Problem Set 3 Solutions

(Exercises were not to be turned in, but we're providing the solutions for your own interest.)

Exercise 3-1. With groups of 3, the median-of-medians is larger than (about) $n/3$ of the elements, and smaller than (about) $n/3$ of the elements. Thus, the recursive call is on at most $2n/3$ elements. The recurrence is then $T(n) = T(n/3) + T(2n/3) + \Theta(n)$; solving it by Akra-Bazzi, we get only that $T(n) = \Theta(n \log n)$. So in fact with groups of 3, selection may not complete in $O(n)$ time.

With groups of 7, the median-of-medians is larger than (about) $2n/7$ elements. Thus, the recursive call is on at most $5n/7$ elements. The recurrence is then $T(n) = T(n/7) + T(5n/7) + \Theta(n)$, which we solve as $T(n) = \Theta(n)$.

Exercise 3-2. Suppose we are given a "black-box" linear-time algorithm to find the median. We will describe an algorithm to select the ith element out of n unordered elements: first we use the black-box to find the median, then we partition around it, yielding a lower half and upper half. If $i = \lfloor n/2 \rfloor$, then we are done (the median is the element we want). If $i < \lfloor n/2 \rfloor$, we recursively select the ith element from the lower half. If $i > \lfloor n/2 \rfloor$, we recursively select the $(i - \lfloor n/2 \rfloor)$th element from the upper half (which is the ith element of the entire array). A recurrence for the running time of this procedure is $T(n) = T(n/2) + O(n)$, which we solve as $T(n) = O(n)$, as desired.

Exercise 3-3. We define an indicator random variable $I_{x,y}$ to be 1 if $h(x) = h(y)$, and 0 otherwise. Then the number of collisions is the sum of random variables: $\sum_{x \neq y} I_{x,y}$, and we want to know what its expectation is. Using linearity of expectation, we get

$$E\left[\sum_{x \neq y} I_{x,y}\right] = \sum_{x \neq y} E[I_{x,y}] = \sum_{x \neq y} \Pr[h(x) = h(y)].$$

Because we are assuming uniform hashing, $\Pr[h(x) = h(y)] = 1/m$ for every pair $x \neq y$. There are $\binom{n}{2}$ such pairs, so the expected number of collisions is $n(n-1)/2m$.

Exercise 3-4. We prove correctness of RADIX-SORT by induction on the digit positions, from least- to most-significant. When we say the numbers are "sorted by their last i digits," we mean that they are sorted if one only considers their values modulo k^i (where k is the base we're using). Upon initialization, the numbers are trivially sorted by their last 0 digits. Now suppose the numbers are sorted by their last i digits, and we then we run COUNTING-SORT by their $i + 1$st digits. Consider the last $i + 1$ digits of any two elements x and y, where $x \bmod k^{i+1} > y \bmod k^{i+1}$ (we need to prove that x now follows y). If the $i + 1$st digit of x is larger than that of y, then indeed the sort we just performed put x after y. If their $i + 1$st digits are equal, then $x \bmod k^i > y \bmod k^i$, so by the inductive assumption x appeared after y before we sorted by the $i + 1$st digits. By the stability of the sorting subroutine, x still follows y. Therefore the elements are sorted by their last $i + 1$ digits. By induction, after sorting by the most significant digit, the elements are completely sorted.

Problem 3-1. RADIX-SORT on larger ranges

(a) The ith digit is $\lfloor x/r^{i-1} \rfloor \bmod r$. To see this, note that dividing by r^{i-1} and taking the floor is just "shifting right" the base-r representation of x by $i-1$ digits, and taking that value $\bmod r$ is just keeping the least significant remaining digit. If we can compute r^{i-1} in $O(1)$ time, then an algorithm for computing the ith digit is straightforward. Alterately, we can get the digits in sequence by computing $1, r, r^2, r^3, \ldots$, which yields $O(1)$ time per digit.

(b) Using base 10, the numbers have $d = \log n^5 = 5 \log n$ digits. Each COUNTING-SORT call takes $\Theta(n + 10) = \Theta(n)$ time, so the running time of RADIX-SORT is $\Theta(nd) = \Theta(n \log n)$.

