know that L_{i_1} and L_{j_q} are both visible since they have the minimum and maximum slopes, respectively, among all lines.

We merge the sorted lists $a_1, \ldots, a_{p-1}$ and $b_1, \ldots, b_{q-1}$ into a single list of points $c_1, c_2, \ldots, c_{p+q-2}$ ordered by increasing x-coordinates. This takes $O(n)$ time. For each k, we consider the line that is uppermost in $\mathcal{L}$ at x-coordinate of c_k, and the line that is uppermost in $\mathcal{L}'$ at x-coordinate of c_k. Let ℓ be the smallest index for which the uppermost line in $\mathcal{L}'$ lies above the uppermost line in $\mathcal{L}$ at x-coordinate c_ℓ. Let the two lines at this point be $L_{i_s} \in \mathcal{L}$ and $L_{j_t} \in \mathcal{L}'$.

Let (x^*, y^*) denote the point at which L_{i_s} and L_{j_t} intersect. We thus see that x^* lies between the x-coordinates of $c_{\ell-1}$ and c_ℓ. This means that L_{i_s} is uppermost and immediately to the left of x^*, and L_{j_t} is uppermost immediately to the right of x^*. Consequently, the sequence of visible lines among $\mathcal{L} \cup \mathcal{L}'$ is $L_{i_1}, \ldots, L_{i_s}, L_{j_t}, \ldots, L_{j_q}$ and the sequence of intersection points is $a_{i_1}, \ldots, a_{i_s-1}, (x^*, y^*), b_{j_{t+1}}, \ldots, b_{j_{q-1}}$. This is exactly what we need to return to the next level of the recursion.

The running time of the conquer step is $O(n)$ since we are only doing a linear scan of the lists with comparisons.

3. (3 + 5 + 7 = 15 points) Modes and majority elements

A *mode* of an array $A[1..n]$ is an element of A that occurs most number of times in A. Thus, the mode of an array $A = [7, 4, 12, 4, 1, 1, 4]$ is 4, which occurs three times.

(a) Given an array $A[1..n]$ of n integers, show how a mode of A can be determined in $O(n \log n)$ time.

An array $A[1..n]$ is said to have a *majority element* if more than half of its entries are the same. We would like to determine whether a given array A has a majority element, and if so, find the element. Unlike in part (a), however, we will not restrict the elements to be integers. In fact, we assume that the elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is $A[i] > A[j]$?"; only questions of the form "is $A[i] = A[j]$?" can be answered.

Answer: Sort the n numbers in nondecreasing order. Go through the n numbers in sorted order, keeping a count of the number of occurences of the current element while also maintaining the element that has occured the maximum number of times thus far (also storing its count). At the end of the process, return the element that has occured the maximum number of times.

Sorting takes $\Theta(n \log n)$ time using mergesort. The linear scan through the sorted list takes $\Theta(n)$ time. So the total time is $\Theta(n \log n)$.

(b) Show how to solve the majority element problem in $O(n \log n)$ time.

(*Hint:* Split the array into two arrays A_1 and A_2 of half the size. Use a divide-and-conquer approach that finds the majority element of A, if it exists, using the knowledge of majority elements of A_1 and A_2, if they exist.)

Answer: The following algorithm returns a majority element, if one exists, and $\perp$ otherwise. The basic idea is to split the array into two halves and compute the majority element of each half. If any of these elements is a majority element in the whole array, then return it; otherwise, return $\perp$.

```
1. MAJORITY(A, p, q)
2. if q − p < 1
3. return A[p]
4. else
5.     r ← ⌊(q + p)/2⌋
```

6. $m_1 \leftarrow \textsc{Majority}(A, p, r)$
7. $m_2 \leftarrow \textsc{Majority}(A, r+1, q)$
8. **if** $m_1 = m_2$
9. **return** m_1
10. **else**
11. **if** $m_1 \neq \perp$ and m_1 occurs more than $(q-p+1)/2$ times
12. **return** m_1
13. **if** $m_2 \neq \perp$ and m_2 occurs more than $(q-p+1)/2$ times
14. **return** m_2
15. **return** $\perp$

The algorithm can be proved correct by induction on the size n of the list A. The base case $n = 1$ is clearly correct since the lone element is a majority of the list, and the algorithm does return this element (line 3).

