Problem 3-1. Popular Elements

Given an array $A[1..n]$ of $n$ elements, and given a constant $0 < \alpha < 1$, we say that an element $x$ is $\alpha$-popular in $A$ if $x$ appears more than $\alpha n$ times in $A$. Our goal is to find an algorithm that reports all $\alpha$-popular elements in $A$, i.e., that computes the set

$$B_\alpha = \{ x : |\{ i : A[i] = x \}| > \alpha n \}.$$

Our goal is to obtain a running time of $O(n \log(1/\alpha))$.

(a) How many $\alpha$-popular elements can there be, i.e., how large can $|B_\alpha|$ be?

Solution: If $A$ has $n$ elements, we must have less than $1/\alpha$ elements that appear more than $\alpha n$ times. Thus, $|B_\alpha| < 1/\alpha$.

(b) Give a deterministic $O(n)$-time algorithm for computing all $\alpha$-popular elements when $\alpha = 1/2$. Your algorithm should call $\text{SELECT} O(1)$ times. Prove that your algorithm reports the correct answer.

Solution: By part (a), $|B_{1/2}| < 2$, i.e., $|B_{1/2}| \leq 1$. Thus, there can be at most one $1/2$-popular element $x$. If such an $x$ exists, we will show that $x$ must be the median. The algorithm is to compute the median element $y$ of $A$, and then scan $A$ to count whether $y$ appears more than $n/2$ times. Computing the median element $y$ and scanning the array both take $O(n)$ time.

Suppose for contradiction that $x$ is $1/2$-popular, but $y \neq x$ is the median element. Suppose $y < x$; the case $y > x$ is symmetric. If $n$ is odd, then there are at least $(n + 1)/2$ elements greater than $y$ in $A$. If $n$ is even, then there are at least $n/2 + 1$ elements greater than $y$ in $A$. Both contradict the fact that $y$ is the median.

(c) Generalize your algorithm from part (b) to find an $O(n/\alpha)$-time algorithm to compute $B_\alpha$ for any constant $\alpha$. Your algorithm should call $\text{SELECT} O(1/\alpha)$ times.

Solution: Let $s_i = i\lfloor \alpha n \rfloor$, for $i = 1, 2, \ldots \lfloor n/\alpha n \rfloor$, and let $t_i$ be the element in $A$ with rank $s_i$. If any object $x$ appears more than $\alpha n$ times in the list, then we must have $x = t_i$ for some $i$.

The proof of this fact is a generalization of the argument from part (b). Let $A'$ be the sorted version of $A$. Then any element $x$ which is $\alpha$-popular must appear in a
consecutive segment of $A'$ containing at least $\lceil \alpha n \rceil$ elements. By our choice of $s_i$, such a segment must overlap one of the $s_i$’s, i.e., $A'[s_i] = x$ for some $i$. Thus, if an element $x$ is $\alpha$-popular, then it must be one of the $O(1/\alpha)$ order-statistics $t_i$ we computed above. We can compute the $O(1/\alpha)$ order statistics in $O(n/\alpha)$ time using $O(1/\alpha)$ applications of the deterministic select algorithm. Finally, for each of the $O(1/\alpha)$ values $t_i$ we compute, we can scan the array and count the number of occurrences of $t_i$ to determine whether it is $\alpha$-popular. This phase requires a total of $O(n/\alpha)$ time.

(d) Give a more efficient algorithm that computes $B_\alpha$ in worst-case time $O(n \lg (1/\alpha))$. 

**Hint:** Use **PARTITION**.

**Solution:**

The intuition behind solving this part is to notice that as we run (deterministic) quick-sort, we know the rank of each pivot element $x$ after each call to **PARTITION**, and that the rank of these pivot elements are approximately evenly spaced in the array. Thus, we will be able to find $O(1/\alpha)$ candidates for $\alpha$-popular elements as in part (c), and determine whether each element is $\alpha$-popular, all in $O(n \lg 1/\alpha)$ time.