(c) If we use base n, the numbers have $d = \log_n n^5 = 5$ digits, and by part **(a)** we can compute each digit in $O(1)$ time. Therefore each COUNTING-SORT call takes $\Theta(n + n) = \Theta(n)$ time, so the running time of RADIX-SORT is $\Theta(nd) = \Theta(n)$. This method extends to numbers in the range $o, \ldots, n^c - 1$ for any constant c.

Problem 3-2. Majority and plurality elements

(a) If a majority element exists, then it must by the median of the elements in the array. To see this, note that *if* the array were in sorted order, and the median were some other element, then there would not be enough room on either side of the median for all the copies of the majority element. Therefore we simply SELECT the median x, and then pass over the array to count the number of times x appears. If it appears more than $n/2$ times, we output x, otherwise we output $\perp$ (because x is not a majority element, and no other element can be either). The running time of this algorithm is $O(n)$ for SELECT, and $O(n)$ to pass over the array again, for $O(n)$ total.

(b) By the above reasoning, any $1/k$-plurality element must have rank in/k for some i in $1, \ldots, k$. Therefore, our algorithm works as follows: for each i in $1, \ldots, k$, SELECT the in/kth order statistic x, then pass over the entire array and output x if x appears at least n/k times.

Each iteration of algorithm takes $O(n)$ time, and there are k iterations, for a running time of $O(nk)$.

Problem 3-3. Set operations

(a) We use any universal hash family $\mathcal{H}$ we have seen in lecture. First, choose a random $h \in \mathcal{H}$, and construct a hash table with $2n$ slots. Then insert every element of S into the hash table using h. Next, for each element $s \in S$ (where $s \neq x/2$), search for $t = x - s$ in the hash table. If it exists, then $s \neq x - s = t$, so output s, t (since both

s and t are in S and sum to x). If $s = x/2 \in S$, then it only occurs once because S is a set, so there is no other element $t \in S$ such that $s + t = x$. If all the searches fail, then output $\perp$, because there is no pair of elements in S that sum to x.

Constructing the hash table takes $O(n)$ expected time, because each hash operation takes $O(1 + \alpha) = O(1 + 1/2) = O(1)$ expected time and we can sum the times by linearity of expectation. The running time of the search stage is again $O(n)$, because each search takes $O(1)$ expected time.

(b) This solution again uses a universal hash family $\mathcal{H}$. We describe an algorithm to determine if $S \subseteq T$; this can be used once in each direction to determine whether the sets are equal.

First choose a random $h \in \mathcal{H}$. Next, build a hash table of size $2n$ and hash all of T's elements into it. Then, for each $s \in S$, search for s in the hash table. If all the searches succeed, then $S \subseteq T$, otherwise not.

Building the hash table takes $O(n)$ expected time, and so do all the searches.

Problem 3-4. Universal and 2-universal hashing

(a) If $\mathcal{H}$ is 2-universal, then for every pair of distinct keys x, y and for every $i \in \{0, \ldots, m-1\}$,

$$\Pr_{h \in \mathcal{H}}[\langle h(x), h(y) \rangle = \langle i, i \rangle] = 1/m^2.$$

The only way for x and y to collide is if they both take on the same value i. Then $\Pr[h(x) = h(y)] = \sum_{i=0}^{m-1} \Pr[\langle h(x), h(y) \rangle = \langle i, i \rangle] = m/m^2 = 1/m$, so $\mathcal{H}$ is universal by definition.

(b) We set $m = 2$, the number of keys to be 2, and $\mathcal{H}$ will have 2 hash functions. Here is the family $\mathcal{H}$:

	x	y
h_1	0	0
h_2	1	0

It is easy to see that $\mathcal{H}$ is universal, because x and y collide only under h_1, which is selected with probability $1/2 = 1/m$. However, $\mathcal{H}$ is not 2-universal, because $\langle h(x), h(y) \rangle$ never equals the pair $\langle 0, 1 \rangle$ or $\langle 1, 1 \rangle$ for any $h \in \mathcal{H}$.