For the induction hypothesis, suppose the algorithm is correct on lists of size smaller than n. Consider the algorithm on a list of length n. Let A_1 and A_2 be the two halves. Let m_1 (resp., m_2) be the majority element of A_1 (resp., A_2), if it exists, and $\perp$ otherwise. By the induction hypothesis, the values m_1 and m_2 computed in lines 6 and 7 are correct. We now claim that no element other than m_1 and m_2 can be the majority element of A. This is because any other element occurs at most $|A_1|/2 + |A_2|/2 = n/2$ times, and thus cannot be a majority element of A. Therefore, the majority element of A can now be computed by counting the number of occurences of m_1 and m_2 in A and returning one, if it occurs more than $n/2$ times, and $\perp$ otherwise. This is exactly what the computations in lines 11 through 15 do.

For the running time, we have a recurrence:

$$T(n) = 2T(n/2) + \Theta(n),$$

which we already know is $\Theta(n \log n)$.

(c) Can you give a linear-time algorithm?

(*Hint:* Another divide-and-conquer approach is as follows: (a) pair up the elements of A arbitrarily, to get $n/2$ pairs; (b) if two elements of a pair are different, then discard both of them, else keep just one of them. Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if A does.)

Answer: The algorithm is already given in the hint. We make it complete as follows.

If A has only one element, then return the lone element. Otherwise: (a) pair up the elements of A arbitrarily, to get $\lfloor n/2 \rfloor$ pairs; (b) if two elements of a pair are different, then discard both of them, else keep just one of them; (c) recursively call the procedure on the remaining elements.

We will argue its correctness and analyze its running time. Let A be a given list of n elements. Furthermore, let A have a majority element and let this element be m. If A has only one element, then the claim is trivial. Otherwise, let B be the list of elements obtained after step (b). We will show that m is also a majority of B, from which the correctness follows using induction. We know that the number of occurences of m in A is greater than the total number of occurences of all other elements in A. In obtaining B from A, whenever we discard m in step (b), we discard one element different than m as well! So the number of occurences

of m in B is also greater than the total number of occurences of all other elements in B. So m is the majority of B as well, completing the proof.

For the running time, we get the recurrence $T(1) = 1$ and for $n > 1$

$$T(n) = T(n/2) + n,$$

which, on unraveling and applying the formula for geometric progressions, yields

$$T(n) = n + n/2 + \ldots = \Theta(n).$$

4. (6 + 6 + 3 = 15 points) A new 3-way sorting algorithm

Consider the following "3-way" sorting algorithm.

```
ThreeWaySort(A, i, j)
1. if A[i] > A[j]
2.    exchange A[i] ↔ A[j]
3. if i + 1 ≥ j
4.    return
5. k ← ⌊(j − i + 1)/3⌋          // Round down.
6. ThreeWaySort(A, i, j − k)     // First two-thirds.
7. ThreeWaySort(A, i + k, j)     // Last two-thirds.
8. ThreeWaySort(A, i, j − k)     // First two-thirds again.
```

(a) Argue that ThreeWaySort$(A, 1, n)$ correctly sorts the input array $A[1..n]$.

Answer: Here is a formal proof of the correctness of ThreeWaySort. (For getting credit, an informal but substantial argument would suffice.)

To prove that ThreeWaySort(A, i, j) sorts correctly, we use induction on $\ell = j - i + 1$. (That is, ℓ is the number of elements in $A[i..j]$.) For the induction basis, we consider two cases: $\ell = 1$ or $\ell = 2$, i.e., $i = j$ or $i = j - 1$. For each of these cases, in Steps 1 and 2, $A[i]$ and $A[j]$ are exchanged if $A[i] > A[j]$. Also, in both cases the condition in Step 3 holds. Therefore, ThreeWaySort(A, i, j) returns after having sorted A.