**Algorithm:**

Formally, given a $A$ of length $n$, the **FINDCANDIDATES** looks at the subarray of $A$ indexed between $i$ and $j$, and computes potential $\alpha$-popular elements. **FINDCANDIDATES** first computes the median element of $A[i..j]$ and stores it as a potential candidate for an $\alpha$-popular element. If the subarray has size less than $\alpha n$, the algorithm terminates; otherwise, it partitions $A[i..j]$ about the median, and recurses on each half of the array. As **FINDCANDIDATES** runs, it stores potential candidates in an array $Y$, where each element in $Y[i]$ is a pair of value $Y[i].value$ and rank $Y[i].rank$.

```
1 FINDCANDIDATES(A, i, j, \alpha, n)
2     if (j - i + 1) < \alpha n return
3     k ← \lceil \frac{i + j}{2} \rceil
4     x ← SELECT(A, i, j, k - i + 1) \triangleright Find median element
5     STORECANDIDATE(x, k) \triangleright Remember pivot element x and its rank k
6     PARTITION(A, i, j, k) \triangleright Partition about the median
7     FINDCANDIDATES(A, i, k - 1, \alpha, n)
8     FINDCANDIDATES(A, k + 1, j, \alpha, n)
```

```
1 STORECANDIDATE(x, k)
2     i ← Y.size
3     Y[i].value ← x
4     Y[i].rank ← k
5     Y.size ← Y.size + 1
```
Once we have stored the candidate $\alpha$-popular elements in $Y$, we scan through $A$ to count the number of occurrences of each element $Y[j].value$. One simple way to generate this count is to first sort $Y$, then scan through $A$, and for each $A[i]$, binary search for $A[i]$ and increment its count if it belongs to $Y$.

```
1 FINDPOPULAR(A, n, $\alpha$)
2 Y.size ← 0
3 FINDCANDIDATES(A, 1, n, $\alpha$, n)
4 SORT(Y)  ▷ Sort candidate array Y
5 for $j ← 1$ to Y.size  ▷ Initialize counts for each of the candidates
6   count[j] ← 0
7 for $i ← 1$ to n  ▷ Count the occurrences of each candidate
8   $j ←$ BINARYSEARCHBYVALUE(Y, A[i]) ▷ Returns null if A[i] not found
9   if ($j ≠ \text{null}$)
10      count[j] ← count[j] + 1
11 for $j ← 1$ to Y.size  ▷ Print $\alpha$-popular elements
12   if (count > $\alpha n$) print $Y[j].value$
```

**Correctness:**

PARTITION maintains the following invariant: after calling PARTITION($A$, $i$, $j$, $k$), the rank-$k$ element of $A$ is stored at $A[k]$. After FINDCANDIDATES chooses an element $x = A[k]$ as a pivot, that element $x$ is never moved again. Thus, after FINDCANDIDATES completes, for every pair $Y[i] = (x, k)$ stored into $Y$, $x$ is the element in $A$ with rank $k$.

Using this fact we argue that any element $y$ which is $\alpha$-popular in $A$ must appear in $Y$ after FINDCANDIDATES completes. As we argued in (b), if $y$ is $\alpha$-popular in $A$, it will appear as a string of consecutive $\lceil \alpha n \rceil$ elements in $A'$, the sorted version of $A$. This consecutive segment of $y$'s must overlap at least one pivot element $(x, k)$ in $Y$; otherwise, FINDCANDIDATES must have terminated the recursion when it reaches a subarray $A[i..j]$ of size $(j - i + 1) ≥ \lceil \alpha n \rceil$, contradicting the construction of the algorithm above.

Finally, once we know any $\alpha$-popular element must appear in $Y$, then scanning through the array to count the number of occurrences of each $Y[i]$ gives us the final answer we want.

**Runtime:** We can show that FINDCANDIDATES has a runtime of $O(n \log \frac{1}{\alpha})$. Since SELECT and PARTITION both run in linear time, we know FINDCANDIDATES method has a runtime $T(n)$ given by the following recurrence:

$$T(m) = \begin{cases} 2T\left(\frac{m}{2}\right) + O(m) & \text{if } m > B \\ O(1) & \text{if } m \leq B, \end{cases}$$

where in this case $B = \alpha n$. 
Intuitively, we see that the corresponding recursion tree has approximately $\lg n - \lg B$ levels, with each level performing $O(n)$ work. Thus, we guess that $T(n) = O(n \lg(2n/B))$, and prove this guess via substitution.

In the base case, for $n \leq B$, suppose we know $T(n) = c_0$ for some constant $c_0$. Then, we have $T(n) = c_0 < cB \lg(2B/B) = cB$ if we choose $c > c_0/B$.