(c) The adversary can in fact force a collision. We modify the above hash family by adding a key z as follows:

	x	y	z
h_1	0	0	1
h_2	1	0	1

Note that this family is still universal: x and z only collide under h_2, and y and z never collide (this is allowed by the definition of universality). Here is how the adversary can force a collision: first we choose a random hash function $h \in \mathcal{H}$, which is either

h_1 or h_2. The adversary gives us x to hash, and learns the value $h(x)$. If $h(x) = 1$, the adversary knows that $h = h_2$, and can force a collision by making us hash z. If $h(x) = 0$, then $h = h_1$ and the adversary forces a collision by making us hash y.

(d) With a 2-universal hash family, the adversary cannot force a collision with probability better than $1/m$. Essentially, knowing $h(x)$ gives the adversary *no information* about $h(y)$ for any other key y. We can prove this formally using conditional probabilities. Suppose we choose a random hash function $h \in \mathcal{H}$, then the adversary forces us to hash some key x and learns the value $h(x) = X$. Then the adversary gives us a key y, hoping to cause a collision. By definition of 2-universal, we have for *any* x, y with $x \neq y$:

$$\Pr_{h \in \mathcal{H}}[h(y) = h(x) \mid h(x) = X] = \frac{\Pr_h[h(y) = h(x) \ \wedge \ h(x) = X]}{\Pr_h[h(x) = X]} = \frac{1/m^2}{1/m} = 1/m.$$

Therefore, no matter which x the adversary first chooses, and which $h(x) = X$ value it learns, the probability of any particular y colliding with x is only $1/m$.

Problem 3-5. Weighted 3-median.

(a) First, we compute W by adding all the weights. Then we sort the x_i elements. Finally, for $j = 1, \ldots, n$, we maintain the running sum $S = \sum_{x_i < x_j} w_i$ by adding the weight of the previous element x_{j-1} with each iteration. As soon as $W - S - w_j \leq 2W/3$ (which must happen before $S > W/3$), we output x_j as the weighted 3-median element.

(b) Assume the elements are given to us in an array X, and their weights in an array Y. Every time we move elements within X, we will make identical moves in Y, to maintain the correspondence between elements and their weights.

We use a divide-and-conquer algorithm to essentially perform a binary search for the proper x_k. The algorithm maintains a "window" $[s, t]$, with the invariant that all elements with ranks s through t are contained in $X[s \, .. \, t]$, and the weighted 3-median x_k is somewhere in $X[s \, .. \, t]$. It also maintains two sums S and T, where S is the total weight of all elements whose ranks are less than s (i.e., $S = \sum_{x_j < x_s} w_j$), and T is the total weight of all elements whose ranks are greater than t. At the start of the algorithm, $[s, t] = [1, n]$ and $S = T = 0$ (with these values, the invariants are clearly true). The algorithm then proceeds as follows:

When searching window $[s, t]$ with S and T, we first find the median x_i of $X[s \, .. \, t]$ and partition $X[s \, .. \, t]$ around x_i. We then compute A to be the total weight of all elements less than x_i in the window $X[s \, .. \, t]$, and B to be the total weight of all elements greater than x_i in the window. There are three cases:

- If $A + S \leq W/3$ and $B + T \leq 2W/3$, then x_i is the weighted 3-median and we return it.
- If $A + S > W/3$, then we let $T \leftarrow B + T + w_i$ and loop on the lower half of $X[s \, .. \, t]$ (that is, all elements in the window smaller than x_i).

- If $B + T > 2W/3$, then we let $S \leftarrow A + S + w_i$ and loop on the upper half of $X[s \,.\,.\, t]$ (all elements in the window greater than x_i).

Given the invariant, the correctness of the first case is apparent (this case also acts as the base case, if the window shrinks down to one element). It is also clear that the final two cases preserve the invariant.

We now analyze the running time of the algorithm: each iteration takes time linear in the size of the window to find the median, partition, and compute A and B. Each iteration halves the size of the window, therefore a recurrence for the running time is $T(n) = T(n/2) + O(n)$, which solves to $T(n) = O(n)$. This completes the proof.