For the induction hypothesis, we assume that ThreeWaySort sorts correctly for all arrays of length less than m, where $m > 2$. We now show that ThreeWaySort sorts correctly for all inputs of length m. Since $m > 2$, the condition in Step 3 is not true; hence ThreeWaySort does not return in Step 4. Moreover, for the discussion that follows, it does not matter whether the exchange of $A[i]$ and $A[j]$ took place in Step 2. So we now consider Steps 5 through 8. The variable k is set to $\lfloor (j - i + 1)/3 \rfloor$ in Step 5. Since $j - i + 1 \geq 3$, $k \geq 1$; therefore, $j - i - k + 1 = m - k < m$. Therefore, the length of the array $A[i..j - k]$ on which ThreeWaySort is called in Step 6 is less than m. By the induction hypothesis, it follows that $A[i..j - k]$ is in sorted order after Step 6. Hence, A has the following property: (P1) for all x in $[i..i + k - 1]$ and y in $[i + k..j - k]$, $A[x] \leq A[y]$. In particular, we see that there are at least k elements in $A[i + k..j]$ that are greater than or equal to every element in $A[i..i + k - 1]$. Next, ThreeWaySort sorts $A[i + k, .., j]$. Again, the length of the array is less than m. So we invoke the induction hypothesis and claim that after Step 7, $A[i + k..j]$ is sorted. Since

$A[i+k..j]$ is sorted, A has the following property: (P2) $A[j-k+1..j]$ is sorted correctly, and (P3) for all x in $[i+k..j-k]$ and y in $[j-k+1..j]$, $A[x] \leq A[y]$.

Since the elements in indices $[i..i+k-1]$ do not move, the following property (that follows from (P1)) still holds: (P4) there are at least k elements in $A[i+k..j]$ that are greater than or equal to every element in $A[i..i+k-1]$. Therefore, combining with property (P3) above, we have: (P5) for all x in $[i..j-k]$ and y in $[j-k+1..j]$, $A[x] \leq A[y]$. Thus, after step 6, properties (P2) and (P5) hold.

Finally, we apply ThreeWaySort on $A[i..j-k]$. Again the length of the array, $j-k-i+1$ is less than m. So applying the induction hypothesis, we obtain that $A[i..j-k]$ is sorted correctly. Moreover, since the indices $[j-k+1..j]$ are unaffected, properties (P2) and (P5) still hold. We thus obtain that the entire array $A[i..j]$ is sorted correctly.

(b) Give a recurrence for the worst-case running-time of ThreeWaySort and a tight asymptotic (Θ-notation) bound on the worst-case running time.

Answer: Let $T(n)$ be the worst-case running time of ThreeWaySort on an array of length n. Steps 1 through 5 take $\Theta(1)$ time. Steps 6 through 8 take $T(n-k)$ time. Since $k = \lfloor n/3 \rfloor$, $2n/3 \leq n-k \leq 2n/3+1$. Ignoring floors and ceilings, we can write the recurrence as

$$T(n) = 3T(2n/3) + 1.$$

When we unravel the recurrence, we get a pattern like

$$T(n) = 1 + 3 + 3^2 + \ldots$$

In order to calculate this sum, we need to calculate the number of terms. Note that the problem size in the ith level of the unraveling is $(2/3)^i n$. So the problem size the value of i for which $(2/3)^i n = 1$, which gives $i = \log_{3/2} n$. Thus, we have

$$\begin{aligned} T(n) &= 1 + 3 + 3^2 + \ldots + 3^{\log_{3/2} n} \\ &= \frac{3^{\log_{3/2} n + 1} - 1}{2} \\ &= \Theta(n^{\log_{3/2} 3}). \end{aligned}$$

where the second step follows from geometric series and the third step follows from standard manipulation of logarithms.

(c) Compare the worst-case running time of ThreeWaySort with that of insertion sort and mergesort.

Answer: Since $T(n) = \Theta(n^{\log_{3/2} 3})$, and $\log_{3/2} 3 > 2$, ThreeWaySort is worse than each of insertion sort, merge sort, heapsort, and quicksort.