In the inductive step, assume for all $B < k < n$, that $T(k) < ck \lg 2k/B$. Then, there exists constants $c_1$ and $n_0$ such that for all $n > n_0$, we have

$$T(n) \leq 2T\left(\frac{n}{2}\right) + c_1n$$

$$\leq 2c\frac{n}{2} \lg \frac{2n}{2B} + c_1n$$

$$= cn \lg \frac{2n}{B} - cn \lg 2 + c_1n$$

$$= cn \lg \frac{2n}{B} - (c - c_1)n.$$ 

If we choose $c > \max\{c_0/B, c_1\}$, then $T(n) \leq cn \lg \frac{2n}{B}$ for all $n > n_0$. Thus, we have $T(n) = O(n \lg \frac{2n}{B}) = O(n \lg \frac{1}{\alpha})$.

Choosing $B = \alpha n$, we get that FINDCANDIDATES has a runtime of $O(n \lg \frac{1}{\alpha})$. We also know that $Y$ has at most $O(1/\alpha)$ elements.

Once we have computed and stored the candidate $\alpha$-popular elements in $Y$, sorting $Y$ takes $O(\frac{1}{\alpha} \lg \frac{1}{\alpha})$ time. To count the occurrences of each candidate requires $n$ calls to BINARYSEARCHBYVALUE, with each call requiring $O(\lg \frac{1}{\alpha})$ time. Thus, the entire algorithm takes $O(n \lg \frac{1}{\alpha})$ time.

**Problem 3.2. Set Operations**

You are a member of the Extragalactic Society, an organization whose goal is to search for signs of life outside the known galaxy. Recently, the captain of an outpost on a remote planet on the Outer Rim has mysteriously disappeared, and you have been designated as his replacement.

Rumor has it that your predecessor was obsessed with the idea that an intergalactic invasion was coming. Thus, oddly enough, he configured the outpost to collect integer data and occasionally search the data to see whether any two of those numbers sum to a particular value $x$.

Despite doubts about your predecessor’s sanity, you have been ordered to continue the operation. You know that your predecessor has stored the existing data as a set of integers $S$ in an array $A$. Given an integer $x$, your task is to search for any two elements $a, b \in S$ that satisfy $a + b = x$.

You search the office, looking for any documentation describing how the data is organized. Apparently your predecessor wasn’t human, because you find a datapad filled with notes written in an obscure alien language you can not understand. The only two words in the entire documentation you can decipher are the words *array* and *sort*.


(b) Assume that the array $A$ is sorted. Prove that, if $A[1] + A[n] > x$, then it is safe to ignore $A[n]$, i.e., $A[n] + b \neq x$ for all $b$ in the array $A$.


(c) Assume that the array $A$ is sorted. Give an $O(n)$-time deterministic algorithm for determining whether there exist two elements $a$ and $b$ in the array that sum to $x$.

**Solution:** We have the following algorithm:

```plaintext
2SUM(A, x)
left ← 1
right ← n
while left ≤ right
    if sum = x return (left, right)
    if sum > x
        right ← right − 1
    else
        left ← left + 1
return false
```


The quantity $right − left + 1$ starts at $n$ and decreases by 1 after every iteration of the while loop. Because there are at most $n$ loop iterations, and each iteration performs a constant amount of work, the algorithm runs in $O(n)$ time.

(d) You implement your algorithm, but are dismayed to discover that the system is returning the wrong answer. After a bit more investigation, you discover that the data is
not actually sorted. Not to be deterred by this minor setback, you proceed to devise another solution that does not require sorting the data.

Give an algorithm that solves this problem in $O(n)$ time in expectation and uses only $O(n)$ space.

Solution: Because we are storing integer data in the array, we can hash each element in the array by its value. We scan through the array, and at each index $i$:

1. Insert an element into a hash table with key $A[i]$ and value $i$. If there already exists an element in the table with key $A[i]$, then do nothing.
2. Compute and search for an element with key $x - A[i]$. If we find such an element with key $x - A[i]$ and value $j$, then return the answer $(i, j)$.

If we scan through all elements in the array without the second step succeeding, then return false.

Arguing correctness of the algorithm is relatively straightforward. If we have two items in the array with the same value, say $A[i_1] = A[i_2]$ where $i_1 < i_2$, then it is safe to ignore $A[i_2]$ because we need to report only one pair $(i, j)$.


Using a hash function randomly chosen from a universal hash family, each operation on the hash table takes $O(1)$ time in expectation. If the integers $x \in S$ are $r + 1$-digit numbers, with each digit belonging to $\mathbb{Z}_p = \{0, 1, \ldots, p - 1\}$ for some prime $p$, then in lecture, we showed the following hash family is universal:

$$\mathcal{H} = \{h_a : a \in \mathbb{Z}_p^{(r+1)}\}$$

$$h_a(k) = \sum_{i=0}^{r} a_i k_i \mod p.$$

If we choose $p$ to be a prime which is larger than $n$ but still $O(n)$, then our hash table requires only $O(n)$ space. Each operation on the hash table runs in $O(1+n/p) = O(1)$ time. Thus, the entire algorithm runs in $O(n)$ expected time.

(e) After a bit more work, you implement your algorithm. Your implementation is correct, but you discover that someone appears to have intentionally corrupted the data by deleting parts of the array. Conspiracy! Fortunately, you were able to recover a complete set of data from the backup system.

This sabotage is quite significant, and you wish to report to your superiors exactly what data was deleted. Assuming that you have a complete copy of the data stored in array $B$ of length $m \geq n$, give an efficient randomized algorithm to report all the
elements in \( B - A \). Your algorithm should have an \( O(m) \) expected runtime and use \( O(m) \) space. Neither \( A \) nor \( B \) is sorted.

**Solution:**

Use the following algorithm:

1. For \( 1 \leq i \leq n \), insert an element with key \( A[i] \) and value 1 into the hash table \( H \).
2. For \( 1 \leq j \leq m \), search for an element with key \( B[j] \) in the hash table \( H \). If it does not exist, report \( j \).

We choose a hash function \( h_a \) from a universal hash family as in part (d). The only difference from the previous part is we \( p = O(m) \) instead of \( p = O(n) \).

**Problem 3-3. 2-Universal Hashing**

Let \( \mathcal{H} \) be a class of hash functions in which each \( h \in \mathcal{H} \) maps the universe \( U \) of keys to \( \{0, 1, \ldots, m - 1\} \).

We say that \( \mathcal{H} \) is 2-universal if, for every fixed pair \( \langle x, y \rangle \) of keys where \( x \neq y \), and for any \( h \) chosen uniformly at random from \( \mathcal{H} \), the pair \( \langle h(x), h(y) \rangle \) is equally likely to be any of the \( m^2 \) pairs of elements from \( \{0, 1, \ldots, m - 1\} \). (The probability is taken only over the random choice of the hash function.)

(a) Show that if \( \mathcal{H} \) is 2-universal, then \( \mathcal{H} \) is universal.

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

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

There are exactly \( m \) possible ways for to have \( x \) and \( y \) collide, i.e., \( h(x) = h(y) = i \) for \( i \in \{0, 1, \ldots, m - 1\} \). Thus,

\[
\Pr_{h \in \mathcal{H}} [h(x) = h(y)] = \sum_{i=0}^{m-1} \left( \Pr_{h \in \mathcal{H}} [\langle h(x), h(y) \rangle = \langle i, i \rangle] \right) = \frac{m}{m^2} = \frac{1}{m}.
\]

Therefore, by definition, \( \mathcal{H} \) is universal.

(b) Suppose \( m \) is prime. Define \( Z_m = \{0, 1, \ldots, m - 1\} \), and let \( U = \mathbb{Z}_m^{(r+1)} \). Thus we can decompose each key \( k \in U \) into \( r + 1 \) digits, \( k = \langle k_0, k_1, \ldots, k_r \rangle \), with \( k_i \in \mathbb{Z}_m \) for each \( i \).

For each \( a = \langle a_0, a_1, \ldots, a_r \rangle \in \mathbb{Z}_m^{(r+1)} \), define the hash function \( h_a : U \rightarrow \mathbb{Z}_m \) as follows:

\[
h_a(k) = \sum_{i=0}^{r} a_i k_i \mod m.
\]
In lecture, we show that the family $\mathcal{H}$ of all such hash functions, given by

$$\mathcal{H} = \{h_a : a \in \mathbb{Z}_m^{(r+1)}\},$$

is universal. Is this family 2-universal? Justify your answer.

**Solution:** $\mathcal{H}$ is not a 2-universal hash family. Consider two keys $x = \langle 0, 0, \ldots, 0 \rangle$, and any $y \neq x$. For all possible choices of $a$, we always have $h_a(x) = 0$. Thus, $\langle h_a(x), h_a(y) \rangle$ is not equally likely to be any of the $m^2$ pairs of elements from $\mathbb{Z}_m$. Therefore $\mathcal{H}$ is not 2-universal.

(c) Again, let $m$ be prime, with $\mathbb{Z}_m$ defined as in part (b) above. For each $a = \langle a_0, a_1, \ldots, a_r \rangle \in \mathbb{Z}_m^{(r+1)}$ and $b \in \mathbb{Z}_m$, define the hash function $h'_{a,b} : U \to \mathbb{Z}_m$ as follows:

$$h'_{a,b}(k) = \left( \sum_{i=0}^{r} a_i k_i \right) + b \mod m.$$

Let $\mathcal{H}'$ be the family of all such hash functions, defined as

$$\mathcal{H}' = \{h'_{a,b} : a \in \mathbb{Z}_m^{(r+1)}, b \in \mathbb{Z}_m\}.$$

Show that the family $\mathcal{H}'$ is 2-universal.

**Solution:** We first note that, since there are $m$ choices for each of $a_0, a_1, \ldots, a_r, b$, we have $|\mathcal{H}'| = m^{r+2}$.

Let $x = \langle x_0, x_1, \ldots, x_r \rangle$ and $y = \langle y_0, y_1, \ldots, y_r \rangle$ be any pair of distinct keys. Then $x$ and $y$ differ in at least one position; without loss of generality, assume $x_0 \neq y_0$.

Let $\langle s, t \rangle \in \mathbb{Z}_m \times \mathbb{Z}_m$ be any pair of elements from $\mathbb{Z}_m$. We want to show that, if a hash function $h'_{a,b}$ is chosen uniformly at random from $\mathcal{H}'$, then the probability that $\langle h'_{a,b}(x), h'_{a,b}(y) \rangle = \langle s, t \rangle$ is $1/m^2$. In other words, we want to show that

$$\left| \{h'_{a,b} \in \mathcal{H}' : \langle h'_{a,b}(x), h'_{a,b}(y) \rangle = \langle s, t \rangle \} \right| = \frac{|\mathcal{H}'|}{m^2} = \frac{m^{r+2}}{m^2} = m^r.$$

Now, if $\langle h'_{a,b}(x), h'_{a,b}(y) \rangle = \langle s, t \rangle$, then we have

$$\left( \sum_{i=0}^{r} a_i x_i \right) + b \mod m = s \quad (1)$$

$$\left( \sum_{i=0}^{r} a_i y_i \right) + b \mod m = t. \quad (2)$$
This implies that
\[
\sum_{i=0}^{r} a_i (x_i - y_i) \equiv (s - t) \pmod{m},
\] (3)
or
\[
a_0 (x_0 - y_0) + \sum_{i=1}^{r} a_i (x_i - y_i) \equiv (s - t) \pmod{m},
\]
which implies that
\[
a_0 (x_0 - y_0) \equiv (s - t) - \sum_{i=1}^{r} a_i (x_i - y_i) \pmod{m}.
\]

Since \(x_0 \neq y_0\), an inverse \((x_0 - y_0)^{-1}\) must exist, which implies that
\[
a_0 \equiv \left[ (s - t) - \sum_{i=1}^{r} a_i (x_i - y_i) \right] \cdot (x_0 - y_0)^{-1} \pmod{m}.
\]

Consequently, for any choices of \(a_1, \ldots, a_r\), exactly one choice of \(a_0\) causes Eqn. (3) to be satisfied. Furthermore, for any choices of \(a_0, a_1, \ldots, a_r\), exactly one choice of \(b\) causes Eq. (1) to be satisfied, namely,
\[
b \equiv s - \sum_{i=0}^{r} a_i x_i \pmod{m}.
\]

Therefore, it follows that for any choices of \(a_1, \ldots, a_r\), there is exactly one choice of \(a_0\) and one choice of \(b\) that together cause Eqs. (1) and (3) to be satisfied. Since Eqs. (1) and (2) are satisfied if and only if Eqs. (1) and (3) are satisfied, we have that there are \(m\) choices for each of \(a_1, \ldots, a_r\), but once these are chosen, exactly one choice of \(a_0\) and one choice of \(b\) cause \(h'_a(b)(x)\) to be equal to \(s\) and \(h'_a(b)(y)\) to be equal to \(t\). This implies that the number of hash functions \(h'_a(b)\) in \(H'\) that cause \(h'_a(b)(x)\) to be equal to \(s\) and \(h'_a(b)(y)\) to be equal to \(t\) is \(m^r\), as desired.